universidade de aveiro
deti departamento de eletrónica, telecomunicações e informática

HW1: Mid-term assignment report

Diogo Gonçalves Silva [89348], v2020-04-15, https://gitlab.com/HerouFenix/aether

1 Introduction

1.1 Overview of the work

This project was elaborated as a solo effort in the context of the TQS (Testes e Qualidade de Serviço) class of the Engenharia Informática course at Universidade de Aveiro. The main affair of this work, going alongside with the class' structure and thematic guidelines, was the creation of a REST API service and its proper testing and integration within a CI pipeline. More specifically, it served as practice in the creation of an API service capable of gathering information about Air Quality and Pollutants, through the calling of external third-party APIs, and the supplying of endpoints that would allow a developer to interact with said backend service, alongside with the implementation of Unit and Integration tests to assure it's correct functioning. Besides this, it was also requested that we built a frontend application to interact with the built backend, as well as ellaborating the necessary Functional tests to verify it's expected behavior when in the hands of a possible consumer.

The built end-product, dubbed **Aether API**, permits a user to gather current, past or future information on the main air pollutants (CO, NO2, O3, PM10, PM25, SO2) as well as the overall Air Quality Index, either by coordinates (latitude and longitude), or by a preset locale. Said locales are preset locations (stored in a local H2 Database) that the user can create, giving them a name and corresponding coordinates. Besides the creation, the API also allows for the deletion and update of the created locales. All information is gathered from the external API's provided by BreezeOMeter and WeatherBit. The API also makes use of a local Cache that stores up to 150 calls for as long as 1 minute, allowing for the fast querying of previously searched for information. Besides this, we also offer a slick web page that allows you to easily interact with our API and test it's main features.

1.2 Limitations

Some limitations were experienced, especially when it came to the interactions with the third-party APIs. BreezeOMeter has the issue of only allowing free accounts to use their API for a period of 14 days, and has a daily query limit of 1000. On the other hand, the service doesn't offer results for some coordinates, and doesn't allow free users to get information from over 92 hours of difference compared to the current date. To counteract these measures we implemented interactions with the WeatherBit API, which gets called, either when specified, or when the call to BreezeOMeter fails. However, this API doesn't offer information on the concentration of pollutants and only allows for the search of air quality from up to 72 hours of difference from the current date. Given more time to work on this project, features like the dynamic swapping between BreezeOMeter keys (when one exceeds its daily limit or has expired) could have been implemented.

Other features that were planned but in the end omitted due to time constraints, complexity or complications with the external services, include the ability to include/exclude information on certain pollutants, and the querying for certain dates.

I'll also use this section to mention that, if the external API's tokens expire, these can be replaced in the **application.properties** file and on the **AetherServiceUT.java** test class (line 45) (the reason for the need to change the test is further expanded upon on section 3.1).

2 Product specification

2.1 Functional scope and supported interactions

Right from the start the **Aether API** was designed to be an easy and extensive form for users to get information on the quality of the air they're breathing. As such, the intended audience wasn't any specific demographic, but more so, the entire subset of the populous that, either by curiosity or medical necessity, tech-savvy or not, requires the quick and extensive visualization of the air's overall quality and pollutant concentration.

Figure 1. The frontend web page

With this in mind, in our frontend SPA (single page application) we offer the capability to **gather the air quality using a given set of coordinates**. Besides this, we also let our users **save a given set of coordinates under a name** in order for them to, in the future, get information for said locale without having to re-input it's coordinates. The user can also **delete or update the locale's** coordinates, in case they no longer need that set place saved, or want to alter what coordinates the name points to. We also permit the user to **specify**, or not, from which **API** they want the information to be gathered from, in case they present a preference, as well as getting the information for the **current** air quality, the **past history** (past 3 days in case of BreezeOMeter or last 72 hours in case of WeatherBit) or **future forecast** (following 3 days in case of BreezeOMeter or next 72 hours in case of WeatherBit).

