# RxMicro User Guide

## Table of Contents


1. Introduction . . . 9
   1.1. RxMicro Features . . . 10
   1.2. RxMicro Benefits . . . 13
   1.3. Requirements . . . 14
2. What are Microservices? . . . 15
3. Quick Start . . . 18
   3.1. Creating a Project . . . 18
      3.1.1. Using the `IntelliJ IDEA` . . . 18
      3.1.2. Using the Terminal . . . 21
      3.1.3. Using Other IDE . . . 21
   3.2. Configuring the Project . . . 23
      3.2.1. Definition the Versions of the Used Libraries . . . 23
      3.2.2. Adding the Required Dependencies . . . 23
      3.2.3. Configuring the `maven-compiler-plugin` . . . 23
      3.2.4. The Final Version of `pom.xml` File . . . 24
   3.3. Creating the Source Code . . . 27
      3.3.1. A `module-info.java` Descriptor . . . 27
      3.3.2. An HTTP Response Model Class . . . 27
      3.3.3. A REST-Based Microservice Class . . . 28
      3.3.4. A Structure of the Microservice Project . . . 29
   3.4. Compiling the Project . . . 30
      3.4.1. Using the `maven` . . . 30
      3.4.2. Using the `IntelliJ IDEA` . . . 31
   3.5. Starting the Microservice . . . 33
      3.5.1. Using the IDE: . . . 33
      3.5.2. Using the Terminal: . . . 34
   3.6. Verifying the Microservice . . . 40
   3.7. Automated Test . . . 41
      3.7.1. Configuring the Project . . . 41
      3.7.2. Creating a Test Class . . . 43
      3.7.3. Starting the Test Class . . . 45
   3.8. The Project at the Github . . . 48
4. Core Concepts . . . 49
   4.1. How It Works? . . . 49
      4.1.1. A Common Work Schema . . . 50
      4.1.2. Generating of Additional Classes. . . . 51

4.1.3. Using the Debug Mode . . . 61
4.2. `RxMicro Annotation Processor` Options . . . 63
4.3. Don't Block Current Thread! . . . 65
4.3.1. Prohibited Operations . . . 65
4.3.2. Recommended Approach . . . 65
4.4. Reactive Libraries Support . . . 66
4.4.1. Expected Business Process Results . . . 66
4.4.2. Recommendations for Choosing a Library . . . 67
4.5. Base Model . . . 69
4.6. Strings Formatting . . . 70
4.7. Encapsulation . . . 72
4.7.1. A `private` Modifier Usage . . . 72
4.7.2. A Separate Package Usage . . . 73
4.7.3. A `getters` Usage . . . 73
4.7.4. Performance Comparison . . . 74
4.7.5. Approach Selection Recommendations . . . 77
4.8. Configuration . . . 78
4.8.1. Basic Structure of the Config Module . . . 79
4.8.2. Configuration Types . . . 81
4.8.3. Config Sources Setting . . . 92
4.8.4. Dynamic Configuration . . . 96
4.8.5. User Defined Configurations . . . 97
4.8.6. Supported Parameter Types . . . 99
4.8.7. Configuration Verifiers . . . 106
4.9. Logger . . . 107
4.9.1. Logger Configuration . . . 108
4.9.2. Logger Handler . . . 111
4.9.3. Pattern Formatter . . . 112
4.9.4. Custom Log Event . . . 116
4.9.5. Multiline Logs Issue For Docker Environment . . . 118
4.10. JSON . . . 119
4.10.1. A Mapping Between JSON and Java Types . . . 120
4.10.2. `rxmicro.json` Module Usage . . . 121
4.10.3. Json Wrappers . . . 126
4.11. Native Transports . . . 128
5. REST Controller . . . 130
5.1. REST Controller Implementation Requirements. . . . 131
5.1.1. REST Controller Class Requirements. . . . 131
5.1.2. HTTP Request Handler Requirements. . . . 132
5.2. `RxMicro Annotations` . . . 134
5.3. Return Types . . . 136

5.3.1. Supported Return Types for HTTP Response without Body . . . 137
5.3.2. Supported Return Types for HTTP Response with Body . . . 141
5.4. Routing of Requests . . . 146
5.4.1. Routing of Requests Based on HTTP Method . . . 147
5.4.2. Routing of Requests Based on URL Path . . . 149
5.4.3. Routing of Requests Based on HTTP Body . . . 151
5.5. `Not Found` Logic . . . 153
5.6. Exceptions Usage . . . 156
5.6.1. Basic Class of Exceptions . . . 157
5.6.2. Using the User Defined Exceptions . . . 158
5.6.3. Error Signal Generation Methods . . . 160
5.6.4. Predefined Classes of Exceptions . . . 160
5.7. Overriding the Status Code . . . 161
5.8. Redirecting of Requests . . . 163
5.9. HTTP Headers Support . . . 166
5.9.1. Basic Rules . . . 166
5.9.2. Supported Data Types . . . 170
5.9.3. Static HTTP headers . . . 172
5.9.4. Repeating HTTP headers . . . 173
5.10. HTTP Parameters Handling . . . 175
5.10.1. Basic Rules . . . 175
5.10.2. Supported Data Types . . . 180
5.10.3. Complex Model Support . . . 184
5.11. The `path variables` Support . . . 188
5.11.1. Basic Rules . . . 188
5.11.2. Supported Data Types . . . 191
5.12. Support of Internal Data Types . . . 192
5.12.1. Basic Rules . . . 192
5.12.2. Supported Internal Data Types . . . 197
5.13. Versioning of REST Controllers . . . 198
5.13.1. Versioning Based on HTTP Header Analysis . . . 199
5.13.2. Versioning Based on URL Path Fragment Analysis . . . 201
5.14. Base URL Path for All Handlers . . . 203
5.15. `CORS` Support . . . 204
5.16. Request ID . . . 207
5.17. Configuring the Code Generation Process . . . 208
6. REST Client . . . 210
6.1. REST Client Implementation Requirements . . . 211
6.1.1. REST Client Interface Requirements . . . 211
6.1.2. HTTP Request Handler Requirements . . . 212
6.2. `RxMicro Annotations` . . . 214

6.3. HTTP Request Handler Return Types . . . 217
6.3.1. Supported Return Result Types for HTTP Response without Body . . . 218
6.3.2. Supported Return Result Types for HTTP Response with Body . . . 223
6.4. HTTP Headers Handling . . . 227
6.4.1. Basic Rules . . . 227
6.4.2. Supported Data Types . . . 232
6.4.3. Static HTTP Headers . . . 233
6.4.4. Repeating HTTP Headers . . . 235
6.5. HTTP Parameters Handling . . . 238
6.5.1. Basic Rules . . . 238
6.5.2. Supported Data Types . . . 243
6.5.3. Complex Model Support . . . 244
6.5.4. Static Query Parameters . . . 250
6.5.5. Repeating Query Parameters . . . 252
6.6. The `path variables` Support . . . 255
6.6.1. Basic Rules . . . 255
6.6.2. Supported Data Types . . . 258
6.7. Support of Internal Data Types . . . 259
6.7.1. Basic Rules . . . 259
6.7.2. Supported Internal Data Types . . . 263
6.8. Versioning of REST Clients . . . 264
6.8.1. Versioning Based on HTTP Header Analysis . . . 265
6.8.2. Versioning Based on URL Path Fragment Analysis . . . 268
6.9. Base URL Path for All Handlers . . . 271
6.10. Expressions . . . 273
6.11. Request ID . . . 278
6.12. Partial Implementation . . . 279
6.13. Configuring the Code Generation Process . . . 282
7. Validation . . . 284
7.1. Preparatory Steps . . . 285
7.1.1. Adding the Required Dependencies: . . . 285
7.1.2. Adding the `rxmicro.validation` Module to `module-info.java` . . . 285
7.2. Built-in Constraints. . . . 286
7.3. HTTP Requests Server Validation . . . 291
7.4. HTTP Responses Server Validation . . . 294
7.5. HTTP Responses Client Validation . . . 297
7.6. HTTP Requests Client Validation . . . 300
7.7. Creating Custom Constraints . . . 301
7.8. Disabling Validation . . . 304
7.8.1. Removing the `rxmicro.validation` Module . . . 304
7.8.2. Using `GenerateOption.DISABLED` Option . . . 304

7.8.3. Using `@DisableValidation` Annotation . . . 305
8. REST-based Microservice Documentation . . . 306
8.1. Basic Usage . . . 307
8.1.1. Min Settings . . . 307
8.1.2. `Asciidoctor-maven-plugin` Settings . . . 308
8.1.3. `Asciidoctor-dependency-plugin` Settings . . . 309
8.2. `RxMicro Annotations` . . . 312
8.3. `@Example` and `@Description` Usage . . . 314
8.4. Sections Customization . . . 315
8.5. Integration with `rxmicro.validation` Module . . . 318
8.6. REST-based Microservice Metadata Configuration . . . 320
8.6.1. Using `pom.xml` . . . 320
8.6.2. Using Annotations . . . 322
8.7. Error Documentation . . . 324
9. Postgre SQL Data Repositories . . . 326
9.1. Basic Usage . . . 327
9.2. `RxMicro Annotations` . . . 330
9.3. Repositories Testing . . . 332
9.3.1. Test Database . . . 332
9.3.2. Test Templates . . . 335
9.4. DataBase Models . . . 339
9.4.1. Primitives . . . 339
9.4.2. Entities . . . 340
9.5. Universal Placeholder . . . 341
9.6. `@RepeatParameter` Annotation . . . 342
9.7. SQL Operations . . . 343
9.7.1. `@Select` . . . 344
9.7.2. `@Insert` . . . 360
9.7.3. `@Update` . . . 362
9.7.4. `@Delete` . . . 364
9.8. Variables Support . . . 366
9.9. Primary Keys Support . . . 369
9.10. `@ExpectedUpdatedRowsCount` annotation . . . 371
9.11. Transactions Support . . . 373
9.11.1. DataBase Transactions . . . 373
9.11.2. Concurrent Access Example . . . 376
9.12. Partial Implementation . . . 379
9.13. Logging . . . 381
10. Mongo Data Repositories . . . 382
10.1. Basic Usage . . . 383
10.2. `RxMicro Annotations` . . . 386

10.3. Repositories Testing . . . 388
10.3.1. Test Database . . . 388
10.3.2. Test Templates . . . 391
10.4. Universal Placeholder . . . 395
10.5. `@RepeatParameter` Annotation . . . 396
10.6. Mongo Operations . . . 397
10.6.1. `@Find` . . . 398
10.6.2. `@Aggregate` . . . 399
10.6.3. `@Distinct` . . . 400
10.6.4. `@CountDocuments` . . . 401
10.6.5. `@EstimatedDocumentCount` . . . 402
10.6.6. `@Insert` . . . 403
10.6.7. `@Update` . . . 404
10.6.8. `@Delete` . . . 405
10.7. Partial Implementation . . . 406
10.8. Logging . . . 408
11. Contexts and Dependency Injection . . . 409
11.1. Basic Usage . . . 410
11.2. `RxMicro Annotations` . . . 413
11.3. All Beans are Singletons! . . . 414
11.4. Constructor Injection . . . 416
11.5. Method Injection . . . 418
11.6. Ambiguity Resolving . . . 419
11.6.1. Default Ambiguity Resolving . . . 419
11.6.2. The `@Named` (`@Qualifier`) Annotation Usage . . . 423
11.6.3. Custom Annotations Usage . . . 427
11.7. Dependency Injection Using Spring Style . . . 430
11.8. `@PostConstruct` . . . 432
11.9. RxMicro Components Injection . . . 435
11.9.1. Basic Usage . . . 435
11.9.2. All Supported RxMicro Components . . . 436
11.10. Factory Method . . . 437
11.11. Factory Class . . . 439
11.12. External resource injection . . . 441
11.12.1. Basic Usage . . . 441
11.12.2. Additional Info . . . 442
11.13. Optional Injection . . . 443
11.14. Multibindings . . . 445
12. Monitoring . . . 447
12.1. Request Id . . . 448
12.1.1. Basic Rules . . . 448

12.1.2. Supported Generator Providers . . . 451
12.2. Request Tracing Usage Example . . . 453
13. Java EcoSystem Integration . . . 458
13.1. Unnamed Modules Support . . . 458
13.1.1. Uber Jar . . . 460
13.1.2. Module Configuration . . . 465
13.2. Using GraalVM to Build a Native Image . . . 470
13.2.1. Setup a GraalVM . . . 470
13.2.2. RxMicro Project Configuration . . . 472
13.2.3. Creation of the Native Image . . . 476
13.2.4. Verification of the Native Image . . . 476
14. Testing . . . 477
14.1. Preparatory Steps . . . 478
14.1.1. Definition the Versions of the Used Libraries: . . . 478
14.1.2. Adding the Required Dependencies: . . . 478
14.1.3. Configuring the `maven-compiler-plugin`: . . . 479
14.1.4. Configuring the `maven-surefire-plugin`: . . . 481
14.2. `RxMicro Test Annotations` . . . 483
14.3. Alternatives . . . 486
14.3.1. Injection Algorithm for the Alternative of the RxMicro Component . . . 487
14.3.2. Injection Algorithm for the Alternative of the Custom Component . . . 488
14.3.3. Alternative Usage . . . 489
14.3.4. Components with Custom Constructors . . . 493
14.3.5. Ambiguity Resolving . . . 496
14.3.6. CDI Beans Alternatives . . . 501
14.4. How It Works . . . 505
14.5. The `@BeforeThisTest` and `@BeforeIterationMethodSource` Annotations . . . 509
14.6. REST-based Microservice Testing . . . 514
14.6.1. Basic Principles . . . 515
14.6.2. Features of Testing Complex Microservices that use Alternatives . . . 516
14.6.3. Custom and the RxMicro Framework Code Execution Order . . . 527
14.7. Testing of Microservice Components . . . 531
14.7.1. Basic Principles . . . 531
14.7.2. Features of Testing Complex Components that Use Alternatives . . . 531
14.7.3. Custom and the RxMicro Framework Execution Order . . . 532
14.8. REST-based Microservice Integration Testing . . . 535
14.8.1. Basic Principles . . . 536
14.8.2. The `BlockingHttpClient` Settings . . . 539
14.8.3. The `docker` Usage . . . 540
14.9. Database Testing Using DBUnit . . . 542
14.9.1. Test Database Configuration . . . 543

14.9.2. Retrieve Connection Strategies . . . 544
14.9.3. `@InitialDataSet` Annotation . . . 548
14.9.4. `@ExpectedDataSet` Annotation . . . 550
14.9.5. `@RollbackChanges` Annotation . . . 552
14.9.6. Ordered Comparison . . . 554
14.9.7. Supported Expressions . . . 556
15. Appendices . . . 565
15.1. Appendix A: FAQ . . . 565
15.1.1. Does the RxMicro framework modify my byte code? . . . 565
15.1.2. Can the RxMicro framework be used for purposes other than microservices? . . . 565
15.1.3. Why I receive `class not found` error? . . . 565
15.1.4. Why I receive `The Kotlin standard library is not found in the module graph` error? 565
15.1.5. Why I receive `java.lang.NullPointerException: autoRelease couldn't be null` error during unit testing? . . . 565
15.1.6. Why I receive `java.lang.reflect.InaccessibleObjectException: Unable to make MicroServiceTest() accessible: module module.name does not "opens test.package" to unnamed module` error during unit testing? . . . 566
15.2. Appendix B: Useful Links . . . 567

© 2019-2022 rxmicro.io. Free use of this software is granted under the terms of the `Apache License 2.0`.

Copies of this entity may be made for Your own use and for distribution to others, provided that You do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

If You find errors or omissions in this entity, please don't hesitate to submit an **issue** or open **a pull request** with a fix.

# 1. Introduction

**RxMicro** is a modern, JVM-based, full stack framework designed to develop distributed reactive applications that use a microservice architecture.

The RxMicro framework provides developers with a convenient tool to focus on writing an application business logic. Meanwhile, routine and standard operations, which are the prerequisite for launching an application, are delegated to the framework.

The RxMicro framework is small and lightweight. Even though the RxMicro framework is designed to create microservices, a developer can easily use separate **RxMicro** modules to develop any type of application using a reactive approach.

The RxMicro framework is a framework that uses reactive programming as the main and **only** approach when designing microservices.

**Any blocking operations are not supported!**

When developing a project using the RxMicro framework, use only **non-blocking drivers** to interact with databases, network connections and files. Otherwise Your project will work **too slow**, and won't be able to process a large number of clients' requests.

## 1.1. RxMicro Features

The RxMicro framework provides the following feature set:

1. Declarative programming using annotations.
2. Reactive programming using common libraries:
    a. **java.util.concurrent**: CompletableFuture, CompletionStage.
    b. **Project Reactor**: Mono, Flux.
    c. **RxJava**: Completable,Single, Maybe, Flowable.
3. Configuring using java configuration, annotations, files, system properties and environment variables.
4. Declarative handlers of HTTP requests to the microservice.
    a. Request routing based on HTTP method, URL path and HTTP body analysis.
    b. HTTP header processing.
    c. HTTP parameter processing.
    d. Path variable support.
    e. Automatic conversion of request data into Java model and vice versa.
    f. Built-in validation of requests and responses.
    g. Static configuration support.
    h. Handler versioning support.
    i. `Cross-Origin Resource Sharing (CORS)` support.
    j. Support for request identification during inter-service interaction.
5. Declarative REST client.
    a. HTTP header processing.
    b. HTTP parameter processing.
    c. Path variable support.
    d. Automatic conversion of request data into Java model and vice versa.
    e. Built-in validation of requests and responses.
    f. Static configuration support.
    g. REST client versioning support.
    h. Request timeout.
    i. Automatic redirection support.
    j. Customization option for standard client implementation.
6. Contexts and Dependency Injection (CDI).
    a. Dependencies can be explicitly managed without using CDI.
    b. Dependency injection by fields, methods and constructors.

c. Qualifier support.
d. Factory method support.
e. Post construct method support.
f. Class factory support.
g. Optional injection.
h. Resource injection.
i. Multibinder support.
j. Dependency injection using JEE and Spring style.

7. Generation of REST-based microservice documentation.
    a. Documenting with annotations.
    b. asciidoc support (widely used and multifunctional documenting format).
    c. Configuration of the project documentation standard sections.
8. Data Repositories.
    a. Postgre SQL Data Repositories.
        i. `SELECT`, `INSERT`, `UPDATE`, `DELETE` operation support.
        ii. Auto-generated primary key support.
        iii. Composite primary key support.
        iv. Transaction support.
        v. Variable support in SQL query.
        vi. Customized `SELECT` queries support.
        vii. Possibility to customize a standard repository implementation.
        viii. Access to a low-level API.
        ix. Auto registration of enum codecs.
    b. Mongo Data Repositories.
        i. `find`, `aggregate`, `distinct`, `insert`, `update`, `delete`, `countDocuments` and `estimatedDocumentCount` operation support.
        ii. Auto-generated entity id support.
        iii. Query parameter logging.
        iv. Possibility to customize a standard repository implementation.
        v. Access to a low-level API.
9. Monitoring
    a. Health checks.
    b. Request tracing.
10. Testing.
    a. Component testing.

    b. REST-based microservices testing.
    c. Integration testing.
    d. Support for alternatives and mocks.
    e. Integration with JUnit 5 and Mockito frameworks.
    f. Integration with DBUnit framework.
11. Monitoring.
    a. Health checks.
    b. Request tracing.
12. Integration with other Java libraries and frameworks.
    a. A GraalVM native image support.

## 1.2. RxMicro Benefits

The RxMicro framework provides the following benefits:

- Declarative programming using annotations.
- CDI by demand.
- Human readable generated code.
- Verifier of the redundant and inefficient source code.
- Runtime without `reflection`.
- Fast startup time.
- Reduced memory footprint.

These benefits are gained due to:

- using of **Java annotation processors**, which generates standard code based on `RxMicro Annotations`;
- replacing standard Java libraries that require `reflection` for their work with analogs that do not need `reflection`;
- using of **Netty** as the primary NIO framework for **non-blocking asynchronous IO operations**;
- generation of low-level code avoiding unnecessary abstractions and proxies.

## 1.3. Requirements

The RxMicro framework requires JDK 11 LTS or higher.

To succeed in studying this guide, it is assumed that the reader is familiar with the following technologies:

- `JDK 11 base API`;
- `Apache Maven`;
- `JUnit 5`;
- `DBUnit`;
- `Mockito`;
- `Hypertext Transfer Protocol (HTTP)`;
- `Representational state transfer (REST)`;
- `Docker`.

The RxMicro framework uses the following Java modules:

- Common module(s):
  - `java.base`;
- The `rxmicro.logger` module requires the following module(s):
  - `java.logging`;
- REST client and REST based microservice test modules require the following module(s):
  - `java.net.http`;
- The `rxmicro.data.r2dbc.postgresql` module requires the following module(s):
  - `java.naming`;
  - `java.management`;
- Netty requires the the following module(s):
  - `jdk.unsupported`.

# 2. What are Microservices?

Microservices - also known as the microservice architecture - is an architectural style that structures an application as a collection of services that are:

- Highly maintainable and testable.
- Loosely coupled.
- Independently deployable.
- Organized around business capabilities.
- Owned by a small team.

*(Read more at* `https://microservices.io/`*...)*

Thus, a microservice project consists of several microservices. Each microservice must fulfill only one business task.

Let's look at a microservice that displays the current date and time in UTC format:

```
public final class MicroService1 {

    public static void main(final String[] args) {
        System.out.println(Instant.now());
    }
}
```

Does this microprogram constitute a microservice?
Yes, since this microprogram fulfills a business task.
Unfortunately, this program has a serious disadvantage: it interacts with clients through the console. Therefore, **only a client's program with a console interface launched in a session of the current logged-in OS user will be able to interact with this microservice!** This restriction makes it impossible to scale this microservice!

Can we improve this situation? Yes, we can:

```
public final class MicroService2 {

    public static void main(final String[] args) throws Exception {
        Files.write(
                Paths.get("/var/microservice/now-instant.txt"),
                Instant.now().toString().getBytes(UTF_8)
        );
    }
}
```

This microservice uses a file system to interact with client's programs. In this way, the only requirement for the client's program is to be run on the same computer on which the microservice

is running. The situation has improved, but it is still impossible to scale this microservice horizontally!

Can we improve this situation? Yes, we can:

```
public final class MicroService3 {

    public static void main(final String[] args) throws Exception {
        try (final ServerSocket serverSocket = new ServerSocket(8080)) {
            try (final Socket clientSocket = serverSocket.accept()) {
                try (final OutputStream out = clientSocket.getOutputStream()) {
                    // read command from input stream
                    out.write(Instant.now().toString().getBytes(UTF_8));
                }
            }
        }
    }
}
```

Now, the microservice uses the network to interact with clients. This implementation of the microservice is scalable as the microservice can now be run on several networked computers. The situation has improved markedly, but there are problems with networking:

- Presence of firewalls.
- The need to create an interaction protocol.
- Independence from the programming language is an important criterion for the interaction protocol.

Can we improve this situation? Yes, for this purpose You can use the HTTP protocol with the REST architecture style:

```
public final class MicroService4 {

    public static void main(final String[] args) throws IOException {
        final HttpServer server = HttpServer.create(new InetSocketAddress("localhost",
8080), 0);
        server.createContext("/now-instant", exchange -> {
            final String content = Instant.now().toString();
            exchange.sendResponseHeaders(200, content.length());
            exchange.getResponseHeaders().add("Content-Type", "text/txt");
            try (final OutputStream body = exchange.getResponseBody()) {
                body.write(content.getBytes(UTF_8));
            }
        });
        server.start();
    }
}
```

**That's why microservices are often referred to as REST-based microservices**

For simple tasks, the entire logic of the microservice can be found in one class, which is often called **microservice**. If a microservice has to solve a complex task, then this microservice is divided into two logical components:

1. REST controller, the main task of which is:
    a. to accept HTTP requests;
    b. to validate HTTP requests;
    c. to convert HTTP requests into Java models;
    d. to invoke request handlers;
    e. once the response model is received, convert it to an HTTP response.
2. Business service, the main task of which is:
    a. if the task is of medium complexity, then independently calculate the result and return it to the REST controller;
    b. if it is a high-complexity task, then decompose it into sub-tasks and delegate its execution to other microservices. After all sub-tasks have been completed, merge the result and return it to the REST controller.

Therefore, the following is implied in this guide:

1. If You find the term **microservice**, it means **REST-based microservice**, unless stated otherwise!
2. If You find the term **REST controller**, it means a logical component of the microservice that performs its direct functions!

# 3. Quick Start

This section describes in detail the steps to be taken in order to create the REST-based microservice that returns the `"Hello World!"` message, using the RxMicro framework.

In order to successfully execute these instructions, You need to install JDK 11 LTS or higher on Your computer. For Your convenience it is also recommended to use a modern IDE, for example IntelliJ IDEA.

The features of the **IntelliJ IDEA Community Edition** version are enough for a complete and convenient work on a project that uses the RxMicro framework.

The RxMicro framework consists of several dozens of modules, so for convenient handling it is recommended to install `maven` on Your computer.

Any modern IDE for Java (`IntelliJ IDEA, Eclipse, NetBeans`) already contains a built-in `maven`, so there is no need in its additional installation on Your computer.

To run `maven` commands, You can use Your IDE instead of the terminal.

## 3.1. Creating a Project

For creating a project, it is recommended to use a modern IDE, for example IntelliJ IDEA

### 3.1.1. Using the `IntelliJ IDEA`

To create a new project, proceed as follows: `File` → `New` → `Project` or `Create a New Project`.

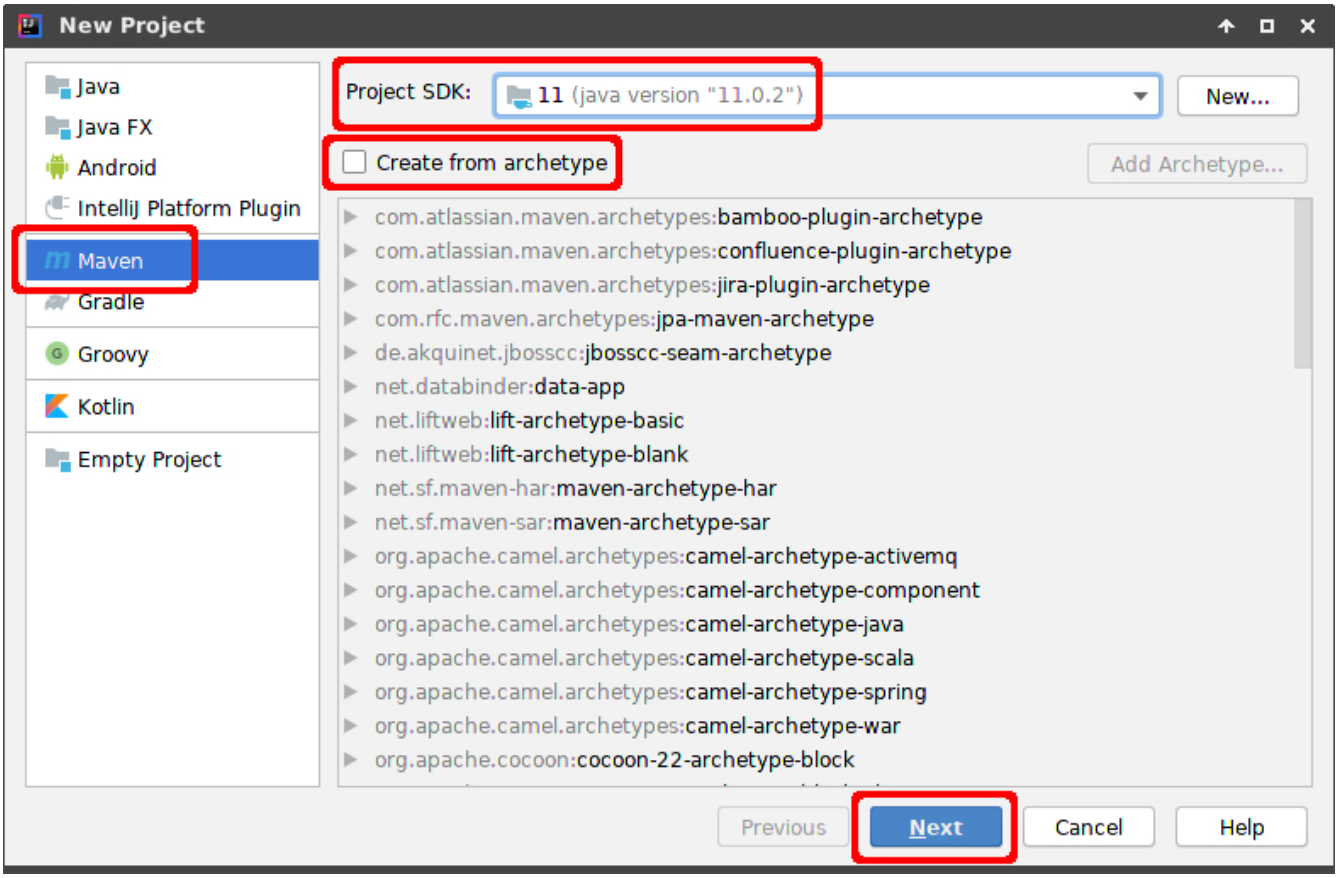



*Figure 1. Creating the simplest project in IntelliJ IDEA: Choosing a project type.*

In the appeared dialog box select the `Maven` type, make sure that `Project SDK` version 11 or higher will be used, remove the `Create from archetype` checkbox and click `Next`.

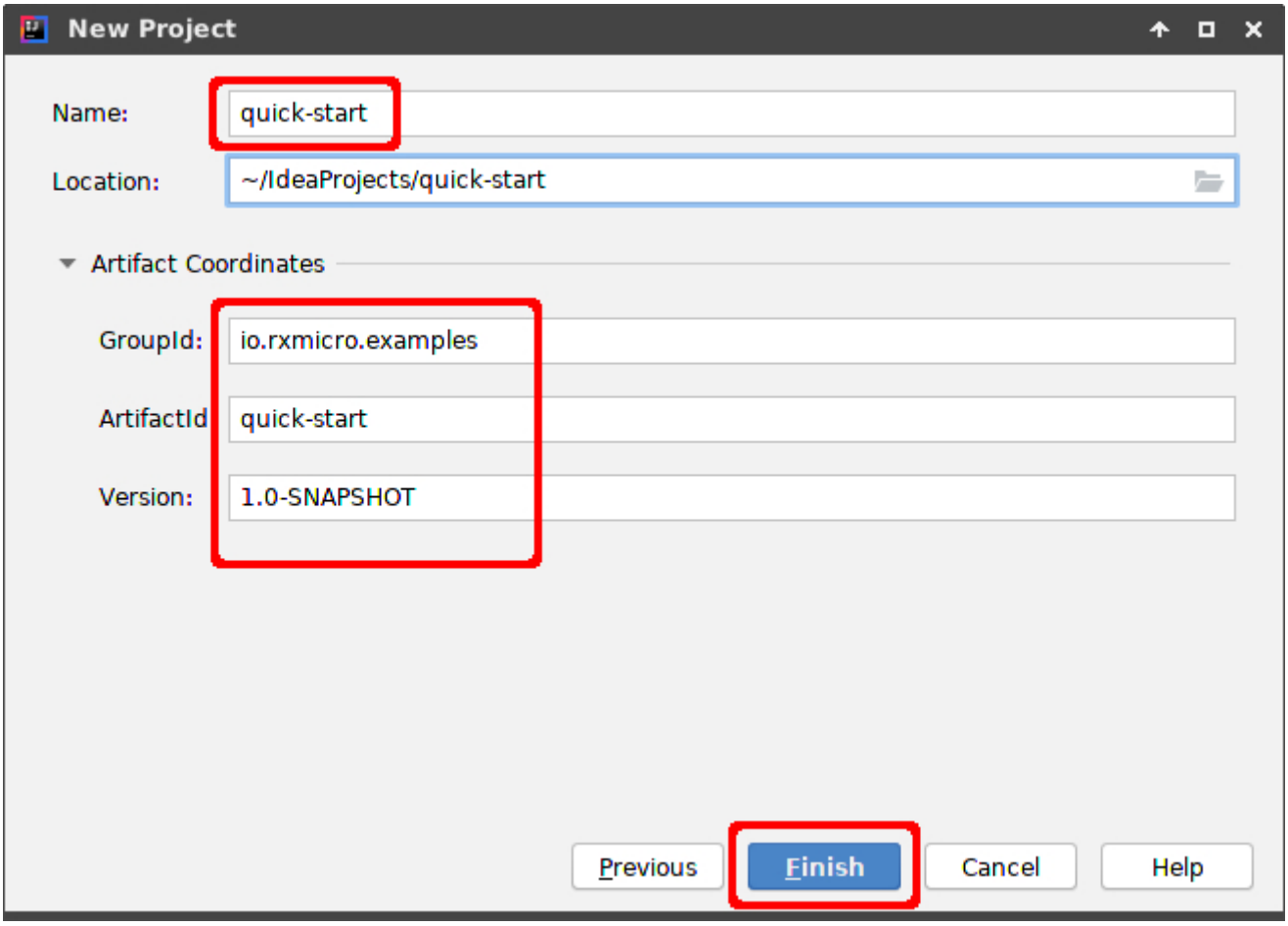



*Figure 2. Creating the simplest project in IntelliJ IDEA: Basic settings.*

In the appeared dialog box type `Name`, `Location` (if the default value is wrong) and `GroupId` (if the default value is wrong), and click `Finish`.

As a result, IntelliJ IDEA will generate the following project template using maven settings:

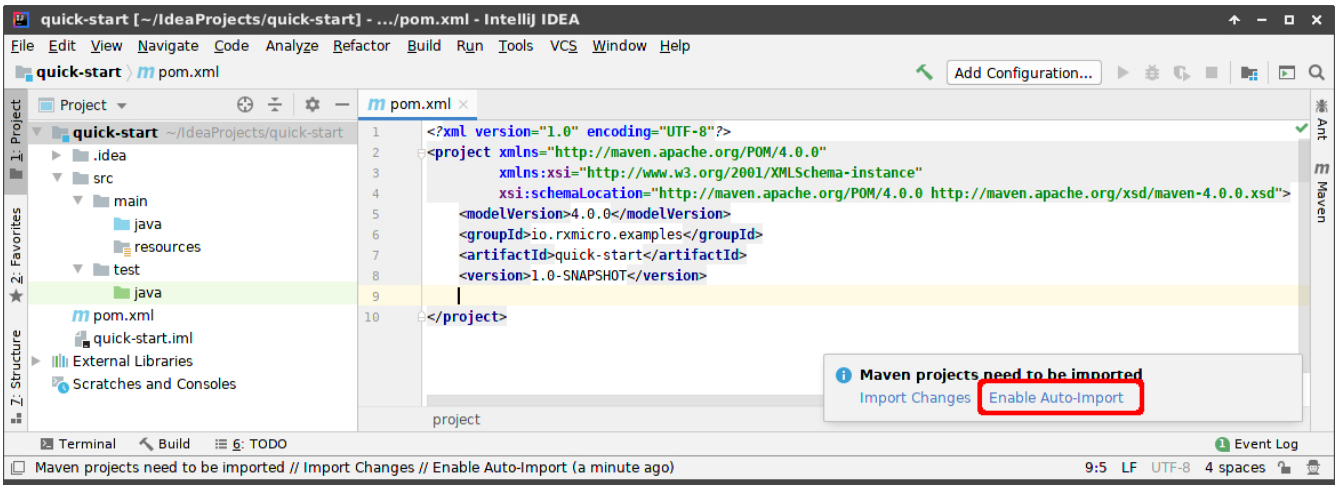



*Figure 3. Creating the simplest project in IntelliJ IDEA: Basic project template.*

After creating the standard template, activate the `Enable Auto-Import` option.

If for some reason the IntelliJ IDEA or another IDE You use for Java coding (e.g. Eclipse or NetBeans) has generated another project template, delete unnecessary files, create un-created folders and leave only the following sections in pom.xml: modelVersion, groupId, artifactId, version.

As a result, Your template should fully match the template: Figure 3, “Creating the simplest project in IntelliJ IDEA: Basic project template.”.

### 3.1.2. Using the Terminal

It is possible to create a new maven project without using the IDE.

If You don’t intend to write the source code of a project in notepad, but rather use the IDE to do this, You should directly create a maven project using the IDE.

To do this, open the terminal and run the following command:

```
mvn archetype:generate -DgroupId=io.rxmicro.examples -DartifactId=quick-start
-DinteractiveMode=false
```

In order to run maven commands in the terminal it is necessary to:

- Install JDK and maven on Your computer;
- Set the JAVA_HOME environment variable;
- Add the path to $MAVEN_HOME/bin folder to the $PATH environment variable.

*A detailed instruction on the maven installation for its further use in the terminal can be found at the following link:* Installing Apache Maven.

As a result, the quick-start folder with the basic project template will be created in the current folder. After that, the created project must be imported into the IDE.

By default, mvn archetype:generate doesn’t generate an empty project, but a project with two App and AppTest classes, as well as a connected junit library of 3.8.1 version in pom.xml. The specified classes and dependencies should be deleted before performing the following steps of this guide.

As a result, Your template should fully match the template: Figure 3, “Creating the simplest project in IntelliJ IDEA: Basic project template.”.

### 3.1.3. Using Other IDE

Creating the simplest project with other IDEs does not differ much from creating it with IntelliJ IDEA. When creating, You should also specify maven archetype, groupId, artifactId and version.

The main thing is that after creation **Your project template should fully match the template:**

**Figure 3, “Creating the simplest project in IntelliJ IDEA: Basic project template.”**.

## 3.2. Configuring the Project

Before writing the code of a REST-based microservice, You should configure `pom.xml` of Your project by performing the following steps:

1. Define the versions of used libraries.
2. Add the required dependencies to the `pom.xml`.
3. Configure the `maven-compiler-plugin`.

### 3.2.1. Definition the Versions of the Used Libraries

To make further updating of library versions convenient, it is recommended to use `maven properties`:

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <rxmicro.version>0.11</rxmicro.version> ①
    <maven-compiler-plugin.version>3.10.1</maven-compiler-plugin.version> ②
</properties>
```

① The latest stable version of the RxMicro framework.

② The latest stable version of the `maven-compiler-plugin`.

### 3.2.2. Adding the Required Dependencies

Before using RxMicro modules, the following dependencies must be added to the project:

```
<dependencies>
    <dependency>
        <groupId>io.rxmicro</groupId>
        <artifactId>rxmicro-rest-server-netty</artifactId> ①
        <version>${rxmicro.version}</version>
    </dependency>
    <dependency>
        <groupId>io.rxmicro</groupId>
        <artifactId>rxmicro-rest-server-exchange-json</artifactId> ②
        <version>${rxmicro.version}</version>
    </dependency>
</dependencies>
```

① Library for building REST-based microservices based on HTTP server that uses Netty;

② Library for converting Java model to JSON format and vice versa;

### 3.2.3. Configuring the `maven-compiler-plugin`

Since the RxMicro framework uses the Java annotation processors, You need to set up `maven-`

`compiler-plugin`:

```
<plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>${maven-compiler-plugin.version}</version>
    <configuration>
        <release>11</release> ①
        <annotationProcessorPaths>
            <annotationProcessorPath>
                <groupId>io.rxmicro</groupId>
                <artifactId>rxmicro-annotation-processor</artifactId> ②
                <version>${rxmicro.version}</version>
            </annotationProcessorPath>
        </annotationProcessorPaths>
    </configuration>
    <executions>
        <execution>
            <id>source-compile</id>
            <goals>
                <goal>compile</goal>
            </goals>
            <configuration>
                <annotationProcessors>
                    <annotationProcessor>
                        io.rxmicro.annotation.processor.RxMicroAnnotationProcessor ③
                    </annotationProcessor>
                </annotationProcessors>
                <generatedSourcesDirectory>
                    ${project.build.directory}/generated-sources/ ④
                </generatedSourcesDirectory>
            </configuration>
        </execution>
    </executions>
</plugin>
```

① The RxMicro framework requires a Java compiler of v11 or higher;

② The annotation processor library, that will handle all `RxMicro Annotations`;

③ The annotation processor class, that handles the launch configuration;

④ Location of the generated Java classes by the `RxMicro Annotation Processor`;

### 3.2.4. The Final Version of `pom.xml` File

After all the above changes, the final version of the `pom.xml` file should look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
```

```
                             http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>io.rxmicro.examples</groupId>
    <artifactId>quick-start</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <rxmicro.version>0.11</rxmicro.version>
        <maven-compiler-plugin.version>3.10.1</maven-compiler-plugin.version>
    </properties>

    <build>
        <pluginManagement>
            <plugins>
                <plugin>
                    <artifactId>maven-compiler-plugin</artifactId>
                    <version>${maven-compiler-plugin.version}</version>
                    <configuration>
                        <release>11</release>
                        <annotationProcessorPaths>
                            <annotationProcessorPath>
                                <groupId>io.rxmicro</groupId>
                                <artifactId>rxmicro-annotation-processor</artifactId>
                                <version>${rxmicro.version}</version>
                            </annotationProcessorPath>
                        </annotationProcessorPaths>
                    </configuration>
                    <executions>
                        <execution>
                            <id>source-compile</id>
                            <goals>
                                <goal>compile</goal>
                            </goals>
                            <configuration>
                                <annotationProcessors>
                                    <annotationProcessor>
                            io.rxmicro.annotation.processor.RxMicroAnnotationProcessor
                                    </annotationProcessor>
                                </annotationProcessors>
                                <generatedSourcesDirectory>
                                    ${project.build.directory}/generated-sources/
                                </generatedSourcesDirectory>
                            </configuration>
                        </execution>
                    </executions>
                </plugin>
            </plugins>
        </pluginManagement>
    </build>
```

```
    <dependencies>
        <dependency>
            <groupId>io.rxmicro</groupId>
            <artifactId>rxmicro-rest-server-netty</artifactId>
            <version>${rxmicro.version}</version>
        </dependency>
        <dependency>
            <groupId>io.rxmicro</groupId>
            <artifactId>rxmicro-rest-server-exchange-json</artifactId>
            <version>${rxmicro.version}</version>
        </dependency>
    </dependencies>
</project>
```

## 3.3. Creating the Source Code

The source code of the simplest REST-based microservice consists of one module, one package and two classes. The source code of each of these components is described below.

### 3.3.1. A `module-info.java` Descriptor

Java 9 has introduced the JPMS.

Therefore, the RxMicro framework, which requires the use of JDK 11 or higher, requires a `module-info.java` descriptor for any of Your microservice projects.

```
module examples.quick.start {
    requires rxmicro.rest.server.netty;         ①
    requires rxmicro.rest.server.exchange.json; ②
}
```

① Module for building REST-based microservices based on HTTP server that uses Netty, with all required transitive dependencies.

② Module for converting a Java model to JSON format and vice versa, with all required transitive dependencies.

Usually `module-info.java` depends on all RxMicro modules listed in the `dependencies` section of the `pom.xml` file of any of Your microservice projects. Therefore, it is enough to duplicate all the dependencies in `module-info.java`.

*Thanks to the transitive dependencies of the RxMicro framework, the number of modules required has been greatly reduced. Only basic RxMicro modules must be specified!*

### 3.3.2. An HTTP Response Model Class

```
package io.rxmicro.examples.quick.start;

import static java.util.Objects.requireNonNull;

@SuppressWarnings("SameParameterValue")
final class Response {

    final String message;

    Response(final String message) {
        this.message = requireNonNull(message);
    }
}
```

According to the specification, JSON format supports the following data types: object, array and primitives: strings, logical type, numeric type and `null`.

To simplify communication between REST-based microservices, the RxMicro framework supports only **JSON object** as a return type of any REST-based microservice. Thus, any REST-based microservice built via the RxMicro framework can return **only JSON objects**. In case You need to return a primitive or an array, **You need to create a wrapper class**.

Therefore, to display the `"Hello World!"` string, it is necessary to create a `Response` wrapper class, which is a wrapper above the string data type.

### 3.3.3. A REST-Based Microservice Class

```
package io.rxmicro.examples.quick.start;

import io.rxmicro.rest.method.GET;
import io.rxmicro.rest.server.RxMicroRestServer;

import java.util.concurrent.CompletableFuture;

public final class HelloWorldMicroService {

    @GET("/")
    CompletableFuture<Response> sayHelloWorld() {       ①
        return CompletableFuture.supplyAsync(() ->
                new Response("Hello World!"));          ②
    }

    public static void main(final String[] args) {              ③
        RxMicroRestServer.startRestServer(HelloWorldMicroService.class);  ④
    }
}
```

① REST-based microservice contains a handler of HTTP GET method: `sayHelloWorld`, which doesn't accept any parameters and returns a `CompletableFuture` reactive type.

② The `CompletableFuture.supplyAsync()` static method is used to create an object of `CompletableFuture` class.

③ To launch a REST-based microservice the `main` method is used.

④ The launch is performed using the `RxMicroRestServer.startRestServer(Class<?>)` static method, which requires the REST-based microservice class as parameter.

Note that the HTTP request handler method doesn't need to be `public`.

The `protected` and `<default>` modifiers are also supported by the RxMicro framework.

### 3.3.4. A Structure of the Microservice Project

The above-mentioned components of the microservice project should be located in the project according to the following screenshot:

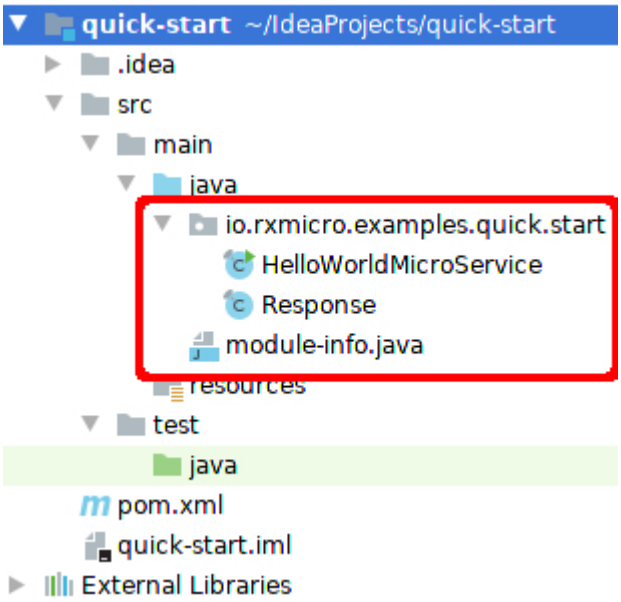



*Figure 4. A structure of the microservice project*

## 3.4. Compiling the Project

### 3.4.1. Using the `maven`

To compile a project using the `maven`, open the terminal in the project root folder and proceed with the following command:

```
mvn clean compile
```

In order to run `maven` commands in the terminal it is necessary to:

- Install `JDK` and `maven` on Your computer;
- Set the `JAVA_HOME` environment variable;
- Add the path to `$MAVEN_HOME/bin` folder to the `$PATH` environment variable.

*A detailed instruction on the `maven` installation for its further use in the terminal can be found at the following link: `Installing Apache Maven`.*

It is possible to compile the project with `maven` even without using the terminal. Since any modern IDE for Java contains built-in `maven`, You can use this built-in `maven` tool.

To do this, open the `maven panel` and execute the specified commands with a mouse or touchpad manipulator. For example, the `maven panel` in IntelliJ IDEA looks like:

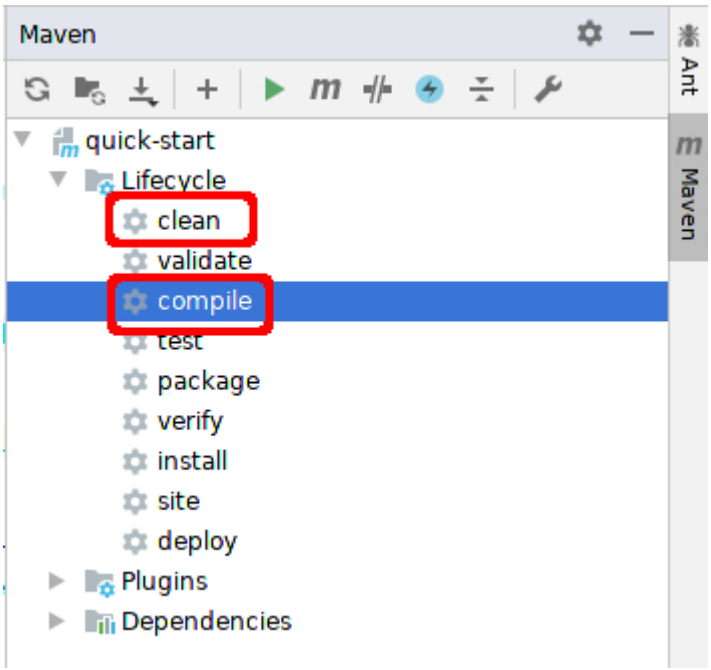



*Figure 5. Maven panel in IntelliJ IDEA*

After successful compilation, the `RxMicro Annotation Processor` work results are displayed in the terminal:

```
[INFO] ------------------------------------------------------------------------
[INFO]  RX-MICRO ANNOTATIONS PROCESSING
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] Current environment context is:
  RxMicro version: 0.11
  Current module:
    examples.quick.start
  Available RxMicro modules:
    rxmicro.common;
    rxmicro.reflection;
    rxmicro.model;
    rxmicro.runtime;
    rxmicro.config;
    rxmicro.logger;
    rxmicro.files;
    rxmicro.http;
    rxmicro.rest;
    rxmicro.rest.server;
    rxmicro.rest.server.netty;
    rxmicro.json;
    rxmicro.exchange.json;
    rxmicro.rest.server.exchange.json;
  Include packages: <none>
  Exclude packages: <none>
[INFO] Found the following REST controllers:
  io.rxmicro.examples.quick.start.HelloWorldMicroService:
    'GET /' -> sayHelloWorld();
[INFO] Generating java classes...
[INFO] All java classes generated successful in 0.031 seconds. ①
[INFO] ------------------------------------------------------------------------
[INFO] Annotations processing completed successful.
[INFO] ------------------------------------------------------------------------
```

① The given information indicates that all files needed to run the microservice have been generated.

In the `target` folder You can find all generated and compiled classes of the microservice project.

To understand how the RxMicro framework works, please go to Section 4.1, “How It Works?” section.

### 3.4.2. Using the `IntelliJ IDEA`

The IntelliJ IDEA allows annotation processors to be launched automatically when building a project. So if You want to compile a microservice project using IntelliJ IDEA rather than `maven`, You need to set up the `Annotation Processors` section in the IntelliJ IDEA.

#### 3.4.2.1. Enable Annotation Processing

To enable annotation processing while building a project with IntelliJ IDEA, You need to set up the `Annotation Processors` section. To do so, open the menu: `File` → `Settings` and get to the tab:`Build, Execution, Deployment` → `Compiler` → `Annotation Processors`.

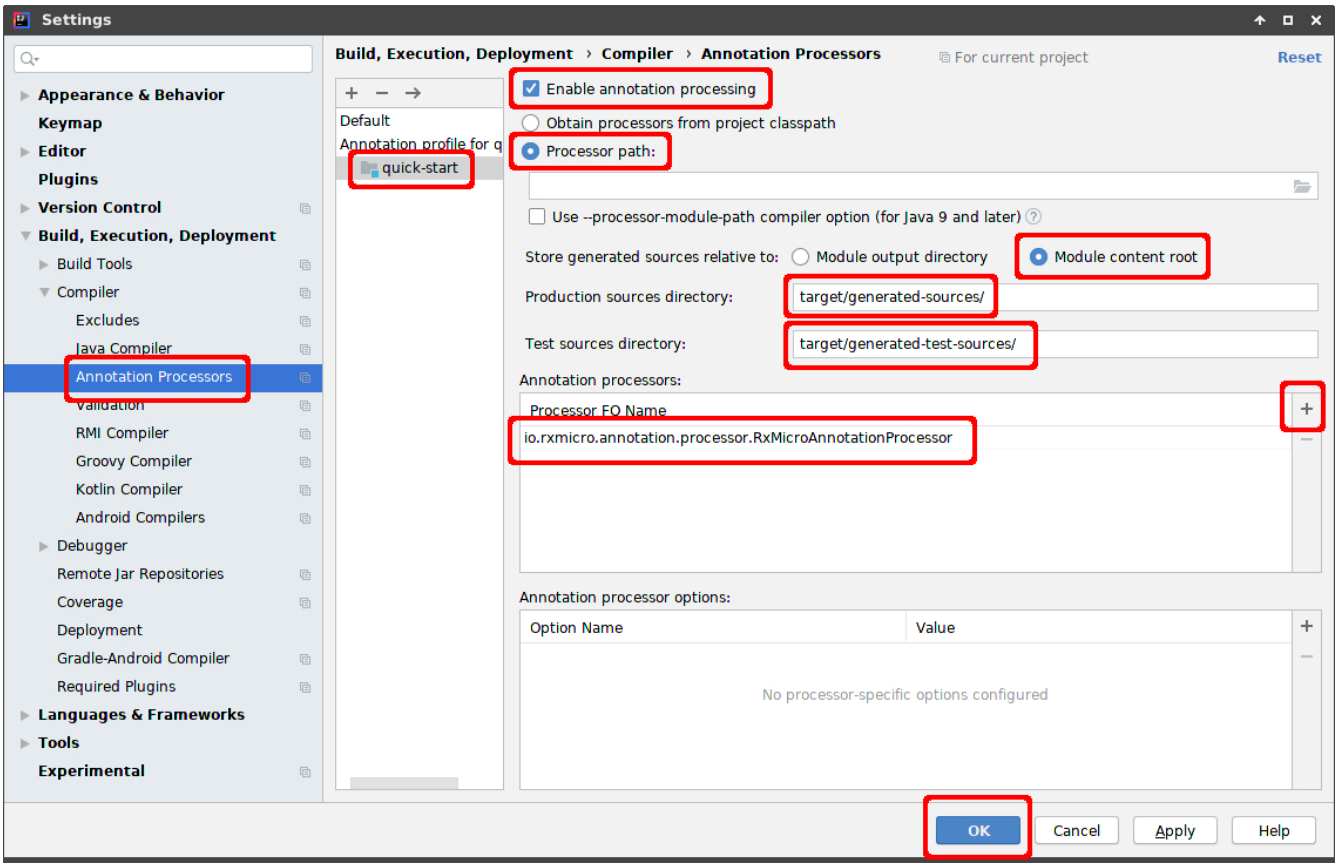



*Figure 6. Required Annotation Processing settings*

Make sure that **all** Your settings of the `Annotation Processors` section correspond to the settings shown in the figure above.

#### 3.4.2.2. Rebuilding the Project

After setting up the `Annotation Processors` section, the project must be rebuilt. To do so, run the following command from the main menu: `Build` → `Rebuild project`.

# 3.5. Starting the Microservice

## 3.5.1. Using the IDE:

You can run the REST-based microservice using the IntelliJ IDEA launch context menu

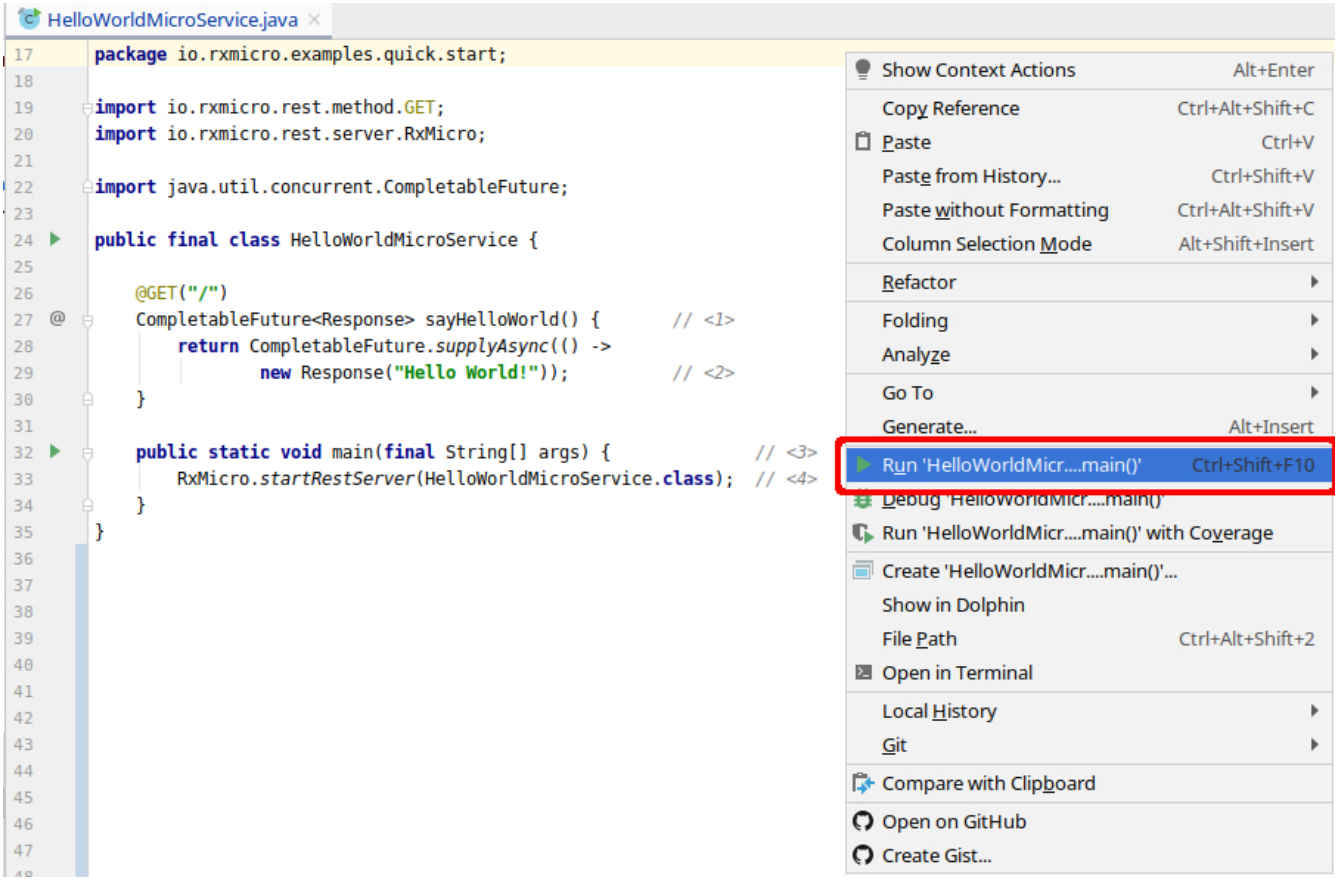
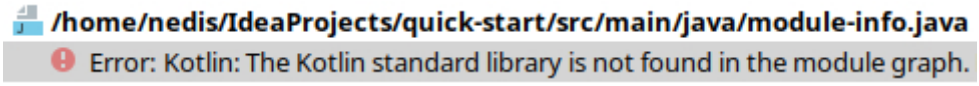



*Figure 7. Run the REST-based micro service using IntelliJ IDEA context menu.*

If You get the following error while starting the REST-based `HelloWorldMicroService` microservice:




rebuild the project!

*(To do this, run the command **Rebuild project** from the main menu: `Build` → `Rebuild project`.)*

After starting, the console will display the following information:

```
2020-02-02 20:14:11.707 [INFO]
io.rxmicro.rest.server.netty.internal.component.NettyServer :
Server started at 0.0.0.0:8080 using NETTY transport in 500 millis ①
```

① The `Server started in ⋯ millis` message means that the RxMicro HTTP server has been successfully started.

If an error occurs during the starting process, the console will display a stack trace of this error.

### 3.5.2. Using the Terminal:

Go to the `target` folder of the microservice project, open the terminal in this folder and run the following command:

```
java -Dfile.encoding=UTF-8 -p ./classes:$M2_REPO/io/rxmicro/rxmicro-rest-server-
netty/0.11/rxmicro-rest-server-netty-0.11.jar:$M2_REPO/io/rxmicro/rxmicro-rest-
server/0.11/rxmicro-rest-server-0.11.jar:$M2_REPO/io/rxmicro/rxmicro-
rest/0.11/rxmicro-rest-0.11.jar:$M2_REPO/io/rxmicro/rxmicro-model/0.11/rxmicro-model-
0.11.jar:$M2_REPO/io/rxmicro/rxmicro-http/0.11/rxmicro-http-
0.11.jar:$M2_REPO/io/rxmicro/rxmicro-logger/0.11/rxmicro-logger-
0.11.jar:$M2_REPO/io/rxmicro/rxmicro-common/0.11/rxmicro-common-
0.11.jar:$M2_REPO/io/rxmicro/rxmicro-config/0.11/rxmicro-config-
0.11.jar:$M2_REPO/io/rxmicro/rxmicro-runtime/0.11/rxmicro-runtime-
0.11.jar:$M2_REPO/io/rxmicro/rxmicro-files/0.11/rxmicro-files-
0.11.jar:$M2_REPO/io/netty/netty-codec-http/4.1.85.Final/netty-codec-http-
4.1.85.Final.jar:$M2_REPO/io/netty/netty-common/4.1.85.Final/netty-common-
4.1.85.Final.jar:$M2_REPO/io/netty/netty-buffer/4.1.85.Final/netty-buffer-
4.1.85.Final.jar:$M2_REPO/io/netty/netty-codec/4.1.85.Final/netty-codec-
4.1.85.Final.jar:$M2_REPO/io/netty/netty-handler/4.1.85.Final/netty-handler-
4.1.85.Final.jar:$M2_REPO/io/netty/netty-transport/4.1.85.Final/netty-transport-
4.1.85.Final.jar:$M2_REPO/io/netty/netty-resolver/4.1.85.Final/netty-resolver-
4.1.85.Final.jar:$M2_REPO/io/rxmicro/rxmicro-rest-server-exchange-json/0.11/rxmicro-
rest-server-exchange-json-0.11.jar:$M2_REPO/io/rxmicro/rxmicro-exchange-
json/0.11/rxmicro-exchange-json-0.11.jar:$M2_REPO/io/rxmicro/rxmicro-
json/0.11/rxmicro-json-0.11.jar -m
examples.quick.start/io.rxmicro.examples.quick.start.HelloWorldMicroService
```

It is assumed that the `M2_REPO` environment variable is set on Your computer. This variable must contain a path to the maven local repository.

By default, the local repository is located in the `$HOME/.m2/repository` folder, where `$HOME` is the home directory *(i.e.* `System.getProperty("user.home")`*)*:

- for Linux platform the `$HOME` directory is `/home/$USERNAME`;
- for MacOS platform the `$HOME` directory is `/Users/$USERNAME`;
- for Windows platform the `$HOME` directory is `C:\Documents and Settings\%USERNAME%` or `C:\Users\%USERNAME%`.

The above example of launching a microservice project using a terminal won't work on `Windows` OS.

Inoperability is caused by the use of different special symbols on Unix (Linux and MacOS) and Windows platforms:

- `Path separator`:
  - Unix uses the `:` symbol.
  - Windows uses the `;` symbol.
- `Path name separator`:
  - Unix uses the `/` symbol.
  - Windows uses the `\` symbol.
- Address to environment variable with the `M2_REPO` name:
  - Unix uses the `$M2_REPO` expression.
  - Windows uses the `%M2_REPO%` expression.

Therefore, in order to launch a microservice project on the `Windows` OS, it is necessary to replace all special symbols.

After starting, the console will display the following information:

```
java -p ./classes: \
        $M2_REPO/io/rxmicro/rxmicro-rest-server-netty/0.11/rxmicro-rest-server-netty-
0.11.jar: \
        $M2_REPO/io/rxmicro/rxmicro-rest-server/0.11/rxmicro-rest-server-0.11.jar: \
        $M2_REPO/io/rxmicro/rxmicro-rest/0.11/rxmicro-rest-0.11.jar: \
        $M2_REPO/io/rxmicro/rxmicro-model/0.11/rxmicro-model-0.11.jar: \
        $M2_REPO/io/rxmicro/rxmicro-http/0.11/rxmicro-http-0.11.jar: \
        $M2_REPO/io/rxmicro/rxmicro-logger/0.11/rxmicro-logger-0.11.jar: \
        $M2_REPO/io/rxmicro/rxmicro-common/0.11/rxmicro-common-0.11.jar: \
        $M2_REPO/io/rxmicro/rxmicro-config/0.11/rxmicro-config-0.11.jar: \
        $M2_REPO/io/rxmicro/rxmicro-runtime/0.11/rxmicro-runtime-0.11.jar: \
        $M2_REPO/io/rxmicro/rxmicro-files/0.11/rxmicro-files-0.11.jar: \
        $M2_REPO/io/rxmicro/rxmicro-rest-server-exchange-json/0.11/rxmicro-rest-
server-exchange-json-0.11.jar: \
        $M2_REPO/io/rxmicro/rxmicro-exchange-json/0.11/rxmicro-exchange-json-0.11.jar:
\
        $M2_REPO/io/rxmicro/rxmicro-json/0.11/rxmicro-json-0.11.jar: \
        $M2_REPO/io/netty/netty-codec-http/4.1.85.Final/netty-codec-http-
4.1.85.Final.jar: \
        $M2_REPO/io/netty/netty-common/4.1.85.Final/netty-common-4.1.85.Final.jar: \
        $M2_REPO/io/netty/netty-buffer/4.1.85.Final/netty-buffer-4.1.85.Final.jar: \
        $M2_REPO/io/netty/netty-codec/4.1.85.Final/netty-codec-4.1.85.Final.jar: \
        $M2_REPO/io/netty/netty-handler/4.1.85.Final/netty-handler-4.1.85.Final.jar: \
        $M2_REPO/io/netty/netty-transport/4.1.85.Final/netty-transport-
4.1.85.Final.jar: \
        $M2_REPO/io/netty/netty-resolver/4.1.85.Final/netty-resolver-4.1.85.Final.jar
\
    -m examples.quick.start/io.rxmicro.examples.quick.start.HelloWorldMicroService

2020-02-02 20:14:11.707 [INFO]
io.rxmicro.rest.server.netty.internal.component.NettyServer :
Server started at 0.0.0.0:8080 using NETTY transport in 500 millis ①
```

① The `Server started in ⋯ millis` message means that the RxMicro HTTP server has been successfully started.

When starting the microservice via the terminal, it's quite inconvenient to list all dependencies and their versions. To solve this problem, You can use the `maven-dependency-plugin`, which can copy all project dependencies. To activate the `maven-dependency-plugin`, You must add it to `pom.xml`:

```
<plugin>
    <artifactId>maven-dependency-plugin</artifactId>
    <version>${maven-dependency-plugin.version}</version>①
    <executions>
        <execution>
            <id>copy-dependencies</id>
            <phase>package</phase>②
            <goals>
                <goal>copy-dependencies</goal>
            </goals>
            <configuration>
                <outputDirectory>${project.build.directory}/lib</outputDirectory>③
                <includeScope>compile</includeScope>④
            </configuration>
        </execution>
    </executions>
</plugin>
```

① The latest stable version of the `maven-dependency-plugin`.

② The plugin is invoked during the `package` phase.

③ Target folder all dependencies should be copied to.
*(In the example above, this is the `target/lib` folder.)*

④ This setting specifies what scope of dependencies should be copied.
*(This option allows excluding libraries required for testing or libraries, those already present on the client's computer.)*

After adding the plugin, You need to execute the command:

```
mvn clean package
```

As a result of running the command, the `maven-dependency-plugin` will copy all the dependencies to the `target/lib` folder:

*Figure 8. The required dependencies for the simplest REST-based micro service.*

Now You can simplify the start command
*(Instead of listing all the libraries, specify the* `lib` *folder)*:

```
java -p ./classes:lib -m
examples.quick.start/io.rxmicro.examples.quick.start.HelloWorldMicroService
```

The above example of launching a microservice project using a terminal won't work on `Windows` OS.

Inoperability is caused by the use of different special symbols on Unix (Linux and MacOS) and Windows platforms:

- `Path separator`:
  - Unix uses the `:` symbol.
  - Windows uses the `;` symbol.
- `Path name separator`:
  - Unix uses the `/` symbol.
  - Windows uses the `\` symbol.
- Address to environment variable with the `M2_REPO` name:
  - Unix uses the `$M2_REPO` expression.
  - Windows uses the `%M2_REPO%` expression.

Therefore, in order to launch a microservice project on the `Windows` OS, it is necessary to replace all special symbols.

## 3.6. Verifying the Microservice

To receive the `"Hello World!"` message from the created REST-based microservice, execute `GET` request to `localhost:8080` endpoint:

```
:$ curl -v localhost:8080

* Rebuilt URL to: localhost:8080/
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 8080 (#0)

> GET / HTTP/1.1 ①
> Host: localhost:8080
> User-Agent: curl/7.58.0
> Accept: /
>

< HTTP/1.1 200 OK ②
< Server: RxMicro-NettyServer/0.11
< Date: Thu, 2 Jan 2020 11:48:13 GMT
< Content-Type: application/json
< Content-Length: 25
< Request-Id: 62jJeu8x1310662
<
* Connection #0 to host localhost left intact
{"message":"Hello World!"} ③
```

① **curl** sends a GET request.

② HTTP server successfully returns a response.

③ The HTTP body contains a JSON response with the `"Hello World!"` message.

Therefore, the created REST-based microservice works correctly!

You can also use Your favorite browser instead of `curl` to verify if the REST-based microservice is working correctly.

# 3.7. Automated Test

The RxMicro framework provides modules for effective writing of any type of tests. Among all supported test types, a REST-based microservice test is required for the current project.

## 3.7.1. Configuring the Project

Before writing a REST-based microservice test, You need to configure `pom.xml` of Your project by performing the following steps:

1. Add the required dependencies to `pom.xml`.
2. Configure the `maven-compiler-plugin`.
3. Configure the `maven-surefire-plugin`.

### 3.7.1.1. Adding the Required Dependencies

Before using RxMicro modules for testing, You need to add the following dependencies to the project:

```
<dependencies>
    <dependency>
        <groupId>io.rxmicro</groupId>
        <artifactId>rxmicro-test-junit</artifactId> ①
        <version>${rxmicro.version}</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>io.rxmicro</groupId>
        <artifactId>rxmicro-rest-client-exchange-json</artifactId> ②
        <version>${rxmicro.version}</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

① Unit testing library based on the JUnit 5 framework

② Library for Java model conversion to JSON format and vice versa on the HTTP client side;

The REST-based microservice testing process consists in launching the REST-based microservice and sending a request to the microservice via HTTP client.
*(Therefore, in `maven` dependencies it's necessary to add the library supporting the JSON format on the HTTP client side (`rxmicro-rest-client-exchange-json`)).*
After receiving a response from the microservice, the response is compared to the expected one.

### 3.7.1.2. Configuring the `maven-compiler-plugin`

Since the RxMicro framework uses the Java annotation processors, You need to configure `maven-compiler-plugin`:

```
<plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>${maven-compiler-plugin.version}</version>
    <configuration>
        <release>11</release>
        <annotationProcessorPaths>
            <annotationProcessorPath>
                <groupId>io.rxmicro</groupId>
                <artifactId>rxmicro-annotation-processor</artifactId>
                <version>${rxmicro.version}</version>
            </annotationProcessorPath>
        </annotationProcessorPaths>
    </configuration>
    <executions>
        <execution>
            <id>source-compile</id>
            <goals>
                <goal>compile</goal>
            </goals>
            <configuration>
                <annotationProcessors>
                    <annotationProcessor>
            io.rxmicro.annotation.processor.RxMicroAnnotationProcessor
                    </annotationProcessor>
                </annotationProcessors>
                <generatedSourcesDirectory>
                    ${project.build.directory}/generated-sources/
                </generatedSourcesDirectory>
            </configuration>
        </execution>
        <execution>
            <id>test-compile</id> ①
            <goals>
                <goal>testCompile</goal>
            </goals>
            <configuration>
                <annotationProcessors>
                    <annotationProcessor>
            io.rxmicro.annotation.processor.RxMicroTestsAnnotationProcessor ②
                    </annotationProcessor>
                </annotationProcessors>
                <generatedTestSourcesDirectory>
                    ${project.build.directory}/generated-test-sources/ ③
                </generatedTestSourcesDirectory>
            </configuration>
        </execution>
    </executions>
</plugin>
```

① The tests require a separate configuration, so a new `execution` must be added.

② The annotation processor class that handles test configuration.

③ Location of Java-generated classes by the test annotation processor.

To learn more about how the `RxMicro Annotation Processor` works in the test environment, please go to the following section: Section 14.4, “How It Works”.

#### 3.7.1.3. Configuring the `maven-surefire-plugin`

For a successful tests launch while building a project with `maven` it is necessary to update `maven-surefire-plugin`:

```
<plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>${maven-surefire-plugin.version}</version> ①
    <configuration>
        <properties>
            <!--
https://junit.org/junit5/docs/5.5.1/api/org/junit/jupiter/api/Timeout.html -->
            <configurationParameters>
                junit.jupiter.execution.timeout.default = 60                ②
                junit.jupiter.execution.timeout.mode = disabled_on_debug    ③
                junit.jupiter.execution.parallel.enabled = false            ④
            </configurationParameters>
        </properties>
    </configuration>
</plugin>
```

① Last stable version of `maven-surefire-plugin`.
*(The plugin version must be `2.22.1` or higher, otherwise `maven` will ignore the tests!.)*

② In case of an error in the code which uses reactive programming, an infinite function execution may occur. In order to detect such cases, it is necessary to set a global timeout for all methods in the tests. *(By default, timeout is set in seconds. More detailed information on timeouts configuration is available in official JUnit5 documentation.)*

③ While debugging, timeouts can be turned off.

④ This property is useful for the tests debugging from IDE or `maven`.
*(By setting this property the speed of test performance will decrease, so use this property for debugging only!)*

### 3.7.2. Creating a Test Class

REST-based microservice test is a one class containing one test method:

```
package io.rxmicro.examples.quick.start;

import io.rxmicro.test.BlockingHttpClient;
import io.rxmicro.test.ClientHttpResponse;
import io.rxmicro.test.junit.RxMicroRestBasedMicroServiceTest;
import org.junit.jupiter.api.Test;

import static io.rxmicro.test.json.JsonFactory.jsonObject;
import static org.junit.jupiter.api.Assertions.assertEquals;

@RxMicroRestBasedMicroServiceTest(HelloWorldMicroService.class) ①
final class HelloWorldMicroServiceTest {

    private BlockingHttpClient blockingHttpClient; ②

    @Test
    void Should_return_Hello_World_message() {
        final ClientHttpResponse response = blockingHttpClient.get("/"); ③
        assertEquals(
                jsonObject("message", "Hello World!"), ④
                response.getBody()
        );
        assertEquals(200, response.getStatusCode()); ⑤
    }
}
```

① The `RxMicro Test Annotation` indicating which microservice should be run for testing.

② The `BlockingHttpClient` is a basic HTTP client interface designed for use in tests. This interface allows executing **blocking** requests to the microservice via the `HTTP` protocol. This field is initialized automatically when running the test with `reflection`. Upon initialization it refers to the test HTTP server that was automatically started for the test.

③ Blocking request to the microservice.

④ Comparing the contents of an HTTP body with an expected value.

⑤ Comparing the HTTP status code with an expected value.

For low-level and effective work with JSON format, the RxMicro framework provides a separate `rxmicro.json` module.

To get a common idea of the capabilities of this module, which are required when writing tests, go to the following section: Section 4.10, “JSON”.

**In microservice tests it is recommended to compare the HTTP request body before comparing the HTTP status, when the microservice constantly returns a text error message! (This will make it easier to understand the error in case it occurred during the testing.)**

**If the microservice returns ONLY the status when an error occurs, the HTTP body comparison should be skipped!**

For further information on how to test REST-based microservices, go to the following section: Section 14.6, “REST-based Microservice Testing”.

### 3.7.3. Starting the Test Class

To start the tests, You need to run the command:

```
mvn clean test
```

After starting, the console will display the following information:

```
...
[INFO] ------------------------------------------------------------------------
[INFO]  RX-MICRO ANNOTATIONS PROCESSING ①
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] Current environment context is:
  RxMicro version: 0.11
  Current module:
    examples.quick.start
  Available RxMicro modules:
    ...
  Include packages: <none>
  Exclude packages: <none>
[INFO] Found the following REST controllers:
  io.rxmicro.examples.quick.start.HelloWorldMicroService:
    'GET /' -> sayHelloWorld();
[INFO] Generating java classes...
[INFO] All java classes generated successful in 0.030 seconds. ②
[INFO] ------------------------------------------------------------------------
[INFO] Annotations processing completed successful.
[INFO] ------------------------------------------------------------------------
...
[INFO] ------------------------------------------------------------------------
[INFO]  RX-MICRO TEST ANNOTATIONS PROCESSING ③
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] Current environment context is:
  RxMicro version: 0.11
  Current module:
    examples.quick.start
  Available RxMicro modules:
    ...
  Include packages: <none>
  Exclude packages: <none>
[INFO] Generating java classes...
[INFO] Test fixer class generated successfully:
rxmicro.$$RestBasedMicroServiceTestFixer ④
```

```
[INFO] All java classes generated successful in 0.009 seconds. ④
[INFO] ------------------------------------------------------------------------
[INFO] Annotations processing completed successful.
[INFO] ------------------------------------------------------------------------
...
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
...
[INFO] Fix the environment for REST based microservice test(s)... ⑤
[INFO] opens examples.quick.start/rxmicro to ALL-UNNAMED ⑤
[INFO] opens examples.quick.start/io.rxmicro.examples.quick.start to ALL-UNNAMED ⑤
[INFO] opens examples.quick.start/rxmicro to rxmicro.reflection ⑤
[INFO] opens examples.quick.start/io.rxmicro.examples.quick.start to
rxmicro.reflection ⑤
[INFO] opens rxmicro.rest.server.netty/io.rxmicro.rest.server.netty.local to ALL-
UNNAMED ⑤
[INFO] opens rxmicro.runtime/io.rxmicro.runtime.local to ALL-UNNAMED ⑤
[INFO] opens rxmicro.runtime/io.rxmicro.runtime.local.error to ALL-UNNAMED ⑤
[INFO] opens rxmicro.runtime/io.rxmicro.runtime.local.provider to ALL-UNNAMED ⑤
[INFO] opens rxmicro.config/io.rxmicro.config.local to ALL-UNNAMED ⑤
[INFO] opens rxmicro.rest.server/io.rxmicro.rest.server.local.model to ALL-UNNAMED ⑤
[INFO] opens rxmicro.rest.server/io.rxmicro.rest.server.local.component to ALL-UNNAMED
⑤
[INFO] opens rxmicro.common/io.rxmicro.common.local to ALL-UNNAMED ⑤
[INFO] opens rxmicro.http/io.rxmicro.http.local to ALL-UNNAMED ⑤
[INFO] Running io.rxmicro.examples.quick.start.HelloWorldMicroServiceTest ⑥
[INFO] ...NettyServer: Server started at 0.0.0.0:38751 using NETTY transport. ⑦
[INFO] ...Router: Mapped "GET '/'  onto ...HelloWorldMicroService.sayHelloWorld()
[INFO] ...Router: Mapped "GET '/bad-request'  onto
...BadHttpRequestRestController.handle(...)
[INFO] ...HelloWorldMicroServiceTest: JdkHttpClient released ⑧
[INFO] ...NettyServer: Retrieved shutdown request ...
[INFO] ...NettyServer: Server stopped ⑨
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0 ⑩
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

① Starting of the `RxMicro Annotation Processor`.

