Farm Water Delivery - Water Ordering API

This application is a water ordering API which provides a set of endpoints covering the specification of the requirements described here.

The application features multiple endpoints for ordering water, querying existing orders and cancelling an order. These operations have side effects on the data structure of an in-memory PersistenceMechanism which internally uses a `Map<K,V>` data structure while keeping track of these changes. These changes are also written to disk to prevent lost updates on next application restart and to keep both the in-memory cache and dataset in a eventually consistent state. The dataset path is located here. On startup the application gets preloaded and initialized with a set of delivery orders from the same JSON dataset.

Deliverables

1. Swagger2 API documentation
2. Sample request and response documentation: Sample request and response snippets
3. Source code
4. Documentation
5. Test harness
6. Test coverage

Table of Contents

Application design and some design decisions

A class diagram showing how the various pieces and components fit together can be found here. Public interface methods to the system contain code documentation describing the operation.

Package structure

The project source code is partitioned into three packages as follows:

1. **businessactivities**: This package contains all business domain and API related source files.
2. **infrastructure**: Framework-level source files, utilities and configuration source files.
3. **sharedkernel**: Shared domain entities.

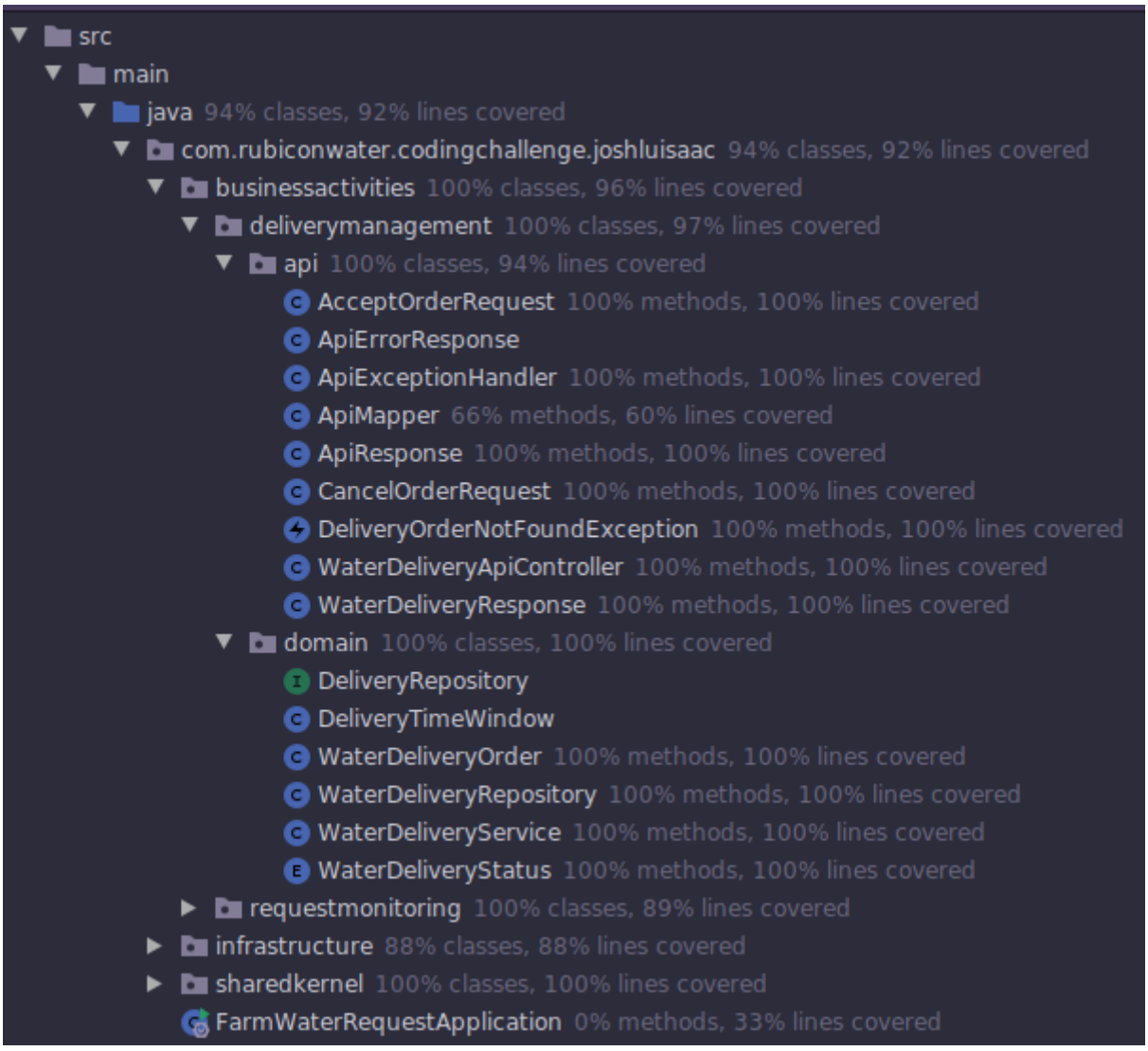

Figure 1: Project package structure

Swagger - Generated API Documentation

The project has got a dependency on Swagger2 which facilitated the generation of API docs. This documents the API contracts and allows the user to interact with the various endpoints. This could be accessed using `http://localhost:8887/swagger-ui.html`

Prerequisites

- Adopt/Open/Oracle JDK 11+ or higher (One of Adopt, Open or Oracle JDK)
- Apache Maven v3.6.1 or Gradle v6.0.1 (SDKMAN makes it very easy to set up these build tools).

You can follow the steps here on how to setup SDKMAN and to install gradle using SDKMAN refer to this link

Gradle and gradle wrapper

The application comes bundled with Gradle wrapper which makes it easy to compile, test, build and run the application without having to worry about downloading Gradle. The following gradle wrapper scripts *nix or on Windows will take care of this.

Framework stack

The following libraries and dependencies were used to develop this project:

1. **SpringBoot**: Web MVC and DI framework.
2. **Hibernate validator**: For validating HTTP request body and cascading validation constraints.
3. **Lombok** : Used to auto generate getters, setters, constructors and builders for entity and value objects.
4. **FasterXML Jackson**: For deserialization of request body and serialization of response.