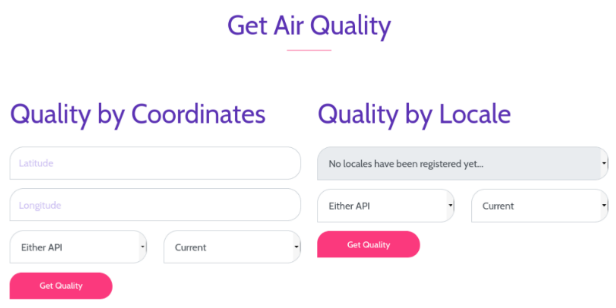
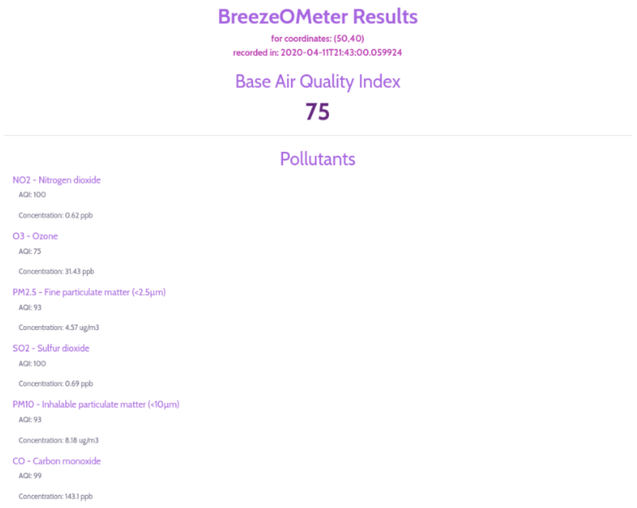


Figure 2. The UI elements that allow a user to get the air quality by specific parameters



BreezeOMeter Results
for coordinates: (50,40)
recorded in: 2020-04-11T21:43:00.059924

Base Air Quality Index
75

Pollutants

NO2 - Nitrogen dioxide
AQI: 100
Concentration: 0.62 ppb

O3 - Ozone
AQI: 75
Concentration: 31.43 ppb

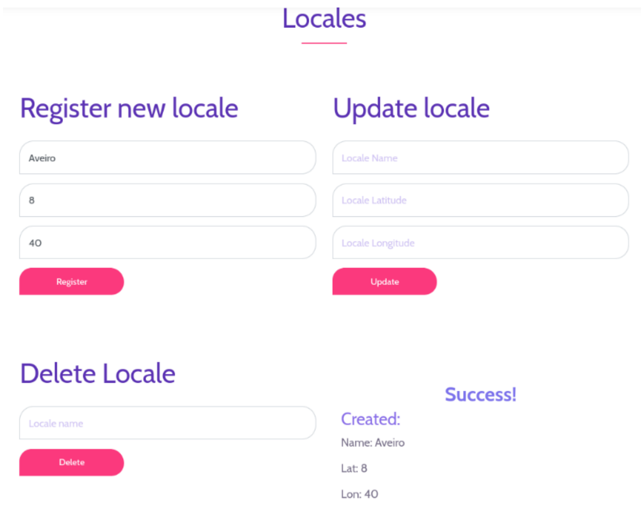
PM2.5 - Fine particulate matter (<2.5µm)
AQI: 93
Concentration: 4.57 ug/m3

SO2 - Sulfur dioxide
AQI: 100
Concentration: 0.69 ppb

PM10 - Inhalable particulate matter (<10µm)
AQI: 93
Concentration: 8.18 ug/m3

CO - Carbon monoxide
AQI: 99
Concentration: 143.1 ppb

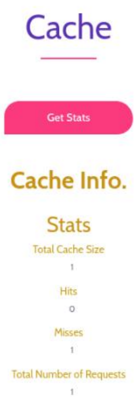
Figure 3. The UI element showing the result of an Air Quality Search



Success!
Created:
Name: Aveiro
Lat: 8
Lon: 40

Figure 4. The UI element that allows for the creation, update and deletion of locales, alongside a success message after having created a new locale

Furthermore, it should be noted that our main objective was to create a robust API to be used by other developers who might want to integrate it into their own projects. As such, we found that it would be convenient to include the option to check the internal cache’s stats (Hits/Misses, total requests made and number of elements currently stored) in the frontend, in order to showcase to possible developers that there is a cache at play, and that it is, in fact, working.


Cache

Get Stats

Cache Info.

Stats

Total Cache Size
1
Hits
0
Misses
1
Total Number of Requests
1

Figure 5. The UI element showing the possibility to get the Cache's current stats at the click of a button

To summarize, the following are all of the possible interactions with our frontend interface:

- Search for Air Quality, present, past or future, by Coordinate
- Search for Air Quality, present, past or future, by preset Locale
- Specify used external API
- Save a set of coordinates under a name (Locales)
- Edit a saved Locale
- Delete a saved Locale
- Get Cache Stats (size, total requests made, hits/misses)

2.2 System architecture

The following image illustrates the service's internal architecture. As showcased, the backend was built with the Spring Boot framework, written in Java. The frontend was built with the usage of the Thymeleaf engine to boot start a simple single-page-application that uses both Bootstrap4 for aesthetics and jQuery to call our REST API