② `RxMicro Annotation Processor` has successfully completed its work.

③ Starting of the `RxMicro Tests Annotation Processor`.

④ `RxMicro Tests Annotation Processor` has successfully completed its work.

⑤ For the test configuration, missing exports were automatically added using the capabilities of the `java.lang.Module` class.

⑥ REST-based microservice test starting.

⑦ HTTP server has started automatically on random free port.

⑧ The resources of the `BlockingHttpClient` component have been released.

⑨ HTTP server has stopped successfully.

For further information on how the `RxMicro Annotation Processor` works for the test environment, go to the following section: Section 14.4, “How It Works”.

## 3.8. The Project at the Github

The REST-based `HelloWorldMicroService` microservice project is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/quick-start`.

The `pom.xml` configuration file is a reference configuration and can be copied to Your next project using the RxMicro framework.

**DON'T FORGET to remove the link to the parent project**:

```
<parent>
    <artifactId>examples</artifactId>
    <groupId>io.rxmicro</groupId>
    <version>0.11</version>
</parent>
```

**and add the `properties` section describing the versions of the libraries used**:

```
<properties>
    <rxmicro.version>0.11</rxmicro.version> ①

    <maven-compiler-plugin.version>3.10.1</maven-compiler-
plugin.version> ②
    <maven-surefire-plugin.version>3.0.0-M7</maven-surefire-
plugin.version> ③
    <maven-dependency-plugin.version>3.2.0</maven-dependency-
plugin.version> ④
</properties>
```

① The latest stable version of the RxMicro framework.

② The latest stable version of the `maven-compiler-plugin`.

③ The latest stable version of the `maven-surefire-plugin`.

④ The latest stable version of the `maven-dependency-plugin`.

# 4. Core Concepts

This section will describe the basic working concepts of the RxMicro framework.

## 4.1. How It Works?

The RxMicro framework uses the Java annotation processor, which generates standard code using `RxMicro Annotations`.

Thus, the RxMicro framework is a framework of declarative programming.

Using the RxMicro framework, the developer focuses on writing the business logic of a microservice. Then he configures the desired standard behavior with `RxMicro Annotations`. When compiling a project, the `RxMicro Annotation Processor` generates additional classes. Generated classes contain a standard logic that ensures the functionality of the created microservice.

### 4.1.1. A Common Work Schema

The common work schema can be presented as follows:

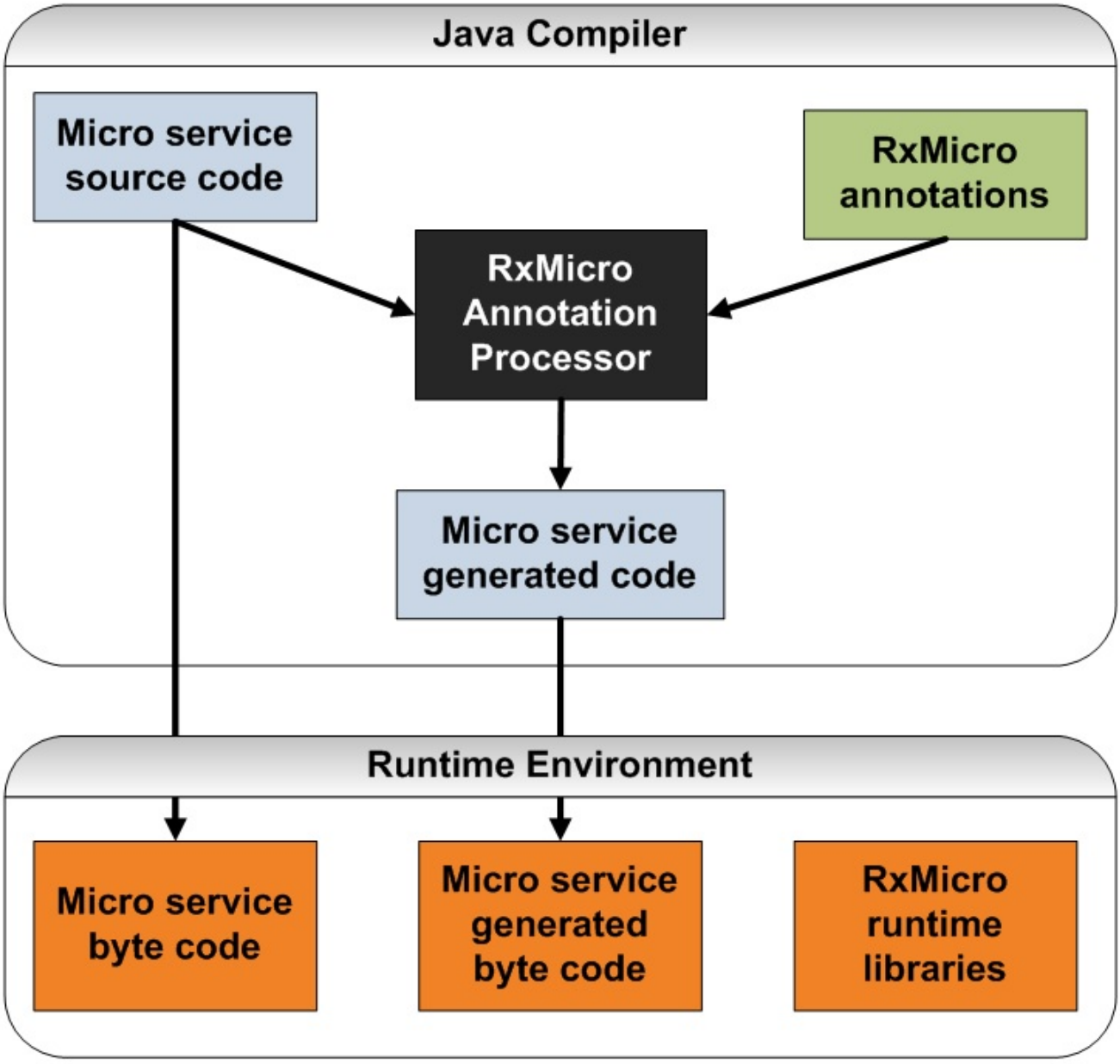



*Figure 9. The RxMicro framework common work schema.*

While solving a business task, the developer writes `Micro service source code`. Then the developer configures the desired standard microservice behavior via `RxMicro Annotations`. After that, the developer compiles the project.

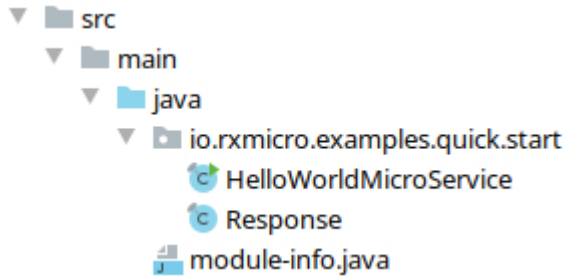
Since the `RxMicro Annotation Processor` is configured in `maven`, when compiling a project this processor handles the source code of the microservice and generates the additional classes: `Micro service generated code`. After that, the compiler compiles the source and generated microservice codes: `Micro service byte code` and `Micro service generated byte code`.

The compiled source and generated codes along with the `RxMicro runtime libraries` perform useful work.

### 4.1.2. Generating of Additional Classes.

Let's have a look at the RxMicro framework common work schema, by the example of the REST-based microservice project, which displays the `"Hello World!"` message in JSON format.
*(This project was considered in the Section 3, "Quick Start" section.)*

While implementing a business task (in this example, it's a task of displaying the `"Hello World!"` message in JSON format) the developer wrote the following `Micro service source code`:



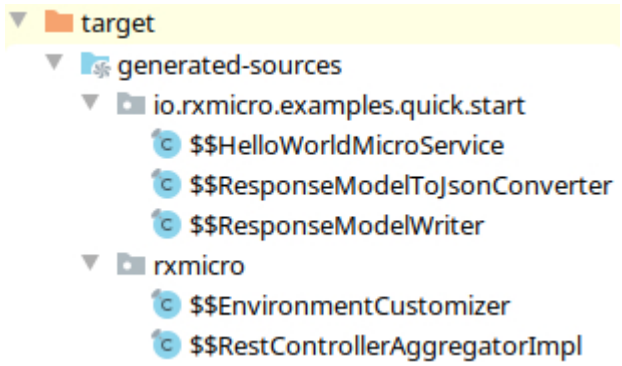

In order to inform the RxMicro framework about the need to generate additional classes by which a written `Micro service source code` can be built into an HTTP server to handle the desired HTTP requests, the developer added the following `RxMicro Annotation`:

```
@GET("/")
```

Since the `RxMicro Annotation Processor` is configured in maven:

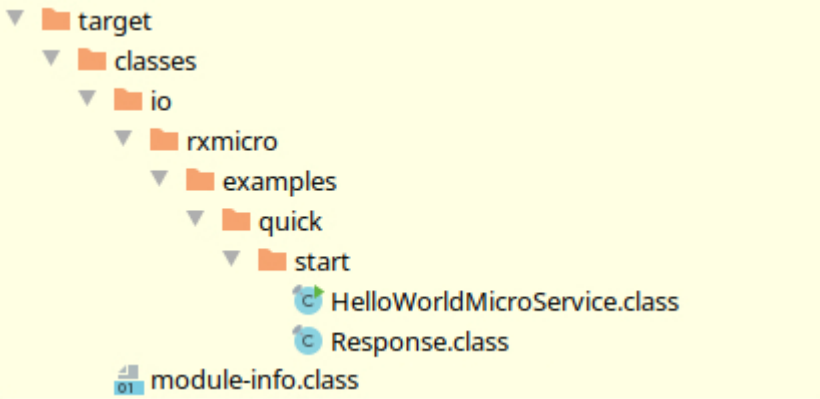
```
<configuration>
    <annotationProcessors>
        <annotationProcessor>
            io.rxmicro.annotation.processor.RxMicroAnnotationProcessor
        </annotationProcessor>
    </annotationProcessors>
</configuration>
```

then when compiling a project this processor handles the source code of the REST-based microservice and generates the `Micro service generated code` additional classes:

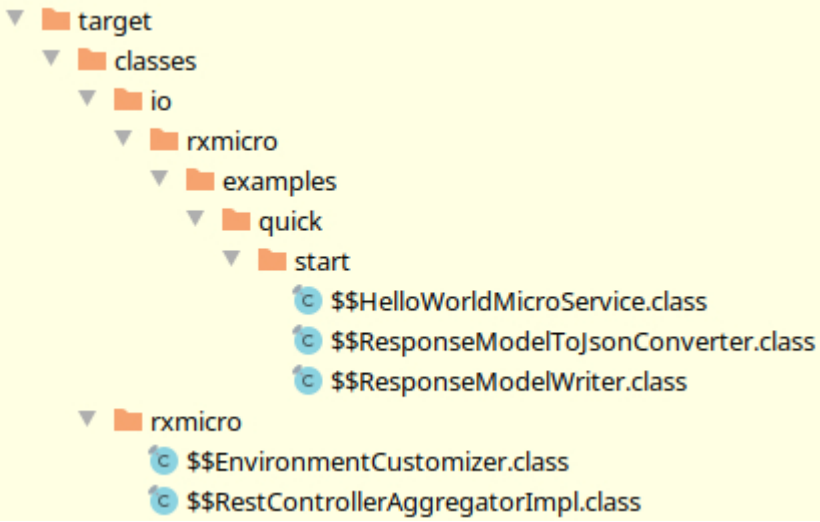



After the source code of additional classes was successfully generated by the `RxMicro Annotation Processor`, the compiler compiles:

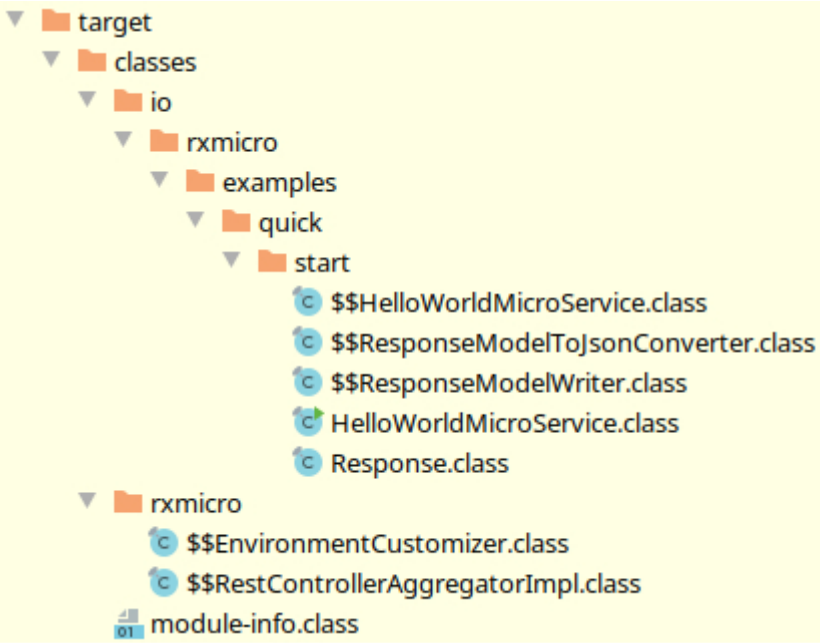
- REST-based microservice source code in `Micro service byte code`:




- Generated code of additional classes in `Micro service byte code`:




As a result of the compiler's work, the REST-based microservice byte code and the byte code of the generated additional classes will be stored jointly in the same `jar` archive:




For successful start of the compiled classes, the `RxMicro runtime libraries` are required:

The `Micro service byte code`, `Micro service byte code` and `RxMicro runtime libraries` are the program components of microservice, which perform useful work.

Below we will look closely at each generated additional class and what functions it performs.

The names of all classes generated by the RxMicro framework start with the `$$` prefix.

#### 4.1.2.1. An Additional Class for the REST Controller.

Any REST-based microservice, contains at least one REST controller. For the simplest project, REST-based microservice and REST controller are the same class.

Therefore, when analyzing such projects, such terms as **REST controller**, **REST-based microservice** and **microservice** are synonymous, because physically they are the same class.

The considered REST-based microservice, which displays the `"Hello World!"` message, is the simplest project, therefore the `HelloWorldMicroService` class is a REST controller.

For more information on the differences between **REST controller**, **REST-based microservice** and **microservice**, refer to the {microservice}.pdf section.

For each REST controller class the RxMicro framework generates an additional class that performs the following functions:

- Creates a REST controller object.
  *(In case of `rxmicro.cdi` module activation, after creation it also injects the required dependencies.)*
- Creates `ModelReader` objects that convert HTTP request parameters, headers and body to Java model.
- Creates `ModelWriter` objects that convert the Java response model to HTTP response headers and body;
- Registers all HTTP request handlers of current REST controller in the router.
- When receiving an HTTP request via the `ModelReader` object, converts the HTTP request to the Java request model and invokes the corresponding REST controller handler.
- After receiving the resulting Java response model via the `ModelWriter` object, converts the Java model into an HTTP response and sends the response to the client.

Such an additional class for the `HelloWorldMicroService` class is the `$$HelloWorldMicroService` class:

```
public final class $$HelloWorldMicroService extends AbstractRestController {

    private HelloWorldMicroService restController;

    private $$ResponseModelWriter responseModelWriter;

    @Override
    protected void postConstruct() {
        restController = new HelloWorldMicroService(); ①
        responseModelWriter =
                new $$ResponseModelWriter(restServerConfig.isHumanReadableOutput());
②
    }

    @Override
    public void register(final RestControllerRegistrar registrar) { ③
        registrar.register(
                this,
                new Registration(
                        "",
                        "sayHelloWorld()",
                        this::sayHelloWorld, ④
                        false,
                        new ExactUrlRequestMappingRule( ⑤
                                "GET",
                                "/",
                                false
                        )
                )
        );
    }
```

```
    private CompletionStage<HttpResponse> sayHelloWorld(final PathVariableMapping
mapping,
                                                        final HttpRequest request) {
        final HttpHeaders headers = HttpHeaders.of();
        return restController.sayHelloWorld() ⑥
                .thenApply(response -> buildResponse(response, 200, headers)); ⑦
    }

    private HttpResponse buildResponse(final Response model,
                                       final int statusCode,
                                       final HttpHeaders headers) {
        final HttpResponse response = httpResponseBuilder.build();
        response.setStatus(statusCode);
        response.setOrAddHeaders(headers);
        responseModelWriter.write(model, response); ⑧
        return response;
    }

}
```

① The `$$HelloWorldMicroService` component creates an instance of the REST controller class.

② The `$$HelloWorldMicroService` component creates an instance of the `ModelWriter` that converts the Java response model to the HTTP response headers and body.

③ The `$$HelloWorldMicroService` component registers all HTTP request handlers of the current REST controller.

④ The registration object contains a reference to the HTTP request handler of the current REST controller.

⑤ The registration object contains a rule, according to which the router determines whether to invoke this HTTP request handler.

⑥ When receiving HTTP request, the `$$HelloWorldMicroService` invokes REST controller method.

⑦ After invoking the REST controller method, an asynchronous result handler is added.
*(When using the reactive approach, the current thread cannot be blocked, so the `thenApply` method is used for delayed result handling.)*

⑧ After receiving the Java response model object, the result handler creates an HTTP response based on the data received from the model, which is subsequently sent to the client.

#### 4.1.2.2. An `ModelWriter` Class.

To convert a Java model to an HTTP response, You will need a separate component that performs the following functions:

- Defines in what format to return an HTTP response depending on the project settings.
- Creates converter objects that support the specified messaging format.
- When converting a Java model to an HTTP response, manages the conversion process by delegating invocations to the appropriate components.

Such a separate component for the `Response` model class is the `$$ResponseModelWriter` class:

The code of the `$$ResponseModelWriter` generated class depends on the response model class structure, and the format used for message exchange with the client.

Since the format of message exchange with the client is set in `module-info.java` of the project (`requires rxmicro.rest.server.exchange.json;`), and is the configuration for all REST controllers and all their handlers, then within the current project, the `$$ResponseModelWriter` will depend only on the response model class structure.

Therefore, if several handlers from different REST controllers will return the `Response` class model, only one `$$ResponseModelWriter` class will be generated. As a result, in each additional REST controller class, the instance of this class will be used.

```
public final class $$ResponseModelWriter extends ModelWriter<Response> {

    private final $$ResponseModelToJsonConverter responseModelToJsonConverter; ①

    private final ExchangeDataFormatConverter<Object> exchangeDataFormatConverter; ②

    private final String outputMimeType;

    public $$ResponseModelWriter(final boolean humanReadableOutput) {
        exchangeDataFormatConverter =
            new JsonExchangeDataFormatConverter(humanReadableOutput); ③
        responseModelToJsonConverter = new $$ResponseModelToJsonConverter();
        outputMimeType = exchangeDataFormatConverter.getMimeType();
    }

    @Override
    public void write(final Response model,
                      final HttpResponse response) {
        response.setHeader(HttpHeaders.CONTENT_TYPE, outputMimeType); ④
        final Map<String, Object> json =
responseModelToJsonConverter.toJsonObject(model); ⑤
        response.setContent(exchangeDataFormatConverter.toBytes(json)); ⑥
    }

}
```

① Since the JSON message exchange format is specified in the settings, a component that can convert the Java response model to a JSON response model is required. *(This task is specific for each response model, so to avoid using `reflection`, You need to generate a separate converter component.)*

② To convert any low-level model (in this example, it's a JSON response model) into a byte array, You also need a separate converter component.

③ Since the JSON messaging format is specified in the settings, it is assumed that the JSON model

will be converted to an byte array, which will be created from the Java response model.

④ Since the JSON message exchange format is specified in the settings, it is necessary to set the HTTP header: `Content-Type = application/json`.

⑤ When the HTTP response is formed, it is necessary to convert Java response model to JSON model.

⑥ The last step is to convert the JSON model to a byte array, that will be written to the HTTP response body.

#### 4.1.2.3. A Java Model Converter.

To avoid using `reflection`, You need a component that can convert Java model to JSON model.

This component must support the following functions:

- Convert Java model to JSON model of any complexity.
- Support all possible class field access models to be an all-purpose tool.
  *(Supported class field access models are described in details in the Section 4.7, "Encapsulation".)*

Such a separate component for the `Response` model class is the `$$ResponseModelToJsonConverter` class:

```
public final class $$ResponseModelToJsonConverter extends
ModelToJsonConverter<Response> {

    @Override
    ①
    public Map<String, Object> toJsonObject(final Response model) {
        final JsonObjectBuilder builder = new JsonObjectBuilder();
        putValuesToBuilder(model, builder);
        return builder.build();
    }

    public void putValuesToBuilder(final Response model,
                                   final JsonObjectBuilder builder) {
        builder.put("message", model.message); ②
    }
}
```

① JSON object is presented as `Map<String, Object>`.
*(More information about JSON format support by the RxMicro framework can be found in the Section 4.10, "JSON".)*

② The value of the `message` field is read from the Java model by direct reference to the field.
*(Supported class field access models are described in details in the Section 4.7, "Encapsulation".)*

#### 4.1.2.4. An Aggregator of the REST Controllers.

To integrate developer code into the RxMicro framework infrastructure, You need aggregators.

The aggregators perform the following functions:

- Register all generated additional classes for REST controllers;
- Customize the runtime environment;

The aggregators are invoked by the RxMicro framework using `reflection`.
*(That' s why aggregators have a permanent and predefined names and are located in the special package: `rxmicro`.)*

**An Aggregator of the REST Controllers for any project is always the `rxmicro.MODULE_NAME.$$RestControllerAggregatorImpl` class**:

```
package rxmicro.MODULE_NAME; ①

public final class $$RestControllerAggregatorImpl extends RestControllerAggregator {
②

    static {
        $$EnvironmentCustomizer.customize(); ③
    }

    protected List<AbstractMicroService> listAllRestControllers() {
        return List.of(
                new io.rxmicro.examples.quick.start.$$HelloWorldMicroService() ④
        );
    }

}
```

① All aggregators are **always** generated in the special package: `rxmicro.MODULE_NAME`, where `MODULE_NAME` is the module name or `unnamed` constant if the current project does not declare the `module-info.java` descriptor (Read more: `Unnamed Modules Support`).

② The predefined name of the REST controller aggregator class is always `$$RestControllerAggregatorImpl`.

③ When the aggregator class is loaded by the RxMicro framework, the component of the current environment customization is invoked.

④ The aggregator registers all generated additional classes for REST controllers;

#### 4.1.2.5. An Environment Customizer.

Java 9 has introduced the `JPMS`.

This system requires that a developer configures access to classes in the `module-info.java` file of the microservice project.

To enable the RxMicro framework to load aggregator classes, You must export the `rxmicro.MODULE_NAME` package to the `rxmicro.reflection` module:

```
module examples.quick.start {
    requires rxmicro.rest.server.netty;
    requires rxmicro.rest.server.exchange.json;

    exports rxmicro.MODULE_NAME to rxmicro.reflection; ①
}
```

① Allow access of reflection util classes from the `rxmicro.reflection` module to all classes from the `rxmicro.MODULE_NAME` package.

But the `rxmicro.MODULE_NAME` package is created automatically and after deleting all the generated files, it won't be possible to compile the `module-info.java` because of the following error:

`package is empty or does not exist: rxmicro.MODULE_NAME`.

To solve this problem, the RxMicro framework generates the `rxmicro.MODULE_NAME.$$EnvironmentCustomizer` class:

```
package rxmicro.MODULE_NAME; ①

final class $$EnvironmentCustomizer {

    static {
        exportTheRxmicroPackageToReflectionModule(); ②
        invokeAllStaticSections($$EnvironmentCustomizer.class.getModule(),
"$$EnvironmentCustomizer"); ③
        // All required customization must be here
    }

    public static void customize() {
        //do nothing. All customization is done at the static section
    }

    private static void exportTheRxmicroPackageToReflectionModule() {
        final Module currentModule = $$EnvironmentCustomizer.class.getModule();
        currentModule.addExports("rxmicro.MODULE_NAME", RX_MICRO_REFLECTION_MODULE);
④
    }

    private $$EnvironmentCustomizer() {
    }
}
```

① All customizers are **always** generated in the special package: `rxmicro.MODULE_NAME`, where `MODULE_NAME` is the module name or `unnamed` constant if the current project does not declare the `module-info.java` descriptor (Read more: `Unnamed Modules Support`).

② When the class is loaded, the `exportTheRxmicroPackageToReflectionModule()` static method is invoked.

③ Finds all `$$EnvironmentCustomizer` classes that are defined in other modules in the current module path and invokes static sections.

④ In this method body, the export of the `rxmicro.MODULE_NAME` package to the `rxmicro.reflection` module is performed dynamically using the capabilities of the `java.lang.Module` class.

Due to this additional class, all necessary settings for the `JPMS` are created automatically.

If the RxMicro framework needs additional automatic settings for its correct work, these settings will be automatically added by the `RxMicro Annotation Processor` to the `rxmicro.MODULE_NAME.$$EnvironmentCustomizer` class.

### 4.1.3. Using the Debug Mode

To get a better idea of how the RxMicro framework works, You can use the debug mode.

To do this, set breakpoints and start the microservice in debug mode.

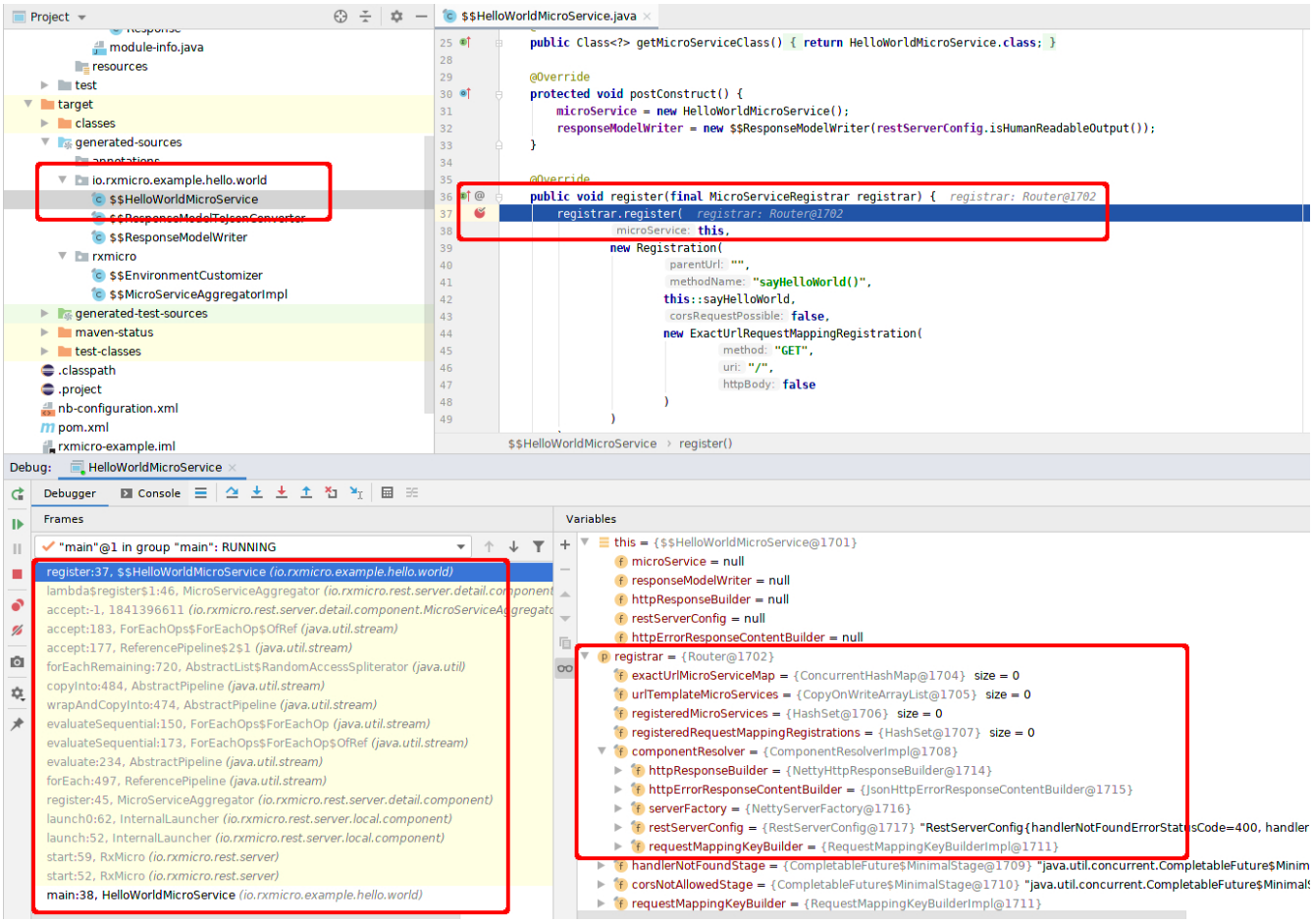



*Figure 10. Starting the microservice in debug mode.*

The code generated by the `RxMicro Annotation Processor` is a usual source code. So You can use breakpoints for this code as well as for Your source code.

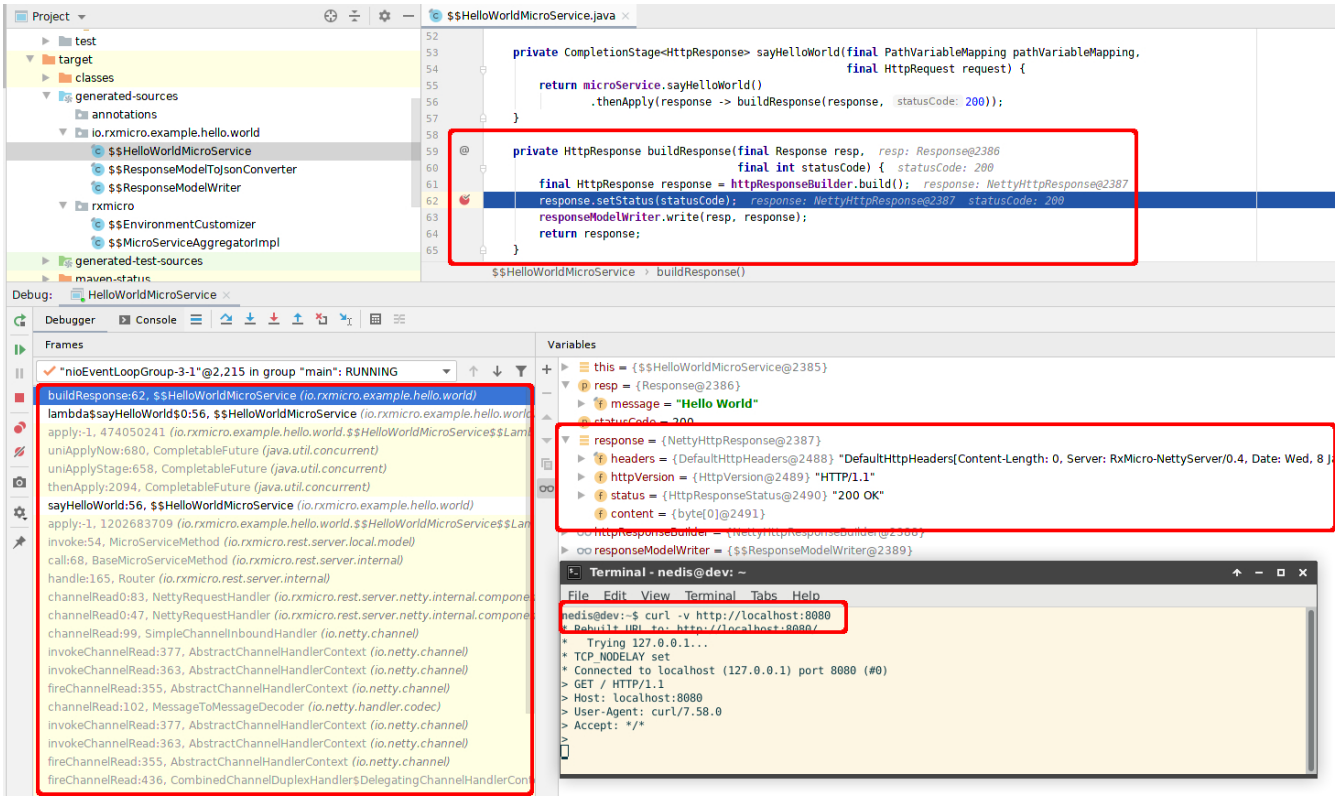



*Figure 11. HTTP request handling by microservice in debug mode.*

## 4.2. `RxMicro Annotation Processor` Options

The `RxMicro Annotation Processor` supports the following options:

*Table 1. Options supported by the `RxMicro Annotation Processor`.*

| Option | Description | Type | Default value |
|---|---|---|---|
| `RX_MICRO_MAX_JSON_NESTED_DEPTH` | maximum stack size for recursive invocations when analyzing models containing JSON nested objects. | `positive int` | `20` |
| `RX_MICRO_LOG_LEVEL` | `RxMicro Annotation Processor` logging level. | Enum {`OFF`, `INFO`, `DEBUG`} | `INFO` |
| `RX_MICRO_DOC_DESTINATION_DIR` | the resulting directory for generated documentation. | `String` | Asciidoc: `./src/main/asciidoc` |
| `RX_MICRO_BUILD_UNNAMED_MODULE` | the `unnamed module` support for a microservice project. | `boolean` | `false` |
| `RX_MICRO_DOC_ANALYZE_PARENT_POM` | this option allows analyzing parent `pom.xml` if child `pom.xml` does not contain required property. | `boolean` | `true` |
| `RX_MICRO_STRICT_MODE` | activates additional validation rules during compilation process.<br><br>**The RxMicro team strong recommends enabling the strict mode for your production code.** | `boolean` | `false` |

These options are set using the compiler arguments in `maven-compiler-plugin`:

```
<plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>${maven-compiler-plugin.version}</version>
    <configuration>
        <release>11</release>
        <compilerArgs>
            <arg>-ARX_MICRO_MAX_JSON_NESTED_DEPTH=20</arg>
            <arg>-ARX_MICRO_LOG_LEVEL=INFO</arg>
            <arg>-ARX_MICRO_DOC_DESTINATION_DIR=./src/main/asciidoc</arg>
            <arg>-ARX_MICRO_BUILD_UNNAMED_MODULE=false</arg>
            <arg>-ARX_MICRO_DOC_ANALYZE_PARENT_POM=true</arg>
            <arg>-ARX_MICRO_STRICT_MODE=false</arg>
        </compilerArgs>
    </configuration>
</plugin>
```

Note that it is necessary to add the `-A` prefix before setting the value of the option.

The common format is as follows: `-A${name}=${value}`. For example: `-ARX_MICRO_LOG_LEVEL=OFF`

# 4.3. Don’t Block Current Thread!

In modern computer architecture, IO operations are the slowest ones. As a result, when using multithreading programming model, the use of CPU and RAM is extremely inefficient. For a single-user monolithic application such inefficiency is imperceptible. But for a multi-user distributed application with a much higher number of IO operations, this problem generates huge financial costs for additional hardware and coordination between client data streams (Read more: C10k problem).

Therefore, the Event-driven architecture (EDA) is used for efficient use of the hardware resources of a multi-user distributed application.

The most popular Java framework that uses Event-driven architecture for IO operations is Netty. To write efficient programs using Netty, it is necessary to comply with certain rules and restrictions. The most important of these is the following requirement: **Don’t block current thread!**

The RxMicro framework is a framework that runs on Netty. Therefore, all requirements for applications that utilize Netty also cover the RxMicro framework.

## 4.3.1. Prohibited Operations

Consequently, when writing microservice applications based on the RxMicro framework, the following operations are **prohibited**:

- Data reading from a socket or file in a blocking style using `java.io.InputStream` and child classes.
- Data writing to a socket or file in a blocking style using `java.io.OutputStream` and child classes.
- Interaction with a database using of the blocking driver (all `JDBC` drivers).
- Waiting on a lock or a monitor (`java.util.concurrent.locks.Lock`, `Object.wait`).
- Putting the thread into sleep mode (`Thread.sleep`, `TimeUnit.sleep`).
- Any other **blocking operations**.

## 4.3.2. Recommended Approach

The absence of blocking operations in the microservice allows handling many concurrent connections, using a small number of threads and, as a result, to effectively use hardware resources of the computer.

Therefore, when designing microservices via the RxMicro framework, You must follow by the following rule:

**When implementing a microservice, if the result can be obtained immediately, it must be returned immediately.**

**Otherwise, You must return Publisher or CompletableFuture, which will generate the result later.**

# 4.4. Reactive Libraries Support

The RxMicro framework supports the following reactive programming libraries:

- **java.util.concurrent**: CompletableFuture, CompletionStage;
- **Project Reactor**: Mono, Flux;
- **RxJava**: Completable, Single, Maybe, Flowable;

## 4.4.1. Expected Business Process Results

When writing reactive programs, the following 4 expected results of any business process are possible:

1. It is important to complete the business process, but the result is missing or unimportant.
2. The business process returns the result in a single instance or nothing.
3. The business process returns the required result in a single instance.
4. The business process returns the result as `(0 .. n)` object stream.

When writing a business service using reactive libraries, it is recommended to comply with the following agreements:

*Table 2. Which class from a reactive library must be choose?*

| Reactive Library | java.util.concurrent | Project Reactor | RxJava3 |
|---|---|---|---|
| **Without result** | CompletableFuture<Void><br>CompletionStage<Void> | Mono<Void> | Completable |
| **Optional result** | CompletableFuture<Optional<MODEL>><br>CompletionStage<Optional<MODEL>> | Mono<MODEL> | Maybe<MODEL> |
| **Required result** | CompletableFuture<MODEL><br>CompletionStage<MODEL> | Mono<MODEL> | Single<MODEL> |
| **Stream result** | CompletableFuture<List<MODEL>><br>CompletionStage<List<MODEL>> | Flux<MODEL>,<br>Mono<List<MODEL>> | Flowable<MODEL>,<br>Single<List<MODEL>> |

**The following types of results `Flux<MODEL>` and `Mono<List<MODEL>>`, as well as `Flowable<MODEL>`, `Single<List<MODEL>>` are not absolutely equivalent!**

For the `Flux<MODEL>` and `Flowable<MODEL>` types, the result handler can be invoked before all data is received from a data source, e.g. from a database.

Whereas for `Mono<List<MODEL>>` and `Single<List<MODEL>>` the result handler is invoked **only** after **all** the data is received from the data source!

### 4.4.2. Recommendations for Choosing a Library

General recommendation for choosing a reactive programming library when using the RxMicro framework:

1. If Your microservice contains simple logic, You can use the lightweight and Java-provided **java.util.concurrent** library, represented by the `CompletableFuture` class and the `CompletionStage` interface.
2. If Your microservice contains more complex logic, to describe which You need to use complex operators, it is recommended to choose **Project Reactor** or **RxJava**.
3. When choosing between the **Project Reactor** and **RxJava** follow the recommendations:
    a. If You are more familiar with the **Project Reactor**, then use it, otherwise use **RxJava**.
    b. If You need `r2dbc` based reactive SQL repositories (`rxmicro.data.sql.r2dbc` module), then use the **Project Reactor**.
    *(Since `r2dbc` drivers already use the **Project Reactor**.)*

Thus, when writing microservices via the RxMicro framework, You can use any Java reactive programming library that You prefer!

**FYI** All libraries support a blocking getting of the result:

```
public final class BlockingGetResult {

    public static void main(final String[] args) {
        final String blockingResult1 =
                CompletableFuture.completedFuture("Hello")
                        .join();

        final String blockingResult2 =
                Mono.just("Hello")
                        .block();

        final String blockingResult3 =
                Single.just("Hello")
                        .blockingGet();

        System.out.println(blockingResult1);
        System.out.println(blockingResult2);
        System.out.println(blockingResult3);
    }

}
```

The main need in blocking getting of the result, using reactive programming, arises at Unit testing implementation. Since the main popular frameworks for unit testing (`JUnit`, `TestNG`, etc.) use the usual thread (blocking) programming model.

## 4.5. Base Model

Java applications use `java.lang.Object.toString()` method very often for debug purposes. Thus a developer must override this method for all model classes in his(her) project.

To help with overriding of this method for all model classes the RxMicro framework introduces the `BaseModel` class. This class uses the `reflection` to generate string representation of the model class on fly.

```
public class CustomModel extends BaseModel {

    String string;

    Integer integer;

    //...

    //toString method not necessary!
}
```

The `reflection` mechanism is slow one, so use the generated string representation of the model class only for debug purposes!

According to `JPMS` requirements all `reflection` access must be configured at the `module-info.java` descriptor using `exports` or opens` instructions. It means that for correct usage the `BaseModel` feature, it is necessary to add the following instruction:

```
opens io.rxmicro.examples.base.model.model.package4 to
        rxmicro.reflection;
```

where `io.rxmicro.examples.base.model.model.package4` is the package that contains custom model classes.

But the `BaseModel` feature is designed for debug purposes, so required `exports` or `opens` instructions are added automatically via generated `$$EnvironmentCustomizer` class if You do not add these instructions manually! So You can use the `BaseModel` feature without any `module-info.java` modifications for Your project!

# 4.6. Strings Formatting

While developing a software product it is necessary to format strings.

For this purpose, Java provides different approaches:

```
Mono<? extends Result> executeQuery(final Connection connection,
                                    final Long id) {
    final String sql = "SELECT * FROM account WHERE id = $1"; ①
    SLF4J_LOGGER.info("SQL: {}", sql); ②
    return Mono.from(connection.createStatement(sql)
            .bind(0, id)
            .execute())
            .onErrorResume(e -> Mono.error(
                    new IllegalArgumentException(
                            String.format(
                                    "SQL '%s' contains syntax error: %s", sql,
e.getMessage() ③
                            ))
                    )
            );
}
```

① To generate an SQL query, You need to use `$1` placeholder.
*(This placeholder depends on the used R2DBC driver. For postgresql, it's a `$1` symbol.)*

② To generate a logging message, You need to use `{}` placeholder.
*(This placeholder depends on the logging library used. For SLF4J, it's a `{}` symbol.)*

③ To generate an error message, You need to use `%s` placeholder from a separate utility class, for example `String.format`.

While writing the code, a developer can easily confuse the required placeholder.

**To avoid such a problem, the RxMicro framework recommends using the universal `?` placeholder**

```
Mono<? extends Result> executeQuery(final Connection connection,
                                    final Long id) {
    final String sql = "SELECT * FROM account WHERE id = ?"; ①
    RX_MICRO_LOGGER.info("SQL: ?", sql); ②
    return Mono.from(connection.createStatement(sql)
            .bind(0, id)
            .execute())
            .onErrorResume(e -> Mono.error(
                    new InvalidStateException(
                            "SQL '?' contains syntax error: ?", sql, e.getMessage())
③
                    )
            );
}
```

① To generate an SQL query, You need to use ? placeholder.

② To generate a logging message, You need to use ? placeholder.

③ To generate an error message, You need to use ? placeholder.

# 4.7. Encapsulation

When designing Java request and response models, there is a need to protect data from unauthorized modification.

### 4.7.1. A `private` Modifier Usage

The standard solution to this problem in Java is using the `private` modifier:

```
final class Response {

    private final String message; ①

    Response(final String message) {
        this.message = requireNonNull(message);
    }
}
```

① By declaring the `message` field as `private`, the developer allows access to this field only from inside the class.

To violate encapsulation principles when necessary, Java provides powerful `reflection` mechanism.

The RxMicro framework is aware of this mechanism, so when generating a converter, the framework uses it:

```
import static rxmicro.$$Reflections.getFieldValue; ①

public final class $$ResponseModelToJsonConverter extends
ModelToJsonConverter<Response> {

    @Override
    public Map<String, Object> toJsonObject(final Response model) {
        final JsonObjectBuilder builder = new JsonObjectBuilder();
        putValuesToBuilder(model, builder);
        return builder.build();
    }

    public void putValuesToBuilder(final Response model,
                                   final JsonObjectBuilder builder) {
        builder.put("message", getFieldValue(model, "message")); ②
    }
}
```

① Static import of method that allows reading field value with `reflection`.

② Reading the value of the `message` field from the response model.

Using `reflection` when converting from Java model to JSON model while processing each request

can reduce microservice performance, where this problem can be avoided. Therefore, when compiling this class, the `RxMicro Annotation Processor` generates a warning message:

```
[WARNING] Response.java:[27,26] PERFORMANCE WARNING: To read a value from
io.rxmicro.example.hello.world.Response.message rxmicro will use the reflection.
It is recommended to add a getter or change the field modifier: from private to
default, protected or public.
```

If the `RX_MICRO_STRICT_MODE` is set, the RxMicro Annotation Processor throws a compilation error instead of showing the **PERFORMANCE WARNING**.

**By default the reflection usage for model classes is not allowed for strict mode!**

### 4.7.2. A Separate Package Usage

The best and **recommended** solution to this problem is to create a separate package (e.g. `model`) and declare all fields of the model classes without any access modifier (i.e. `default/package`). Under this approach, fields can be accessed **only** from classes of the same package. And the package contains **only** classes of models without any business logic:

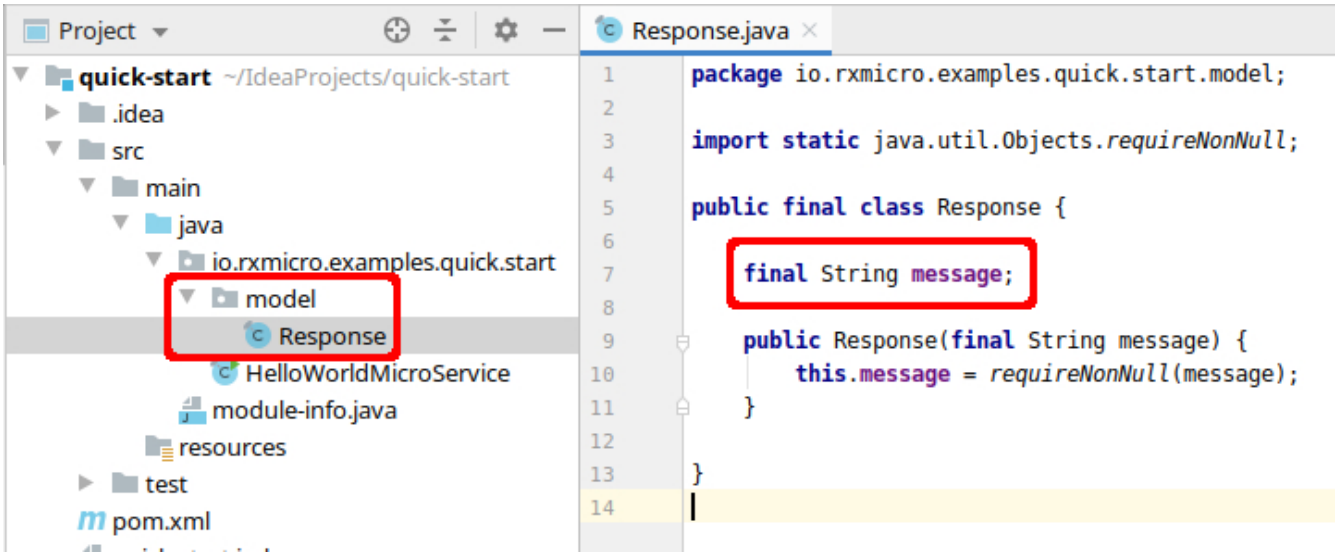



*Figure 12. Recommended structure to support encapsulation in Java models.*

**Using model class fields without any access modifier (i.e. `default/package`) allows You to generate a converter that can read or write a value using direct access to the field by `.` operator.**

### 4.7.3. A `getters` Usage

If the simplest logic is required when reading a value from a model field, You can use `getter`.

To do this, declare the field as `private` and add `getter`:

```
public final class Response {

    private final String message;

    public Response(final String message) {
        this.message = requireNonNull(message);
    }

    public String getMessage() {
        if (message.isEmpty()) {
            return "<Empty>";
        } else {
            return message;
        }
    }
}
```

In this case, `getter` will be used in the generated converter to get the result:

```
public final class $$ResponseModelToJsonConverter extends
ModelToJsonConverter<Response> {

    @Override
    public Map<String, Object> toJsonObject(final Response model) {
        final JsonObjectBuilder builder = new JsonObjectBuilder();
        putValuesToBuilder(model, builder);
        return builder.build();
    }

    public void putValuesToBuilder(final Response model,
                                   final JsonObjectBuilder builder) {
        builder.put("message", model.getMessage()); ①
    }
}
```

① `getter` invoking to get the value of the response model field.

### 4.7.4. Performance Comparison

Performance test source code:

```
@Warmup(iterations = 5)
@Measurement(iterations = 5)
@Fork(value = 2, jvmArgsAppend = "-server")
@BenchmarkMode({
        Mode.Throughput,
        Mode.AverageTime,
        Mode.SingleShotTime
```

```
})
@State(Scope.Benchmark)
@OutputTimeUnit(MILLISECONDS)
@Threads(1)
public class ReadWriteFieldBenchmark {

    private final CustomClass customClass = new CustomClass("text");

    private final Map<Class<?>, Map<String, Field>> cache = new HashMap<>();

    public ReadWriteFieldBenchmark() {
        try {
            final Field field = CustomClass.class.getDeclaredField("value");
            if (!field.canAccess(customClass)) {
                field.setAccessible(true);
            }
            cache.put(CustomClass.class, new HashMap<>(Map.of("value", field)));
        } catch (final NoSuchFieldException e) {
            throw new RuntimeException(e);
        }
    }

    @Benchmark
    public void readDirectField() {
        var v = customClass.value;
    }

    @Benchmark
    public void writeDirectField() {
        customClass.value = "string";
    }

    @Benchmark
    public void readUsingGetter() {
        var v = customClass.getValue();
    }

    @Benchmark
    public void writeUsingSetter() {
        customClass.setValue("string");
    }

    @Benchmark // read field value using reflection with field search at the cache
    public void readUsingReflection() throws IllegalAccessException {
        var v = cache.get(CustomClass.class).get("value").get(customClass);
    }

    @Benchmark // write field value using reflection with field search at the cache
    public void writeUsingReflection() throws IllegalAccessException {
        cache.get(CustomClass.class).get("value").set(customClass, "string");
    }
```

```
}
```

Performance test results:

```
Benchmark                                        Mode  Cnt       Score       Error
Units
ReadWriteFieldBenchmark.readDirectField         thrpt   10  753348.188 ± 90286.947
ops/ms
ReadWriteFieldBenchmark.readUsingGetter         thrpt   10  764112.155 ± 94525.371
ops/ms
ReadWriteFieldBenchmark.readUsingReflection     thrpt   10   26241.478 ±  3838.172
ops/ms
ReadWriteFieldBenchmark.writeDirectField        thrpt   10  344623.904 ± 18961.759
ops/ms
ReadWriteFieldBenchmark.writeUsingReflection    thrpt   10   19430.735 ±  2135.813
ops/ms
ReadWriteFieldBenchmark.writeUsingSetter        thrpt   10  323596.205 ± 28416.707
ops/ms
ReadWriteFieldBenchmark.readDirectField          avgt   10       □ 10□□
ms/op
ReadWriteFieldBenchmark.readUsingGetter          avgt   10       □ 10□□
ms/op
ReadWriteFieldBenchmark.readUsingReflection      avgt   10       □ 10□□
ms/op
ReadWriteFieldBenchmark.writeDirectField         avgt   10       □ 10□□
ms/op
ReadWriteFieldBenchmark.writeUsingReflection     avgt   10       □ 10□□
ms/op
ReadWriteFieldBenchmark.writeUsingSetter         avgt   10       □ 10□□
ms/op
ReadWriteFieldBenchmark.readDirectField            ss   10        0.001 ±     0.001
ms/op
ReadWriteFieldBenchmark.readUsingGetter            ss   10        0.001 ±     0.001
ms/op
ReadWriteFieldBenchmark.readUsingReflection        ss   10        0.008 ±     0.005
ms/op
ReadWriteFieldBenchmark.writeDirectField           ss   10        0.002 ±     0.001
ms/op
ReadWriteFieldBenchmark.writeUsingReflection       ss   10        0.011 ±     0.008
ms/op
ReadWriteFieldBenchmark.writeUsingSetter           ss   10        0.001 ±     0.001
ms/op
```

Test results show that reading/writing by a direct reference or via `getter`/`setter` is performed **28/17 times faster**, than when using `reflection`!

### 4.7.5. Approach Selection Recommendations

Thus, the RxMicro framework uses the following algorithm to read (write) from the fields of the Java model:

1. If the field is declared with `public`, `protected` or `default/package` modifiers, the generated converter uses direct access to the field using the `.` operator;
2. If the field is declared with the `private` modifier and the developer created `getter`/`setter`, the generated converter uses the `getter`/`setter` invocation to get or write the field value;
3. If the field is declared with the `private` modifier without `getter`/`setter` declaration, the generated converter uses `reflection` to access the model field. *(When generating this type of converter the `RxMicro Annotation Processor` informs the developer about `PERFORMANCE WARNING`.)*

To benefit from the encapsulation advantages when designing microservices via the RxMicro framework, follow the recommendations:

**When creating request and response models, use a separate package and `default/package` access modifier!**

**If the simplest logic of reading (writing) the value of the model field is required, use `getter` (`setter`) and `private` field access modifier.**

**Do not use the `private` modifier to access the model field without `getter` (`setter`)!**
**This approach offers no benefits!**

If the `RX_MICRO_STRICT_MODE` is set, the RxMicro Annotation Processor throws a compilation error instead of showing the **PERFORMANCE WARNING**.

**By default the reflection usage for model classes is not allowed for strict mode!**

# 4.8. Configuration

The RxMicro framework provides the `rxmicro.config` module for flexible configuration of microservices to any environment. This module provides the following features:

- Support for different types of configuration sources: files, classpath resources, environment variables, command line arguments, etc.;
- Inheritance and redefinition of settings from different configuration sources;
- Changing the order in which the configuration sources are read;
- Configuration using annotations and Java classes.

### 4.8.1. Basic Structure of the Config Module

Each class that extends the `Config` abstract class is a configuration class.

Each configuration class defines its own namespace. Each namespace clearly defines the corresponding configuration class. The namespace is necessary to set the settings of the configuration class fields in text form. (*Further details will be described below.*)

To work with configurations, the RxMicro framework provides the `Configs` configuration manager.

To read the current settings, You must use the `Configs.getConfig(Class<T>)` method:

```
public final class ReadConfigMicroService {

    @GET("/")
    void readHttpServerPort() {
        final HttpServerConfig config = getConfig(HttpServerConfig.class); ①
        System.out.println("HTTP server port: " + config.getPort());
    }

    public static void main(final String[] args) {
        startRestServer(ReadConfigMicroService.class);
    }
}
```

① Getting the current HTTP server configuration using the `Configs.getConfig` static method.

To change the standard configuration settings, You must use the `Configs.Builder` class:

```
public final class CustomizeConfigMicroService {

    @GET("/")
    void test() {
        // do something
    }

    public static void main(final String[] args) {
        new Configs.Builder()
                .withConfigs(new HttpServerConfig()
                        .setPort(9090)) ①
                .build(); ②
        startRestServer(CustomizeConfigMicroService.class); ③
    }
}
```

① Setting the HTTP server custom port.

② Creating the configuration manager object.

③ REST-based microservice should be started after configuration manager settings, otherwise changes will not take effect.

Each subsequent invocation of the `Configs.Builder.build()` method overrides all configuration manager settings. *(In any microservice project there is only one configuration manager object!)*

It means that if the developer creates several `Configs.Builder` instances, it will be the last invocation of the `build` method that matters, the others will be ignored.

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-common/config-basic`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

Settings customization via the `Configs.Builder.build()` is one of the types of configuration. This type of configuration is called configuration using Java classes.

### 4.8.2. Configuration Types

The RxMicro framework supports the following configuration types:

1. Configuration using `classpath` resources.
2. Configuration using `properties` files.
3. Configuration using environment variables.
4. Configuration using Java system properties.
5. Configuration using Java classes.
6. Configuration using Java annotations.
7. Configuration using command line arguments.

#### 4.8.2.1. Configuration Using `classpath` Resources.

The RxMicro framework supports shared and separate classpath resources for the external configuration in relation to the microservice source code.

The only supported classpath resource format is the `properties` file format.

##### 4.8.2.1.1. Configuration Using Separate `classpath` Resource.

If the classpath of the current project contains the `http-server.properties` resource with the following content:

```
port=9090 ①
```

① Custom port for HTTP server.

then the RxMicro framework when reading the `HttpServerConfig` class configuration will read this resource:

```
2020-01-11 12:44:27.518 [INFO]
io.rxmicro.rest.server.netty.internal.component.NettyServer :
Server started at 0.0.0.0:9090 using NETTY transport in 500 millis. ①
```

① The HTTP server has started at the `9090` port instead of the standard `8080` port.

The `http-server` name is the default namespace for the `HttpServerConfig` class. That's why these settings are automatically applied when requesting the `HttpServerConfig` class configuration.

The default namespace for the configuration class is calculated using the `Config.getDefaultNameSpace(Class<? extends Config>)` method,

#### 4.8.2.1.2. Configuration Using Shared `classpath` Resource.

If the classpath of the current project contains the `rxmicro.properties` resource with the following content:

```
http-server.port=9090 ①
```

① Custom port for HTTP server.

then the RxMicro framework when reading the `HttpServerConfig` class configuration will read this resource:

```
2020-01-11 12:44:27.518 [INFO]
io.rxmicro.rest.server.netty.internal.component.NettyServer :
Server started at 0.0.0.0:9090 using NETTY transport in 500 millis. ①
```

① The HTTP server has started at the `9090` port instead of the standard `8080` port.

The `rxmicro` name for the `properties` file is constant. That's why when requesting any configuration, the RxMicro framework tries to read the content of this file, if it exists.

The `rxmicro` name is a named constant: `Config.RX_MICRO_CONFIG_FILE_NAME`,

Since the `rxmicro.properties` resource is a shared resource for **any** configuration, You must specify a namespace when specifying the settings.

That's why when specifying the HTTP server port, You should specify the `http-server` prefix in the `rxmicro.properties` file. *(When using the `http-server.properties` file, there was no such need!)*

That means

```
http-server.port=9090
```

instead of

```
port=9090
```

#### 4.8.2.2. Configuration Using `properties` Files.

Similar to classpath resources, the RxMicro framework also supports shared and separate `properties` files for the external configuration in relation to the microservice source code.

Configuration files can be located:

- in the current directory in relation to the microservice;

- in the `$HOME` directory:
  - for Linux platform the `$HOME` directory is `/home/$USERNAME`;
  - for MacOS platform the `$HOME` directory is `/Users/$USERNAME`;
  - for Windows platform the `$HOME` directory is `C:\Documents and Settings\%USERNAME%` or `C:\Users\%USERNAME%`.
- in the default rxmicro config directory: `$HOME/.rxmicro/` (predefined name and location).
  *(Using `$HOME/.rxmicro/` directory instead of `$HOME` one allows configuring this directory as docker or kubernetes volume.)*

To find out the location of the `$HOME` directory on Your computer using Java, start the following program:

```
public final class GetHomeDirectory {

    public static void main(final String[] args) {
        System.out.println(System.getProperty("user.home"));
    }
}
```

**By default, the function of searching and reading configuration files is disabled in the RxMicro framework!**

To activate this function, You must use the `Configs.Builder` class:

```
new Configs.Builder()
        .withAllConfigSources() ①
        .build();
```

① Activation of all available configuration sources for the current microservice.

Besides activating all available sources, it is possible to activate only configuration files in a given location.

For details on how to do this, go to the Section 4.8.3.2, “Custom Reading Order of Config Sources.”.

#### 4.8.2.2.1. Configuration Using Separate `properties` File.

If the current directory (or `$HOME`, or `$HOME/.rxmicro/`) directory contains the `http-server.properties` file with the following content

```
port=9090 ①
```

① Custom port for HTTP server.

then the RxMicro framework when reading the `HttpServerConfig` class configuration will read this file:

```
2020-01-11 12:44:27.518 [INFO]
io.rxmicro.rest.server.netty.internal.component.NettyServer :
Server started at 0.0.0.0:9090 using NETTY transport in 500 millis. ①
```

① The HTTP server has started at the `9090` port instead of the standard `8080` port.

The `http-server` name is the default namespace for the `HttpServerConfig` class. That's why these settings are automatically applied when requesting the `HttpServerConfig` class configuration.

The default namespace for the configuration class is calculated using the `Config.getDefaultNameSpace(Class<? extends Config>)` method,

#### 4.8.2.2.2. Configuration Using Shared `properties` File.

If the current directory (or `$HOME`, or `$HOME/.rxmicro/`) directory contains the `rxmicro.properties` file with the following content

```
http-server.port=9090 ①
```

① Custom port for HTTP server.

then the RxMicro framework when reading the `HttpServerConfig` class configuration will read this resource:

```
2020-01-11 12:44:27.518 [INFO]
io.rxmicro.rest.server.netty.internal.component.NettyServer :
Server started at 0.0.0.0:9090 using NETTY transport in 500 millis. ①
```

① The HTTP server has started at the `9090` port instead of the standard `8080` port.

The `rxmicro` name for the `properties` file is constant. That's why when requesting any configuration, the RxMicro framework tries to read the content of this file, if it exists.

The `rxmicro` name is a named constant: `Config.RX_MICRO_CONFIG_FILE_NAME`,

Since the `rxmicro.properties` file is a shared file for **any** configuration, You must specify a namespace when specifying the settings.

That's why when specifying the HTTP server port, You should specify the `http-server` prefix in the `rxmicro.properties` file. *(When using the* `http-server.properties` *file, there was no such need!)*

That means

```
http-server.port=9090
```

instead of

```
port=9090
```

#### 4.8.2.3. Configuration Using Environment Variables.

When using environment variables, the format of configurations matches the following format:

`export ${name-space}.${property-name} = ${value}`:

```
export http-server.port=9090 ①

java -p ./classes:lib -m
examples.quick.start/io.rxmicro.examples.quick.start.HelloWorldMicroService
2020-01-02 18:49:58.372 [INFO]
io.rxmicro.rest.server.netty.internal.component.NettyServer :
Server started at 0.0.0.0:9090 using NETTY transport in 500 millis. ②
```

① Setting the `http-server.port` environment variable = 9090 (custom port for HTTP server).

② The HTTP server has started at the `9090` port instead of the standard `8080` port.

Thus, the format of configurations using environment variables corresponds to the format of `rxmicro.properties` file or classpath resource.

**Allowed characters in environment variable names!**

From The Open Group:

*These strings have the form name=value; names shall not contain the character '='. For values to be portable across systems conforming to IEEE Std 1003.1-2001, the value shall be composed of characters from the portable character set (except NUL and as indicated below).*

So names may contain any character except `=` and `NUL`, but:

*Environment variable names used by the utilities in the Shell and Utilities volume of IEEE Std 1003.1-2001 consist solely of* ***uppercase letters, digits, and the '_' (underscore)*** from the characters defined in Portable Character Set and do not begin with a digit. Other characters may be permitted by an implementation; applications shall tolerate the presence of such names._

So for such restricted environment the RxMicro framework supports the following format for environment variable mapping as well:

```
export HTTP_SERVER_PORT=9090 ①

java -p ./classes:lib -m
examples.quick.start/io.rxmicro.examples.quick.start.HelloWorldMicroSer
vice
2020-01-02 18:49:58.372 [INFO]
io.rxmicro.rest.server.netty.internal.component.NettyServer :
Server started at 0.0.0.0:9090 using NETTY transport in 500 millis. ②
```

① Setting the `HTTP_SERVER_PORT` environment variable = 9090 (custom port for HTTP server).

② The HTTP server has started at the `9090` port instead of the standard `8080` port.

**Configuring with environment variables is very convenient when using docker containers!**

To protect Your secret data, use configuration via properties files instead of environment variables. The config directory with secret config files (for example `$HOME/.rxmicro/`) must be mount as external volume using `tmpfs` file system.

If it is necessary to separate environment variables used for the configuration of the RxMicro environment from other environment variables, You must define the standard environment variable with name `RX_MICRO_CONFIG_ENVIRONMENT_VARIABLE_PREFIX`!

For example if You runtime contains the following environment variable `RX_MICRO_CONFIG_ENVIRONMENT_VARIABLE_PREFIX=MY_RUNTIME_` than it is necessary to use

```
export MY_RUNTIME_HTTP_SERVER_PORT=9090
```

instead of

```
export HTTP_SERVER_PORT=9090
```

setting for configuring a port for the HTTP server!

#### 4.8.2.4. Configuration Using Java System Properties

When using the `Java System Properties`, the format of configurations matches the following format:

`${name-space}.${property-name} = ${value}`:

```
java -p ./classes:lib \
    -Dhttp-server.port=9090 \ ①
    -m examples.quick.start/io.rxmicro.examples.quick.start.HelloWorldMicroService

2020-01-02 18:49:58.372 [INFO]
io.rxmicro.rest.server.netty.internal.component.NettyServer :
Server started at 0.0.0.0:9090 using NETTY transport in 500 millis. ②
```

① Setting the `http-server.port` Java system variable = 9090 (custom port for HTTP server).

② The HTTP server has started at the `9090` port instead of the standard `8080` port.

Thus, the format of configurations using Java system variables corresponds to the format of configuration using environment variables, and also to the format of `rxmicro.properties` file or classpath resource.

#### 4.8.2.5. Configuration Using Java Classes.

Configuring with Java classes is the easiest and most explicit configuration method:

```
public static void main(final String[] args) {
    new Configs.Builder()
            .withConfigs(new HttpServerConfig()
                    .setPort(9090)) ①
            .build();
    startRestServer(MicroService.class);
}
```

① Changing the HTTP server port.

```
2020-01-02 18:49:58.372 [INFO]
io.rxmicro.rest.server.netty.internal.component.NettyServer :
Server started at 0.0.0.0:9090 using NETTY transport in 500 millis. ①
```

① The HTTP server has started at the `9090` port instead of the standard `8080` port.

**The main difference between this type of configuration and the others is that when using Java classes, other configuration sources are always ignored!**

**Therefore this type is recommended to be used ONLY for test purposes!**

(*It does not have enough flexibility for the production environment!*)

**For the production environment, use the configuration with annotations instead of the configuration with Java classes!**

#### 4.8.2.6. Configuration Using Java Annotations.

To override the default value, the RxMicro framework provides the `@DefaultConfigValue` annotation.

```
@DefaultConfigValue(
        configClass = HttpServerConfig.class,   ①
        name = "port",                          ②
        value = "9090"
)
@DefaultConfigValue(
        name = "http-server.host",  ③
        value = "localhost"
)
module examples.config.annotations { ④
    requires rxmicro.rest.server.netty;
    requires rxmicro.rest.server.exchange.json;
}
```

① When overriding the configuration value, You must specify the configuration class.

② If the configuration class is specified, the namespace may not be specified.
*(It means the field of the specified configuration class.)*

③ If no configuration class is specified, the name **must** contain a namespace.
*(The namespace allows You to clearly define to which configuration class the specified setting belongs.)*

④ When configuring a microservice project, the annotation must be specified on the `module-info.java` descriptor.
*(A microservice project may contain several classes of REST controllers, so the common settings are configured using the `module-info.java` descriptor rather than the REST controller class.)*

**Please, pay attention when overriding the default value with the annotations!**

If You make a mistake when specifying a setting name (this refers to the namespace and field name), no error will occur upon starting! The overridden value will be simply **ignored**!

By overriding the default values using the `module-info.java` descriptor, You can start the microservice. While reading the configuration of the `HttpServerConfig` class, the RxMicro framework reads the default values set with annotations:

```
2020-01-13 13:09:44.236 [INFO]
io.rxmicro.rest.server.netty.internal.component.NettyServer :
Server started at localhost:9090 using NETTY transport in 500 millis. ①
```

① HTTP server started on `localhost:9090`

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-common/config-annotations

The main difference between configuring with annotations and configuring with Java classes is support of the **settings inheritance and overriding**.

In other words, when using configuration via annotations, the RxMicro framework can also read other configuration sources.

When using configuration via Java classes, other configuration sources **are always ignored**.

For the test environment only, the RxMicro framework provides special `@WithConfig` annotation. Using this annotation it is convenient to configure the configuration manager for the test environment:

```
@WithConfig
private static final HttpServerConfig SERVER_CONFIG =
        new HttpServerConfig()
                .setPort(56789);
```

The source code of the project using the `@WithConfig` annotation is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/testing-microservice-with-config

`@DefaultConfigValue` annotation can be applied to override primitive values only:

- `strings`,
- `booleans`,
- `numbers`,
- `dates`,
- `times`,
- `enums`.

If You need to override a complex value, it is necessary to use `@DefaultConfigValueSupplier`

annotation instead.

The source code of the project using the `@DefaultConfigValueSupplier` annotation is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/config-annotations-supplier`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

#### 4.8.2.7. Configuration Using Command Line Arguments.

To override configs You can use command line arguments.

This type of configuration has **the highest priority and overrides all other types**. *(Except of configuration using Java classes.)*

**Configuration using command line arguments is disabled by default.**

To enable it is necessary to configure the `Configs` configuration manager:

```
public static void main(final String[] args) {
    new Configs.Builder()
            .withCommandLineArguments(args) ①
            .build();
    startRestServer(MicroService.class);
}
```

① Use `withCommandLineArguments(args)` method to enable the configuration using command line arguments.

For example, If You want to start HTTP server at `9191` port, You can pass the following command line argument:

```
java -p ./classes:lib -m
examples.quick.start/io.rxmicro.examples.quick.start.HelloWorldMicroService http-server.port=9191
```

Result:

```
2020-01-02 18:49:58.372 [INFO]
io.rxmicro.rest.server.netty.internal.component.NettyServer :
Server started at 0.0.0.0: 9191 using NETTY transport in 500 millis.
```

### 4.8.3. Config Sources Setting

The RxMicro framework allows You to customize the order in which the configuration sources are read.

With this feature, the RxMicro framework automatically supports inheritance and redefinition of launch configurations.

#### 4.8.3.1. Default Reading Order of Config Sources.

By default, the RxMicro framework reads configuration sources in the following order:

- Configuration using `@DefaultConfigValue` annotations;
- Configuration using the `rxmicro.properties` classpath resource;
- Configuration using the separate (`${name-space}.properties`) classpath resource;
- Configuration using environment variables;
- Configuration using Java system variables;

Thus, if there are two classpath resources for a microservice:

The `rxmicro.properties` resource with the following content:

```
http-server.port=9090
http-server.host=localhost
```

and the `http-server.properties` resource with the following content:

```
port=9876
```

then the result will be as follows:

```
2020-01-11 16:52:26.797 [INFO]
io.rxmicro.rest.server.netty.internal.component.NettyServer :
Server started at localhost:9876 using NETTY transport in 500 millis. ①
```

① HTTP server has started at `localhost:9876`.

The configuration reading algorithm for the above example is as follows:

1. By default, the HTTP server should start at `0.0.0.0:8080`.
2. But in the `rxmicro.properties` classpath resource there is a different IP address and port: `localhost:9090`.
3. If the `http-server.properties` classpath resource had not existed, the HTTP server would have run at `localhost:9090`.
4. But in the `http-server.properties` classpath resource it is specified the `9876` port.

5. Therefore, when starting, the IP address is inherited from the `rxmicro.properties` resource and the overridden port value is read from the `http-server.properties` resource.
*(This behavior corresponds to the order of reading the default configurations.)*

#### 4.8.3.2. Custom Reading Order of Config Sources.

To change the order of the configuration reading it is necessary to use the `Configs.Builder.withOrderedConfigSources(ConfigSource…)` method:

```
public static void main(final String[] args) {
    new Configs.Builder()
            .withOrderedConfigSources(
                    SEPARATE_CLASS_PATH_RESOURCE, ①
                    RXMICRO_CLASS_PATH_RESOURCE   ②
            )
            .build();
    startRestServer(MicroService.class);
}
```

① First, it is necessary to read the configuration from the `${name-space}.properties` classpath resource (In our case, it's a `http-server.properties`)

② and then from the `rxmicro.properties` classpath resource.

Thus, the order of the configuration reading from classpath resources has been changed in comparison with the default order.

When starting the microservice, the settings from the `http-server.properties` classpath resource will be overwritten by the settings from the `rxmicro.properties` classpath resource:

```
2020-01-11 16:52:26.797 [INFO]
io.rxmicro.rest.server.netty.internal.component.NettyServer :
Server started at localhost:9090 using NETTY transport in 500 millis. ①
```

① HTTP server has started at `localhost:9090`.

The `Configs.Builder.withOrderedConfigSources(ConfigSource…)` method allows not only to change the order of reading the configuration sources, but also to activate/deactivate the sources.

**In the above example, the RxMicro framework will ignore any configuration sources except classpath resources!**

The `Configs.Builder.withOrderedConfigSources(ConfigSource…)` method is universal.

The RxMicro framework also provides other additional methods:

- `Configs.Builder.withAllConfigSources()` - activation of all configuration types in the order given by the list: `ConfigSource`

- `Configs.Builder.withContainerConfigSources()` - this combination is recommended for microservices operating in docker or kubernetes.

The `Configs.Builder.withAllConfigSources()` method was used to activate the reading of configurations from properties files in the Section 4.8.2.2, "Configuration Using `properties` Files." subsection.

If You plan to use only `properties` files, it is recommended to specify only these types, excluding all other types:

```
public static void main(final String[] args) {
    new Configs.Builder()
            .withOrderedConfigSources(
                    RXMICRO_FILE_AT_THE_HOME_DIR,    ①
                    RXMICRO_FILE_AT_THE_CURRENT_DIR, ②
                    SEPARATE_FILE_AT_THE_HOME_DIR,   ③
                    SEPARATE_FILE_AT_THE_CURRENT_DIR ④
            )
            .build();
    startRestServer(MicroService.class);
}
```

① Activation of configuration reading from the `$HOME/rxmicro.properties` file.

② Activation of configuration reading from the `rxmicro.properties` file in the current directory.

③ Activation of configuration reading from the `$HOME/${name-space}.properties` files (for example, `http-server.properties`).

④ ctivation of configuration reading from the `${name-space}.properties` files (for example, `http-server.properties`) in the current directory.

**The order of reading is set by the argument order of the `Configs.Builder.withOrderedConfigSources(ConfigSource…)` method**

**If You know exactly which configuration sources should be used by the microservice, ALWAYS specify them explicitly!**

**With this approach, at the microservice starting, the RxMicro framework won't try to search for non-existent sources, spending precious microseconds!**

#### 4.8.3.3. Logging the Config Reading Process

To debug the configuration reading process You can activate the logger:

```
.level=INFO
io.rxmicro.config.level=DEBUG ①
```

① For all classes and subpackages of the `io.rxmicro.config` package activate the `DEBUG`(`FINE`) logging

level.

After activating the logger, the start of the microservice with default settings will be as follows:

```
[DEBUG] Discovering properties for 'rest-server' namespace from sources:
[DEFAULT_CONFIG_VALUES, RXMICRO_CLASS_PATH_RESOURCE, SEPARATE_CLASS_PATH_RESOURCE,
ENVIRONMENT_VARIABLES, JAVA_SYSTEM_PROPERTIES]

[DEBUG] Classpath resource not found: rest-server.properties

[DEBUG] All properties discovered for 'rest-server' namespace ①

[DEBUG] Discovering properties for 'http-server' namespace from sources:
[DEFAULT_CONFIG_VALUES, RXMICRO_CLASS_PATH_RESOURCE, SEPARATE_CLASS_PATH_RESOURCE,
ENVIRONMENT_VARIABLES, JAVA_SYSTEM_PROPERTIES]

[DEBUG] Discovered properties from 'rxmicro.properties' classpath resource: [http-
server.port=9090, http-server.host=localhost] ②

[DEBUG] Discovered properties from 'http-server.properties' classpath resource:
[port=9876] ③

[DEBUG] All properties discovered for 'http-server' namespace

[DEBUG] Discovering properties for 'netty-rest-server' namespace from sources:
[DEFAULT_CONFIG_VALUES, RXMICRO_CLASS_PATH_RESOURCE, SEPARATE_CLASS_PATH_RESOURCE,
ENVIRONMENT_VARIABLES, JAVA_SYSTEM_PROPERTIES]

[DEBUG] Classpath resource not found: netty-rest-server.properties

[DEBUG] All properties discovered for 'netty-rest-server' namespace ④

[INFO] Server started at 0.0.0.0:9876 using NETTY transport in 500 millis. ⑤
```

① There is no configuration customization for the `rest-server` namespace, so the default configuration will be used.

② Configuration reading for the `http-server` namespace from the `rxmicro.properties` classpath resource (Read values: `http-server.port=9090`, `http-server.host=localhost`).

③ Configuration reading for the `http-server` namespace from the `http-server.properties` classpath resource (Read value: `port=9876`).

④ There is no configuration customization for the `netty-rest-server` namespace, so the default configuration will be used.