5. **Google Guava**: For manipulating collections.
6. **Swagger2**: API documentation.

Testing stack

1. **Mockito**
2. **Junit5**
3. **AssertJ**
4. **MockMvc**

Maven/Gradle - Running the test suite

Running this command will compile as well as run all tests

```
mvn compile test
```

or using Gradle

```
./gradlew test
```

Executing this command using maven will yield the following console output:

Test Summary

56	0	0	1.389s
tests	failures	ignored	duration

100% successful

Packages Classes

Package	Tests
com.rubiconwater.codingchallenge.joshluisaac.businessactivities.deliverymanagement.api	15
com.rubiconwater.codingchallenge.joshluisaac.businessactivities.deliverymanagement.domain	24
com.rubiconwater.codingchallenge.joshluisaac.businessactivities.requestmonitoring	5
com.rubiconwater.codingchallenge.joshluisaac.infrastructure	6
com.rubiconwater.codingchallenge.joshluisaac.infrastructure.common	2
com.rubiconwater.codingchallenge.joshluisaac.infrastructure.interceptors	4

Figure 2: Gradle test summary

```
[INFO] Tests run: 13, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
1.284 s - in
com.rubiconwater.codingchallenge.joshluisaac.businessactivities.deliveryma
nagement.api.WaterDeliveryApiControllerTest
[INFO] Running
com.rubiconwater.codingchallenge.joshluisaac.businessactivities.deliveryma
nagement.api.CancelOrderRequestTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.002 s - in
com.rubiconwater.codingchallenge.joshluisaac.businessactivities.deliveryma
nagement.api.CancelOrderRequestTest
2019-12-27 23:49:30,099 INFO  Shutting down ExecutorService
'applicationTaskExecutor'
2019-12-27 23:49:30,099 INFO  Shutting down ExecutorService
'applicationTaskExecutor'
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 54, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- jacoco-maven-plugin:0.8.2:report (report) @ farm-water-delivery
---
[INFO] Loading execution data file
/media/joshua/martian/jobs/WaterDelivery/target/jacoco.exec
[INFO] Analyzed bundle 'Farm Water Delivery' with 37 classes
[INFO] ------------------------------------------------------------------
-----
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------
```

```
-----
[INFO] Total time: 15.113 s
[INFO] Finished at: 2019-12-27T23:49:30+11:00
[INFO] ------------------------------------------------------------------------
-----
```

Maven/Gradle - Building the source

The commands below will download all the required dependencies and create an executable JAR file in the target directory. The executable JAR was created using Spring Boot Maven Plugin

```
mvn clean install
```

or using Gradle

```
./gradlew clean build
```

Maven/Gradle - Building and running the app from terminal in one command

Execute the below command to build and execute the app from terminal

```
mvn clean install && java -jar target/farm-water-delivery-0.0.1-
SNAPSHOT.jar
```

or using Gradle

```
./gradlew clean build && java -jar build/libs/FarmWaterDelivery-0.0.1-
SNAPSHOT.jar
```

