Homework 1
Software Quality and Testing – 2018/19

Author: Filipe Pires, 85122
Teaching Staff: Ilídio C. Oliveira

Table of Contents

Introduction

Test driven development (TDD) is a programming practice aimed to ensure that all production code is clear, simple and bug-free. It has been adopted by many of today's software development teams as it offers undeniable advantages on the final product's quality.
The homework this report reffers to is an exercise proposed by professor Ilídio C. Oliveira for the discipline of Software Quality and Testing of the degree in Informatics Engineering at the University of Aveiro and aims to practice and demonstrate the use of several testing levels in a web project with multiple layers. The proposed web project was a simple weather forecast service with a minimalist webpage, an integrated REST API and the use of external data sources. The task is described in detail in the document given by the professor and it is assumed that the reader has knowledge about its content (1).
All code developed is publicly available through GitHub at https://github.com/FilipePires98/TQS-HW1.

1. System Architecture

In this chapter the architecture of solution is discussed, along with the testing applied to the entities during the development phase. But first of all, it is important to provide a clean definition of what the product should be.
The weather forecast system (WFS) is a Java-based web system that makes available the daily weather conditions of any place on earth at any given past time and the predictions for the future up to seven days ahead. It consumes data freely sent by an external API and, in return, supports HTTP requests to its own API. It deploys a web application (WebApp) once started that accepts requests from users through a web browser and consumes the internal API to respond to those requests. The system also supports a caching system in-memory that stores previously made requests to the API and deletes each entry after a given amount of time.
The WFS is ment to be used by two sorts of final consumers: a developer desiring to make use of the data provided by the weather API on his/her own system; and a user of the WebApp simply hoping to know the weather prediction for a given place at a given time.

1.1. Architecture and Services

As it has been previously mentioned, the solution was developed in Java. The Spring Boot framework (2) was the chosen tool as it offers good features to develop micro services. Maven served as the build automation tool for the project (3), where the following dependencies were described (4) (5) (6) (7):

- JUnit 5 for unit testing (with AssertJ and EasyMock for readable assertions and mocks creation)
- Selenium for integration tests between internal API and WebApp

The Maven project is organized by packages and resources, where inside the resources are present all files of the WebApp and inside the main package there is the executable class that starts the system and 3 subpackages for the services provided, the REST controllers and the memory cache. These components are present in figure 1.

Starting from the bottom components, the External Service (ES) is the one responsible for the requests made to an external weather API called DarkSky. In truth, this API makes available all the data needed for the WFS and so, the internal API is a sort of limited specialization of DarkSky and offers no aditional value. Its purpose is solely educational and aims to demonstrate how the consumption of exeternal data must be put to practise in a test-driven development scenario.
Nevertheless the ES serves the Weather Service (WS), if the Memory Cache (MC) does not have the desired information. All new information retrieved from DarkSky by the WS is stored in cache to potentially reduce the number of requests made to the external source.
The Weather Controller (WC) is the component responsible for mapping all requests made to the internal API to the WS's functionalities. It serves as an interface between service and clients.
The WebApp is described in chapter 3. All java classes have detailed comments and javadoc written to better understanding what are their attributes and what to their methods do.

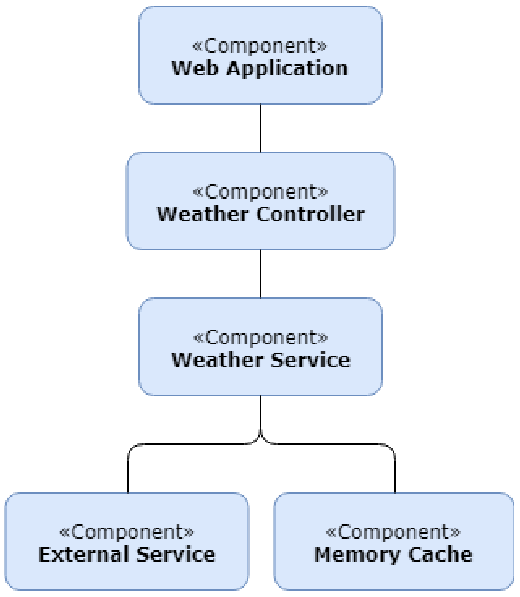


Figure 1: WFS' deployment diagram.

1.2. Unit and Integration Tests

As TDD starts with designing and developing tests for every small functionality of an application – where each will specify and validate what the code must do –, a few classes were developed in order to ensure the correct development of the components previously mentioned. The development of these components followed a cicle of writting tests, writting code that passes the tests and refining the solution. The list of testing classes developed, along with some useful code are in the end of this section. The tests applied belong to different levels of testing, illustrated in figure 2, and were a total of 24. In this section the focus is on the unit and integration tests.

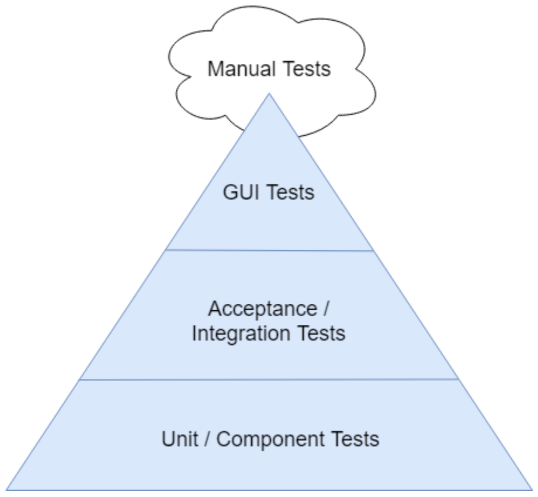


Figure 2: Testing Layers present in the development of the WFS.