⑤ HTTP server has started at `localhost:9876`.

**Additional debugging information will show the order of reading the configuration sources and overriding the configuration parameter values!**

### 4.8.4. Dynamic Configuration

If Your runtime environment can contain dynamic properties You can use `AsMapConfig` configuration:

```
public final class DynamicAsMapConfig extends AsMapConfig {
}
```

If the `rxmicro.properties` classpath resource with the following content exists:

```
dynamic.bigDecimal=3.1423456676
dynamic.bigInteger=99999999999999999999999999999999999999999999999999
dynamic.boolean=true
dynamic.integer=34
dynamic.string=staging
```

then the following code will return configured dynamic parameters:

```
public static void main(final String[] args) {
    final DynamicAsMapConfig config = getConfig(DynamicAsMapConfig.class);

    System.out.println(config.getBigDecimal("bigDecimal"));
    System.out.println(config.getBigInteger("bigInteger"));
    System.out.println(config.getBoolean("boolean"));
    System.out.println(config.getInteger("integer"));
    System.out.println(config.getString("string"));
}
```

### 4.8.5. User Defined Configurations

The developer can use the configuration module for custom configurations.

To do this, it is necessary to create a separate class:

```
public final class BusinessServiceConfig extends Config {

    private boolean production = true;

    public boolean isProduction() {
        return production;
    }

    public BusinessServiceConfig setProduction(final boolean production) {
        this.production = production;
        return this;
    }
}
```

The custom configuration class must meet the following requirements:

1. The class must be public.
2. The class must contain a public constructor without parameters.
3. The class must extend the `Config` abstract class.
4. To set property values, the class must contain `setters`.
   *(Only those fields that contain `setters` can be initialized!)*

Since this class will be created and initialized by the reflection util classes from `rxmicro.reflection` module automatically, it is necessary to export the package of the custom config class to this module in the `module-info.java` descriptor.
*(These are the `JPMS` requirements.)*

```
module examples.config.custom {
    requires rxmicro.rest.server.netty;
    requires rxmicro.rest.server.exchange.json;

    exports io.rxmicro.examples.config.custom to
            rxmicro.reflection;  ①
}
```

① Allow the access of reflection util classes from `rxmicro.reflection` module to config classes from the `io.rxmicro.example.config.custom` package.

After these changes, a class of custom configurations is available for use:

```
final class MicroService {

    @GET("/")
    void test() {
        final BusinessServiceConfig config = getConfig(BusinessServiceConfig.class);
        System.out.println("Production: " + config.isProduction());
    }
}
```

The `production` flag can now be set using any type of configuration, for example using the classpath of the `business-service.properties` resource:

```
production=false
```

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-common/config-custom`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 4.8.6. Supported Parameter Types

The RxMicro framework supports the primitive, custom and container types, which can be config parameters for any configuration.

#### 4.8.6.1. Supported Primitive Parameter Types

The RxMicro framework supports the following primitive Java types:

- `? extends Enum<?>`;
- `boolean`;
- `java.lang.Boolean`;
- `byte`;
- `java.lang.Byte`;
- `short`;
- `java.lang.Short`;
- `int`;
- `java.lang.Integer`;
- `long`;
- `java.lang.Long`;
- `java.math.BigInteger`;
- `float`;
- `java.lang.Float`;
- `double`;
- `java.lang.Double`;
- `java.math.BigDecimal`;
- `char`;
- `java.lang.Character`;
- `java.lang.CharSequence`;
- `java.lang.String`;
- `java.time.Instant`;
- `java.time.LocalDate`;
- `java.time.LocalDateTime`;
- `java.time.LocalTime`;
- `java.time.MonthDay`;
- `java.time.OffsetDateTime`;
- `java.time.OffsetTime`;
- `java.time.Year`;

- `java.time.YearMonth`;
- `java.time.ZonedDateTime`;
- `java.time.Duration`;
- `java.time.ZoneOffset`;
- `java.time.ZoneId`;
- `java.time.Period`;
- `java.nio.file.Path`;

For temporal classes the RxMicro framework uses `parse` or `of` factory method to converts string value to the appropriate Java representation.

#### 4.8.6.2. Supported Custom Parameter Types

The RxMicro framework supports a custom type as valid config parameter type.

For example if Your project contain the following custom type:

```
public interface CustomType {

    String getValue();
}
```

then Your custom config class can use this type as valid config parameter type:

```
public final class ExampleConfig extends Config {

    private CustomType type = () -> "DEFAULT_CONSTANT";

    public CustomType getType() {
        return type;
    }

    public void setType(final CustomType type) {
        this.type = type;
    }
}
```

Instances of the custom type can be created as:

- Enum constant:

```
public enum CustomEnum implements CustomType {

    ENUM_CONSTANT;

    @Override
    public String getValue() {
        return "ENUM_CONSTANT";
    }
}
```

- Class `public static final` constant:

```
public class CustomClass {

    public static final CustomType CLASS_CONSTANT = () -> "CLASS_CONSTANT";
}
```

- Interface constant:

```
public interface CustomInterface {

    CustomType INTERFACE_CONSTANT = () -> "INTERFACE_CONSTANT";
}
```

- Annotation constant:

```
public @interface CustomAnnotation {

    CustomType ANNOTATION_CONSTANT = () -> "ANNOTATION_CONSTANT";
}
```

To inform the RxMicro framework which instance must be created and injected to config parameter, it is necessary to use the following syntax:

```
@${FULL_CLASS_NAME}:${CONSTANT_FIELD_NAME}
```

For example if Your environment contains the following Java system properties:

```
System.setProperty(
        "enum-constant.type",
        "@io.rxmicro.examples.config.custom.type._enum.CustomEnum:ENUM_CONSTANT"
);
System.setProperty(
        "class-constant.type",
        "@io.rxmicro.examples.config.custom.type._class.CustomClass:CLASS_CONSTANT"
);
System.setProperty(
        "interface-constant.type",

"@io.rxmicro.examples.config.custom.type._interface.CustomInterface:INTERFACE_CONSTANT
"
);
System.setProperty(
        "annotation-constant.type",

"@io.rxmicro.examples.config.custom.type._annotation.CustomAnnotation:ANNOTATION_CONST
ANT"
);
```

and Your application read configuration for all configured namespaces:

```
System.out.println(
        "Default constant: " +
                getConfig(ExampleConfig.class).getType().getValue()
);
System.out.println(
        "Enum constant: " +
                getConfig("enum-constant", ExampleConfig.class).getType().getValue()
);
System.out.println(
        "Class constant: " +
                getConfig("class-constant", ExampleConfig.class).getType().getValue()
);
System.out.println(
        "Interface constant: " +
                getConfig("interface-constant",
ExampleConfig.class).getType().getValue()
);
System.out.println(
        "Annotation constant: " +
                getConfig("annotation-constant",
ExampleConfig.class).getType().getValue()
);
```

Result will be the following:

```
Default constant: DEFAULT_CONSTANT
Enum constant: ENUM_CONSTANT
Class constant: CLASS_CONSTANT
Interface constant: INTERFACE_CONSTANT
Annotation constant: ANNOTATION_CONSTANT
```

For class or interface or annotation constants the RxMicro framework uses the `reflection` to read value.
So before using this type of custom type instances don't forget to export packages to `rxmicro.reflection` module, because this module contains the `reflection` util class which is used to read this value:

```
exports io.rxmicro.examples.config.custom.type._class to
        rxmicro.reflection;
exports io.rxmicro.examples.config.custom.type._interface to
        rxmicro.reflection;
exports io.rxmicro.examples.config.custom.type._annotation to
        rxmicro.reflection;
```

**The RxMicro team recommends using an enum for custom type injection to config instances!**

#### 4.8.6.3. Supported Container Parameter Types

The RxMicro framework supports the following container Java types:

- `java.util.List<V>`;
- `java.util.Set<V>`;
- `java.util.SortedSet<V>`;
- `java.util.Map<K, V>`;

where `K` and `V` can be:

- `java.lang.Boolean`;
- `java.lang.Long`;
- `java.math.BigDecimal`;
- `java.lang.String`;
- `CUSTOM_TYPE`.

The RxMicro framework uses `,` character as collection (List, Set, SortedSet, Map.Entry) value delimiter and '=' character as key-value separator:

```
list=red,green,blue
set=red,green,blue
sorted-set=red,green,blue
map=red=0xFF0000,green=0x00FF00,blue=0x0000FF
```

The RxMicro framework uses `reflection` to initialize config instances. But container parametrization types are not available for the `reflection` reading.

Thus the RxMicro framework tries to guess which type must be created using the following algorithm:

1. Try to convert to `java.lang.Boolean` type. If failed then goto `2` step.
2. Try to convert to `java.math.BigDecimal` type. If failed then goto `3` step.
3. Try to convert to `java.lang.Long` type. If failed then goto `4` step.
4. Try to convert to `CUSTOM_TYPE`. If failed then goto `5` step.
5. Return `java.lang.String` instance.

This means that if You provide a config list with different types, the RxMicro framework create a `java.util.List` with different types:

For example:

```
list=red,1.2,4,true
```

the result will be:

```
java.util.List.of(new String("red"), new BigDecimal("1.2"), Long.valueOf(4),
Boolean.valueOf(true));
```

and the `ClassCastException` will be thrown if Your config parameter is not of `java.util.List<java.lang.Object>` type.

To avoid the `ClassCastException` use the following recommendation:

- For `boolean` lists use `java.util.List<java.lang.Boolean>` type.
- For `integer number` lists use `java.util.List<java.lang.Long>` type.
- For `decimal number` lists use `java.util.List<java.math.BigDecimal>` type.
- For `string` lists use `java.util.List<java.lang.String>` type.
- For `custom type` lists use `java.util.List<CUSTOM_TYPE>` type.

**DON’T USE ANY OTHER TYPES FOR COLLECTION PARAMETRIZATION!**

### 4.8.7. Configuration Verifiers

If Your runtime has complex configuration the RxMicro team strong recommends enabling runtime strict mode.

If the runtime strict mode activated the RxMicro runtime invokes additional checks to find unused or redundant configurations.

To enable the runtime strict mode set `RX_MICRO_RUNTIME_STRICT_MODE` environment variable to the `true` value! *(Instead of environment variable You can use Java System property as well.)*

If runtime strict mode successful activated the following log message can be found:

```
[INFO] RxMicroRuntime !!! RxMicro Runtime Strict Mode is activated !!!
```

# 4.9. Logger

**Logger** is an integral component of any software system.

The RxMicro framework provides the `rxmicro.logger` module for logging important events during the work of microservices.

Creation and usage of a logger in the source code is no different from other logging frameworks:

```
final class MicroService {

    private static final Logger LOGGER = LoggerFactory.getLogger(MicroService.class);
①

    @GET("/")
    void test() {
        LOGGER.info("test message"); ②
    }
}
```

① Logger creation for the current microservice class.

② Logging of a message with `INFO` level.

The `Logger` interface is an abstraction over the real logger.

At the moment, there is only one implementation of this interface that delegates logging to the `java.logging` module.

### 4.9.1. Logger Configuration

#### 4.9.1.1. Using Configuration File

The main configuration file of the `java.logging` logger is the `jul.properties` classpath resource.

If classpath contains the `jul.test.properties` resource, then this resource overrides **all** configurations of the `jul.properties` resource.

This function allows configuring the logger for a test environment.

The `jul.properties` classpath resource must contain a configuration in the standard format for the `java.logging` module.

Example of logger configuration:

```
.level=INFO
io.rxmicro.config.level=DEBUG
```

The `rxmicro.logger` module supports all logging levels from the following sets: `{OFF, ERROR, WARN, INFO, DEBUG, TRACE, ALL}` and `{OFF, SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST, ALL}`.

Therefore, in the `jul.properties` configuration file You can use any logging level. When activating the `java.logging` logger, the RxMicro framework automatically converts levels from a set of `{OFF, ERROR, WARN, INFO, DEBUG, TRACE, ALL}` into `{OFF, SEVERE, WARNING, INFO, FINE, FINEST, ALL}`.

This option makes it possible to use unsupported but widely used in other logging frameworks logging levels for the `java.logging` logger.

#### 4.9.1.2. Default Reading Order of Config Sources.

By default, the `rxmicro.logger` module reads configuration sources in the following order *(From the lowest to the highest priority)*:

- Default config.
- Configuration from the `jul.properties` classpath resource if the resource found.
- Configuration from the `jul.test.properties` classpath resource if the resource found.

#### 4.9.1.3. Using Additional Config Sources for Logging Configuration

Besides `jul.properties` and `jul.test.properties` classpath resources the `rxmicro.logger` module supports the following configuration sources:

- Configuration using environment variables;
- Configuration using Java system properties;

- Configuration using properties file that is located at the following paths:
  - `./jul.properties`;
  - `$HOME/jul.properties`;
  - `$HOME/.rxmicro/jul.properties`;

where

- `.` - is the current project directory *(i.e.* `System.getProperty("user.dir")`*)*;
- `$HOME` - is the home directory *(i.e.* `System.getProperty("user.home")`*)*:
  - for Linux platform the `$HOME` directory is `/home/$USERNAME`;
  - for MacOS platform the `$HOME` directory is `/Users/$USERNAME`;
  - for Windows platform the `$HOME` directory is `C:\Documents and Settings\%USERNAME%` or `C:\Users\%USERNAME%`.

To enable the additional config sources use the `LoggerConfigSources` class:

```
①
static {
    LoggerConfigSources.setLoggerConfigSources(
            LoggerConfigSource.DEFAULT,
            LoggerConfigSource.CLASS_PATH_RESOURCE,
            LoggerConfigSource.TEST_CLASS_PATH_RESOURCE,
            LoggerConfigSource.FILE_AT_THE_HOME_DIR,
            LoggerConfigSource.FILE_AT_THE_CURRENT_DIR,
            LoggerConfigSource.FILE_AT_THE_RXMICRO_CONFIG_DIR,
            LoggerConfigSource.ENVIRONMENT_VARIABLES,
            LoggerConfigSource.JAVA_SYSTEM_PROPERTIES
    );
}

private static final Logger LOGGER =
LoggerFactory.getLogger(LoggerConfigSourceTest.class);

public static void main(final String[] args) {
    LOGGER.info("Hello World!");
}
```

① - The logger config sources must be configured before the first usage of the `LoggerFactory`, otherwise these config settings will be ignored.

**If You know exactly which configuration sources should be used by the microservice, ALWAYS specify them explicitly!**

**With this approach, at the microservice starting, the RxMicro framework won't try to search for non-existent sources, spending precious microseconds!**

#### 4.9.1.4. Using Environment Variables And Java System Properties

After activation of the configuration using environment variables and(or) Java system properties, the RxMicro framework parses environment variables and(or) Java system properties.
If Your runtime contains an environment variable and (or) java system property with name that starts with `logger.` phrase, the `rxmicro.logger` module interprets it as configuration. For example to enable `TRACE` logger level for `MicroServiceLauncher` it is necessary to provide one of the following configurations:

- Example of the configuration using environment variable:

```
export logger.io.rxmicro.rest.server.level=TRACE

java -p lib:. -m module/package.MicroServiceLauncher
```

or

- Example of the configuration using Java system property:

```
java -p lib:. -Dlogger.io.rxmicro.rest.server.level=TRACE -m
module/package.MicroServiceLauncher
```

### 4.9.2. Logger Handler

By default the `INFO` logging level is activated for the all loggers.

All logger events are sent to the `SystemConsoleHandler` appender, which outputs:

- Logger events with `ERROR` level into `System.err`;
- Logger events with other levels into `System.out`.

The `rxmicro.logger` module follows the logs recommendations, that are defined at The Twelve-Factor App Manifest and logs messages to `System.out`/`System.err` only!

If these functions are not enough You can use any other logging framework: `Logback`, `Apache Log4j 2` and others.

P.S. You can use also the `FileHandler`, `SocketHandler`, etc that defined at the `java.logging` module.

If the `SystemConsoleHandler` appender must output all logging information into `System.out` or `System.err` it is necessary to set `stream` parameter:

- To enable `System.err` output for the all log levels use the following configuration:

```
io.rxmicro.logger.jul.SystemConsoleHandler.stream=stderror
```

- To enable `System.out` output for the all log levels use the following configuration:

```
io.rxmicro.logger.jul.SystemConsoleHandler.stream=stdout
```

To get more information about the configuration of the `SystemConsoleHandler` component read javadoc for this component:
SystemConsoleHandler.html

### 4.9.3. Pattern Formatter

The current version of the `rxmicro.logger` module is supported only one logger formatter: `PatternFormatter` with the default configuration:

```
io.rxmicro.logger.jul.PatternFormatter.pattern=%d{yyyy-MM-dd HH:mm:ss.SSS} [%p] %c:
%m%n
```

This class supports conversion specifiers that can be used as format control expressions. Each conversion specifier starts with a percent sign `%` and is followed by optional format modifiers, a conversion word and optional parameters between braces. The conversion word controls the data field to convert, e.g. logger name or date format.

The `PatternFormatter` supports the following conversion specifiers:

*Table 3. Conversion specifiers supported by the `PatternFormatter`.*

<table>
<tr><th>Conversion specifiers</th><th>Description</th></tr>
<tr><td>c{length}<br>lo{length}<br>logger{length}</td><td>Outputs the name of the logger at the origin of the logging event.<br>This conversion specifier takes a string as its first and only option.<br>Currently supported only one of the following options: {short}, {0}, {full}.<br>{short} is synonym for {0} option.<br>If no option defined this conversion specifier uses {full} option.<br><br>The following table describes option usage results:<br><br>
<table>
<tr><th>Conversion specifier</th><th>Logger name</th><th>Result</th></tr>
<tr><td>%logger</td><td>package.sub.Bar</td><td>package.sub.Bar</td></tr>
<tr><td>%logger{full}</td><td>package.sub.Bar</td><td>package.sub.Bar</td></tr>
<tr><td>%logger{short}</td><td>package.sub.Bar</td><td>Bar</td></tr>
<tr><td>%logger{0}</td><td>package.sub.Bar</td><td>Bar</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Conversion specifiers</th><th>Description</th></tr>
<tr><td>C{length}<br>class{length}</td><td>Outputs the fully-qualified class name of the caller issuing the logging request.<br>This conversion specifier takes a string as its first and only option.<br>Currently supported only one of the following options: {short}, {0}, {full}.<br>{short} is synonym for {0} option.<br>If no option defined this conversion specifier uses {full} option.<br><br>The following table describes option usage results:<br><br>
<table>
<tr><th>Conversion specifier</th><th>Logger name</th><th>Result</th></tr>
<tr><td>%logger</td><td>package.sub.Bar</td><td>package.sub.Bar</td></tr>
<tr><td>%logger{full}</td><td>package.sub.Bar</td><td>package.sub.Bar</td></tr>
<tr><td>%logger{short}</td><td>package.sub.Bar</td><td>Bar</td></tr>
<tr><td>%logger{0}</td><td>package.sub.Bar</td><td>Bar</td></tr>
</table>
<br>Generating the caller class information is not particularly fast. Thus, its use should be avoided unless execution speed is not an issue!</td></tr>
<tr><td>d{pattern}<br>date{pattern}<br>d{pattern, timezone}<br>date{pattern, timezone}</td><td>Used to output the date of the logging event.<br>The date conversion word admits a pattern string as a parameter.<br>The pattern syntax is compatible with the format accepted by DateTimeFormatter.<br>If {@code timezone} is specified, this conversion specifier uses ZoneId.of(String) method to parse it, so timezone syntax must be compatible with zone id format.<br><br>The following table describes option usage results:<br><br>
<table>
<tr><th>Conversion specifier</th><th>Result</th></tr>
<tr><td>%date</td><td>2020-01-02 03:04:05.123</td></tr>
<tr><td>%date{yyyy-MM-dd}</td><td>2020-01-02</td></tr>
<tr><td>%date{HH:mm:ss.SSS}</td><td>03:04:05.123</td></tr>
<tr><td>%date{, UTC}</td><td>2020-01-02 03:04:05.123</td></tr>
</table>
<br>If pattern is missing (For example: %d, %date, %date{, UTC}}, the default pattern will be used: yyyy-MM-dd HH:mm:ss.SSS</td></tr>
</table>

| Conversion specifiers | Description |
|---|---|
| **F**<br>**file`** | Outputs the file name of the Java source file where the logging request was issued.<br><br>**Generating the file information is not particularly fast. Thus, its use should be avoided unless execution speed is not an issue!** |
| **L**<br>**line`** | Outputs the line number from where the logging request was issued.<br><br>**Generating the file information is not particularly fast. Thus, its use should be avoided unless execution speed is not an issue!** |
| **m**<br>**mes`**<br>**message** | Outputs the application-supplied message associated with the logging event. |
| **M**<br>**method`** | Outputs the method name where the logging request was issued.<br><br>**Generating the method name is not particularly fast. Thus, its use should be avoided unless execution speed is not an issue.** |
| **n** | Outputs the platform dependent line separator character or characters.<br><br>This conversion word offers practically the same performance as using non-portable line separator strings such as `\n`, or `\r\n`. Thus, it is the preferred way of specifying a line separator. |
| **p**<br>**le`**<br>**level** | Outputs the level of the logging event. |
| **r**<br>**relative** | Outputs the number of milliseconds elapsed since the start of the application until the creation of the logging event. |
| **t**<br>**thread** | Outputs the name of the thread that generated the logging event. |
| **id**<br>**rid**<br>**request-id**<br>**request_id**<br>**requestId** | Outputs the request id if specified.<br><br>This specifier displays the request id that retrieved from HTTP request if the request tracing is enabled. |

For request tracing feature usage Your code must use the overloaded logger methods with `RequestIdSupplier` argument:

```
public interface Logger {

    // ...

    void trace(RequestIdSupplier requestIdSupplier, Object...
otherArguments);

    void debug(RequestIdSupplier requestIdSupplier, Object...
otherArguments);

    void info(RequestIdSupplier requestIdSupplier, Object...
otherArguments);

    void warn(RequestIdSupplier requestIdSupplier, Object...
otherArguments);

    void error(RequestIdSupplier requestIdSupplier, Object...
otherArguments);

    // ...
}
```

### 4.9.4. Custom Log Event

The RxMicro framework provides a factory method to build a custom logger event:

```
private static final Logger LOGGER =
LoggerFactory.getLogger(LoggerEventBuilderTest.class);

public static void main(final String[] args) {
    final LoggerEventBuilder builder = LoggerFactory.newLoggerEventBuilder();
    builder.setMessage("Hello World!");
    LOGGER.info(builder.build());
}
```

A custom logger event can be useful if Your log message depends on some parameter.
For example, the following methods implement the same logic:

```
public void log1(final Throwable throwable,
                 final boolean withStackTrace) {
    final LoggerEventBuilder builder = LoggerFactory.newLoggerEventBuilder()
            .setMessage("Some error message: ?", throwable.getMessage());
    if (withStackTrace) {
        builder.setThrowable(throwable);
    }
    LOGGER.error(builder.build());
}

public void log2(final Throwable throwable,
                 final boolean withStackTrace) {
    if (withStackTrace) {
        LOGGER.error(throwable, "Some error message: ?", throwable.getMessage());
    } else {
        LOGGER.error("Some error message: ?", throwable.getMessage());
    }
}
```

Besides that a custom logger event allows customizing the following auto detect parameters:

- Thread id;
- Thread name;
- Source class name;
- Source method name;
- Source file name;
- Source line number.

```
LOGGER.info(
        LoggerFactory.newLoggerEventBuilder()
                .setMessage("Some error message")
                .setThreadName("Test-Thread-Name")
                .setThreadId(34L)
                .setStackFrame("package.Class", "method", "Test.java", 85)
                .setThrowable(throwable)
                .build()
);
```

### 4.9.5. Multiline Logs Issue For Docker Environment

By default the `docker` log driver does not support multiline log data.
It means that if Your microservice prints a stacktrace of any exception to the `System.out` or `System.err` each stack trace element will be processed as separate log event.

There are several standard solutions to this problem. The RxMicro framework adds the one of them.

#### 4.9.5.1. Solution From RxMicro Framework

If Your logger configuration contains the following setting:

```
io.rxmicro.logger.jul.PatternFormatter.singleLine=true
```

than all multiline log events are processed by the RxMicro framework as single line ones:
i.e. the `PatternFormatter` component replaces `'\n'` character by the `"\\n"` string before printing it to the `System.out` or `System.err`.
*(For Windows platform the `"\r\n""` character combination will be replaced by `"\\r\\n"` string!)*

#### 4.9.5.2. How To View Original Log Events?

To view original logs You can use the `sed` util:

```
docker logs microservice-container | sed -e 's/\\n/\n/g'
```

or

```
kubectl logs microservice-pod | sed -e 's/\\n/\n/g'
```

To view original logs on Your log aggregation tool if `fluentd` open source data collector is used, it is necessary to add the following filter:

```
<filter exampleTag>
    @type record_transformer
    enable_ruby true
    <record>
        # --- Replace "\n" string by '\n' character ---
        log ${record["log"].gsub("\\n", "\n")}
    </record>
</filter>
```

FYI: This filter requires that your log messages are parsed and converted to the json property with `log` name before invocation of this filter!

# 4.10. JSON

**JSON** is a widely used message exchange format for distributed applications.

The RxMicro framework provides the `rxmicro.json` module for low-level and efficient work with this format.

Unfortunately **JSON** format is defined in at least seven different documents:

- `2002` - json.org, and the business card
- `2006` - IETF RFC 4627, which set the application/json MIME media type
- `2011` - ECMAScript 262, section 15.12
- `2013` - ECMA 404 according to Tim Bray (RFC 7159 editor)
- `2014` - IETF RFC 7158
- `2014` - IETF RFC 7159
- `2017` - IETF RFC 8259

(Read more at Parsing JSON is a Minefield article).

So the RxMicro framework implementation of JSON parser can be described using the following test suites:

- JSON Parsing Tests, Only Java Parsers
- JSON Parsing Tests, All Parsers

The RxMicro framework uses classes from the `rxmicro.json` module when automatically converting Java models to JSON format and vice versa.

**Therefore, a developer should not explicitly use this module!**

*However, the common idea about the capabilities of this module is needed for correct test writing!*

### 4.10.1. A Mapping Between JSON and Java Types

Since this module is used automatically, it is optimized for machine operations. Therefore, this module doesn't provide separate classes for JSON types. Instead, standard Java classes are used:

*Table 4. Mapping table between JSON and Java types.*

| JSON type | Java type |
|---|---|
| `object` | `java.util.Map<String, Object>` |
| `array` | `java.util.List<Object>` |
| `boolean` | `java.lang.Boolean` |
| `null` | `null` |
| `string` | `java.lang.String` |
| `number` | `io.rxmicro.json.JsonNumber` |

*Table 5. Mapping table between Java and JSON types.*

| Java type | JSON type |
|---|---|
| `java.util.Map<String, Object>` | `object` |
| `java.util.List<Object>` | `array` |
| `java.lang.Boolean` | `boolean` |
| `null` | `null` |
| `java.lang.String` | `string` |
| `? extends java.lang.Number` | `number` |
| `io.rxmicro.json.JsonNumber` | `number` |
| `Any Java Class` | `string` |

### 4.10.2. `rxmicro.json` Module Usage

To write tests correctly, it is necessary to have a common idea about the `rxmicro.json` module.

Let's look at a microservice that returns the result in JSON object format:

```
final class MicroService {

    @GET("/")
    CompletionStage<Response> produce() {
        return completedStage(new Response());
    }
}
```

The `Response` class - Java model of HTTP response in JSON format:

```
public final class Response {

    final Child child = new Child(); ①

    final List<Integer> values = List.of(25, 50, 75, 100); ②

    final String string = "text"; ③

    final Integer integer = 10; ④

    final BigDecimal decimal = new BigDecimal("0.1234"); ⑤

    final Boolean logical = true; ⑥
}
```

① Nested JSON object.

② JSON numeric array.

③ String data type.

④ Integer numeric data type.

⑤ Floating-point numeric data type.

⑥ Logical data type.

The `Child` class - Java model of the nested JSON object:

```
public final class Child {

    final Integer integer = 20;
}
```

For simplicity, each response model field is immediately initialized with test data.

As a result, the microservice returns the following JSON object:

```
{
  "child": {
    "integer": 20
  },
  "values": [
    25,
    50,
    75,
    100
  ],
  "string": "text",
  "integer": 10,
  "decimal": 0.1234,
  "logical": true
}
```

While writing REST-based microservice test or integration test, it is necessary to compare the expected JSON response with the one returned by REST-based microservice. *(The response returned by the microservice is available through the* `ClientHttpResponse.getBody()` *method)*:

```
@RxMicroRestBasedMicroServiceTest(MicroService.class)
final class MicroServiceTest {
    //..
    @Test
    void Should_return_Response() {
        final ClientHttpResponse response = httpClient.get("/").join();
        final Object actualBody = response.getBody(); ①
        // ...
    }
}
```

① The `ClientHttpResponse.getBody()` method returns the HTTP response body. Since the REST-based microservice returns a JSON object, this method returns the result as the `java.util.Map<String, Object>` object.

Therefore, to compare JSON objects, You need to create the `java.util.Map<String, Object>` object containing the expected properties of the JSON object:

```
@RxMicroRestBasedMicroServiceTest(MicroService.class)
final class MicroServiceTest {
    //..
    @Test
    void Should_return_Response() {
        final ClientHttpResponse response = httpClient.get("/").join();
        final Object actualBody = response.getBody(); ①
        final Object expectedBody = Stream.of(
                Map.entry("child", Map.of(
                        "integer", new io.rxmicro.json.JsonNumber("20")
                )),
                Map.entry("values", List.of(
                        new io.rxmicro.json.JsonNumber("25"),
                        new io.rxmicro.json.JsonNumber("50"),
                        new io.rxmicro.json.JsonNumber("75"),
                        new io.rxmicro.json.JsonNumber("100")
                )),
                Map.entry("string", "text"),
                Map.entry("integer", new io.rxmicro.json.JsonNumber("10")),
                Map.entry("decimal", new io.rxmicro.json.JsonNumber("0.1234")),
                Map.entry("logical", true)
        ).collect(Collectors.toMap(
                Map.Entry::getKey,
                Map.Entry::getValue,
                (u, v) -> u, LinkedHashMap::new
        )); ②
        assertEquals(expectedBody, actualBody); ③
    }
}
```

① A JSON object returned by microservice.

② Expected JSON object.

③ Comparison of JSON objects using the `java.lang.Object.equals()` method.

According to the JSON specification, the property order in JSON object is undefined.

Thus, the following JSON objects:

```
{
    "firstname" : "David",
    "lastname" : "Smith"
}
```

and

```
{
    "lastname" : "Smith",
    "firstname" : "David"
}
```

are considered to be the same.

**The RxMicro framework always arranges JSON object properties!**

**Thus, the order of JSON properties always corresponds to the order of fields in the Java model!**

This allows You to compare JSON objects in the form of `java.util.Map<String, Object>` using the `java.lang.Object.equals()` method.

If Your microservice returns the JSON object with unordered properties, use `JsonFactory.orderedJsonObject(Object jsonObject)` method to sort properties before comparison!

For convenient creation of the expected JSON object, it is recommended to use the `JsonFactory` utility class. This class arranges JSON properties and automatically converts all `java.lang.Number` types into the `io.rxmicro.json.JsonNumber` type:

```
@RxMicroRestBasedMicroServiceTest(MicroService.class)
final class MicroServiceTest {

    private BlockingHttpClient blockingHttpClient;

    @Test
    void Should_return_Response() {
        final ClientHttpResponse response = blockingHttpClient.get("/");

        final Object actualBody = response.getBody(); ①
        final Object expectedBody = jsonObject(
                "child", jsonObject("integer", 20), ④
                "values", jsonArray(25, 50, 75, 100),   ⑤
                "string", "text",
                "integer", 10,
                "decimal", new BigDecimal("0.1234"),
                "logical", true
        ); ②
        assertEquals(expectedBody, actualBody); ③
    }
}
```

① A JSON object returned by microservice.

② Expected JSON object.

③ Comparison of JSON objects using the `java.lang.Object.equals()` method.

④ To create JSON object, it is recommended to use the `JsonFactory.jsonObject` method.

⑤ To create JSON array, it is recommended to use the `JsonFactory.jsonArray` method.

The `ClientHttpResponse` interface besides `body()` method also contains `bodyAsBytes()` and `bodyAsString()` ones.

Therefore in the test You can compare JSON objects by comparing their string representation or using a third-party Java library that supports the JSON format. (For example, `JSON-java`, `Java API for JSON Processing`, etc.)

**Thus, the RxMicro framework recommends using the `rxmicro.json` module to compare JSON objects, but at the same time provides an opportunity to use any other JSON framework!**

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-common/json`

### 4.10.3. Json Wrappers

If you want to work with JSON format using usual Java style, the RxMicro framework provides wrapper classes for JSON object and array:

```
@RxMicroRestBasedMicroServiceTest(MicroService.class)
final class MicroServiceTestWithWrappers {

    private BlockingHttpClient blockingHttpClient;

    static Stream<Function<ClientHttpResponse, JsonObject>>
toJsonObjectWrapperConverterProvider() {
        return Stream.of(
                ①
                response -> JsonWrappers.toJsonObject(response.getBody()),
                ②
                response ->
JsonWrappers.readJsonObject(response.getBodyAsString(UTF_8))
        );
    }

    @ParameterizedTest
    @MethodSource("toJsonObjectWrapperConverterProvider")
    void Should_return_Response(final Function<ClientHttpResponse, JsonObject>
converter) {
        final ClientHttpResponse response = blockingHttpClient.get("/");

        final JsonObject actualBody = converter.apply(response);

        assertEquals(
                new JsonObject()
                        .set("integer", 20),
                actualBody.getJsonObject("child")
        );
        assertEquals(
                new JsonArray()
                        .add(25).add(50).add(75).add(100),
                actualBody.getJsonArray("values")
        );
        assertEquals(
                "text",
                actualBody.getString("string")
        );
        assertEquals(
                10,
                actualBody.getNumber("integer").intValueExact()
        );
        assertEquals(
                new BigDecimal("0.1234"),
                actualBody.getNumber("decimal").bigDecimalValueExact()
```

```
        );
        assertTrue(
                actualBody.getBoolean("logical")
        );
    }
}
```

① - To convert a internal view of JSON object to the JSON wrapper use `JsonWrappers.toJsonObject` method.

② - To convert a string view of JSON object to the JSON wrapper use `JsonWrappers.readJsonObject` method.

## 4.11. Native Transports

The RxMicro framework uses `Netty` to perform asynchronous non-blocking IO operations. To increase productivity, `Netty` allows the use of `native transports`.

To enable `native transports` feature, add one of the following dependencies:

```
<dependency>
    <groupId>io.rxmicro</groupId>
    <artifactId>rxmicro-netty-native-linux</artifactId>
    <version>${rxmicro.version}</version>
</dependency>
```

```
<dependency>
    <groupId>io.rxmicro</groupId>
    <artifactId>rxmicro-netty-native-osx</artifactId>
    <version>${rxmicro.version}</version>
</dependency>
```

```
<dependency>
    <groupId>io.rxmicro</groupId>
    <artifactId>rxmicro-netty-native</artifactId>
    <version>${rxmicro.version}</version>
</dependency>
```

```
<dependency>
    <groupId>io.rxmicro</groupId>
    <artifactId>rxmicro-netty-native-all</artifactId>
    <version>${rxmicro.version}</version>
</dependency>
```

- The `rxmicro-netty-native-linux` dependency adds the `netty-transport-native-epoll` artifact;
- The `rxmicro-netty-native-osx` dependency adds the `netty-transport-native-kqueue` artifact;
- The `rxmicro-netty-native` dependency activates `native transports` for the current platform:
  - in case of the `Linux` current platform, the `netty-transport-native-epoll` artifact is added;
  - in case of the `MacOS` current platform, the `netty-transport-native-kqueue` artifact is added;
  - otherwise `native transports` is not activated for the current platform;
- the `rxmicro-netty-native-all` dependency adds the `netty-transport-native-epoll` and `netty-transport-native-kqueue` artifacts.

Adding a dependency to the microservice project:

```
<dependency>
    <groupId>io.rxmicro</groupId>
    <artifactId>rxmicro-netty-native-linux</artifactId>
    <version>${rxmicro.version}</version>
</dependency>
```

instead of

```
<dependency>
    <groupId>io.netty</groupId>
    <artifactId>netty-transport-native-epoll</artifactId>
    <version>${netty.version}</version>
    <classifier>linux-x86_64</classifier>
</dependency>
```

allows using the `Netty native transports` library version compatible with all other `Netty` libraries used by the RxMicro framework.

Therefore, the `rxmicro-netty-native-···` **modules don't contain any logic. They just add `Netty native transports` libraries of the correct version.**

If `native transports` has been successfully activated, the information message about the start of the HTTP server will display the corresponding type of transport.

```
2020-02-02 20:14:11.707 [INFO]
io.rxmicro.rest.server.netty.internal.component.NettyServer :
Server started at 0.0.0.0:8080 using EPOLL transport in 500 millis. ①
```

① The `using EPOLL transport` message format means that `Netty` will use the `netty-transport-native-epoll` library.

```
2020-02-02 20:14:11.707 [INFO]
io.rxmicro.rest.server.netty.internal.component.NettyServer :
Server started at 0.0.0.0:8080 using KQUEUE transport in 500 millis. ①
```

① The `using KQUEUE transport` message format means that `Netty` will use the `netty-transport-native-kqueue` library.

# 5. REST Controller

REST Controller is a class that contains at least one declarative HTTP request handler.

Each REST-based microservice project must contain at least one REST Controller.
*(The REST Controller is the entry point for the client.)*

To create REST controllers, the RxMicro framework provides the following modules:

- The `rxmicro.rest` is a basic module that defines basic `RxMicro Annotations`, required when using the `REST` architecture of building program systems;
- The `rxmicro.rest.server` is a basic HTTP server module used to create REST controllers and run REST-based microservices;
- The `rxmicro.rest.server.netty` is an HTTP server implementation module based on `Netty`;
- The `rxmicro.rest.server.exchange.json` is a module for converting Java models to `JSON` format and vice versa;

Due to transit dependencies only two modules usually need to be added to a project: `rxmicro.rest.server.netty` and `rxmicro.rest.server.exchange.json`.
*(All other modules are automatically added to the project!)*

# 5.1. REST Controller Implementation Requirements.

## 5.1.1. REST Controller Class Requirements.

REST Controller is a Java class:

```
import io.rxmicro.rest.method.GET;

final class MicroService {

    @GET("/")
    void requestHandler() {

    }
}
```

that must meet the following requirements:

1. The class must extend the `java.lang.Object` one.
2. The class couldn't be an `abstract` one.
3. The class couldn't be a nested one.
4. The class must contain an accessible (not private) constructor without parameters.
   *(The last requirement can be ignored if Your project depends on `rxmicro.cdi` module.)*

A REST Controller class must be a public one, only if it contains the method: `public static void main(final String[] args)`. Otherwise, a REST Controller class can be nonpublic.

### 5.1.2. HTTP Request Handler Requirements.

HTTP request handler is a method, that must meet the following requirements:

1. The method couldn't be a `private` one.
2. The method couldn't be an `abstract` one.
3. The method couldn't be a `synchronized` one.
4. The method couldn't be a `static` one.
5. The method couldn't be a `native` one.
6. The method must be annotated by at least one of the following annotations:
    a. `GET`;
    b. `POST`;
    c. `PUT`;
    d. `DELETE`;
    e. `PATCH`;
    f. `HEAD`;
    g. `OPTIONS`.
7. The method must return a `void` type or one of the following reactive types:
    a. `CompletableFuture`;
    b. `CompletionStage`;
    c. `Mono`;
    d. `Completable`;
    e. `Single`;
    f. `Maybe`.
8. If the method returns a reactive type, that this type must be parametrized by a HTTP response model type. (The additional types such as `java.lang.Void` and `java.util.Optional` are supported also. *(Read more: Table 2, "Which class from a reactive library must be choose?")*).

The `Flux` and `Flowable` types are developed to handle data stream that may arrive within a certain period of time.

For such cases the HTTP protocol provides the special `Transfer-Encoding` mechanism.

But this mechanism is not supported by the RxMicro framework yet, so the `Flux` and `Flowable` types cannot be used in the HTTP request handler.

The RxMicro framework supports the following parameter types for the HTTP request handler:

1. Handler without parameters.
   *(This type is recommended for the simplest `stateless` microservices without parameters.)*

2. List of primitive parameters.
   *(This type is recommended for microservices, the behavior of which depends on 1-2 parameters.)*
3. Custom class modeling an HTTP request.
   *(This type is recommended for microservices, the behavior of which depends on 3 or more parameters.)*

When using the **Project Reactor** and **RxJava** reactive libraries, You need:

1. Add dependencies to `pom.xml`.
2. Add modules to `module-info.java`.

Adding dependencies to `pom.xml`:

```
<dependencies>
    <dependency>
        <groupId>io.projectreactor</groupId>
        <artifactId>reactor-core</artifactId>
        <version>${projectreactor.version}</version> ①
    </dependency>
    <dependency>
        <groupId>io.reactivex.rxjava3</groupId>
        <artifactId>rxjava</artifactId>
        <version>${rxjava.version}</version> ②
    </dependency>
</dependencies>
```

① The latest stable version of the `Project Reactor` library.

② The latest stable version of the `RxJava3` library.

Adding modules to `module-info.java`:

```
module example {
    requires reactor.core; ①
    requires io.reactivex.rxjava3; ②
}
```

① The `reactor.core` module when using the **Project Reactor** library.

② The `io.reactivex.rxjava3` module when using the **RxJava** library.

## 5.2. RxMicro Annotations

The RxMicro framework supports the following RxMicro Annotations, which are used to create and configure REST Controllers.

*Table 6. Supported RxMicro Annotations.*

| Annotation | Description |
|---|---|
| @GET | Denotes a method, that must handle a GET HTTP request. |
| @POST | Denotes a method, that must handle a POST HTTP request. |
| @PUT | Denotes a method, that must handle a PUT HTTP request. |
| @DELETE | Denotes a method, that must handle a DELETE HTTP request. |
| @PATCH | Denotes a method, that must handle a PATCH HTTP request. |
| @HEAD | Denotes a method, that must handle a HEAD HTTP request. |
| @OPTIONS | Denotes a method, that must handle a OPTIONS HTTP request. |
| @BaseUrlPath | Denotes a base URL path for the all request handlers at the REST controller. |
| @Version | Denotes a version of the REST controller. |
| @Header | Denotes that a field of Java model class is a HTTP header. |
| @HeaderMappingStrategy | Declares a strategy of header name formation based on Java model field name analysis.<br><br>*(By default, the CAPITALIZE_WITH_HYPHEN strategy is used. Thus, by using this strategy, the Header-Name name header corresponds to the headerName field name.)* |
| @AddHeader | Denotes a static HTTP header that must be added to the response, created by the request handler. |
| @SetHeader | Denotes a static HTTP header that must be set to the response, created by the request handler. |
| @RepeatHeader | Denotes the header, which name needs to be repeated for each element in the list.<br><br>*(This annotation applies only to fields with the java.util.List<?> type.)* |
| @Parameter | Denotes that a field of Java model class is a HTTP parameter. |
| @ParameterMappingStrategy | Declares a strategy of parameter name formation based on Java model field name analysis.<br><br>*(By default, the LOWERCASE_WITH_UNDERSCORED strategy is used. Thus, by using this strategy, the header_name name header corresponds to the headerName field name.)* |
| @PathVariable | Denotes that a field of Java model class is a path variable. |

| Annotation | Description |
| --- | --- |
| `@RemoteAddress` | Declares the Java model field as a field, in which the RxMicro framework must inject the remote client connection address. |
| `@RequestMethod` | Declares the Java model field as a field, in which the RxMicro framework must inject a method of the received request.<br><br>*(This feature is useful for request logging when one handler supports different HTTP methods.)* |
| `@RequestUrlPath` | Declares the Java model field as a field, in which the RxMicro framework must inject `URL path` of the received request.<br><br>*(This feature is useful for request logging using `path-variables`.)* |
| `@RequestBody` | Declares the Java model field as a field, in which the RxMicro framework must inject a body of the received request. |
| `@ResponseStatusCode` | Indicates to the RxMicro framework that the value of the Java model field should be used as a status code to be sent to the client. |
| `@ResponseBody` | Indicates to the RxMicro framework that the value of the Java model field should be used as a body to be sent to the client. |
| `@RequestId` | Declares the Java model field as a field, in which the RxMicro framework must inject a unique request ID. |
| `@SetStatusCode` | Declares a status code, which should be sent to the client in case of successful execution of the HTTP request handler. |
| `@NotFoundMessage` | Declares a message returned by the handler in case of no result. |
| `@RestServerGeneratorConfig` | Allows to configure the process of code generation by the `RxMicro Annotation Processor` for REST controllers. |
| `@EnableCrossOriginResourceSharing` | Activates the `Cross Origin Resource Sharing` for all request handlers in the REST controller. |

## 5.3. Return Types

The HTTP request handler supports two categories of returned results:

- HTTP response without body;
- HTTP response with body;

### 5.3.1. Supported Return Types for HTTP Response without Body

The RxMicro framework supports the following return result types for an HTTP response without body:

```
final class RestControllerWithoutBody {

    ①
    @GET("/void/void1")
    void void1() {
        //do something
    }

    @GET("/void/void2")
    void void2(final Request request) {
        //do something with request
    }

    @GET("/void/void3")
    void void3(final String requestParameter) {
        //do something with requestParameter
    }

    ②

    @GET("/jse/completedFuture1")
    CompletableFuture<Void> completedFuture1() {
        return CompletableFuture.completedFuture(null);
    }

    @GET("/jse/completedFuture2")
    CompletableFuture<Void> completedFuture1(final Request request) {
        return CompletableFuture.completedFuture(null);
    }

    @GET("/jse/completedFuture3")
    CompletableFuture<Void> completedFuture1(final String requestParameter) {
        return CompletableFuture.completedFuture(null);
    }

    ③

    @GET("/jse/completionStage1")
    CompletionStage<Void> completionStage1() {
        return CompletableFuture.completedStage(null);
    }

    @GET("/jse/completionStage2")
    CompletionStage<Void> completionStage2(final Request request) {
        return CompletableFuture.completedStage(null);
```

```
    }

    @GET("/jse/completionStage3")
    CompletionStage<Void> completionStage3(final String requestParameter) {
        return CompletableFuture.completedStage(null);
    }

    ④

    @GET("/spring-reactor/mono1")
    Mono<Void> mono1() {
        return Mono.just("").then();
    }

    @GET("/spring-reactor/mono2")
    Mono<Void> mono2(final String requestParameter) {
        return Mono.just("").then();
    }

    @GET("/spring-reactor/mono3")
    Mono<Void> mono4(final Request request) {
        return Mono.just("").then();
    }

    ⑤

    @GET("/rxjava3/completable1")
    Completable completable1() {
        return Completable.complete();
    }

    @GET("/rxjava3/completable2")
    Completable completable2(final Request request) {
        return Completable.complete();
    }

    @GET("/rxjava3/completable3")
    Completable completable3(final String requestParameter) {
        return Completable.complete();
    }
}
```

① The `void` type is supported mainly for test purposes.
*(The `void` type can also be applied when the request handler does not use any blocking operations. In cases where the request handler uses blocking operations, one of the reactive types must be used instead of the `void` type.)*

② The `CompletableFuture<Void>` type is recommended when using the **java.util.concurrent** library.

③ Instead of the `CompletableFuture<Void>` type, can also be used the `CompletionStage<Void>` type.

④ When using the **Project Reactor** library, only the `Mono<Void>` type can be used.

⑤ When using the **RxJava** library, only the `Completable` type can be used.

All the above mentioned handlers return an HTTP response without body:

```
@RxMicroRestBasedMicroServiceTest(RestControllerWithoutBody.class)
final class RestControllerWithoutBodyTest {

    private BlockingHttpClient blockingHttpClient;

    @ParameterizedTest
    @ValueSource(strings = {
            "/void/void1",
            "/void/void2",
            "/void/void3",

            "/jse/completedFuture1",
            "/jse/completedFuture2",
            "/jse/completedFuture3",

            "/jse/completionStage1",
            "/jse/completionStage2",
            "/jse/completionStage3",

            "/spring-reactor/mono1",
            "/spring-reactor/mono2",
            "/spring-reactor/mono3",

            "/rxjava3/completable1",
            "/rxjava3/completable2",
            "/rxjava3/completable3"
    })
    void Should_support_HTTP_responses_without_body(final String urlPath) {
        final ClientHttpResponse response = blockingHttpClient.get(urlPath);

        assertTrue(response.isBodyEmpty(), "Body not empty: " + response.getBody());
①
    }
}
```

① When testing all `RestControllerWithoutBody` class handlers, each handler returns an HTTP response without body.

The RxMicro framework recommends for request handlers that depend on 3 or more HTTP headers, HTTP parameters or `path variables` to create separate classes of request model.

**Upon implementation of HTTP headers, HTTP parameters or `path variables` directly into the handler, the code becomes hard to read!**

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-controller/rest-controller-handlers`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 5.3.2. Supported Return Types for HTTP Response with Body

The RxMicro framework supports the following return result types for an HTTP response with body:

```
final class RestControllerWithBody {

    ①

    @GET("/jse/completedFuture1")
    CompletableFuture<Response> completedFuture1() {
        return CompletableFuture.completedFuture(new Response());
    }

    @GET("/jse/completedFuture2")
    CompletableFuture<Response> completedFuture2(final Request request) {
        return CompletableFuture.completedFuture(new Response());
    }

    @GET("/jse/completedFuture3")
    CompletableFuture<Response> completedFuture3(final String requestParameter) {
        return CompletableFuture.completedFuture(new Response());
    }

    @GET("/jse/completedFuture4")
    CompletableFuture<Optional<Response>> completedFuture4() {
        return CompletableFuture.completedFuture(Optional.of(new Response()));
    }

    @GET("/jse/completedFuture5")
    CompletableFuture<Optional<Response>> completedFuture5(final Request request) {
        return CompletableFuture.completedFuture(Optional.of(new Response()));
    }

    @GET("/jse/completedFuture6")
    CompletableFuture<Optional<Response>> completedFuture6(final String
requestParameter) {
        return CompletableFuture.completedFuture(Optional.of(new Response()));
    }

    ②

    @GET("/jse/completionStage1")
    CompletionStage<Response> completionStage1() {
        return CompletableFuture.completedStage(new Response());
    }

    @GET("/jse/completionStage2")
    CompletionStage<Response> completionStage2(final Request request) {
        return CompletableFuture.completedStage(new Response());
```

```
    }

    @GET("/jse/completionStage3")
    CompletionStage<Response> completionStage3(final String requestParameter) {
        return CompletableFuture.completedStage(new Response());
    }

    @GET("/jse/completionStage4")
    CompletionStage<Optional<Response>> completionStage4() {
        return CompletableFuture.completedStage(Optional.of(new Response()));
    }

    @GET("/jse/completionStage5")
    CompletionStage<Optional<Response>> completionStage5(final Request request) {
        return CompletableFuture.completedStage(Optional.of(new Response()));
    }

    @GET("/jse/completionStage6")
    CompletionStage<Optional<Response>> completionStage6(final String
requestParameter) {
        return CompletableFuture.completedStage(Optional.of(new Response()));
    }

    ③

    @GET("/spring-reactor/mono1")
    Mono<Response> mono1() {
        return Mono.just(new Response());
    }

    @GET("/spring-reactor/mono2")
    Mono<Response> mono2(final Request request) {
        return Mono.just(new Response());
    }

    @GET("/spring-reactor/mono3")
    Mono<Response> mono3(final String requestParameter) {
        return Mono.just(new Response());
    }

    ④

    @GET("/rxjava3/single1")
    Single<Response> single1() {
        return Single.just(new Response());
    }

    @GET("/rxjava3/single2")
    Single<Response> single2(final Request request) {
        return Single.just(new Response());
    }
```

```
    @GET("/rxjava3/single3")
    Single<Response> single3(final String requestParameter) {
        return Single.just(new Response());
    }

    @GET("/rxjava3/maybe1")
    Maybe<Response> maybe1() {
        return Maybe.just(new Response());
    }

    @GET("/rxjava3/maybe2")
    Maybe<Response> maybe2(final Request request) {
        return Maybe.just(new Response());
    }

    @GET("/rxjava3/maybe3")
    Maybe<Response> maybe3(final String requestParameter) {
        return Maybe.just(new Response());
    }
}
```

① The `CompletableFuture<MODEL>` type is recommended when using the **java.util.concurrent** library.

② Instead of the `CompletableFuture<MODEL>` type, can also be used the `CompletionStage<MODEL>` type.

③ When using the **Project Reactor** library, only the `Mono<MODEL>` type can be used.

④ When using the **RxJava** library, the `Single<MODEL>` or `Maybe<MODEL>` can be used.

**Note that the reactive types must be parameterized by the HTTP response model class!**

All the above mentioned handlers return an HTTP response with body:

```
@RxMicroRestBasedMicroServiceTest(RestControllerWithBody.class)
final class RestControllerWithBodyTest {

    private BlockingHttpClient blockingHttpClient;

    @ParameterizedTest
    @ValueSource(strings = {
            "/jse/completedFuture1",
            "/jse/completedFuture2",
            "/jse/completedFuture3",
            "/jse/completedFuture4",
            "/jse/completedFuture5",
            "/jse/completedFuture6",

            "/jse/completionStage1",
            "/jse/completionStage2",
            "/jse/completionStage3",
            "/jse/completionStage4",
            "/jse/completionStage5",
            "/jse/completionStage6",

            "/spring-reactor/mono1",
            "/spring-reactor/mono2",
            "/spring-reactor/mono3",

            "/rxjava3/single1",
            "/rxjava3/single2",
            "/rxjava3/single3",

            "/rxjava3/maybe1",
            "/rxjava3/maybe2",
            "/rxjava3/maybe3"
    })
    void Should_support_HTTP_responses_with_body(final String urlPath) {
        final ClientHttpResponse response = blockingHttpClient.get(urlPath);

        assertEquals(
                jsonObject("message", "Hello World!"), ①
                response.getBody()
        );
    }
}
```

① When testing all `RestControllerWithBody` class handlers, each handler returns an HTTP response containing a JSON object, with the `Hello World!` message.

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-controller/rest-controller-handlers`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

## 5.4. Routing of Requests

The RxMicro framework supports request routing based on HTTP method, URL Path and HTTP body:

### 5.4.1. Routing of Requests Based on HTTP Method

The RxMicro framework allows handling identical HTTP requests, differing **only** in HTTP method, by different handlers:

```
final class RoutingUsingHTTPMethod {

    @GET("/")
    void get() {
        System.out.println("GET");
    }

    @HEAD("/")
    void head() {
        System.out.println("HEAD");
    }

    @OPTIONS("/")
    void options() {
        System.out.println("OPTIONS");
    }
}
```

```
@RxMicroRestBasedMicroServiceTest(RoutingUsingHTTPMethod.class)
final class RoutingUsingHTTPMethodTest {

    private BlockingHttpClient blockingHttpClient;

    private SystemOut systemOut;

    @ParameterizedTest
    @ValueSource(strings = {"GET", "HEAD", "OPTIONS"})
    void Should_route_to_valid_request_handler(final String method) {
        blockingHttpClient.send(method, "/");
        assertEquals(method, systemOut.asString());
    }
}
```

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-controller/rest-controller-routing

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 5.4.2. Routing of Requests Based on URL Path

The RxMicro framework allows handling HTTP requests on different URL Paths by different handlers:

```
final class RoutingUsingUrlPath {

    @GET("/urlPath1")
    void get1() {
        System.out.println("/urlPath1");
    }

    @GET("/urlPath2")
    void get2() {
        System.out.println("/urlPath2");
    }

    @GET("/${type}")
    void get3(final @PathVariable String type) { ①
        System.out.println("/${type}: " + type);
    }
}
```

① In addition to static URL Paths, the RxMicro framework also supports `path variables`.

```
@RxMicroRestBasedMicroServiceTest(RoutingUsingUrlPath.class)
final class RoutingUsingUrlPathTest {

    private BlockingHttpClient blockingHttpClient;

    private SystemOut systemOut;

    @ParameterizedTest
    @CsvSource({
            "/urlPath1,     /urlPath1",
            "/urlPath2,     /urlPath2",
            "/urlPath3,     /${type}: urlPath3",
            "/put,          /${type}: put",
            "/get,          /${type}: get"
    })
    void Should_route_to_valid_request_handler(final String urlPath,
                                               final String expectedOut) {
        blockingHttpClient.get(urlPath);
        assertEquals(expectedOut, systemOut.asString());
    }
}
```

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-controller/rest-controller-routing`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 5.4.3. Routing of Requests Based on HTTP Body

Some HTTP methods allows transferring request parameters both in the start line and in the request body.

Therefore, in addition to standard routing types, the RxMicro framework also supports routing based on the HTTP body availability:

```
final class RoutingUsingHttpBody {

    @GET("/")
    @HEAD("/")
    @OPTIONS("/")
    @DELETE("/")
    @PATCH("/")
    @POST(value = "/", httpBody = false)
    @PUT(value = "/", httpBody = false)
    void handleRequestsWithoutBody(final String parameter) { ①
        System.out.println("without body");
    }

    @PATCH(value = "/", httpBody = true)
    @POST("/")
    @PUT("/")
    void handleRequestsWithBody(final String parameter) { ②
        System.out.println("with body");
    }
}
```

① The request handler with parameters transferred in the start line.

② The request handler with parameters transferred in the request body.

For the `GET`, `HEAD`, `OPTIONS` and `DELETE` HTTP methods, the request parameters are **always** transferred in the start line.

For the `POST`, `PUT` and `PATCH` HTTP methods, the request parameters can be transferred both in the start line and in the request body. *(To configure the parameter transfer type, the* `httpBody` *option is used.)*

```
@RxMicroRestBasedMicroServiceTest(RoutingUsingHttpBody.class)
final class RoutingUsingHttpBodyTest {

    private BlockingHttpClient blockingHttpClient;

    private SystemOut systemOut;

    @ParameterizedTest
    @ValueSource(strings = {"GET", "HEAD", "POST", "PUT", "PATCH", "OPTIONS",
"DELETE"})
    void Should_route_to_handleRequestsWithoutBody(final String method) {
        blockingHttpClient.send(
                method,
                "/?parameter=test" ①
        );
        assertEquals("without body", systemOut.asString());
    }

    @ParameterizedTest
    @ValueSource(strings = {"POST", "PUT", "PATCH"})
    void Should_route_to_handleRequestsWithBody(final String method) {
        blockingHttpClient.send(
                method,
                "/",
                jsonObject("parameter", "test") ②
        );
        assertEquals("with body", systemOut.asString());
    }
}
```

① Parameter transfer in the start line.

② Parameter transfer in the request body as JSON object.

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-controller/rest-controller-routing`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

## 5.5. `Not Found` Logic

The RxMicro framework supports `Not Found` Logic for HTTP request handlers.

To activate this feature it's necessary to return a reactive type that supports optional result:

```
final class NotFoundMicroService {

    @GET("/get1")
    CompletableFuture<Optional<Response>> getOptional1(final Boolean found) { ①
        return completedFuture(found ? Optional.of(new Response()) :
Optional.empty());
    }

    @GET("/get2")
    Mono<Response> getOptional2(final Boolean found) { ②
        return found ? Mono.just(new Response()) : Mono.empty();
    }

    @GET("/get3")
    Maybe<Response> getOptional3(final Boolean found) { ③
        return found ? Maybe.just(new Response()) : Maybe.empty();
    }
}
```

① When using `CompletableFuture`/`CompletionStage` it is necessary to use the `java.util.Optional` contract, since the `CompletableFuture`/`CompletionStage` reactive types do not support optional result by default.

② Unlike `CompletableFuture`/`CompletionStage`, the `Mono` reactive type supports optional result.

③ Unlike `CompletableFuture`/`CompletionStage`, the `Maybe` reactive type also supports optional result.

When handling requests, the RxMicro framework checks the handler result:

- If the handler returns a response model, the RxMicro framework will convert it to an HTTP response with the `200` status and JSON representation of this model.
- If the handler returns an empty result, the RxMicro framework generates an HTTP response with the `404` and the standard `"Not Found"` error message.

```
@RxMicroRestBasedMicroServiceTest(NotFoundMicroService.class)
final class NotFoundMicroServiceTest {

    private BlockingHttpClient blockingHttpClient;

    @ParameterizedTest
    @ValueSource(strings = {"/get1", "/get2", "/get3"})
    void Should_return_found_response(final String urlPath) {
        final ClientHttpResponse response = blockingHttpClient.get(urlPath +
"?found=true");

        assertEquals(jsonObject("message", "Hello World!"), response.getBody()); ①
        assertEquals(200, response.getStatusCode()); ①
    }

    @ParameterizedTest
    @ValueSource(strings = {"/get1", "/get2", "/get3"})
    void Should_return_not_found_response(final String urlPath) {
        final ClientHttpResponse response = blockingHttpClient.get(urlPath +
"?found=false");

        assertEquals(jsonErrorObject("Not Found"), response.getBody()); ②
        assertEquals(404, response.getStatusCode()); ②
    }
}
```

① In case there is a result, the result is returned in JSON format, and the HTTP response status is `200`.

② If there is no result, a standard error message is returned, and the HTTP response status is `404`.

The RxMicro framework provides an option to customize the `Not Found` message:

```
final class CustomizeNotFoundMicroService {

    @GET("/")
    ①
    @NotFoundMessage("Custom not found message")
    CompletableFuture<Optional<Response>> getOptional1(final Boolean found) {
        return completedFuture(found ? Optional.of(new Response()) :
Optional.empty());
    }
}
```

① The `@NotFoundMessage` annotation allows You to specify a message that will be displayed to the client in case of missing result instead of the standard `"Not Found"` message.

```
@Test
void Should_return_custom_not_found_message() {
    final ClientHttpResponse response = blockingHttpClient.get("/?found=false");

    assertEquals(jsonErrorObject("Custom not found message"), response.getBody()); ①
    assertEquals(404, response.getStatusCode()); ②
}
```

① If there is no result, the `@NotFoundMessage` message, which is set by the annotation, is returned.

② The HTTP status is `404`.
(*The RxMicro framework does not allow overriding the HTTP status for* `Not Found` *logic!*)

For more control over the HTTP response generated in case of an error, use exception instead of `Not Found` Logic feature!

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-controller/rest-controller-notfound`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

# 5.6. Exceptions Usage

Each reactive type of returned request handler result supports two states:

- Successful completion signal.
- Completion signal with an error.

The feature of a successful completion signal consists in its uniqueness, i.e. if such a signal has occurred, it ensures successful completion of the business task. The feature of a completion signal with an error is that different types of errors may occur during the execution of the business task:

- validation error;
- data source connection error;
- computational algorithm error, etc.

It means that each request handler can return **only one successful result and several results with errors**.

So the RxMicro framework introduces the `error` concept. An error means any unsuccessful result.

**For simplified error handling, the RxMicro framework recommends using HTTP status codes for each error category!**

In case the HTTP code status is not sufficient, the RxMicro framework recommends using an additional text description.

For this purpose, the RxMicro framework defines a standard JSON model which is returned in case of any error:

```
{
  "message": "Not Found"
}
```

Thus, in case of an error, the client determines the error category basing on HTTP status code analysis. For more information, the client should analyze a text message.

### 5.6.1. Basic Class of Exceptions

When handling an HTTP request, the RxMicro framework defines the `HttpErrorException` base exception class.

**All custom exception types must extend this base class!**

For all child classes which extend the `HttpErrorException` class, when creating an exception instance **the stack trace is not filled**, as this information is redundant.

*(This behavior is achieved by using the* `protected RuntimeException(String, Throwable, boolean, boolean)` *protected constructor.)*

### 5.6.2. Using the User Defined Exceptions

If the request handler must return an HTTP status code other than the successful one, a separate exception class must be created:

```
public final class CustomNotFoundException extends HttpErrorException { ①

    private static final int STATUS_CODE = 404; ②

    public CustomNotFoundException(final String message) { ③
        super(STATUS_CODE, message);
    }
}
```

① The custom exception class must extend the `HttpErrorException` class.

② Each class must define the static `int STATUS_CODE` field with a status code that identifies the error category. *(This name is constant and is used by the `rxmicro.documentation` module when building the project documentation.)*

③ If You want to display a detailed message to the client, You need to add the constructor parameter.
*(If You want to return to the client only the HTTP status without the HTTP body, then create a constructor without parameters.)*

**The RxMicro framework does not support `null` as a business value!**

So if the `null` value is passed to the exception constructor as a message, then the RxMicro framework will return an HTTP response without a message body!
*(i.e. the `new CustomNotFoundException(null)` and `new CustomNotFoundException()` constructions are identical in terms of the RxMicro framework!)*

You can use two approaches to return an HTTP status code other than successful one:

```
final class MicroService {

    @PUT(value = "/business-object-1", httpBody = false)
    CompletableFuture<Void> updateObject1(final Integer id) {
        return failedFuture(new CustomNotFoundException("Object not found by id=" +
id)); ①
    }

    @PUT(value = "/business-object-2", httpBody = false)
    CompletableFuture<Void> updateObject2(final Integer id) {
        throw new CustomNotFoundException("Object not found by id=" + id); ②
    }
}
```

① Use the reactive type method to generate a completion signal with an error.

② Create and throw an exception instance.

From the client's point of view, the two above-mentioned approaches are exactly the same:

```
@RxMicroRestBasedMicroServiceTest(MicroService.class)
final class MicroServiceTest {

    private BlockingHttpClient blockingHttpClient;

    @ParameterizedTest
    @ValueSource(strings = {"/business-object-1", "/business-object-2"})
    void Should_return_404(final String urlPath) {
        final ClientHttpResponse response = blockingHttpClient.put(urlPath + "?id=0");

        assertEquals(jsonErrorObject("Object not found by id=0"), response.getBody());
①
        assertEquals(404, response.getStatusCode()); ②
    }
}
```

① As a detailed error message, the message passed to the exception constructor is returned. *(The `jsonErrorObject("…")` utility method is synonymous with the `jsonObject("message", "…")` method.)*

② The returned status code equals the status code defined in the exception class in the `int STATUS_CODE` field.

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-controller/rest-controller-errors`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 5.6.3. Error Signal Generation Methods

To return an HTTP response with an error it is necessary to use the appropriate reactive type method to generate the error signal.

*Table 7. Error signal generation method.*

| Reactive type | Static method |
|---|---|
| `CompletableFuture` | `CompletableFuture.failedFuture` |
| `CompletionStage` | `CompletableFuture.failedStage` |
| `Mono` | `Mono.error` |
| `Completable` | `Completable.error` |
| `Single` | `Single.error` |
| `Maybe` | `Maybe.error` |

Using the error signal generation method instead of throwing an exception instance requires writing more code but saves processor resources!

So it's up to You to determine which method to use in Your project!

### 5.6.4. Predefined Classes of Exceptions

The RxMicro framework defines the following predefined exception classes:

*Table 8. List of predefined exception classes.*

| Exception class | Status code | Description |
|---|---|---|
| `RedirectException` | `300..399` | A base class to inform the client about the need to perform `redirect`.<br><br>*(Instead of using a base class, it is recommended to use one of the following child ones:* `TemporaryRedirectException` *or* `PermanentRedirectException`*.)* |
| `TemporaryRedirectException` | `307` | A class that signals the need to perform `Temporary Redirect`. |
| `PermanentRedirectException` | `308` | A class that signals the need to perform `Permanent Redirect`. |
| `ValidationException` | `400` | A class signaling that the client has sent a bad request. |
| `InternalHttpErrorException` | `500` | A class signaling that an internal error has occurred during execution. |
| `UnexpectedResponseException` | `500` | A class signaling that the HTTP response, generated by the request handler, contains validation errors. |
| `HttpClientTimeoutException` | `504` | A class signaling that the HTTP client didn't receive a response from the server in given time. |

## 5.7. Overriding the Status Code

By default, if there are no errors during HTTP request handling, the HTTP server returns the `200` status code. If You need to change the status code returned by default, You must use the `@SetStatusCode` annotation:

```
final class MicroService {

    @GET("/200")
    void success() {
    }

    @GET("/307")
    @SetStatusCode(307)
    void permanentRedirect() {
    }

    @GET("/404")
    @SetStatusCode(404)
    void notFound() {
    }

    @GET("/500")
    @SetStatusCode(500)
    void internalError() {
    }
}
```

```
@RxMicroRestBasedMicroServiceTest(MicroService.class)
final class MicroServiceTest {

    private BlockingHttpClient blockingHttpClient;

    @ParameterizedTest
    @ValueSource(ints = {200, 307, 404, 500})
    void Should_return_valid_status_codes(final int expectedStatus) {
        final ClientHttpResponse response = blockingHttpClient.get("/" +
expectedStatus);

        assertEquals(expectedStatus, response.getStatusCode());
    }
}
```

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-controller/rest-controller-set-status-code

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

## 5.8. Redirecting of Requests

The RxMicro framework supports request redirection:

```
final class MicroService {

    @PUT(value = "/old-url")
    CompletableFuture<Void> redirect() {
        return failedFuture(new PermanentRedirectException("/new-url")); ①
    }

    ②
    @PUT(value = "/new-url")
    void put(final String parameter) {
        System.out.println(parameter);
    }
}
```

① To redirect a request, You need to return the `PermanentRedirectException` instance, indicating a new URL Path.

② Once the HTTP response with request redirection is returned, the HTTP client will automatically repeat the request to a new URL Path.

**If temporary redirection is required, the `TemporaryRedirectException` class must be used instead of the `PermanentRedirectException` class!**

The predefined redirection classes (`TemporaryRedirectException` and `PermanentRedirectException`) return the `307` and `308` HTTP statuses.

**According to the HTTP protocol rules, the HTTP method is preserved at such redirection!**

```
@RxMicroRestBasedMicroServiceTest(MicroService.class)
final class MicroServiceTest {

    ①
    @BlockingHttpClientSettings(followRedirects = ENABLED)
    private BlockingHttpClient blockingHttpClient;

    private SystemOut systemOut;

    @Test
    void Should_redirect() {
        blockingHttpClient.put(
                "/old-url",
                jsonObject("parameter", "test")
        ); ②

        assertEquals("test", systemOut.asString()); ③
    }
}
```

① By default, the predefined HTTP client, that is used to perform test requests to the microservice, does not support redirection. Therefore, automatic redirection must be activated using the `@BlockingHttpClientSettings(followRedirects = ENABLED)` setting.

② The `PUT` request to the `/put1` URL Path is performed.

③ Consequently, the `MicroService.put` handler is automatically performed after the `MicroService.redirect` handler.

Besides using the `PermanentRedirectException`/`TemporaryRedirectException` exceptions, a custom model class or the `@SetStatusCode` and `@SetHeader` annotation composition can be used for redirection:

```
@PUT(value = "/old-url1")
CompletableFuture<RedirectResponse> redirect1() {
    return completedFuture(new RedirectResponse("/new-url")); ①
}

@PUT(value = "/old-url2")
②
@SetStatusCode(307)
@SetHeader(name = LOCATION, value = "/new-url")
void redirect2() {
}
```

① Using a custom model class with the `Location` header.

② Using the `@SetStatusCode` and `@SetHeader` annotation composition.

**FYI:** The `RedirectResponse` model class:

```
public final class RedirectResponse {

    @ResponseStatusCode
    final Integer status = 307;

    @Header(LOCATION)
    final String location;

    public RedirectResponse(final String location) {
        this.location = requireNonNull(location);
    }
}
```

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-controller/rest-controller-redirect

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

# 5.9. HTTP Headers Support

## 5.9.1. Basic Rules

The RxMicro framework supports HTTP headers in request and response models:

```
①
@HeaderMappingStrategy
public final class Request {

    ②
    @Header
    String endpointVersion;

    ③
    @Header("UseProxy")
    Boolean useProxy;

    public String getEndpointVersion() {
        return endpointVersion;
    }

    public Boolean getUseProxy() {
        return useProxy;
    }
}
```

```
①
@HeaderMappingStrategy
public final class Response {

    ②
    @Header
    final String endpointVersion;

    ③
    @Header("UseProxy")
    final Boolean useProxy;

    public Response(final Request request) {
        this.endpointVersion = request.getEndpointVersion();
        this.useProxy = request.getUseProxy();
    }

    public Response(final String endpointVersion,
                    final Boolean useProxy) {
        this.endpointVersion = endpointVersion;
        this.useProxy = useProxy;
    }
}
```

① The `@HeaderMappingStrategy` annotation allows setting common conversion rules for all header names from HTTP to Java format and vice versa in the current model class.
*(By default, the `CAPITALIZE_WITH_HYPHEN` strategy is used. The field name is used as the basic name, and then, following the rules of the specified strategy, the HTTP header name is generated.)*

② In order to declare a model field as the HTTP header field, it is necessary to use the `@Header` annotation.

③ Using the `@Header` annotation, it is possible to specify the HTTP header name, which does not correspond to the used strategy declared by the `@HeaderMappingStrategy` annotation.

The RxMicro framework uses the following algorithm to define the HTTP header name for the specified model field:

1. If the field is annotated by the `@Header` annotation with an explicit indication of the HTTP header name, the specified name is used;
2. If no HTTP header name is specified in the `@Header` annotation, the RxMicro framework checks for the `@HeaderMappingStrategy` annotation above the model class;
3. If the model class is annotated by the `@HeaderMappingStrategy` annotation, then the specified naming strategy is used.
   *(The field name is used as the basic name, and then, following the rules of the specified strategy, the HTTP header name is generated.)*
4. If the `@HeaderMappingStrategy` annotation is missing, the model class field name is used as the HTTP header name.

After creating model classes, it is necessary to create a request handler that uses the following models:

```
final class SimpleUsageMicroService {

    @GET("/get1")
    CompletableFuture<Response> get1(final Request request) { ①
        return completedFuture(new Response(request));
    }

    ②
    @GET("/get2")
    ③
    @HeaderMappingStrategy
    CompletableFuture<Response> get2(final @Header String endpointVersion,          ④
                                     final @Header("UseProxy") Boolean useProxy) {  ⑤
        return completedFuture(new Response(endpointVersion, useProxy));
    }
}
```

① If a separate model class has been created for an HTTP request, then this class must be passed to the method parameter that handles the HTTP request.

② Besides supporting HTTP request model classes, the RxMicro framework supports request handlers that accept HTTP headers as method parameters.

③ To define a common naming strategy for all HTTP headers which are passed as method parameters, the request handler must be annotated by the `@HeaderMappingStrategy` annotation.

④ To declare a method parameter as an HTTP header field, use the `@Header` annotation.

⑤ Using the `@Header` annotation, it is possible to specify an HTTP header name that does not correspond to the strategy used, declared by the `@HeaderMappingStrategy` annotation.

The RxMicro framework recommends for request handlers that depend on 3 or more HTTP headers to create separate classes of request model.

**Upon implementation of HTTP headers directly into the handler, the code becomes hard to read!**

Despite the different approaches to HTTP header handling support, from the client's point of view the two above-mentioned handlers are absolutely equal:

```
@RxMicroRestBasedMicroServiceTest(SimpleUsageMicroService.class)
final class SimpleUsageMicroServiceTest {

    private BlockingHttpClient blockingHttpClient;

    @ParameterizedTest
    @ValueSource(strings = {"/get1", "/get2"})
    void Should_process_HTTP_headers(final String path) {
        final ClientHttpResponse response = blockingHttpClient.get(
                path, ①
                HttpHeaders.of(
                        "Endpoint-Version", "v1",    ②
                        "UseProxy", true             ③
                )
        );

        final HttpHeaders responseHeaders = response.getHeaders();
        assertEquals("v1", responseHeaders.getValue("Endpoint-Version"));    ④
        assertEquals("true", responseHeaders.getValue("UseProxy"));          ④
    }
}
```

① When performing a request to different URL Paths, the result is the same.

② The `Endpoint-Version` name corresponds to the `endpointVersion` field of the request model. *(This correspondence is formed basing on the default strategy use, which is defined by the* `@HeaderMappingStrategy` *annotation.)*

③ The `UseProxy` name corresponds to the `useProxy` field of the request model, since this name is specified in the `@Header` annotation.

④ After executing the request in the resulting HTTP response, the `Endpoint-Version` and `UseProxy` headers are equal to `v1` and `true` respectively. *(The handler returns the same header values it received from an HTTP request.)*

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-controller/rest-controller-headers`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 5.9.2. Supported Data Types

The RxMicro framework supports the following Java types, which can be HTTP request model headers:

- `? extends Enum<?>`;
- `java.lang.Boolean`;
- `java.lang.Byte`;
- `java.lang.Short`;
- `java.lang.Integer`;
- `java.lang.Long`;
- `java.math.BigInteger`;
- `java.lang.Float`;
- `java.lang.Double`;
- `java.math.BigDecimal`;
- `java.lang.Character`;
- `java.lang.String`;
- `java.time.Instant`;

For floating point numbers, it is suggested to use the `java.math.BigDecimal` type instead of `java.lang.Float` or `java.lang.Double`.

Using the `java.math.BigDecimal` allows excluding errors of decimal number notation. **This is particularly important if the decimal number is used to represent currency-related data!**

and also the `java.util.List<T>`, `java.util.Set<T>` and the `java.util.Map<String, T>` type is parameterized by any of the above mentioned primitive types.