Maven/Gradle - Running the app from spring boot

Execute `mvn spring-boot:run` from terminal

Or using Gradle `./gradlew bootRun`

These commands would run the application in-place without actually building a target JAR file

You should see the following logged to console:

```
2019-12-27 23:58:38,459 INFO  Starting FarmWaterRequestApplication on
xubuntuVostro with PID 21847 (/media/joshua/martian/jobs/
```

```
.....
2019-12-27 23:58:40,696 INFO  Completed initialization in 7 ms
2019-12-27 23:58:40,742 INFO  Started ServerConnector@16c63f5{HTTP/1.1,
[http/1.1]}{0.0.0.0:8887}
2019-12-27 23:58:40,744 INFO  Jetty started on port(s) 8887 (http/1.1)
with context path '/'
2019-12-27 23:58:40,749 INFO  Started FarmWaterRequestApplication in 2.782
seconds (JVM running for 3.606)
```

Accessing the application

You an access the app using this URL `http://localhost:8887` running on default port **8887**.

Port number is configurable. Just in case you have another service running on port 8887, you can update the port number in application.properties using this property below

`server.port=YOUR_NEW_PORT_NUMBER`

Code coverage

Jacoco code coverage

While the goal of the test harness was to cover most edge and corner cases, that naturally led to a wider coverage of over 85%. Code coverage was both executed as part of maven build cycle using JaCoCo and from IntelliJ IDE.