The LocalCache class is the one responsible for storing in memory the requests made to the Weather API; it contains an internal class called CachedObject that basically works like a wrapper around the object to be cached including a Long value holding the last time that object was accessed in milliseconds; it is a generic class, meaning that it can store any type of object and have any type of key – this allowed the tests to remain simple while offering the chance to reuse the class in other scenarios without refactoring. The tests done to the LocalCache targeted its methods and covered the possible uses of each one:

testPutPrimitives()	testClear()
testPutNonPrimitives()	testSizeEmpty()
testGet()	testSizeNonEmpty()
testGetAll()	testContainsKeyTrue()
testRemove()	testContainsKeyFalse()
testRemoveEmpty()	

The ExternalService is the last of the test classes whose methods did not depend on any other class. As it contacted directly with the DarkSky API, the tests applied to its methods aimed to cover all possible requests made by the system to the external API. As these took only two formats, the classes needed only two tests:

testGetWeatherForecastNow()	testGetWeatherForecastPastPeriod()

The WeatherService, however, was the first of the remaining classes whose dependencies on others made it necessary to inject mock objects that would allow the isolation of this class when tested. As previously mentioned, the WS is served by both the LocalCache and the ExternalService to provide data to the WeatherController. With this in mind, two mock objects were used in the WS class. The code presented next refers to the configurations needed to prepare these mock objects.

```
@ExtendWith(MockitoExtension.class)
@MockitoSettings(strictness = Strictness.LENIENT)
public class WeatherServiceTest {

	@TestConfiguration
	static class WeatherServiceTestContextConfiguration {
		@Bean
		public WeatherService weatherService() {
			return new WeatherService();
		}
	}

	@Autowired
	@InjectMocks
	WeatherService weatherService;

	@Mock
	ExternalService externalService;

	@Mock
	LocalCache localCache;

	// (...)
}
```

All test methods used the localCache variable, since the service verifies if a new request is in cache before making a new request to the external service. The externalService variable was used only on the get methods to simulate the behaviour of the ES:

```
testGetNow()            testGetAllEmpty()
testGetRecent()         testGetAllNonEmpty()
testGetPeriod()
```

Depending on the WeatherService, the WeatherController needed to be tested with an abstraction as well. To do so, the test class required the creation of a new mock object of the WS. Since the WC works as the interface between API clients and the back-end, the tested methods were, as the reader can imagine, those who mapped the API's HTTP paths:

```
testGetWeatherNow()               testGetWeatherPeriod()
testGetWeatherRecent()            testGetWeatherPeriodBadInput()
testGetWeatherRecentBadInput()    testGetCache()
```

And so the fleet of component and acceptance tests was ready to be put to practice. The quality and coverage of these tests was submitted to an analysis and improvement during the development of the project, as you will see in the next section. However, the basis remained pretty much the same and offered a good guidance for the proper development of the Weather Forecast System. Tests on the User Interface are explained in chapter 3. The manual tests are not discussed in this report, as they are part (or should be) of any developer's work routine and were present throughout the project in various forms.

1.3. Static Code Analysis

The static code analysis was only introduced to the project once a working solution had been accomplished with all tests passing with success. This form of automatically examining the source code before the program is run allowed the detection of system flaws and bad practises.
SonarQube was the designated tool for the job, bringing the advantage of offering help on understanding how much coverage did the created tests have on the solution (8). The first analysis resulted in: 20 code smells, 4 bugs and 2 vulnerabilities. The problems were mainly inefficient code, repetitions, absence of exception dealing, etc. and were solved in proper time.

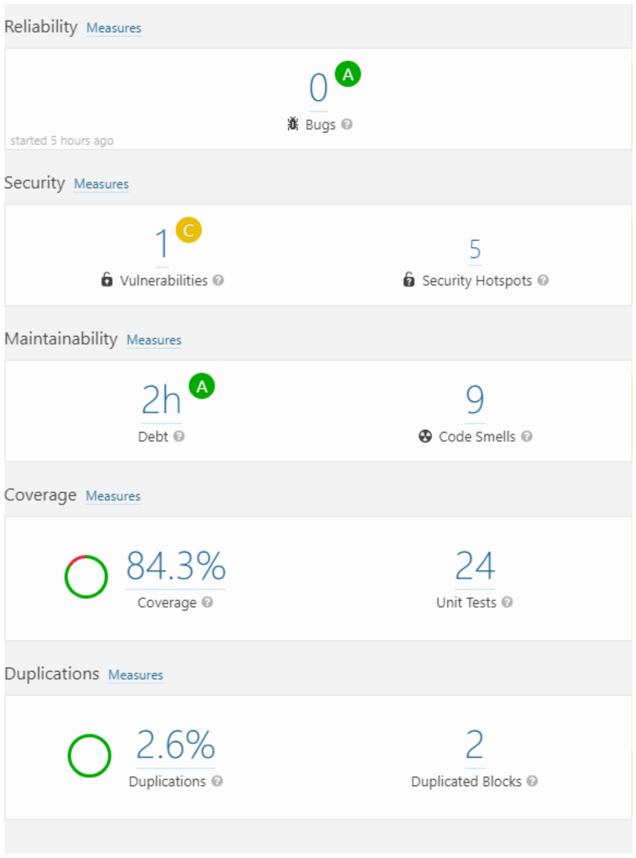


Figure 3: SonarQube Quality Gate final results.