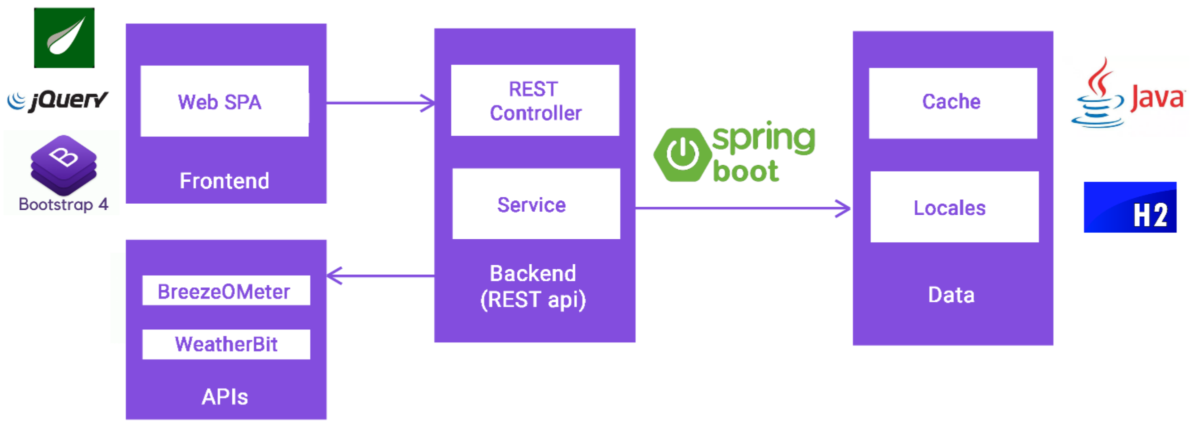


Figure 6. Aether API's base system architecture

For databases, the H2 in-memory database was utilized to store the Locales created by the users, and a simple HashMap was utilized to store the cache's information. As for the external APIs, and as aforementioned, we resourced to BreezeOMeter and WeatherBit to gather our information.

Below is a class diagram showcasing the backend classes and corresponding dependencies, methods and attributes. As can be observed, an MVC (Model-View-Controller) architecture was implemented, with the usage of the **AetherRestController.java** class as a controller for all of the possible API's endpoints. This class calls the **AetherService.java** class which contains the entirety of the business logic necessary for our backend. For facilitating the return of responses, a class called **AirResponse.java** was utilized. This class contains information on the target third-party API, the targeted coordinates, and the datetime at which the request was made. It also contains an internal hashmap that stores the information to be returned and transformed into the response's body. One or several **AirQuality.java** objects can be placed inside the response. This class is an interface that represents and should be implemented by all possible classes used to store the information gathered from the external API's. As of the time of writing this report, two APIs are used, hence, two objects implement this interface – **BreezeAirQuality.java**, that stores the information gotten from calling the BreezeOMeter API, and **WeatherAirQuality.java**, for the WeatherBit API. Furthermore, and due to the way the BreezeOMeter's response Json is structured, a **Pollutant.java** was created to store each of the pollutant's info. This class is stored in an hashmap, exclusively present in the **BreezeAirQuality.java** class. Besides these, a class called **Coordinates.java** was also made in order to represent a set of latitude and longitude.

When it comes to the locales, an entity class named **Locale.java**, containing an automatically generated ID, a name and a set of coordinates was created, alongside a repository class, **LocaleRepository.java**, for the permit the easy interaction with the entity (saving, deleting, updating and searching by name). Since, both for the creation and update of locales, it's required that a user passes a **Locale**-like object, a **LocalePOJO.java** class, that is in all ways equal to the **Locale.java** class, except for the ID and the @Entity annotation, was built. The suggestion to pass a POJO (plain old java object), rather than an entity to the controller was pushed by the SonarQube tool, discussed in a further section.

Finally the cache was implemented using two classes, a **CacheKey.java** class that's used as a unique identifier of each element in the cache and contains information on the coordinates, request type (past, current or future) and API of the item stored in the cache, and the **LocalCache.java** class that wraps a hashmap (with CacheKey objects as keys and AirQuality objects as values). This latter class also gives methods for the gathering of stats on the cache, such as the number of hits and misses (alongside logic to properly increment these variables), adding an item to the cache, checking if a given item is in the cache (given a set of coordinates, type of request and intended API). Furthermore this class also has a scheduled method that automatically removes the older item from the cache after a timeout is met (which, by default, is set to 1 minute). It should also be stated that the cache has a fixed size of, by default, 150 items, and that the order of insertion of items is recorded using a linked list queue. As a side note it should also be stated that, even though the remove method is called every minute, in the eventuality that the oldest item still isn't 1 minute old (due to it having been added midway through the timeout), it won't be removed, and that if, when getting an item from the cache, it's verified that it is over 1 minute old, then it'll automatically be removed from the cache and won't be returned.