If the `java.util.Map<String, T>` type is used, then REST model can contain dynamic properties, where `keys` are the names of properties and `values` are the properties values.

```
final class ListHeaderMicroService {

    @GET("/")
    @HeaderMappingStrategy
    void consume(final @Header List<Status> supportedStatuses) {
        System.out.println(supportedStatuses);
    }
}
```

```
@RxMicroRestBasedMicroServiceTest(ListHeaderMicroService.class)
final class ListHeaderMicroServiceTest {

    private BlockingHttpClient blockingHttpClient;

    private SystemOut systemOut;

    static Stream<Arguments> supportedListProvider() {
        return Stream.of(
                arguments("created|approved|rejected"),                          ①
                arguments(new Status[]{created, approved, rejected}, 0),         ②
                arguments(new String[]{"created", "approved", "rejected"}, 0),   ③
                arguments(List.of(created, approved, rejected)),                 ④
                arguments(List.of("created", "approved", "rejected")),           ⑤
                arguments(HttpHeaders.of(                                        ⑥
                        "Supported-Statuses", "created",
                        "Supported-Statuses", "approved",
                        "Supported-Statuses", "rejected"
                )),
                arguments(HttpHeaders.of(                                        ⑥
                        "Supported-Statuses", "created|approved",
                        "Supported-Statuses", "rejected"
                ))
        );
    }

    @ParameterizedTest
    @MethodSource("supportedListProvider")
    void Should_support_list_headers(final Object headerValue) {
        blockingHttpClient.get(
                "/",
                headerValue instanceof HttpHeaders ?
                        (HttpHeaders) headerValue :
                        HttpHeaders.of("Supported-Statuses", headerValue) ⑦
        );

        assertEquals(
                "[created, approved, rejected]", ⑧
                systemOut.asString()
        );
    }
}
```

① If the HTTP header is a list of values, the list elements are transferred via the HTTP protocol as a string separated by the `|` symbol.
*(For HTTP response headers it is possible to activate the repeating headers mode.)*

② Besides transferring the list using a comma-separated string, the `BlockingHttpClient` component also supports:
an array of enumerations,

③ an array of string values,

④ a list of enumerations,

⑤ a list of string values.
*(The `BlockingHttpClient` component converts these types to a comma-separated string automatically!)*

⑥ Besides transferring HTTP headers as a comma-separated string, the `BlockingHttpClient` component also supports repeatable HTTP headers with different values.

⑦ All specified value types for the HTTP header, which is a list of values, are transferred as the `java.lang.Object` type. *(The `BlockingHttpClient` component automatically converts them to a comma-separated string and transfers via the HTTP protocol.)*

⑧ The RxMicro framework converts different types to a list of enumerations and displays it in the console.

### 5.9.3. Static HTTP headers

For setting of the static headers, the RxMicro framework provides the `@AddHeader` and `@SetHeader` annotations:

```
①
@SetHeader(name = "Mode", value = "Demo")
final class StaticHeadersMicroService {

    @GET("/get1")
    void get1() {
    }

    @GET("/get2")
    ②
    @SetHeader(name = "Debug", value = "true")
    void get2() {
    }
}
```

① If the HTTP header is set for the REST controller, it is added to the HTTP responses for each handler.

② If the HTTP header is set for the handler, it is added to the HTTP responses only for that handler.

From the client's point of view, static headers do not differ from any others:

```
@Test
void Should_use_parent_header_only() {
    final ClientHttpResponse response = blockingHttpClient.get("/get1");

    assertEquals("Demo", response.getHeaders().getValue("Mode"));
}

@Test
void Should_use_parent_and_child_headers() {
    final ClientHttpResponse response = blockingHttpClient.get("/get2");

    assertEquals("Demo", response.getHeaders().getValue("Mode"));
    assertEquals("true", response.getHeaders().getValue("Debug"));
}
```

In order to understand the differences between the `@AddHeader` and `@SetHeader` annotations, take a look to the following example:

- `ComplexStaticHeadersMicroService.java`
- `ComplexStaticHeadersMicroServiceTest.java`

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-controller/rest-controller-headers`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 5.9.4. Repeating HTTP headers

If the HTTP header of an HTTP response is a list of values, the list elements are transferred by default via the HTTP protocol as a string separated by the `|` symbol.

If You want the HTTP header to be repeated for each value, You need to use the `@RepeatHeader` annotation:

```
@Header
List<Status> singleHeader = List.of(created, approved, rejected);

@Header
@RepeatHeader
List<Status> repeatingHeader = List.of(created, approved, rejected);
```

**The use of the `@RepeatHeader` annotation is supported for response models only!**

For request models it makes no sense, because the RxMicro framework converts any of the supported formats into a list of values.

As a result of converting the Java model to HTTP response, the result will be as follows:

```
assertEquals(
        "created|approved|rejected", ①
        response.getHeaders().getValue("Single-Header")
);
assertEquals(
        List.of("created", "approved", "rejected"), ②
        response.getHeaders().getValues("Repeating-Header")
);
```

① By default, HTTP header list elements are transferred via the HTTP protocol as a string separated by the `|` symbol.

② If the field is annotated by the `@RepeatHeader` annotation, then the header is repeated for each element of the list.

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-controller/rest-controller-headers

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

# 5.10. HTTP Parameters Handling

## 5.10.1. Basic Rules

The RxMicro framework supports HTTP parameters in request and response models:

```
①
@ParameterMappingStrategy
public final class Request {

    ②
    String endpointVersion;

    ③
    @Parameter("use-Proxy")
    Boolean useProxy;

    public String getEndpointVersion() {
        return endpointVersion;
    }

    public Boolean getUseProxy() {
        return useProxy;
    }
}
```

```
①
@ParameterMappingStrategy
public final class Response {

    ②
    final String endpointVersion;

    ③
    @Parameter("use-Proxy")
    final Boolean useProxy;

    public Response(final Request request) {
        this.endpointVersion = request.getEndpointVersion();
        this.useProxy = request.getUseProxy();
    }

    public Response(final String endpointVersion,
                    final Boolean useProxy) {
        this.endpointVersion = endpointVersion;
        this.useProxy = useProxy;
    }
}
```

① The `@ParameterMappingStrategy` annotation allows setting common conversion rules for all parameter names from HTTP to Java format and vice versa in the current model class. *(By default, the* `LOWERCASE_WITH_UNDERSCORED` *strategy is used. The field name is used as the basic name, and then, following the rules of the specified strategy, the HTTP parameter name is generated.)*

② In order to declare a model field as the HTTP parameter field, it is necessary to use the `@Parameter` annotation. *(Unlike the* `@Header` *annotation, the* `@Parameter` *annotation is optional. Thus, if the model field is not annotated by any annotation, then by default it is assumed that there is the* `@Parameter` *annotation above the model field.)*

③ Using the `@Parameter` annotation, it is possible to specify the HTTP parameter name, which does not correspond to the used strategy declared by the `@ParameterMappingStrategy` annotation.

The RxMicro framework uses the following algorithm to define the HTTP parameter name for the specified model field:

1. If the field is annotated by the `@Parameter` annotation with an explicit indication of the HTTP parameter name, the specified name is used;
2. If no HTTP parameter name is specified in the `@Parameter` annotation, the RxMicro framework checks for the `@ParameterMappingStrategy` annotation above the model class;
3. If the model class is annotated by the `@ParameterMappingStrategy` annotation, then the specified naming strategy is used. *(The field name is used as the basic name, and then, following the rules of the specified strategy, the HTTP parameter name is generated.)*
4. If the `@ParameterMappingStrategy` annotation is missing, the model class field name is used as the HTTP parameter name.

**Unlike the `@Header` annotation, the `@Parameter` annotation is optional!**

Thus, if the model field is not annotated by any annotation, then **by default it is assumed that there is the `@Parameter` annotation above the model field!**

After creating model classes, it is necessary to create an HTTP request handler that uses the following models:

```
final class SimpleUsageMicroService {

    @GET("/get1")
    @POST("/post1")
    CompletableFuture<Response> get1(final Request request) { ①
        return completedFuture(new Response(request));
    }

    ②
    @GET("/get2")
    @POST("/post2")
    ③
    @ParameterMappingStrategy
    CompletableFuture<Response> get2(final String endpointVersion,
④
                                     final @Parameter("use-Proxy") Boolean useProxy) {
⑤
        return completedFuture(new Response(endpointVersion, useProxy));
    }
}
```

① If a separate model class has been created for an HTTP request, then this class must be passed by the method parameter that handles the HTTP request.

② Besides supporting HTTP request model classes, the RxMicro framework supports request handlers that accept HTTP request parameters as method parameters.

③ To define a common naming strategy for all HTTP parameters which are passed as method parameters, the request handler must be annotated by the `@ParameterMappingStrategy` annotation.

④ To declare a method parameter as an HTTP parameter field, You do not need to use any additional annotations.

⑤ Using the `@Parameter` annotation, it is possible to specify an HTTP parameter name that does not correspond to the strategy used, declared by the `@ParameterMappingStrategy` annotation.

**Note that the RxMicro framework does not distinguish whether HTTP request parameters will be transferred to the handler using the start line or HTTP body!**

Therefore, if the parameters are the same, it is possible to use the same handler with the specified request model to handle both `GET` (parameters are transferred in the start line) and `POST` (parameters are transferred in the HTTP body) request.

The RxMicro framework recommends for request handlers that depend on 3 or more HTTP parameters to create separate classes of request model.

**Upon implementation of HTTP parameters directly into the handler, the code becomes hard to read!**

Despite the different approaches to HTTP parameter handling support, from the client's point of view the two above-mentioned handlers are absolutely equal to `GET` and `POST` requests respectively:

```
@RxMicroRestBasedMicroServiceTest(SimpleUsageMicroService.class)
final class SimpleUsageMicroServiceTest {

    private BlockingHttpClient blockingHttpClient;

    @ParameterizedTest
    @ValueSource(strings = {"/get1", "/get2"})
    void Should_process_HTTP_query_params(final String path) {
        final ClientHttpResponse response = blockingHttpClient.get(
                path, ①
                QueryParams.of(
                        "endpoint_version", "v1",   ②
                        "use-Proxy", true           ③
                )
        );

        assertHttpBody(response);
    }

    @ParameterizedTest
    @ValueSource(strings = {"/post1", "/post2"})
    void Should_process_HTTP_body_params(final String path) {
        final ClientHttpResponse response = blockingHttpClient.post(
                path, ①
                jsonObject(
                        "endpoint_version", "v1",   ②
                        "use-Proxy", true           ③
                )
        );

        assertHttpBody(response);
    }

    private void assertHttpBody(final ClientHttpResponse response) {
        assertEquals(
                jsonObject(
                        "endpoint_version", "v1", ④
                        "use-Proxy", true         ④
                ),
                response.getBody()
        );
    }
}
```

① When performing a request to different URL Paths, the result is the same. *(Despite the differences in parameter transfer for `GET` and `POST` requests.)*

② The `endpoint_version` name corresponds to the `endpointVersion` field of the request model. *(This*

*correspondence is formed basing on the default strategy use, which is defined by the* `@ParameterMappingStrategy` *annotation.)*

③ The `use-Proxy` name corresponds to the `useProxy` field of the request model, since this name is specified in the `@Parameter` annotation.

④ After executing the request in the resulting HTTP response, the `endpoint_version` and `use-Proxy` HTTP parameters are equal to `v1` and `true` respectively. *(The handler returns the same header values it received from an HTTP request.)*

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-controller/rest-controller-params`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 5.10.2. Supported Data Types

The RxMicro framework supports the following Java types, which can be HTTP request model parameters:

- `? extends Enum<?>`;
- `java.lang.Boolean`;
- `java.lang.Byte`;
- `java.lang.Short`;
- `java.lang.Integer`;
- `java.lang.Long`;
- `java.math.BigInteger`;
- `java.lang.Float`;
- `java.lang.Double`;
- `java.math.BigDecimal`;
- `java.lang.Character`;
- `java.lang.String`;
- `java.time.Instant`;
- `? extends Object`;
- `java.util.List<? extends Object>`;
- `java.util.Set<? extends Object>`;
- `java.util.Map<java.lang.String, ? extends Object>`;

For floating point numbers, it is suggested to use the `java.math.BigDecimal` type instead of `java.lang.Float` or `java.lang.Double`.

Using the `java.math.BigDecimal` allows excluding errors of decimal number notation. **This is particularly important if the decimal number is used to represent currency-related data!**

and also the `java.util.List<T>`, `java.util.Set<T>` and the `java.util.Map<String, T>` type is parameterized by any of the above mentioned primitive types.

If the `java.util.Map<String, T>` type is used, then REST model can contain dynamic properties, where `keys` are the names of properties and `values` are the properties values.

```
final class ListQueryParamMicroService {

    @GET("/")
    @ParameterMappingStrategy
    void consume(final List<Status> supportedStatuses) {
        System.out.println(supportedStatuses);
    }
}
```

```
@RxMicroRestBasedMicroServiceTest(ListQueryParamMicroService.class)
final class ListQueryParamMicroServiceTest {

    private BlockingHttpClient blockingHttpClient;

    private SystemOut systemOut;

    static Stream<Arguments> supportedListProvider() {
        return Stream.of(
                arguments("created|approved|rejected"),                          ①
                arguments(new Status[]{created, approved, rejected}, 0),         ②
                arguments(new String[]{"created", "approved", "rejected"}, 0),   ③
                arguments(List.of(created, approved, rejected)),                 ④
                arguments(List.of("created", "approved", "rejected")),           ⑤
                arguments(QueryParams.of(                                        ⑥
                        "supported_statuses", "created",
                        "supported_statuses", "approved",
                        "supported_statuses", "rejected"
                )),
                arguments(QueryParams.of(                                        ⑳
                        "supported_statuses", "created|approved",
                        "supported_statuses", "rejected"
                ))
        );
    }

    @ParameterizedTest
    @MethodSource("supportedListProvider")
    void Should_support_list_headers(final Object queryParamValue) {
        blockingHttpClient.get(
                "/",
                queryParamValue instanceof QueryParams ?
                        (QueryParams) queryParamValue :
                        QueryParams.of("supported_statuses", queryParamValue) ⑦
        );

        assertEquals(
                "[created, approved, rejected]", ⑧
                systemOut.asString()
        );
    }
}
```

① If the HTTP parameter is a list of values, the list elements are transferred via the HTTP protocol as a string separated by the `|` symbol.

② Besides transferring the list using a comma-separated string, the `BlockingHttpClient` component also supports:
an array of enumerations,

③ an array of string values,

④ a list of enumerations,

⑤ a list of string values.
*(The `BlockingHttpClient` component converts these types to a comma-separated string automatically!)*

⑥ Besides transferring HTTP parameters as a comma-separated string, the `BlockingHttpClient` component also supports repeatable query params with different values.

⑦ All specified value types for the HTTP parameter, which is a list of values, are transferred as the `java.lang.Object` type. *(The `BlockingHttpClient` component automatically converts them to a comma-separated string and transfers via the HTTP protocol.)*

⑧ The RxMicro framework converts different types to a list of enumerations and displays it in the console.

### 5.10.3. Complex Model Support

Unlike HTTP headers, HTTP parameters can be transferred in the request body.

Therefore, besides primitives the RxMicro framework also supports nested JSON objects and arrays.

Therefore, the list of supported types includes the following types:

- `? extends Object` - custom class of the nested Java model;
- `java.util.List<? extends Object>` - list of the nested Java model custom classes;
- `java.util.Set<? extends Object>` - set of the nested Java model custom classes;
- `java.util.Map<String, ? extends Object>` - map of the nested Java model custom classes;

Nested Model Example:

```
@ParameterMappingStrategy
public final class NestedModel {

    String stringParameter;

    BigDecimal bigDecimalParameter;

    Instant instantParameter;
}
```

As well as examples of more complex Java models that use a nested model:

```
@ParameterMappingStrategy
public final class ComplexRequest {

    Integer integerParameter; ①

    Status enumParameter; ①

    List<Status> enumsParameter; ②

    NestedModel nestedModelParameter; ③

    List<NestedModel> nestedModelsParameter; ④
}
```

```
@ParameterMappingStrategy
public final class ComplexResponse {

    final Integer integerParameter; ①

    final Status enumParameter; ①

    final List<Status> enumsParameter; ②

    final NestedModel nestedModelParameter; ③

    final List<NestedModel> nestedModelsParameter; ④

    public ComplexResponse(final ComplexRequest request) {
        this.integerParameter = request.integerParameter;
        this.enumParameter = request.enumParameter;
        this.enumsParameter = request.enumsParameter;
        this.nestedModelParameter = request.nestedModelParameter;
        this.nestedModelsParameter = request.nestedModelsParameter;
    }
}
```

① Primitive JSON type field.

② Primitive JSON array type.

③ Nested JSON object.

④ JSON array of JSON nested objects.

After creating model classes, it is necessary to create a request handler that uses the following models:

```
final class ComplexModelMicroService {

    @POST("/")
    CompletableFuture<ComplexResponse> handle(final ComplexRequest request) {
        return completedFuture(new ComplexResponse(request));
    }
}
```

Since the request is transferred in an HTTP body, the sent and received JSON objects must be the same:

```
@RxMicroRestBasedMicroServiceTest(ComplexModelMicroService.class)
final class ComplexModelMicroServiceTest {

    private BlockingHttpClient blockingHttpClient;

    @Test
    void Should_support_complex_models() {
        final Object jsonObject = jsonObject(
                "integer_parameter", 1,
                "enum_parameter", created,
                "enums_parameter", jsonArray(created, approved),
                "nested_model_parameter", jsonObject(
                        "string_parameter", "test",
                        "big_decimal_parameter", new BigDecimal("3.14"),
                        "instant_parameter", Instant.now()
                ),
                "nested_models_parameter", jsonArray(
                        jsonObject(
                                "string_parameter", "test1",
                                "big_decimal_parameter", new BigDecimal("1.1"),
                                "instant_parameter", Instant.now()
                        ),
                        jsonObject(
                                "string_parameter", "test2",
                                "big_decimal_parameter", new BigDecimal("1.2"),
                                "instant_parameter", Instant.now()
                        )
                )
        );
        final ClientHttpResponse response = blockingHttpClient.post("/", jsonObject);

        assertEquals(jsonObject, response.getBody());
    }
}
```

For automatic conversion of Java types to JSON types, it is recommended to use the `JsonFactory` test class from the `rxmicro-test-json` module.

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-controller/rest-controller-params

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

## 5.11. The `path variables` Support

### 5.11.1. Basic Rules

The RxMicro framework supports `path variables` in request models:

```
public final class Request {

    ①
    @PathVariable
    String category;

    ②
    @PathVariable("class")
    String type;

    @PathVariable
    String subType;

    public String getCategory() {
        return category;
    }

    public String getType() {
        return type;
    }

    public String getSubType() {
        return subType;
    }
}
```

① In order to declare a model field as the `path variable`, it is necessary to use the `@PathVariable` annotation.

② Using the `@PathVariable` annotation, it is possible to specify the `path variable` name.
*(If no name is specified, the model field name is used as the `path variable` name.)*

Unlike HTTP headers and parameters that are available also on the client side, `path variables` are available **only** within the HTTP request handlers.

**So for simplicity it is recommended to always use the model field name as the `path variable` name!**

After creating model classes, it is necessary to create an HTTP request handler that uses the following model:

```
final class MicroService {

    ①
    @GET("/1/${category}/${class}/${subType}")
    @GET("/1/${category}/${class}_${subType}")
    @GET("/1/${category}-${class}-${subType}")
    @GET("/1-${category}-${class}-${subType}")
    void consume(final Request request) {
        System.out.println(format(
                "?-?-?",
                request.getCategory(), request.getType(), request.getSubType()
        ));
    }

    ①
    @GET("/2/${category}/${class}/${subType}")
    @GET("/2/${category}/${class}_${subType}")
    @GET("/2/${category}-${class}-${subType}")
    @GET("/2-${category}-${class}-${subType}")
    void consume(final @PathVariable String category,       ②
                 final @PathVariable("class") String type,  ③
                 final @PathVariable String subType) {
        System.out.println(format(
                "?-?-?",
                category, type, subType
        ));
    }
}
```

① When using `path variables` in request models, be sure to use **all** `path variables` in the URL Path. *(Based on the analysis of the URL Path, and considering all `path variables`, the RxMicro framework selects the required HTTP request handler.)*

② In order to declare a method parameter as `path variable`, You must use the `@PathVariable` annotation.

③ Using `@PathVariable` annotation it is possible to specify the `path variable` name, which does not match the name of the method parameter.

The RxMicro framework recommends for request handlers that depend on 3 or more `path variables` to create separate classes of request model.

**Upon implementation of `path variables` directly into the handler, the code becomes hard to read!**

Despite the different approaches to `path variables` handling support, from the client's point of view the two above-mentioned handlers are absolutely equal:

```
@RxMicroRestBasedMicroServiceTest(MicroService.class)
final class MicroServiceTest {

    private BlockingHttpClient blockingHttpClient;

    private SystemOut systemOut;

    @ParameterizedTest
    @CsvSource({
            "/1/category/class/subType,     category-class-subType",
            "/1/category/class_subType,     category-class-subType",
            "/1/category-class-subType,     category-class-subType",
            "/1-category-class-subType,     category-class-subType",

            "/2/category/class/subType,     category-class-subType",
            "/2/category/class_subType,     category-class-subType",
            "/2/category-class-subType,     category-class-subType",
            "/2-category-class-subType,     category-class-subType",

            "/1/5/6/7,                      5-6-7",
            "/1/5/6_7,                      5-6-7",
            "/1/5-6-7,                      5-6-7",
            "/1-5-6-7,                      5-6-7",

            "/2/5/6/7,                      5-6-7",
            "/2/5/6_7,                      5-6-7",
            "/2/5-6-7,                      5-6-7",
            "/2-5-6-7,                      5-6-7",
    })
    void Should_support_path_variables(final String path,           ①
                                       final String expectedOut) {  ②
        blockingHttpClient.get(path);

        assertEquals(expectedOut, systemOut.asString());
    }
}
```

① The current `path`, which corresponds to one of the specified `path` templates.

② The expected string containing all extracted `path variables` from the current `path`, displayed in the console.

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-controller/rest-controller-path-variables

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 5.11.2. Supported Data Types

The RxMicro framework supports the following Java types, which can be `path variables` of the request model:

- `? extends Enum<?>`;
- `java.lang.Boolean`;
- `java.lang.Byte`;
- `java.lang.Short`;
- `java.lang.Integer`;
- `java.lang.Long`;
- `java.math.BigInteger`;
- `java.lang.Float`;
- `java.lang.Double`;
- `java.math.BigDecimal`;
- `java.lang.Character`;
- `java.lang.String`;
- `java.time.Instant`;

For floating point numbers, it is suggested to use the `java.math.BigDecimal` type instead of `java.lang.Float` or `java.lang.Double`.

Using the `java.math.BigDecimal` allows excluding errors of decimal number notation. **This is particularly important if the decimal number is used to represent currency-related data!**

# 5.12. Support of Internal Data Types

## 5.12.1. Basic Rules

The RxMicro framework works with internal data. Due to this, the developer can extract additional information about the HTTP request and gain more control over the generated HTTP response.

To work with internal data, it is necessary to create request and response models:

```
public final class Request {

    ①
    @RemoteAddress
    String remoteAddress1;

    ②
    @RequestUrlPath
    String urlPath;

    ③
    @RequestMethod
    String method;

    ④
    HttpVersion httpVersion;

    ⑤
    HttpHeaders headers;

    ⑥
    @RequestBody
    byte[] bodyBytes;

    public String getRemoteAddress1() {
        return remoteAddress1;
    }

    public String getUrlPath() {
        return urlPath;
    }

    public String getMethod() {
        return method;
    }

    public HttpVersion getHttpVersion() {
        return httpVersion;
    }

    public HttpHeaders getHeaders() {
        return headers;
    }

    public byte[] getBodyBytes() {
        return bodyBytes;
    }
}
```

① To extract the remote address of the client connection, it is necessary to use the `@RemoteAddress`

annotation.

② To extract the current `URL path`, it is necessary to use the `@RequestUrlPath` annotation.
*(This feature is useful for request logging using `path-variables`.)*

③ To extract the current HTTP method, it is necessary to use the `@RequestMethod` annotation.
*(This feature is useful for request logging when one handler supports different HTTP methods.)*

④ To extract the HTTP protocol version, it is necessary to use the `HttpVersion` type.

⑤ To extract all HTTP headers, it is necessary to use the `HttpHeaders` type.

⑥ To extract the request body content as a byte array, it is necessary to use the `@RequestBody` annotation.

```
public final class Response {

    ①
    @ResponseStatusCode
    final Integer status;

    ②
    final HttpVersion version;

    ③
    final HttpHeaders headers;

    ④
    @ResponseBody
    final byte[] body;

    public Response(final Integer status,
                    final HttpVersion version,
                    final HttpHeaders headers,
                    final byte[] body) {
        this.status = status;
        this.version = version;
        this.headers = headers;
        this.body = body;
    }
}
```

① To override the HTTP status code, it is necessary to use the `@ResponseStatusCode` annotation.

② To set the HTTP protocol version, it is necessary to use the `HttpVersion` type.
*(Currently only `1.0` and `1.1` HTTP versions are supported.)*

③ To set all HTTP headers, it is necessary to use the `HttpHeaders` type.

④ To set the response body content as a byte array, it is necessary to use the `@ResponseBody` annotation.

After creating model classes, it is necessary to create an HTTP request handler that uses the following models:

```
final class MicroService {

    @POST("/")
    CompletableFuture<Response> handle(final Request request) {
        return completedFuture(new Response(
                201,
                request.getHttpVersion(),
                HttpHeaders.of(
                        "Remote-Address", request.getRemoteAddress1(),
                        "Url-Path", request.getUrlPath(),
                        "Method", request.getMethod(),
                        "Custom-Header", request.getHeaders().getValue("Custom-
Header")
                ),
                request.getBodyBytes()
        ));
    }
}
```

After creating the REST controller, let's check that everything works:

```
@RxMicroRestBasedMicroServiceTest(MicroService.class)
final class MicroServiceTest {

    private BlockingHttpClient blockingHttpClient;

    @Test
    void Should_support_internal_types() {
        final Object body = jsonObject("message", "Hello World!");
        final ClientHttpResponse response = blockingHttpClient.post(
                "/",
                HttpHeaders.of("Custom-Header", "Custom-Value"),
                body
        );

        assertEquals(body, response.getBody());
        assertEquals(201, response.getStatusCode());
        assertSame(HTTP_1_1, response.getVersion());
        final HttpHeaders responseHeaders = response.getHeaders();
        final String remoteAddress = responseHeaders.getValue("Remote-Address");
        assertTrue(
                remoteAddress.contains("127.0.0.1"),
                "Invalid Remote-Address: " + remoteAddress
        );
        assertEquals("/", responseHeaders.getValue("Url-Path"));
        assertEquals("POST", responseHeaders.getValue("Method"));
        assertEquals("Custom-Value", responseHeaders.getValue("Custom-Header"));
    }
}
```

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-controller/rest-controller-internals

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 5.12.2. Supported Internal Data Types

*Table 9. Supported internal data types for a request model.*

| Java type | RxMicro Annotation | Is the annotation required? | Description |
|---|---|---|---|
| `HttpRequest` | None | - | HTTP request model |
| `HttpVersion` | None | - | HTTP protocol version |
| `HttpHeaders` | None | - | HTTP header model |
| `SocketAddress` | `@RemoteAddress` | **No** | Remote client address |
| `String` | `@RemoteAddress` | **Yes** | Remote client address |
| `String` | `@RequestUrlPath` | **Yes** | URL Path of the current request |
| `String` | `@RequestMethod` | **Yes** | HTTP request method |
| `byte[]` | `@RequestBody` | **Yes** | HTTP request body content |

Currently only `1.0` and `1.1` HTTP versions are supported.

*Table 10. Supported internal data types for a response model.*

| Java type | RxMicro Annotation | Is the annotation required? | Description |
|---|---|---|---|
| `HttpVersion` | None | - | HTTP protocol version |
| `HttpHeaders` | None | - | HTTP header model |
| `Integer` | `@ResponseStatusCode` | **Yes** | HTTP response status code |
| `byte[]` | `@ResponseBody` | **Yes** | HTTP response body content |

## 5.13. Versioning of REST Controllers

The RxMicro framework supports versioning of REST Controllers using two strategies:

- Versioning based on HTTP header analysis with the `Api-Version` name.
- Versioning based on URL Path fragment analysis.

### 5.13.1. Versioning Based on HTTP Header Analysis

The RxMicro framework allows creating identical REST controllers that differ only in version:

```
@Version(value = "v1", strategy = Version.Strategy.HEADER) ①
final class OldMicroService {

    @PATCH("/patch")
    void update() {
        System.out.println("old");
    }
}
```

① REST controller of the old `v1` version, using the `Version.Strategy.HEADER` strategy;

```
@Version(value = "v2", strategy = Version.Strategy.HEADER) ①
final class NewMicroService {

    @PATCH("/patch")
    void update() {
        System.out.println("new");
    }
}
```

① REST controller of the new `v2` version, using the `Version.Strategy.HEADER` strategy;

**Note that the rules for defining a handler are the same for two different classes!**

The correct selection of the appropriate REST controller handler can be checked with the following test:

```
@RxMicroRestBasedMicroServiceTest({OldMicroService.class, NewMicroService.class}) ①
final class MicroServiceTest {

    private BlockingHttpClient blockingHttpClient;

    private SystemOut systemOut;

    @ParameterizedTest
    @CsvSource({
            "v1,    old",
            "v2,    new"
    })
    void Should_route_to_valid_request_handler(final String versionHeaderValue,
                                               final String expectedOut) {
        blockingHttpClient.patch(
                "/patch",    ②
                HttpHeaders.of(
                        API_VERSION, ③
                        versionHeaderValue
                )
        );

        assertEquals(expectedOut, systemOut.asString());
    }
}
```

① The `@RxMicroRestBasedMicroServiceTest` annotation allows You to run several REST controllers in test mode.

② The test runs the `PATCH` request to a URL Path: `/`.

③ To specify the handler version, the standard `Api-Version` HTTP header is used.

If only REST controllers of a certain version need to be tested, then the `BlockingHttpClient` component can be set up for operation of only certain versions of REST controller using the `@BlockingHttpClientSettings` annotation.

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-controller/rest-controller-versioning-header`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 5.13.2. Versioning Based on URL Path Fragment Analysis

The RxMicro framework allows creating identical REST controllers that differ only in version:

```
@Version(value = "v1", strategy = Version.Strategy.URL_PATH) ①
final class OldMicroService {

    @PATCH("/patch")
    void update() {
        System.out.println("old");
    }
}
```

① REST controller of the old `v1` version, using the `Version.Strategy.URL_PATH` strategy;

```
@Version(value = "v2", strategy = Version.Strategy.URL_PATH) ①
final class NewMicroService {

    @PATCH("/patch")
    void update() {
        System.out.println("new");
    }
}
```

① REST controller of the new `v2` version, using the `Version.Strategy.URL_PATH` strategy;

**Note that the rules for defining a handler are the same for two different classes!**

The correct selection of the appropriate REST controller handler can be checked with the following test:

```
@RxMicroRestBasedMicroServiceTest({OldMicroService.class, NewMicroService.class}) ①
final class MicroServiceTest {

    private BlockingHttpClient blockingHttpClient;

    private SystemOut systemOut;

    @ParameterizedTest
    @CsvSource({
            "/v1,    old",
            "/v2,    new"
    })
    void Should_route_to_valid_request_handler(final String urlVersionPath,
                                               final String expectedOut) {
        blockingHttpClient.patch(
                urlVersionPath + "/patch"    ②
        );

        assertEquals(expectedOut, systemOut.asString());
    }
}
```

① The `@RxMicroRestBasedMicroServiceTest` annotation allows You to run several REST controllers in test mode.

② The test runs the `PATCH` request to different URL Paths;

If only REST controllers of a certain version need to be tested, then the `BlockingHttpClient` component can be set up for operation of only certain versions of REST controller using the `@BlockingHttpClientSettings` annotation.

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-controller/rest-controller-versioning-url-path`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

## 5.14. Base URL Path for All Handlers

To configure a base URL Path for all HTTP request handlers, the RxMicro framework provides the `@BaseUrlPath` annotation:

```
@BaseUrlPath("base-url-path")
final class MicroService {

    @GET("/path")
    void path() {
    }
}
```

```
@RxMicroRestBasedMicroServiceTest(MicroService.class)
final class MicroServiceTest {

    private BlockingHttpClient blockingHttpClient;

    @Test
    void Should_support_base_url() {
        final ClientHttpResponse response = blockingHttpClient.get("/base-url-
path/path");

        assertEquals(200, response.getStatusCode());
    }
}
```

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-controller/rest-controller-base-url-path`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

## 5.15. CORS Support

The RxMicro framework supports the `Cross Origin Resource Sharing (CORS)`.

To activate this function it is necessary to add the `@EnableCrossOriginResourceSharing` annotation:

```
@EnableCrossOriginResourceSharing ①
final class MicroService {

    @PATCH("/")
    void handle() {
        System.out.println("handle");
    }
}
```

① The `Cross Origin Resource Sharing (CORS)` feature is activated for **all** handlers of the specified REST controller.

When activating this feature, the RxMicro framework automatically adds a standard handler:

```
@RxMicroRestBasedMicroServiceTest(MicroService.class)
final class MicroServiceTest {

    private BlockingHttpClient blockingHttpClient;

    private SystemOut systemOut;

    @Test
    void Should_handle_PATCH_request() {
        blockingHttpClient.patch("/");

        assertEquals("handle", systemOut.asString());
    }

    ①
    @Test
    void Should_support_CORS_Options_request() {
        final ClientHttpResponse response = blockingHttpClient.options(
                "/",
                HttpHeaders.of(
                        ORIGIN, "test.rxmicro.io",
                        ACCESS_CONTROL_REQUEST_METHOD, PATCH
                )
        );

        assertEquals(ORIGIN, response.getHeaders().getValue(VARY));
        assertEquals("*",
response.getHeaders().getValue(ACCESS_CONTROL_ALLOW_ORIGIN));
        assertEquals(PATCH.name(),
response.getHeaders().getValue(ACCESS_CONTROL_ALLOW_METHODS));
    }
}
```

① The standard handler can handle `OPTIONS` requests with additional HTTP headers.

For more information on the `Cross Origin Resource Sharing (CORS)` support, check out the following examples:

- `ComplexCORSMicroService.java`
- `ComplexCORSMicroServiceTest.java`

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-controller/rest-controller-cors`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

## 5.16. Request ID

The Request Id feature described at Monitoring section.

## 5.17. Configuring the Code Generation Process

The RxMicro framework provides an ability to configure the code generation process for REST controllers. For this purpose, it is necessary to use the `@RestServerGeneratorConfig` annotation, that annotates the `module-info.java` module descriptor:

```
import io.rxmicro.rest.model.GenerateOption;
import io.rxmicro.rest.model.ServerExchangeFormatModule;
import io.rxmicro.rest.server.RestServerGeneratorConfig;

@RestServerGeneratorConfig(
        exchangeFormat = ServerExchangeFormatModule.AUTO_DETECT, ①
        generateRequestValidators = GenerateOption.AUTO_DETECT,  ②
        generateResponseValidators = GenerateOption.DISABLED     ③
)
module rest.controller.generator {
    requires rxmicro.rest.server.netty;
}
```

① The `exchangeFormat` parameter allows You to specify a format for message exchange with a client. *(By default, it is used the message exchange format added to the `module-info.java` descriptor. If several modules supporting the message exchange format are added to the `module-info.java` descriptor, then using the `exchangeFormat` parameter, You need to specify which module should be used for REST controllers.)*

② The `generateRequestValidators` parameter allows enabling/disabling the option of generating HTTP request validators for all handlers in the project. *(The `AUTO_DETECT` value means that validators will be generated only if the developer adds the `rxmicro.validation` module to the `module-info.java` descriptor.)*

③ The `generateResponseValidators` parameter allows enabling/disabling the option of generating HTTP response validators for all handlers in the project. *(The `DISABLED` value means that validators won't be generated by the `RxMicro Annotation Processor`.)*

The RxMicro team strong recommends using the `AUTO_DETECT` generate option always! (This is default value for `@RestServerGeneratorConfig` annotation).

*(HTTP response validation can be useful for identifying errors in business task implementation algorithms. For example, instead of returning an incorrect response model to a client, the microservice will throw an error. This approach increases the speed of error search and debugging of the source code that performs the business task.)*

**FYI:** By default the response validators are generated but not invoked! To activate the validation of responses it is necessary to set `enableAdditionalValidations` property:

```
new Configs.Builder()
        .withConfigs(new RestServerConfig()
                .setEnableAdditionalValidations(true)) ①
        .build();
```

① Enables the response validators

or

```
export rest-server.enableAdditionalValidations=true ①
```

① Enables the response validators

or using any other supported config types

Thus the RxMicro team recommends the following approach:

- Your project configuration must enable generation of validators for all possible model classes.
- Your `development` and `staging` environment must enable additional validation.
- Your `production` environment must disable additional validation!

# 6. REST Client

REST Client is an interface that contains at least one declarative HTTP request handler.

To create REST clients, the RxMicro framework provides the following modules:

- The `rxmicro.rest` is a basic module that defines basic `RxMicro Annotations`, required when using the `REST` architecture of building program systems;
- The `rxmicro.rest.client` is a basic module used to create and run REST clients;
- The `rxmicro.rest.client.jdk` is an HTTP client implementation module based on `Java HTTP Client`;
- The `rxmicro.rest.client.exchange.json` is a module for converting Java models to `JSON` format and vice versa;

Due to transit dependencies only two modules usually need to be added to a project: `rxmicro.rest.client.jdk` and `rxmicro.rest.client.exchange.json`.
*(All other modules are automatically added to the project!)*

# 6.1. REST Client Implementation Requirements

## 6.1.1. REST Client Interface Requirements

REST Client is a Java interface that annotated by the `@RestClient` annotation:

```
import java.util.concurrent.CompletableFuture;

import io.rxmicro.rest.client.RestClient;
import io.rxmicro.rest.method.GET;

@RestClient
public interface RESTClient {

    @GET("/")
    CompletableFuture<Model> getContent();
}
```

that must meet the following requirements:

1. The interface must be a `public` one.
2. The interface must be annotated by the required `@RestClient` annotation.
3. The interface couldn't extend any other interfaces.
4. The interface couldn't be a nested one.

### 6.1.2. HTTP Request Handler Requirements

HTTP request handler is a method, that must meet the following requirements:

1. The method couldn't be a `private` one.
2. The method couldn't be an `abstract` one.
3. The method couldn't be a `static` one.
4. The method must be annotated by **only one** of the following annotations:
   a. `GET`;
   b. `POST`;
   c. `PUT`;
   d. `DELETE`;
   e. `PATCH`;
   f. `HEAD`;
   g. `OPTIONS`.
5. The method must return one of the following reactive types:
   a. `CompletableFuture`;
   b. `CompletionStage`;
   c. `Mono`;
   d. `Completable`;
   e. `Single`.
6. If the method returns a reactive type, that this type must be parametrized by a HTTP response model type. (The additional types such as `java.lang.Void` and `java.util.Optional` are supported also. *(Read more: Table 2, "Which class from a reactive library must be choose?")*).

A REST client can contain `static`, `private` and `default` methods, but such methods couldn't be HTTP request handlers!

The `Flux` and `Flowable` types are developed to handle data stream that may arrive within a certain period of time.

For such cases the HTTP protocol provides the special `Transfer-Encoding` mechanism.

But this mechanism is not supported by the RxMicro framework yet, so the `Flux` and `Flowable` types cannot be used in the HTTP request handler.

The RxMicro framework supports the following parameter types for the HTTP request handler:

1. Handler without parameters.
   *(This type is recommended for the simplest `stateless` REST clients without parameters.)*
2. List of primitive parameters.

*(This type is recommended for REST clients, the behavior of which depends on 1-2 parameters.)*

3. Custom class modeling an HTTP request.
   *(This type is recommended for REST clients, the behavior of which depends on 3 or more parameters.)*

When using the **Project Reactor** and **RxJava** reactive libraries, You need:

1. Add dependencies to `pom.xml`.
2. Add modules to `module-info.java`.

Adding dependencies to `pom.xml`:

```
<dependencies>
    <dependency>
        <groupId>io.projectreactor</groupId>
        <artifactId>reactor-core</artifactId>
        <version>${projectreactor.version}</version> ①
    </dependency>
    <dependency>
        <groupId>io.reactivex.rxjava3</groupId>
        <artifactId>rxjava</artifactId>
        <version>${rxjava.version}</version> ②
    </dependency>
</dependencies>
```

① The latest stable version of the `Project Reactor` library.

② The latest stable version of the `RxJava3` library.

Adding modules to `module-info.java`:

```
module example {
    requires reactor.core; ①
    requires io.reactivex.rxjava3; ②
}
```

① The `reactor.core` module when using the **Project Reactor** library.

② The `io.reactivex.rxjava3` module when using the **RxJava** library.

## 6.2. RxMicro Annotations

The RxMicro framework supports the following RxMicro Annotations, which are used to create and configure REST Clients.

*Table 11. Supported RxMicro Annotations.*

| Annotation | Description |
|---|---|
| @RestClient | Denotes that an interface is a dynamic generated REST client. |
| @GET | Denotes a method, that must handle a GET HTTP request. |
| @POST | Denotes a method, that must handle a POST HTTP request. |
| @PUT | Denotes a method, that must handle a PUT HTTP request. |
| @DELETE | Denotes a method, that must handle a DELETE HTTP request. |
| @PATCH | Denotes a method, that must handle a PATCH HTTP request. |
| @HEAD | Denotes a method, that must handle a HEAD HTTP request. |
| @OPTIONS | Denotes a method, that must handle a OPTIONS HTTP request. |
| @BaseUrlPath | Denotes a base URL path for the all request handlers at the REST controller. |
| @Version | Denotes a version of the REST controller. |
| @Header | Denotes that a field of Java model class is a HTTP header. |
| @HeaderMappingStrategy | Declares a strategy of header name formation based on Java model field name analysis.<br><br>*(By default, the CAPITALIZE_WITH_HYPHEN strategy is used. Thus, by using this strategy, the Header-Name name header corresponds to the headerName field name.)* |
| @AddHeader | Denotes a static HTTP header that must be added to the response, created by the request handler. |
| @SetHeader | Denotes a static HTTP header that must be set to the response, created by the request handler. |
| @RepeatHeader | Denotes the header, which name needs to be repeated for each element in the list.<br><br>*(This annotation applies only to fields with the java.util.List<?> type.)* |
| @Parameter | Denotes that a field of Java model class is a HTTP parameter. |
| @ParameterMappingStrategy | Declares a strategy of parameter name formation based on Java model field name analysis.<br><br>*(By default, the LOWERCASE_WITH_UNDERSCORED strategy is used. Thus, by using this strategy, the header_name name header corresponds to the headerName field name.)* |

| Annotation | Description |
|---|---|
| `@PathVariable` | Denotes that a field of Java model class is a `path variable`. |
| `@RemoteAddress` | Declares the Java model field as a field, in which the RxMicro framework must inject the remote client connection address. |
| `@RequestMethod` | Declares the Java model field as a field, in which the RxMicro framework must inject a method of the received request.<br><br>*(This feature is useful for request logging when one handler supports different HTTP methods.)* |
| `@RequestUrlPath` | Declares the Java model field as a field, in which the RxMicro framework must inject `URL path` of the received request.<br><br>*(This feature is useful for request logging using `path-variables`.)* |
| `@RequestBody` | Declares the Java model field as a field, in which the RxMicro framework must inject a body of the received request. |
| `@ResponseStatusCode` | Indicates to the RxMicro framework that the value of the Java model field should be used as a status code to be sent to the client. |
| `@ResponseBody` | Indicates to the RxMicro framework that the value of the Java model field should be used as a body to be sent to the client. |
| `@RequestId` | Declares the Java model field as a field, in which the RxMicro framework must inject a unique request ID. |
| `@SetStatusCode` | Declares a status code, which should be sent to the client in case of successful execution of the HTTP request handler. |
| `@NotFoundMessage` | Declares a message returned by the handler in case of no result. |
| `@RestServerGeneratorConfig` | Allows to configure the process of code generation by the `RxMicro Annotation Processor` for REST controllers. |
| `@EnableCrossOriginResourceSharing` | Activates the `Cross Origin Resource Sharing` for all request handlers in the REST controller. |
| `@BaseUrlPath` | Denotes a base URL path for the all request handlers at the REST client. |
| `@Version` | Denotes a version of the REST client. |
| `@Header` | Denotes that a field of Java model class is a HTTP header. |
| `@HeaderMappingStrategy` | Declares a strategy of header name formation based on Java model field name analysis.<br><br>*(By default, the `CAPITALIZE_WITH_HYPHEN` strategy is used. Thus, by using this strategy, the `Header-Name` name header corresponds to the `headerName` field name.)* |
| `@AddHeader` | Denotes a static HTTP header that must be added to the request, created by REST client implementation. |

| Annotation | Description |
| --- | --- |
| `@SetHeader` | Denotes a static HTTP header that must be set to the request, created by REST client implementation. |
| `@RepeatHeader` | Denotes the header, which name needs to be repeated for each element in the list.<br><br>*(This annotation applies only to fields with the `java.util.List<?>` type.)* |
| `@Parameter` | Denotes that a field of Java model class is a HTTP parameter. |
| `@ParameterMappingStrategy` | Declares a strategy of parameter name formation based on Java model field name analysis.<br><br>*(By default, the `LOWERCASE_WITH_UNDERSCORED` strategy is used. Thus, by using this strategy, the `header_name` name header corresponds to the `headerName` field name.)* |
| `@AddQueryParameter` | Denotes a static query parameter that must be added to the request, created by REST client implementation. |
| `@SetQueryParameter` | Denotes a static query parameter that must be set to the request, created by REST client implementation. |
| `@RepeatQueryParameter` | Denotes the query parameter, which name needs to be repeated for each element in the list.<br><br>*(This annotation applies only to fields with the `java.util.List<?>` type.)* |
| `@PathVariable` | Denotes that a field of Java model class is a `path variable`. |
| `@RequestId` | Declares the Java model field as a field, that the RxMicro framework must used as a unique request ID and sent it to the server. |
| `@ResponseStatusCode` | Indicates to the RxMicro framework that the value of the Java model field should be used as a status code that received from the server. |
| `@ResponseBody` | Indicates to the RxMicro framework that the value of the Java model field should be used as a body that received from the server. |
| `@PartialImplementation` | Denotes an abstract class that contains a partial implementation of the annotated by this annotation REST client interface. |
| `@RestClientGeneratorConfig` | Allows to configure the process of code generation by the `RxMicro Annotation Processor` for REST clients. |

## 6.3. HTTP Request Handler Return Types

The HTTP request handler supports two categories of returned results:

- HTTP response without body;
- HTTP response with body;

### 6.3.1. Supported Return Result Types for HTTP Response without Body

The RxMicro framework supports the following return result types for an HTTP response without body:

```
@RestClient
public interface RestClientWithoutBody {

    ①

    @GET("/jse/completedFuture1")
    CompletableFuture<Void> completedFuture1();

    @GET("/jse/completedFuture2")
    CompletableFuture<Void> completedFuture2(final Request request);

    @GET("/jse/completedFuture3")
    CompletableFuture<Void> completedFuture3(final String requestParameter);

    ②

    @GET("/jse/completionStage1")
    CompletionStage<Void> completionStage1();

    @GET("/jse/completionStage2")
    CompletionStage<Void> completionStage2(final Request request);

    @GET("/jse/completionStage3")
    CompletionStage<Void> completionStage3(final String requestParameter);

    ③

    @GET("/spring-reactor/mono1")
    Mono<Void> mono1();

    @GET("/spring-reactor/mono2")
    Mono<Void> mono2(final Request request);

    @GET("/spring-reactor/mono3")
    Mono<Void> mono3(final String requestParameter);

    ④

    @GET("/rxjava3/completable1")
    Completable completable1();

    @GET("/rxjava3/completable2")
    Completable completable2(final Request request);

    @GET("/rxjava3/completable3")
    Completable completable3(final String requestParameter);
}
```

① The `CompletableFuture<Void>` type is recommended when using the **java.util.concurrent** library.

② Instead of the `CompletableFuture<Void>` type, can also be used the `CompletionStage<Void>` type.

③ When using the **Project Reactor** library, only the `Mono<Void>` type can be used.

④ When using the **RxJava** library, only the `Completable` type can be used.

All the above mentioned HTTP request handlers shouldn't throw any exceptions:

```
@InitMocks
@RxMicroComponentTest(RestClientWithoutBody.class)
final class RestClientWithoutBodyTest {

    private RestClientWithoutBody restClient;

    @Mock(answer = RETURNS_DEEP_STUBS)
    @Alternative
    private HttpClientFactory httpClientFactory;

    static Stream<Consumer<RestClientWithoutBody>> clientMethodsProvider() {
        return Stream.of(
                client -> client.completedFuture1().join(),
                client -> client.completedFuture2(new Request("param")).join(),
                client -> client.completedFuture3("param").join(),

                client -> client.completionStage1().toCompletableFuture().join(),
                client -> client.completionStage2(
                        new Request("param")).toCompletableFuture().join(),
                client ->
client.completionStage3("param").toCompletableFuture().join(),

                client -> client.mono1().block(),
                client -> client.mono2(new Request("param")).block(),
                client -> client.mono3("param").block(),

                client -> client.completable1().blockingAwait(),
                client -> client.completable2(new Request("param")).blockingAwait(),
                client -> client.completable3("param").blockingAwait()
        );
    }

    private void prepare() {
        prepareHttpClientMock(
                httpClientFactory,
                new HttpRequestMock.Builder().setAnyRequest().build(),
                true
        );
    }

    @ParameterizedTest
    @MethodSource("clientMethodsProvider")
    @BeforeThisTest(method = "prepare")
    void Should_be_invoked_successfully(final Consumer<RestClientWithoutBody>
clientMethod) {
        assertDoesNotThrow(() -> clientMethod.accept(restClient));
    }
}
```

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-client/rest-client-handlers`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 6.3.2. Supported Return Result Types for HTTP Response with Body

The RxMicro framework supports the following return result types for an HTTP response with body:

```
@RestClient
public interface RestClientWithBody {

    ①

    @GET("/jse/completedFuture1")
    CompletableFuture<Response> completedFuture1();

    @GET("/jse/completedFuture2")
    CompletableFuture<Response> completedFuture2(final Request request);

    @GET("/jse/completedFuture3")
    CompletableFuture<Response> completedFuture3(final String requestParameter);

    ②

    @GET("/jse/completionStage1")
    CompletionStage<Response> completionStage1();

    @GET("/jse/completionStage2")
    CompletionStage<Response> completionStage2(final Request request);

    @GET("/jse/completionStage3")
    CompletionStage<Response> completionStage3(final String requestParameter);

    ③

    @GET("/spring-reactor/mono1")
    Mono<Response> mono1();

    @GET("/spring-reactor/mono2")
    Mono<Response> mono2(final Request request);

    @GET("/spring-reactor/mono3")
    Mono<Response> mono3(final String requestParameter);

    ④

    @GET("/rxjava3/single1")
    Single<Response> single1();

    @GET("/rxjava3/single2")
    Single<Response> single2(final Request request);

    @GET("/rxjava3/single3")
    Single<Response> single3(final String requestParameter);
}
```

① The `CompletableFuture<MODEL>` type is recommended when using the **java.util.concurrent** library.

② Instead of the `CompletableFuture<MODEL>` type, can also be used the `CompletionStage<MODEL>` type.

③ When using the **Project Reactor** library, only the `Mono<MODEL>` type can be used.

④ When using the **RxJava** library, only the `Single<MODEL>` type can be used.

**Note that the reactive types must be parameterized by the HTTP response model class!**

All of the above mentioned handlers return the `Response` class instance with the `Hello World!` message:

```
@InitMocks
@RxMicroComponentTest(RestClientWithBody.class)
final class RestClientWithBodyTest {

    private RestClientWithBody restClient;

    @Mock(answer = RETURNS_DEEP_STUBS)
    @Alternative
    private HttpClientFactory httpClientFactory;

    static Stream<Function<RestClientWithBody, Response>> clientMethodsProvider() {
        return Stream.of(
                client -> client.completedFuture1().join(),
                client -> client.completedFuture2(new Request("param")).join(),
                client -> client.completedFuture3("param").join(),

                client -> client.completionStage1().toCompletableFuture().join(),
                client -> client.completionStage2(
                        new Request("param")).toCompletableFuture().join(),
                client ->
client.completionStage3("param").toCompletableFuture().join(),

                client -> client.mono1().block(),
                client -> client.mono2(new Request("param")).block(),
                client -> client.mono3("param").block(),

                client -> client.single1().blockingGet(),
                client -> client.single2(new Request("param")).blockingGet(),
                client -> client.single3("param").blockingGet()
        );
    }

    private void prepare() {
        prepareHttpClientMock(
                httpClientFactory,
                new HttpRequestMock.Builder().setAnyRequest().build(),
                jsonObject("message", "Hello World!"),
                true
        );
```

```
    }

    @ParameterizedTest
    @MethodSource("clientMethodsProvider")
    @BeforeThisTest(method = "prepare")
    void Should_return_message_Hello_World(final Function<RestClientWithBody,
Response> clientMethod) {
        final Response response = assertDoesNotThrow(() ->
clientMethod.apply(restClient));

        assertEquals("Hello World!", response.getMessage());
    }
}
```

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-client/rest-client-handlers`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

# 6.4. HTTP Headers Handling

## 6.4.1. Basic Rules

The RxMicro framework supports HTTP headers in request and response models:

```
①
@HeaderMappingStrategy
public final class Request {

    ②
    @Header
    String endpointVersion;

    ③
    @Header("UseProxy")
    Boolean useProxy;

    public String getEndpointVersion() {
        return endpointVersion;
    }

    public Boolean getUseProxy() {
        return useProxy;
    }
}
```

```
①
@HeaderMappingStrategy
public final class Response {

    ②
    @Header
    final String endpointVersion;

    ③
    @Header("UseProxy")
    final Boolean useProxy;

    public Response(final Request request) {
        this.endpointVersion = request.getEndpointVersion();
        this.useProxy = request.getUseProxy();
    }

    public Response(final String endpointVersion,
                    final Boolean useProxy) {
        this.endpointVersion = endpointVersion;
        this.useProxy = useProxy;
    }
}
```

① The `@HeaderMappingStrategy` annotation allows setting common conversion rules for all header names from HTTP to Java format and vice versa in the current model class.
*(By default, the `CAPITALIZE_WITH_HYPHEN` strategy is used. The field name is used as the basic name, and then, following the rules of the specified strategy, the HTTP header name is generated.)*

② In order to declare a model field as the HTTP header field, it is necessary to use the `@Header` annotation.

③ Using the `@Header` annotation, it is possible to specify the HTTP header name, which does not correspond to the used strategy declared by the `@HeaderMappingStrategy` annotation.

The RxMicro framework uses the following algorithm to define the HTTP header name for the specified model field:

1. If the field is annotated by the `@Header` annotation with an explicit indication of the HTTP header name, the specified name is used;
2. If no HTTP header name is specified in the `@Header` annotation, the RxMicro framework checks for the `@HeaderMappingStrategy` annotation above the model class;
3. If the model class is annotated by the `@HeaderMappingStrategy` annotation, then the specified naming strategy is used.
   *(The field name is used as the basic name, and then, following the rules of the specified strategy, the HTTP header name is generated.)*
4. If the `@HeaderMappingStrategy` annotation is missing, the model class field name is used as the HTTP header name.

After creating model classes, it is necessary to create a request handler that uses the following models:

```
@RestClient
public interface SimpleUsageRestClient {

    @GET("/get1")
    CompletableFuture<Response> get1(final Request request); ①


    ②
    @GET("/get2")
    ③
    @HeaderMappingStrategy
    CompletableFuture<Response> get2(final @Header String endpointVersion,        ④
                                     final @Header("UseProxy") Boolean useProxy); ⑤
}
```

① If a separate model class has been created for an HTTP request, then this class must be passed by the method parameter that handles the HTTP request.

② Besides supporting HTTP request model classes, the RxMicro framework supports request handlers that accept HTTP headers as method parameters.

③ To define a common naming strategy for all HTTP headers which are passed as method parameters, the request handler must be annotated by the `@HeaderMappingStrategy` annotation.

④ To declare a method parameter as an HTTP header field, use the `@Header` annotation.

⑤ Using the `@Header` annotation, it is possible to specify an HTTP header name that does not correspond to the strategy used, declared by the `@HeaderMappingStrategy` annotation.

The RxMicro framework recommends for request handlers that depend on 3 or more HTTP headers to create separate classes of request model.

**Upon implementation of HTTP headers directly into the handler, the code becomes hard to read!**

Despite the different approaches to HTTP header handling support, as to the message generated by the HTTP protocol, the two above-mentioned handlers are absolutely equal:

```
@InitMocks
@RxMicroComponentTest(SimpleUsageRestClient.class)
final class SimpleUsageRestClientTest {

    private static final String ENDPOINT_VERSION = "v1";

    private static final Boolean USE_PROXY = true;

    private SimpleUsageRestClient restClient;
```

```
    @Mock(answer = RETURNS_DEEP_STUBS)
    @Alternative
    private HttpClientFactory httpClientFactory;

    static Stream<Function<SimpleUsageRestClient, Response>>
restClientExecutableProvider() {
        return Stream.of(
                client -> client.get1(new Request(ENDPOINT_VERSION,
USE_PROXY)).join(),
                client -> client.get2(ENDPOINT_VERSION, USE_PROXY).join()
        );
    }

    private void prepare() {
        final HttpHeaders headers = HttpHeaders.of(
                "Endpoint-Version", ENDPOINT_VERSION, ②
                "UseProxy", USE_PROXY                 ③
        );
        prepareHttpClientMock(
                httpClientFactory,
                new HttpRequestMock.Builder()
                        .setMethod(GET)
                        .setAnyPath()
                        .setHeaders(headers)
                        .build(),
                new HttpResponseMock.Builder()
                        .setHeaders(headers)
                        .build(),
                true
        );
    }

    @ParameterizedTest
    @MethodSource("restClientExecutableProvider")
    @BeforeThisTest(method = "prepare")
    void Should_process_HTTP_headers(
            final Function<SimpleUsageRestClient, Response> clientMethod) {
        final Response response = assertDoesNotThrow(() ->
                clientMethod.apply(restClient)); ①

        assertEquals(ENDPOINT_VERSION, response.getEndpointVersion()); ④
        assertEquals(USE_PROXY, response.getUseProxy());               ④
    }
}
```

① When performing a request to different URL Paths, the result is the same.

② The `Endpoint-Version` name corresponds to the `endpointVersion` field of the request model. *(This correspondence is formed basing on the default strategy use, which is defined by the* `@HeaderMappingStrategy` *annotation.)*

③ The `UseProxy` name corresponds to the `useProxy` field of the request model, since this name is

specified in the `@Header` annotation.

④ After executing the request in the resulting HTTP response, the `Endpoint-Version` and `UseProxy` headers are equal to `v1` and `true` respectively. *(The handler returns the same header values it received from an HTTP request.)*

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-client/rest-client-headers`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 6.4.2. Supported Data Types

The RxMicro framework supports the following Java types, which can be HTTP request model headers:

- `? extends Enum<?>`;
- `java.lang.Boolean`;
- `java.lang.Byte`;
- `java.lang.Short`;
- `java.lang.Integer`;
- `java.lang.Long`;
- `java.math.BigInteger`;
- `java.lang.Float`;
- `java.lang.Double`;
- `java.math.BigDecimal`;
- `java.lang.Character`;
- `java.lang.String`;
- `java.time.Instant`;

For floating point numbers, it is suggested to use the `java.math.BigDecimal` type instead of `java.lang.Float` or `java.lang.Double`.

Using the `java.math.BigDecimal` allows excluding errors of decimal number notation. **This is particularly important if the decimal number is used to represent currency-related data!**

and also the `java.util.List<T>`, `java.util.Set<T>` and the `java.util.Map<String, T>` type is parameterized by any of the above mentioned primitive types.

If the `java.util.Map<String, T>` type is used, then REST model can contain dynamic properties, where `keys` are the names of properties and `values` are the properties values.

### 6.4.3. Static HTTP Headers

For static header installation, the RxMicro framework provides the `@AddHeader` and `@SetHeader` annotations:

```
@RestClient
①
@SetHeader(name = "Mode", value = "Demo")
public interface StaticHeadersRestClient {

    @GET("/get1")
    CompletableFuture<Void> get1();

    @GET("/get2")
    ②
    @SetHeader(name = "Debug", value = "true")
    CompletableFuture<Void> get2();
}
```

① If the HTTP header is set for the REST client, it is added to the HTTP requests for each handler.

② If the HTTP header is set for the handler, it is added to the HTTP requests only for that handler.

In terms of the HTTP protocol, static headers do not differ from any others:

```
@InitMocks
@RxMicroComponentTest(StaticHeadersRestClient.class)
final class StaticHeadersRestClientTest {

    private StaticHeadersRestClient restClient;

    @Mock(answer = RETURNS_DEEP_STUBS)
    @Alternative
    private HttpClientFactory httpClientFactory;

    private void prepareParentHeaderOnly() {
        prepareHttpClientMock(
                httpClientFactory,
                new HttpRequestMock.Builder()
                        .setMethod(GET)
                        .setPath("/get1")
                        .setHeaders(HttpHeaders.of(
                                "Mode", "Demo"
                        ))
                        .build(),
                true
        );
    }

    @Test
```

```
    @BeforeThisTest(method = "prepareParentHeaderOnly")
    void Should_use_parent_header_only() {
        assertDoesNotThrow(() -> restClient.get1().join());
    }

    private void prepareParentAndChildHeaders() {
        prepareHttpClientMock(
                httpClientFactory,
                new HttpRequestMock.Builder()
                        .setMethod(GET)
                        .setPath("/get2")
                        .setHeaders(HttpHeaders.of(
                                "Mode", "Demo",
                                "Debug", "true"
                        ))
                        .build(),
                true
        );
    }

    @Test
    @BeforeThisTest(method = "prepareParentAndChildHeaders")
    void Should_use_parent_and_child_headers() {
        assertDoesNotThrow(() -> restClient.get2().join());
    }
}
```

In order to understand the differences between the `@AddHeader` and `@SetHeader` annotations, check out the following example:

- `ComplexStaticHeadersRestClient.java`
- `ComplexStaticHeadersRestClientTest.java`

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-client/rest-client-headers`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 6.4.4. Repeating HTTP Headers

If the HTTP header of an HTTP request is a list of values, the list elements are transferred by default via the HTTP protocol as a string separated by the `|` symbol.

If You want the HTTP header to be repeated for each value, You need to use the `@RepeatHeader` annotation:

```
@RestClient
public interface RepeatingHeadersRestClient {

    @PUT("/")
    @HeaderMappingStrategy
    CompletableFuture<Void> put(@Header List<Status> singleHeader,
                                @Header @RepeatHeader List<Status> repeatingHeader);
}
```

**The use of the `@RepeatHeader` annotation is supported for request models only!**

For response models it makes no sense, because the RxMicro framework converts any of the supported formats into a list of values.

As a result of converting the Java model to HTTP response, the result will be as follows:

```
@InitMocks
@RxMicroComponentTest(RepeatingHeadersRestClient.class)
final class RepeatingHeadersRestClientTest {

    private RepeatingHeadersRestClient restClient;

    @Mock(answer = RETURNS_DEEP_STUBS)
    @Alternative
    private HttpClientFactory httpClientFactory;

    private void prepare() {
        prepareHttpClientMock(
                httpClientFactory,
                new HttpRequestMock.Builder()
                        .setMethod(PUT)
                        .setPath("/")
                        .setHeaders(HttpHeaders.of(
                                "Single-Header", "created|approved|rejected", ①
                                "Repeating-Header", "created",
                                "Repeating-Header", "approved",               ②
                                "Repeating-Header", "rejected"
                        ))
                        .build(),
                true
        );
    }

    @Test
    @BeforeThisTest(method = "prepare")
    void Should_support_repeating_headers() {
        final List<Status> headers = List.of(created, approved, rejected);
        assertDoesNotThrow(() -> restClient.put(headers, headers).join());
    }
}
```

① By default, HTTP header list elements are transferred via the HTTP protocol as a string separated by the `|` symbol.

② If the field is annotated by the `@RepeatHeader` annotation, then the header is repeated for each element of the list.

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-client/rest-client-headers

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

# 6.5. HTTP Parameters Handling

## 6.5.1. Basic Rules

The RxMicro framework supports HTTP parameters in request and response models:

```
①
@ParameterMappingStrategy
public final class Request {

    ②
    String endpointVersion;

    ③
    @Parameter("use-Proxy")
    Boolean useProxy;

    public String getEndpointVersion() {
        return endpointVersion;
    }

    public Boolean getUseProxy() {
        return useProxy;
    }
}
```

```
①
@ParameterMappingStrategy
public final class Response {

    ②
    final String endpointVersion;

    ③
    @Parameter("use-Proxy")
    final Boolean useProxy;

    public Response(final Request request) {
        this.endpointVersion = request.getEndpointVersion();
        this.useProxy = request.getUseProxy();
    }

    public Response(final String endpointVersion,
                    final Boolean useProxy) {
        this.endpointVersion = endpointVersion;
        this.useProxy = useProxy;
    }
}
```

① The `@ParameterMappingStrategy` annotation allows setting common conversion rules for all parameter names from HTTP to Java format and vice versa in the current model class. *(By default, the* `LOWERCASE_WITH_UNDERSCORED` *strategy is used. The field name is used as the basic name, and then, following the rules of the specified strategy, the HTTP parameter name is generated.)*

② In order to declare a model field as the HTTP parameter field, it is necessary to use the `@Parameter` annotation. *(Unlike the* `@Header` *annotation, the* `@Parameter` *annotation is optional. Thus, if the model field is not annotated by any annotation, then by default it is assumed that there is the* `@Parameter` *annotation above the model field.)*

③ Using the `@Parameter` annotation, it is possible to specify the HTTP parameter name, which does not correspond to the used strategy declared by the `@ParameterMappingStrategy` annotation.

The RxMicro framework uses the following algorithm to define the HTTP parameter name for the specified model field:

1. If the field is annotated by the `@Parameter` annotation with an explicit indication of the HTTP parameter name, the specified name is used;
2. If no HTTP parameter name is specified in the `@Parameter` annotation, the RxMicro framework checks for the `@ParameterMappingStrategy` annotation above the model class;
3. If the model class is annotated by the `@ParameterMappingStrategy` annotation, then the specified naming strategy is used. *(The field name is used as the basic name, and then, following the rules of the specified strategy, the HTTP parameter name is generated.)*
4. If the `@ParameterMappingStrategy` annotation is missing, the model class field name is used as the HTTP parameter name.

**Unlike the `@Header` annotation, the `@Parameter` annotation is optional!**

Thus, if the model field is not annotated by any annotation, then **by default it is assumed that there is the `@Parameter` annotation above the model field!**

After creating model classes, it is necessary to create an HTTP request handler that uses the following models:

```
@RestClient
public interface SimpleUsageRestClient {

    @GET("/get1")
    CompletableFuture<Response> get1(final Request request); ①

    ②
    @GET("/get2")
    ③
    @ParameterMappingStrategy
    CompletableFuture<Response> get2(final String endpointVersion,        ④
                                     final @Parameter("use-Proxy") Boolean useProxy);
⑤
}
```

① If a separate model class has been created for an HTTP request, then this class must be passed by the method parameter that handles the HTTP request.

② Besides supporting HTTP request model classes, the RxMicro framework supports request handlers that accept HTTP request parameters as method parameters.

③ To define a common naming strategy for all HTTP parameters which are passed as method parameters, the request handler must be annotated by the `@ParameterMappingStrategy` annotation.

④ To declare a method parameter as an HTTP parameter field, You do not need to use any additional annotations.

⑤ Using the `@Parameter` annotation, it is possible to specify an HTTP parameter name that does not correspond to the strategy used, declared by the `@ParameterMappingStrategy` annotation.

**Note that the RxMicro framework does not distinguish whether HTTP request parameters will be transferred to the handler using the start line or HTTP body!**

The RxMicro framework recommends for request handlers that depend on 3 or more HTTP parameters to create separate classes of request model.

**Upon implementation of HTTP parameters directly into the handler, the code becomes hard to read!**

Despite the different approaches to HTTP parameter handling support, as to the message generated by the HTTP protocol, the two above-mentioned handlers are absolutely equal:

```
@InitMocks
@RxMicroComponentTest(SimpleUsageRestClient.class)
final class SimpleUsageRestClientTest {

    private static final String ENDPOINT_VERSION = "v1";

    private static final Boolean USE_PROXY = true;

    private static final QueryParams QUERY_PARAMETERS = QueryParams.of(
            "endpoint_version", ENDPOINT_VERSION,
            "use-Proxy", USE_PROXY
    );

    private static final Object BODY = jsonObject(
            "endpoint_version", ENDPOINT_VERSION,
            "use-Proxy", USE_PROXY
    );

    private SimpleUsageRestClient restClient;

    @Mock(answer = RETURNS_DEEP_STUBS)
    @Alternative
```

```
    private HttpClientFactory httpClientFactory;

    private void prepare1() {
        prepareHttpClientMock(
                httpClientFactory,
                new HttpRequestMock.Builder()
                        .setMethod(HttpMethod.GET)
                        .setPath("/get1")
                        .setQueryParameters(QUERY_PARAMETERS)
                        .build(),
                BODY,
                true
        );
    }

    @Test
    @BeforeThisTest(method = "prepare1")
    void Should_use_handler_with_request_class() {
        final Response response =
                assertDoesNotThrow(() ->
                        restClient.get1(new Request(ENDPOINT_VERSION,
USE_PROXY)).join());

        assertEquals(ENDPOINT_VERSION, response.getEndpointVersion());
        assertEquals(USE_PROXY, response.getUseProxy());
    }

    private void prepare2() {
        prepareHttpClientMock(
                httpClientFactory,
                new HttpRequestMock.Builder()
                        .setMethod(HttpMethod.GET)
                        .setPath("/get2")
                        .setQueryParameters(QUERY_PARAMETERS)
                        .build(),
                BODY,
                true
        );
    }

    @Test
    @BeforeThisTest(method = "prepare2")
    void Should_use_handler_with_request_params() {
        final Response response =
                assertDoesNotThrow(() ->
                        restClient.get2(ENDPOINT_VERSION, USE_PROXY).join());

        assertEquals(ENDPOINT_VERSION, response.getEndpointVersion());
        assertEquals(USE_PROXY, response.getUseProxy());
    }
}
```

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-client/rest-client-params`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 6.5.2. Supported Data Types

The RxMicro framework supports the following Java types, which can be HTTP request model parameters:

- `? extends Enum<?>`;
- `java.lang.Boolean`;
- `java.lang.Byte`;
- `java.lang.Short`;
- `java.lang.Integer`;
- `java.lang.Long`;
- `java.math.BigInteger`;
- `java.lang.Float`;
- `java.lang.Double`;
- `java.math.BigDecimal`;
- `java.lang.Character`;
- `java.lang.String`;
- `java.time.Instant`;
- `? extends Object`;
- `java.util.List<? extends Object>`;
- `java.util.Set<? extends Object>`;
- `java.util.Map<java.lang.String, ? extends Object>`;

For floating point numbers, it is suggested to use the `java.math.BigDecimal` type instead of `java.lang.Float` or `java.lang.Double`.

Using the `java.math.BigDecimal` allows excluding errors of decimal number notation. **This is particularly important if the decimal number is used to represent currency-related data!**

and also the `java.util.List<T>`, `java.util.Set<T>` and the `java.util.Map<String, T>` type is parameterized by any of the above mentioned primitive types.

If the `java.util.Map<String, T>` type is used, then REST model can contain dynamic properties, where `keys` are the names of properties and `values` are the properties values.

### 6.5.3. Complex Model Support

Unlike HTTP headers, HTTP parameters can be transferred in the request body.

Therefore, besides primitives the RxMicro framework also supports nested JSON objects and arrays.

Therefore, the list of supported types includes the following types:

- `? extends Object` - custom class of the nested Java model;
- `java.util.List<? extends Object>` - list of the nested Java model custom classes;
- `java.util.Set<? extends Object>` - set of the nested Java model custom classes;
- `java.util.Map<String, ? extends Object>` - map of the nested Java model custom classes;

Nested Model Example:

```
@ParameterMappingStrategy
public final class NestedModel {

    String stringParameter;

    BigDecimal bigDecimalParameter;

    Instant instantParameter;

    public NestedModel(final String stringParameter,
                       final BigDecimal bigDecimalParameter,
                       final Instant instantParameter) {
        this.stringParameter = stringParameter;
        this.bigDecimalParameter = bigDecimalParameter;
        this.instantParameter = instantParameter;
    }

    public NestedModel() {
    }

    @Override
    public int hashCode() {
        return Objects.hash(stringParameter, bigDecimalParameter, instantParameter);
    }

    @Override
    public boolean equals(final Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        final NestedModel that = (NestedModel) o;
        return stringParameter.equals(that.stringParameter) &&
                bigDecimalParameter.equals(that.bigDecimalParameter) &&
                instantParameter.equals(that.instantParameter);
    }
}
```

As well as examples of more complex Java models that use a nested model:

```
@ParameterMappingStrategy
public final class ComplexRequest {

    final Integer integerParameter; ①

    final Status enumParameter; ①

    final List<Status> enumsParameter; ②

    final NestedModel nestedModelParameter; ③

    final List<NestedModel> nestedModelsParameter; ④

    public ComplexRequest(final Integer integerParameter,
                          final Status enumParameter,
                          final List<Status> enumsParameter,
                          final NestedModel nestedModelParameter,
                          final List<NestedModel> nestedModelsParameter) {
        this.integerParameter = integerParameter;
        this.enumParameter = enumParameter;
        this.enumsParameter = enumsParameter;
        this.nestedModelParameter = nestedModelParameter;
        this.nestedModelsParameter = nestedModelsParameter;
    }
}
```

```
@ParameterMappingStrategy
public final class ComplexResponse {

    Integer integerParameter; ①

    Status enumParameter; ①

    List<Status> enumsParameter; ②

    NestedModel nestedModelParameter; ③

    List<NestedModel> nestedModelsParameter; ④

    public Integer getIntegerParameter() {
        return integerParameter;
    }

    public Status getEnumParameter() {
        return enumParameter;
    }

    public List<Status> getEnumsParameter() {
        return enumsParameter;
    }

    public NestedModel getNestedModelParameter() {
        return nestedModelParameter;
    }

    public List<NestedModel> getNestedModelsParameter() {
        return nestedModelsParameter;
    }
}
```

① Primitive JSON type field.

② Primitive JSON array type.

③ Nested JSON object.

④ JSON array of JSON nested objects.

After creating model classes, it is necessary to create a request handler that uses the following models:

```
@RestClient
public interface ComplexModelRestClient {

    @POST("/")
    CompletableFuture<ComplexResponse> post(ComplexRequest request);
}
```

Since the request is transferred in an HTTP body, the sent and received JSON objects must be the same:

```
@InitMocks
@RxMicroComponentTest(ComplexModelRestClient.class)
final class ComplexModelRestClientTest {

    private final Integer integerParameter = 1;

    private final Status enumParameter = created;

    private final List<Status> enumsParameter = List.of(created, approved);

    private final NestedModel nestedModelParameter =
            new NestedModel("test", new BigDecimal("3.14"), Instant.now());

    private final List<NestedModel> nestedModelsParameter = List.of(
            new NestedModel("test1", new BigDecimal("1.1"), Instant.now()),
            new NestedModel("test2", new BigDecimal("1.2"), Instant.now())
    );

    private ComplexModelRestClient restClient;

    @Mock(answer = RETURNS_DEEP_STUBS)
    @Alternative
    private HttpClientFactory httpClientFactory;

    void prepare() {
        prepareHttpClientMock(
                httpClientFactory,
                new HttpRequestMock.Builder()
                        .setMethod(POST)
                        .setPath("/")
                        .setAnyBody()
                        .build(),
                new HttpResponseMock.Builder()
                        .setReturnRequestBody()
                        .build(),
                true
        );
    }
```

```
    @Test
    @BeforeThisTest(method = "prepare")
    void Should_support_complex_requests_and_responses() {
        final ComplexResponse response = assertDoesNotThrow(() ->
                restClient.post(new ComplexRequest(
                        integerParameter,
                        enumParameter,
                        enumsParameter,
                        nestedModelParameter,
                        nestedModelsParameter
                )).join());

        assertEquals(integerParameter, response.getIntegerParameter());
        assertEquals(enumParameter, response.getEnumParameter());
        assertEquals(enumsParameter, response.getEnumsParameter());
        assertEquals(nestedModelParameter, response.getNestedModelParameter());
        assertEquals(nestedModelsParameter, response.getNestedModelsParameter());
    }
}
```

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-client/rest-client-params`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 6.5.4. Static Query Parameters

For setting static query parameters, the RxMicro framework provides the `@AddQueryParameter` and `@SetQueryParameter` annotations:

```
@RestClient
①
@SetQueryParameter(name = "mode", value = "demo")
public interface StaticQueryParametersRestClient {

    @GET("/get1")
    CompletableFuture<Void> get1();

    @GET("/get2")
    ②
    @SetQueryParameter(name = "debug", value = "true")
    CompletableFuture<Void> get2();
}
```

① If the static query parameter is set for the REST client, it is added to the HTTP requests for each handler.

② If the static query parameter is set for the handler, it is added to the HTTP requests only for that handler.

In terms of the HTTP protocol, static query parameters do not differ from any standard parameter:

```
@InitMocks
@RxMicroComponentTest(StaticQueryParametersRestClient.class)
final class StaticQueryParametersRestClientTest {

    private StaticQueryParametersRestClient restClient;

    @Mock(answer = RETURNS_DEEP_STUBS)
    @Alternative
    private HttpClientFactory httpClientFactory;

    private void prepareParentParamOnly() {
        prepareHttpClientMock(
                httpClientFactory,
                new HttpRequestMock.Builder()
                        .setMethod(GET)
                        .setPath("/get1")
                        .setQueryParameters(QueryParams.of(
                                "mode", "demo"
                        ))
                        .build(),
                true
        );
    }
```

```
    @Test
    @BeforeThisTest(method = "prepareParentParamOnly")
    void Should_use_parent_param_only() {
        assertDoesNotThrow(() -> restClient.get1().join());
    }

    private void prepareParentAndChildParams() {
        prepareHttpClientMock(
                httpClientFactory,
                new HttpRequestMock.Builder()
                        .setMethod(GET)
                        .setPath("/get2")
                        .setQueryParameters(QueryParams.of(
                                "debug", true,
                                "mode", "demo"
                        ))
                        .build(),
                true
        );
    }

    @Test
    @BeforeThisTest(method = "prepareParentAndChildParams")
    void Should_use_parent_and_child_params() {
        assertDoesNotThrow(() -> restClient.get2().join());
    }
}
```

In order to understand the differences between the `@AddQueryParameter` and `@SetQueryParameter` annotations, check out the following example:

- `ComplexStaticQueryParametersRestClient.java`
- `ComplexStaticQueryParametersRestClientTest.java`

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-client/rest-client-params`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 6.5.5. Repeating Query Parameters

If the static query parameter of an HTTP request is a list of values, the list elements are transferred by default via the HTTP protocol as a string separated by the `|` symbol.