Figure 3 shows the final evaluation of SonarQube to the project. The remaining vulnerability reffers to the direct use of a URL received by the ExternalService – the tool warns that leaving the client with the power to define what URL to give could potentially compromise the system integrity. However, as the only way to access the methods in question is through the WeatherController and then the WeatherService, these two middle classes offer the security that the tool does not notice.
The security hotspots, on the other hand, refer to the safe use of the command line arguments in the system's main class and of the HTTP endpoints on the WC – which, for the purpose of this project, seemed to have little relevance.
Achieving a coverage of 84% was a fairly satisfactory, as it reassured the worthiness of the efforts made around the tests development. A too higher value could also meen bad news, as it could be the result of an unbalanced effort distribution between tests development and the actual system development.

2. REST API

The REST API was one of the requirements of the task and has a key role on the solution. In this chapter I discuss how the weather forecast data is retreived, how is the weather API organized, what does it offer and how does it take advantage of the memory cache.

2.1. External API

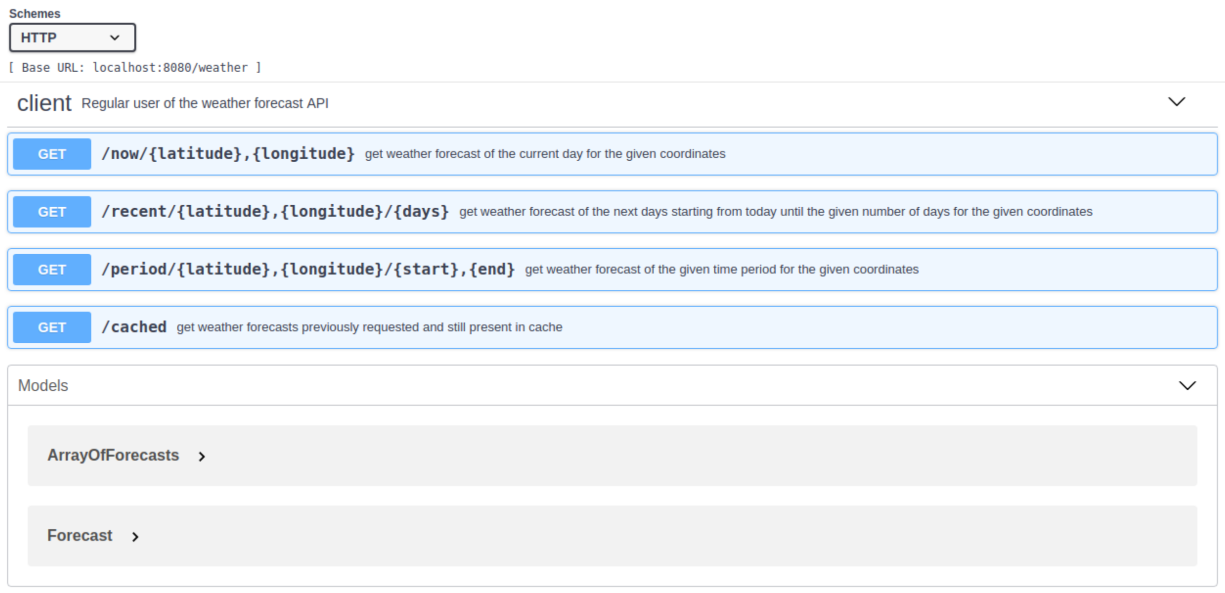
DarkSky was already mentioned in this report as the data source, and indeed it is the chosen API for all the forecasts provided by the WFS. After analysing what it offers, it was clear that more than enough data could be retrieved without any costs associated and with several trust-worthy data sources.
An access key was provided by the creators after signing up on their website and was embeded in the system as a static final variable. All requests to the API required the use of this key. These requests followed the format:

https://api.darksky.net/forecast/{access_key}/{latitude},{longitude}[,{time}]

Here, the *access_key* is replaced by the key, the *latitude* and *longitude* by the coordinates of the target location for the desired forecast or weather history, and the *time* being an optional path variable representing the target time of the forecast (if in the near future) or weather history (if in the past) in milliseconds.
DarkSky's replies follow the application/json format and the only field of the response is the 'daily', where an array of JSON objects is returned (each object representing the forecast of a day). On the next page it is present a screenshot (figure 5) taken from the API documentation created with the Swagger tool (9) that shows all the fields extracted from the 'daily' field and supported by the Weather API.