To recapitulate, and offer a more in depth of each classes inner workings, following are all the classes and their purpose:

- **AetherRestController**
    - Our REST controller, contains all of our endpoints, calls our service class to process the received request and handles possible errors and exceptions.
    - Contains PUT, POST, GET and DELETE request mappings and usually responds, either with a ResponseEntity object or a ResponseStatusException, in case of errors. Regardless, when interacting with our API the responses are sent in the form of a JSON object.

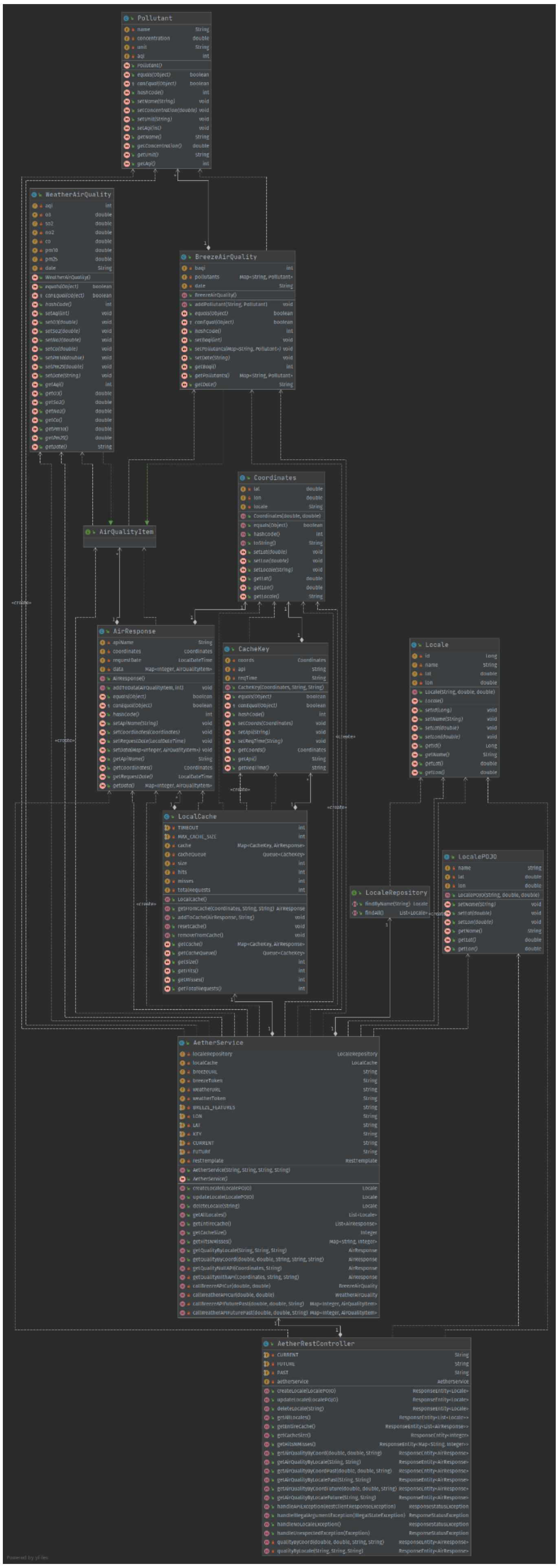


Figure 7. Aether API's backend classes

- **AetherService**
    - The class that implements all of our main business logic.
    - Is called by our REST controller and contains an instance of the LocaleRepository and LocalCache components.
    - Is tasked with properly processing requests received by the controller such as invoking the LocaleRepository to create a new Locale or getting the quality of the air for a certain set of coordinates.
    - Calls the APIs when needed. API base URLs and Token Keys variables get their values from the environment variables set in the **application.properties** file in order to more easily change these when need be. Despite this it also offers a constructor that accepts new values to assign to these variables.
    - When requested to get the air's quality, it first checks to see if the LocalCache already contains the response the request is looking for (by invoking the LocalCache's *getFromCache* method).
    - When it's requested to get the quality of air by a given locale it first queries the LocaleRepository's *findByName* method to try to find the locale in the H2 database. Afterwards if there indeed exists a locale, it extract it's coordinates and call its own *getByCoordinates* method.
    - When it needs to call an API it checks if an API was specified when requesting the controller. If so, and the item is not in the cache, it calls the appropriate API. When none is specified it first tries to get the information from the BreezeOMeter API and, if it returns an error, it then tries to get the same information from the WeatherBit API.
- **Locale**
    - Our only class marked with the @Entity annotation. Represents the Locale entities to be stored in our H2 database.
    - Contains a self generating long ID, the locale's name and it's set latitude and longitude.
- **LocaleRepository**
    - A class that serves as an interface to more easily interact with our H2 database. It implements the JPARepository interface.
    - Contains a method that allows us to find a Locale in our H2 database with a given name – *findByName*
- **LocalePOJO**
    - A "clone" class of our Locale class with the difference of being a POJO (plain old java object) rather than an entity.
    - When our REST controller receives a request to either create or update a Locale, the call should include in it's body a JSON mappable to a LocalePOJO object.
    - Originally the controller asked for a Locale rather than a LocalePOJO, but after putting our service through the SonarQube tool it was advised that we pass a POJO to our rest rather than an entity – "*Persistent entities should not be used as arguments of "@RequestMapping"*". For more information, please visit https://rules.sonarsource.com/java/tag/spring/RSPEC-4684 .
- **Coordinates**
    - A class that is used to represent a set of coordinates – Latitude and Longitude and, optionally, a Name.