If You want the static query parameter to be repeated for each value, You need to use the `@RepeatQueryParameter` annotation:

```
@RestClient
public interface RepeatingQueryParamsRestClient {

    @PUT(value = "/", httpBody = false)
    @ParameterMappingStrategy
    CompletableFuture<Void> put(List<Status> singleHeader,
                                @RepeatQueryParameter List<Status> repeatingHeader);
}
```

As a result of converting the Java model to HTTP response, the result will be as follows:

```
@InitMocks
@RxMicroComponentTest(RepeatingQueryParamsRestClient.class)
final class RepeatingQueryParamsRestClientTest {

    private RepeatingQueryParamsRestClient restClient;

    @Mock(answer = RETURNS_DEEP_STUBS)
    @Alternative
    private HttpClientFactory httpClientFactory;

    private void prepare() {
        prepareHttpClientMock(
                httpClientFactory,
                new HttpRequestMock.Builder()
                        .setMethod(PUT)
                        .setPath("/")
                        .setQueryParameters(QueryParams.of(
                                "single_header", "created|approved|rejected", ①
                                "repeating_header", "created",
                                "repeating_header", "approved",               ②
                                "repeating_header", "rejected"
                        ))
                        .build(),
                true
        );
    }

    @Test
    @BeforeThisTest(method = "prepare")
    void Should_support_repeating_headers() {
        final List<Status> headers = List.of(created, approved, rejected);
        assertDoesNotThrow(() -> restClient.put(headers, headers).join());
    }
}
```

① By default, static query parameter list elements are transferred via the HTTP protocol as a string separated by the `|` symbol.

② If the field is annotated by the `@RepeatQueryParameter` annotation, then the static query parameter is repeated for each element of the list.

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-client/rest-client-params

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

## 6.6. The `path variables` Support

### 6.6.1. Basic Rules

The RxMicro framework supports `path variables` in request models:

```
public final class Request {

    ①
    @PathVariable
    String category;

    ②
    @PathVariable("class")
    String type;

    @PathVariable
    String subType;

    public String getCategory() {
        return category;
    }

    public String getType() {
        return type;
    }

    public String getSubType() {
        return subType;
    }
}
```

① In order to declare a model field as the `path variable`, it is necessary to use the `@PathVariable` annotation.

② Using the `@PathVariable` annotation, it is possible to specify the `path variable` name.
*(If no name is specified, the model field name is used as the `path variable` name.)*

Unlike HTTP headers and parameters that are available also on the client side, `path variables` are available **only** within the HTTP request handlers.

**So for simplicity it is recommended to always use the model field name as the `path variable` name!**

After creating model classes, it is necessary to create an HTTP request handler that uses the following model:

```
@RestClient
public interface RESTClient {

    ①
    @GET("/${category}/${class}-${subType}")
    CompletableFuture<Void> consume(final Request request);

    ①
    @GET("/${category}/${class}-${subType}")
    CompletableFuture<Void> consume(@PathVariable String category,       ②
                                    @PathVariable("class") String type,  ③
                                    @PathVariable String subType);
}
```

① When using `path variables` in request models, be sure to use **all** `path variables` in the URL Path. *(Based on the analysis of the URL Path, and considering all `path variables`, the RxMicro framework generates the final URL to which the HTTP request will be sent.)*

② In order to declare a method parameter as `path variable`, You must use the `@PathVariable` annotation.

③ Using the `@PathVariable` annotation it is possible to specify the `path variable` name, which does not match the name of the method parameter.

The RxMicro framework recommends for request handlers that depend on 3 or more `path variables` to create separate classes of request model.

**Upon implementation of `path variables` directly into the handler, the code becomes hard to read!**

Despite the different approaches to `path variables` handling support, in terms of the final URL the two above-mentioned handlers are absolutely equal:

```
@InitMocks
@RxMicroComponentTest(RESTClient.class)
final class RESTClientTest {

    private RESTClient restClient;

    @Mock(answer = RETURNS_DEEP_STUBS)
    @Alternative
    private HttpClientFactory httpClientFactory;

    static Stream<Consumer<RESTClient>> clientMethodsProvider() {
        return Stream.of(
                client -> client.consume(new Request("CATEGORY", "TYPE", "SUB-
TYPE")).join(),
                client -> client.consume("CATEGORY", "TYPE", "SUB-TYPE").join()
        );
    }

    private void prepare() {
        prepareHttpClientMock(
                httpClientFactory,
                new HttpRequestMock.Builder()
                        .setMethod(HttpMethod.GET)
                        .setPath("/CATEGORY/TYPE-SUB-TYPE") ①
                        .build(),
                true
        );
    }

    @ParameterizedTest
    @MethodSource("clientMethodsProvider")
    @BeforeThisTest(method = "prepare")
    void Should_return_message_Hello_World(final Consumer<RESTClient> clientMethod) {
        assertDoesNotThrow(() -> clientMethod.accept(restClient));
    }
}
```

① As a result of the HTTP request handler execution, the RxMicro framework generates the same URL: `/CATEGORY/TYPE-SUB-TYPE`

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-client/rest-client-path-variables

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 6.6.2. Supported Data Types

The RxMicro framework supports the following Java types, which can be `path variables` of the request model:

- `? extends Enum<?>`;
- `java.lang.Boolean`;
- `java.lang.Byte`;
- `java.lang.Short`;
- `java.lang.Integer`;
- `java.lang.Long`;
- `java.math.BigInteger`;
- `java.lang.Float`;
- `java.lang.Double`;
- `java.math.BigDecimal`;
- `java.lang.Character`;
- `java.lang.String`;
- `java.time.Instant`;

For floating point numbers, it is suggested to use the `java.math.BigDecimal` type instead of `java.lang.Float` or `java.lang.Double`.

Using the `java.math.BigDecimal` allows excluding errors of decimal number notation. **This is particularly important if the decimal number is used to represent currency-related data!**

# 6.7. Support of Internal Data Types

## 6.7.1. Basic Rules

The RxMicro framework works with internal data. Due to this, the developer can extract additional information about the HTTP response.

To work with internal data, it is necessary to create a response model:

```
public final class Response {

    ①
    @ResponseStatusCode
    Integer status;

    ②
    HttpVersion version;

    ③
    HttpHeaders headers;

    ④
    @ResponseBody
    byte[] body;

    public Integer getStatus() {
        return status;
    }

    public HttpVersion getVersion() {
        return version;
    }

    public HttpHeaders getHeaders() {
        return headers;
    }

    public byte[] getBody() {
        return body;
    }
}
```

① To get the HTTP status code, it is necessary to use the `@ResponseStatusCode` annotation.

② To get the HTTP protocol version, it is necessary to use the `HttpVersion` type.
*(Currently only `1.0` and `1.1` HTTP versions are supported.)*

③ To get all HTTP headers, it is necessary to use the `HttpHeaders` type.

④ To get the response body content as a byte array, it is necessary to use the `@ResponseBody` annotation.

After creating model classes, it is necessary to create a REST client method that uses this model:

```
@RestClient
public interface RESTClient {

    @GET("/")
    CompletableFuture<Response> get();
}
```

After creating the REST client, let’s check that everything works:

```
@InitMocks
@RxMicroComponentTest(RESTClient.class)
final class RESTClientTest {

    private RESTClient restClient;

    @Mock(answer = RETURNS_DEEP_STUBS)
    @Alternative
    private HttpClientFactory httpClientFactory;

    @Mock
    private HttpHeaders headers;

    private void prepare() {
        prepareHttpClientMock(
                httpClientFactory,
                new HttpRequestMock.Builder()
                        .setMethod(GET)
                        .setPath("/")
                        .build(),
                new HttpResponseMock.Builder()
                        .setStatus(201)
                        .setVersion(HTTP_1_0)
                        .setHeaders(headers)
                        .setBody("<BODY>")
                        .build(),
                true
        );
    }

    @Test
    @BeforeThisTest(method = "prepare")
    void Should_support_internals() {
        final Response response = assertDoesNotThrow(() -> restClient.get().join());

        assertEquals(201, response.getStatus());
        assertEquals(HTTP_1_0, response.getVersion());
        assertSame(headers, response.getHeaders());
        assertArrayEquals("<BODY>".getBytes(UTF_8), response.getBody());

    }
}
```

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-client/rest-client-internals

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 6.7.2. Supported Internal Data Types

*Table 12. Supported internal data types for a response model.*

| Java type | RxMicro Annotation | Is the annotation required? | Description |
|---|---|---|---|
| `HttpVersion` | None | - | HTTP protocol version |
| `HttpHeaders` | None | - | HTTP header model |
| `Integer` | `@ResponseStatusCode` | **Yes** | HTTP response status code |
| `byte[]` | `@ResponseBody` | **Yes** | HTTP response body content |

Currently only `1.0` and `1.1` HTTP versions are supported.

## 6.8. Versioning of REST Clients

The RxMicro framework supports versioning of REST Clients using two strategies:

- Versioning based on HTTP header analysis with the `Api-Version` name.
- Versioning based on URL Path fragment analysis.

### 6.8.1. Versioning Based on HTTP Header Analysis

The RxMicro framework allows creating identical REST clients that differ only in version:

```
@RestClient
@Version(value = "v1", strategy = Version.Strategy.HEADER) ①
public interface OldRestClient {

    @PATCH("/patch")
    CompletableFuture<Void> update();
}
```

① REST client of the old `v1` version, using the `Version.Strategy.HEADER` strategy;

```
@RestClient
@Version(value = "v2", strategy = Version.Strategy.HEADER) ①
public interface NewRestClient {

    @PATCH("/patch")
    CompletableFuture<Void> update();
}
```

① REST client of the new `v2` version, using the `Version.Strategy.HEADER` strategy;

The correctness of HTTP request generation for these REST clients can be checked with the following tests:

```
@InitMocks
@RxMicroComponentTest(OldRestClient.class)
final class OldRestClientTest {

    private OldRestClient restClient;

    @Mock(answer = RETURNS_DEEP_STUBS)
    @Alternative
    private HttpClientFactory httpClientFactory;

    private void prepare() {
        prepareHttpClientMock(
                httpClientFactory,
                new HttpRequestMock.Builder()
                        .setMethod(HttpMethod.PATCH)
                        .setPath("/patch")
                        .setHeaders(HttpHeaders.of(
                                API_VERSION, "v1"
                        ))
                        .build(),
                true
        );
    }

    @Test
    @BeforeThisTest(method = "prepare")
    void Should_add_version_url_path() {
        assertDoesNotThrow(() -> restClient.update().join());
    }
}
```

```
@InitMocks
@RxMicroComponentTest(NewRestClient.class)
final class NewRestClientTest {

    private NewRestClient restClient;

    @Mock(answer = RETURNS_DEEP_STUBS)
    @Alternative
    private HttpClientFactory httpClientFactory;

    private void prepare() {
        prepareHttpClientMock(
                httpClientFactory,
                new HttpRequestMock.Builder()
                        .setMethod(HttpMethod.PATCH)
                        .setPath("/patch")
                        .setHeaders(HttpHeaders.of(
                                API_VERSION, "v2"
                        ))
                        .build(),
                true
        );
    }

    @Test
    @BeforeThisTest(method = "prepare")
    void Should_add_version_url_path() {
        assertDoesNotThrow(() -> restClient.update().join());
    }
}
```

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-client/rest-client-versioning-header

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 6.8.2. Versioning Based on URL Path Fragment Analysis

The RxMicro framework allows creating identical REST clients that differ only in version:

```
@RestClient
@Version(value = "v1", strategy = Version.Strategy.URL_PATH) ①
public interface OldRestClient {

    @PATCH("/patch")
    CompletableFuture<Void> update();
}
```

① REST client of the old `v1` version, using the `Version.Strategy.URL_PATH` strategy;

```
@RestClient
@Version(value = "v2", strategy = Version.Strategy.URL_PATH) ①
public interface NewRestClient {

    @PATCH("/patch")
    CompletableFuture<Void> update();
}
```

① REST client of the new `v2` version, using the `Version.Strategy.URL_PATH` strategy;

The correctness of HTTP request generation for these REST clients can be checked with the following tests:

```
@InitMocks
@RxMicroComponentTest(OldRestClient.class)
final class OldRestClientTest {

    private OldRestClient restClient;

    @Mock(answer = RETURNS_DEEP_STUBS)
    @Alternative
    private HttpClientFactory httpClientFactory;

    private void prepare() {
        prepareHttpClientMock(
                httpClientFactory,
                new HttpRequestMock.Builder()
                        .setMethod(HttpMethod.PATCH)
                        .setPath("/v1/patch")
                        .build(),
                true
        );
    }

    @Test
    @BeforeThisTest(method = "prepare")
    void Should_add_version_url_path() {
        assertDoesNotThrow(() -> restClient.update().join());
    }
}
```

```
@InitMocks
@RxMicroComponentTest(NewRestClient.class)
final class NewRestClientTest {

    private NewRestClient restClient;

    @Mock(answer = RETURNS_DEEP_STUBS)
    @Alternative
    private HttpClientFactory httpClientFactory;

    private void prepare() {
        prepareHttpClientMock(
                httpClientFactory,
                new HttpRequestMock.Builder()
                        .setMethod(HttpMethod.PATCH)
                        .setPath("/v2/patch")
                        .build(),
                true
        );
    }

    @Test
    @BeforeThisTest(method = "prepare")
    void Should_add_version_url_path() {
        assertDoesNotThrow(() -> restClient.update().join());
    }
}
```

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-client/rest-client-versioning-url-path`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

## 6.9. Base URL Path for All Handlers

To configure a base URL Path for all methods sending an HTTP request, the RxMicro framework provides the `@BaseUrlPath` annotation:

```
@RestClient
@BaseUrlPath("base-url-path")
public interface RESTClient {

    @GET("/path")
    CompletableFuture<Void> path();
}
```

```
@InitMocks
@RxMicroComponentTest(RESTClient.class)
final class RESTClientTest {

    private RESTClient restClient;

    @Mock(answer = RETURNS_DEEP_STUBS)
    @Alternative
    private HttpClientFactory httpClientFactory;

    private void prepare() {
        prepareHttpClientMock(
                httpClientFactory,
                new HttpRequestMock.Builder()
                        .setMethod(HttpMethod.GET)
                        .setPath("/base-url-path/path")
                        .build(),
                true
        );
    }

    @Test
    @BeforeThisTest(method = "prepare")
    void Should_support_base_url() {
        assertDoesNotThrow(() -> restClient.path().join());
    }
}
```

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-client/rest-client-base-url-path

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

## 6.10. Expressions

The RxMicro framework supports expressions for REST clients.

Expressions can be useful to send configuration parameters to the server.

To use expressions You need to create a configuration class:

```
public final class CustomRestClientConfig extends RestClientConfig {

    private boolean useProxy = true;

    private Mode mode = Mode.PRODUCTION;

    public boolean isUseProxy() {
        return useProxy;
    }

    public CustomRestClientConfig setUseProxy(final boolean useProxy) {
        this.useProxy = useProxy;
        return this;
    }

    public Mode getMode() {
        return mode;
    }

    public CustomRestClientConfig setMode(final Mode mode) {
        this.mode = mode;
        return this;
    }

    @Override
    public CustomRestClientConfig setSchema(final ProtocolSchema schema) {
        return (CustomRestClientConfig) super.setSchema(schema);
    }

    @Override
    public CustomRestClientConfig setHost(final String host) {
        return (CustomRestClientConfig) super.setHost(host);
    }

    @Override
    public CustomRestClientConfig setPort(final int port) {
        return (CustomRestClientConfig) super.setPort(port);
    }

    public enum Mode {

        PRODUCTION,

        TEST
    }
}
```

, which must meet the following requirements:

1. The class must be public.

2. The class must contain a public constructor without parameters.
3. The class must extend the `RestClientConfig` class.
4. To set property values, the class must contain `setters`.
   *(Only those fields, that will contain `setters`, can be initialized!)*

To attach this configuration class to a REST client, You must specify it in the `@RestClient` annotation parameter:

```
@RestClient(
        configClass = CustomRestClientConfig.class,    ①
        configNameSpace = "custom"                     ②
)
@AddHeader(name = "Use-Proxy", value = "${useProxy}")  ③
public interface RESTClient {

    @PUT("/")
    @AddHeader(name = "Debug", value = "Use-Proxy=${useProxy}, Mode=${mode}")
③
    @AddHeader(name = "Endpoint", value = "Schema=${schema}, Host=${host},
Port=${port}") ③
    CompletableFuture<Void> put();
}
```

① Attaching the configuration class to a REST client.

② The `name space` specification for this configuration file.
*(For more information on `name space`, refer to the core.pdf section)*

③ After attaching the configuration class, its `properties` can be used in expressions.

All packages with custom config classes must be exported to `rxmicro.reflection` module!

So dont't forget to add the following `opens` instruction to your `module-info.java`:

```
exports io.rxmicro.examples.rest.client.expressions to
        rxmicro.reflection;
```

The functionality of expressions can be demonstrated through the test:

```
@InitMocks
@RxMicroComponentTest(RESTClient.class)
final class RESTClientTest {

    @WithConfig("custom")
    private static final CustomRestClientConfig config =
            new CustomRestClientConfig()
                    .setHost("rxmicro.io")
                    .setPort(8443)
                    .setSchema(ProtocolSchema.HTTPS)
                    .setUseProxy(false)
                    .setMode(CustomRestClientConfig.Mode.TEST);

    private RESTClient restClient;

    @Mock(answer = RETURNS_DEEP_STUBS)
    @Alternative
    private HttpClientFactory httpClientFactory;

    private void prepare() {
        prepareHttpClientMock(
                httpClientFactory,
                new HttpRequestMock.Builder()
                        .setMethod(HttpMethod.PUT)
                        .setPath("/")
                        .setHeaders(HttpHeaders.of(
                                "Use-Proxy", "false",
                                "Debug", "Use-Proxy=false, Mode=TEST",
                                "Endpoint", "Schema=HTTPS, Host=rxmicro.io, Port=8443"
                        ))
                        .build(),
                true
        );
    }

    @Test
    @BeforeThisTest(method = "prepare")
    void Should_support_expressions() {
        assertDoesNotThrow(() -> restClient.put().join());
    }
}
```

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-client/rest-client-expressions

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

## 6.11. Request ID

The Request Id feature described at Monitoring section.

# 6.12. Partial Implementation

If the REST client generated by the `RxMicro Annotation Processor` contains errors, incorrect or non-optimized logic, the developer can use the `Partial Implementation` feature.

This feature allows You to implement HTTP request handlers for the REST client on Your own, instead of generating them by the RxMicro framework.

To activate this feature, You need to use the `@PartialImplementation` annotation, and specify an abstract class that contains a partial implementation of HTTP request handler(s) for REST client:

```
@RestClient
①
@PartialImplementation(AbstractRESTClient.class)
public interface RESTClient {

    @GET("/")
    CompletableFuture<Void> generatedMethod();

    CompletableFuture<Void> userDefinedMethod();
}
```

① Using the `@PartialImplementation` annotation, the `AbstractRESTClient` class is specified.

An `AbstractRESTClient` contains the following content:

```
abstract class AbstractRESTClient extends AbstractRestClient implements RESTClient {

    @Override
    public final CompletableFuture<Void> userDefinedMethod() {
        return CompletableFuture.completedFuture(null);
    }
}
```

An abstract class that contains a partial implementation must meet the following requirements:

1. The class must be an `abstract` one.
2. The class must extend the `AbstractRestClient` one.
3. The class must implement the REST client interface.
4. The class must contain an implementation of all methods that are not generated automatically.

In terms of infrastructure, the HTTP request handlers generated and defined by the developer for REST client do not differ:

```
@InitMocks
@RxMicroComponentTest(RESTClient.class)
final class RESTClientTest {

    private RESTClient restClient;

    @Mock(answer = RETURNS_DEEP_STUBS)
    @Alternative
    private HttpClientFactory httpClientFactory;

    private void prepareGeneratedMethod() {
        prepareHttpClientMock(
                httpClientFactory,
                new HttpRequestMock.Builder()
                        .setMethod(GET)
                        .setPath("/")
                        .build(),
                true
        );
    }

    @Test
    @BeforeThisTest(method = "prepareGeneratedMethod")
    void Should_invoke_generated_method() {
        assertDoesNotThrow(() -> restClient.generatedMethod().join());
    }

    private void prepareUserDefinedMethod() {
        prepareHttpClientMock(
                httpClientFactory,
                new HttpRequestMock.Builder()
                        .setAnyRequest()
                        .build(),
                true
        );
    }

    @Test
    @BeforeThisTest(method = "prepareUserDefinedMethod")
    void Should_invoke_user_defined_method() {
        assertDoesNotThrow(() -> restClient.userDefinedMethod().join());

        final HttpClient httpClient = getPreparedHttpClientMock();
        verify(httpClient, never()).sendAsync(anyString(), anyString(), any());
        verify(httpClient, never()).sendAsync(anyString(), anyString(), any(), any());
    }
}
```

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-client/rest-client-partial-implementation`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

## 6.13. Configuring the Code Generation Process

The RxMicro framework provides an option to configure the code generation process for REST client. For this purpose, it is necessary to use the `@RestClientGeneratorConfig` annotation, that annotates the `module-info.java` module descriptor:

```
import io.rxmicro.rest.client.RestClientGeneratorConfig;
import io.rxmicro.rest.model.ClientExchangeFormatModule;
import io.rxmicro.rest.model.GenerateOption;

@RestClientGeneratorConfig(
        exchangeFormat = ClientExchangeFormatModule.AUTO_DETECT,                    ①
        generateRequestValidators = GenerateOption.DISABLED,                        ②
        generateResponseValidators = GenerateOption.DISABLED,                       ③
        requestValidationMode =
                RestClientGeneratorConfig.RequestValidationMode.RETURN_ERROR_SIGNAL ④
)
module rest.client.generator {
    requires rxmicro.rest.client;
}
```

① The `exchangeFormat` parameter allows You to specify a format for message exchange with a server.
*(By default, it is used the message exchange format which added to the `module-info.java` descriptor. If several modules supporting the message exchange format are added to the `module-info.java` descriptor, then using the `exchangeFormat`, You need to specify which module should be used for REST clients.)*

② The `generateRequestValidators` parameter allows enabling/disabling the option of generating HTTP request validators for all REST client methods in the project.
*(The `DISABLED` value means that validators won't be generated by the `RxMicro Annotation Processor`.)*

③ The `generateResponseValidators` parameter allows enabling/disabling the option of generating HTTP response validators for all REST client methods in the project.
*(The `AUTO_DETECT` value means that validators will be generated only if the developer adds the `rxmicro.validation` module to the `module-info.java` descriptor.)*

④ The `requestValidationMode` parameter specifies how the HTTP request parameters should be checked.

If the `requestValidationMode` parameter is equal to the `RETURN_ERROR_SIGNAL`, then error handling should be performed in reactive style:

```
restClient.put("not_valid_email")
        .exceptionally(throwable -> {
            // do something with ValidationException
            return null;
        });
```

If the `requestValidationMode` parameter is equal to the `THROW_EXCEPTION`, then it is necessary to catch the exception

```
try {
    restClient.put("not_valid_email");
} catch (final ValidationException e) {
    // do something with ValidationException
}
```

The RxMicro team strong recommends using the `AUTO_DETECT` generate option always! (This is default value for `@RestClientGeneratorConfig` annotation).

*(HTTP request validation can be useful for identifying errors in business task implementation algorithms. For example, instead of returning an incorrect request model to a server, the REST client will throw an error. This approach increases the speed of error search and debugging of the source code that performs the business task.)*

**FYI:** By default the request validators are generated but not invoked! To activate the validation of requests it is necessary to set `enableAdditionalValidations` property:

```
new Configs.Builder()
        .withConfigs(new RestClientConfig()
                .setEnableAdditionalValidations(true)) ①
        .build();
```

① Enables the requests validators

or

```
export rest-client.enableAdditionalValidations=true ①
```

① Enables the response validators

or using any other supported config types

Thus the RxMicro team recommends the following approach:

- Your project configuration must enable generation of validators for all possible model classes.
- Your `development` and `staging` environment must enable additional validation.
- Your `production` environment must disable additional validation!

# 7. Validation

The RxMicro framework supports validation of HTTP requests and responses.

All classes and annotations required for data validation are available in the `rxmicro.validation` module.

# 7.1. Preparatory Steps

To activate the validation module in a microservices project, the following steps must be taken:

1. Add the required dependencies to `pom.xml`.
2. Add the `rxmicro.validation` module to `module-info.java`.

## 7.1.1. Adding the Required Dependencies:

To activate the validation module in a microservices project, it is necessary to add the `rxmicro-validation` library:

```
<dependencies>
    <dependency>
        <groupId>io.rxmicro</groupId>
        <artifactId>rxmicro-validation</artifactId>
        <version>${rxmicro.version}</version>
    </dependency>
</dependencies>
```

## 7.1.2. Adding the `rxmicro.validation` Module to `module-info.java`

```
module example {
    requires rxmicro.validation; ①
}
```

① Adding the request and response validation module.

## 7.2. Built-in Constraints.

The RxMicro framework supports the following built-in constraints, that are available at the `io.rxmicro.validation.constraint` package:

*Table 13. Supported Built-in Constraints.*

| Annotation | Supported Type | Description |
|---|---|---|
| `@AllowEmptyString` | `java.lang.String` | The annotated element may be optional, i.e. `empty string` BUT must be not `null`! |
| `@AssertFalse` | `java.lang.Boolean` | The annotated element must be `false`. |
| `@AssertTrue` | `java.lang.Boolean` | The annotated element must be `true`. |
| `@Base64URLEncoded` | `java.lang.String` | The annotated element must be a valid Base64 string:<br>• Base;<br>• URL |
| `@CountryCode` | `java.lang.String` | The annotated element must be a valid country code:<br>• ISO 3166-1 alpha2;<br>• ISO 3166-1 alpha3;<br>• ISO 3166-1 numeric; |
| `@DigitsOnly` | `java.lang.String` | The annotated element must be a string value with digit characters only. |
| `@Email` | `java.lang.String` | The annotated element must be a well-formed email address. |
| `@EndsWith` | `java.lang.String` | The annotated element value must end with the provided suffix. |

| Annotation | Supported Type | Description |
| --- | --- | --- |
| `@Enumeration` | • `java.lang.String`<br>• `java.lang.Character` | The annotated element must be an element of the predefined enumeration.<br><br>*This validation rule is useful when a Java `enum` type is not applicable. For example: if an enum item name equals to a Java keyword. To solve this issue use `@Enumeration` annotation, otherwise use a Java `enum`.* |
| `@Future` | `java.time.Instant` | The annotated element must be an instant in the future. |
| `@FutureOrPresent` | `java.time.Instant` | The annotated element must be an instant in the present or in the future. |
| `@HostName` | `java.lang.String` | The annotated element must be a valid host name. |
| `@IP` | `java.lang.String` | The annotated element must be a valid IP address:<br><br>• IP version 4;<br>• IP version 6; |
| `@Lat` | `java.math.BigDecimal` | The annotated element must be a valid latitude coordinate. |
| `@LatinAlphabetOnly` | `java.lang.String` | The annotated element must be a string with latin alphabet letters only. |
| `@Length` | `java.lang.String` | The annotated element must have the expected string length. |
| `@Lng` | `java.math.BigDecimal` | The annotated element must be a valid longitude coordinate. |
| `@Lowercase` | `java.lang.String` | The annotated element must a lowercase string. |
| `@MaxDouble` | • `java.lang.Float`;<br>• `java.lang.Double`; | The annotated element must be a double whose value must be lower to the specified maximum. |

| Annotation | Supported Type | Description |
| --- | --- | --- |
| `@MaxInt` | • `java.lang.Byte`;<br>• `java.lang.Short`;<br>• `java.lang.Integer`;<br>• `java.lang.Long`; | The annotated element must be a `byte` or `short` or `integer` or `long` whose value must be lower or equal to the specified maximum. |
| `@MaxLength` | `java.lang.String` | The annotated element must have a string length whose value must be lower or equal to the specified maximum. |
| `@MaxNumber` | • `java.lang.Byte`;<br>• `java.lang.Short`;<br>• `java.lang.Integer`;<br>• `java.lang.Long`;<br>• `java.math.BigInteger`;<br>• `java.lang.Float`;<br>• `java.lang.Double`;<br>• `java.math.BigDecimal`; | The annotated element must be a number whose value must be lower or equal to the specified maximum. |
| `@MaxSize` | • `java.util.List`<br>• `java.util.Set`<br>• `java.util.Map` | The annotated element must have a list size whose value must be lower or equal to the specified maximum. |
| `@MinDouble` | • `java.lang.Float`;<br>• `java.lang.Double`; | The annotated element must be a double whose value must be higher or equal to the specified minimum. |
| `@MinInt` | • `java.lang.Byte`;<br>• `java.lang.Short`;<br>• `java.lang.Integer`;<br>• `java.lang.Long`; | The annotated element must be a `byte` or `short` or `integer` or `long` whose value must be higher or equal to the specified minimum. |
| `@MinLength` | `java.lang.String` | The annotated element must have a string length whose value must be higher or equal to the specified minimum. |

| Annotation | Supported Type | Description |
| --- | --- | --- |
| `@MinNumber` | • `java.lang.Byte`;<br>• `java.lang.Short`;<br>• `java.lang.Integer`;<br>• `java.lang.Long`;<br>• `java.math.BigInteger`;<br>• `java.lang.Float`;<br>• `java.lang.Double`;<br>• `java.math.BigDecimal`; | The annotated element must be a number whose value must be higher or equal to the specified minimum. |
| `@MinSize` | • `java.util.List`<br>• `java.util.Set`<br>• `java.util.Map` | The annotated element must have a list size whose value must be higher or equal to the specified minimum. |
| `@Nullable` | `? extends java.lang.Object` | The annotated element may be optional, i.e. `null`. |
| `@NullableArrayItem` | • `java.util.List`<br>• `java.util.Set` | The annotated array element may be optional, i.e. `null`. |
| `@Numeric` | `java.math.BigDecimal` | The annotated element must be a decimal within accepted range (scale and precision). |
| `@Past` | `java.time.Instant` | The annotated element must be an instant in the past. |
| `@PastOrPresent` | `java.time.Instant` | The annotated element must be an instant in the past or in the present. |
| `@Pattern` | `java.lang.String` | The annotated `String` must match the specified regular expression. The regular expression follows the Java regular expression conventions.<br><br>*(See `java.util.regex.Pattern`).* |
| `@Phone` | `java.lang.String` | The annotated element must be a valid phone number. |
| `@Size` | • `java.util.List`<br>• `java.util.Set`<br>• `java.util.Map` | The annotated element must have the expected list size. |

| Annotation | Supported Type | Description |
| --- | --- | --- |
| `@Skype` | `java.lang.String` | The annotated element must be a valid `skype` number. |
| `@StartsWith` | `java.lang.String` | The annotated element value must start with the provided prefix. |
| `@SubEnum` | `? extends java.lang.Enum` | The annotated element must be an enumeration with predefined sub sequence. |
| `@Telegram` | `java.lang.String` | The annotated element must be a valid `telegram` number. |
| `@TruncatedTime` | `java.time.Instant` | The annotated element must be an instant with truncated time value. |
| `@UniqueItems` | `java.util.List` | The annotated element must contain unique items. |
| `@Uppercase` | `java.lang.String` | The annotated element must an uppercase string. |
| `@URI` | `java.lang.String` | The annotated element must be a valid `java.net.URI` address.<br><br>*(See Uniform Resource Identifier)* |
| `@URLEncoded` | `java.lang.String` | The annotated element must be a valid URL encoded value. |
| `@Viber` | `java.lang.String` | The annotated element must be a valid `viber` number. |
| `@WhatsApp` | `java.lang.String` | The annotated element must be a valid `whatsapp` number. |

The RxMicro framework analyzes built-in constraints when drawing up the project documentation.

**Therefore, a properly selected annotation, in addition to its main purpose, makes it possible to automatically generate more accurate project documentation!**

## 7.3. HTTP Requests Server Validation

After activation of the `rxmicro.validation` module in the `module-info.java` descriptor

```
module examples.validation.server.basic {
    requires rxmicro.rest.server.netty;
    requires rxmicro.rest.server.exchange.json;
    requires rxmicro.validation; ①
}
```

① `rxmicro.validation` is a module for request and response validation.

the developer can use built-in constraints to validate HTTP request parameters:

```
final class MicroService {

    @PUT(value = "/", httpBody = false)
    void consume(final @Email String email) { ①
        System.out.println("Email: " + email);
    }

}
```

① The `email` parameter is annotated by the `@Email` annotation.

Due to this annotation the `RxMicro Annotation Processor` will automatically generate a validator for `email` HTTP parameter:

```
@RxMicroRestBasedMicroServiceTest(MicroService.class)
final class MicroServiceTest {

    private BlockingHttpClient blockingHttpClient;

    private SystemOut systemOut;

    @ParameterizedTest
    @CsvSource({
            "/,                     Parameter \"email\" is required!",  ①
            "/?email=,              Parameter \"email\" is required!",
            "/?email=rxmicro,       Invalid parameter \"email\": Expected a valid
email format!",
            "/?email=rxmicro.io,    Invalid parameter \"email\": Expected a valid
email format!",
            "/?email=@rxmicro.io,   Invalid parameter \"email\": Expected a valid
email format!",
            "/?email=rxmicro.io@,   Invalid parameter \"email\": Expected a valid
email format!",
            "/?email=@.rxmicro.io,  Invalid parameter \"email\": Expected a valid
email format!",
            "/?email=rxmicro.io@.,  Invalid parameter \"email\": Expected a valid
domain name!"
    })
    void Should_return_invalid_request_status(final String path,
                                              final String expectedErrorMessage) {
        final ClientHttpResponse response = blockingHttpClient.put(path);

        assertEquals(jsonErrorObject(expectedErrorMessage), response.getBody()); ②
        assertEquals(400, response.getStatusCode()); ②
        assertTrue(systemOut.isEmpty(), "System.out is not empty: " +
systemOut.asString()); ③
    }

    @Test
    void Should_handle_request() {
        final ClientHttpResponse response =
blockingHttpClient.put("/?email=welcome@rxmicro.io");

        assertEquals("Email: welcome@rxmicro.io", systemOut.asString()); ④
        assertEquals(200, response.getStatusCode()); ④
    }

}
```

① When activating the `rxmicro.validation` module, all query parameters are automatically considered as required.
*(Therefore the `RxMicro Annotation Processor` automatically adds a `required` validator for each parameter. If the parameter should be `optional`, the model field should be annotated with the `@Nullable` annotation.)*

② If request parameters are invalid, HTTP server automatically returns a status code `400` and JSON object of standard structure with detailed error description.

③ If an HTTP request validation error occurs, the request handler isn't invoked from the REST controller.

④ If the request parameters are valid, control is transferred to the request handler from the REST controller.

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-validation/validation-server-basic`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

## 7.4. HTTP Responses Server Validation

In addition to validating HTTP requests, the RxMicro framework also provides the option to validate HTTP responses.

*(HTTP response validation can be useful for identifying errors in business task implementation algorithms. For example, instead of returning an incorrect response model to a client, the microservice will throw an error. This approach increases the speed of error search and debugging of the source code that performs the business task.)*

If current implementation of the business task doesn't contain any errors, then HTTP response validation will consume computing resources senselessly. In this case, HTTP response validation must be disabled!

**By default the response validators are generated but not invoked!**

To activate the validation of responses it is necessary to set `enableAdditionalValidations` property:

```
new Configs.Builder()
        .withConfigs(new RestServerConfig()
                .setEnableAdditionalValidations(true)) ①
        .build();
```

① Enables the response validators

or

```
export rest-server.enableAdditionalValidations=true ①
```

① Enables the response validators

or using any other supported config types

The HTTP response model class can contain any built-in or custom constraint annotations:

```
public final class Response {

    final String message; ①

    public Response(final String message) {
        this.message = message;
    }

}
```

① In this example, the `message` field doesn't explicitly contain any constraint annotations. But since to the `module-info.java` descriptor was added the `rxmicro.validation` module, then all model

fields not marked with the `@Nullable` annotation are automatically required. *(In other words, such fields are implicitly marked by a virtual constraint annotation* `@Required`*.)*

To emulate an incorrect business value, `null` value is passed in the request handler to the `message` field:

```
final class MicroService {

    @GET("/")
    CompletableFuture<Response> get() {
        return CompletableFuture.completedFuture(new Response(null));
    }

}
```

In case of invalid HTTP response, the RxMicro framework returns HTTP response with a status `500` and standard error model:

```
@RxMicroRestBasedMicroServiceTest(MicroService.class)
final class MicroServiceTest {

    private BlockingHttpClient blockingHttpClient;

    @Test
    void Should_produce_internal_error() {
        final ClientHttpResponse response = blockingHttpClient.get("/");

        assertEquals(
                jsonErrorObject("Response is invalid: Parameter \"message\" is
required!"),
                response.getBody()
        );
        assertEquals(500, response.getStatusCode());
    }

}
```

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-validation/validation-server-response

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

# 7.5. HTTP Responses Client Validation

After activation of the `rxmicro.validation` module in the `module-info.java` descriptor

```
module examples.validation.client.basic {
    requires rxmicro.rest.client;
    requires rxmicro.rest.client.exchange.json;
    requires rxmicro.validation; ①
}
```

① `rxmicro.validation` is a module for request and response validation.

the developer can use built-in constraints to validate HTTP request parameters:

```
public final class Response {

    ①
    @Email
    String email;

    public String getEmail() {
        return email;
    }

}
```

① The `email` parameter is annotated by the `@Email` annotation.

After setting up the model class, this model must be used in the REST client:

```
@RestClient
public interface RESTClient {

    @GET("/")
    CompletableFuture<Response> get();

}
```

When converting the content of an HTTP response to the `Response` class object, the validator will be invoked automatically:

```
@InitMocks
@RxMicroComponentTest(RESTClient.class)
final class RESTClientTest {

    private static final HttpRequestMock HTTP_REQUEST_MOCK = new
HttpRequestMock.Builder()
```

```
            .setMethod(HttpMethod.GET)
            .setPath("/")
            .build();

    private RESTClient restClient;

    @Alternative
    @Mock
    private HttpClientFactory httpClientFactory;

    private void prepareValidResponse() {
        prepareHttpClientMock(
                httpClientFactory,
                HTTP_REQUEST_MOCK,
                jsonObject("email", "welcome@rxmicro.io") ①
        );
    }

    @Test
    @BeforeThisTest(method = "prepareValidResponse")
    void Should_return_received_email() {
        final Response response = restClient.get().join();
        assertEquals("welcome@rxmicro.io", response.getEmail()); ①
    }

    private void prepareInvalidResponse() {
        prepareHttpClientMock(
                httpClientFactory,
                HTTP_REQUEST_MOCK,
                jsonObject("email", "rxmicro.io") ②
        );
    }

    @Test
    @BeforeThisTest(method = "prepareInvalidResponse")
    void Should_throw_UnexpectedResponseException() {
        final Throwable throwable =
                ④
                getRealThrowable(assertThrows(RuntimeException.class, () ->
restClient.get().join())); ②
        assertEquals(UnexpectedResponseException.class, throwable.getClass()); ③
        assertEquals(
                "Response is invalid: " +
                        "Invalid parameter \"email\": Expected a valid email format!",
③
                throwable.getMessage()
        );
    }

}
```

① If the REST client receives a valid HTTP response, no errors will occur.

② If the `email` field is incorrect in the HTTP response, the REST client will return an error signal.

③ If the HTTP response is incorrect, the `UnexpectedResponseException` class exception is returned with a detailed text message.

④ **Since `CompletableFuture` when receiving an error signal ALWAYS returns the `CompletionException` class exception, to get the original exception class, which was thrown during the lazy evaluation, it is necessary to use the `Exceptions.getRealThrowable(Throwable)` utility method!**

Check out the following example to find out the features of the RxMicro framework for HTTP request validation in REST clients:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-validation/validation-client-all-types`

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-validation/validation-client-basic`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

# 7.6. HTTP Requests Client Validation

In addition to validating HTTP responses, the RxMicro framework also provides the option to validate HTTP requests.

*(HTTP request validation can be useful for identifying errors in business task implementation algorithms. For example, instead of returning an incorrect request model to a server, the REST client will throw an error. This approach increases the speed of error search and debugging of the source code that performs the business task.)*

If current implementation of the business task doesn't contain any errors, then HTTP request validation will consume computing resources senselessly. In this case, HTTP request validation must be disabled!

**By default the request validators are generated but not invoked!** To activate the validation of requests it is necessary to set `enableAdditionalValidations` property:

```
new Configs.Builder()
        .withConfigs(new RestClientConfig()
                .setEnableAdditionalValidations(true)) ①
        .build();
```

① Enables the requests validators

or

```
export rest-client.enableAdditionalValidations=true ①
```

① Enables the response validators

or using any other supported config types

# 7.7. Creating Custom Constraints

If built-in constraints are not enough, the developer can create custom constraint. To do so, the following steps must be taken:

1. Create a constraint annotation.
2. Implement a validator.

A validation annotation is an annotation that meets the following requirements:

```
①
@Retention(CLASS)
②
@Target({FIELD, METHOD, PARAMETER})
③
@ConstraintRule(
        supportedTypes = {
                BigDecimal.class                        ④
        },
        validatorClass = {
                ExpectedZeroConstraintValidator.class   ⑤
        }
)
public @interface ExpectedZero {

    boolean off() default false; ⑥

}
```

① The annotation is only available at the compilation level.

② This annotation allows validating class fields, class methods (`setters` and `getters`) and method parameters.

③ The `@ConstraintRule` annotation is used to indicate:

④ data type;

⑤ validator class.

⑥ Each constraint annotation requires a required `boolean off() default false;` parameter, that allows You to disable the validator.
*(This feature is useful for model inheritance when a parameter from a child class should not be validated and a parameter from a parent class should be validated!)*

Validator is a class that meets the following requirements:

```
public final class ExpectedZeroConstraintValidator
        implements ConstraintValidator<BigDecimal> { ①

    @Override
    public void validateNonNull(final BigDecimal value,
                                final HttpModelType httpModelType,
                                final String modelName) throws ValidationException {
        if (value.compareTo(BigDecimal.ZERO) != 0) { ②
            throw new ValidationException(
                "Invalid ? \"?\": Expected a zero value!",
                httpModelType, modelName
            );
        }
    }

}
```

① The validator class must implement the `ConstraintValidator` interface parameterized by the data type.
*(If constraint annotation can be applied to different data types, a separate validator class must be created for each data type.)*

② If the parameter is incorrect, the `ValidationException` exception with a clear error message must be thrown.

Using a custom validator is no different from using a predefined validator:

```
final class MicroService {

    @PATCH("/")
    void consume(final @ExpectedZero BigDecimal value) {
        System.out.println(value);
    }

}
```

```
@ParameterizedTest
@CsvSource({
        "/,                     Parameter \"value\" is required!",
        "/?value=,              Parameter \"value\" is required!",
        "/?value=1.23,          Invalid parameter \"value\": Expected a zero value!",
        "/?value=-3.45,         Invalid parameter \"value\": Expected a zero value!"
})
void Should_return_invalid_request_status(final String path,
                                          final String expectedErrorMessage) {
    final ClientHttpResponse response = blockingHttpClient.patch(path);

    assertEquals(jsonErrorObject(expectedErrorMessage), response.getBody());
    assertEquals(400, response.getStatusCode());
    assertEquals("", systemOut.asString());
}

@ParameterizedTest
@ValueSource(strings = {"0", "0.0", "0.000", "-0.0"})
void Should_handle_request(final String value) {
    final ClientHttpResponse response = blockingHttpClient.patch("/?value=" + value);

    assertEquals(new BigDecimal(value).toString(), systemOut.asString());
    assertEquals(200, response.getStatusCode());
}
```

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-validation/validation-server-custom

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

# 7.8. Disabling Validation

To disable the generation of validators, You must perform one of the following steps:

1. Delete the `rxmicro.validation` module from the `module-info.java` descriptor.
2. Use `GenerateOption.DISABLED` options to disable specific categories of validators.
3. Use the `@DisableValidation` annotation to disable validators of a selected group of classes in the project.

## 7.8.1. Removing the `rxmicro.validation` Module

The easiest and fastest way to disable the generation of validators for **all** classes in the current module is to remove the `rxmicro.validation` module from the `module-info.java` descriptor.

**After deleting the `rxmicro.validation` module DON'T FORGET to recompile the whole project for the changes to take effect: `mvn clean compile`!**

## 7.8.2. Using `GenerateOption.DISABLED` Option

To disable the generation of validators by category, it is necessary to use annotations:

- `@RestServerGeneratorConfig` (to set up the REST controllers).
- `@RestClientGeneratorConfig` (to set up the REST clients).

```
@RestServerGeneratorConfig(
        generateRequestValidators = GenerateOption.DISABLED,    ①
        generateResponseValidators = GenerateOption.DISABLED    ②
)
@RestClientGeneratorConfig(
        generateRequestValidators = GenerateOption.DISABLED,    ③
        generateResponseValidators = GenerateOption.DISABLED    ④
)
module examples.validation {
    requires rxmicro.rest.server;
    requires rxmicro.rest.client;
    requires rxmicro.validation;
}
```

① Validators for all models of HTTP requests in the current project won't be generated.

② Validators for all models of HTTP responses in the current project won't be generated.

③ Validators for all models of HTTP requests in the current project won't be generated.

④ Validators for all models of HTTP responses in the current project won't be generated.

**Upon activation of the `rxmicro.validation` module, by default the RxMicro framework generates only HTTP request validators for REST controllers and HTTP response validators for REST clients!**

All other categories of validators must be manually activated using the `@RestServerGeneratorConfig` and `@RestClientGeneratorConfig` annotations!

**After changing the settings using the `@RestServerGeneratorConfig` and `@RestClientGeneratorConfig` annotations DON'T FORGET to recompile the whole project for the changes to take effect: `mvn clean compile`!**

### 7.8.3. Using `@DisableValidation` Annotation

The `@DisableValidation` annotation provides an opportunity to disable the generation of validators for the selected group of classes in the project:

1. If a model class is annotated by this annotation, then **only** for this model class the validator won't be generated.
2. If this annotation annotates the `package-info.java` class, then for **all classes from the specified package and all its subpackages** no validators will be generated.
3. If this annotation annotates the `module-info.java` descriptor, then for **all classes in the current module** no validators will be generated.
   *(This behavior is similar to the removal of the `rxmicro.validation` module from the `module-info.java` descriptor.)*

**After adding the `@DisableValidation` annotation DON'T FORGET to recompile the whole project for the changes to take effect: `mvn clean compile`!**

# 8. REST-based Microservice Documentation

REST-based microservice documentation describes rules of interaction with REST-based microservices for fast and easy integration with them.

Currently, the RxMicro framework only supports specialized documentation format based on the `AsciiDoc` format.

# 8.1. Basic Usage

Default settings allow You to generate REST-based microservice documentation using minimal configurations.
*(This advantage is achieved through close integration with other RxMicro modules.)*

## 8.1.1. Min Settings

To generate REST-based microservice documentation during the compilation of the microservice project, the following two steps must be done:

- Add the `rxmicro-documentation-asciidoctor` dependency to `pom.xml` of the microservice project:

```
<dependency>
    <groupId>io.rxmicro</groupId>
    <artifactId>rxmicro-documentation-asciidoctor</artifactId>
    <version>${rxmicro.version}</version>
</dependency>
```

- Add the `rxmicro.documentation.asciidoctor` module to the `module-info.java` descriptor of the microservice project:

```
module examples.documentation.asciidoctor.quick.start {
    requires rxmicro.rest.server.netty;
    requires rxmicro.rest.server.exchange.json;
    requires static rxmicro.documentation.asciidoctor; ①
}
```

① The `rxmicro.documentation.asciidoctor` must be added using the `static` modifier.
*(For more information on the benefits of the `static` modifier, refer to the Section 8.1.3, "`Asciidoctor-dependency-plugin` Settings".)*

After performing these steps during compiling the microservice project, the `RxMicro Annotation Processor` will generate the `ExamplesDocumentationAsciidoctorQuickStartDocumentation.adoc` file. The generated file will contain REST-based microservice documentation in the `AsciiDoc` format, and by default will be located in the `./src/main/asciidoc` folder.

In terms of the version control system, the `./src/main/asciidoc` folder is the source code, so it will be controlled by this system. Therefore, if REST-based microservice documentation is to be dynamically generated, it is necessary to:

- specify that files in the `./src/main/asciidoc` folder must be ignored by the version control system;
- or configure the `rxmicro.documentation.asciidoctor` module, so that it generates REST-based microservice documentation in another folder, for example in the `target/docs`.
  *(To configure the folder in which REST-based microservice documentation should be generated, refer to the core.pdf.)*

Using the `./src/main/asciidoc` folder by default provides the following benefits:

- The `asciidoctor-maven-plugin`, which converts the `AsciiDoc` format to `HTML` or `PDF` , by default reads files from the `./src/main/asciidoc` folder.
- Automatic building of REST-based microservice documentation can be used as an initial version of the document. After formation of the initial version, the generation can be disabled and all changes to the documentation will be made manually by the developer and stored in the version control system.
- Compared to the previous version, a more efficient way to keep REST-based microservice documentation up to date is to divide all documentation into fragments. Static fragments (business task description, charts and figures) will be created manually by the developer. Dynamic fragments will be generated by the `RxMicro Annotation Processor` at each project compilation. As a result, a final document containing static and dynamic fragments will be compiled from these fragments.
  *(When using this approach, do not forget to add the all dynamic fragments to ignore list for Your version control system!)*

### 8.1.2. `Asciidoctor-maven-plugin` Settings

The `asciidoctor-maven-plugin` plugin allows You to convert documentation from the `AsciiDoc` format to `HTML`, `PDF` and other formats.

```
<plugin>
    <groupId>org.asciidoctor</groupId>
    <artifactId>asciidoctor-maven-plugin</artifactId> ①
    <version>${asciidoctor-maven-plugin.version}</version>
    <executions>
        <execution>
            <id>output-html5</id>
            <phase>package</phase> ②
            <goals>
                <goal>process-asciidoc</goal>
            </goals>
            <configuration>
                <backend>html5</backend> ③
                <sourceHighlighter>highlight.js</sourceHighlighter> ④
                <preserveDirectories>true</preserveDirectories>
                <relativeBaseDir>true</relativeBaseDir>
                <attributes> ⑤
                    <icons>font</icons>
                    <pagenums/>
                    <toclevels>3</toclevels>
                </attributes>
            </configuration>
        </execution>
    </executions>
</plugin>
```

① The latest stable version of the `asciidoctor-maven-plugin`.

② The `asciidoctor-maven-plugin` must convert REST-based microservice documentation at the `package` phase.

③ Using the `backend` directive, You can specify in which format the `AsciiDoc` document should be converted.

④ If the `AsciiDoc` document contains examples of source code in a programming language, You need to add the js library for highlighting the syntax of that language.

⑤ Using the `attributes` directive, it is possible to override the attributes of the `AsciiDoc` document.

For more information about the `asciidoctor-maven-plugin` plugin features, refer to the documentation:

`https://asciidoctor.org/docs/asciidoctor-maven-plugin/`

### 8.1.3. `Asciidoctor-dependency-plugin` Settings

REST-based microservice documentation is generated during the compilation process, therefore the libraries used to configure the REST-based microservice documentation generation process are not required in `runtime`. Therefore, the `maven-dependency-plugin` should not copy artifacts related to the generation of REST-based microservice documentation to the `lib` folder:

```
<plugin>
    <artifactId>maven-dependency-plugin</artifactId>
    <version>${maven-dependency-plugin.version}</version>
    <executions>
        <execution>
            <id>copy-dependencies</id>
            <phase>package</phase>
            <goals>
                <goal>copy-dependencies</goal>
            </goals>
            <configuration>
                <outputDirectory>${project.build.directory}/lib</outputDirectory>
                <includeScope>compile</includeScope>
                <excludeArtifactIds> ①
                    rxmicro-documentation,
                    rxmicro-documentation-asciidoctor
                </excludeArtifactIds>
            </configuration>
        </execution>
    </executions>
</plugin>
```

① The `rxmicro-documentation` and `rxmicro-documentation-asciidoctor` artifacts are required **only** during the compilation process.

In order that in `runtime` no errors occur when excluding the artifacts required for REST-based microservice documentation generation, it is required to use the `static` modifier in the `module-info.java` descriptor: `requires static rxmicro.documentation.asciidoctor;`.

Without this modifier the following error will occur when starting the microservice: `Module rxmicro.documentation.asciidoctor not found`!

When compiling a project, the RxMicro framework will generate the following project documentation according to the settings of the `rxmicro.documentation.asciidoctor` module defined in this section:

`ExamplesDocumentationAsciidoctorQuickStartDocumentation.html`

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-documentation-asciidoctor/documentation-asciidoctor-quick-start`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

## 8.2. RxMicro Annotations

The RxMicro framework supports the following RxMicro Annotations:

*Table 14. Supported RxMicro Annotations.*

| Annotation | Description |
|---|---|
| @Author | Denotes the author of the generated REST-based microservice documentation.<br><br>*(Allows You to override the author specified in the developer directive to pom.xml.)* |
| @BaseEndpoint | Denotes the basic endpoint in the generated REST-based microservice documentation.<br><br>*(Allows You to override the basic endpoint specified in the url directive to pom.xml.)* |
| @Description | Denotes the description of the generated REST-based microservice documentation.<br><br>*(Allows You to override the description specified in the description directive to pom.xml.)*<br><br>In addition to the description of all REST-based microservice documentation, this annotation also allows You to specify a description of separate elements: sections, model fields, etc. |
| @DocumentationDefinition | A composite annotation that specifies the settings for generating a whole document. |
| @DocumentationVersion | Denotes the version of REST-based microservice in the generated REST-based microservice documentation.<br><br>*(Allows You to override the version of REST-based microservice specified in the version directive to pom.xml.)* |
| @Example | Denotes the model field value used as an example in the generated REST-based microservice documentation. |
| @IncludeDescription | Denotes the AsciiDoc fragment to be imported into the generated REST-based microservice documentation.<br><br>In addition to the description of all REST-based microservice documentation, this annotation also allows You to specify the AsciiDoc fragment for separate elements: sections, model fields, etc. |
| @IntroductionDefinition | A composite annotation that specifies the settings for generating the Introduction section. |

| Annotation | Description |
| --- | --- |
| `@License` | Denotes the license of REST-based microservice in the generated REST-based microservice documentation.<br><br>*(Allows You to override the license of REST-based microservice specified in the `license` directive to `pom.xml`.)* |
| `@ModelExceptionErrorResponse` | Denotes the exception class to be analyzed by the `RxMicro Annotation Processor` for generating the unsuccessful HTTP response description of REST-based microservice. |
| `@ResourceDefinition` | A composite annotation that specifies the settings for generating the `ResourceDefinition` section. |
| `@ResourceGroupDefinition` | A composite annotation that specifies the settings for generating the `ResourceGroupDefinition` section. |
| `@SimpleErrorResponse` | Contains metadata about the unsuccessful HTTP response of REST-based microservice. |
| `@Title` | Denotes the name of the generated REST-based microservice documentation.<br><br>*(Allows You to override the name of the generated REST-based microservice documentation specified in the `name` directive to `pom.xml`.)* |
| `@DocumentAttributes` | Allows You to specify AsciiDoc attributes for the generated REST-based microservice documentation. |

## 8.3. `@Example` and `@Description` Usage

Using the `@Example` annotation, the developer can specify the data that is as close to the real data as possible, which will be used to build examples of usage in REST-based microservice documentation.

Using the `@Description` annotation, the developer can specify the detailed description of separate model fields to be used in building REST-based microservice documentation.

```
public final class Echo {

    @Example("EchoExample")
    @Description("EchoDescription")
    String echo;
}
```

```
final class MicroService {

    @GET("/")
    @POST("/")
    @POST(value = "/", httpBody = false)
    CompletableFuture<Echo> echo(final Echo echo) {
        return completedFuture(echo);
    }
}
```

When compiling a project, the RxMicro framework will generate the following project documentation according to the settings of the `rxmicro.documentation.asciidoctor` module defined in this section:

`ExamplesDocumentationAsciidoctorEchoDocumentation.html`

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-documentation-asciidoctor/documentation-asciidoctor-echo`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

## 8.4. Sections Customization

Using the following composite annotations:

- `@DocumentationDefinition`;
- `@IncludeDescription`;
- `@ResourceGroupDefinition`;
- `@ResourceDefinition`

the developer can regroup standard sections of REST-based microservice documentation, as well as add his own fragments as sections.

```
@DocumentationDefinition(
        withGeneratedDate = false,
        introduction = @IntroductionDefinition(
                sectionOrder = {
                        IntroductionDefinition.Section.LICENSES,
                        IntroductionDefinition.Section.SPECIFICATION,
                        IntroductionDefinition.Section.CUSTOM_SECTION
                },
                customSection = {
                        "${PROJECT-DIR}/src/main/asciidoc/_fragment/" +
                                "custom-introduction-content.asciidoc"
                },
                includeMode = IncludeMode.INLINE_CONTENT
        ),
        resourceGroup = @ResourceGroupDefinition(
                sectionOrder = {
                        ResourceGroupDefinition.Section.CUSTOM_SECTION
                },
                customSection = {
                        "${PROJECT-DIR}/src/main/asciidoc/_fragment/" +
                                "custom-resource-group-content.asciidoc"
                },
                includeMode = IncludeMode.INLINE_CONTENT
        ),
        resource = @ResourceDefinition(
                withInternalErrorResponse = false,
                withJsonSchema = false,
                withRequestIdResponseHeader = false,
                withQueryParametersDescriptionTable = false,
                withBodyParametersDescriptionTable = false
        )
)
module examples.documentation.asciidoctor.custom.sections {
    requires rxmicro.rest.server.netty;
    requires rxmicro.rest.server.exchange.json;
    requires static rxmicro.documentation.asciidoctor;
}
```

Besides generating the final `AsciiDoc` document, the `RxMicro Annotation Processor` allows You to generate separate dynamic fragments.

For more information on this feature, check out the following example:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-documentation-asciidoctor/documentation-asciidoctor-custom-sections

When compiling a project, the RxMicro framework will generate the following project documentation according to the settings of the `rxmicro.documentation.asciidoctor` module defined in this section:

`ExamplesDocumentationAsciidoctorCustomSectionsDocumentation.html`

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-documentation-asciidoctor/documentation-asciidoctor-custom-sections`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

## 8.5. Integration with `rxmicro.validation` Module

The `rxmicro.documentation.asciidoctor` module is integrated with the `rxmicro.validation` module. Thanks to this integration, when building the REST-based microservice documentation, the RxMicro framework analyzes all available built-in constraints for automatic generation of model fields description.

```
@DocumentationDefinition(
        introduction = @IntroductionDefinition(sectionOrder = {}),
        withGeneratedDate = false
)
module examples.documentation.asciidoctor.validation {
    requires rxmicro.rest.server.netty;
    requires rxmicro.rest.server.exchange.json;
    requires rxmicro.validation;
    requires static rxmicro.documentation.asciidoctor;
}
```

```
final class MicroService {

    @PUT("/")
    void consume(final @Phone String phone) { ①
        // do something
    }
}
```

① The `phone` parameter must be validated via built-in constraint for phones.

Thus, when building the REST-based microservice documentation, the RxMicro framework will automatically generate a description for the `phone` parameter:

HTTP Request Body Parameters Description

| Name | Type | Restrictions | Description |
|---|---|---|---|
| phone | string | • required: true<br>• format: phone number | Phone number in the international format.<br>**Read more:**<br>• *What is phone number format?* |

*Figure 13. The `phone` field description, formed based on built-in constraints analysis*

When compiling a project, the RxMicro framework will generate the following project documentation according to the settings of the `rxmicro.documentation.asciidoctor` module defined in this section:

`ExamplesDocumentationAsciidoctorValidationDocumentation.html`

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-documentation-asciidoctor/documentation-asciidoctor-validation`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

## 8.6. REST-based Microservice Metadata Configuration

To obtain metadata while building REST-based microservice documentation, the RxMicro framework can use:

- The `pom.xml` file;
- The `rxmicro.documentation` module annotations.

The REST-based microservice metadata is:

- the REST-based microservice name;
- the REST-based microservice description;
- the REST-based microservice version;
- the list of licenses that cover the REST-based microservice;
- the list of REST-based microservice developers;
- the REST-based microservice basic endpoint.

### 8.6.1. Using `pom.xml`

To specify the metadata needed to generate the REST-based microservice documentation, You can use the `pom.xml` file

```
<project>
    <name>Metadata Pom Xml</name>
    <description>*Project* _Description_</description>
    <developers>
        <developer>
            <name>Richard Hendricks</name>
            <email>richard.hendricks@piedpiper.com</email>
        </developer>
        <developer>
            <name>Bertram Gilfoyle</name>
            <email>bertram.gilfoyle@piedpiper.com</email>
        </developer>
        <developer>
            <name>Dinesh Chugtai</name>
            <email>dinesh.chugtai@piedpiper.com</email>
        </developer>
    </developers>
    <url>https://api.rxmicro.io</url>
    <licenses>
        <license>
            <name>Apache License 2.0</name>
            <url>https://github.com/rxmicro/rxmicro/blob/master/LICENSE</url>
        </license>
    </licenses>
</project>
```

When compiling a project, the RxMicro framework will generate the following project documentation according to the settings of the `rxmicro.documentation.asciidoctor` module defined in this section:

`MetadataPomXmlDocumentation.html`

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-documentation-asciidoctor/documentation-asciidoctor-metadata-pomxml`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 8.6.2. Using Annotations

To specify the metadata needed to generate the REST-based microservice documentation, You can use the `rxmicro.documentation` module annotations.

```
@Title("Metadata Annotations")
@Description("*Project* _Description_")
@DocumentationVersion("0.0.1")
@Author(
        name = "Richard Hendricks",
        email = "richard.hendricks@piedpiper.com"
)
@Author(
        name = "Bertram Gilfoyle",
        email = "bertram.gilfoyle@piedpiper.com"
)
@Author(
        name = "Dinesh Chugtai",
        email = "dinesh.chugtai@piedpiper.com"
)
@BaseEndpoint("https://api.rxmicro.io")
@License(
        name = "Apache License 2.0",
        url = "https://github.com/rxmicro/rxmicro/blob/master/LICENSE"
)
@DocumentationDefinition(
        introduction = @IntroductionDefinition(
                sectionOrder = {
                        IntroductionDefinition.Section.BASE_ENDPOINT,
                        IntroductionDefinition.Section.LICENSES
                }
        ),
        resource = @ResourceDefinition(
                withInternalErrorResponse = false
        ),
        withGeneratedDate = false
)
module examples.documentation.asciidoctor.metadata.annotations {
    requires rxmicro.rest.server.netty;
    requires rxmicro.rest.server.exchange.json;
    requires static rxmicro.documentation.asciidoctor;
}
```

When compiling a project, the RxMicro framework will generate the following project documentation according to the settings of the `rxmicro.documentation.asciidoctor` module defined in this section:

`MetadataAnnotationsDocumentation.html`

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-documentation-asciidoctor/documentation-asciidoctor-metadata-annotations`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

# 8.7. Error Documentation

To describe unsuccessful HTTP responses, the `rxmicro.documentation` module provides two annotations:

- `@SimpleErrorResponse`;
- `@ModelExceptionErrorResponse`.

```
Unresolved directive in _fragment/_project-documentation/errors.adoc -
include::../../../../../../examples/group-documentation-asciidoctor/documentation-
asciidoctor-
errors/src/main/java/io/rxmicro/examples/documentation/asciidoctor/errors/ProxyMicroSe
rvice.java[tags=content]
```

① The `@SimpleErrorResponse` annotation allows You to explicitly specify the HTTP response description.

② The `status` parameter describes the HTTP status code.

③ The `description` parameter describes the text description.

④ The `exampleErrorMessage` parameter denotes the value used as an example in the generated REST-based microservice documentation.

⑤ The `@ModelExceptionErrorResponse` annotation allows You to specify the exception class of the standard

⑥ or custom type.

When using the custom exception type, this class contains all necessary parameters for building REST-based microservice documentation:

```
Unresolved directive in _fragment/_project-documentation/errors.adoc -
include::../../../../../../examples/group-documentation-asciidoctor/documentation-
asciidoctor-
errors/src/main/java/io/rxmicro/examples/documentation/asciidoctor/errors/model/NotAcc
eptableException.java[tags=content]
```

① The required `STATUS_CODE` static field describes the HTTP status code.

② The `@Description` annotation describes the text description.

③ The `@Example` annotation describes the value used as an example in the generated REST-based microservice documentation.

The custom exception class can contain not string parameter(s). For such classes the RxMicro framework returns custom JSON model instead of standard one.

```
Unresolved directive in _fragment/_project-documentation/errors.adoc -
include::../../../../../../examples/group-documentation-asciidoctor/documentation-
asciidoctor-
errors/src/main/java/io/rxmicro/examples/documentation/asciidoctor/errors/model/Custom
ErrorModelException.java[tags=content]
```

① The `@Description` annotation describes the text description.

② The required `STATUS_CODE` static field describes the HTTP status code.

③ The custom field can contain `@Example` and `@Description` annotations.

④ The custom field(s) must be initialized via constructor.

⑤ For custom exception classes with custom field(s) the RxMicro Annotation Processor does not generate `Writer`, so You must override the `getResponseData` method manually!

⑥ Overridden method must return values from all declared fields at custom exception class.

If `RX_MICRO_STRICT_MODE` option is enabled, the RxMicro Annotation Processor validates the overidden `getResponseData` method and throws compilation error if some field values not returned!

When compiling a project, the RxMicro framework will generate the following project documentation according to the settings of the `rxmicro.documentation.asciidoctor` module defined in this section:

`ExamplesDocumentationAsciidoctorErrorsDocumentation.html`

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-documentation-asciidoctor/documentation-asciidoctor-errors`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

# 9. Postgre SQL Data Repositories

The RxMicro framework supports creation of dynamic repositories for interaction with databases.

To interact with `PostgreSQL DB`, using the reactive R2DBC PostgreSQL driver, the RxMicro framework provides the `rxmicro.data.sql.r2dbc.postgresql` module.

## 9.1. Basic Usage

To use the `rxmicro.data.sql.r2dbc.postgresql` module in the project, the following two steps must be taken:

- Inject the `rxmicro.data.sql.r2dbc.postgresql` dependency to the `pom.xml` file:

```
<dependency>
    <groupId>io.rxmicro</groupId>
    <artifactId>rxmicro-data-sql-r2dbc-postgresql</artifactId>
    <version>${rxmicro.version}</version>
</dependency>
```

- Add the `rxmicro.data.sql.r2dbc.postgresql` module to the `module-info.java` descriptor:

```
module examples.data.r2dbc.postgresql.basic {
    requires rxmicro.data.sql.r2dbc.postgresql;
}
```

By default, the reactive R2DBC PostgreSQL driver uses the **Project Reactor** library, so when adding the `rxmicro.data.sql.r2dbc.postgresql` module, the `reactor.core` module is automatically added and available for usage!

After adding the `rxmicro.data.sql.r2dbc.postgresql` module, You can create a data model class and dynamic repository:

```
@Table
①
@ColumnMappingStrategy
public final class Account {

    @Column(length = Column.UNLIMITED_LENGTH)
    String firstName;

    @Column(length = Column.UNLIMITED_LENGTH)
    String lastName;

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }
}
```

① The `@ColumnMappingStrategy` annotation sets the strategy of forming column names of the

relational database table based on the analysis of the Java model class field names.
*(Thus, the `firstName` field corresponds to the `first_name` column, and the `lastName` field corresponds to the `last_name` column.)*

```
①
@PostgreSQLRepository
public interface DataRepository {

    ②
    @Select("SELECT * FROM ${table} WHERE email = ?")
    Mono<Account> findByEmail(String email);
}
```

① In order for a standard interface to be recognized by the RxMicro framework as a dynamic repository for interaction with `PostgreSQL DB`, this interface should be annotated by `@PostgreSQLRepository` annotation.

② The dynamic repository may contain methods that form a query to the `PostgreSQL DB`.
*(The query that used for a request for data uses the `SQL` and is specified in the annotation parameters.)*

Since the dynamic repository is a RxMicro component, for its testing You need to use the microservice component testing approach:

The common approach recommended for testing dynamic repositories, that interact with `PostgreSQL DB`, is described in the Section 9.3.2, “Test Templates”.

```
@Testcontainers
@RxMicroComponentTest(DataRepository.class)
final class DataRepositoryTest {

    @Container
    private final GenericContainer<?> postgresqlTestDb =
            new GenericContainer<>("rxmicro/postgres-test-db")
                    .withExposedPorts(5432);

    @WithConfig
    private final PostgreSQLConfig config = new PostgreSQLConfig()
            .setDatabase("rxmicro")
            .setUser("rxmicro")
            .setPassword("password");

    private DataRepository dataRepository;

    @BeforeEach
    void beforeEach() {
        config
                .setHost(postgresqlTestDb.getHost())
                .setPort(postgresqlTestDb.getFirstMappedPort());
    }

    @Test
    void Should_find_account() {
        final Account account = requireNonNull(
                dataRepository.findByEmail("richard.hendricks@piedpiper.com").block()
        );

        assertEquals("Richard", account.getFirstName());
        assertEquals("Hendricks", account.getLastName());
    }
}
```

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-data-r2dbc-postgresql/data-r2dbc-postgresql-basic

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

## 9.2. RxMicro Annotations

The RxMicro framework supports the following RxMicro Annotations:

*Table 15. Supported RxMicro Annotations.*

| Annotation | Description |
|---|---|
| @Column | Sets mapping between the column name in the PostgreSQL DB table and the Java model class field name.<br><br>*(By default, the RxMicro framework uses the Java model class field name as the column name in the PostgreSQL DB table. If the name should differ for some reason, (for example, as a column name in the PostgreSQL DB table the keyword Java is used), it should be specified using this annotation!)*<br><br>Required length parameter must contain the max character count that can be stored to table column. If actual length will be more that expected length, value will be trimmed automatically.<br><br>The nullable parameter does not disable any validation. It used as the descriptive characteristic only. |
| @ColumnMappingStrategy | Sets the strategy of column name formation in the PostgreSQL DB table, based on the analysis of the Java model class field names.<br><br>*(If this annotation annotates the Java model class, then the set strategy will be used for **all** fields in this class. For example, if You set the default LOWERCASE_WITH_UNDERSCORED strategy, then the parentId field in the Java class will correspond to the parent_id column in the PostgreSQL DB table.)* |
| @DataRepositoryGeneratorConfig | Allows You to configure the repository generation process. |
| @RepeatParameter | Allows setting mapping between one method parameter marked with this annotation and several universal placeholders that are used in the query to PostgreSQL DB. |
| @Select | Denotes a repository method that must execute a SELECT SQL operation |
| @CustomSelect | Denotes a string parameter of repository method, the value of that must be used as custom SELECT. |
| @Insert | Denotes a repository method that must execute a INSERT SQL operation |
| @Update | Denotes a repository method that must execute a UPDATE SQL operation |
| @Delete | Denotes a repository method that must execute a DELETE SQL operation |
| @ExpectedUpdatedRowsCount | Enables validation for updated rows count during DML operation, like Insert, Update and Delete operations.<br><br>If current database has invalid state the InvalidDatabaseStateException will be thrown! |

| Annotation | Description |
| --- | --- |
| `@NotInsertable` | Denotes a model field, the value of that ignored during `INSERT` SQL operation. |
| `@NotUpdatable` | Denotes a model field, the value of that ignored during `UPDATE` SQL operation. |
| `@PrimaryKey` | Denotes a model field that must be used as primary key. |
| `@Schema` | Denotes a schema of a database table. |
| `@SequenceGenerator` | Denotes a sequence that must be used to get the next unique value for model field. |
| `@Table` | Denotes a table name for entity. |
| `@EnumType` | Denotes a db type name for enum. |
| `@VariableValues` | Denotes a storage with the values of the predefined variables. |
| `@PostgreSQLRepository` | Denotes that an interface is a dynamic generated PostgreSQL data repository. |
| `@PartialImplementation` | Denotes an abstract class that contains a partial implementation of the annotated by this annotation a PostgreSQL Data Repository interface. |

## 9.3. Repositories Testing

For successful functional testing of dynamic repositories, that interact with `PostgreSQL DB`, it is required:

- Presence of a script that creates a test database.
- Mechanism for preparing a database for testing: creating a database before starting the test and deleting a database after completing the test.

### 9.3.1. Test Database

A test database was created for testing the `rxmicro.data.sql.r2dbc.postgresql` module features, which are described in this section.

The test database contains three tables: `account`, `product` and `order`:

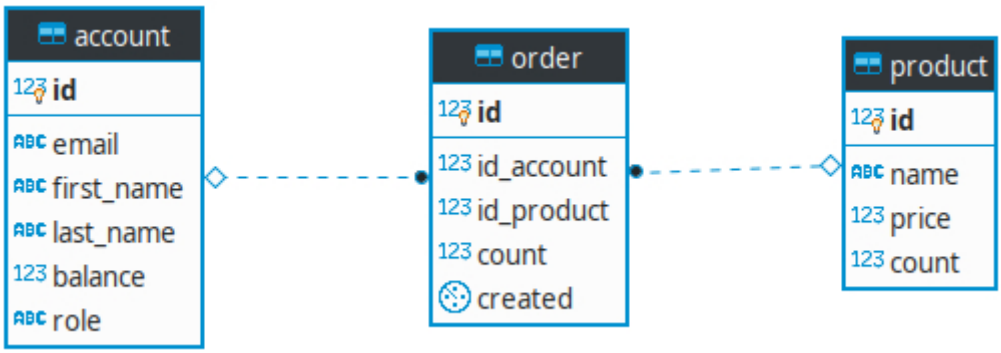



*Figure 14. ER-Diagram for Test Database*

SQL scripts for creating a test database are available at the following link:

`https://github.com/rxmicro/rxmicro-usage/blob/master/examples/docker-image-postgres-test-db/src/main/sql/`

The following classes of Java models correspond to the tables created in the test database:

```
@Table
@ColumnMappingStrategy
public final class Account {

    @PrimaryKey
    @SequenceGenerator
    Long id;

    @Column(length = Column.UNLIMITED_LENGTH)
    @NotUpdatable
    String email;

    @Column(length = Column.UNLIMITED_LENGTH)
    String firstName;

    @Column(length = Column.UNLIMITED_LENGTH)
    String lastName;

    BigDecimal balance;

    Role role;
}
```

```
@Table
@ColumnMappingStrategy
public final class Order {

    @PrimaryKey
    Long id;

    Long idAccount;

    Integer idProduct;

    Integer count;

    @NotInsertable
    Instant created;
}
```

```
@Table
@ColumnMappingStrategy
public final class Product {

    @PrimaryKey(autoGenerated = false)
    Integer id;

    @Column(length = Column.UNLIMITED_LENGTH)
    String name;

    BigDecimal price;

    Integer count;
}
```

```
public enum Role {

    CEO,

    Lead_Engineer,

    Systems_Architect
}
```

For ease of studying the `rxmicro.data.sql.r2dbc.postgresql` module, You can use the ready-made `PostgreSQL DB` image with the `rxmicro/postgres-test-db` test database.

The source code of the project used as a base for building this `docker` image, is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/docker-image-postgres-test-db`

### 9.3.2. Test Templates

As a mechanism for preparing a database for testing (creating a database before starting the test and deleting a database after completing the test), it is most convenient to use `docker`.

To start `docker` containers in the functional test it is convenient to use the `Testcontainers` Java library:

```
①
@Testcontainers
②
@RxMicroComponentTest(DataRepository.class)
final class DataRepositoryTestTemplate1 {

    ③
    @Container
    private static final GenericContainer<?> POSTGRESQL_TEST_DB =
            new GenericContainer<>("rxmicro/postgres-test-db")
                    .withExposedPorts(5432); ④

    ⑤
    @WithConfig
    private static final PostgreSQLConfig CONFIG = new PostgreSQLConfig()
            .setDatabase("rxmicro")
            .setUser("rxmicro")
            .setPassword("password"); ⑥

    @BeforeAll
    static void beforeAll() {
        POSTGRESQL_TEST_DB.start(); ⑦
        CONFIG
                .setHost(POSTGRESQL_TEST_DB.getHost()) ⑧
                .setPort(POSTGRESQL_TEST_DB.getFirstMappedPort());
    }

    private DataRepository dataRepository; ⑨

    // ... test methods must be here

    @AfterAll
    static void afterAll() {
        POSTGRESQL_TEST_DB.stop(); ⑩
    }
}
```

① The `@Testcontainers` annotation activates the start and stop of the `docker` containers to be used in this test.

② Since the dynamic repository is a RxMicro component, for its testing You need to use the microservice component testing approach.

③ The `@Container` annotation indicates the `docker` container that will be used in this test. As an image on the basis of which it is necessary to create the `docker` container, the `PostgreSQL DB` ready-made image with the `rxmicro/postgres-test-db` test database is used.

④ When starting the `docker` container, You need to open the standard port for `PostgreSQL DB`.

⑤ Using the `@WithConfig` annotation, the configuration available only during the test is declared.

⑥ Setting up the configuration to interact with the test database.

⑦ Before running all tests, You must start the `docker` container.

⑧ After starting the `docker` container, You need to read the random IP address and port that will be used when connecting to the running `docker` container.

⑨ When testing microservice components, it is necessary to specify a reference to the component in which the RxMicro framework will inject the tested component.

⑩ After completing all the tests, You must stop the `docker` container.