2.2. Weather API

The WFS's REST API offers 4 HTTP paths, 3 for weather forecasts and historical data and 1 to retrieve the requests to the API stored in cache. Figure 4 shows the structure of these paths and a small description of what each returns. The path variables are similar to those used for the DarkSky API in terms of coordinates, but the days variable refers to the numbers of days in the future desired to see the prediction (with a maximum of 7 days ahead) nad the start and end variables are date strings in the format 'yyyy-MM-dd' used to delimit the target days for the prediction or historical data retrieval. The API requests follow a simplistic approach, avoiding the need for specific variables in request headers or bodies and the need of any access keys.



Schemes

HTTP

[Base URL: localhost:8080/weather]

client Regular user of the weather forecast API

GET /now/{latitude},{longitude} get weather forecast of the current day for the given coordinates

GET /recent/{latitude},{longitude}/{days} get weather forecast of the next days starting from today until the given number of days for the given coordinates

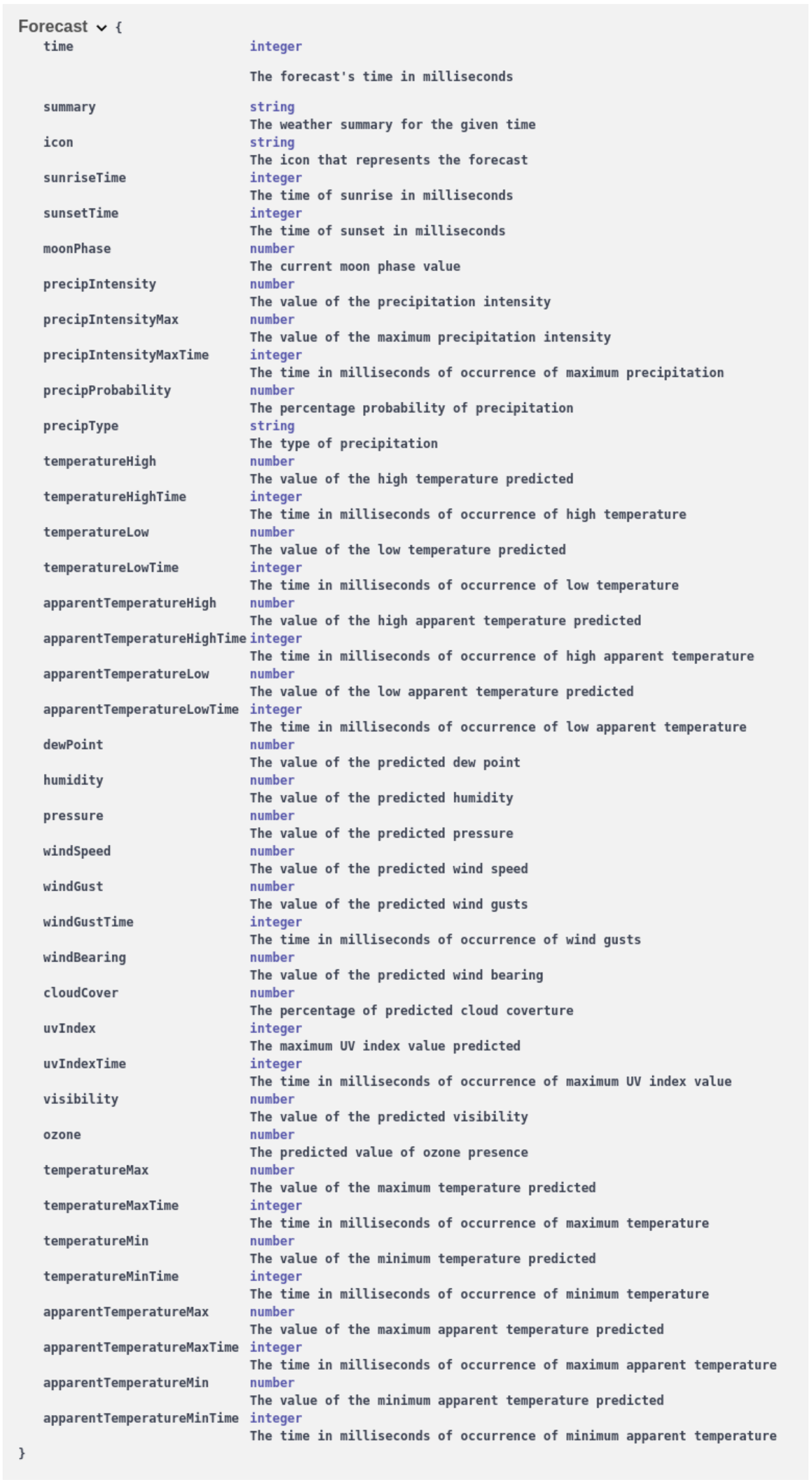
GET /period/{latitude},{longitude}/{start},{end} get weather forecast of the given time period for the given coordinates

GET /cached get weather forecasts previously requested and still present in cache