    - Contains Lombok auto generated methods, as well as a custom Equals (and by correlation, HashCode) method that qualifies to Coordinates objects as equivalent if they possess the same latitude and longitude.

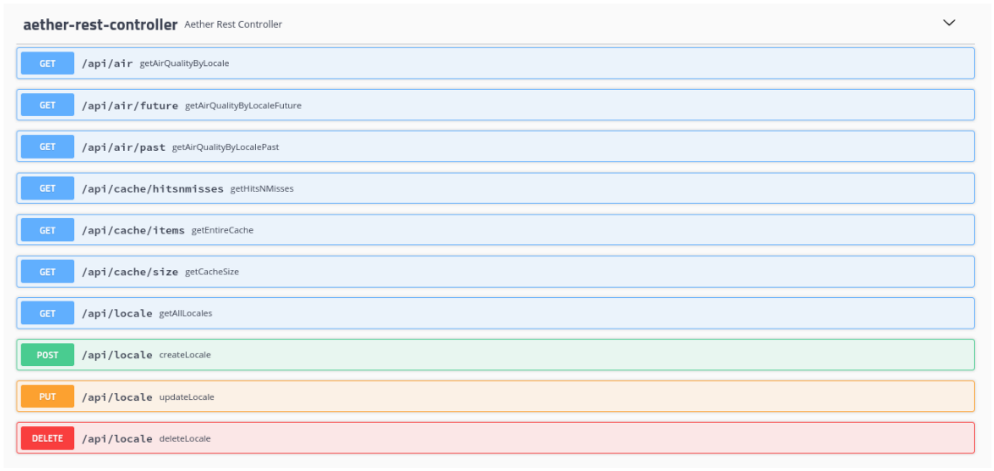
- **AirResponse**
  - This class serves as a wrapper for all responses that involve the quality of air. As such, it includes parameters such as the requested coordinates, the time at which the request was made, as well as whether the request wants the current, past or future data.
  - Also contains a HashMap of integer keys and AirQualityItem values. This is the map that will hold all of the external API's responses to our request.
  - This is one of the main entities that may be included in the ResponseEntity object returned by the REST controller, and as such, it's a JSON form of this object that is included in the reply to all of the get air requests.
- **AirQualityItem**
  - This is an interface that all of the classes that represent the external API's responses must implement in order to be able to be included in the AirResponse.
- **WeatherAirQuality**
  - This is the class the JSON response from the WeatherBit API is mapped to.
  - Contains the Air's Quality Index and the concentration for all main pollutants for a given set of coordinates.
- **BreezeAirQuality**
  - This is the class the JSON response from the BreezeOMeter API is mapped to.
  - Contains the Air's Quality Index and a HashMap of Pollutant objects (in which the pollutant's names are the keys) for a given set of coordinates.
- **Pollutant**
  - A class that represents a given pollutant's information gathered from BreezeOMeter's response. Created due to the fact that that API's response includes it's pollutant's information in the form of nested JSONS, hence it was natural to create a class to encompass each of them.
- **CacheKey**
  - This class is used to uniquely identify a given API response within our cache.
  - Contains the request's coordinates, specified API and type of request (past, current or future)
- **LocalCache**
  - Our custom cache that stores our created AirResponses and is called every time before resorting to the external APIs.
  - Stores up to 150 items (this value can be changed internally) in a HashMap of CacheKey keys and AirResponse values for up to one minute (this value can be changed internally).
  - Order of insertion is stored by using a LinkedList Queue.
  - Has a method to add a new item. In case the cache is full the oldest item is deleted to open up space for the new one.
  - Has a method to try to find an item, by providing the intended coordinates, type of request and API, within the cache. If no API is specified, it tries to find, first if a response using BreezeOMeter is registered, and if there isn't, if there is one using WeatherBit. If no matching items are found, a *null* is returned. If an item is found but it's older than the intended timeout (which by default is 1 minute), it removes the item from the cache and returns a null. Otherwise the item is returned.
  - Has a method annotated with @Scheduled which repeats every minute (by default) and removes the oldest item in the cache, if any. Note that the item is only removed in case it's older than the timeout. This is done to prevent the situation where an item is added just before the scheduled method is called, causing it to be removed before it