The main advantage of this template is the speed of testing. Since the `docker` container is created once before starting **all** test methods, the total runtime of all test methods is reduced. The main disadvantage of this template is that if any test method changes the `PostgreSQL DB` state, the following test method may end with an error.

**Therefore, this functional test template should be used for queries to `PostgreSQL DB` that do not change the database state!**

If You need to test methods that change the `PostgreSQL DB` state, You should use another template:

```
①
@Testcontainers
②
@RxMicroComponentTest(DataRepository.class)
final class DataRepositoryTestTemplate2 {

    ③
    @Container
    private final GenericContainer<?> postgresqlTestDb =
            new GenericContainer<>("rxmicro/postgres-test-db")
                    .withExposedPorts(5432); ④

    ⑤
    @WithConfig
    private final PostgreSQLConfig config = new PostgreSQLConfig()
            .setDatabase("rxmicro")
            .setUser("rxmicro")
            .setPassword("password"); ⑥

    private DataRepository dataRepository; ⑦

    @BeforeEach
    void beforeEach() {
        config
                .setHost(postgresqlTestDb.getHost()) ⑧
                .setPort(postgresqlTestDb.getFirstMappedPort());
    }

    // ... test methods must be here
}
```

① The `@Testcontainers` annotation activates the start and stop of the `docker` containers to be used in this test.

② Since the dynamic repository is a RxMicro component, for its testing You need to use the microservice component testing approach

③ The `@Container` annotation indicates the `docker` container that will be used in this test. As an image on the basis of which it is necessary to create the `docker` container, the `PostgreSQL DB` ready-made image with the `rxmicro/postgres-test-db` test database is used.

④ When starting the `docker` container, You need to open the standard port for `PostgreSQL DB`.

⑤ Using the `@WithConfig` annotation, the configuration available only during the test is declared.

⑥ Setting up the configuration to interact with the test database.

⑦ When testing microservice components, it is necessary to specify a reference to the component in which the RxMicro framework will inject the tested component.

⑧ After starting the `docker` container, You need to read the random IP address and port that will be used when connecting to the running `docker` container.

This template **for each test method** will create and drop the `docker` container, which may increase the total runtime of all test methods.

**Therefore, select the most appropriate functional test template based on the requirements of the tested functionality!**

The `Testcontainers` library starts the `docker` container before running the test and stops the `docker` container automatically after completing the test!

So You should start and stop the `docker` container manually only if You want to use one `docker` container for **all** test methods!

# 9.4. DataBase Models

The RxMicro framework supports the following database model types:

- primitives;
- entities.

## 9.4.1. Primitives

A primitive is a supported Java type that can be mapped to database table column.

The `rxmicro.data.sql.r2dbc.postgresql` module supports the following primitive type:

- `? extends Enum<?>`;
- `java.lang.Boolean`;
- `java.lang.Byte`;
- `java.lang.Short`;
- `java.lang.Integer`;
- `java.lang.Long`;
- `java.math.BigInteger`;
- `java.lang.Float`;
- `java.lang.Double`;
- `java.math.BigDecimal`;
- `java.lang.Character`;
- `java.lang.String`;
- `java.time.Instant`;
- `java.time.LocalTime`;
- `java.time.LocalDate`;
- `java.time.LocalDateTime`;
- `java.time.OffsetDateTime`;
- `java.time.ZonedDateTime`;
- `java.net.InetAddress`;
- `java.util.UUID`;

For floating point numbers, it is suggested to use the `java.math.BigDecimal` type instead of `java.lang.Float` or `java.lang.Double`.

Using the `java.math.BigDecimal` allows excluding errors of decimal number notation. **This is particularly important if the decimal number is used to represent currency-related data!**

### 9.4.2. Entities

An entity is a composition of primitives only.

For example:

```
@Table
@ColumnMappingStrategy
public final class Account {

    @PrimaryKey
    @SequenceGenerator
    Long id;

    @Column(length = Column.UNLIMITED_LENGTH)
    @NotUpdatable
    String email;

    @Column(length = Column.UNLIMITED_LENGTH)
    String firstName;

    @Column(length = Column.UNLIMITED_LENGTH)
    String lastName;

    BigDecimal balance;

    Role role;
}
```

## 9.5. Universal Placeholder

The RxMicro framework recommends using the universal placeholder (?) as parameter value placeholder in the SQL queries:

```
@Select("SELECT * FROM ${table} WHERE email=?")
Mono<Account> findByEmail(String email);
```

If this method invoked with the following parameter:
`repository.findByEmail("welcome@rxmicro.io");`,

the RxMicro framework will generate the following `PostgreSQL DB` query:
`SELECT * FROM account WHERE email='welcome@rxmicro.io'`.

## 9.6. `@RepeatParameter` Annotation

The universal placeholder (`?`) is the simplest type of placeholders.

But unfortunately, it has one disadvantage: if a query parameter must be repeated, a developer must define a copy of this parameter:

```
@Select("SELECT * FROM ${table} WHERE firstName=? OR lastName=?")
Mono<Account> findByFirstOrLastNames(String name1, String name2);
```

The `@RepeatParameter` annotation fixes this disadvantage.

The following code is an equivalent to the code with a copy of the `name` parameter:

```
@Select("SELECT * FROM ${table} WHERE firstName=? OR lastName=?")
Mono<Account> findByFirstOrLastNames(@RepeatParameter(2) String name);
```

## 9.7. SQL Operations

### 9.7.1. `@Select`

The `rxmicro.data.sql.r2dbc.postgresql` module supports the `SELECT` SQL operation.

#### 9.7.1.1. Returning Types Support

##### 9.7.1.1.1. Reactive Types Support

PostgreSQL Data Repositories that generated by the RxMicro frameworks support the following return reactive types:

- If expected an asynchronous `0-1` result:
  a. `Mono<MODEL>`;
  b. `CompletableFuture<MODEL>`;
  c. `CompletionStage<MODEL>`;
  d. `Single<MODEL>`;
  e. `Maybe<MODEL>`.

```
@PostgreSQLRepository
public interface SelectSingleDataRepository {

    @Select("SELECT * FROM ${table} WHERE email = ?")
    Mono<Account> findByEmail1(String email);

    @Select("SELECT * FROM ${table} WHERE email = ?")
    CompletableFuture<Account> findByEmail2(String email);

    @Select("SELECT * FROM ${table} WHERE email = ?")
    CompletionStage<Account> findByEmail3(String email);

    @Select("SELECT * FROM ${table} WHERE email = ?")
    CompletableFuture<Optional<Account>> findByEmail4(String email);

    @Select("SELECT * FROM ${table} WHERE email = ?")
    CompletionStage<Optional<Account>> findByEmail5(String email);

    @Select("SELECT * FROM ${table} WHERE email = ?")
    Single<Account> findByEmail6(String email);

    @Select("SELECT * FROM ${table} WHERE email = ?")
    Maybe<Account> findByEmail7(String email);
}
```

- If expected an asynchronous `0-n` result:
  a. `Mono<java.util.List<MODEL>>`;
  b. `Flux<MODEL>`;

c. `CompletableFuture<java.util.List<MODEL>>`;

d. `CompletionStage<java.util.List<MODEL>>`;

e. `Single<java.util.List<MODEL>>`;

f. `Flowable<MODEL>`.

```
@PostgreSQLRepository
public interface SelectManyDataRepository {

    @Select("SELECT * FROM ${table} ORDER BY id")
    Mono<List<Account>> findAll1();

    @Select("SELECT * FROM ${table} ORDER BY id")
    Flux<Account> findAll2();

    @Select("SELECT * FROM ${table} ORDER BY id")
    CompletableFuture<List<Account>> findAll3();

    @Select("SELECT * FROM ${table} ORDER BY id")
    CompletionStage<List<Account>> findAll4();

    @Select("SELECT * FROM ${table} ORDER BY id")
    Single<List<Account>> findAll5();

    @Select("SELECT * FROM ${table} ORDER BY id")
    Flowable<Account> findAll6();
}
```

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-data-r2dbc-postgresql/data-r2dbc-postgresql-select-return-reactive-types`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

#### 9.7.1.1.2. Model Types Support

PostgreSQL Data Repositories that generated by the RxMicro frameworks support the following return model types:

- If expected an asynchronous `0-1` result:
  a. Entity;
  b. `EntityFieldMap`;
  c. `EntityFieldList`;
  d. Primitive.

```
@PostgreSQLRepository
@VariableValues({
        "${table}", "account"
})
public interface SelectSingleDataRepository {

    @Select("SELECT * FROM ${table} " +
            "WHERE email='richard.hendricks@piedpiper.com'")
    CompletableFuture<Account> findSingleAccount();

    @Select("SELECT first_name, last_name FROM ${table} " +
            "WHERE email='richard.hendricks@piedpiper.com'")
    CompletableFuture<EntityFieldMap> findSingleEntityFieldMap();

    @Select("SELECT first_name, last_name FROM ${table} " +
            "WHERE email='richard.hendricks@piedpiper.com'")
    CompletableFuture<EntityFieldList> findSingleEntityFieldList();

    @Select("SELECT email FROM ${table} " +
            "WHERE email='richard.hendricks@piedpiper.com'")
    CompletableFuture<String> findSingleEmail();

    @Select("SELECT role FROM ${table} " +
            "WHERE email='richard.hendricks@piedpiper.com'")
    CompletableFuture<Role> findSingleRole();

    @Select("SELECT balance FROM ${table} " +
            "WHERE email='richard.hendricks@piedpiper.com'")
    CompletableFuture<BigDecimal> findSingleBalance();
}
```

- If expected an asynchronous `0-n` result:
  a. A list of entities;
  b. `java.util.List<EntityFieldMap>`;
  c. `java.util.List<EntityFieldList>`;

d. A list of primitives.

```
@PostgreSQLRepository
public interface SelectManyDataRepository {

    @Select("SELECT first_name, last_name FROM ${table} ORDER BY id")
    CompletableFuture<List<Account>> findAllAccounts();

    @Select(
            value = "SELECT first_name, last_name FROM ${table} ORDER BY id",
            entityClass = Account.class
    )
    CompletableFuture<List<EntityFieldMap>> findAllEntityFieldMapList();

    @Select(
            value = "SELECT first_name, last_name FROM ${table} ORDER BY id",
            entityClass = Account.class
    )
    CompletableFuture<List<EntityFieldList>> findAllEntityFieldList();

    @Select(
            value = "SELECT email FROM ${table} ORDER BY id",
            entityClass = Account.class
    )
    CompletableFuture<List<String>> findAllEmails();

    @Select(
            value = "SELECT DISTINCT role FROM ${table} ORDER BY role",
            entityClass = Account.class
    )
    CompletableFuture<List<Role>> findAllRoles();

    @Select(
            value = "SELECT DISTINCT balance FROM ${table} ORDER BY balance",
            entityClass = Account.class
    )
    CompletableFuture<List<BigDecimal>> findAllBalances();
}
```

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-data-r2dbc-postgresql/data-r2dbc-postgresql-select-return-model-types

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

#### 9.7.1.1.3. All Supported Return Types

For more information, we recommend that You familiarize yourself with the following examples:

All Supported Return Types

#### 9.7.1.2. `WHERE`, `ORDER BY` and Other `SELECT` Operations

The `rxmicro.data.sql.r2dbc.postgresql` module supports all SQL nested operators that are supported by the `SELECT` operation:

- `WHERE` operator:

```
@PostgreSQLRepository
public interface SelectByFilterRepository {

    @Select("SELECT * FROM ${table} WHERE role=?")
    CompletableFuture<List<Account>> findByRole(Role role);

    @Select("SELECT * FROM ${table} WHERE first_name=? OR first_name=? OR
first_name=?")
    CompletableFuture<List<Account>> findByFirstName(
            String firstName1, String firstName2, String firstName3
    );

    @Select("SELECT * FROM ${table} WHERE balance BETWEEN ? AND ?")
    CompletableFuture<List<Account>> findByBalance(BigDecimal minBalance, BigDecimal
maxBalance);

    @Select("SELECT * FROM ${table} WHERE first_name=? OR last_name=?")
    CompletableFuture<List<Account>> findByFirstOrLastName(String name1, String
name2);

    @Select("SELECT * FROM ${table} WHERE first_name ILIKE ? OR last_name ILIKE ?")
    CompletableFuture<List<Account>> findByFirstOrLastName(@RepeatParameter(2) String
name);
}
```

- `IN` operator:

```
@PostgreSQLRepository
public interface SelectByFilterUsingINOperatorRepository {

    @Select("SELECT * FROM ${table} WHERE role IN ('CEO'::role,
'Systems_Architect'::role)")
    CompletableFuture<List<Account>> findByRole();

    @Select("SELECT * FROM ${table} WHERE email NOT IN (SELECT email FROM
blocked_accounts)")
    CompletableFuture<List<Account>> findNotBlockedAccount();

    @Select("SELECT * FROM ${table} WHERE email in ?")
    CompletableFuture<Optional<Account>> findByEmail(String email);

    @Select("SELECT * FROM ${table} WHERE email in ?")
    CompletableFuture<List<Account>> findByEmail(List<String> emails);

    @Select("SELECT * FROM ${table} WHERE role in (?)")
    CompletableFuture<List<Account>> findByRole(Role role);

    @Select("SELECT * FROM ${table} WHERE role in (?)")
    CompletableFuture<List<Account>> findByRole(List<Role> roles);

    @Select("SELECT * FROM ${table} WHERE balance in (?)")
    CompletableFuture<List<Account>> findByBalance(BigDecimal balance);

    @Select("SELECT * FROM ${table} WHERE balance in ?")
    CompletableFuture<List<Account>> findByBalance(List<BigDecimal> balances);
}
```

- `ORDER BY` operator:

```
@PostgreSQLRepository
public interface SelectOrderedDataRepository {

    @Select("SELECT * FROM ${table} ORDER BY id")
    CompletableFuture<List<Account>> findAllOrderedById();

    @Select("SELECT * FROM ${table} ORDER BY ( id ? )")
    CompletableFuture<List<Account>> findAllOrderedById(SortOrder sortOrder);

    @Select("SELECT * FROM ${table} ORDER BY ? ?")
    CompletableFuture<List<Account>> findAllOrderedBy(String columnName, SortOrder
sortOrder);

    @Select("SELECT * FROM ${table} ORDER BY (id ?, email ?) LIMIT 10")
    CompletableFuture<List<Account>> findAllOrderedByIdAndEmail(
            @RepeatParameter(2) SortOrder sortOrder
    );
}
```

- `LIMIT` and/or `OFFSET` operator(s):

```
@PostgreSQLRepository
public interface SelectLimitedDataRepository {

    @Select("SELECT * FROM ${table} ORDER BY id LIMIT 2")
    CompletableFuture<List<Account>> findFirst2Accounts();

    @Select("SELECT * FROM ${table} ORDER BY id LIMIT ?")
    CompletableFuture<List<Account>> findAccounts(int limit);

    @Select("SELECT * FROM ${table} ORDER BY id LIMIT ? OFFSET ?")
    CompletableFuture<List<Account>> findAccounts(int limit, int offset);

    @Select("SELECT * FROM ${table} ORDER BY id LIMIT ? OFFSET ?")
    CompletableFuture<List<Account>> findAccounts(Pageable pageable);
}
```

- Composition of `WHERE`, `ORDER BY`, `LIMIT` and `OFFSET` operators:

```
@PostgreSQLRepository
public interface SelectComplexDataRepository {

    @Select("SELECT * FROM ${table} WHERE " +
            "first_name ILIKE ? AND role IN (?) AND balance < ? " +
            "ORDER BY (id ?, email ?) " +
            "LIMIT ? " +
            "OFFSET ?")
    CompletableFuture<List<Account>> find01(
            String firstNameTemplate,
            List<Role> roles,
            BigDecimal balance,
            @RepeatParameter(2) SortOrder sortOrder,
            int limit,
            int offset
    );
}
```

- etc.

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-data-r2dbc-postgresql/data-r2dbc-postgresql-select-parameters`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

#### 9.7.1.3. Selected Projections

The `rxmicro.data.sql.r2dbc.postgresql` module supports projections from selected table(s).

To use projections, developer must specify required columns at `SELECT` query:

```
@PostgreSQLRepository
public interface SelectProjectionDataRepository {

    @Select("SELECT * FROM ${table} " +
            "WHERE email='richard.hendricks@piedpiper.com'")
    CompletableFuture<Account> findAllColumns();

    @Select("SELECT id, email, first_name, last_name, balance FROM ${table} " +
            "WHERE email='richard.hendricks@piedpiper.com'")
    CompletableFuture<Account> findAllColumnsExceptRole1();

    @Select("SELECT id, email, last_name, first_name, balance FROM ${table} " +
            "WHERE email='richard.hendricks@piedpiper.com'")
    CompletableFuture<Account> findAllColumnsExceptRole2();

    @Select("SELECT 1 as id, " +
            "'richard.hendricks@piedpiper.com' as email, " +
            "'Hendricks' as last_name, " +
            "'Richard' as first_name, " +
            "70000.00 as balance")
    CompletableFuture<Account> findAllColumnsExceptRole3();

    @Select("SELECT first_name, last_name FROM ${table} " +
            "WHERE email='richard.hendricks@piedpiper.com'")
    CompletableFuture<Account> findFirstAndLastName();

    @Select("SELECT id, " +
            "'***@***' as email, " +
            "upper(last_name) as last_name, " +
            "first_name, " +
            "(20000 + 50000.00) as balance " +
            "FROM ${table} " +
            "WHERE email='richard.hendricks@piedpiper.com'")
    CompletableFuture<Account> findModifiedColumns();
}
```

For each nonstandard projection, the RxMicro framework generates a separate converter method.

For example for `SelectProjectionDataRepository` the RxMicro framework generates the following converter:

```
public final class $$AccountEntityFromDBConverter
        extends EntityFromDBConverter<Row, RowMetadata, Account> {

    ①
    public Account fromDB(final Row dbRow,
                          final RowMetadata metadata) {
        final Account model = new Account();
        model.id = dbRow.get(0, Long.class);
        model.email = dbRow.get(1, String.class);
        model.firstName = dbRow.get(2, String.class);
        model.lastName = dbRow.get(3, String.class);
        model.balance = dbRow.get(4, BigDecimal.class);
        model.role = toEnum(Role.class, dbRow.get(5, String.class), "role");
        return model;
    }

    public Account fromDBFirst_nameLast_name(final Row dbRow,
                                             final RowMetadata metadata) {
        final Account model = new Account();
        model.firstName = dbRow.get(0, String.class);
        model.lastName = dbRow.get(1, String.class);
        return model;
    }

    public Account fromDBIdEmailFirst_nameLast_nameBalance(final Row dbRow,
                                                           final RowMetadata metadata)
{
        final Account model = new Account();
        model.id = dbRow.get(0, Long.class);
        model.email = dbRow.get(1, String.class);
        model.firstName = dbRow.get(2, String.class);
        model.lastName = dbRow.get(3, String.class);
        model.balance = dbRow.get(4, BigDecimal.class);
        return model;
    }

    public Account fromDBIdEmailLast_nameFirst_nameBalance(final Row dbRow,
                                                           final RowMetadata metadata)
{
        final Account model = new Account();
        model.id = dbRow.get(0, Long.class);
        model.email = dbRow.get(1, String.class);
        model.lastName = dbRow.get(2, String.class);
        model.firstName = dbRow.get(3, String.class);
        model.balance = dbRow.get(4, BigDecimal.class);
        return model;
    }
}
```

① It is standard converter example.

*(This converter is a standard one, because an order of the selected columns is defined by the order of fields of Java model class. (`Account` class for current example.))*

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-data-r2dbc-postgresql/data-r2dbc-postgresql-select-projection`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

#### 9.7.1.4. Custom Select

The `rxmicro.data.sql.r2dbc.postgresql` module introduces a `@CustomSelect` annotation that allows working with a `Custom SELECT`.

The `Custom SELECT` is a string parameter that sends to a repository method and contains a SQL, built dynamically during an execution of a microservice:

```
@PostgreSQLRepository
public interface CustomSelectRepository {

    @Select
    CompletableFuture<List<EntityFieldMap>> findAll(
            @CustomSelect String sql ①
    );

    @Select
    CompletableFuture<Optional<Account>> findAccount(
            @CustomSelect(supportUniversalPlaceholder = false) String sql, ②
            String firstName
    );

    @Select
    CompletableFuture<Optional<Account>> findFirstAndLastName(
            @CustomSelect(selectedColumns = {"first_name", "last_name"}) String sql,
③
            String firstName
    );

    @Select
    CompletableFuture<Optional<Account>> findLastAndFirstName(
            @CustomSelect(selectedColumns = {"last_name", "first_name"}) String sql,
④
            String firstName
    );
}
```

① This is example of a repository method that can execute any `SELECT` query.

② This repository method selects all columns defined at `Account` entity.
*(Disabling of universal placeholder means that developer must use postgres specific placeholder (`$1`, `$2`, etc) instead of universal placeholder (`?`). Otherwise error will be thrown!)*

③ This repository method selects only selected columns (`first_name` and `last_name`) from `account` table.
*(This method supports universal placeholder!)*

④ This repository method selects only selected columns (`last_name` and `first_name`) from `account` table.
*(This method supports universal placeholder!)*

Using of the `Custom SELECT` feature requires that dynamic built SQL contains the predefined selected columns, otherwise the model converter will convert a table row to the instance of Java class incorrectly!

The following test describes how the `Custom SELECT` feature can be tested:

```
@Test
void findAll() {
    final List<EntityFieldMap> entityFieldMaps = dataRepository.findAll(
            "SELECT email, first_name, last_name FROM account WHERE id = 1"
    ).join();
    assertEquals(
            List.of(
                    orderedMap(
                            "email", "richard.hendricks@piedpiper.com",
                            "first_name", "Richard",
                            "last_name", "Hendricks"
                    )
            ),
            entityFieldMaps
    );
}

@Test
void findAccount() {
    final Optional<Account> optionalAccount = dataRepository.findAccount(
            "SELECT * FROM account WHERE first_name = $1",
            "Richard"
    ).join();
    assertEquals(
            Optional.of(
                    new Account(
                            1L,
                            "richard.hendricks@piedpiper.com",
                            "Richard",
                            "Hendricks",
                            new BigDecimal("70000.00")
                    )
            ),
            optionalAccount
    );
}

@Test
void findFirstAndLastName() {
    final Optional<Account> optionalAccount = dataRepository.findFirstAndLastName(
            "SELECT first_name, last_name FROM account WHERE first_name = ?",
            "Richard"
    ).join();
    assertEquals(
```

```
            Optional.of(new Account("Richard", "Hendricks")),
            optionalAccount
    );
}

@Test
void findLastAndFirstName() {
    final Optional<Account> optionalAccount = dataRepository.findLastAndFirstName(
            "SELECT last_name, first_name FROM account WHERE first_name = ?",
            "Richard"
    ).join();
    assertEquals(
            Optional.of(new Account("Richard", "Hendricks")),
            optionalAccount
    );
}
```

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-data-r2dbc-postgresql/data-r2dbc-postgresql-select-custom`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 9.7.2. `@Insert`

The `rxmicro.data.sql.r2dbc.postgresql` module supports the `INSERT` SQL operation:

```
@PostgreSQLRepository
public interface DataRepository {

    @Insert
    CompletableFuture<Boolean> insert1(Account account);

    @Insert
    CompletableFuture<Account> insert2(Account account);

    @Insert("INSERT INTO ${table} VALUES(nextval('account_seq'),?,?,?,?,?)")
    ①
    @VariableValues({
            "${table}", "account"
    })
    CompletableFuture<Long> insert3(
            String email, String firstName, String lastName, BigDecimal balance, Role
role
    );

    @Insert("INSERT INTO ${table} VALUES(nextval('account_seq'),?,?,?,?,?) RETURNING
*")
    CompletableFuture<Account> insert4(
            String email, String firstName, String lastName, BigDecimal balance, Role
role
    );

    @Insert(
            value = "INSERT INTO ${table} VALUES(nextval('account_seq'),?,?,?,?,?)",
            entityClass = Account.class
    )
    CompletableFuture<Long> insert5(
            String email, String firstName, String lastName, BigDecimal balance, Role
role
    );

    @Insert(
            value = "INSERT INTO ${table} VALUES(nextval('account_seq'),?,?,?,?,?)
RETURNING *",
            entityClass = Account.class
    )
    CompletableFuture<EntityFieldMap> insert6(
            String email, String firstName, String lastName, BigDecimal balance, Role
role
    );

    @Insert("INSERT INTO ${table}(${inserted-columns}) VALUES(${values}) " +
```

```
            "RETURNING ${returning-columns}")
    CompletableFuture<AccountResult> insert7(Account account);

    @Insert("INSERT INTO ${table}(${inserted-columns}) VALUES(${values}) " +
            "ON CONFLICT (${id-columns}) DO UPDATE SET ${on-conflict-update-inserted-
columns}" +
            "RETURNING ${returning-columns}")
    CompletableFuture<AccountResult> insert8(Account account);

    @Insert("INSERT INTO ${table}(${inserted-columns}) VALUES(${values}) " +
            "ON CONFLICT (${id-columns}) DO UPDATE SET ${on-conflict-update-inserted-
columns}")
    CompletableFuture<Void> insert9(Account account);

    @Insert("INSERT INTO ${table}(${inserted-columns}) VALUES(${values}) " +
            "ON CONFLICT (${id-columns}) DO NOTHING")
    CompletableFuture<Void> insert10(Account account);

    @Insert("INSERT INTO ${table} SELECT * FROM dump RETURNING *")
    CompletableFuture<List<Account>> insertMany1();

    @Insert("INSERT INTO account SELECT * FROM dump")
    CompletableFuture<Long> insertMany2();
}
```

① The variable values are used to resolve predefined variables at the SQL query.
*(Read more about the algorithm of the variables resolving at Section 9.8, "Variables Support".)*

For more information, we recommend that You familiarize yourself with the following examples:

All Supported Return Types

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-data-r2dbc-postgresql/data-r2dbc-postgresql-insert`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 9.7.3. @Update

The rxmicro.data.sql.r2dbc.postgresql module supports the UPDATE SQL operation:

```
@PostgreSQLRepository
public interface DataRepository {

    @Update
    CompletableFuture<Boolean> update1(Account account);

    @Update("UPDATE ${table} SET first_name=?, last_name=? WHERE id=?")
    ①
    @VariableValues({
            "${table}", "account"
    })
    CompletableFuture<Long> update2(String firstName, String lastName, Long id);

    @Update("UPDATE ${table} SET first_name=?, last_name=? WHERE ${by-id-filter}
RETURNING *")
    CompletableFuture<Account> update3(String firstName, String lastName, Long id);

    @Update(
            value = "UPDATE ${table} SET first_name=?, last_name=? " +
                    "WHERE id = ?",
            entityClass = Account.class
    )
    CompletableFuture<Long> update4(String firstName, String lastName, Long id);

    @Update(
            value = "UPDATE ${table} SET first_name=?, last_name=? " +
                    "WHERE ${by-id-filter} RETURNING *",
            entityClass = Account.class
    )
    CompletableFuture<EntityFieldMap> update5(String firstName, String lastName, Long
id);
}
```

① The variable values are used to resolve predefined variables at the SQL query.
*(Read more about the algorithm of the variables resolving at Section 9.8, "Variables Support".)*

For more information, we recommend that You familiarize yourself with the following examples:

All Supported Return Types

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-data-r2dbc-postgresql/data-r2dbc-postgresql-update

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 9.7.4. @Delete

The rxmicro.data.sql.r2dbc.postgresql module supports the DELETE SQL operation:

```
@PostgreSQLRepository
public interface DataRepository {

    @Delete
    CompletableFuture<Boolean> delete1(Account account);

    @Delete("DELETE FROM ${table} WHERE balance < ?")
    ①
    @VariableValues({
            "${table}", "account"
    })
    CompletableFuture<Long> delete2(BigDecimal minRequiredBalance);

    @Delete("DELETE FROM ${table} WHERE ${by-id-filter} RETURNING *")
    CompletableFuture<Account> delete3(Long id);

    @Delete(entityClass = Account.class)
    CompletableFuture<Long> delete4(Long id);

    @Delete(
            value = "DELETE FROM ${table} WHERE ${by-id-filter} RETURNING *",
            entityClass = Account.class
    )
    CompletableFuture<EntityFieldMap> delete5(Long id);
}
```

① The variable values are used to resolve predefined variables at the SQL query.
*(Read more about the algorithm of the variables resolving at Section 9.8, "Variables Support".)*

For more information, we recommend that You familiarize yourself with the following examples:

All Supported Return Types

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-data-r2dbc-postgresql/data-r2dbc-postgresql-delete

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

## 9.8. Variables Support

When building SQL queries, sometimes it is necessary to specify the table name as a string constant. This feature provides the developer with more flexibility: the table names may vary depending on the environment.

For better readability of SQL query, the RxMicro framework recommends using predefined variables instead of string concatenation:

```
①
public static final String TABLE_NAME = "table1";

②
@Select("SELECT id, value FROM " + TABLE_NAME + " WHERE id = ?")
CompletableFuture<EntityFieldMap> findById1(long id);

③
@Select("SELECT id, value FROM ${table} WHERE id = ?")
@VariableValues({
        "${table}", TABLE_NAME
})
CompletableFuture<EntityFieldMap> findById2(long id);
```

① String constant with table name.

② When strings are concatenated, the readability of an entire SQL query gets worse.

③ Instead of string concatenation, the RxMicro framework recommends using predefined variables.

All predefined variables supported by the RxMicro framework are declared in the `SupportedVariables` class

To determine the value of the predefined variable used in the query specified for the repository method, the RxMicro framework uses the following algorithm:

1. If the repository method returns or accepts the entity model as a parameter, the entity model class is used to define the variable value.
2. Otherwise, the RxMicro framework analyzes the optional `entityClass` parameter defined in the `@Select`, `@Insert`, `@Update` and `@Delete` annotations.
3. If the optional `entityClass` parameter is set, the class specified in this parameter is used to define the variable value.
4. If the optional `entityClass` parameter is missing, the RxMicro framework tries to extract the variable value from the `@VariableValues` annotation, which annotates this repository method.
5. If the repository method is not annotated with the `@VariableValues` annotation or the `@VariableValues` annotation does not contain the value of a predefined variable, then the RxMicro framework tries to extract the value of this variable from the `@VariableValues` annotation, which annotates the repository interface.

6. If the variable value is undefined in all specified places, then the RxMicro framework notifies the developer about the error.

```
@PostgreSQLRepository
@VariableValues({
        "${table}", SelectDataRepository.GLOBAL_TABLE
})
public interface SelectDataRepository {

    public static final String GLOBAL_TABLE = "global_table";

    public static final String ENTITY_TABLE = "entity_table";

    public static final String LOCAL_TABLE = "local_table";

    ①
    @Select("SELECT * FROM ${table}")
    CompletableFuture<List<Entity>> findFromEntityTable1();

    ②
    @Select(value = "SELECT * FROM ${table}", entityClass = Entity.class)
    CompletableFuture<List<EntityFieldMap>> findFromEntityTable2();

    ③
    @Select("SELECT * FROM ${table}")
    CompletableFuture<List<EntityFieldMap>> findFromGlobalTable();

    ④
    @Select("SELECT * FROM ${table}")
    @VariableValues({
            "${table}", SelectDataRepository.LOCAL_TABLE
    })
    CompletableFuture<List<EntityFieldMap>> findFromLocalTable();
}
```

① The `${table}` variable value will be equal to `entity_table`.
*(The variable value is read from the `Entity` class, which is returned by this method.)*

② The `${table}` variable value will be equal to `entity_table`.
*(The variable value is read from the `Entity` class, since this class is specified in the `entityClass` parameter.)*

③ The `${table}` variable value will be equal to `global_table`.
*(The variable value is read from the `@VariableValues` annotation, which annotates the repository interface.)*

④ The `${table}` variable value will be equal to `local_table`.
*(The variable value is read from the `@VariableValues` annotation, which annotates the repository method.)*

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-data-r2dbc-postgresql/data-r2dbc-postgresql-variables`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

## 9.9. Primary Keys Support

The `rxmicro.data.sql.r2dbc.postgresql` module supports four types of the primary keys:

- Auto generated primary key. (`SERIAL` type.)
  *(A uniqueness of this type of primary key is controlled by the database server!)*:

```
@PrimaryKey
Long id;
```

- Auto generated primary key that uses a sequence to get the next unique value.
  *(A uniqueness of this type of primary key is controlled by the database server!)*:

```
@PrimaryKey
@SequenceGenerator
Long id;
```

- Manually set primary key.
  *(A developer must control a uniqueness of this type of primary key!)*:

```
@PrimaryKey(autoGenerated = false)
Integer id;
```

- Complex primary key:
  *(A developer must control a uniqueness of this type of primary key!)*:

```
@PrimaryKey(autoGenerated = false)
Long idCategory;

@PrimaryKey(autoGenerated = false)
@Column(length = Column.UNLIMITED_LENGTH)
String idType;

@PrimaryKey(autoGenerated = false)
Role idRole;
```

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-data-r2dbc-postgresql/data-r2dbc-postgresql-primary-keys

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

## 9.10. `@ExpectedUpdatedRowsCount` annotation

Enables validation for updated rows count during DML operation, like `Insert`, `Update` and `Delete` operations. This annotation adds additional runtime validator that validates the actual updated rows during SQL operation.

If current database has invalid state the `InvalidDatabaseStateException` will be thrown!

The following examples demonstrate the `@ExpectedUpdatedRowsCount` annotation usage:

```
@ExpectedUpdatedRowsCount(10)
@Insert("INSERT INTO ${table} SELECT * FROM dump")
Mono<Long> insert12();

@ExpectedUpdatedRowsCount(1)
@Insert("INSERT INTO ${table} VALUES(nextval('account_seq'),?,?)")
Mono<Boolean> insert13(String firstName, String lastName);

@ExpectedUpdatedRowsCount(1)
@Insert("INSERT INTO ${table} VALUES(nextval('account_seq'),?,?) RETURNING *")
Mono<Account> insert14(String firstName, String lastName);

@ExpectedUpdatedRowsCount(0)
@Insert("INSERT INTO ${table} VALUES(nextval('account_seq'),?,?) RETURNING *")
Mono<Account> insert15(String firstName, String lastName);
```

```
@ExpectedUpdatedRowsCount(10)
@Update("UPDATE ${table} SET first_name=?, last_name=? WHERE email=?")
Mono<Long> update12(String firstName, String lastName, String email);

@ExpectedUpdatedRowsCount(1)
@Update("UPDATE ${table} SET first_name=?, last_name=? WHERE id=?")
Mono<Boolean> update13(String firstName, String lastName, Long id);

@ExpectedUpdatedRowsCount(1)
@Update("UPDATE ${table} SET first_name=?, last_name=? WHERE ${by-id-filter} RETURNING
*")
Mono<Account> update14(String firstName, String lastName, Long id);

@ExpectedUpdatedRowsCount(0)
@Update("UPDATE ${table} SET first_name=?, last_name=? WHERE ${by-id-filter} RETURNING
*")
Mono<Account> update15(String firstName, String lastName, Long id);
```

```
@ExpectedUpdatedRowsCount(1)
@Delete(entityClass = Account.class)
Mono<Void> delete11(Long id);

@ExpectedUpdatedRowsCount(10)
@Delete("DELETE FROM ${table} WHERE first_name ILIKE ? OR last_name ILIKE ?")
Mono<Long> delete12(Transaction transaction, @RepeatParameter(2) String name);

@ExpectedUpdatedRowsCount(1)
@Delete(entityClass = Account.class)
Mono<Boolean> delete13(Long id);

@ExpectedUpdatedRowsCount(1)
@Delete("DELETE FROM ${table} WHERE ${by-id-filter} RETURNING *")
Mono<Account> delete14(Long id);

@ExpectedUpdatedRowsCount(0)
@Delete("DELETE FROM ${table} WHERE ${by-id-filter} RETURNING *")
Mono<Account> delete15(Long id);
```

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-data-r2dbc-postgresql/data-r2dbc-postgresql-expected-updated-rows-count

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

# 9.11. Transactions Support

## 9.11.1. DataBase Transactions

To work with database transactions the RxMicro framework introduces a basic transaction model:

```
public interface Transaction {

    ReactiveType commit();

    ReactiveType rollback();

    ReactiveType create(SavePoint savePoint);

    ReactiveType release(SavePoint savePoint);

    ReactiveType rollback(SavePoint savePoint);

    IsolationLevel getIsolationLevel();

    ReactiveType setIsolationLevel(IsolationLevel isolationLevel);
}
```

where `ReactiveType` can be `Mono<Void>`, `Completable` or `CompletableFuture<Void>`.

This basic transaction model has adaptation for all supported reactive libraries:

- If You want to use the **Project Reactor** library:
  - `ReactiveType` will be a `Mono<Void>`.
  - You must use the `io.rxmicro.data.sql.model.reactor.Transaction` interface.
  - A repository method that creates a new transaction must return `Mono<io.rxmicro.data.sql.model.reactor.Transaction>` reactive type:

```
import io.rxmicro.data.sql.model.reactor.Transaction;

@PostgreSQLRepository
public interface BeginReactorTransactionRepository {

    Mono<Transaction> beginTransaction();

    Mono<Transaction> beginTransaction(IsolationLevel isolationLevel);
}
```

- If You want to use the **RxJava** library:
  - `ReactiveType` will be a `Completable`.
  - You must use the `io.rxmicro.data.sql.model.rxjava3.Transaction` interface.

    - A repository method that creates a new transaction must return `Single<io.rxmicro.data.sql.model.rxjava3.Transaction>` reactive type:

```
import io.rxmicro.data.sql.model.rxjava3.Transaction;

@PostgreSQLRepository
public interface BeginRxJava3TransactionRepository {

    Single<Transaction> beginTransaction();

    Single<Transaction> beginTransaction(IsolationLevel isolationLevel);
}
```

- If You want to use the **java.util.concurrent** library:
    - `ReactiveType` will be a `CompletableFuture<Void>`.
    - You must use the `io.rxmicro.data.sql.model.completablefuture.Transaction` interface.
    - A repository method that creates a new transaction must return `CompletableFuture<io.rxmicro.data.sql.model.completablefuture.Transaction>` reactive type:

```
import io.rxmicro.data.sql.model.completablefuture.Transaction;

@SuppressWarnings("unused")
@PostgreSQLRepository
public interface BeginCompletableFutureTransactionRepository {

    CompletionStage<Transaction> beginTransaction1();

    CompletionStage<Transaction> beginTransaction1(IsolationLevel isolationLevel);

    CompletableFuture<Transaction> beginTransaction2();

    CompletableFuture<Transaction> beginTransaction2(IsolationLevel isolationLevel);
}
```

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-data-r2dbc-postgresql/data-r2dbc-postgresql-transactional

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 9.11.2. Concurrent Access Example

The following example demonstrates how developer can use the RxMicro framework to build microservice that requires concurrent access:

```
@PostgreSQLRepository
public interface ConcurrentRepository {

    Mono<Transaction> beginTransaction();

    @Select("SELECT * FROM ${table} WHERE id=? FOR UPDATE")
    Mono<Account> findAccountById(Transaction transaction, long id);

    @Select("SELECT * FROM ${table} WHERE id=? FOR UPDATE")
    Mono<Product> findProductById(Transaction transaction, int id);

    @Update(value = "UPDATE ${table} SET balance=? WHERE id=?", entityClass =
Account.class)
    Mono<Void> updateAccountBalance(Transaction transaction, BigDecimal balance, long
id);

    @Update(value = "UPDATE ${table} SET count=? WHERE id=?", entityClass =
Product.class)
    Mono<Void> updateProductCount(Transaction transaction, int count, long id);

    @Insert
    Mono<Order> createOrder(Transaction transaction, Order order);
}
```

```
public final class ConcurrentBusinessService {

    private final ConcurrentRepository repository =
getRepository(ConcurrentRepository.class);

    /**
     * @return order id if purchase is successful or
     *         error signal if:
     *                  - account not found or
     *                  - product not found or
     *                  - products ran out or
     *                  - money ran out
     */
    public Mono<Long> tryToBuy(final long idAccount,
                               final int idProduct,
                               final int count) {
        return repository.beginTransaction()
                .flatMap(transaction -> repository.findAccountById(transaction,
idAccount)
                        .flatMap(account -> repository.findProductById(transaction,
```

```
idProduct)
                                .flatMap(product ->
                                        tryToBuy(transaction, account, product,
count))
                                .switchIfEmpty(Mono.error(() ->
                                        // product not found
                                        new ProductNotFoundException(idProduct))))
                        // account not found
                        .switchIfEmpty(Mono.error(() -> new
AccountNotFoundException(idAccount)))

.onErrorResume(transaction.createRollbackThenReturnErrorFallback())
                );
    }

    private Mono<Long> tryToBuy(final Transaction transaction,
                                final Account account,
                                final Product product,
                                final int count) {
        if (count <= product.getCount()) {
            final BigDecimal cost =
product.getPrice().multiply(BigDecimal.valueOf(count));
            if (cost.compareTo(account.getBalance()) <= 0) {
                return buy(transaction, account, product, count, cost);
            } else {
                // money ran out
                return Mono.error(new NotEnoughFundsException(cost,
account.getBalance()));
            }
        } else {
            // products ran out
            return Mono.error(new NotEnoughProductCountException(count,
product.getCount()));
        }
    }

    // purchase is successful, returns order id
    private Mono<Long> buy(final Transaction transaction,
                           final Account account,
                           final Product product,
                           final int count,
                           final BigDecimal cost) {
        final int newProductCount = product.getCount() - count;
        final BigDecimal newBalance = account.getBalance().subtract(cost);
        final Order order = new Order(account.getId(), product.getId(), count);

        return repository.updateProductCount(transaction, newProductCount,
product.getId())
                .then(repository.updateAccountBalance(transaction, newBalance,
account.getId())
                        .then(repository.createOrder(transaction, order)
```

```
                                .map(Order::getId)
                                .flatMap(id -> transaction.commit()
                                        .thenReturn(id))
                        )
                );
    }
}
```

For more information, we recommend that You familiarize yourself with the following examples:

- **Concurrent Example Using Spring Reactor Library**;
- **Concurrent Example Using RxJava 3 Library**;

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

# 9.12. Partial Implementation

If the Postgre SQL data repository generated by the `RxMicro Annotation Processor` contains errors, incorrect or non-optimized logic, the developer can use the `Partial Implementation` feature.

This feature allows You to implement methods for the Postgre SQL data repository on Your own, instead of generating them by the RxMicro framework.

To activate this feature, You need to use the `@PartialImplementation` annotation, and specify an abstract class that contains a partial implementation of method(s) for Postgre SQL data repository:

```
@PostgreSQLRepository
①
@PartialImplementation(AbstractDataRepository.class)
public interface DataRepository {

    @Select("SELECT 1 + 1")
    CompletableFuture<Long> generatedMethod();

    CompletableFuture<Long> userDefinedMethod();
}
```

① Using the `@PartialImplementation` annotation, the `AbstractDataRepository` class is specified.

An `AbstractDataRepository` contains the following content:

```
public abstract class AbstractDataRepository extends AbstractPostgreSQLRepository
        implements DataRepository {

    protected AbstractDataRepository(final Class<?> repositoryClass, final
ConnectionPool pool) {
        super(repositoryClass, pool);
    }

    @Override
    public CompletableFuture<Long> userDefinedMethod() {
        return CompletableFuture.completedFuture(100L);
    }
}
```

An abstract class that contains a partial implementation must meet the following requirements:

1. The class must be an `abstract` one.
2. The class must extend the `AbstractPostgreSQLRepository` one.
3. The class must implement the PostgreSQL data repository interface.
4. The class must contain an implementation of all methods that are not generated automatically.

In terms of infrastructure, the repository methods generated and defined by the developer for Postgre SQL data repository do not differ:

```
@Test
void generatedMethod() {
    assertEquals(2L, dataRepository.generatedMethod().join());
}

@Test
void userDefinedMethod() {
    assertEquals(100L, dataRepository.userDefinedMethod().join());
}
```

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-data-r2dbc-postgresql/data-r2dbc-postgresql-partial-implementation

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

## 9.13. Logging

PostgreSQL Data Repositories use the **R2DBC PostgreSQL Driver**, so in order to activate database request logging, You must configure the **R2DBC PostgreSQL Driver Logger**:

`https://github.com/pgjdbc/r2dbc-postgresql#logging`.

For example, if to the `classpath` of the current project add the `jul.properties` resource:

```
io.r2dbc.postgresql.QUERY.level=TRACE
```

,then PostgreSQL Data Repositories will generate request logs to the database while working:

```
[DEBUG] io.r2dbc.postgresql.QUERY : Executing query: SHOW TRANSACTION ISOLATION LEVEL
[DEBUG] io.r2dbc.postgresql.QUERY : Executing query: SELECT 2+2
[DEBUG] io.r2dbc.postgresql.QUERY : Executing query: SELECT first_name, last_name FROM
account WHERE email = $1
```

# 10. Mongo Data Repositories

The RxMicro framework supports creation of dynamic repositories for interaction with databases.

To interact with `Mongo DB`, the RxMicro framework provides the `rxmicro.data.mongo` module.

## 10.1. Basic Usage

To use the `rxmicro.data.mongo` module in the project, the following two steps must be taken:

- Inject the `rxmicro.data.mongo` dependency to the `pom.xml` file:

```
<dependency>
    <groupId>io.rxmicro</groupId>
    <artifactId>rxmicro-data-mongo</artifactId>
    <version>${rxmicro.version}</version>
</dependency>
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-core</artifactId> ①
    <version>${projectreactor.version}</version>
</dependency>
```

① Besides the `rxmicro.data.mongo` dependency, it is also recommended to add the reactive programming library.

Instead of adding a third-party reactive programming library, You can also use the **java.util.concurrent** library built into the JDK, but often in practice the **java.util.concurrent** library's features are not enough.

Therefore, it is recommended to use the **Project Reactor** or the **RxJava** library!

- Add the `rxmicro.data.mongo` module to the `module-info.java` descriptor:

```
module examples.data.mongo.basic {
    requires rxmicro.data.mongo;
    requires reactor.core;       ①
}
```

① When using a third-party reactive programming library, do not forget to add the corresponding module.

Once the `rxmicro.data.mongo` module is added, You can create a data model class and a dynamic repository:

```
public final class Account {

    String firstName;

    String lastName;

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }
}
```

```
①
@MongoRepository(collection = "account")
public interface DataRepository {

    ②
    @Find(query = "{email: ?}")
    Mono<Account> findByEmail(String email);
}
```

① In order for a standard interface to be recognized by the RxMicro framework as a dynamic repository for interaction with `Mongo DB`, this interface should be annotated with the `@MongoRepository` annotation.

② The dynamic repository may contain methods that form a query to the `Mongo DB`. *(The query that used for a request for data uses the JSON format (Specialized request format for `Mongo DB`) and is specified in the annotation parameters. For each operation supported by `Mongo DB`, the RxMicro framework defines a separate annotation: The `@Find` annotation describes the `db.collection.find()`operation.)*

Since the dynamic repository is a RxMicro component, for its testing You need to use the microservice component testing approach:

The common approach recommended for testing dynamic repositories, that interact with `Mongo DB`, is described in the Section 10.3.2, “Test Templates”.

```
@Testcontainers
@RxMicroComponentTest(DataRepository.class)
final class DataRepositoryTest {

    @Container
    private final GenericContainer<?> mongoTestDb =
            new GenericContainer<>("rxmicro/mongo-test-db")
                    .withExposedPorts(27017);

    @WithConfig
    private final MongoConfig mongoConfig = new MongoConfig()
            .setDatabase("rxmicro");

    private DataRepository dataRepository;

    @BeforeEach
    void beforeEach() {
        mongoConfig
                .setHost(mongoTestDb.getHost())
                .setPort(mongoTestDb.getFirstMappedPort());
    }

    @Test
    void Should_find_account() {
        final Account account =
                dataRepository.findByEmail("richard.hendricks@piedpiper.com")
                        .blockOptional()
                        .orElseThrow();

        assertEquals("Richard", account.getFirstName());
        assertEquals("Hendricks", account.getLastName());
    }
}
```

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-data-mongo/data-mongo-basic

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

## 10.2. RxMicro Annotations

The RxMicro framework supports the following RxMicro Annotations:

*Table 16. Supported RxMicro Annotations.*

| Annotation | Description |
|---|---|
| `@Column` | Sets mapping between the field name in the Mongo DB document and the Java model class field name.<br><br>*(By default, the RxMicro framework uses the Java model class field name as the field name in the Mongo DB document. If the name should differ for some reason, (for example, as a field name in the Mongo DB document the keyword Java is used), it should be specified using this annotation!)* |
| `@ColumnMappingStrategy` | Sets the strategy of field name formation in the Mongo DB document, based on the analysis of the Java model class field names.<br><br>*(If this annotation annotates the Java model class, then the set strategy will be used for **all** fields in this class. For example, if You set the default LOWERCASE_WITH_UNDERSCORED strategy, then the parentId field in the Java class will correspond to the parent_id field in the Mongo DB document.)* |
| `@DataRepositoryGeneratorConfig` | Allows You to configure the repository generation process. |
| `@RepeatParameter` | Allows setting mapping between one method parameter marked with this annotation and several universal placeholders that are used in the request to Mongo DB. |
| `@Find` | Denotes a repository method that must execute a `db.collection.find()` operation. |
| `@Aggregate` | Denotes a repository method that must execute a `db.collection.aggregate()` operation. |
| `@Distinct` | Denotes a repository method that must execute a `db.collection.distinct()` operation. |
| `@CountDocuments` | Denotes a repository method that must execute a `db.collection.countDocuments()` operation. |
| `@EstimatedDocumentCount` | Denotes a repository method that must execute a `db.collection.estimatedDocumentCount()` operation. |
| `@Insert` | Denotes a repository method that must execute a `db.collection.insertOne()` operation. |
| `@Update` | Denotes a repository method that must execute a `db.collection.updateOne()` operation. |
| `@Delete` | Denotes a repository method that must execute a `db.collection.deleteOne()` operation. |
| `@DocumentId` | Denotes a model field that must be used as document unique identifier. |

| Annotation | Description |
| --- | --- |
| `@MongoRepository` | Denotes that an interface is a dynamic generated Mongo data repository. |
| `@PartialImplementation` | Denotes an abstract class that contains a partial implementation of the annotated by this annotation a Mongo Data Repository interface. |

# 10.3. Repositories Testing

For successful functional testing of dynamic repositories, that interact with `Mongo DB`, it is required:

- Presence of a script that creates a test database.
- Mechanism for preparing a database for testing: creating a database before starting the test and deleting a database after completing the test.

## 10.3.1. Test Database

A test database was created for testing the `rxmicro.data.mongo`, module features, which are described in this section.

The test database consists of one `account` collection, which contains 6 documents describing the accounts of the test users:

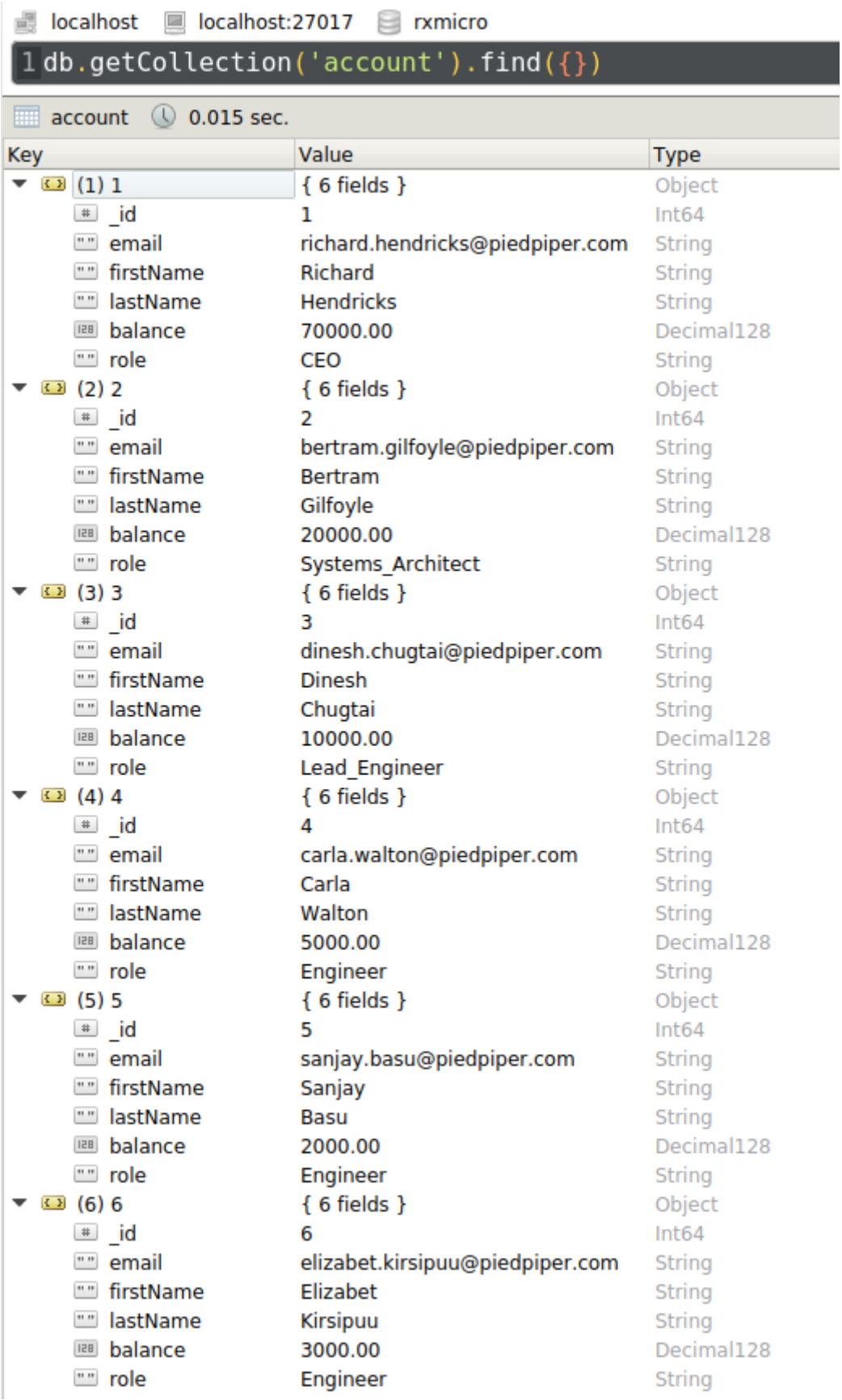



*Figure 15. Mongo Test Database*

Scripts for creating a test database are available at the following link:

https://github.com/rxmicro/rxmicro-usage/blob/master/examples/docker-image-mongo-test-db/src/main/js/

The following classes of Java models correspond to the documents created in the test database:

```
public class Account {
    @DocumentId
    Long id;

    String email;

    String firstName;

    String lastName;

    BigDecimal balance;

    Role role;
}
```

```
public enum Role {

    CEO,

    Systems_Architect,

    Lead_Engineer,

    Engineer
}
```

For ease of studying the `rxmicro.data.mongo` module, You can use the ready-made `Mongo DB` image with the `rxmicro/mongo-test-db` test database.

The source code of the project used as a base for building this `docker` image, is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/docker-image-mongo-test-db`

### 10.3.2. Test Templates

As a mechanism for preparing a database for testing (creating a database before starting the test and deleting a database after completing the test), it is most convenient to use `docker`.

To start `docker` containers in the functional test it is convenient to use the `Testcontainers` Java library:

```
①
@Testcontainers
②
@RxMicroComponentTest(DataRepository.class)
final class DataRepositoryTestTemplate1 {

    ③
    @Container
    private static final GenericContainer<?> MONGO_TEST_DB =
            new GenericContainer<>("rxmicro/mongo-test-db")
                    .withExposedPorts(27017); ④

    ⑤
    @WithConfig
    private static final MongoConfig MONGO_CONFIG = new MongoConfig()
            .setDatabase("rxmicro"); ⑥

    @BeforeAll
    static void beforeAll() {
        MONGO_TEST_DB.start(); ⑦
        MONGO_CONFIG
                .setHost(MONGO_TEST_DB.getHost()) ⑧
                .setPort(MONGO_TEST_DB.getFirstMappedPort());
    }

    private DataRepository dataRepository; ⑨

    // ... test methods must be here

    @AfterAll
    static void afterAll() {
        MONGO_TEST_DB.stop(); ⑩
    }
}
```

① The `@Testcontainers` annotation activates the start and stop of the `docker` containers to be used in this test.

② Since the dynamic repository is a RxMicro component, for its testing You need to use the microservice component testing approach.

③ The `@Container` annotation indicates the `docker` container that will be used in this test. As an image on the basis of which it is necessary to create the `docker` container, the `Mongo DB` ready-

made image with the `rxmicro/mongo-test-db` test database is used.

④ When starting the `docker` container, You need to open the standard port for `Mongo DB`.

⑤ Using the `@WithConfig` annotation, the configuration available only during the test is declared.

⑥ Setting up the configuration to interact with the test database.

⑦ Before running all tests, You must start the `docker` container.

⑧ After starting the `docker` container, You need to read the random IP address and port that will be used when connecting to the running `docker` container.

⑨ When testing microservice components, it is necessary to specify a reference to the component in which the RxMicro framework will inject the tested component.

⑩ After completing all the tests, You must stop the `docker` container.

The main advantage of this template is the speed of testing. Since the `docker` container is created once before starting **all** test methods, the total runtime of all test methods is reduced. The main disadvantage of this template is that if any test method changes the `Mongo DB` state, the following test method may end with an error.

**Therefore, this functional test template should be used for queries to `Mongo DB` that do not change the database state!**

If You need to test methods that change the `Mongo DB` state, You should use another template:

```
①
@Testcontainers
②
@RxMicroComponentTest(DataRepository.class)
final class DataRepositoryTestTemplate2 {

    ③
    @Container
    private final GenericContainer<?> mongoTestDb =
            new GenericContainer<>("rxmicro/mongo-test-db")
                    .withExposedPorts(27017); ④

    ⑤
    @WithConfig
    private final MongoConfig mongoConfig = new MongoConfig()
            .setDatabase("rxmicro"); ⑥

    private DataRepository dataRepository; ⑦

    @BeforeEach
    void beforeEach() {
        mongoConfig
                .setHost(mongoTestDb.getHost()) ⑧
                .setPort(mongoTestDb.getFirstMappedPort());
    }

    // ... test methods must be here
}
```

① The `@Testcontainers` annotation activates the start and stop of the `docker` containers to be used in this test.

② Since the dynamic repository is a RxMicro component, for its testing You need to use the microservice component testing approach

③ The `@Container` annotation indicates the `docker` container that will be used in this test. As an image on the basis of which it is necessary to create the `docker` container, the `Mongo DB` ready-made image with the `rxmicro/mongo-test-db` test database is used.

④ When starting the `docker` container, You need to open the standard port for `Mongo DB`.

⑤ Using the `@WithConfig` annotation, the configuration available only during the test is declared.

⑥ Setting up the configuration to interact with the test database.

⑦ When testing microservice components, it is necessary to specify a reference to the component in which the RxMicro framework will inject the tested component.

⑧ After starting the `docker` container, You need to read the random IP address and port that will be used when connecting to the running `docker` container.

This template **for each test method** will create and drop the `docker` container, which may increase the total runtime of all test methods.

**Therefore, select the most appropriate functional test template based on the requirements of the tested functionality!**

The `Testcontainers` library starts the `docker` container before running the test and stops the `docker` container automatically after completing the test!

So You should start and stop the `docker` container manually only if You want to use one `docker` container for **all** test methods!

## 10.4. Universal Placeholder

The RxMicro framework recommends using the universal placeholder (`?`) as parameter value placeholder in the `Mongo DB` queries:

```
@Find(query = "{email: ?}")
Mono<Account> findByEmail(String email);
```

If this method invoked with the following parameter:
`repository.findByEmail("welcome@rxmicro.io");`,

the RxMicro framework will generate the following `Mongo DB` query:
`{email: "welcome@rxmicro.io"}`.

## 10.5. `@RepeatParameter` Annotation

The universal placeholder (`?`) is the simplest type of placeholders.

But unfortunately, it has one disadvantage: if a query parameter must be repeated, a developer must define a copy of this parameter:

```
@Find(query = "{$or: [{firstName: ?}, {lastName: ?}]}")
Mono<Account> findByFirstOrLastNames(String name1, String name2);
```

The `@RepeatParameter` annotation fixes this disadvantage.

The following code is an equivalent to the code with a copy of the `name` parameter:

```
@Find(query = "{$or: [{firstName: ?}, {lastName: ?}]}")
Mono<Account> findByFirstOrLastNames(@RepeatParameter(2) String name);
```

## 10.6. Mongo Operations

### 10.6.1. `@Find`

The `rxmicro.data.mongo` module supports the `db.collection.find()` operation:

```
@Find(query = "{_id: ?}")
Mono<Account> findById(long id);

@Find(query = "{_id: ?}", projection = "{firstName: 1, lastName: 1}")
Mono<Account> findWithProjectionById(long id);

@Find(query = "{role: ?}", sort = "{role: 1, balance: 1}")
Flux<Account> findByRole(Role role, Pageable pageable);

@Find(query = "{role: ?}", sort = "{role: ?, balance: ?}")
Flux<Account> findByRole(Role role, @RepeatParameter(2) SortOrder sortOrder, Pageable
pageable);
```

For more information, we recommend that You familiarize yourself with the following examples:

- All Supported Return Types;
- All Supported Parameter Types.

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-data-mongo/data-mongo-find`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 10.6.2. `@Aggregate`

The `rxmicro.data.mongo` module supports the `db.collection.aggregate()` operation:

```
public final class Report {

    @DocumentId
    Role id;

    BigDecimal total;
}
```

```
@Aggregate(pipeline = {
        "{ $group : { _id: '$role', total : { $sum: '$balance'}} }",
        "{ $sort: {total: -1, _id: -1} }"
})
Flux<Report> aggregate();
```

For more information, we recommend that You familiarize yourself with the following examples:

- All Supported Return Types;
- All Supported Parameter Types.

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-data-mongo/data-mongo-aggregate`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 10.6.3. `@Distinct`

The `rxmicro.data.mongo` module supports the `db.collection.distinct()` operation:

```
@Distinct(field = "email", query = "{_id: ?}")
Mono<String> getEmailById(long id);

@Distinct(field = "role")
Flux<Role> getAllUsedRoles();
```

For more information, we recommend that You familiarize yourself with the following examples:

- All Supported Return Types;
- All Supported Parameter Types.

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-data-mongo/data-mongo-distinct`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 10.6.4. `@CountDocuments`

The `rxmicro.data.mongo` module supports the `db.collection.countDocuments()` operation:

```
@CountDocuments
Mono<Long> countDocuments();

@CountDocuments(query = "{role:?}", skip = 0, limit = 100)
Mono<Long> countDocuments(Role role);
```

For more information, we recommend that You familiarize yourself with the following examples:

- All Supported Return Types;
- All Supported Parameter Types.

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-data-mongo/data-mongo-count-documents`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 10.6.5. `@EstimatedDocumentCount`

The `rxmicro.data.mongo` module supports the `db.collection.estimatedDocumentCount()` operation:

```
@EstimatedDocumentCount
Mono<Long> estimatedDocumentCount();
```

For more information, we recommend that You familiarize yourself with the following examples:

- All Supported Return Types.

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-data-mongo/data-mongo-estimated-document-count`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 10.6.6. `@Insert`

The `rxmicro.data.mongo` module supports the `db.collection.insertOne()` operation:

```
@Insert
Mono<Void> insert(Account account);
```

For more information, we recommend that You familiarize yourself with the following examples:

- All Supported Return Types.

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-data-mongo/data-mongo-insert`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 10.6.7. `@Update`

The `rxmicro.data.mongo` module supports the `db.collection.updateOne()` operation:

```
@Update
Mono<Boolean> updateEntity(AccountEntity accountEntity);

@Update(filter = "{_id: ?}")
Mono<Void> updateDocument(AccountDocument accountDocument, long id);

@Update(update = "{$set: {balance: ?}}", filter = "{_id: ?}")
Mono<Long> updateById(BigDecimal balance, long id);

@Update(update = "{$set: {balance: ?}}", filter = "{role: ?}")
Mono<UpdateResult> updateByRole(BigDecimal balance, Role role);

@Update(update = "{$set: {balance: ?}}")
Mono<UpdateResult> updateAll(BigDecimal balance);
```

For more information, we recommend that You familiarize yourself with the following examples:

- All Supported Return Types;
- All Supported Parameter Types.

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-data-mongo/data-mongo-update`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 10.6.8. `@Delete`

The `rxmicro.data.mongo` module supports the `db.collection.deleteOne()` and `db.collection.deleteMany()` operations:

```
@Delete
Mono<Boolean> deleteById(Long id);

@Delete(filter = "{role: ?}")
Mono<Long> deleteByRole(Role role);
```

For more information, we recommend that You familiarize yourself with the following examples:

- All Supported Return Types;
- All Supported Parameter Types.

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-data-mongo/data-mongo-delete`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

# 10.7. Partial Implementation

If the Mongo data repository generated by the `RxMicro Annotation Processor` contains errors, incorrect or non-optimized logic, the developer can use the `Partial Implementation` feature.

This feature allows You to implement methods for the Mongo data repository on Your own, instead of generating them by the RxMicro framework.

To activate this feature, You need to use the `@PartialImplementation` annotation, and specify an abstract class that contains a partial implementation of method(s) for Mongo data repository:

```
@MongoRepository(collection = DataRepository.COLLECTION_NAME)
①
@PartialImplementation(AbstractDataRepository.class)
public interface DataRepository {

    String COLLECTION_NAME = "account";

    @CountDocuments
    CompletableFuture<Long> generatedMethod();

    CompletableFuture<Long> userDefinedMethod();
}
```

① Using the `@PartialImplementation` annotation, the `AbstractDataRepository` class is specified.

An `AbstractDataRepository` contains the following content:

```
public abstract class AbstractDataRepository extends AbstractMongoRepository
        implements DataRepository {

    protected AbstractDataRepository(final Class<?> repositoryClass,
                                     final MongoCollection<Document> collection) {
        super(repositoryClass, collection);
    }

    @Override
    public CompletableFuture<Long> userDefinedMethod() {
        return CompletableFuture.completedFuture(100L);
    }
}
```

An abstract class that contains a partial implementation must meet the following requirements:

1. The class must be an `abstract` one.
2. The class must extend the `AbstractMongoRepository` one.
3. The class must implement the Mongo data repository interface.

4. The class must contain an implementation of all methods that are not generated automatically.

In terms of infrastructure, the repository methods generated and defined by the developer for Mongo data repository do not differ:

```
@Test
void generatedMethod() {
    assertEquals(6L, dataRepository.generatedMethod().join());
}

@Test
void userDefinedMethod() {
    assertEquals(100L, dataRepository.userDefinedMethod().join());
}
```

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-data-mongo/data-mongo-partial-implementation`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

## 10.8. Logging

Mongo Data Repositories use the **MongoDB Async Driver**, so in order to activate database request logging to the Mongo DB, You must configure the **MongoDB Async Driver Logger**:

`http://mongodb.github.io/mongo-java-driver/4.0/driver/reference/logging/`.

For example, if to the `classpath` of the current project add the `jul.properties` resource:

```
org.mongodb.driver.protocol.level=TRACE
```

,then Mongo Data Repositories will generate request logs to the Mongo DB while working:

```
2020-03-08 13:15:03.912 [DEBUG] org.mongodb.driver.protocol.command : Sending command
'{"find": "account", "filter": {"_id": 1}, "batchSize": 2147483647, "$db": "rxmicro"}'
with request id 6 to database rxmicro on connection [connectionId{localValue:2,
serverValue:4}] to server localhost:27017

2020-03-08 13:15:03.914 [DEBUG] org.mongodb.driver.protocol.command : Execution of
command with request id 6 completed successfully in 3.11 ms on connection
[connectionId{localValue:2, serverValue:4}] to server localhost:27017
```

# 11. Contexts and Dependency Injection

The `rxmicro.cdi` module is an implementation of the **Dependency Injection** design pattern, that is integrated to the RxMicro framework.

## 11.1. Basic Usage

To use the `rxmicro.cdi` module in the project, the following two steps must be taken:

- Add the `rxmicro-cdi` dependency to the `pom.xml` file:

```
<dependency>
    <groupId>io.rxmicro</groupId>
    <artifactId>rxmicro-cdi</artifactId>
    <version>${rxmicro.version}</version>
</dependency>
```

- Add the `rxmicro.cdi` module to the `module-info.java` descriptor:

```
module examples.cdi.basic {
    requires rxmicro.rest.server.netty;
    requires rxmicro.rest.server.exchange.json;
    requires rxmicro.cdi; ①
}
```

After adding the `rxmicro.cdi` module, You can create a business service:

```
public interface BusinessService {

    String getValue();
}
```

```
public final class BusinessServiceImpl implements BusinessService {

    @Override
    public String getValue() {
        return "IMPL";
    }
}
```

In order to inject a business service implementation, it is necessary to use the `@Inject` annotation:

```
public final class RestController {

    ①
    @Inject
    BusinessService businessService;

    @PATCH("/")
    void handle() {
        System.out.println(businessService.getValue());
    }
}
```

① The usage of the `@io.rxmicro.cdi.Inject` annotation does not differ fundamentally from that of the `@javax.inject.Inject` or `@com.google.inject.Inject` annotations.

The correctness of injection can be checked using REST-based microservice test:

```
@RxMicroRestBasedMicroServiceTest(RestController.class)
final class RestControllerTest {

    private BlockingHttpClient blockingHttpClient;

    private SystemOut systemOut;

    @Test
    void Should_print_to_system_out() {
        blockingHttpClient.patch("/");

        assertEquals("IMPL", systemOut.asString());
    }
}
```

Business service implementation can be injected not only into REST controller, but into any component:

```
public final class BusinessServiceFacade {

    @Inject
    BusinessService businessService;

    public String getValue() {
        return businessService.getValue();
    }
}
```

Since the injection is performed into any component, it is necessary to use the microservice component testing approach:

```
@RxMicroComponentTest(BusinessServiceFacade.class)
final class BusinessServiceFacadeTest {

    private BusinessServiceFacade businessServiceFacade;

    @Test
    void Should_invoke_BusinessService_getValue() {
        assertEquals("IMPL", businessServiceFacade.getValue());
    }
}
```

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-cdi/cdi-basic`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

## 11.2. RxMicro Annotations

The RxMicro framework supports the following RxMicro Annotations:

*Table 17. Supported RxMicro Annotations.*

| Annotation | Description |
|---|---|
| `@Inject` | Indicates the need to inject the component implementation into the annotated class field or method parameter.<br><br>*(Is a synonym of the `@Autowired` annotation, and is recommended for developers who have used JEE or Google Guice as CDI implementation in their previous projects.)* |
| `@Named` | Allows to customize an injection point by specifying a string value or custom annotation. This annotation can also be used to specify a component name.<br><br>*(Is a synonym of the `@Qualifier` annotation, and is recommended for developers who have used JEE or Google Guice as CDI implementation in their previous projects.)* |
| `@Autowired` | Indicates the need to inject the component implementation into the annotated class field or method parameter.<br><br>*(Is a synonym of the `@Inject` annotation, and is recommended for developers who have used Spring DI as CDI implementation in their previous projects.)* |
| `@Qualifier` | Allows to customize an injection point by specifying a string value or custom annotation. This annotation can also be used to specify a component name.<br><br>*(Is a synonym of the `@Named` annotation, and is recommended for developers who have used Spring DI as CDI implementation in their previous projects.)* |
| `@PostConstruct` | Denotes a method, that should be invoked automatically after all dependencies have been injected into the current component.<br><br>*(In its semantics, it completely corresponds to the `@javax.annotation.PostConstruct` annotation.)* |
| `@Factory` | Denotes a factory method or a factory, that creates instances of the specified class. |
| `@Resource` | Indicates the need to inject the external resource into the annotated class field or method parameter. |

## 11.3. All Beans are Singletons!

The RxMicro framework focuses on creating microservice projects. One of the key features of microservices is their simplicity. That's why `singleton` scope was chosen as the main and only one.

Thus, **all CDI components are `singletons`!**

It means that when starting a microservice project, **only one instance** of the component implementation class is created and injected into all necessary injection points.

If it is necessary to inject a separate implementation class instance (`prototype` scope) to each injection point, then You must use the factory that creates instances of the given class!

This behavior can be checked with the following code:

```
public final class BusinessServiceImpl implements BusinessService1, BusinessService2 {
}
```

```
public final class BusinessServiceFacade {

    @Inject
    BusinessService1 businessService1;

    @Inject
    BusinessService2 businessService2;
}
```

```
@RxMicroComponentTest(BusinessServiceFacade.class)
final class BusinessServiceFacadeTest {

    private BusinessServiceFacade facade;

    @Test
    void Should_use_singleton_instance() {
        assertSame(facade.businessService1, facade.businessService2); ①
    }
}
```

① Since the `BusinessServiceImpl` class implements two interfaces, the same implementation instance is injected when injecting different types of interfaces!

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-cdi/cdi-singletons`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

## 11.4. Constructor Injection

The RxMicro framework also supports constructor injection:

```
public final class BusinessServiceFacade {

    private final BusinessService businessService;

    ①
    @Inject
    public BusinessServiceFacade(final BusinessService businessService) {
        System.out.println(businessService.getClass().getSimpleName());
        this.businessService = businessService;
    }

}
```

① To enable constructor injection, it is necessary to create a constructor with parameters and annotate it by the `@Inject` or `@Autowired` annotation.

When using the constructor injection mechanism, this constructor is automatically invoked by the RxMicro framework:

```
@RxMicroComponentTest(BusinessServiceFacade.class)
final class BusinessServiceFacadeTest {

    private SystemOut systemOut;

    @Test
    void Should_support_constructor_injection() {
        assertEquals(BusinessServiceImpl.class.getSimpleName(), systemOut.asString());
    }
}
```

Constructor injection requires more code to be written, but also allows You to create `final` fields as injection points.

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-cdi/cdi-constructor-injection

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

## 11.5. Method Injection

The RxMicro framework also supports injection with `setters` (method injection):

```
public final class BusinessServiceFacade {

    private BusinessService businessService;

    ①
    @Inject
    public void setBusinessService(final BusinessService businessService) {
        System.out.println(businessService.getClass().getSimpleName());
        this.businessService = businessService;
    }

}
```

① To enable the injection with `setters`, it is necessary to create `setter` and annotate it by the `@Inject` or `@Autowired` annotation.

When using the injection mechanism with `setters`, this method is invoked automatically by the RxMicro framework:

```
@RxMicroComponentTest(BusinessServiceFacade.class)
final class BusinessServiceFacadeTest {

    private SystemOut systemOut;

    @Test
    void Should_support_constructor_injection() {
        assertEquals(BusinessServiceImpl.class.getSimpleName(), systemOut.asString());
    }
}
```

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-cdi/cdi-method-injection`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

## 11.6. Ambiguity Resolving

If there are two or more implementations of the same interface in the current project module, the problem of ambiguity resolving may arise in the process of dependencies injection. In the field with the interface type potentially can be injected an implementation of any of the child classes of this interface, that's why such a problem occurs.

In order to solve such problems definitively, the RxMicro framework uses the ambiguity resolving algorithm by default.

### 11.6.1. Default Ambiguity Resolving

If there are two or more implementations of the same interface in the current project module, then the RxMicro framework implicitly sets the name for each implementation. This name corresponds to a simple class name starting with a small letter, for example:

- For the `io.rxmicro.examples.cdi.ProductionBusinessService` class, the name is equal to `productionBusinessService`.
- For the `io.rxmicro.examples.cdi.DevelopmentBusinessService` class, the name is equal to `developmentBusinessService`.
- For the `io.rxmicro.ProductionBusinessService` class, the name is equal to `productionBusinessService`.
- For the `ProductionBusinessService` class, the name is equal to `productionBusinessService`.

The ambiguity resolving problem may occur **only between classes implementing the same interface!**.

This means that if there are two following interfaces in the project:

- `package1.BusinessService` and
- `package2.BusinessService`;

and four implementation classes:

- `package1.impl.ProductionBusinessService implements package1.BusinessService`;
- `package1.impl.DevelopmentBusinessService implements package1.BusinessService`;
- `package2.impl.ProductionBusinessService implements package2.BusinessService`;
- `package2.impl.DevelopmentBusinessService implements package2.BusinessService`;

then despite the same names for different classes:

- `productionBusinessService` for `package1.impl.ProductionBusinessService` and `package2.impl.ProductionBusinessService`;
- `developmentBusinessService` for `package1.impl.DevelopmentBusinessService` and `package2.impl.DevelopmentBusinessService`;

no implementation errors will occur! Everything will work correctly as the same component names are created for **different data types!**

Thus, for `ProductionBusinessService` and `DevelopmentBusinessService` implementation classes, the following names are set accordingly: `productionBusinessService` and `developmentBusinessService`:

```
public final class ProductionBusinessService implements BusinessService {

    @Override
    public String getValue() {
        return "PRODUCTION";
    }
}
```

```
public final class DevelopmentBusinessService implements BusinessService {

    @Override
    public String getValue() {
        return "DEVELOPMENT";
    }
}
```

When injecting implementations, the RxMicro framework reads the name of the class or method parameter field. If the names of the class fields correspond to the names of implementation components, the successful injection is performed:

If there is **only one implementation class**, in the current module, then regardless of the class field names or method parameters, the instance of this class will be successfully injected!

In other words, the ambiguity resolving algorithm is enabled by default **only if there are such ambiguities!**

```
public final class BusinessServiceFacade {

    @Inject
    BusinessService productionBusinessService; ①

    @Inject
    BusinessService developmentBusinessService; ②

    public String getValue(final boolean production) {
        if (production) {
            return productionBusinessService.getValue();
        } else {
            return developmentBusinessService.getValue();
        }
    }
}
```

① In the `productionBusinessService` field, the `ProductionBusinessService` class instance is injected.

② In the `developmentBusinessService` field, the `DevelopmentBusinessService` class instance is injected.