Models

ArrayOfForecasts

Forecast



```
Forecast ⌄ {
   time                        integer

                               The forecast's time in milliseconds

   summary                     string
                               The weather summary for the given time
   icon                        string
                               The icon that represents the forecast
   sunriseTime                 integer
                               The time of sunrise in milliseconds
   sunsetTime                  integer
                               The time of sunset in milliseconds
   moonPhase                   number
                               The current moon phase value
   precipIntensity             number
                               The value of the precipitation intensity
   precipIntensityMax          number
                               The value of the maximum precipitation intensity
   precipIntensityMaxTime      integer
                               The time in milliseconds of occurrence of maximum precipitation
   precipProbability           number
                               The percentage probability of precipitation
   precipType                  string
                               The type of precipitation
   temperatureHigh             number
                               The value of the high temperature predicted
   temperatureHighTime         integer
                               The time in milliseconds of occurrence of high temperature
   temperatureLow              number
                               The value of the low temperature predicted
   temperatureLowTime          integer
                               The time in milliseconds of occurrence of low temperature
   apparentTemperatureHigh     number
                               The value of the high apparent temperature predicted
   apparentTemperatureHighTime integer
                               The time in milliseconds of occurrence of high apparent temperature
   apparentTemperatureLow      number
                               The value of the low apparent temperature predicted
   apparentTemperatureLowTime  integer
                               The time in milliseconds of occurrence of low apparent temperature
   dewPoint                    number
                               The value of the predicted dew point
   humidity                    number
                               The value of the predicted humidity
   pressure                    number
                               The value of the predicted pressure
   windSpeed                   number
                               The value of the predicted wind speed
   windGust                    number
                               The value of the predicted wind gusts
   windGustTime                integer
                               The time in milliseconds of occurrence of wind gusts
   windBearing                 number
                               The value of the predicted wind bearing
   cloudCover                  number
                               The percentage of predicted cloud coverture
   uvIndex                     integer
                               The maximum UV index value predicted
   uvIndexTime                 integer
                               The time in milliseconds of occurrence of maximum UV index value
   visibility                  number
                               The value of the predicted visibility
   ozone                       number
                               The predicted value of ozone presence
   temperatureMax              number
                               The value of the maximum temperature predicted
   temperatureMaxTime          integer
                               The time in milliseconds of occurrence of maximum temperature
   temperatureMin              number
                               The value of the minimum temperature predicted
   temperatureMinTime          integer
                               The time in milliseconds of occurrence of minimum temperature
   apparentTemperatureMax      number
                               The value of the maximum apparent temperature predicted
   apparentTemperatureMaxTime  integer
                               The time in milliseconds of occurrence of maximum apparent temperature
   apparentTemperatureMin      number
                               The value of the minimum apparent temperature predicted
   apparentTemperatureMinTime  integer
                               The time in milliseconds of occurrence of minimum apparent temperature
}
```

Figures 4 and 5: HTTP Paths of the Weather API and data parameters.

An example of a method of the WC is included in this section for the reader to understand the structure of the mapping methods. All methods in this class follow the same process of this one, with minor changes:

```
@GetMapping("/recent/{latitude},{longitude}/{days}")
public String getWeatherRecent(@PathVariable("latitude") double latitude, @PathVariable("longitude")
double longitude, @PathVariable("days") int days) {
        // validating path variables
        if(days < 1){ days = 1; }
        if(days > 7){ days = 7; }
         // using the weather service
        return weatherService.get(latitude + "," + longitude, "recent", new
Long[]{Long.valueOf(days)}).toString();
}
```

Once the request is passed to the WS, this service adopts the strategy of avoiding the use of the ES by always checking the memory cache (LC) for existing entries that correctly answer the pending request. If it is found, the response is then immediatly sent to the destination. If not, the request proceeds to the ES. In this example, the javadoc is made available for better contextualization.