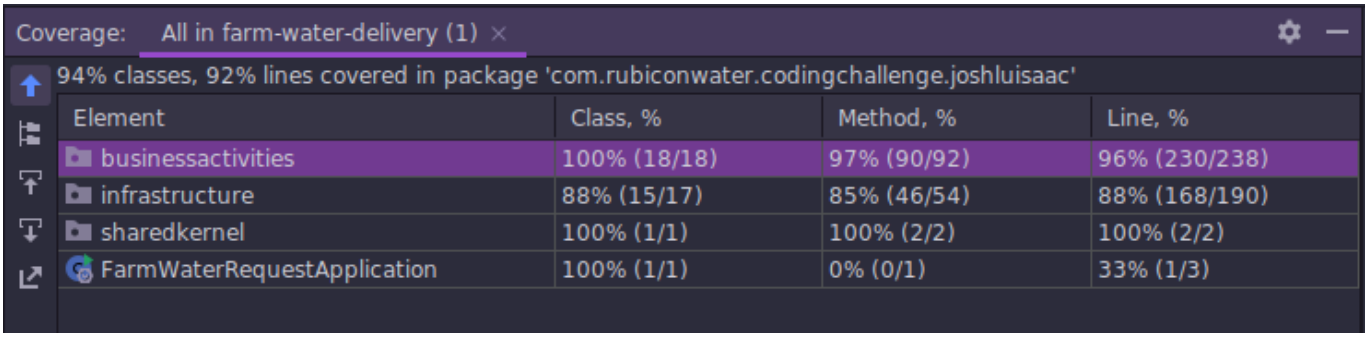

Figure 3: IntelliJ code coverage report

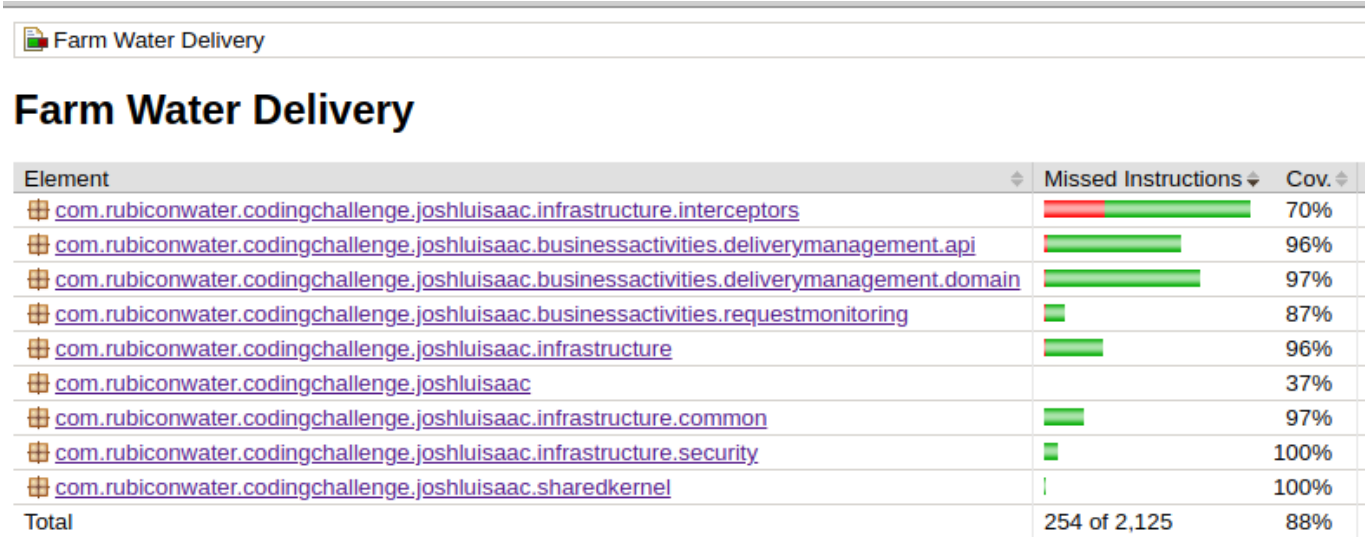

Figure 4: Jacoco code coverage report

Coverall report

Executing the following command will generate Jacoco and coveralls coverage reports.

```
mvn clean test jacoco:report coveralls:report
```

```
./gradlew build jacocoTestReport
```

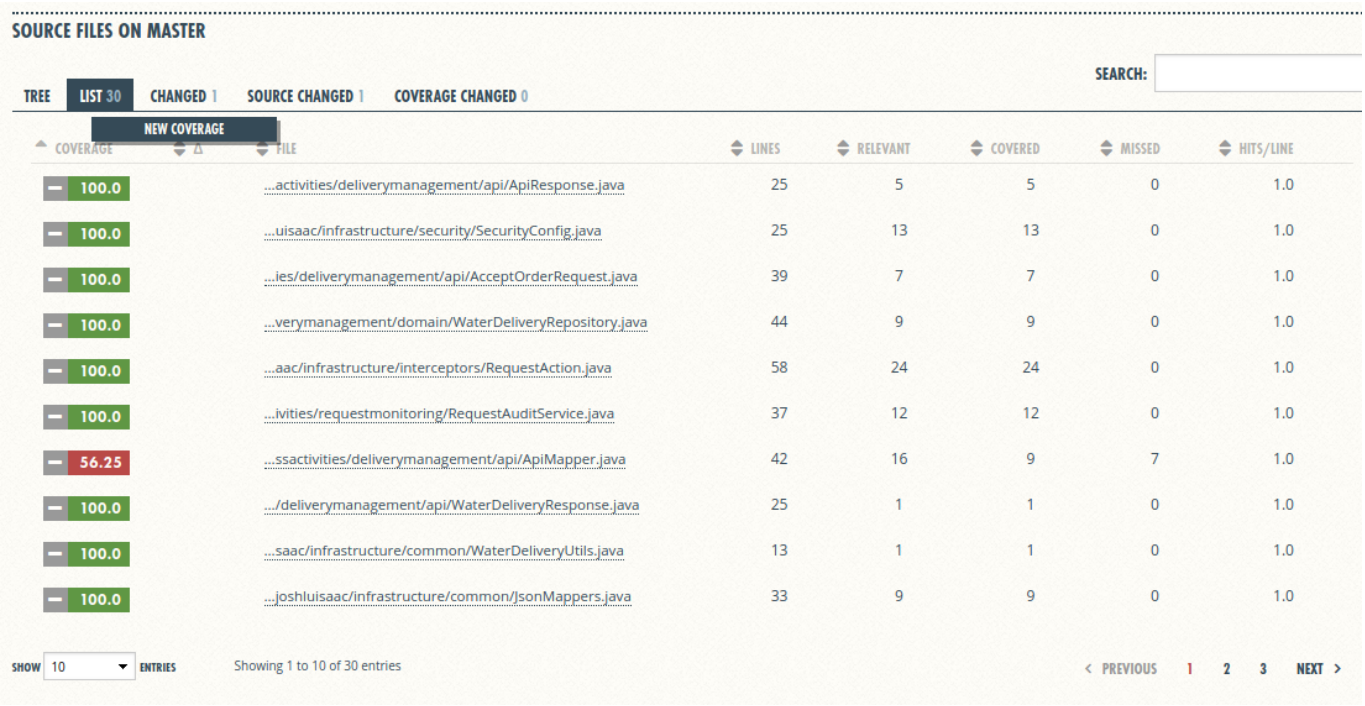

Figure 5: Coveralls

Code formatting

Source code was formatted using google-java-format