This behavior can be tested with the following test:

```
@RxMicroComponentTest(BusinessServiceFacade.class)
final class BusinessServiceFacadeTest {

    private BusinessServiceFacade businessServiceFacade;

    @Test
    void Should_return_PRODUCTION() {
        assertEquals("PRODUCTION", businessServiceFacade.getValue(true));
    }

    @Test
    void Should_return_DEVELOPMENT() {
        assertEquals("DEVELOPMENT", businessServiceFacade.getValue(false));
    }
}
```

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-cdi/cdi-ambiguty-resolving-by-impl-classname-and-field-name`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 11.6.2. The `@Named` (`@Qualifier`) Annotation Usage

In case if the ambiguity resolving algorithm by default does not meet the needs of a business task, the developer can set up the implementation process using the following annotations: `@Named` or `@Qualifier`:

```
public final class BusinessServiceFacade {

    @Inject
    ①
    @Named("productionBusinessService")
    BusinessService businessService1;

    @Inject
    ②
    @Named("developmentBusinessService")
    BusinessService businessService2;

    public String getValue(final boolean production) {
        if (production) {
            return businessService1.getValue();
        } else {
            return businessService2.getValue();
        }
    }
}
```

① To inject the `ProductionBusinessService` class instance to the `businessService1` field, in the `@Named` annotation parameter, You need to specify the `productionBusinessService` name.
*(This is an implicit name that is set by the RxMicro framework for the `ProductionBusinessService` class!)*

② To inject the `DevelopmentBusinessService` class instance to the `businessService2` field, in the `@Named` annotation parameter, You need to specify the `developmentBusinessService` name.
*(This is an implicit name that is set by the RxMicro framework for the `DevelopmentBusinessService` class!)*

This behavior can be tested with the following test:

```
@RxMicroComponentTest(BusinessServiceFacade.class)
final class BusinessServiceFacadeTest {

    private BusinessServiceFacade businessServiceFacade;

    @Test
    void Should_return_PRODUCTION() {
        assertEquals("PRODUCTION", businessServiceFacade.getValue(true));
    }

    @Test
    void Should_return_DEVELOPMENT() {
        assertEquals("DEVELOPMENT", businessServiceFacade.getValue(false));
    }
}
```

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-cdi/cdi-ambiguty-resolving-by-impl-classname-and-Named-annotation

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

An implicitly created name for an implementation class can be set explicitly using the `@Named` or `@Qualifier` annotations:

```
@Named("Production")
public final class ProductionBusinessService implements BusinessService {

    @Override
    public String getValue() {
        return "PRODUCTION";
    }
}
```

```
@Named("Development")
public final class DevelopmentBusinessService implements BusinessService {

    @Override
    public String getValue() {
        return "DEVELOPMENT";
    }
}
```

When using explicit names for implementation classes, it is necessary to specify these explicit names as the qualifier of an injection point:

```
public final class BusinessServiceFacade {

    @Inject
    ①
    @Named("Production")
    BusinessService businessService1;

    @Inject
    ②
    @Named("Development")
    BusinessService businessService2;

    public String getValue(final boolean production) {
        if (production) {
            return businessService1.getValue();
        } else {
            return businessService2.getValue();
        }
    }
}
```

① To inject the `ProductionBusinessService` class instance to the `businessService1` field, in the `@Named` annotation parameter, You need to specify the `Production` name.
*(This name is explicitly set for the `ProductionBusinessService` class, using the `@Named` annotation!)*

② To inject the `DevelopmentBusinessService` class instance to the `businessService2` field, in the `@Named` annotation parameter, You need to specify the `Development` name.
*(This name is explicitly set for the `DevelopmentBusinessService` class, using the `@Named` annotation!)*

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-cdi/cdi-ambiguty-resolving-by-Named-implementation-and-Named-field

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 11.6.3. Custom Annotations Usage

When using string names for implementation classes and injection points, the developer may make a mistake. Since the compiler does not check the specified names, the error can be detected only during runtime.

If such a situation is unacceptable, custom annotations should be used as qualifiers:

```
@Documented
@Retention(CLASS)
@Target({FIELD, METHOD, TYPE})
①
@Named("")
public @interface EnvironmentType {

    Type value(); ②

    enum Type {

        PRODUCTION,

        DEVELOPMENT
    }
}
```

① For a custom annotation to be defined by the RxMicro framework as a qualifier, it must be annotated by the `@Named` or `@Qualifier` annotation with an empty string value.

② To control the component names with the compiler, it is recommended to use enumerations.

After creating the custom annotation that serves as a qualifier, it is necessary to annotate the implementation classes by it

```
@EnvironmentType(EnvironmentType.Type.PRODUCTION)
public final class ProductionBusinessService implements BusinessService {

    @Override
    public String getValue() {
        return "PRODUCTION";
    }
}
```

```
@EnvironmentType(EnvironmentType.Type.DEVELOPMENT)
public final class DevelopmentBusinessService implements BusinessService {

    @Override
    public String getValue() {
        return "DEVELOPMENT";
    }
}
```

and injection points:

```
public final class BusinessServiceFacade {

    @Inject
    ①
    @EnvironmentType(EnvironmentType.Type.PRODUCTION)
    BusinessService businessService1;

    @Inject
    ②
    @EnvironmentType(EnvironmentType.Type.DEVELOPMENT)
    BusinessService businessService2;

    public String getValue(final boolean production) {
        if (production) {
            return businessService1.getValue();
        } else {
            return businessService2.getValue();
        }
    }
}
```

When using custom annotations, the result of ambiguity resolving will be the same as when using string names:

```
@RxMicroComponentTest(BusinessServiceFacade.class)
final class BusinessServiceFacadeTest {

    private BusinessServiceFacade businessServiceFacade;

    @Test
    void Should_return_PRODUCTION() {
        assertEquals("PRODUCTION", businessServiceFacade.getValue(true));
    }

    @Test
    void Should_return_DEVELOPMENT() {
        assertEquals("DEVELOPMENT", businessServiceFacade.getValue(false));
    }
}
```

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-cdi/cdi-ambiguty-resolving-using-custom-Named-annotation`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

## 11.7. Dependency Injection Using Spring Style

The developers with a rich experience in using Spring in their previous projects can instead of the following annotations: `@Inject` and `@Named` use the `@Autowired` and `@Qualifier` annotations respectively, which are absolutely similar in features:

```
@Qualifier("Production")
public final class ProductionBusinessService implements BusinessService {

    @Override
    public String getValue() {
        return "PRODUCTION";
    }
}
```

```
@Qualifier("Development")
public final class DevelopmentBusinessService implements BusinessService {

    @Override
    public String getValue() {
        return "DEVELOPMENT";
    }
}
```

```
public final class BusinessServiceFacade {

    @Autowired
    ①
    @Qualifier("Production")
    BusinessService businessService1;

    @Autowired
    ②
    @Qualifier("Development")
    BusinessService businessService2;

    public String getValue(final boolean production) {
        if (production) {
            return businessService1.getValue();
        } else {
            return businessService2.getValue();
        }
    }
}
```

① To inject the `ProductionBusinessService` class instance in the `businessService1` field, it is necessary in the `@Qualifier` annotation parameter specify the `Production` name.

*(This is the name that is set explicitly for the* `ProductionBusinessService` *class by the* `@Qualifier` *annotation!)*

② To inject the `DevelopmentBusinessService` class instance in the `businessService2` field, it is necessary in the `@Qualifier` annotation parameter specify the `Development` name.
*(This is the name that is set explicitly for the* `DevelopmentBusinessService` *class by the* `@Qualifier` *annotation!)*

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-cdi/cdi-spring-style`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

## 11.8. @PostConstruct

If You need to run some code while creating a class instance, Java provides a special method for doing so, namely the constructor. However, when using the dependency injection mechanisms, the dependencies are injected after creating an instance, and accordingly after invoking the constructor. *(Except for the constructor injection mechanism!)*. In order to run some code while creating an instance, but only after introducing all dependencies into this instance, the RxMicro framework provides a special `@PostConstruct` annotation.

Thus, if there is a class method, annotated by the `@PostConstruct` annotation, then this method is automatically invoked after all dependencies are injected into the created instance of this class.

```
public final class BusinessService2Facade {

    @Inject
    BusinessService businessService;

    @PostConstruct
    void anyName() { ①
        System.out.println(businessService.getClass().getSimpleName());
    }
}
```

① After injection of the implementation class instance in the `businessService` field, the RxMicro framework will automatically invoke the `anyName` method, since this method is annotated by the `@PostConstruct` annotation.

For the convenience of developers, the RxMicro framework introduces an additional convention:

**If there is a method with the `postConstruct` name in the class, this method may not be annotated by the `@PostConstruct` annotation!**

The method with the specified name will be invoked automatically after all dependencies are injected:

```
public final class BusinessService1Facade {

    @Inject
    BusinessService businessService;

    void postConstruct() { ①
        System.out.println(businessService.getClass().getSimpleName());
    }

}
```

Thus, for any method in the class to be defined by the RxMicro framework as the method to be invoked automatically after all dependencies are injected, it is necessary to:

- to annotate this method by the `@PostConstruct` annotation;
- or has the predefined `postConstruct` name.

The `postConstruct` method must meet the following requirements:

- This method should be a single method in the class.
- The method must not be `static`.
- The method must not be `abstract`.
- The method must be non-`native`.
- The method must not be `synchronized`.
- The method must not contain parameters.
- The method must return the `void` type.

The facts of invoking the `postConstruct` and `anyName` methods can be checked using the following tests:

```
@RxMicroComponentTest(BusinessService1Facade.class)
final class BusinessService1FacadeTest {

    private SystemOut systemOut;

    @Test
    void Should_invoke_postConstruct_method() {
        assertEquals(BusinessServiceImpl.class.getSimpleName(), systemOut.asString());
    }
}
```

```
@RxMicroComponentTest(BusinessService2Facade.class)
final class BusinessService2FacadeTest {

    private SystemOut systemOut;

    @Test
    void Should_invoke_postConstruct_method() {
        assertEquals(BusinessServiceImpl.class.getSimpleName(), systemOut.asString());
    }
}
```

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-cdi/cdi-post-construct

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

In many frameworks implementing the **Dependency Injection** design template, besides the `postConstruct` methods, there are also the `preDestroy` methods. These methods are invoked automatically, and usually clear resources when deleting an instance from the CDI container.

For a microservice project that uses the RxMicro framework, there is no need for the `preDestroy` methods. Since the instances are removed from the CDI container when the java process ends. And upon completion of any process in the operating system, the operating system automatically clears **all** occupied resources!

# 11.9. RxMicro Components Injection

## 11.9.1. Basic Usage

Besides custom classes, the RxMicro framework supports the RxMicro components injection.

For example, if a declarative REST client is declared in the project:

```
@RestClient
public interface RESTClient {

    @PATCH("/")
    CompletableFuture<Void> patch();
}
```

then instead of getting an explicit reference to the implementation class instance using the `RestClientFactory.getRestClient(Class<?>)`, the developer can use the dependency injection mechanism:

```
public final class BusinessServiceFacade {

    ①
    @Inject
    RESTClient restClient;

    ②
    @Inject
    RestClientConfig config;

    void postConstruct() {
        System.out.println(restClient.getClass().getSimpleName());
        System.out.println(config.getClass().getSimpleName());
    }
}
```

① To get a reference to the REST client implementation class, the developer can use the `@Inject` annotation.

② Besides injecting REST clients, the RxMicro framework also supports the configuration injection. *(A list of all supported RxMicro components that can be injected is described in the Section 11.9.2, "All Supported RxMicro Components".)*

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-cdi/cdi-rxmicro-component`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 11.9.2. All Supported RxMicro Components

This section describes all supported RxMicro components that can be injected in any class using the injection mechanisms.

*Table 18. All Supported RxMicro Components.*

| Name | Feature |
|---|---|
| Config instance. | Any class that extends the basic configuration class: `Config`.<br><br>*(For example:* `MongoConfig`, `HttpClientConfig`, `HttpServerConfig`, *etc.)* |
| Mongo repository. | The interface annotated by the `@MongoRepository` annotation. |
| Mongo client. | Predefined type: `com.mongodb.reactivestreams.client.MongoClient` |
| PostgreSQL repository. | The interface annotated by the `@PostgreSQLRepository` annotation. |
| R2DBC connection factory. | Predefined type: `io.r2dbc.spi.ConnectionFactory`. |
| R2DBC connection pool. | Predefined type: `io.r2dbc.pool.ConnectionPool`. |
| REST client. | The interface annotated by the `@RestClient` annotation. |

The source code of the project demonstrating all supported for injection RxMicro components is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-cdi/cdi-all-rxmicro-components`

# 11.10. Factory Method

When using the dependency injection mechanisms, the RxMicro framework creates instances of the specified classes and injects references to them to injection points. For successful implementation of this behavior, each class, the instance of which should be injected, must contain an accessible constructor without parameters or a constructor annotated by the `@Inject` or `@Autowired` annotation.

In other words, the RxMicro framework determines the instance of which class should be created and creates this instance automatically at the start of the CDI container. If it is necessary to get more control over creation of the implementation instance, it is necessary to use the Factory Method template:

```
public final class BusinessServiceFacade {

    @Inject
    BusinessService businessService;

    ①
    @Factory
    static BusinessServiceFacade create() {
        System.out.println("Use factory method");
        return new BusinessServiceFacade();
    }

    ②
    private BusinessServiceFacade() {
    }

    void postConstruct() {
        System.out.println(businessService.getClass().getSimpleName());
    }

}
```

① The class must contain the static method annotated by the `@Factory`annotation.

② The private constructor restricts the possibility of creating the instance of this class externally. Thus, the instance of this class can only be created using the `create()` factory method.

If the RxMicro framework detects a method in the class, annotated by the `@Factory`annotation, then this method is used instead of the constructor when creating the instance of this class:

```
@RxMicroComponentTest(BusinessServiceFacade.class)
final class BusinessServiceFacadeTest {

    private SystemOut systemOut;

    @Test
    void Should_support_constructor_injection() {
        assertEquals(
                List.of(
                        "Use factory method",
                        BusinessServiceImpl.class.getSimpleName()
                ),
                systemOut.asStrings()
        );
    }
}
```

The factory method must meet the following requirements:

- The method must be `static`.
- The method must be non-`native`.
- The method must not be `synchronized`.
- The method must return the class instance in which it is declared.
- The method must not contain parameters.

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-cdi/cdi-factory-method

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

# 11.11. Factory Class

The RxMicro framework supports creation of factory classes, that can be used to create instances of other types.

By using factory classes, it is possible to get the following benefits:

- Create dynamic classes. *(For example, using the* `Proxy` *class.)*
- Implement a `prototype` scope.

To create a factory class, it is necessary:

- Create a class implementing the `Supplier` interface.
- Annotate this class by the `@Factory`annotation.
- Implement the `get` method, which should return the instance of the created class.

For example, to create a dynamic class, it is necessary to use the following factory class:

```
@Factory
public final class BusinessServiceFactoryImpl implements Supplier<BusinessService> {

    @Override
    public BusinessService get() {
        final Object o = new Object();
        return (BusinessService) Proxy.newProxyInstance(
                BusinessService.class.getClassLoader(),
                new Class[]{BusinessService.class},
                (proxy, method, args) -> {
                    if ("getValue".equals(method.getName())) {
                        return "PROXY";
                    } else {
                        return method.invoke(o, args);
                    }
                }
        );
    }
}
```

Injection of an instance created by the factory class does not differ from injection of an instance automatically created by the RxMicro framework:

```
public final class BusinessServiceFacade {

    @Inject
    BusinessService businessService;

    public String getValue() {
        return businessService.getValue();
    }

}
```

When invoking the `getValue` method, the dynamic class returns a predefined value:

```
@RxMicroComponentTest(BusinessServiceFacade.class)
final class BusinessServiceFacadeTest {

    private BusinessServiceFacade businessServiceFacade;

    @Test
    void Should_support_factory_classes() {
        assertEquals("PROXY", businessServiceFacade.getValue());
    }
}
```

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-cdi/cdi-factory-class`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

# 11.12. External resource injection

## 11.12.1. Basic Usage

Indicates the need to inject the external resource into the annotated class field or method parameter.

External resource is:

- File.
- Directory.
- Classpath resource.
- URL Resource.
- etc.

**The RxMicro framework uses blocking API to inject resource during startup!**

To inject external resource it is necessary to use `@Resource` annotation:

```
public class Component {

    @Resource("classpath:resources.properties")
    Map<String, String> resources;

    public Map<String, String> getResources() {
        return resources;
    }
}
```

Content of the `classpath:resources.properties` is:

```
name=value
```

After startup external resource is injected successful:

```
@RxMicroComponentTest(Component.class)
final class ComponentTest {

    private Component component;

    @Test
    void Should_the_resources_be_injected() {
        assertEquals(Map.of("name", "value"), component.getResources());
    }
}
```

### 11.12.2. Additional Info

To customize resource injection mechanism it is necessary to use `@Resource` annotation.

Example of valid resource paths:

- `/home/rxmicro/config.json`;
- `/home/rxmicro/config.properties`;
- `file:///home/rxmicro/config.json`;
- `file:///home/rxmicro/config.properties`;
- `classpath:config.json`;
- `classpath:config.properties`;

If converter class is not specified, the RxMicro framework tries to detect valid resource converter automatically using the following algorithm:

- `io.rxmicro.cdi.resource.ClasspathJsonArrayResourceConverter` is used if:
  - Resource path starts with `classpath:` prefix.
  - Resource path ends with `json` extension.
  - Annotated by `@Resource` annotation field has `java.util.List<Object>` type.
- `io.rxmicro.cdi.resource.ClasspathJsonObjectResourceConverter` is used if:
  - Resource path starts with `classpath:` prefix.
  - Resource path ends with `json` extension.
  - Annotated by `@Resource` annotation field has `java.util.Map<String, Object>` type.
- `io.rxmicro.cdi.resource.ClasspathPropertiesResourceConverter` is used if:
  - Resource path starts with `classpath:` prefix.
  - Resource path ends with `properties` extension.
- `io.rxmicro.cdi.resource.FileJsonArrayResourceConverter` is used if:
  - Resource path starts with `file://` prefix or prefix is missing.
  - Resource path ends with `json` extension.
  - Annotated by `@Resource` annotation field has `java.util.List<Object>` type.
- `io.rxmicro.cdi.resource.FileJsonObjectResourceConverter` is used if:
  - Resource path starts with `file://` prefix or prefix is missing.
  - Resource path ends with `json` extension.
  - Annotated by `@Resource` annotation field has `java.util.Map<String, Object>` type.
- `io.rxmicro.cdi.resource.FilePropertiesResourceConverter` is used if:
  - Resource path starts with `file://` prefix or prefix is missing.
  - Resource path ends with `properties` extension.

# 11.13. Optional Injection

By default, all injection points are required. Thus, if during the process of dependencies injection, the RxMicro framework does not find a suitable instance, an error will occur.

If the current project allows the situation when a suitable instance may be missing, then the `optional injection` mode should be used:

```
public final class BusinessServiceFacade {

    ①
    @Inject(optional = true)
    BusinessService productionBusinessService = null;

    ②
    @Autowired(required = false)
    BusinessService developmentBusinessService = new BusinessService() {
        @Override
        public String toString() {
            return "DefaultImpl";
        }
    };

    void postConstruct() {
        System.out.println(productionBusinessService);
        System.out.println(developmentBusinessService);
    }
}
```

① When using the `@Inject` annotation, the `optional = true` parameter must be set to enable the `optional injection` mode.

② When using the `@Autowired` annotation, the `required = false` parameter must be set to enable the `optional injection` mode.

If the `optional injection` mode is enabled, the RxMicro framework uses the following injection algorithm:

1. If the dependency is found, it will be successfully injected.
2. If there's no dependency, nothing happens.
   *(In this case, the behaviour appears to be as if the field is not annotated by any annotation!)*

The correctness of this algorithm can be checked with the following test:

```
@RxMicroComponentTest(BusinessServiceFacade.class)
final class BusinessServiceFacadeTest {

    private SystemOut systemOut;

    @Test
    void Should_support_optional_injection() {
        assertEquals(
                List.of(
                        "null",
                        "DefaultImpl"
                ),
                systemOut.asStrings()
        );
    }
}
```

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-cdi/cdi-optional-injection

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

## 11.14. Multibindings

The RxMicro framework supports **Multibindings**.

**Multibindings** is a function of the CDI container, that allows You to find all implementations of this type and inject them.

For example, if there are two implementation classes of the `BusinessService` interface:

```
public final class ProductionBusinessService implements BusinessService {

    @Override
    public String getValue() {
        return "PRODUCTION";
    }
}
```

```
public final class DevelopmentBusinessService implements BusinessService {

    @Override
    public String getValue() {
        return "DEVELOPMENT";
    }
}
```

then the instances of these classes can be injected by the RxMicro framework into the `java.util.Set` type field:

```
public final class BusinessServiceFacade {

    @Inject
    Set<BusinessService> businessServices;

    void postConstruct() {
        System.out.println(
                businessServices.stream()
                        .map(s -> s.getClass().getSimpleName())
                        .collect(joining(", "))
        );
    }
}
```

After successful injection, the `businessServices` field will contain the `ProductionBusinessService` and `DevelopmentBusinessService` class instances:

```
@RxMicroComponentTest(BusinessServiceFacade.class)
final class BusinessServiceFacadeTest {

    private SystemOut systemOut;

    @Test
    void Should_support_multibinder() {
        assertEquals(
                "DevelopmentBusinessService, ProductionBusinessService",
                systemOut.asString()
        );
    }
}
```

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-cdi/cdi-multibinder

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

# 12. Monitoring

TODO

# 12.1. Request Id

To track an HTTP request when using a microservice architecture, the RxMicro framework provides an HTTP request identification feature.

## 12.1.1. Basic Rules

If the current request is identified, the provided unique id is used during the life-cycle of the current request. If the request is not identified, the RxMicro framework generates a unique id, which is further used in the life-cycle of the current request.

To store the request id, the `Request-Id` HTTP additional header is used.

If the HTTP request id is necessary for business logic, a separate field in the HTTP request Java model must be created:

```
public final class Request {

    ①
    //@Header(HttpHeaders.REQUEST_ID)
    ②
    @RequestId
    String requestId;

    public String getRequestId() {
        return requestId;
    }
}
```

① For convenience, instead of specifying an HTTP header using the `@Header` annotation and the `Request-Id` name,

② the special `@RequestId` annotation can be used, which is an alternative to the `@Header(HttpHeaders.REQUEST_ID)` configuration.

Having declared a field in the HTTP request Java model, You can use its value in the handler:

```
final class MicroService {

    @GET("/")
    void handle(final Request request) {
        System.out.println(request.getRequestId()); ①
    }
}
```

① Displaying the value of the current request id in the console.

The following test describes the basic requirements for the behavior of the request id function:

```
@RxMicroRestBasedMicroServiceTest(MicroService.class)
final class MicroServiceTest {

    @WithConfig
    private static final RestServerConfig CONFIG = new RestServerConfig()
            .setRequestIdGeneratorProvider(restServerConfig -> () -> "TestRequestId")
⑥
            .setDevelopmentMode(true);

    private BlockingHttpClient blockingHttpClient;

    private SystemOut systemOut;

    @Test
    void Should_generate_RequestId_automatically() {
        final ClientHttpResponse response = blockingHttpClient.get("/");
        assertEquals("TestRequestId", systemOut.asString()); ①
        assertEquals("TestRequestId", response.getHeaders().getValue(REQUEST_ID));  ②
    }

    @Test
    void Should_use_provided_RequestId() {
        final ClientHttpResponse response =
                blockingHttpClient.get("/", HttpHeaders.of(REQUEST_ID, "Qwerty")); ③

        assertEquals("Qwerty", systemOut.asString()); ④
        assertEquals("Qwerty", response.getHeaders().getValue(REQUEST_ID));  ⑤
    }
}
```

① For HTTP requests without id, the RxMicro framework must generate a unique id automatically. *(This id can be used in the HTTP request handler body.)*

② Each HTTP response must contain the required `Request-Id` HTTP header with the value of the generated request id.
*(If the configuration parameter `RestServerConfig.returnGeneratedRequestId = false`, the HTTP response will not contain the `Request-Id` header.)*

③ If the current HTTP request already contains an id, the RxMicro framework must use it instead of generating a new value.

④ The id set by the client can be used in the HTTP request handler body.

⑤ Each HTTP response must contain the required `Request-Id` HTTP header with a client specified request id value.
*(If the configuration parameter `RestServerConfig.returnGeneratedRequestId = false`, the HTTP response will not contain the `Request-Id` header.)*

⑥ For test purposes it is necessary to set deterministic request id provider with constant value.

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-rest-controller/rest-controller-request-id`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 12.1.2. Supported Generator Providers

The RxMicro framework provides the following predefined request id generator providers.

*Table 19. Generator name and its implementation class.*

| Type | Implementation class | Description |
|---|---|---|
| `UUID_128_BIT` | `RandomRequestIdGenerator` | **Generates unique 16 bytes (128 bits) request id**<br><br>According to specification this generator is similar to the `UUID.randomUUID()` generation. |
| `RANDOM_96_BIT` | `RandomRequestIdGenerator` | **Generates unique 12 bytes (96 bits) request id** |
| `PARTLY_RANDOM_96_BIT` | `PartlyRandom96BitsRequestIdGenerator` | **Generates unique 12 bytes (96 bits) request id**<br><br>Each request id contains 52 random and 44 deterministic bits. |
| `DETERMINISTIC_96_BIT` | `Deterministic96BitsRequestIdGenerator` | **Generates unique 12 bytes (96 bits) request id**<br><br>Each request id contains 44 deterministic bits + 24 incremental counter bits + 28 checksum bits. |
| `DEFAULT_96_BIT` | `PartlyRandom96BitsRequestIdGenerator`, `Deterministic96BitsRequestIdGenerator` | **Generates unique 12 bytes (96 bits) request id**<br><br>This is default request id generator provider.<br><br>By default this request id generator provider tries to use `PARTLY_RANDOM_96_BIT` request id generator. But if during predefined timeout the `java.security.SecureRandom` instance can't generate random bytes, the `DETERMINISTIC_96_BIT` request id generator will be used. |

To change the request id generator provider, You must use the `RestServerConfig` configuration class:

```
new Configs.Builder()
        .withConfigs(new RestServerConfig()
                .setRequestIdGeneratorProvider(
                        PredefinedRequestIdGeneratorProvider.UUID_128_BITS ①
                )
        )
        .build();
```

or

```
export rest-server.requestIdGeneratorProvider=\
@io.rxmicro.rest.server.PredefinedRequestIdGeneratorProvider:UUID_128_BIT
```

or using any other supported config types

To get additional information about how custom type can be used as valid config parameters read Section 4.8.6.2, “Supported Custom Parameter Types”

## 12.2. Request Tracing Usage Example

The following example demonstrates how request tracing feature can be used:

```
public final class Request implements RequestIdSupplier { ①

    ②
    @RequestId
    String requestId;

    @PathVariable
    Long id;

    @Override
    public String getRequestId() {
        return requestId;
    }

    public Long getId() {
        return id;
    }
}
```

```
public final class AccountController {

    @Inject
    private AccountService accountService;

    @GET("/account/${id}")
    Mono<Response> findById(final Request request) {
        return accountService.findById(request);
    }
}
```

```
public final class AccountService {

    ①
    private static final Logger LOGGER =
LoggerFactory.getLogger(AccountService.class);

    @Inject
    private AccountRepository accountRepository;

    public Mono<Response> findById(final Request request) {
        LOGGER.debug(
                request, ②
                "Finding account by id=?...", request.getId()
        );
        return accountRepository.findById(request, request.getId())
                .switchIfEmpty(Mono.error(() -> {
                    LOGGER.error(
                            request, ③
                            "Account not found by id=?!", request.getId()
                    );
                    return new AccountNotFound404Exception();
                }))
                .map(account -> {
                    LOGGER.debug(
                            request, ④
                            "Account exists by id=?: ?",
                            request.getId(), account
                    );
                    return new Response(
                            account.getEmail(), account.getFirstName(),
account.getLastName()
                    );
                });
    }
}
```

```
@PostgreSQLRepository
public interface AccountRepository {

    @Select("SELECT * FROM ${table} WHERE ${by-id-filter}")
    Mono<Account> findById(RequestIdSupplier requestIdSupplier, long id); ①
}
```

The `jul.properties' config classpath resource:

```
①
io.rxmicro.logger.jul.PatternFormatter.pattern=%date{HH:mm:ss.SSS} {%id} [%level]
%logger{0}: %message%n

io.rxmicro.rest.server.level=TRACE
io.rxmicro.examples.monitoring.request.tracing.level=TRACE
```

Invoke the test request:

```
:$ curl -v localhost:8080/account/1
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 8080 (#0)
> GET /account/1 HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.58.0
> Accept: /
>
< HTTP/1.1 200 OK
< Content-Length: 88
< Server: RxMicro-NettyServer/0.7-SNAPSHOT
< Date: Wed, 25 Nov 2020 17:30:51 GMT
< Content-Type: application/json
< Request-Id: AkinnfVzx1752012 ①
<
* Connection #0 to host localhost left intact
{"email":"richard.hendricks@piedpiper.com","firstName":"Richard","lastName":"Hendricks
"}
```

The log output:

```
19:30:50.494 {null} [TRACE] NettyClientConnectionController: Client connection
created: Channel=eaaf0399, IP=/127.0.0.1:33312

19:30:50.589 {AkinnfVzx1752012} [TRACE] NettyRequestHandler: HTTP request:
(Channel=eaaf0399, IP=/127.0.0.1:33312):
GET /account/1 HTTP/1.1
Host: localhost:8080
User-Agent: curl/7.58.0
Accept: /
content-length: 0


19:30:50.591 {AkinnfVzx1752012} [DEBUG] AccountService: Finding account by id=1...

19:30:50.998 {AkinnfVzx1752012} [TRACE] AccountRepository: Execute SQL 'SELECT id,
email, first_name, last_name FROM account WHERE id = $1' with params: [1] using
connection: class='PooledConnection', id='c1275307250'...

19:30:51.074 {AkinnfVzx1752012} [TRACE] AccountRepository: SQL 'SELECT id, email,
first_name, last_name FROM account WHERE id = $1' with params: [1] executed successful

19:30:51.148 {AkinnfVzx1752012} [DEBUG] AccountService: Account exists by id=1:
Account{id=1, email=richard.hendricks@piedpiper.com, firstName=Richard,
lastName=Hendricks}

19:30:51.166 {AkinnfVzx1752012} [TRACE] AccountRepository: Connection closed:
class='PooledConnection', id='c1275307250', signal='onComplete'

19:30:51.167 {AkinnfVzx1752012} [TRACE] NettyRequestHandler: HTTP response:
(Channel=eaaf0399, Duration=588.093042ms):
HTTP/1.1 200 OK
Content-Length: 88
Server: RxMicro-NettyServer/0.7-SNAPSHOT
Date: Wed, 25 Nov 2020 17:30:51 GMT
Content-Type: application/json
Request-Id: AkinnfVzx1752012

{"email":"richard.hendricks@piedpiper.com","firstName":"Richard","lastName":"Hendricks
"}
```

For more information, we recommend that You familiarize yourself with the following examples:

- **Request tracing usage example**;

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

# 13. Java EcoSystem Integration

## 13.1. Unnamed Modules Support

On September 21, 2017, Java 9 has introduced JPMS.

There are four types of modules in the JPMS:

- **System Modules** - There are the Java SE and JDK modules.
  *(For example* `java.base`, `java.logging`, `java.net.http`, *etc)*
- **Application Modules** - The default modules, that are built using JPMS.
  They are `named` and defined in the compiled `module-info.class` file included in the assembled JAR.
- **Automatic Modules** - The modules that are created automatically from all unmodularized jar files at the module path. The name of the modules will be derived from the name of the appropriate JAR files.
- **Unnamed Modules** - The modules that are created automatically if JAR files are loaded onto the classpath instead of the module path. Unnamed modules disable the JPMS and allow to run any code that does not support the JPMS on Java 9 or higher.

Currently, the majority of existing frameworks and libraries are focused on JDK 8 and do not support JPMS.

To support compatibility with libraries that do not support JPMS, Java 9 provides the `unnamed modules` mechanism.

The feature of this mechanism is that the code written in Java 9 and higher does not use the `module-info.java` module descriptor, and uses `class-path` instead of `module-path`. For integration with libraries that do not support JPMS, the RxMicro framework allows You to enable the `unnamed module` mode for the `RxMicro Annotation Processor`.

To enable this mode, You need to add the `RX_MICRO_BUILD_UNNAMED_MODULE` configuration parameter to the `maven-compiler-plugin` settings:

```
<plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>${maven-compiler-plugin.version}</version>
    <configuration>
        <release>11</release>
        <!-- ... -->
        <compilerArgs>
            <arg>-ARX_MICRO_BUILD_UNNAMED_MODULE=true</arg> ①
        </compilerArgs>
    </configuration>
    <!-- ... -->
</plugin>
```

① The `RxMicro Annotation Processor` will use the `unnamed module` mode for the current microservice project.

### 13.1.1. Uber Jar

An uber JAR (also known as a fat JAR or JAR with dependencies) is a JAR file that contains not only a Java program, but embeds its dependencies as well. This means that the JAR functions as an "all-in-one" distribution of the software, without needing any other Java code.

*(You still need a Java runtime, and an underlying operating system, of course.)*

To build a microservice project in the form of `uber.jar`, You need to enable the `unnamed module` mode and configure the `maven-shade-plugin`.

#### 13.1.1.1. Enable the `unnamed module` Mode

To enable the `unnamed module` mode, add the `RX_MICRO_BUILD_UNNAMED_MODULE` compiler option to `maven-compiler-plugin`:

```
<plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>${maven-compiler-plugin.version}</version>
    <configuration>
        <release>11</release>
        <!-- ... -->
        <compilerArgs>
            <arg>-ARX_MICRO_BUILD_UNNAMED_MODULE=true</arg> ①
        </compilerArgs>
    </configuration>
    <!-- ... -->
</plugin>
```

① The `RxMicro Annotation Processor` will use the `unnamed module` mode for the current microservice project.

#### 13.1.1.2. Configuration of `maven-shade-plugin`

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-shade-plugin</artifactId>
    <version>${maven-shade-plugin.version}</version>
    <executions>
        <execution>
            <phase>package</phase>
            <goals>
                <goal>shade</goal>
            </goals>
            <configuration>
                <createDependencyReducedPom>false</createDependencyReducedPom>
                <transformers>
                    <transformer implementation=

"org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                        <manifestEntries>
                            <Main-Class>

io.rxmicro.examples.unnamed.module.uber.jar.HelloWorldMicroService ①
                            </Main-Class>
                        </manifestEntries>
                    </transformer>
                </transformers>
                <filters>
                    <filter>
                        <artifact>*:*</artifact>
                        <excludes>
                            <exclude>ModuleInfo.*</exclude> ②
                        </excludes>
                    </filter>
                </filters>
                <minimizeJar>true</minimizeJar> ③
            </configuration>
        </execution>
    </executions>
</plugin>
```

① It is necessary to specify a class with the `main` method, which will be used as an entry point into the microservice.

② The resulting `uber.jar` must not contain the module virtual descriptor, as this descriptor is only needed at the compilation level. *(More information about the virtual descriptor can be found in Section 13.1.2, "Module Configuration".)*

③ The resulting `uber.jar` can be reduced by excluding unused classes and packages.

The `maven-shade-plugin` plugin automatically removes all `module-info.class` from the resulting `uber.jar`, so all RxMicro modules will work in the `unnamed module` mode!

#### 13.1.1.3. Rebuild Project

After `pom.xml` setting, it is necessary to rebuild the project:

```
mvn clean package
```

As a result, the `uber.jar` with all the necessary dependencies will be created:

```
:$ ls -lh

-rw-rw-r-- 1 nedis nedis 1,4M Mar 28 11:03 unnamed-module-uber-jar-1.0-SNAPSHOT.jar
①
-rw-rw-r-- 1 nedis nedis 9,8K Mar 28 11:03 original-unnamed-module-uber-jar-1.0-
SNAPSHOT.jar ②
```

① The `uber.jar` size is equal to **1,4 MB**.

② The size of the original jar file is equal to **9.8 KB**.

The `uber.jar` contains the source code of the microservice project, the code of the RxMicro and the Netty frameworks:

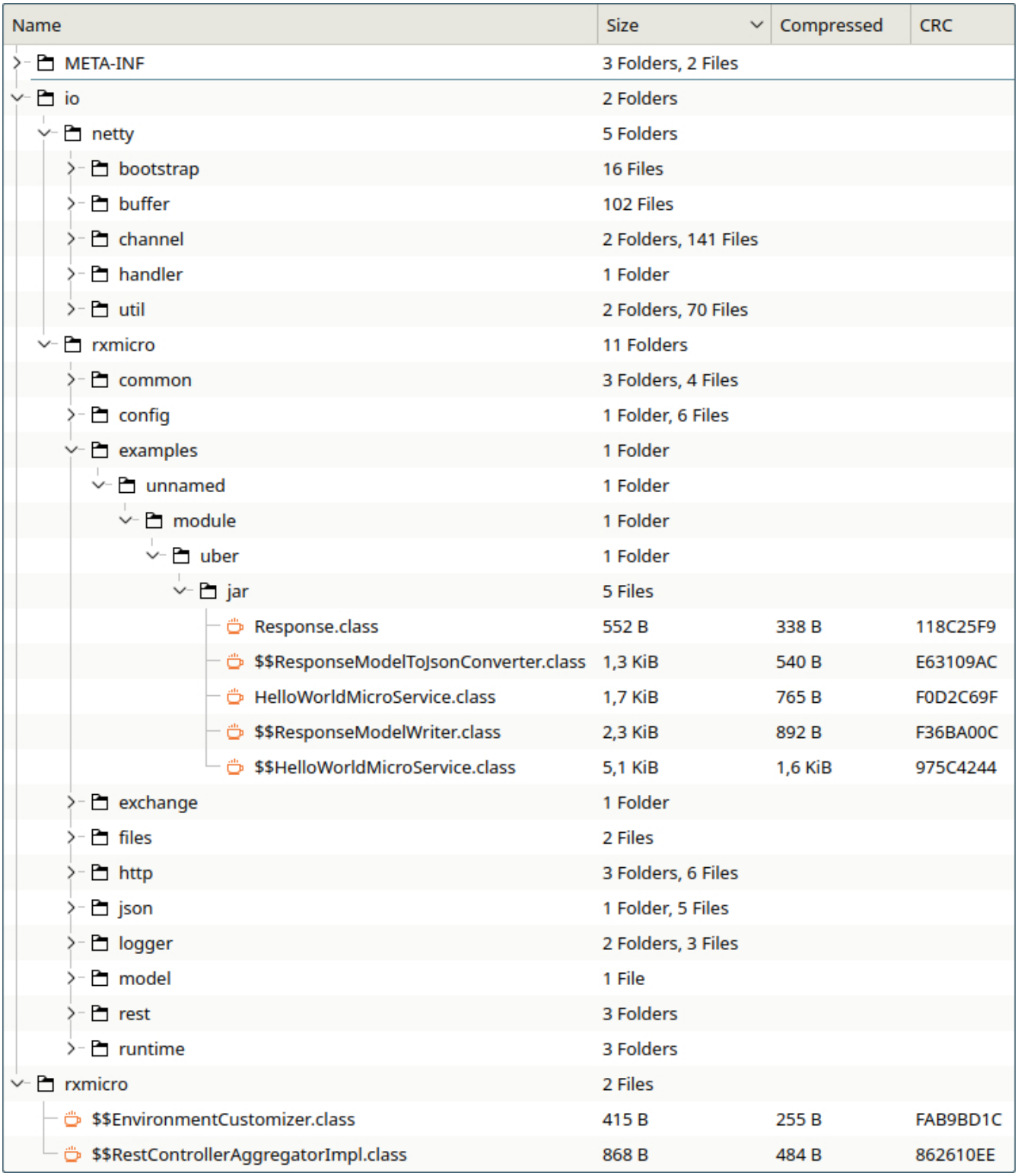



*Figure 16. The `uber.jar` content for a simple microservice project*

#### 13.1.1.4. Run `uber.jar`

To run `uber.jar` it is necessary to provider jar file only:

```
> java -jar unnamed-module-uber-jar-1.0-SNAPSHOT.jar

2020-02-02 20:14:11.707 [INFO]
io.rxmicro.rest.server.netty.internal.component.NettyServer :
Server started at 0.0.0.0:8080 using NETTY transport in 500 millis ①
```

① The `Server started in ⋯ millis` message format means that the RxMicro HTTP server has been successfully started.

To run a modularized micro service project the java requires more arguments:

```
java -p ./classes:lib -m
examples.quick.start/io.rxmicro.examples.quick.start.HelloWorldMicroSer
vice
```

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-unnamed-module/unnamed-module-uber-jar

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 13.1.2. Module Configuration

Some RxMicro modules allow You to set the common configuration with `RxMicro Annotations`, which annotate the `module-info.java` module descriptor. But for the `unnamed module`, the `module-info.java` module descriptor does not exist!

In such cases a virtual descriptor must be created:

```
@Retention(CLASS)
@Target({})
public @interface ModuleInfo {
}
```

A virtual descriptor is a custom annotation that meets the following requirements:

1. The name of the virtual descriptor is fixed and equal to: `ModuleInfo`.
2. The annotation must be contained in the default package (`unnamed package`).
3. The annotation should be available only during the compilation process: `@Retention(CLASS)`.
4. The annotation must contain an empty list of targets: `@Target({})`.

Since the virtual descriptor is available only during the compilation process, then during the `uber.jar` build process this annotation can be excluded from the resulting `jar`.

Therefore, when setting the `maven-shade-plugin`, the following setting was added:

```
<filters>
    <filter>
        <artifact>*:*</artifact>
        <excludes>
            <exclude>ModuleInfo.*</exclude>
        </excludes>
    </filter>
</filters>
```

#### 13.1.2.1. Configuration of the Code Generation Process for Rest Controller

The RxMicro framework provides the option to configure the code generation process for REST controllers. For this purpose, the `@RestServerGeneratorConfig` annotation should be used to annotate the `module-info.java` module descriptor.

For the `unnamed module` instead of the `module-info.java` module descriptor, use the `ModuleInfo` annotation:

```
@Retention(CLASS)
@Target({})
@RestServerGeneratorConfig(
        exchangeFormat = ServerExchangeFormatModule.JSON,
        generateRequestValidators = GenerateOption.DISABLED,
        generateResponseValidators = GenerateOption.AUTO_DETECT
)
public @interface ModuleInfo {
}
```

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-unnamed-module/unnamed-module-rest-controller-generator

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

#### 13.1.2.2. Configuration of the Code Generation Process for Rest Client

The RxMicro framework provides the option to configure the code generation process for REST clients. For this purpose, the `@RestClientGeneratorConfig` annotation should be used to annotate the `module-info.java` module descriptor.

For the `unnamed module` instead of the `module-info.java` module descriptor, use the `ModuleInfo` annotation:

```
@Retention(CLASS)
@Target({})
@RestClientGeneratorConfig(
        exchangeFormat = ClientExchangeFormatModule.JSON,
        generateRequestValidators = GenerateOption.AUTO_DETECT,
        requestValidationMode =
RestClientGeneratorConfig.RequestValidationMode.RETURN_ERROR_SIGNAL,
        generateResponseValidators = GenerateOption.DISABLED
)
public @interface ModuleInfo {
}
```

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-unnamed-module/unnamed-module-rest-client-generator`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

#### 13.1.2.3. REST-based Microservice Metadata Configuration

The RxMicro framework provides the option to configure the REST-based microservice documentation generation process, using the annotations. These annotations annotate the `module-info.java` module descriptor.

For the `unnamed module` instead of the `module-info.java` module descriptor, use the `ModuleInfo` annotation:

```
@Retention(CLASS)
@Target({})
@Title("Metadata Annotations")
@Description("*Project* _Description_")
@Author(
        name = "Richard Hendricks",
        email = "richard.hendricks@piedpiper.com"
)
@Author(
        name = "Bertram Gilfoyle",
        email = "bertram.gilfoyle@piedpiper.com"
)
@Author(
        name = "Dinesh Chugtai",
        email = "dinesh.chugtai@piedpiper.com"
)
@BaseEndpoint("https://api.rxmicro.io")
@License(
        name = "Apache License 2.0",
        url = "https://github.com/rxmicro/rxmicro/blob/master/LICENSE"
)
@DocumentationDefinition(
        introduction = @IntroductionDefinition(
                sectionOrder = {
                        IntroductionDefinition.Section.BASE_ENDPOINT,
                        IntroductionDefinition.Section.LICENSES
                }
        ),
        resource = @ResourceDefinition(
                withInternalErrorResponse = false
        ),
        withGeneratedDate = false
)
public @interface ModuleInfo {
}
```

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-unnamed-module/unnamed-module-documentation-asciidoctor-metadata-annotations

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

# 13.2. Using GraalVM to Build a Native Image

GraalVM is a universal virtual machine for running applications written in different languages.

GraalVM contains a Native Image Tool, that allows You to ahead-of-time compile Java code to a standalone executable, called a native image.

The GraalVM Native Image Tool can be used to build a native image of any RxMicro microservice project. Before build a native image it is necessary to setup a GraalVM developer environment.

## 13.2.1. Setup a GraalVM

GraalVM is distributed as Community Edition and Enterprise Edition.

**Current guide describes a use of the GraalVM Community Edition (GraalVM CE)!**

Setup of the GraalVM CE contains a few simple steps:

1. Make a GraalVM Home Directory.
2. Select a GraalVM CE Distribution for Your Platform.
3. Download the GraalVM CE Distribution.
4. Unzip the GraalVM CE Distribution.
5. Set a `GRAALVM_HOME` Environment Variable.
6. Install a Native Image Module.
7. Add the Native Image Tool to `PATH`.

### 13.2.1.1. Make a GraalVM Home Directory

Make a directory, that will contain a GraalVM distribution:

```
mkdir ~/GraalVM
```

### 13.2.1.2. Select a GraalVM CE Distribution for Your Platform

Visit the github release page `https://github.com/graalvm/graalvm-ce-builds/releases` and select a distribution for Your platform.

**Please select a GraalVM CE based on Java 11!**

*A GraalVM CE based on Java 8 is not supported by the RxMicro framework!*

### 13.2.1.3. Download the GraalVM CE Distribution

Download the selected GraalVM CE distribution onto the GraalVM home directory:

```
wget -P ~/GraalVM https://github.com/graalvm/graalvm-ce-builds/releases/download/vm-
20.0.0/graalvm-ce-java11-linux-amd64-20.0.0.tar.gz
```

#### 13.2.1.4. Unzip the GraalVM CE Distribution

Unzip the GraalVM CE distribution onto the GraalVM home directory:

```
tar --strip-components=1 -vzxf ~/GraalVM/graalvm-ce-java11-linux-amd64-20.0.0.tar.gz
-C ~/GraalVM
rm ~/GraalVM/graalvm-ce-java11-linux-amd64-20.0.0.tar.gz
```

#### 13.2.1.5. Set a `GRAALVM_HOME` Environment Variable

Set a `GRAALVM_HOME` environment variable:

```
export GRAALVM_HOME=~/GraalVM
```

and verify the installation:

```
$GRAALVM_HOME/bin/java -version

openjdk version "11.0.6" 2020-01-14
OpenJDK Runtime Environment GraalVM CE 20.0.0 (build 11.0.6+9-jvmci-20.0-b02)
OpenJDK 64-Bit Server VM GraalVM CE 20.0.0 (build 11.0.6+9-jvmci-20.0-b02, mixed mode,
sharing)
```

If You would use the GraalVM CE only, You can add the `$GRAALVM_HOME/bin` path to the `PATH` variable!

If You use any other version of JDK, add only the separate utils to `PATH` variable. For example only `gu` or `native-image`!

#### 13.2.1.6. Install a Native Image Module

Starting from GraalVM 19.0, Native Image was extracted from the base distribution. So the **Native Image Tool** must be installed to GraalVM using a GraalVM Updater utility:

```
$GRAALVM_HOME/bin/gu install native-image

Downloading: Component catalog from www.graalvm.org
Processing Component: Native Image
Downloading: Component native-image: Native Image  from github.com
Installing new component: Native Image (org.graalvm.native-image, version 20.0.0)
```

To verify the installation show a version of the native Native Image Tool:

```
$GRAALVM_HOME/bin/native-image --version

GraalVM Version 20.0.0 CE
```

#### 13.2.1.7. Add the Native Image Tool to `PATH`

In order not to indicate the folder where the **Native Image Tool** was installed, add the **Native Image Tool** to the `PATH` variable:

```
ln -s $GRAALVM_HOME/bin/native-image ~/bin
```

After that logout and login to the system again. The **Native Image Tool** must be available at the terminal:

```
native-image --version

GraalVM Version 20.0.0 CE
```

### 13.2.2. RxMicro Project Configuration

The first release of GraalVM based on Java 11 is **19.3.0** (2019-11-19). But unfortunately the GraalVM Native Image Tool does not support `JPMS` yet.

Thus we need to use the `unnamed module` mode for RxMicro project.

#### 13.2.2.1. Setup `pom.xml`

##### 13.2.2.1.1. Using `RX_MICRO_BUILD_UNNAMED_MODULE` Option

To enable the `unnamed module` mode, it is necessary to add `RX_MICRO_BUILD_UNNAMED_MODULE` option to the `maven-compiler-plugin`:

```
<plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
        <!-- ... -->
        <compilerArgs>
            <arg>-ARX_MICRO_BUILD_UNNAMED_MODULE=true</arg>
        </compilerArgs>
    </configuration>
    <!-- ... -->
</plugin>
```

#### 13.2.2.1.2. `Maven-shade-plugin` Configuration

To build a native image we need a `uber jar`. For that it is necessary to add `maven-shade-plugin`:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-shade-plugin</artifactId>
    <version>${maven-shade-plugin.version}</version>
    <executions>
        <execution>
            <phase>package</phase>
            <goals>
                <goal>shade</goal>
            </goals>
            <configuration>
                <createDependencyReducedPom>false</createDependencyReducedPom>
                <transformers>
                    <transformer implementation=

"org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                        <manifestEntries>
                            <Main-Class>

io.rxmicro.examples.graalvm.nativeimage.quick.start.HelloWorldMicroService
                            </Main-Class>
                        </manifestEntries>
                    </transformer>
                </transformers>
            </configuration>
        </execution>
    </executions>
</plugin>
```

#### 13.2.2.1.3. `Exec-maven-plugin` Configuration

To build a native image from `maven` it is necessary to use `exec-maven-plugin`:

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>exec-maven-plugin</artifactId>
    <version>${exec-maven-plugin.version}</version>
    <executions>
        <execution>
            <phase>package</phase>
            <goals>
                <goal>exec</goal>
            </goals>
            <configuration>
                <executable>native-image</executable> ①
                <arguments>
                    <argument>--verbose</argument>
                    <argument>-jar</argument>
                    <argument>--install-exit-handlers</argument> ②

<argument>${project.build.directory}/${project.build.finalName}.jar</argument>
                </arguments>
            </configuration>
        </execution>
    </executions>
</plugin>
```

① An executable will be the `native-image` tool with specified arguments.

② This option adds the exit handler for native image.

The Native Image Tool (`native-image`) must be available via `PATH` variable!

#### 13.2.2.2. Microservice Source Code

A microservice source code contains two classes only:

```
final class Response {

    final String message;

    Response(final String message) {
        this.message = requireNonNull(message);
    }
}
```

```
public final class HelloWorldMicroService {

    @GET("/")
    CompletableFuture<Response> sayHelloWorld() {
        return CompletableFuture.supplyAsync(() ->
                new Response("Hello World!"));
    }

    public static void main(final String[] args) {
        new Configs.Builder()
                .withContainerConfigSources() ①
                .build();
        RxMicroRestServer.startRestServer(HelloWorldMicroService.class);
    }
}
```

① Docker based configuration activates the configuration via annotations and environment variables only.

#### 13.2.2.3. Classpath Resources

A RxMicro logger is configured using `jul.properties` classpath resource.

`jul.properties` contains the following content:

```
io.rxmicro.rest.server.level=TRACE
```

A native image is configured using `META-INF/native-image` classpath directory with two files:

- `native-image.properties`
- `resource-config.json`

*(Read more about native image configuration: https://www.graalvm.org/docs/reference-manual/native-image/#native-image-configuration)*

`native-image.properties` contains the following content:

```
Args = --no-fallback \
       --allow-incomplete-classpath \
       -H:ResourceConfigurationResources=${.}/resource-config.json
```

`resource-config.json` contains the following content:

```
{
  "resources":[
    {"pattern":"\\Qjul.properties\\E"}
  ],
  "bundles":[]
}
```

### 13.2.3. Creation of the Native Image

To build a native image run:

```
mvn clean package
```

The built native image executable will be available at the project home directory:

```
ls -lh

drwxrwxr-x 5 nedis nedis 4,0K Mar 25 20:44 src
drwxrwxr-x 8 nedis nedis 4,0K Mar 28 21:30 target
-rw-rw-r-- 1 nedis nedis 8,8K Mar 28 21:02 pom.xml
-rwxrwxr-x 1 nedis nedis  17M Mar 28 21:32 HelloWorldMicroService ①
```

① The built native image executable

### 13.2.4. Verification of the Native Image

To verify the built native image run:

```
./HelloWorldMicroService
```

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-graalvm/graalvm-native-image-quick-start

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

# 14. Testing

To help You write tests efficiently, the RxMicro framework provides the following modules:

- The `rxmicro.test` is a basic module designed for test writing using any modern testing framework;
- The `rxmicro.test.junit` is a module designed for test writing using the JUnit 5 framework;
- The `rxmicro.test.mockito` is a module designed for test writing using the Mockito framework;
- The `rxmicro.test.mockito.junit` is a module designed for test writing using the JUnit 5 and Mockito frameworks.
- The `rxmicro.test.dbunit` is a module designed for test writing using the DbUnit framework;
- The `rxmicro.test.dbunit.junit` is a module designed for test writing using the DbUnit and JUnit 5 frameworks;

Using these modules, the developer can create the following types of tests:

1. REST-based microservice test;
2. Component unit test;
3. Integration test of REST-based microservices.
4. Database Testing Using DBUnit.

## 14.1. Preparatory Steps

Before writing tests, using the RxMicro framework, the following steps must be taken:

1. Define the versions of used libraries.
2. Add the required dependencies to the `pom.xml`.
3. Configure the `maven-compiler-plugin`.
4. Configure the `maven-surefire-plugin`.

### 14.1.1. Definition the Versions of the Used Libraries:

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <rxmicro.version>0.11</rxmicro.version> ①

    <maven-compiler-plugin.version>3.10.1</maven-compiler-plugin.version> ②
    <maven-surefire-plugin.version>3.0.0-M7</maven-surefire-plugin.version>③
</properties>
```

① The latest stable version of the RxMicro framework.

② The latest stable version of the `maven-compiler-plugin`.

③ The latest stable version of the `maven-surefire-plugin`.

### 14.1.2. Adding the Required Dependencies:

Before using the RxMicro modules for testing, You need to add the following dependencies to the project:

```
<dependencies>
    <dependency>
        <groupId>io.rxmicro</groupId>
        <artifactId>rxmicro-test-junit</artifactId> ①
        <version>${rxmicro.version}</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>io.rxmicro</groupId>
        <artifactId>rxmicro-test-mockito-junit</artifactId> ②
        <version>${rxmicro.version}</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>io.rxmicro</groupId>
        <artifactId>rxmicro-rest-client-exchange-json</artifactId> ③
        <version>${rxmicro.version}</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

① `rxmicro-test-junit` - a unit testing library based on the JUnit 5 framework.

② `rxmicro-test-mockito-junit` - a unit testing library based on the JUnit 5 framework with integration of the Mockito framework.

③ `rxmicro-rest-client-exchange-json` - a library for converting Java models to JSON format and vice versa on the HTTP client side.

The `rxmicro-rest-client-exchange-json` libraries are required only during writing the REST-based microservice tests and integration tests. These dependencies can be omitted during writing the component unit tests.

The `rxmicro-test-mockito-junit` library depends on the `rxmicro-test-junit`, so only one of them needs to be added:

1. If only the JUnit 5 framework is required, use the `rxmicro-test-junit` library.
2. If You still need to create mocks, then use the `rxmicro-test-mockito-junit` library.

### 14.1.3. Configuring the `maven-compiler-plugin`:

To solve problems with the Java module system when writing the tests, it is necessary to add the additional execution to the `maven-compiler-plugin` configuration:

```
<plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>${maven-compiler-plugin.version}</version> ①
    <configuration>
        <release>11</release>
        <annotationProcessorPaths>
            <annotationProcessorPath>
                <groupId>io.rxmicro</groupId>
                <artifactId>rxmicro-annotation-processor</artifactId>
                <version>${rxmicro.version}</version>
            </annotationProcessorPath>
        </annotationProcessorPaths>
    </configuration>
    <executions>
        <execution>
            <id>source-compile</id>
            <goals>
                <goal>compile</goal>
            </goals>
            <configuration>
                <annotationProcessors>
                    <annotationProcessor>
                        io.rxmicro.annotation.processor.RxMicroAnnotationProcessor
                    </annotationProcessor>
                </annotationProcessors>
                <generatedSourcesDirectory>
                    ${project.build.directory}/generated-sources/
                </generatedSourcesDirectory>
            </configuration>
        </execution>
        <execution>
            <id>test-compile</id> ②
            <goals>
                <goal>testCompile</goal>
            </goals>
            <configuration>
                <annotationProcessors>
                    <annotationProcessor>

io.rxmicro.annotation.processor.RxMicroTestsAnnotationProcessor ③
                    </annotationProcessor>
                </annotationProcessors>
                <generatedTestSourcesDirectory>
                    ${project.build.directory}/generated-test-sources/ ④
                </generatedTestSourcesDirectory>
            </configuration>
        </execution>
    </executions>
</plugin>
```

① The plugin version defined in the `properties` section.

② The separate configuration is required for the tests, so a new `execution` must be added.

③ The annotation processor class that handles the test configuration.

④ Location of Java classes generated by the `RxMicro Test Annotation Processor`.

The `io.rxmicro.annotation.processor.RxMicroTestsAnnotationProcessor` generates additional classes required **only during testing**.

**Therefore, You must always specify a separate folder for the generated classes!**

### 14.1.4. Configuring the `maven-surefire-plugin`:

For a successful tests launch while building a project with `maven` it is necessary to update `maven-surefire-plugin`:

```
<plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>${maven-surefire-plugin.version}</version> ①
    <configuration>
        <properties>
            <!--
https://junit.org/junit5/docs/5.5.1/api/org/junit/jupiter/api/Timeout.html -->
            <configurationParameters>
                junit.jupiter.execution.timeout.default = 60                ②
                junit.jupiter.execution.timeout.mode = disabled_on_debug    ③
                junit.jupiter.execution.parallel.enabled = false            ④
            </configurationParameters>
        </properties>
    </configuration>
</plugin>
```

① Last stable version of `maven-surefire-plugin`.
*(The plugin version must be `2.22.1` or higher, otherwise `maven` will ignore the tests!.)*

② In case of an error in the code which uses reactive programming, an infinite function execution may occur. In order to detect such cases, it is necessary to set a global timeout for all methods in the tests. *(By default, timeout is set in seconds. More detailed information on timeouts configuration is available in official JUnit5 documentation.)*

③ While debugging, timeouts can be turned off.

④ This property is useful for the tests debugging from IDE or `maven`.
*(By setting this property the speed of test performance will decrease, so use this property for debugging only!)*

At the end of each subsection describing any the RxMicro framework feature, there is a link to the project, the source codes of which were used in this subsection.

Use these links to access a working example demonstrating the RxMicro framework feature described above!

## 14.2. RxMicro Test Annotations

The RxMicro framework supports additional RxMicro Test annotations, that extend the features of the JUnit 5, Mockito and DbUnit test frameworks and add additional features required for efficient writing of all supported test types.

The RxMicro framework supports two types of annotations:

- The RxMicro Annotations used by the RxMicro Annotation Processor during the compiling and therefore available **only during the compilation process**.
- The RxMicro Test Annotations used by the test framework during the test run and therefore available using the reflection mechanism.

These types of annotations do not complement each other! Each of these types is designed to perform its tasks.

**So when writing tests, be careful not to use the RxMicro Annotations as they are not available for the test framework!**

*Table 20. Supported RxMicro Test Annotations*

| Annotation | Description |
|---|---|
| @Alternative | Declares the test class field as an alternative.<br><br>*The RxMicro framework supports alternatives only for REST-based microservice tests and component unit tests.* |
| @WithConfig | Declares the **static** field of the test class as a configuration which must be registered in the configuration manager before starting the test.<br><br>*This annotation allows declaring a configuration using Java classes. (The configuration defined in this way is only available while the test is running.)*<br><br>The source code of the project that uses the @WithConfig annotation, is available at the following link:<br><br>testing-microservice-with-config<br><br>*The RxMicro framework supports test configuration only for REST-based microservice tests and component unit tests.* |

| Annotation | Description |
| --- | --- |
| `@SetConfigValue` | Allows overriding the default value for any configuration available only for the test environment.<br><br>*This annotation allows declaring a configuration using annotations. (The configuration defined in this way is only available while the test is running. It means that this annotation is analogous to the* `DefaultConfigValue` *annotation!)*<br><br>*The RxMicro framework supports test configuration only for REST-based microservice tests and component unit tests.* |
| `@BlockingHttpClientSettings` | Allows to configure the following component: `BlockingHttpClient`, in order to execute HTTP requests in tests.<br><br>*(This annotation applies only to the* `BlockingHttpClient` *type fields.)*<br><br>*The RxMicro framework supports the* `BlockingHttpClient` *component only for REST-based microservice tests and REST-based microservice integration tests.* |
| `@RxMicroRestBasedMicroServiceTest` | Declares the test class as a REST-based microservice test. |
| `@RxMicroComponentTest` | Declares the test class as a component unit test. |
| `@RxMicroIntegrationTest` | Declares the test class as a REST-based microservice integration test. |
| `@DbUnitTest` | Declares the test class as a DBUnit integration test. |
| `@BeforeThisTest` | It is used to specify the method to be invoked by the RxMicro framework before running the test method.<br><br>*The RxMicro framework supports the* `@BeforeThisTest` *annotation only for REST-based microservice tests and component unit tests.* |
| `@BeforeIterationMethodSource` | It is used to specify the methods to be invoked by the RxMicro framework before performing each iteration of the parameterized test.<br><br>*The RxMicro framework supports the* `@BeforeIterationMethodSource` *annotation only for REST-based microservice tests and component unit tests.* |
| `@InitMocks` | Informs the test framework about the need to create mocks and inject them into the test class fields, annotated by the `@Mock` annotation.<br><br>*(Using the* `@InitMocks` *annotation is preferable to the analogous* `@ExtendWith(MockitoExtension.class)` *construction.)* |
| `@InitialDataSet` | Provides the init state of tested database before execution of the test method. |

| Annotation | Description |
| --- | --- |
| `@ExpectedDataSet` | Provides the expected state of tested database after execution of the test method.<br><br>If expected state does not match to the actual database state the `java.lang.AssertionError` will be thrown. |
| `@RollbackChanges` | Declares the transactional test.<br><br>The transaction test means that all changes made by test will rolled back after the test execution. |

# 14.3. Alternatives

For efficient unit testing, the RxMicro framework supports the mechanism of alternatives.

**Alternatives** are test components, usually being mocks with predefined behaviors, that are injected by the RxMicro framework into the tested classes. Alternatives are a powerful mechanism for writing unit tests.

The RxMicro framework supports alternatives only for REST-based microservice tests and component unit tests.

When developing a microservice project, two types of components are distinguished:

- *RxMicro component* - a class that is part of the RxMicro framework (for example, `HttpClientFactory`) or a class generated by the `RxMicro Annotation Processor` (Data Repository, Rest client, etc).
- *Custom component* - a developer-written class that is part of a microservice project.

These two types of components have different life cycles:

- The instances of the RxMicro components are created in the classes generated by the `RxMicro Annotation Processor`, and are registered in the runtime container. When a reference to the RxMicro component is required, the custom class requests it in the runtime container.
- The instances of custom components are created independently by the developer in the code.

Due to the difference in life cycles between the two types of the RxMicro components, the RxMicro framework also supports two types of alternatives:

- **alternatives of the RxMicro components**;
- **alternatives of custom components**.

These types of alternatives differ in the algorithms of injection into the tested class.

### 14.3.1. Injection Algorithm for the Alternative of the RxMicro Component

To inject the alternative of the RxMicro component, the RxMicro framework uses the following algorithm:

1. The alternative instance is created by the developer in the test code or by the testing framework automatically.
2. Once all alternatives have been created, they are registered in the runtime container.
3. Once all alternatives have been registered, the RxMicro framework creates an instance of the tested class.
4. In the constructor or static section of the tested class, a request to the runtime container to get a reference to the RxMicro component is executed.
5. Since the runtime container already contains an alternative instead of the real component, the alternative is injected into the instance of the tested class.
6. After initialization, the instance of the tested class contains references to alternatives instead of the real RxMicro components.

### 14.3.2. Injection Algorithm for the Alternative of the Custom Component

To inject the alternative of the custom component, the RxMicro framework uses the following algorithm:

1. The alternative instance is created by the developer in the test code or by the testing framework automatically.
2. The RxMicro framework creates an instance of the tested class.
3. In the constructor or static section of the tested class, instances of the real custom components are created.
4. After initialization, the instance of the tested class contains references to the real custom components.
5. After creating an instance of the tested class, the RxMicro framework injects the custom component alternatives using the `reflection` mechanism.
   *(I.e. the alternatives replace the real instances already after creating an instance of the tested class.)*;
6. After alternative injection, the instance of the tested class contains references to the alternatives of the RxMicro components instead of the real RxMicro components.
   *(The real component instances will be removed by the garbage collector later.)*;

Thus, the main difference of the injection algorithm for the custom component alternatives is that during the injection process, the real component instances are always created.

If the real component creates a reference to an external resource, then this resource **will not be** released automatically when injecting the alternative!

It is recommended to use the `rxmicro.cdi` module to create the custom component alternatives that work with external resources.

If the `rxmicro.cdi` module is used by the developer to create the custom component instances, then **all** custom components are injected using the algorithm of the RxMicro component alternative injection.

### 14.3.3. Alternative Usage

The alternative mechanism is a universal tool that can be applied to the simplest project, which doesn't use the `RxMicro Annotations`.
Let's look at the project consisting of two components: the `ChildComponent` interface and the `ParentComponent` class:

```
public interface ChildComponent {

    String getValue();
}
```

```
public final class ParentComponent {

    private final ChildComponent childComponent = () -> "production"; ①

    public String getEnvironment() {
        return childComponent.getValue();
    }
}
```

① The `ParentComponent` class depends on the `ChildComponent`.
*(This dependency specified explicitly in the source code.)*

When writing a unit test for the `ParentComponent`, it is necessary to replace the real `ChildComponent` with a mock. Otherwise, it wouldn't be a unit test, but an integration one.

For this replacement, it is most convenient to use the alternative:

```
①
@RxMicroComponentTest(ParentComponent.class)
final class ParentComponent1Test {

    ③
    @Alternative
    private final ChildComponent childComponent = () -> "test"; ④

    private ParentComponent parentComponent; ②

    @Test
    void Should_use_alternative() {
        assertEquals("test", parentComponent.getEnvironment()); ⑤
    }
}
```

① The alternatives are supported by the RxMicro framework only during component unit tests or REST-based microservice tests. Therefore, it must be declared that this test is a test of the `ParentComponent` component.

② The instance of the tested component will be created by the RxMicro framework automatically.
*(The instance is created using the `reflection` mechanism, so the tested class must contain an available constructor without parameters.)*
In order to invoke any method of the tested component, a reference to that component is required. Therefore, the RxMicro framework requires that the developer declares an uninitialized field of the tested component. After starting the test, a reference to the instance of the tested component will be injected into this field using the `reflection` mechanism.

③ In order to use the alternative mechanism, it is necessary to declare test field as alternative by using the `@Alternative` annotation.

④ Alternative is a test component with predefined behavior. Therefore, it is necessary to define what value should be returned when invoking the `getValue` method.

⑤ When testing the `getEnvironment` method, the alternative method is invoked instead of the real component one.

If You would like to inject the alternative to the **final** field, don't forget to configure `maven-surefire-plugin`:

```
<plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <configuration>
        <argLine>
            --add-opens java.base/java.lang.reflect=rxmicro.reflection
①
        </argLine>
    </configuration>
</plugin>
```

① - It is necessary to add this `opens` instruction.

When using alternatives it is very convenient to use a dynamic class with programmable behavior. For this purpose, it is very convenient to use the Mockito framework:

```
②
@InitMocks
@RxMicroComponentTest(ParentComponent.class)
final class ParentComponent2Test {

    private ParentComponent parentComponent;

    ①
    @Mock
    @Alternative
    private ChildComponent childComponent;

    @Test
    void Should_use_alternative() {
        when(childComponent.getValue()).thenReturn("test"); ③
        assertEquals("test", parentComponent.getEnvironment()); ④
    }
}
```

① To create a mock instance, it is necessary to use the `@Mock` annotation.

② In order for JUnit to handle all fields annotated by the `@Mock` annotation before invoking test methods, it is necessary to annotate the test class by the `@InitMocks` annotation.

③ Before testing, it is necessary to program the behavior of the `getValue` method of the declared mock.

④ When testing the `getEnvironment` method, the method from the alternative is invoked instead of the real component one.

If You would like to inject the alternative to the **final** field, don't forget to configure `maven-surefire-plugin`:

```
<plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <configuration>
        <argLine>
            --add-opens java.base/java.lang.reflect=rxmicro.reflection
①
        </argLine>
    </configuration>
</plugin>
```

① - It is necessary to add this `opens` instruction.

When creating mock alternatives, the `@InitMocks` annotation should be over the `@RxMicroComponentTest` annotation (or `@RxMicroRestBasedMicroServiceTest` when writing REST-based microservice tests), otherwise, **the alternative will be injected before creating a mock instance, (i.e. injection of the `null` instance), which will cause an error!**

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-testing-junit/testing-component-alternatives-simplest`

### 14.3.4. Components with Custom Constructors

In case the custom component does not contain an available constructor without parameters:

```
public final class ParentComponent {

    private final String prefix;

    private final ChildComponent childComponent = () -> "production"; ①

    public ParentComponent(final String prefix) { ②
        this.prefix = requireNonNull(prefix);
    }

    public String getEnvironment() {
        return prefix + " " + childComponent.getValue();
    }
}
```

① The `ParentComponent` depends on the `ChildComponent`.
*(This dependency is specified explicitly in the source code.)*

② When creating the `ParentComponent` class instance in the constructor, the value of the `prefix` parameter must be passed.

then the RxMicro framework won't be able to create an instance of this class.

**Therefore, the developer should create an instance of this class in one of the following methods: `@BeforeEach`, `@BeforeThisTest` or `@BeforeIterationMethodSource`:**

```
@InitMocks
@RxMicroComponentTest(ParentComponent.class)
final class ParentComponentTest {

    private ParentComponent parentComponent;

    ②
    @Mock
    @Alternative
    private ChildComponent childComponent;

    @BeforeEach
    void beforeEach() {
        parentComponent = new ParentComponent("prefix"); ①
    }

    @Test
    void Should_use_alternative() {
        when(childComponent.getValue()).thenReturn("test");
        assertEquals("prefix test", parentComponent.getEnvironment()); ③
    }
}
```

① The `ParentComponent` class instance is created inside the method annotated by the `@BeforeEach` annotation.

② The `childComponent` alternative will be injected into the `ParentComponent` class instance after invoking the `beforeEach()` method and before the `Should_use_alternative()` test method.

③ When testing the `getEnvironment` method, the alternative method is invoked instead of the real component one.

If You would like to inject the alternative to the **final** field, don't forget to configure `maven-surefire-plugin`:

```
<plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <configuration>
        <argLine>
            --add-opens java.base/java.lang.reflect=rxmicro.reflection
①
        </argLine>
    </configuration>
</plugin>
```

① - It is necessary to add this `opens` instruction.

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-testing-junit/testing-component-alternatives-simplest`

### 14.3.5. Ambiguity Resolving

The alternative created by the developer can be injected by the RxMicro framework not only in the tested component, but also in any of its child components.

During such injection, the ambiguity problem may occur.

#### 14.3.5.1. Ambiguity Resolving Demonstration

Let's assume there is some business service in the project:

```
public interface BusinessService {

    String getValue();
}
```

This business service is a dependency for three interdependent components: `Child`, `Parent`, `GrandParent`:

```
public final class Child {

    private final BusinessService childBusinessService = () -> "Child";

    public String getValue() {
        return childBusinessService.getValue();
    }
}
```

```
public final class Parent {

    private final Child child = new Child();

    private final BusinessService parentBusinessService = () -> "Parent";

    public String getValue() {
        return parentBusinessService.getValue() + " : " + child.getValue();
    }
}
```

```
public final class GrandParent {

    private final Parent parent = new Parent();

    private final BusinessService grandParentBusinessService = () -> "GrandParent";

    public String getValue() {
        return grandParentBusinessService.getValue() + " : " + parent.getValue();
    }
}
```

When invoking the `GrandParent.getValue` method, this method is invoked on the business services of all `Child`, `Parent` and `GrandParent` dependent components according to the dependency hierarchy:

```
final class GrandParent1Test {

    private final GrandParent grandParent = new GrandParent();

    @Test
    void Should_return_default_values() {
        assertEquals("GrandParent : Parent : Child", grandParent.getValue());
    }
}
```

When using an alternative, the behavior of the `GrandParent.getValue` method is changed:

```
@InitMocks
@RxMicroComponentTest(GrandParent.class)
final class GrandParent2Test {

    private GrandParent grandParent;

    @Mock
    ①
    @Alternative
    private BusinessService businessService;

    @Test
    void Should_inject_alternatives_correctly() {
        when(businessService.getValue()).thenReturn("Mock"); ②

        assertEquals("Mock : Mock : Mock", grandParent.getValue()); ③
    }
}
```

① An alternative to the business service is created.

② Before testing, the behavior of the `getValue` method of created mock is programmed.

③ As a result of the test, You can see that this alternative is injected into **all** `Child`, `Parent` and `GrandParent` dependent components.

If You would like to inject the alternative to the **final** field, don't forget to configure `maven-surefire-plugin`:

```
<plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <configuration>
        <argLine>
            --add-opens java.base/java.lang.reflect=rxmicro.reflection
①
        </argLine>
    </configuration>
</plugin>
```

① - It is necessary to add this `opens` instruction.

If You create 2 or more (no more than 3 in this test example) alternatives, then each alternative can be injected in a separate business component:

```
@InitMocks
@RxMicroComponentTest(GrandParent.class)
final class GrandParent3Test {

    private GrandParent grandParent;

    @Mock
    @Alternative
    private BusinessService grandParentBusinessService;

    @Mock
    @Alternative(name = "childBusinessService")
    private BusinessService businessService;

    @Test
    void Should_inject_alternatives_correctly() {
        when(grandParentBusinessService.getValue()).thenReturn("GrandParentMock");
        when(businessService.getValue()).thenReturn("ChildMock");

        assertEquals("GrandParentMock : Parent : ChildMock", grandParent.getValue());
①
    }
}
```

① The `grandParentBusinessService` alternative is injected into the `GrandParent` component, and the `businessService` alternative is injected into the `Child` component;

If You would like to inject the alternative to the **final** field, don't forget to configure `maven-surefire-plugin`:

```
<plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <configuration>
        <argLine>
            --add-opens java.base/java.lang.reflect=rxmicro.reflection
①
        </argLine>
    </configuration>
</plugin>
```

① - It is necessary to add this `opens` instruction.

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-testing-junit/testing-component-alternatives-ambiguty-resolving

#### 14.3.5.2. Ambiguity Resolving Algorithm

To resolve ambiguities, the RxMicro framework uses the following algorithm:

1. For each tested component, a search for injection candidates is performed.
2. As a result, a map is formed with a user type as its key and a list of candidates for injection as its value. **(The RxMicro framework does not support polymorphism rules when injecting alternatives. Thus, the alternative of the `A` type can only be injected in the field with the `A` type)**.
3. After receiving a map with candidates for injection, the RxMicro framework passes through this map.
4. For each user type, a list of candidates and a list of alternatives is requested.
5. If there is only one alternative and only one candidate for the user type, the RxMicro framework injects this alternative into the candidate field;
6. If more than one alternative and only one candidate is found, the RxMicro framework will throw out an error;
7. If there is more than one candidate and only one alternative, then:
    a. The RxMicro framework analyzes the injection candidate field name:
        i. if the candidate field name matches the alternative field name, the RxMicro framework injects this alternative;
        ii. if the candidate field name matches the value of the `name` parameter of the `@Alternative` annotation, the RxMicro framework injects this alternative;
        iii. otherwise this candidate will be skipped;

b. if no alternative has been injected, the RxMicro framework injects this alternative in **all** candidate fields.
*(This is the behavior that occurs in the* `GrandParent2Test` *test.)*

8. If there is more than one candidate and more than one alternative, then:
   a. The RxMicro framework analyzes the injection candidate field name:
      i. if the candidate field name matches the alternative field name, the RxMicro framework injects this alternative;
      *(In the* `GrandParent3Test` *test, the* `grandParentBusinessService` *alternative is injected in the* `GrandParent` *component field, because the names of the alternative and component fields are equal.)*;
      ii. if the candidate field name matches the value of the `name` parameter of the `@Alternative` annotation, the RxMicro framework injects this alternative;
      *(In the* `GrandParent3Test` *test, the* `businessService` *alternative is injected in the* `Child` *component field, because the* `name` *parameter of the* `@Alternative` *annotation is equal to the* `childBusinessService`*. And in the* `Child` *class, the field name with the* `BusinessService` *type is also equal to the* `childBusinessService`*.)*
      iii. otherwise this candidate will be skipped;
   b. *(When more than one candidate and more than one alternative is found, it is possible that none of the alternatives will be injected.)*

### 14.3.6. CDI Beans Alternatives

If the developer uses the `rxmicro.cdi` module in the project, then **all** custom components are considered as beans and follow the injection algorithm for the alternatives of the RxMicro components.

*When using the `rxmicro.cdi` module,* ***You must always inject dependencies using the CDI mechanism only****:*

```
public final class ParentComponent {

    @Inject
    ChildComponent childComponent; ①

    @Inject
    ChildComponentImpl childComponentImpl; ①

    public String getEnvironment() {
        return childComponent.getValue() + " " + childComponentImpl.getValue();
    }
}
```

① The `ParentComponent` class depends on the `ChildComponent` and `ChildComponentImpl` components. *(These dependencies are injected using the CDI mechanisms.)*

The above example is a demonstration of the features of alternatives of custom components if `rxmicro.cdi` module enabled. That's why the `ChildComponentImpl` implementation is injected in the `ParentComponent` component. In real projects, it's recommended to inject only interfaces to ensure greater flexibility.

*Please note that since the `ChildComponentImpl` class implements the `ChildComponent` interface, and all CDI beans are singletons, the `childComponent` and `childComponentImpl` fields will contain references to the same `ChildComponentImpl` instance!*

```
public final class ChildComponentImpl implements ChildComponent {

    public ChildComponentImpl() {
        System.out.println("ChildComponentImpl created"); ①
    }

    @Override
    public String getValue() {
        return "production";
    }
}
```

① When creating the `ChildComponentImpl` instance, an information message is displayed in the

console.
*(This message is required to ensure that no real custom instance is created when CDI bean alternatives are used!)*

When testing, if no alternatives are created, the tested component uses the real custom component instances:

```
@InitMocks
@RxMicroComponentTest(ParentComponent.class)
final class ParentComponent1Test {

    private ParentComponent parentComponent;

    private SystemOut systemOut;

    @Test
    void Should_use_alternatives() {
        assertEquals("production production", parentComponent.getEnvironment());
①
        assertEquals("ChildComponentImpl created", systemOut.asString()); ②
    }
}
```

① When invoking the `getEnvironment` method, the real instances of custom components are used.

② When starting the test, only one `ChildComponentImpl` class instance is created.

When testing, if alternatives are created, the tested component uses them instead of the real custom component instances:

```
@InitMocks
@RxMicroComponentTest(ParentComponent.class)
final class ParentComponent2Test {

    private ParentComponent parentComponent;

    @Mock
    @Alternative
    private ChildComponent childComponent;

    @Mock
    @Alternative
    private ChildComponentImpl childComponentImpl;

    private SystemOut systemOut;

    @Test
    void Should_use_alternatives() {
        when(childComponent.getValue()).thenReturn("mock");
        when(childComponentImpl.getValue()).thenReturn("mock");

        assertEquals("mock mock", parentComponent.getEnvironment());   ①
        assertTrue(
                systemOut.isEmpty(),   ②
                format("Output not empty: '?'", systemOut.asString())
        );
    }
}
```

① When invoking the `getEnvironment` method, the alternatives of the real custom component instances are used.

② When starting the test, the `ChildComponentImpl` class instance is not created.

When using alternatives for complex components, it is possible to use alternatives together with real components:

```
@InitMocks
@RxMicroComponentTest(ParentComponent.class)
final class ParentComponent3Test {

    private ParentComponent parentComponent;

    @Mock
    @Alternative
    private ChildComponent childComponent;

    private SystemOut systemOut;

    @Test
    void Should_use_alternatives() {
        when(childComponent.getValue()).thenReturn("mock");

        assertEquals("mock production", parentComponent.getEnvironment());
①

        assertEquals("ChildComponentImpl created", systemOut.asString()); ②
    }
}
```

① When invoking the `getEnvironment` method, an alternative and a real instance are used.

② When starting the test, only one `ChildComponentImpl` class instance is created.

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-testing-junit/testing-component-alternatives-cdi

## 14.4. How It Works

Java 9 has introduced the `JPMS`.

This system requires that a developer defines the `module-info.java` descriptor for each project. In this descriptor, the developer must describe all the dependencies of the current project. In the context of the unit module system, the tests required for each project should be configured as a separate module, since they depend on libraries that should not be available in the `runtime`. Usually such libraries are unit testing libraries (e.g. JUnit 5), mock creation libraries (e.g. Mockito), etc.

When trying to create a separate `module-info.java` descriptor available only for unit tests, many modern IDEs report an error.

**Therefore, the simplest and most common solution to this problem is to organize unit tests in the form of automatic module.**
**This solution allows You to correct compilation errors, but when starting tests, there will be runtime errors.**
**To fix runtime errors, when starting the Java virtual machine, You must add options that configure the Java module system at runtime.**

In case the tests are run, these options must be added to the `maven-surefire-plugin`:

```
<plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.22.1</version>
    <configuration>
        <argLine>
            @{argLine}
            --add-exports ...
            --add-opens ...
            --patch-module ...
            --add-modules ...
            --add-reads ...
        </argLine>
    </configuration>
</plugin>
```

The specified configuration options for the Java module system at runtime can also be added using the features of the `java.lang.Module` class.

**In order the developer is relieved of the need to add the necessary options to the `maven-surefire-plugin` configuration, the RxMicro framework provides a special `io.rxmicro.annotation.processor.RxMicroTestsAnnotationProcessor` component.**

To activate this component, it is necessary to add a new execution to the `maven-compiler-plugin` configuration:

```
<execution>
    <id>test-compile</id>
    <goals>
        <goal>testCompile</goal> ①
    </goals>
    <configuration>
        <annotationProcessors>
            <annotationProcessor>
                io.rxmicro.annotation.processor.RxMicroTestsAnnotationProcessor ②
            </annotationProcessor>
        </annotationProcessors>
        <generatedTestSourcesDirectory>
            ${project.build.directory}/generated-test-sources/  ③
        </generatedTestSourcesDirectory>
    </configuration>
</execution>
```

① The separate configuration is required for the tests, so a new `execution` must be added.

② The annotation processor class that handles the test configuration.

③ Location of Java classes generated by the `RxMicro Test Annotation Processor`.

This annotation processor generates one single `rxmicro.$$ComponentTestFixer` class, that automatically opens access to all packages of the current project to unnamed modules:

```
public final class $$ComponentTestFixer {

    static {
        final Module currentModule = $$ComponentTestFixer.class.getModule();
        currentModule.addExports("rxmicro", RX_MICRO_REFLECTION_MODULE);
    }

    public $$ComponentTestFixer() {
        final Module currentModule = getClass().getModule();
        if (currentModule.isNamed()) {
            logInfoTestMessage("Fix the environment for componnet test(s)...");
            final Module unnamedModule =
getClass().getClassLoader().getUnnamedModule(); ①
            final Set<Module> modules = unmodifiableOrderedSet(
                    unnamedModule, RX_MICRO_REFLECTION_MODULE
            );
            for (final Module module : modules) {
                for (final String packageName : currentModule.getPackages()) {
                    currentModule.addOpens(packageName, module); ②
                    logInfoTestMessage(
                            "opens ?/? to ?", ③
                            currentModule.getName(),
                            packageName,
                            module.isNamed() ? module.getName() : "ALL-UNNAMED"
                    );
                }
            }
        }
    }
}
```

① Using the standard Java API, the RxMicro framework retrieves the references to the current and unnamed modules.

② Using the features of the `java.lang.Module` class, the RxMicro framework opens the full access to all classes from all packages from the current module.

③ To inform the developer about the successful performance of the `rxmicro.$$ComponentTestFixer` class, the RxMicro framework displays to the console the information that access was successfully provided.

When running different types of tests, sometimes a different configuration of the Java module system is required. Thus, for each type of test, the RxMicro framework creates a separate class in the `rxmicro` system package:

*Table 21. Names of generated classes*

| Test type | Name of the generated class |
|---|---|
| REST-based microservice test | `$$RestBasedMicroServiceTestFixer` |
| Component unit test | `$$ComponentTestFixer` |
| REST-based microservice integration test | `$$IntegrationTestFixer` |

Before starting tests, the RxMicro framework uses a generated class to configure the module system for the test environment:

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
...
[INFO] Fix the environment for component test(s)...
[INFO] opens examples.testing/io.rxmicro.examples.testing to ALL-UNNAMED ①
[INFO] opens examples.testing/rxmicro to ALL-UNNAMED
[INFO] Running io.rxmicro.examples.testing.ParentComponent1Test ②
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.278 s
[INFO] Running io.rxmicro.examples.testing.ParentComponent2Test
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.378 s
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

① All packages of the current module are opened before starting tests.

② After configuring the module system for the test environment, Unit tests are started.

**Thus, for the successful writing of tests using the RxMicro framework, besides adding the required libraries, do not forget to configure the `maven-compiler-plugin` by adding the following annotation processor for the test environment: `io.rxmicro.annotation.processor.RxMicroTestsAnnotationProcessor`.**

## 14.5. The `@BeforeThisTest` and `@BeforeIterationMethodSource` Annotations

In order to perform any actions before invoking the test method, the JUnit 5 framework provides a special `@BeforeEach` annotation.

According to the working rules of the JUnit 5 framework, if the test class contains a method annotated by the `@BeforeEach` annotation, then this method should be invoked before invoking each test method in the test class.

Let's consider the possibilities of using the `@BeforeEach` annotation. We'll use the example of a project consisting of two components: ChildComponent interface and ParentComponent class.