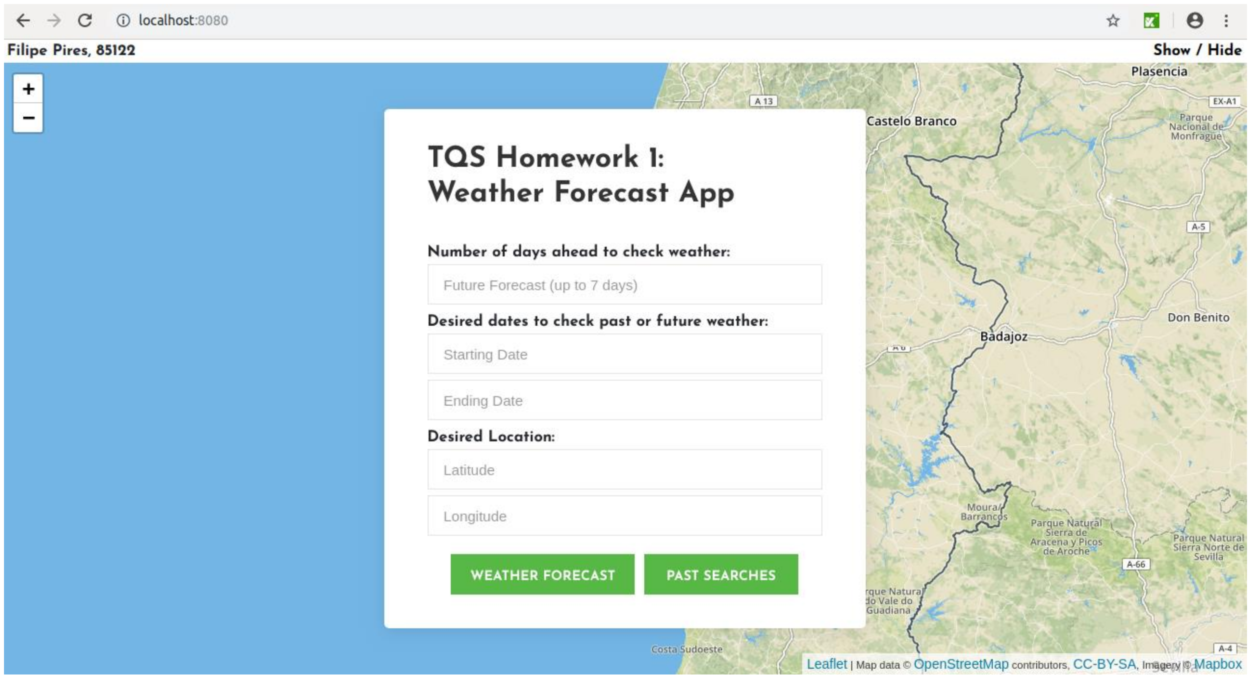
```
/**
 * Provides weather predictions for 3 types of requests:
 * - current day's forecast;
 * - next days' forecasts (with a maximum time distance of 7 days);
 * - daily forecasts of a given period of time (including past, present and future).
 *
 * @param coords String containing the coordinates of the location of the desired forecast, separated by ','
 * @param type type of the desired forecast ('now', 'recent' or 'period')
 * @param options parameters used in some types of requests
 * @return array of JSON objects containing the weather forecasts for the intended time and location
 */
public JsonArray get(String coords, String type, Long[] options) {
        // checking cache
        JsonArray cache = this.getAll(false); // [{"coords":[{day1},{day2}]},{"coords":[{day1}]},...]
        JsonObject entryObj;
        Object cachedObject;
        String key = coords + "," + type;
        if(type.equals("recent")) {
                key += "," + options[0];
        } else if(type.equals("period")) {
                key += "," + options[0] + "," + options[1];
        }
        for (JsonElement entry: cache) {
                entryObj = entry.getAsJsonObject();
                cachedObject = entryObj.get(key);
                if(cachedObject != null) {
                        return (this.localCache.get(key).getAsJsonObject()).get(key).getAsJsonArray();
                }
        }
        // proceed to the External service
        // (...)
}
```

3. Web Application

The WebApp, or the Front-End of the WFS, was developed with HTML5 and CSS, with all interactions controlled by Javascript and the help of JQuery. A template was used to make the application more appealing to the eye and an interactive world map from Leaflet was integrated with the background to make the user experience more confortable and intuitive. In this chapter I present the functionalities implemented of this application, along with the integration tests applied to the user interface.

3.1. Functionalities

The first feature implemented was a Search Form that allowed the user to try all 4 paths offered by the Weather API. Figure 6 shows the WebApp's front (and only) page with the form I speak of. All fields are optional, by default the application searches today's forecast (or the day that the user accesses the page) for Lisbon, Portugal.
If the user fills the 'Number of days ahead to check weather', and assuming that the cache is still empty, the request made to the Weather API is no longer to '/weather/now' but to 'weather/recent' and the results will refer to this week's weather forecast starting from today till the number of days typed.
If the 'Desired dates to check past or future weather' input fields are filled (through the selection of a datepicker, visible in figure 11), it is the request to '/weather/period' that is sent by the application.
The user can choose the target location in 2 ways: either by filling the coordinates in the input fields of 'Desired Location' (option with priority) or by picking the target in the background map itself (as seen in figures 7 and 8, where the form is hidden for better map navigation). There is also an option to increase or decrease the maps' size either by scrolling with the mouse on top of the map or by clicking the top left buttons.
All fields are protected against inputs considered bad. Still, even if a user is able to submit a form that seems to break a request, the API is prepared to deal with bad path variables and fix them in order to still return useful information.



TQS Homework 1:
Weather Forecast App

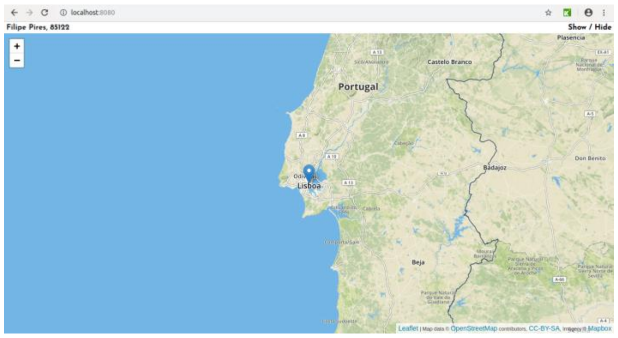
Number of days ahead to check weather:
Future Forecast (up to 7 days)

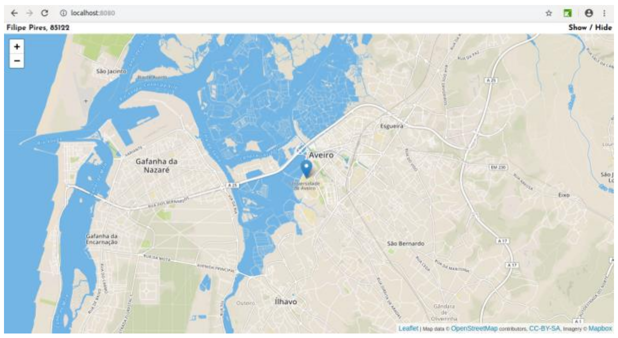
Desired dates to check past or future weather:
Starting Date
Ending Date