technically expires. This is also the reason as to why, when getting an item from the cache, the same verification is proceeded.
  - Contains statistics such as the cache's total requests, the hits (successful data retrieval), misses (unsuccessful data retrieval) and total number of items registered, alongside with the logic needed to calculate these variables.
  - Also contains a method to reset itself (i.e the stats, queue and hashmap).

To start our service, simply navigate to the **AetherApplication.java** class (within src > main > java > tqs > air > aether) and start the project from there. For the frontend, simply navigate to localhost:8080.

2.3 API for developers

A developer using our API has access to a plethora of methods which are documented and can be found in the generated swagger, by starting the Spring application and navigating to localhost:8080/swagger-ui.html. Unfortunately, a slight bug with SpringFox, the tool used to automatically generate a swagger page when running our Spring Application, has a slight, unavoidable bug and doesn't detect when there are two endpoints with the same URL, but different parameters that call different methods. As such the following image shows only most endpoints.



aether-rest-controller Aether Rest Controller

GET /api/air getAirQualityByLocale
GET /api/air/future getAirQualityByLocaleFuture
GET /api/air/past getAirQualityByLocalePast
GET /api/cache/hitsnmisses getHitsNMisses
GET /api/cache/items getEntireCache
GET /api/cache/size getCacheSize
GET /api/locale getAllLocales
POST /api/locale createLocale
PUT /api/locale updateLocale
DELETE /api/locale deleteLocale

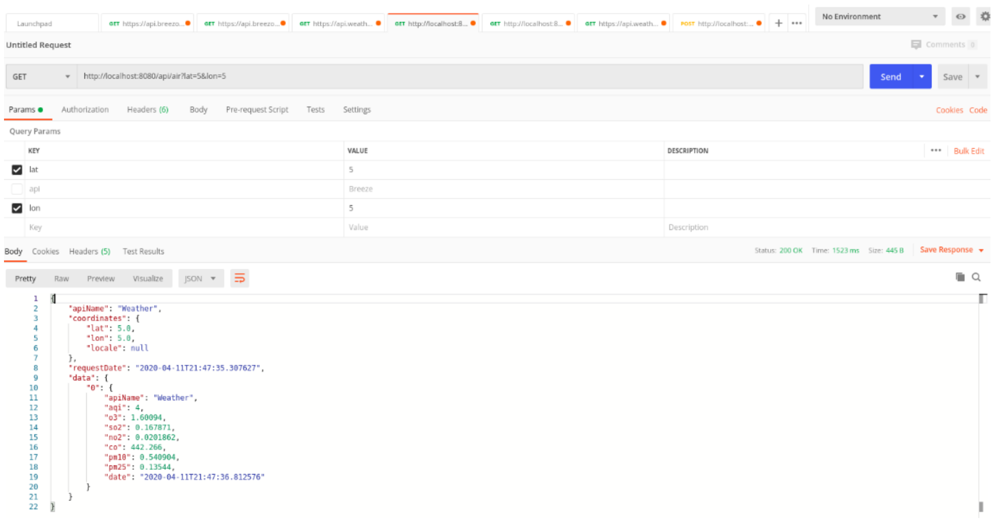
Figure 8. Aether API's generated swagger page

Generally speaking, we provide GET endpoints to get the air quality, either current, past or future, by locale or by given coordinates and with or without specifying the pretended external API. We also created statistical GET endpoints to interact with our internal cache, allowing for the querying of the cache's number of elements registered, hits/misses/total requests and to return all of the AirResponses saved within it. Besides this there are also three endpoints dedicated to the creation (POST), update (PUT) and deletion (DELETE) of locales.