```
public interface ChildComponent {

    String getValue();
}
```

```
public final class ParentComponent {

    private final ChildComponent childComponent = () -> "production";

    public String getEnvironment() {
        return childComponent.getValue();
    }
}
```

When using the mock alternative, it is possible to program the behavior in the method annotated by the `@BeforeEach` annotation:

```
@InitMocks
@RxMicroComponentTest(ParentComponent.class)
final class ParentComponent1Test {

    private ParentComponent parentComponent;

    @Mock
    @Alternative
    private ChildComponent childComponent;

    ①
    @BeforeEach
    void beforeEach() {
        when(childComponent.getValue()).thenReturn("mock");
    }

    @Test
    void Should_use_alternative() {
        assertEquals("mock", parentComponent.getEnvironment());
    }
}
```

① Since the `beforeEach()` method will be surely invoked by the JUnit 5 framework before invoking the `Should_use_alternative()` test method, the programmed mock behavior will be used during the test;

The weak point in using the method annotated by the `@BeforeEach` annotation, is that it is impossible to program different behavior for two or more test methods in the same class:

```
@BeforeEach
void beforeEach() {
    when(childComponent.getValue()).thenReturn("mock");
}

@Test
void Should_use_alternative1() {
    assertEquals("mock1", parentComponent.getEnvironment()); // failed
}

@Test
void Should_use_alternative2() {
    assertEquals("mock2", parentComponent.getEnvironment()); // failed
}
```

The standard solution to this problem consists in programming behavior in the test method:

```
@Test
void Should_use_alternative1() {
    when(childComponent.getValue()).thenReturn("mock1");
    assertEquals("mock1", parentComponent.getEnvironment());
}

@Test
void Should_use_alternative2() {
    when(childComponent.getValue()).thenReturn("mock2");
    assertEquals("mock2", parentComponent.getEnvironment());
}
```

However, the alternative configuration in the test method is not always convenient.
**This can cause a problem, especially when it is necessary to configure the RxMicro component alternative before its registration in the runtime container!**

To solve this problem, the RxMicro framework provides two additional annotations:

- `@BeforeThisTest`;
- `@BeforeIterationMethodSource`;

The `@BeforeThisTest` annotation allows You to specify the method to be invoked before invoking a specific test method:

```
@InitMocks
@RxMicroComponentTest(ParentComponent.class)
final class ParentComponent4Test {

    private ParentComponent parentComponent;

    @Mock
    @Alternative
    private ChildComponent childComponent;

    void beforeTest1() { ①
        when(childComponent.getValue()).thenReturn("mock1");
    }

    @Test
    ②
    @BeforeThisTest(method = "beforeTest1")
    void Should_use_alternative1() {
        assertEquals("mock1", parentComponent.getEnvironment());
    }

    void beforeTest2() {
        when(childComponent.getValue()).thenReturn("mock2");
    }

    @Test
    @BeforeThisTest(method = "beforeTest2")
    void Should_use_alternative2() {
        assertEquals("mock2", parentComponent.getEnvironment());
    }
}
```

① To configure the mock alternative, it is necessary to create a method in the test class.

② If this method needs to be invoked before the test method, it is necessary to specify the name of this method using the `@BeforeThisTest` annotation.
*(The `@BeforeThisTest` annotation must annotate the test method!).*

When creating parametrized tests, it is necessary to use the `@BeforeIterationMethodSource` annotation instead of the `@BeforeThisTest` annotation:

```
@InitMocks
@RxMicroComponentTest(ParentComponent.class)
final class ParentComponent5Test {

    private ParentComponent parentComponent;

    @Mock
    @Alternative
    private ChildComponent childComponent;

    void beforeEachPreparer1() {
        when(childComponent.getValue()).thenReturn(new BigDecimal("23").toString());
    }

    void beforeEachPreparer2() {
        when(childComponent.getValue()).thenReturn("23");
    }

    @ParameterizedTest
    ①
    @BeforeIterationMethodSource(methods = {
            "beforeEachPreparer1",
            "beforeEachPreparer2"
    })
    void Should_use_alternative2(final String method) { ②
        assertEquals("23", parentComponent.getEnvironment());
    }
}
```

① Using the `@BeforeIterationMethodSource` annotation, the developer specifies an array of methods. Each method in this array must be invoked before executing a new iteration of the parameterized test.

② For reporting it is recommended to specify the `final String method` parameter of the parameterized test. This parameter is not used in the test, but if it is present, the JUnit 5 framework automatically passes the method name to it. Thus, in the execution report of the parameterized test, for each iteration the method that was invoked before execution of this iteration will be specified.

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-testing-junit/testing-component-before-test

# 14.6. REST-based Microservice Testing

The REST-based microservice test is a standard Unit test that tests only the source code of a microservice. If the current microservice depends on external services (e.g. database, other REST-based microservices, etc.), then it is allowed to use mocks for these external services during its testing. If several REST-based microservices need to be tested, it is recommended to use the REST-based microservice integration testing.

For easy writing of microservice tests, the RxMicro framework provides:

1. The additional `@RxMicroRestBasedMicroServiceTest` annotation, that informs the RxMicro framework about the need to start the tested REST-based microservice and prepare the environment to execute test HTTP requests.
2. A special **blocking** HTTP client to execute HTTP requests during testing: `BlockingHttpClient`.
3. The `SystemOut` interface for easy console access.

For each microservice test, the RxMicro framework performs the following actions:

1. Before starting all the test methods in the class:
    a. checks the test class for compliance with the rules of REST-based microservice testing defined by the RxMicro framework;
    b. starts an HTTP server on a random free port;
    c. creates an instance of the `BlockingHttpClient` type;
    d. connects the created `BlockingHttpClient` to the running HTTP server.
2. Before starting each test method:
    a. if necessary, invokes the methods defined using the `@BeforeThisTest` or `@BeforeIterationMethodSource` annotations;
    b. if necessary, registers the RxMicro component alternatives in the RxMicro container;
    c. registers the tested REST-based microservice on the running HTTP server;
    d. if necessary, injects the custom component alternatives to the REST-based microservice;
    e. injects a reference to the `BlockingHttpClient` instance into the test class;
    f. if necessary, creates the `System.out` mock, and injects it into the test class.
3. After performing each test method:
    a. deletes all registered components from the RxMicro container;
    b. deletes all registered REST-based microservices on the running HTTP server;
    c. if necessary, restores the `System.out`.
4. After performing all the tests in the class:
    a. clears the resources of the `BlockingHttpClient` component;
    b. stops the HTTP server and releases the selected resources.

### 14.6.1. Basic Principles

To understand the REST-based microservice testing principles, let's create the simplest microservice that returns the `"Hello World!"` message.

Since the microservice will return a JSON object, it is necessary to create a response model:

```
public final class Response {

    final String message;

    public Response(final String message) {
        this.message = requireNonNull(message);
    }
}
```

When there is the `GET` request to the microservice, it should return the `"Hello World!"` message:

```
final class MicroService {

    @GET("/")
    CompletableFuture<Response> get() {
        return completedFuture(new Response("Hello World!"));
    }
}
```

The testing process of the REST-based microservice is to perform an HTTP request after the RxMicro framework starts the tested microservice on the HTTP server. After receiving a response from the microservice, this response is compared to the expected response:

```
①
@RxMicroRestBasedMicroServiceTest(MicroService.class)
class MicroServiceTest {

    private BlockingHttpClient blockingHttpClient; ②

    @Test
    void Should_handle_GET_request() {
        final ClientHttpResponse response = blockingHttpClient.get("/");

        assertEquals(jsonObject("message", "Hello World!"), response.getBody()); ③
        assertEquals(200, response.getStatusCode());
    }
}
```

① To start the HTTP server and register the tested REST-based microservice, it is necessary to annotate the test class by the `@RxMicroRestBasedMicroServiceTest` annotation. In the parameter of this annotation it is specified which REST-based microservice class will be tested in the current

test.

② To execute **blocking** HTTP requests, the RxMicro framework supports the special `BlockingHttpClient` component. The developer must declare a reference to this component, and while starting the test, the RxMicro framework will automatically inject the created `BlockingHttpClient` class instance, using the `reflection` mechanism.

③ Upon receiving the HTTP response from the microservice, the developer should compare the response body with the expected result in the test.

To get additional info about writing tests that require JSON object comparison, please read `rxmicro.json` Module Usage section.

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-testing-junit/testing-microservice-basic`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

## 14.6.2. Features of Testing Complex Microservices that use Alternatives

### 14.6.2.1. Features of REST Client Testing

This section will cover the features of testing REST-based microservices that use REST clients.

The source code of such REST-based microservice consists of the `Response` model class, `ExternalMicroService` REST client and `ConsumeMicroService` HTTP request handler:

```
public final class Response {

    String message;

    public Response(final String message) {
        this.message = requireNonNull(message);
    }

    public Response() {
    }
}
```

```
@RestClient
public interface ExternalMicroService {

    @GET("/")
    CompletableFuture<Response> get();
}
```

```
final class MicroService {

    private final ExternalMicroService externalMicroService =
            getRestClient(ExternalMicroService.class);

    @GET("/")
    CompletableFuture<Response> get() {
        return externalMicroService.get();
    }
}
```

The most logical way to test such microservice is to create a mock alternative for the `ExternalMicroService` component:

```
@InitMocks
@RxMicroRestBasedMicroServiceTest(MicroService.class)
final class MicroServiceBusinessLogicOnlyTest {

    private BlockingHttpClient blockingHttpClient;

    @Mock
    @Alternative
    private ExternalMicroService externalMicroService;

    @Test
    void Should_delegate_call_to_ExternalMicroService() {
        when(externalMicroService.get()).thenReturn(completedFuture(new
Response("mock")));

        final ClientHttpResponse response = blockingHttpClient.get("/");

        assertEquals(jsonObject("message", "mock"), response.getBody());
    }
}
```

To get additional info about writing tests that require JSON object comparison, please read `rxmicro.json` Module Usage section.

But if on the basis of such test we build the source code coverage report, this report will show a low

degree of coverage:

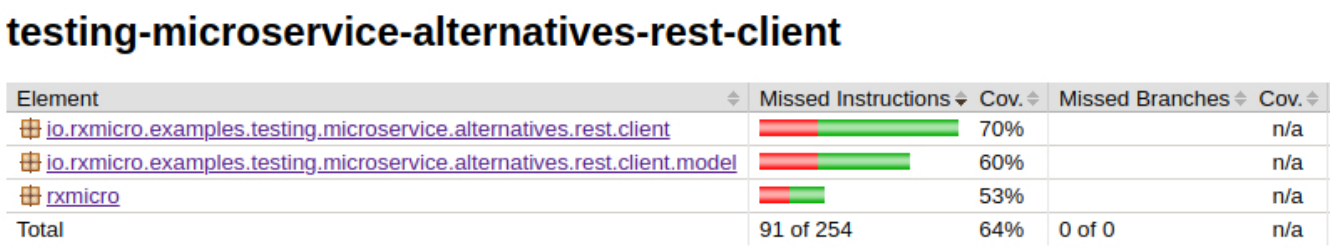



*Figure 17. The test coverage report when using a mock alternative for REST client.*

Such a result is caused by the fact that after creating a mock alternative for the `ExternalMicroService` component, the classes generated by the RxMicro framework are not used in the testing process for the REST client work.

If such a result is not acceptable, it is necessary to:

1. create a mock alternative to the `HttpClientFactory` RxMicro component;
2. use the static methods of the `HttpClientMockFactory` class to program the mock behavior.

```
@InitMocks
@RxMicroRestBasedMicroServiceTest(MicroService.class)
final class MicroServiceWithAllGeneratedCodeTest {

    private BlockingHttpClient blockingHttpClient;

    @Mock
    @Alternative
    private HttpClientFactory httpClientFactory;

    void prepareExternalMicroServiceHttpClient() {
        prepareHttpClientMock(
                httpClientFactory,
                new HttpRequestMock.Builder()
                        .setMethod(GET)
                        .setPath("/")
                        .build(),
                jsonObject("message", "mock")
        );
    }

    @Test
    @BeforeThisTest(method = "prepareExternalMicroServiceHttpClient")
    void Should_delegate_call_to_ExternalMicroService() {
        final ClientHttpResponse response = blockingHttpClient.get("/");

        assertEquals(jsonObject("message", "mock"), response.getBody());
    }
}
```

The modified test shows a coverage rate of 100%:

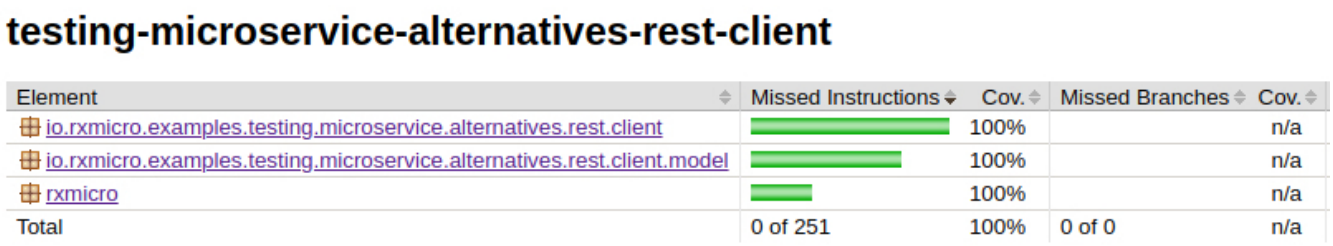



*Figure 18. The test coverage report when using the `HttpClientFactory` mock alternative.*

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-testing-junit/testing-microservice-alternatives-rest-client`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

#### 14.6.2.2. Features of Testing Mongo Repositories

This section will cover the features of testing REST-based microservices that use mongo repositories.

The source code of such REST-based microservice consists of the `Entity` entity model, `Response` model class, `DataRepository` mongo repository and `ConsumeMicroService` HTTP request handler:

```
public final class Response {

    final String message;

    public Response(final String message) {
        this.message = requireNonNull(message);
    }
}
```

```
public final class Entity {

    String data;

    public String getData() {
        return data;
    }
}
```

```
@MongoRepository(collection = "collection")
public interface DataRepository {

    @Find(query = "{_id: ?}")
    CompletableFuture<Optional<Entity>> findById(long id);
}
```

```
final class MicroService {

    private final DataRepository dataRepository =
            getRepository(DataRepository.class);

    @GET("/")
    CompletableFuture<Optional<Response>> get(final Long id) {
        return dataRepository.findById(id).thenApply(optionalEntity ->
                optionalEntity.map(entity ->
                        new Response(entity.getData())));
    }
}
```

The most logical way to test such microservice is to create a mock alternative for the `DataRepository` component:

```
@InitMocks
@RxMicroRestBasedMicroServiceTest(MicroService.class)
final class MicroServiceBusinessLogicOnlyTest {

    private BlockingHttpClient blockingHttpClient;

    @Mock
    @Alternative
    private DataRepository dataRepository;

    @Mock
    private Entity entity;

    @Test
    void Should_return_Entity_data() {
        when(entity.getData())
                .thenReturn("data");
        when(dataRepository.findById(1))
                .thenReturn(completedFuture(Optional.of(entity)));

        final ClientHttpResponse response = blockingHttpClient.get("/?id=1");

        assertEquals(jsonObject("message", "data"), response.getBody());
        assertEquals(200, response.getStatusCode());
    }

    @Test
    void Should_return_Not_Found_error() {
        when(dataRepository.findById(1))
                .thenReturn(completedFuture(Optional.empty()));

        final ClientHttpResponse response = blockingHttpClient.get("/?id=1");

        assertEquals(jsonObject("message", "Not Found"), response.getBody());
        assertEquals(404, response.getStatusCode());
    }
}
```

To get additional info about writing tests that require JSON object comparison, please read `rxmicro.json` Module Usage section.

But if on the basis of such test we build the source code coverage report, this report will show a low degree of coverage:

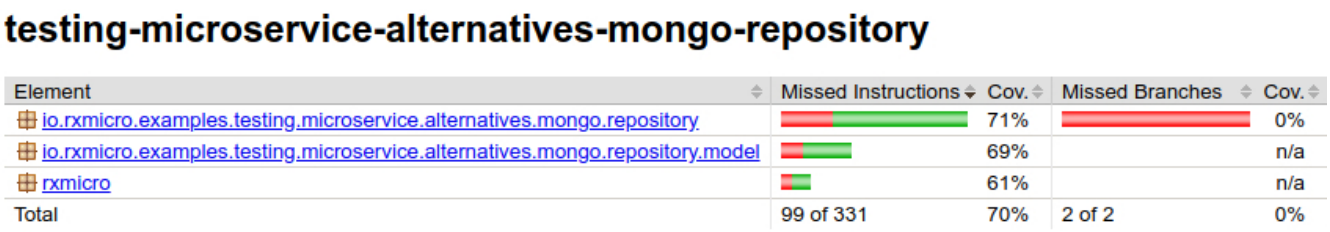



*Figure 19. The test coverage report when using a mock alternative for Mongo repository.*

Such a result is caused by the fact that after creating a mock alternative for the `DataRepository` component, the classes generated by the RxMicro framework are not used in the testing process for the mongo repository work.

If such a result is not acceptable, it is necessary to:

1. create a mock alternative to the `MongoDatabase` RxMicro component;
2. use the static methods of the `MongoMockFactory` class to program the mock behavior.

```
@InitMocks
@RxMicroRestBasedMicroServiceTest(MicroService.class)
final class MicroServiceWithAllGeneratedCodeTest {

    private static final FindOperationMock FIND_OPERATION_MOCK =
            new FindOperationMock.Builder()
                    .setAnyQuery()
                    //.setQuery("{_id: 1}")
                    .build();

    private BlockingHttpClient blockingHttpClient;

    @Mock
    @Alternative
    private MongoDatabase mongoDatabase;

    void prepareOneEntityFound() {
        prepareMongoOperationMocks(
                mongoDatabase,
                "collection",
                FIND_OPERATION_MOCK,
                new Document("data", "data")
        );
    }

    @Test
    @BeforeThisTest(method = "prepareOneEntityFound")
    void Should_return_Entity_data() {
        final ClientHttpResponse response = blockingHttpClient.get("/?id=1");

        assertEquals(jsonObject("message", "data"), response.getBody());
        assertEquals(200, response.getStatusCode());
    }
```

```
    void prepareNoEntityFound() {
        prepareMongoOperationMocks(
                mongoDatabase,
                "collection",
                FIND_OPERATION_MOCK
        );
    }

    @Test
    @BeforeThisTest(method = "prepareNoEntityFound")
    void Should_return_Not_Found_error() {
        final ClientHttpResponse response = blockingHttpClient.get("/?id=1");

        assertEquals(jsonObject("message", "Not Found"), response.getBody());
        assertEquals(404, response.getStatusCode());
    }
}
```

The modified test shows a coverage rate of 100%:

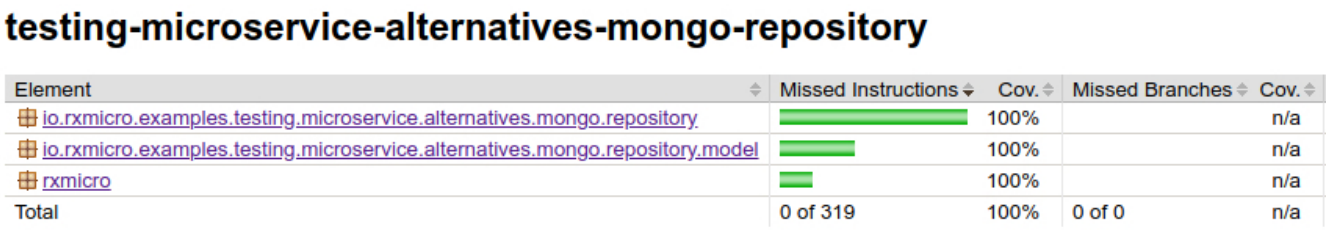



*Figure 20. The test coverage report when using the `MongoDatabase` mock alternative.*

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-testing-junit/testing-microservice-alternatives-mongo-repository`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect: `mvn clean compile`.**

#### 14.6.2.3. Features of Testing Postgres Repositories

This section will cover the features of testing REST-based microservices that use postgres repositories.

The source code of such REST-based microservice consists of the `Entity` entity model, `Response` model class, `DataRepository` mongo repository and `ConsumeMicroService` HTTP request handler:

```
public final class Response {

    final String message;

    public Response(final String message) {
        this.message = requireNonNull(message);
    }
}
```

```
@Table
public final class Entity {

    @Column(length = Column.UNLIMITED_LENGTH)
    String data;

    public String getData() {
        return data;
    }
}
```

```
@PostgreSQLRepository
public interface DataRepository {

    @Select("SELECT data FROM ${table} WHERE id=?")
    CompletableFuture<Optional<Entity>> findById(long id);
}
```

```
final class MicroService {

    private final DataRepository dataRepository = getRepository(DataRepository.class);

    @GET("/")
    CompletableFuture<Optional<Response>> get(final Long id) {
        return dataRepository.findById(id).thenApply(optionalEntity ->
                optionalEntity.map(entity ->
                        new Response(entity.getData())));
    }
}
```

The most logical way to test such microservice is to create a mock alternative for the `DataRepository` component:

```
@InitMocks
@RxMicroRestBasedMicroServiceTest(MicroService.class)
final class MicroServiceBusinessLogicOnlyTest {

    private BlockingHttpClient blockingHttpClient;

    @Mock
    @Alternative
    private DataRepository dataRepository;

    @Mock
    private Entity entity;

    @Test
    void Should_return_Entity_data() {
        when(entity.getData())
                .thenReturn("data");
        when(dataRepository.findById(1))
                .thenReturn(completedFuture(Optional.of(entity)));

        final ClientHttpResponse response = blockingHttpClient.get("/?id=1");

        assertEquals(jsonObject("message", "data"), response.getBody());
        assertEquals(200, response.getStatusCode());
    }

    @Test
    void Should_return_Not_Found_error() {
        when(dataRepository.findById(1))
                .thenReturn(completedFuture(Optional.empty()));

        final ClientHttpResponse response = blockingHttpClient.get("/?id=1");

        assertEquals(jsonObject("message", "Not Found"), response.getBody());
        assertEquals(404, response.getStatusCode());
    }
}
```

To get additional info about writing tests that require JSON object comparison, please read `rxmicro.json` Module Usage section.

But if on the basis of such test we build the source code coverage report, this report will show a low degree of coverage:

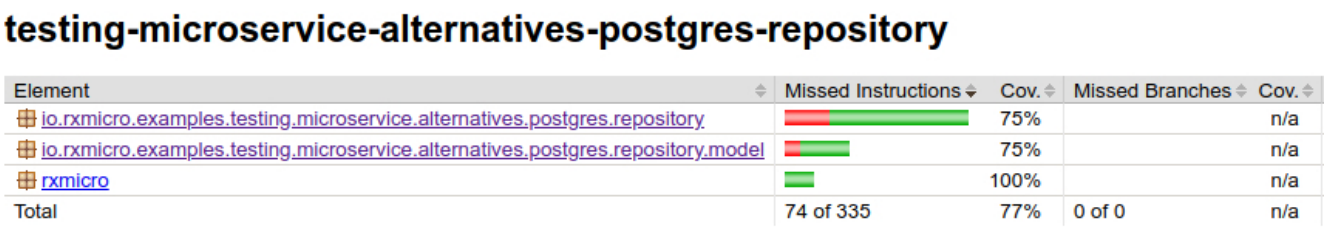



*Figure 21. The test coverage report when using a mock alternative for Postgresql repository.*

Such a result is caused by the fact that after creating a mock alternative for the `DataRepository` component, the classes generated by the RxMicro framework are not used in the testing process for the postgres repository work.

If such a result is not acceptable, it is necessary to:

1. create a mock alternative to the `ConnectionPool` RxMicro component;
2. use the static methods of the `SQLMockFactory` class to program the mock behavior.

```
@InitMocks
@RxMicroRestBasedMicroServiceTest(MicroService.class)
final class MicroServiceWithAllGeneratedCodeTest {

    private static final SQLQueryWithParamsMock SQL_PARAMS_MOCK =
            new SQLQueryWithParamsMock.Builder()
                    .setAnySql()
                    //.setSql("SELECT data FROM entity WHERE id = $1")
                    //.setBindParams(1L)
                    .build();

    private BlockingHttpClient blockingHttpClient;

    @Mock
    @Alternative
    private ConnectionPool connectionPool;

    void prepareOneEntityFound() {
        prepareSQLOperationMocks(
                connectionPool,
                SQL_PARAMS_MOCK,
                "data"
        );
    }

    @Test
    @BeforeThisTest(method = "prepareOneEntityFound")
    void Should_return_Entity_data() {
        final ClientHttpResponse response = blockingHttpClient.get("/?id=1");

        assertEquals(jsonObject("message", "data"), response.getBody());
        assertEquals(200, response.getStatusCode());
    }
```

```
    void prepareNoEntityFound() {
        prepareSQLOperationMocks(
                connectionPool,
                SQL_PARAMS_MOCK,
                List.of()
        );
    }

    @Test
    @BeforeThisTest(method = "prepareNoEntityFound")
    void Should_return_Not_Found_error() {
        final ClientHttpResponse response = blockingHttpClient.get("/?id=1");

        assertEquals(jsonObject("message", "Not Found"), response.getBody());
        assertEquals(404, response.getStatusCode());
    }
}
```

The modified test shows a coverage rate of 100%:

*Figure 22. The test coverage report when using the `ConnectionPool` mock alternative.*

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-testing-junit/testing-microservice-alternatives-postgres-repository

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 14.6.3. Custom and the RxMicro Framework Code Execution Order

For efficient writing of Rest-based microservice tests, it is necessary to know the execution order of user, and the RxMicro framework code.

When testing a Rest-based microservice, using the following test:

```
@InitMocks
@RxMicroRestBasedMicroServiceTest(MicroService.class)
final class MicroServiceTest {

    @BeforeAll
    static void beforeAll() {
    }

    private BlockingHttpClient blockingHttpClient;

    private SystemOut systemOut;

    @Mock
    @Alternative
    private BusinessService businessService;

    public MicroServiceTest() {
    }

    @BeforeEach
    void beforeEach() {
    }

    void beforeTest1UserMethod() {
    }

    @Test
    @BeforeThisTest(method = "beforeTest1UserMethod")
    void test1() {
    }

    void beforeTest2UserMethod() {
    }

    @Test
    @BeforeThisTest(method = "beforeTest2UserMethod")
    void test2() {
    }

    @AfterEach
    void afterEach() {
    }

    @AfterAll
    static void afterAll() {
    }
}
```

the execution order will be as follows:

```
RX-MICRO: Test class validated.
RX-MICRO: HTTP server started without any REST-based microservices using random free
port.
RX-MICRO: Blocking HTTP client created and connected to the started HTTP server.
USER-TEST: '@org.junit.jupiter.api.BeforeAll' invoked.

USER-TEST: new instance of the REST-based microservice test class created.
MOCKITO: All mocks created and injected.
RX-MICRO: Alternatives of the RxMicro components registered in the RxMicro runtime
containers.
RX-MICRO: Blocking HTTP client injected to the instance of the test class.
RX-MICRO: SystemOut instance created and injected to the instance of the test class.
USER-TEST: '@org.junit.jupiter.api.BeforeEach' invoked.
USER-TEST: 'beforeTest1UserMethod' invoked.
RX-MICRO: Current REST-based microservice instance created and registered in the HTTP
server.
RX-MICRO: Alternatives of the user components injected to the REST-based microservice
instance.
USER-TEST: 'test1()' invoked.
USER-TEST: '@org.junit.jupiter.api.AfterEach' invoked.
RX-MICRO: All registered alternatives removed from the RxMicro runtime containers.
RX-MICRO: Current REST-based microservice instance unregistered from the HTTP server.
RX-MICRO: System.out reset.
MOCKITO: All mocks destroyed.

USER-TEST: new instance of the REST-based microservice test class created.
MOCKITO: All mocks created and injected.
RX-MICRO: Alternatives of the RxMicro components registered in the RxMicro runtime
containers.
RX-MICRO: Blocking HTTP client injected to the instance of the test class.
RX-MICRO: SystemOut instance created and injected to the instance of the test class.
USER-TEST: '@org.junit.jupiter.api.BeforeEach' invoked.
USER-TEST: 'beforeTest2UserMethod' invoked.
RX-MICRO: Current REST-based microservice instance created and registered in the HTTP
server.
RX-MICRO: Alternatives of the user components injected to the REST-based microservice
instance.
USER-TEST: 'test2()' invoked.
USER-TEST: '@org.junit.jupiter.api.AfterEach' invoked.
RX-MICRO: All registered alternatives removed from the RxMicro runtime containers.
RX-MICRO: Current REST-based microservice instance unregistered from the HTTP server.
RX-MICRO: System.out reset.
MOCKITO: All mocks destroyed.

USER-TEST: '@org.junit.jupiter.api.AfterAll' invoked.
RX-MICRO: Blocking HTTP client released.
RX-MICRO: HTTP server stopped.
```

In the above execution order of user, and the RxMicro framework code the following clarifications are implied:

1. The `MOCKITO` prefix means that the action is activated by the `@InitMocks` annotation.
2. The `RX-MICRO` prefix means that the action is activated by the `@RxMicroRestBasedMicroServiceTest` annotation.
3. The `USER-TEST` prefix means that at this stage a custom method from the `MicroServiceTest` class is invoked.

# 14.7. Testing of Microservice Components

## 14.7.1. Basic Principles

The basic principles of component testing are covered by the Section 14.3.3, “Alternative Usage” section.

## 14.7.2. Features of Testing Complex Components that Use Alternatives

The features of testing complex components that use alternatives are the same as for REST-based microservice testing.

For more information, we recommend that You familiarize yourself with the following examples:

- Features of testing components that use REST clients:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-testing-junit/testing-component-alternatives-rest-client`;

- Features of testing components that use mongo repositories:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-testing-junit/testing-component-alternatives-mongo-repository`;

- Features of testing components that use postgres repositories:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-testing-junit/testing-component-alternatives-postgres-repository`;

### 14.7.3. Custom and the RxMicro Framework Execution Order

For efficient writing of component tests, it is necessary to know the execution order of user, and the RxMicro framework code.

When testing a component, using the following test:

```
@InitMocks
@RxMicroComponentTest(BusinessService.class)
final class BusinessServiceTest {

    @BeforeAll
    static void beforeAll() {
    }

    private BusinessService businessService;

    private SystemOut systemOut;

    @Mock
    @Alternative
    private BusinessService.ChildBusinessService childBusinessService;

    public BusinessServiceTest() {
    }

    @BeforeEach
    void beforeEach() {
    }

    void beforeTest1UserMethod() {
    }

    @Test
    @BeforeThisTest(method = "beforeTest1UserMethod")
    void test1() {
    }

    void beforeTest2UserMethod() {
    }

    @Test
    @BeforeThisTest(method = "beforeTest2UserMethod")
    void test2() {
    }

    @AfterEach
    void afterEach() {
    }

    @AfterAll
    static void afterAll() {
    }
}
```

the execution order will be as follows:

```
RX-MICRO: Test class validated.
USER-TEST: '@org.junit.jupiter.api.BeforeAll' invoked.

USER-TEST: new instance of the component test class created.
MOCKITO: All mocks created and injected.
RX-MICRO: Alternatives of the RxMicro components registered in the RxMicro runtime
containers.
RX-MICRO: SystemOut instance created and injected to the instance of the test class.
USER-TEST: '@org.junit.jupiter.api.BeforeEach' invoked.
USER-TEST: 'beforeTest1UserMethod' invoked.
RX-MICRO: Tested component instance created, if it is not created by user.
RX-MICRO: Alternatives of the user components injected to the tested component
instance.
USER-TEST: 'test1()' method invoked.
USER-TEST: '@org.junit.jupiter.api.AfterEach' invoked.
RX-MICRO: All registered alternatives removed from the RxMicro runtime containers.
RX-MICRO: System.out reset.
MOCKITO: All mocks destroyed.

USER-TEST: new instance of the component test class created.
MOCKITO: All mocks created and injected.
RX-MICRO: Alternatives of the RxMicro components registered in the RxMicro runtime
containers.
RX-MICRO: SystemOut instance created and injected to the instance of the test class.
USER-TEST: '@org.junit.jupiter.api.BeforeEach' invoked.
USER-TEST: 'beforeTest2UserMethod' invoked.
RX-MICRO: Tested component instance created, if it is not created by user.
RX-MICRO: Alternatives of the user components injected to the tested component
instance.
USER-TEST: 'test2()' method invoked.
USER-TEST: '@org.junit.jupiter.api.AfterEach' invoked.
RX-MICRO: All registered alternatives removed from the RxMicro runtime containers.
RX-MICRO: System.out reset.
MOCKITO: All mocks destroyed.

USER-TEST: '@org.junit.jupiter.api.AfterAll' invoked.
```

In the above execution order of user, and the RxMicro framework code the following clarifications are implied:

1. The `MOCKITO` prefix means that the action is activated by the `@InitMocks` annotation.
2. The `RX-MICRO` prefix means that the action is activated by the `@RxMicroComponentTest` annotation.
3. The `USER-TEST` prefix means that at this stage a custom method from the `BusinessServiceTest` class is invoked.

# 14.8. REST-based Microservice Integration Testing

The REST-based microservice integration testing allows You to test a complete system, which can consist of several REST-based microservices.

For easy writing of the REST-based microservices integration tests, the RxMicro framework provides:

1. The additional `@RxMicroIntegrationTest` annotation, that informs the RxMicro framework about the need to create an HTTP client and inject it into the tested class.
2. A special **blocking** HTTP client to execute HTTP requests during testing: `BlockingHttpClient`.
3. The `SystemOut` interface for easy console access.

The main differences between integration testing and REST-based microservice testing:

1. for integration tests, the RxMicro framework does not run an HTTP server;
2. the developer has to start and stop the system consisting of REST-based microservices;
3. the RxMicro framework does not support alternatives and additional configuration for integration tests;

### 14.8.1. Basic Principles

To demonstrate the features of the integration testing, let's look at the following microservice:

```
public final class MicroService {

    @GET("/")
    void handle() {
        System.out.println("handle");
    }
}
```

The integration test for this microservice will be as follows:

```
①
@RxMicroIntegrationTest
final class MicroServiceIntegrationTest {

    ②
    private static final int PORT = 55555;

    private static ServerInstance serverInstance;

    ⑤
    @BeforeAll
    static void beforeAll() {
        new Configs.Builder()
                .withConfigs(new HttpServerConfig()
                        .setPort(PORT))
                .build(); ④
        serverInstance = startRestServer(MicroService.class);③
    }

    ⑥
    @BlockingHttpClientSettings(port = PORT)
    private BlockingHttpClient blockingHttpClient;

    ⑦
    private SystemOut systemOut;

    @Test
    void test() {
        final ClientHttpResponse response = blockingHttpClient.get("/");

        assertEquals(200, response.getStatusCode()); ⑧
        assertTrue(response.isBodyEmpty(), "Body not empty: " + response.getBody());⑧
        assertEquals("handle", systemOut.asString()); ⑨
    }

    @AfterAll
    static void afterAll() throws InterruptedException {
        serverInstance.shutdownAndWait();
    }
}
```

① The `@RxMicroIntegrationTest` annotation informs the RxMicro framework that this test is an integration test.

② The constant declares the port that will be used to start the HTTP server.

③ The integration test requires the developer to start the HTTP server manually.

④ Before running the HTTP server, You must set the necessary HTTP port.

⑤ The configuration and start of the HTTP server must be done before running the test method.

⑥ Since a port other than the standard HTTP port is used, You need to specify which port the

`BlockingHttpClient` component should use for connecting to the HTTP server. The `BlockingHttpClient` component configuration is performed using the special `@BlockingHttpClientSettings` annotation.

⑦ The integration test supports the possibility of creating the `System.out` mock.

⑧ After receiving a response from the microservice, ensure that the request has been executed successfully.

⑨ After receiving a response from the microservice, ensure that the microservice has sent the specified message to the `System.out`.

This integration test demonstrates **all** features of the integration test activated by the `@RxMicroIntegrationTest` annotation.

Thus, the integration test unlike the REST-based microservice test can only inject a blocking HTTP client and create a mock for the `System.out`.

To get additional info about writing tests that require JSON object comparison, please read `rxmicro.json` Module Usage section.

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-testing-junit/testing-integration-basic`

When compiling, the RxMicro framework searches for `RxMicro Annotations` in the source code and generates additional classes necessary for the integral work of the microservice.

**When changing the `RxMicro Annotations` in the source code, DON'T FORGET to recompile the ALL source code, not just the changed file, for the changes to take effect:** `mvn clean compile`.

### 14.8.2. The `BlockingHttpClient` Settings

The `BlockingHttpClient` component, which is used for writing REST-based microservices and integration tests, is configured by the `@BlockingHttpClientSettings` annotation:

```
private static final int SERVER_PORT = getRandomFreePort(); ④

@BlockingHttpClientSettings(
        schema = HTTPS,                      ①
        host = "examples.rxmicro.io",        ②
        port = 9876,                         ③
        randomPortProvider = "SERVER_PORT", ④
        versionValue = "v1.1",               ⑤
        versionStrategy = HEADER,            ⑥
        requestTimeout = 15,                 ⑦
        followRedirects = Option.ENABLED     ⑧
)
private BlockingHttpClient blockingHttpClient;
```

① The `schema` parameter allows You to specify the HTTP protocol schema.

② The `host` parameter allows You to specify the remote host on which the microservice is running.
*(This parameter allows performing integration testing for remote microservices.)*

③ The `port` parameter allows You to specify the static connection port.

④ The `randomPortProvider` parameter allows You to specify the dynamic connection port.
*(The port will be read from the static final variable of the current class with the `SERVER_PORT` name.)*
*(The `port` and `randomPortProvider` parameters are mutually exclusive.)*

⑤ The `versionValue` allows You to specify the microservice version.

⑥ The `versionStrategy` parameter allows specifying the versioning strategy, which is used in the tested microservice.

⑦ The `requestTimeout` parameter allows specifying the request timeout.

⑧ The `followRedirects` parameter returns follow redirect option for the blocking HTTP client

### 14.8.3. The `docker` Usage

To perform the integration testing of microservices, it is convenient to use the `docker`.

This demonstration example uses the `docker` image: `rxmicro/simple-hello-world`.

The source code of the project, on the basis of which this `docker` image was built, is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/docker-image-hello-world-microservice`

To start the `docker` containers in the integration test it is convenient to use the `Testcontainers` Java library:

```
①
@RxMicroIntegrationTest
②
@Testcontainers
final class HelloWorldMicroService_IT {

    ③
    @Container
    private static final DockerComposeContainer<?> compose =
            new DockerComposeContainer<>(new File("docker-
compose.yml").getAbsoluteFile())
                    .withLocalCompose(true)
                    .withPull(false)
                    .withTailChildContainers(true)
                    .waitingFor("rxmicro-hello-world", Wait.forHttp("/http-health-
check"));

    private BlockingHttpClient blockingHttpClient;

    @Test
    void Should_return_Hello_World() {
        final ClientHttpResponse response = blockingHttpClient.get("/");

        assertEquals(jsonObject("message", "Hello World!"), response.getBody()); ④
        assertEquals(200, response.getStatusCode());
    }
}
```

① The `@RxMicroIntegrationTest` annotation informs the RxMicro framework that this test is an integration test.

② The `@Testcontainers` annotation activates the start and stop of the `docker` containers to be used in this test.

③ The `@Container` annotation indicates the `docker` container to be used in this test. *(To start microservices in the `docker` containers the `docker-compose` utility is used.)*

④ During the testing process, ensure that the tested microservice returns the `"Hello World!"` message.

To start the REST-based microservice in the `docker` container, the following configuration file for the `docker-compose` utility is used:

```
version: '3.7'
services:
  rxmicro-hello-world:
    image: rxmicro/simple-hello-world
    ports:
      - 8080:8080
    healthcheck:
      test: wget http://localhost:8080/http-health-check || exit 1
      interval: 30s
      timeout: 10s
      retries: 3
      start_period: 30s
```

To get additional info about writing tests that require JSON object comparison, please read `rxmicro.json` Module Usage section.

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-testing-junit/testing-integration-docker`

## 14.9. Database Testing Using DBUnit

DbUnit is an extension targeted at database-driven projects that, among other things, puts your database into a known state between test runs. This is an excellent way to avoid the myriad of problems that can occur when one test case corrupts the database and causes subsequent tests to fail or exacerbate the damage.

### 14.9.1. Test Database Configuration

To communicate with test database the RxMicro framework uses the predefined `TestDatabaseConfig` config class. This class is the usual RxMicro configuration class.

It means that settings for test database can be configured using:

- `test-database.properties` classpath resource.
- `./test-database.properties` file.
- `$HOME/test-database.properties` file.
- `$HOME/.rxmicro/test-database.properties` file.
- environment variables.
- Java system properties.
- `@WithConfig` annotation.

Besides that the settings for test database can be changed using `TestDatabaseConfig.getCurrentTestDatabaseConfig()` static method. This approach useful if test database is working at the docker container:

```
@BeforeEach
void beforeEach() {
    getCurrentTestDatabaseConfig()
            .setHost(postgresqlTestDockerContainer.getHost())
            .setPort(postgresqlTestDockerContainer.getFirstMappedPort());
}
```

### 14.9.2. Retrieve Connection Strategies

The RxMicro framework provides the following retrieve connection to the test database strategies:

- One connection per all test classes.
- One connection per test class (**default strategy**).
- One connection per test method.

#### 14.9.2.1. One connection per all test classes

This strategy informs the DBUnit to use single connection per all tests for your project.
The RxMicro team recommends using this strategy for external databases only.

```
@RxMicroIntegrationTest
@Testcontainers
①
@DbUnitTest(retrieveConnectionStrategy = PER_ALL_TEST_CLASSES)
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
// FIXME "org.postgresql.util.PSQLException: FATAL: the database system is starting
up" and remove @Ignore
@Ignore
final class DbStateUsingPerAllTestClassesConnectionTest {

    private static final int DB_PORT = getRandomFreePort();

    @WithConfig
    private static final TestDatabaseConfig CONFIG = new TestDatabaseConfig()
            .setType(DatabaseType.POSTGRES)
            .setPort(DB_PORT)
            .setUser("rxmicro")
            .setPassword("password")
            .setDatabase("rxmicro");

    @Container
    private static final GenericContainer<?> POSTGRESQL_TEST_DB =
            new FixedHostPortGenericContainer<>("rxmicro/postgres-test-db")
                    .withExposedPorts(5432)
                    .withFixedExposedPort(DB_PORT, 5432);

    @Test
    @ExpectedDataSet("dataset/rxmicro-test-dataset.xml")
    @Order(1)
    void Should_contain_expected_dataset() {

    }

    @Test
    @InitialDataSet("dataset/rxmicro-test-dataset-two-rows-only.xml")
    @ExpectedDataSet("dataset/rxmicro-test-dataset-two-rows-only.xml")
    @Order(2)
    void Should_set_and_compare_dataset() {

    }
}
```

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-testing-dbunit/testing-dbunit-basic

#### 14.9.2.2. One connection per test class

This strategy informs the DBUnit to create a new connection before run all tests for each test class and to close after running all tests for each test class.

```
@RxMicroIntegrationTest
@Testcontainers
①
@DbUnitTest
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
// FIXME "org.postgresql.util.PSQLException: FATAL: the database system is starting
up" and remove @Ignore
@Ignore
final class DbStateUsingPerClassConnectionTest {

    @Container
    private static final GenericContainer<?> POSTGRESQL_TEST_DB =
            new GenericContainer<>("rxmicro/postgres-test-db")
                    .withExposedPorts(5432);

    @BeforeAll
    static void beforeAll() {
        getCurrentTestDatabaseConfig()
                .setHost(POSTGRESQL_TEST_DB.getHost())
                .setPort(POSTGRESQL_TEST_DB.getFirstMappedPort());
    }

    @Test
    @ExpectedDataSet("dataset/rxmicro-test-dataset.xml")
    @Order(1)
    void Should_contain_expected_dataset() {

    }

    @Test
    @InitialDataSet("dataset/rxmicro-test-dataset-two-rows-only.xml")
    @ExpectedDataSet("dataset/rxmicro-test-dataset-two-rows-only.xml")
    @Order(2)
    void Should_set_and_compare_dataset() {

    }
}
```

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-testing-dbunit/testing-dbunit-basic

#### 14.9.2.3. One connection per test method

This strategy informs the DBUnit to create a new connection before each test method and to close after each one.

```
@RxMicroIntegrationTest
@Testcontainers
①
@DbUnitTest(retrieveConnectionStrategy = PER_TEST_METHOD)
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
// FIXME "org.postgresql.util.PSQLException: FATAL: the database system is starting
up" and remove @Ignore
@Ignore
final class DbStateUsingPerMethodConnectionTest {

    @Container
    private final GenericContainer<?> postgresqlTestDb =
            new GenericContainer<>("rxmicro/postgres-test-db")
                    .withExposedPorts(5432);

    @BeforeEach
    void beforeEach() {
        getCurrentTestDatabaseConfig()
                .setHost(postgresqlTestDb.getHost())
                .setPort(postgresqlTestDb.getFirstMappedPort());
    }

    @Test
    @ExpectedDataSet("dataset/rxmicro-test-dataset.xml")
    @Order(1)
    void Should_contain_expected_dataset() {

    }

    @Test
    @InitialDataSet("dataset/rxmicro-test-dataset-two-rows-only.xml")
    @ExpectedDataSet("dataset/rxmicro-test-dataset-two-rows-only.xml")
    @Order(2)
    void Should_set_and_compare_dataset() {

    }
}
```

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-testing-dbunit/testing-dbunit-basic

### 14.9.3. `@InitialDataSet` Annotation

The `@InitialDataSet` annotation inform the RxMicro framework that it is necessary to prepare the tested database using DbUnit framework before test execution. The `@InitialDataSet` annotation contains the dataset files that must be exported to the tested database before test execution.

```
@Test
①
@InitialDataSet("dataset/rxmicro-test-dataset-two-rows-only.xml")
void prepare_database() {
    // do something with prepared database
}
```

① Using data from the `@InitialDataSet` annotation the RxMicro framework prepares the tested database.

The init dataset can be provided using `flat xml` format:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE dataset SYSTEM "rxmicro-test-dataset.dtd">

<dataset>
    <account id="1"
             email="richard.hendricks@piedpiper.com"
             first_name="Richard"
             last_name="Hendricks"
             balance="70000.00"
             role="CEO"/>
    <account id="3"
             email="dinesh.chugtai@piedpiper.com"
             first_name="Dinesh"
             last_name="Chugtai"
             balance="10000.00"
             role="Lead_Engineer"/>

    <product id="24"
             name="Apple iMac 27&amp;quot; with Retina 5K Display Late (MQ2Y2)"
             price="6200.00"
             count="7"/>
    <product id="25"
             name="Apple iMac 27&amp;quot; 2017 5K (MNEA2)"
             price="2100.00"
             count="17"/>

    <order/>
</dataset>
```

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-testing-dbunit/testing-dbunit-basic

### 14.9.4. `@ExpectedDataSet` Annotation

The `@ExpectedDataSet` annotation inform the RxMicro framework that it is necessary to compare the actual database state with the expected one, defined using dataset file(s) after test execution.

```
@Test
①
@ExpectedDataSet("dataset/rxmicro-test-dataset-two-rows-only.xml")
void verify_database_state() {
    // change database state
}
```

① Using data from the `@ExpectedDataSet` annotation the RxMicro framework compares actual database state with the expected one after test execution.

The expected dataset can be provided using `flat xml` format:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE dataset SYSTEM "rxmicro-test-dataset.dtd">

<dataset>
    <account id="1"
             email="richard.hendricks@piedpiper.com"
             first_name="Richard"
             last_name="Hendricks"
             balance="70000.00"
             role="CEO"/>
    <account id="3"
             email="dinesh.chugtai@piedpiper.com"
             first_name="Dinesh"
             last_name="Chugtai"
             balance="10000.00"
             role="Lead_Engineer"/>

    <product id="24"
             name="Apple iMac 27&amp;quot; with Retina 5K Display Late (MQ2Y2)"
             price="6200.00"
             count="7"/>
    <product id="25"
             name="Apple iMac 27&amp;quot; 2017 5K (MNEA2)"
             price="2100.00"
             count="17"/>

    <order/>
</dataset>
```

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-testing-dbunit/testing-dbunit-basic

### 14.9.5. `@RollbackChanges` Annotation

The `@RollbackChanges` annotation starts a new transaction before initialization of database by the `@InitialDataSet` annotation (if it is present) and rolls back this transaction after comparing actual database state with expected one provided by the `@ExpectedDataSet` annotation (if it is present).

The isolation level of the test transaction can be configured using `isolationLevel` parameter declared at the `@RollbackChanges` annotation.

```
@RxMicroIntegrationTest
@Testcontainers
①
@DbUnitTest
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
// FIXME "org.postgresql.util.PSQLException: FATAL: the database system is starting
up" and remove @Ignore
@Ignore
final class RollbackChangesTest {

    @Container
    private static final GenericContainer<?> POSTGRESQL_TEST_DB =
            new GenericContainer<>("rxmicro/postgres-test-db")
                    .withExposedPorts(5432);

    @BeforeAll
    static void beforeAll() {
        getCurrentTestDatabaseConfig()
                .setHost(POSTGRESQL_TEST_DB.getHost())
                .setPort(POSTGRESQL_TEST_DB.getFirstMappedPort());
    }

    @Test
    ②
    @RollbackChanges
    @InitialDataSet("dataset/rxmicro-test-dataset-empty.xml")
    @ExpectedDataSet("dataset/rxmicro-test-dataset-empty.xml")
    @Order(1)
    void Should_set_and_compare_dataset() {

    }

    @Test
    @ExpectedDataSet("dataset/rxmicro-test-dataset.xml")
    @Order(2)
    void Should_contain_expected_dataset() {

    }
}
```

① For database testing it is necessary to inform the RxMicro framework that current test is DBUnit

test.

② If test method annotated by the `@RollbackChanges` annotation all changes made by this test method will be rolled back.

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-testing-dbunit/testing-dbunit-basic`

### 14.9.6. Ordered Comparison

If the expected dataset is ordered, it is necessary to inform the RxMicro framework how to compare this dataset correctly. For this case the `@ExpectedDataSet` annotation has `orderBy` parameter:

```
@Test
@ExpectedDataSet(
        value = "dataset/rxmicro-test-dataset-products-order-by-price.xml",
        orderBy = "price" ①
)
@Order(1)
void Should_contain_expected_dataset() {

}
```

① The `orderBy` parameter contains the column name(s) that must be used to sort the actual dataset before comparison.

The `dataset/rxmicro-test-dataset-products-order-by-price.xml` dataset is the following:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE dataset SYSTEM "rxmicro-test-dataset.dtd">

<dataset>
    <product id="19" name="Apple iPod Touch 6 16Гб (MKGX2)" price="310.00"
count="48"/>
    <product id="17" name="Apple iPod Touch 6 16Гб (MKH62)" price="320.00"
count="54"/>
    <product id="18" name="Apple iPod Touch 6 32Гб (MKJ02)" price="380.00"
count="52"/>
    <product id="20" name="Apple iPod Touch 6 32Гб (MKHV2)" price="420.00"
count="55"/>
    <product id="14" name="Apple iPhone 7 Plus 32GB Black" price="510.00" count="3"/>
    <product id="6" name="Apple iPad (2019) 10.2&amp;quot; Wi-Fi 32GB Gold (MW762)"
price="540.00"
             count="32"/>
    <product id="7" name="Apple iPad (2019) 10.2&amp;quot; Wi-Fi 128GB Silver (MW782)"
price="620.00"
             count="37"/>
    <product id="8" name="Apple iPad mini 5 Wi-Fi 64Gb Space Gray (MUQW2)"
price="645.00"
             count="26"/>
    <product id="12" name="Apple iPhone Xr 64GB Black (MRY42)" price="760.00"
count="14"/>
    <product id="10" name="Apple iPhone Xs 64GB Space Gray (MT9E2)" price="840.00"
count="21"/>
    <product id="13" name="Apple iPhone Xs 256GB Space Gray (MT9H2)" price="910.00"
count="10"/>
    <product id="11" name="Apple iPhone 11 128GB Black" price="980.00" count="18"/>
    <product id="2" name="Apple MacBook A1534 12&amp;quot; Space Gray (MNYF2)"
```

```
price="985.00"
             count="12"/>
    <product id="9" name="Apple iPad Pro 11&amp;quot; Wi-Fi 64GB Space Gray 2018 (MTXN2)"
price="1100.00"
             count="18"/>
    <product id="4" name="Apple MacBook Pro 13 Retina Space Gray (MPXT2) 2017"
price="1345.00"
             count="17"/>
    <product id="15" name="Apple iPhone 11 Pro 64GB Space Gray" price="1450.00"
count="42"/>
    <product id="16" name="Apple iPhone 11 Pro 256GB Midnight Green" price="1720.00"
count="38"/>
    <product id="22" name="Apple iMac 21.5&amp;apos; Middle 2017 (MMQA2)" price="1740.00"
count="14"/>
    <product id="5" name="Apple MacBook Pro 15&amp;quot; Retina Z0RF00052 (Mid 2015)"
price="1860.00"
             count="11"/>
    <product id="23" name="Apple iMac 21&amp;quot; Retina 4K MRT32 (Early 2019)"
price="1920.00"
             count="11"/>
    <product id="25" name="Apple iMac 27&amp;quot; 2017 5K (MNEA2)" price="2100.00"
count="17"/>
    <product id="3" name="Apple MacBook Pro 16&amp;quot; 512GB 2019 (MVVJ2) Space Gray"
price="2540.00"
             count="8"/>
    <product id="1" name="Apple MacBook Pro 15&amp;quot; Retina Z0WW00024 Space Gray"
price="5750.00"
             count="10"/>
    <product id="24" name="Apple iMac 27&amp;quot; with Retina 5K Display Late (MQ2Y2)"
price="6200.00"
             count="7"/>
    <product id="21" name="Apple iMac Pro 27&amp;quot; Z0UR000AC / Z0UR8 (Late 2017)"
price="7800.00"
             count="6"/>
</dataset>
```

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-testing-dbunit/testing-dbunit-basic

### 14.9.7. Supported Expressions

The RxMicro framework supports expressions for datasets.
Expressions can be useful to set dynamic parameters.

The RxMicro framework supports the following expressions:

- `${null}` - null value.
- `${now}` - is `java.time.Instant.now()` value for the initial dataset;
  - `${instant:now}` is alias for `${now}`;
  - `${timestamp:now}` is alias for `${now}`;
- `${now}` - is `java.time.Instant.now()` value for the expected dataset;
  - `${instant:now}` is alias for `${now}`;
  - `${timestamp:now}` is alias for `${now}`;
  - `${now:${CUSTOM-DURATION}}` is alias for `${now}`;
  - `${instant:now:${CUSTOM-DURATION}}` is alias for `${now}`;
  - `${timestamp:now:${CUSTOM-DURATION}}` is alias for `${now}`;
- `${interval:${MIN}:${MAX}}` - is an instant interval that can be compared with `java.time.Instant` and `java.sql.Timestamp` instances correctly.
  - `${interval:${MEDIAN}:${LEFT-DELTA}:${RIGHT-DELTA}}` is alias for `${interval:${MIN}:${MAX}}`;
  - `${instant:interval:${MEDIAN}:${LEFT-DELTA}:${RIGHT-DELTA}}` is alias for `${interval:${MIN}:${MAX}}`;
  - `${timestamp:interval:${MEDIAN}:${LEFT-DELTA}:${RIGHT-DELTA}}` is alias for `${interval:${MIN}:${MAX}}`;
  - `${instant:interval:${MIN}:${MAX}}` is alias for `${interval:${MIN}:${MAX}}`;
  - `${timestamp:interval:${MIN}:${MAX}}` is alias for `${interval:${MIN}:${MAX}}`;
- `${int:interval:${MIN}:${MAX}}` - is an integer number interval that can be compared with `java.lang.Short`, `java.lang.Integer` and `java.lang.Long` instances correctly.
  - `${integer:interval:${MIN}:${MAX}}` is alias for `${int:interval:${MIN}:${MAX}}`;
  - `${tinyint:interval:${MIN}:${MAX}}` is alias for `${int:interval:${MIN}:${MAX}}`;
  - `${short:interval:${MIN}:${MAX}}` is alias for `${int:interval:${MIN}:${MAX}}`;
  - `${smallint:interval:${MIN}:${MAX}}` is alias for `${int:interval:${MIN}:${MAX}}`;
  - `${long:interval:${MIN}:${MAX}}` is alias for `${int:interval:${MIN}:${MAX}}`;
  - `${bigint:interval:${MIN}:${MAX}}` is alias for `${int:interval:${MIN}:${MAX}}`;

#### 14.9.7.1. `${null}` Expression

If dataset must contain `null` value, the `${null}` expression must be used:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE dataset SYSTEM "rxmicro-test-dataset.dtd">

<dataset>
    <account id="1"
             email="richard.hendricks@piedpiper.com"
             first_name="Richard"
             last_name="Hendricks"
             role="CEO"
             balance="${null}"/> ①
</dataset>
```

① Set `null` value to the `balance` column.

```
@Test
@InitialDataSet(
        ①
        value = "dataset/with-null-expression-dataset.xml",
        ②
        executeStatementsBefore = "ALTER TABLE account ALTER COLUMN balance DROP NOT
NULL")
④
@ExpectedDataSet("dataset/with-null-expression-dataset.xml")
void Should_set_and_compare_null_correctly() throws SQLException {
    try (Connection connection = getJDBCConnection()) {
        try (Statement st = connection.createStatement()) {
            try (ResultSet rs = st.executeQuery("SELECT balance FROM account WHERE
id=1")) {
                if (rs.next()) {
                    ③
                    assertNull(rs.getTimestamp(1,
GREGORIAN_CALENDAR_WITH_UTC_TIME_ZONE));
                } else {
                    fail("Test database does not contain account with id=1");
                }
            }
        }
    }
}
```

① The dataset with `${null}` expression.

② The test database does not support `null` values for `account.balance` column. To demonstrate how `${null}` expression is worked it is necessary to drop `NOT NULL` constraint.

③ The actual row contains `null` value.

④ After test execution the test database must contain `null` value.

The project source code used in the current subsection is available at the following link:

`https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-testing-dbunit/testing-dbunit-expressions`

#### 14.9.7.2. `${now}` Expression

The `${now}` expression useful if it is necessary to work with current instant: set and compare.

If dataset must contain `java.time.Instant.now()` value, the `${now}` expression must be used:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE dataset SYSTEM "rxmicro-test-dataset.dtd">

<dataset>
    <order id="1"
           id_account="1"
           id_product="1"
           count="10"
           created="${now}"/> ①
</dataset>
```

① Set current instant value to the `created` column.

```
@Test
@InitialDataSet(
        ①
        value = "dataset/with-now-expression-dataset.xml",
        ②
        executeStatementsBefore = {
                "ALTER TABLE \"order\" DROP CONSTRAINT IF EXISTS order_fk_account",
                "ALTER TABLE \"order\" DROP CONSTRAINT IF EXISTS order_fk_product"
        })
③
@ExpectedDataSet("dataset/with-now-expression-dataset.xml")
void Should_set_and_compare_now_correctly() throws SQLException {
    try (Connection connection = getJDBCConnection()) {
        try (Statement st = connection.createStatement()) {
            try (ResultSet rs = st.executeQuery("SELECT created FROM \"order\" WHERE
id=1")) {
                if (rs.next()) {
                    ④
                    final Instant actual =
                            rs.getTimestamp(1,
GREGORIAN_CALENDAR_WITH_UTC_TIME_ZONE).toInstant();
                    ⑤
                    assertInstantEquals(Instant.now(), actual);
                } else {
                    fail("Test database does not contain order with id=1");
                }
            }
        }
    }
}
```

① The dataset with `${now}` expression.

② The test database contains foreign keys. To demonstrate how `${now}` expression is worked it is necessary to drop these foreign keys.

③ The expected dataset must contain current instant value too.

④ The actual row contains current instant value.

⑤ To compare current instant value it is necessary to use `ExAssertions.assertInstantEquals` method. This method verifies that instants are equal within the default `delta` configured via `GlobalTestConfig` config class.

The `${now}` expression can be used to verify that unit test creates `java.time.Instant.now()` value:

```
@Test
@InitialDataSet(
        ①
        value = "dataset/empty-database.xml",
        ②
        executeStatementsBefore = {
                "ALTER TABLE \"order\" DROP CONSTRAINT IF EXISTS order_fk_account",
                "ALTER TABLE \"order\" DROP CONSTRAINT IF EXISTS order_fk_product"
        })
④
@ExpectedDataSet("dataset/with-now-expression-dataset.xml")
void Should_compare_now_correctly() throws SQLException {
    final String sql = "INSERT INTO \"order\" VALUES(?,?,?,?,?)";
    try (Connection connection = getJDBCConnection()) {
        try (PreparedStatement st = connection.prepareStatement(sql)) {
            st.setInt(1, 1);
            st.setInt(2, 1);
            st.setInt(3, 1);
            st.setInt(4, 10);
            final Timestamp now = Timestamp.from(Instant.now()); ③
            st.setTimestamp(5, now, GREGORIAN_CALENDAR_WITH_UTC_TIME_ZONE);
            st.executeUpdate();
        }
    }
}
```

① The empty dataset.

② The test database contains foreign keys. To demonstrate how `${now}` expression is worked it is necessary to drop these foreign keys.

③ The `java.time.Instant.now()` value is stored to test database.

④ After test execution the test database must contain `java.time.Instant.now()` value.

The `dataset/with-now-expression-dataset.xml` classpath resource contains the following content:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE dataset SYSTEM "rxmicro-test-dataset.dtd">

<dataset>
    <order id="1"
           id_account="1"
           id_product="1"
           count="10"
           created="${now}"/> ①
</dataset>
```

① The `created` column must contain `java.time.Instant.now()` value.

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-testing-dbunit/testing-dbunit-expressions

#### 14.9.7.3. `${instant:interval}` Expression

The `${instant:interval}` expression allows comparing this expression with `java.time.Instant` and `java.sql.Timestamp` instances correctly.

This expression is useful if Your business logic generates random instant value within predefined boundaries:

```
@Test
@InitialDataSet(
        ①
        value = "dataset/empty-database.xml",
        ②
        executeStatementsBefore = {
                "ALTER TABLE \"order\" DROP CONSTRAINT IF EXISTS order_fk_account",
                "ALTER TABLE \"order\" DROP CONSTRAINT IF EXISTS order_fk_product"
        })
④
@ExpectedDataSet("dataset/with-instant-interval-expression-dataset.xml")
void Should_compare_instant_interval_correctly() throws SQLException {
    final String sql = "INSERT INTO \"order\" VALUES(?,?,?,?,?)";
    try (Connection connection = getJDBCConnection()) {
        try (PreparedStatement st = connection.prepareStatement(sql)) {
            st.setInt(1, 1);
            st.setInt(2, 1);
            st.setInt(3, 1);
            st.setInt(4, 10);
            final Duration duration = Duration.ofSeconds(new Random().nextInt(10) +
1);
            final Timestamp now = Timestamp.from(Instant.now().plus(duration)); ③
            st.setTimestamp(5, now, GREGORIAN_CALENDAR_WITH_UTC_TIME_ZONE);
            st.executeUpdate();
        }
    }
}
```

① The empty dataset.

② The test database contains foreign keys. To demonstrate how `${now}` expression is worked it is necessary to drop these foreign keys.

③ The `java.time.Instant.now() + [1 SECOND - 10 SECOND]` value is stored to test database.

④ After test execution the test database must contain `java.time.Instant.now() + [1 SECOND - 10 SECOND]` value.

The `dataset/with-now-expression-dataset.xml` classpath resource contains the following content:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE dataset SYSTEM "rxmicro-test-dataset.dtd">

<dataset>
    <order id="1"
           id_account="1"
           id_product="1"
           count="10"
           created="${interval:now:PT1S:PT12S}"/> ①
</dataset>
```

① The `created` column must contain `java.time.Instant.now() + [1 SECOND - 12 SECOND]` value. `12` instead of `10` is used because 2 second is compare delta!

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-testing-dbunit/testing-dbunit-expressions

#### 14.9.7.4. `${int:interval}` Expression

The `${int:interval}` expression allows comparing this expression with `java.lang.Short`, `java.lang.Integer` and `java.lang.Long` instances correctly.

This expression is useful if Your business logic generates random integer number value within predefined boundaries:

```
@Test
@InitialDataSet("dataset/empty-database.xml")
②
@ExpectedDataSet("dataset/with-int-interval-expression-dataset.xml")
void Should_compare_int_interval_correctly() throws SQLException {
    final String sql = "INSERT INTO product VALUES(?,?,?,?)";
    try (Connection connection = getJDBCConnection()) {
        try (PreparedStatement st = connection.prepareStatement(sql)) {
            st.setInt(1, 1);
            st.setString(2, "name");
            st.setBigDecimal(3, new BigDecimal("6200.00"));
            st.setInt(4, new Random().nextInt(10)); ①
            st.executeUpdate();
        }
    }
}
```

① The actual product count is random value from `0` to `9`. To compare the actual dataset it is necessary to use `${int:interval}` expression.

② After test execution the test database must contain random integer number value from `0` to `9`.

The expected dataset with the integer number interval from `0` to `9`:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE dataset SYSTEM "rxmicro-test-dataset.dtd">

<dataset>
    <product id="1"
             name="name"
             price="6200.00"
             count="${int:interval:0:9}"/> ①
</dataset>
```

① Set integer number interval from `0` to `9`.

The project source code used in the current subsection is available at the following link:

https://github.com/rxmicro/rxmicro-usage/tree/master/examples/group-testing-dbunit/testing-dbunit-expressions

# 15. Appendices

## 15.1. Appendix A: FAQ

This section covers a list of possible questions that You can ask yourself when You use the RxMicro framework.

### 15.1.1. Does the RxMicro framework modify my byte code?

No. Your classes are Your classes. The RxMicro framework does not transform classes or modify the Java byte code You write. The RxMicro framework produces additional classes only.

### 15.1.2. Can the RxMicro framework be used for purposes other than microservices?

Yes. The RxMicro framework is very modular and You can choose to use any module You need.

### 15.1.3. Why I receive `class not found` error?

When You use the RxMicro framework You can receive one of the following errors:

- `Class rxmicro.$$RestControllerAggregatorImpl not found`;
- `Class rxmicro.$$RestClientFactoryImpl not found`;
- `Class rxmicro.$$RepositoryFactoryImpl not found`;
- `Class rxmicro.$$BeanFactoryImpl not found`;

These errors mean that the `RxMicro Annotation Processor` does not generate the required additional classes.

To fix it, please verify that:

1. `pom.xml` for Your project contains the valid settings for `maven-compiler-plugin` and
2. You executed command: `mvn clean compile`!

### 15.1.4. Why I receive `The Kotlin standard library is not found in the module graph` error?

Sometimes this issue occurs during the work with java code using IntelliJ IDEA. To fix this issue it is necessary to rebuild Your project: `Build` → `Rebuild project`.

### 15.1.5. Why I receive `java.lang.NullPointerException: autoRelease couldn't be null` error during unit testing?

If You declare an alternative of the `HttpClientFactory` mock and don't configure it this error can happen.

To fix this issue it is necessary to configure the `@Mock` annotation:

```
@Alternative
@Mock(answer = Answers.RETURNS_DEEP_STUBS) ①
private HttpClientFactory httpClientFactory;
```

① Set `Answers.RETURNS_DEEP_STUBS` as answer value.

### 15.1.6. Why I receive `java.lang.reflect.InaccessibleObjectException: Unable to make MicroServiceTest() accessible: module module.name does not "opens test.package" to unnamed module` error during unit testing?

If `test.package` name matches to the `production.package` name, then You need just to add `io.rxmicro.annotation.processor.RxMicroTestsAnnotationProcessor` annotation processor. (Read more at Section 14.1.3, “Configuring the `maven-compiler-plugin`:”).

If `test.package` name does not match to the `production.package` name, then You need configure the `maven-surefire-plugin` manually:

```
<plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>${maven-surefire-plugin.version}</version>
    <configuration>
        <argLine>
            @{argLine}
            --add-opens module.name/test.package=ALL-UNNAMED ①
        </argLine>
    </configuration>
</plugin>
```

① Opens all classes from `test.package` package defined at the `module.name` module to all unnamed modules!

## 15.2. Appendix B: Useful Links

- RxMicro Source Code Repository;
- RxMicro Documentation Repository;
- RxMicro Examples Repository;
- RxMicro Docker Repository;
- RxMicro on Maven Central

```
<script async src="https://www.googletagmanager.com/gtag/js?id=UA-166298262-
2"></script>
<script>
window.dataLayer = window.dataLayer || [];
function gtag(){dataLayer.push(arguments);}
gtag('js', new Date());
gtag('config', 'UA-166298262-2');
</script>
```