Desired Location:
Latitude
Longitude

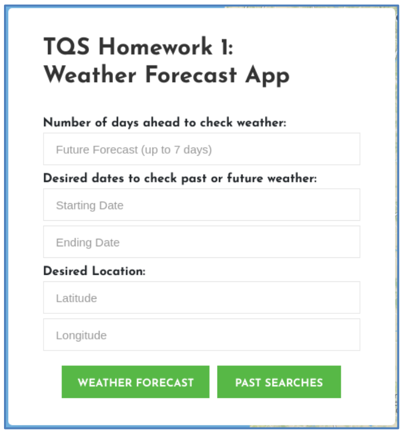
WEATHER FORECAST
PAST SEARCHES

Figure 6: WebApp's front page.





Figures 7 and 8: World map and location picking (Lisbon and Aveiro).



TQS Homework 1: Weather Forecast App

Number of days ahead to check weather:

Future Forecast (up to 7 days)

Desired dates to check past or future weather:

Starting Date

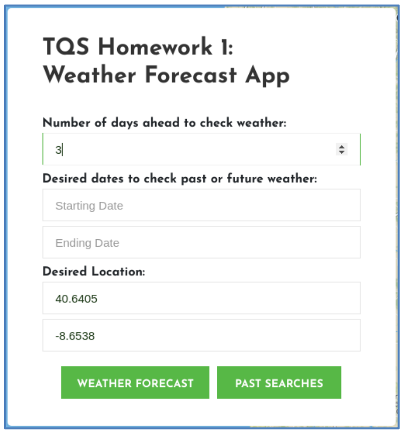
Ending Date

Desired Location:

Latitude

Longitude

WEATHER FORECAST | PAST SEARCHES



TQS Homework 1: Weather Forecast App

Number of days ahead to check weather:

3

Desired dates to check past or future weather:

Starting Date

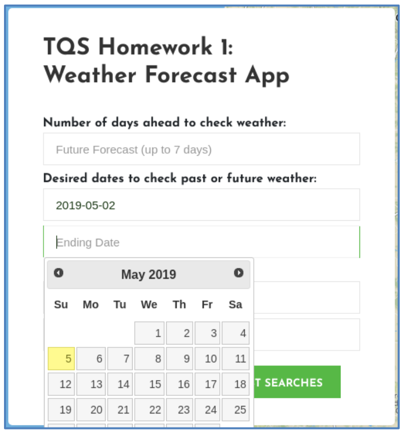
Ending Date

Desired Location:

40.6405

-8.6538

WEATHER FORECAST | PAST SEARCHES



TQS Homework 1: Weather Forecast App

Number of days ahead to check weather:

Future Forecast (up to 7 days)

Desired dates to check past or future weather:

2019-05-02

Ending Date

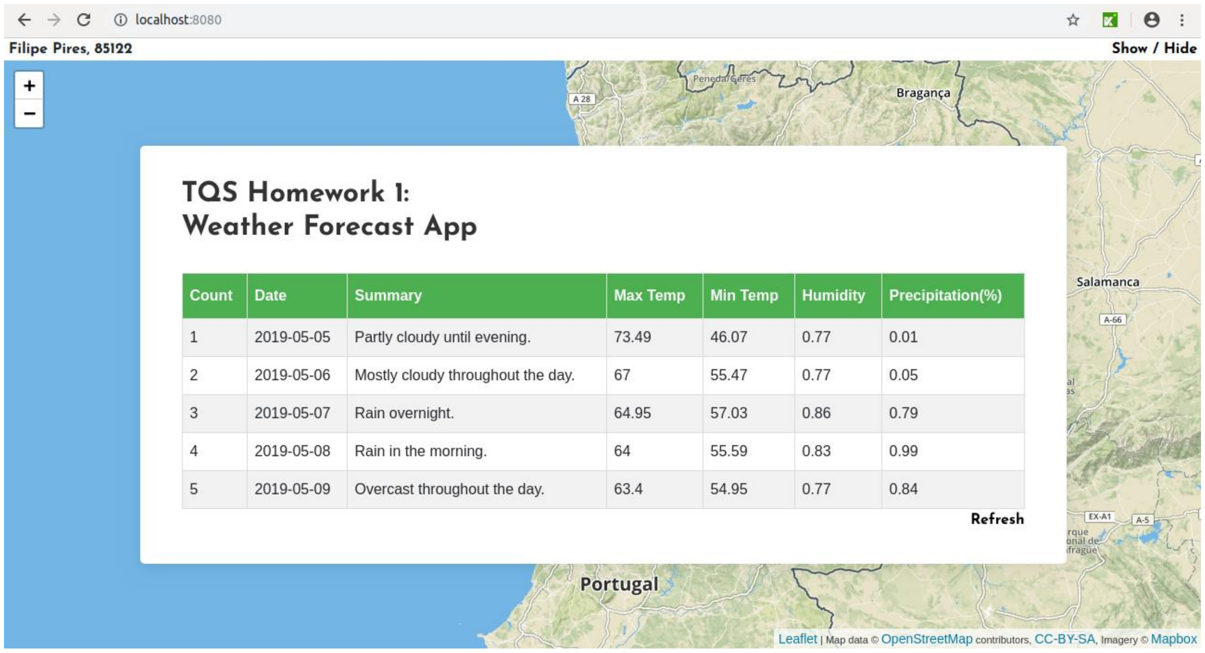
May 2019

Su	Mo	Tu	We	Th	Fr	Sa
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25

T SEARCHES

Figures 9, 10 and 11: Form input fields.