Following is a succinct explanation of all endpoints, their requested parameters/bodies and what is included in their replies.

- ***/api/air*** *(by coordinates)*
  - Used to get the current air quality at given coordinates
  - **Type**:
    - GET
  - **Parameters**:
    - lat (String)
    - lon (String)

        - api (String) (OPTIONAL)
    - **Response**:
        - AirResponse Object (converted to JSON)



GET http://localhost:8080/api/air?lat=5&lon=5

Query Params

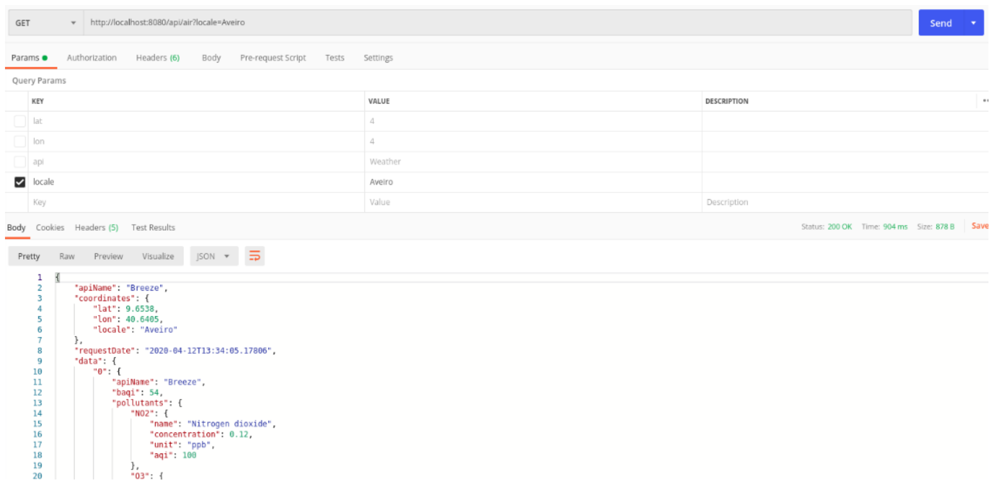
	KEY	VALUE	DESCRIPTION
✓	lat	5	
	api	Breeze	
✓	lon	5	
	Key	Value	Description

Status: 200 OK Time: 1523 ms Size: 445 B

```
{
    "apiName": "Weather",
    "coordinates": {
        "lat": 5.0,
        "lon": 5.0,
        "locale": null
    },
    "requestDate": "2020-04-11T21:47:35.307627",
    "data": {
        "0": {
            "apiName": "Weather",
            "aqi": 4,
            "o3": 1.60094,
            "so2": 0.167871,
            "no2": 0.0201862,
            "co": 442.266,
            "pm10": 0.540904,
            "pm25": 0.13544,
            "date": "2020-04-11T21:47:36.812576"
        }
    }
}
```

Figure 9. Example of usage of the api/air endpoint by coordinates

- ***/api/air** (by locale)*
    - Used to get the current air quality at given locale
    - **Type**:
        - GET
    - **Parameters**:
        - locale (String)
        - api (String) (OPTIONAL)
    - **Response**:
        - AirResponse Object (converted to JSON)



GET http://localhost:8080/api/air?locale=Aveiro

Query Params

	KEY	VALUE	DESCRIPTION
	lat	4	
	lon	4	
	api	Weather	
✓	locale	Aveiro	
	Key	Value	Description