After clicking on the 'Weather Forecast' button, the form is submitted and a request is sent to the WeatherController. Once a response is received, a table is presented with only the most relevant values for all the requested days. If the request has already been made before and the response is still stored in cache, the table will load much faster as no request needs to be sent to the DarkSky API.


localhost:8080
Filipe Pires, 85122
Show / Hide
Bragança
Salamanca
Portugal
Leaflet | Map data © OpenStreetMap contributors, CC-BY-SA, Imagery © Mapbox

TQS Homework 1: Weather Forecast App

Count	Date	Summary	Max Temp	Min Temp	Humidity	Precipitation(%)
1	2019-05-05	Partly cloudy until evening.	73.49	46.07	0.77	0.01
2	2019-05-06	Mostly cloudy throughout the day.	67	55.47	0.77	0.05
3	2019-05-07	Rain overnight.	64.95	57.03	0.86	0.79
4	2019-05-08	Rain in the morning.	64	55.59	0.83	0.99
5	2019-05-09	Overcast throughout the day.	63.4	54.95	0.77	0.84

Refresh

Figures 12: Search results example.

3.2. GUI Testing

Only one class was developed to test the integration of the WebApp with the rest of the system. Selenium allowed the automation of the Google Chrome browser to test the application's features. The test class deploys a Spring Boot test (annotated with @SpringBootTest) in the default port 8080 and tests all features mentioned in the previous section, through 5 test methods and 3 auxiliary private methods.
These tests aim to test the correct functioning of the HTML elements of the application's only page, as well as their behaviors after the execution of specific actions. The test class uses a Chrome WebDriver and it even tests whether the map interacts correctly with the user or not.
The assertions are made with AssertJ fore more readable code and all verifications wait for the page (and other elements) to load before being executed. Input fields are filled through the command:

```
driver.findElement(By.id({element_id})).sendKeys({desired_input});
```

Although SonarQube did not evaluate the code coverage of these tests, taking in consideration the evaluations of the remaining and comparing them, it is believed that it rounds the 85% of coverage like the average.
Below is an example of the tests applied to the user interface. First it accesses the front page, waits for it to load, and then sends the default request and verifies the validity of the results:

```
@Test
public void testWeatherNowHere() throws Exception {
        System.out.println("weather now here");
        // arrange
        driver.get(baseURL);
        this.waitForPageLoad(driver);
        int timeout = 10;
        // act and assert
        driver.findElement(By.id("weather-forecast")).click();
        assertTrue(this.contentLoaded("table"));
        assertTrue(this.loadHttpContent(timeout,"tr-1"));
        assertThat(driver.findElement(By.id("refresh"))).isNotNull();
        driver.findElement(By.id("refresh")).click();
}
```

Conclusion

All the code developed for this assignment is to be delivered with this report. It contains javadoc written in all Java files, including the tests, to help understand each and every component. The API documentation written with the Swagger tool can also be provided, although not publicly available.
In the whole, the project presents itself as a minimalist solution of the proposed task with all requirements met and the basic, but crucial, tests passing successfuly. Also, the aditional feature proposed by the professor of allowing the variation of the number of days of a prediction was implemented to make the user experience a bit more interesting.
The development of the Weather Forecast System served as a valuable training exercise for the group project proposed for the same subject, leaving, as always, room for improvement. The efforts for future work will focus on the larger and more promissing group project, bringing to the table the confort learned from this home assignment.

Bibliography

1. **Oliveira, Ilídio.** HW1: Desenvolvimento de testes para uma aplicação multi-camada (front-end e serviços). *Elearning UA - TQS 2018/19.* [Online] 04 16, 2019. https://docs.google.com/document/d/1vspgs2x0dgvdV1XrsITZXNth2SFTxkAFfQLOa4dms6M/edit.

2. **Webb, Phillip, et al., et al.** Spring Boot Reference Guide. *Spring.io.* [Online] https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/.

3. Maven Documentation. *Apache Maven Project.* [Online] The Apache Software Foundation. https://maven.apache.org/guides/.

4. **Bechtold, Stefan, et al., et al.** JUnit 5 User Guide. *JUnit.org.* [Online] https://junit.org/junit5/docs/current/user-guide/.

5. AssertJ User Guide. *AssertJ.* [Online] https://assertj.github.io/doc/.

6. **Tremblay, Henri.** EasyMock User Guide. *EasyMock.org.* [Online] http://easymock.org/user-guide.html.

7. Selenium Documentation. *Selenium Project.* [Online] https://www.seleniumhq.org/docs/.

8. SonarQube Documentation. *SonarQube.org.* [Online] https://docs.sonarqube.org/latest/.

9. Swagger Documentation. *Swagger.io.* [Online] SmartBear. https://swagger.io/docs/.