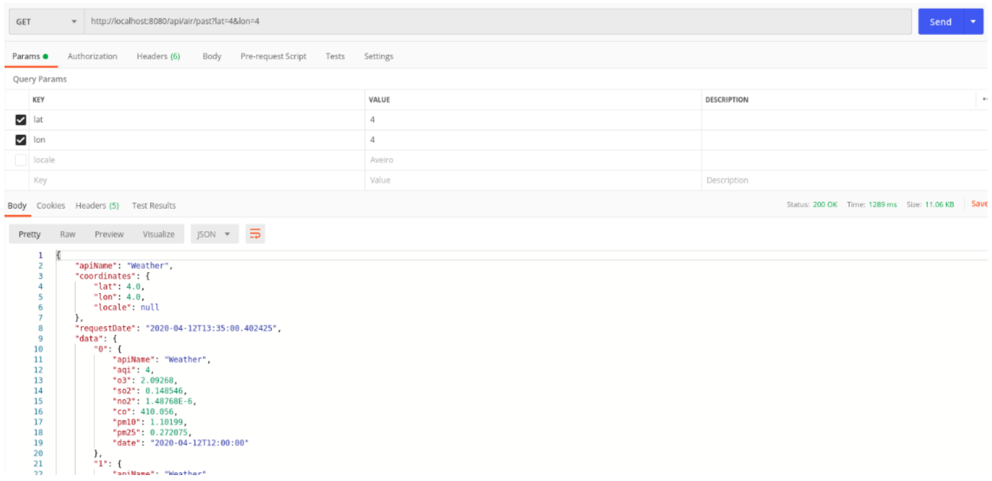
Status: 200 OK Time: 904 ms Size: 878 B

```
{
    "apiName": "Breeze",
    "coordinates": {
        "lat": 9.6538,
        "lon": 40.6405,
        "locale": "Aveiro"
    },
    "requestDate": "2020-04-12T13:34:05.17806",
    "data": {
        "0": {
            "apiName": "Breeze",
            "baqi": 54,
            "pollutants": {
                "NO2": {
                    "name": "Nitrogen dioxide",
                    "concentration": 0.12,
                    "unit": "ppb",
                    "aqi": 100
                },
                "O3": {
```

Figure 10. Example of usage of the api/air endpoint by locale

- ***/api/air/past** (by coordinates)*
    - Used to get the past air quality at given coordinates
    - **Type**:
        - GET

- **Parameters**:
  - lat (String)
  - lon (String)
  - api (String) (OPTIONAL)
- **Response**:
  - AirResponse Object (converted to JSON)



GET http://localhost:8080/api/air/past?lat=4&lon=4 Send

Query Params

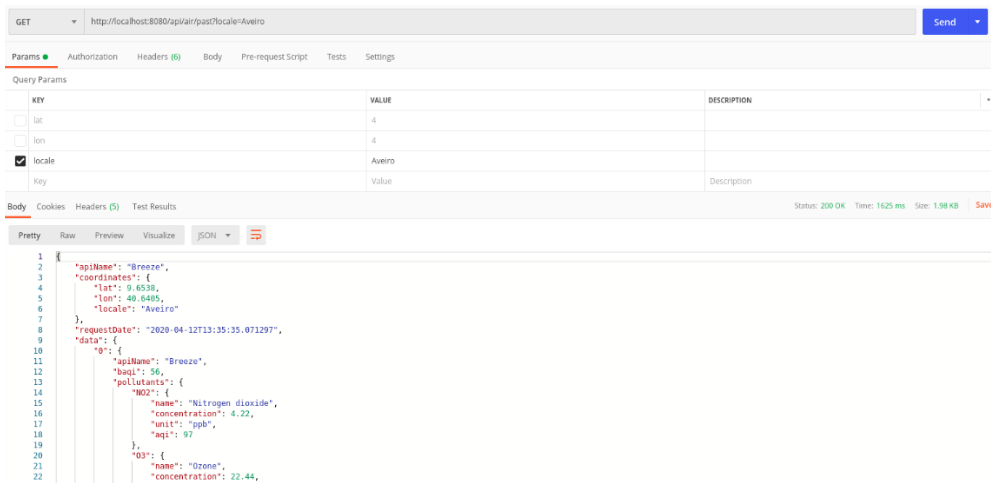
	KEY	VALUE	DESCRIPTION
✓	lat	4	
✓	lon	4	
	locale	Aveiro	
	Key	Value	Description

Status: 200 OK Time: 1289 ms Size: 11.06 KB

```
{
    "apiName": "Weather",
    "coordinates": {
        "lat": 4.0,
        "lon": 4.0,
        "locale": null
    },
    "requestDate": "2020-04-12T13:35:00.402425",
    "data": {
        "0": {
            "apiName": "Weather",
            "aqi": 4,
            "o3": 2.09268,
            "so2": 0.148546,
            "no2": 1.48768E-6,
            "co": 410.056,
            "pm10": 1.10199,
            "pm25": 0.272075,
            "date": "2020-04-12T12:00:00"
        },
        "1": {
            "apiName": "Weather"
```

Figure 11. Example of usage of the api/air/past endpoint by coordinates

- **/api/air/past** *(by locale)*
  - Used to get the past air quality at given locale
  - **Type**:
    - GET
  - **Parameters**:
    - locale (String)
    - api (String) (OPTIONAL)
  - **Response**:
    - AirResponse Object (converted to JSON)



GET http://localhost:8080/api/air/past?locale=Aveiro Send

Query Params

	KEY	VALUE	DESCRIPTION
	lat	4	
	lon	4	
✓	locale	Aveiro	
	Key	Value	Description

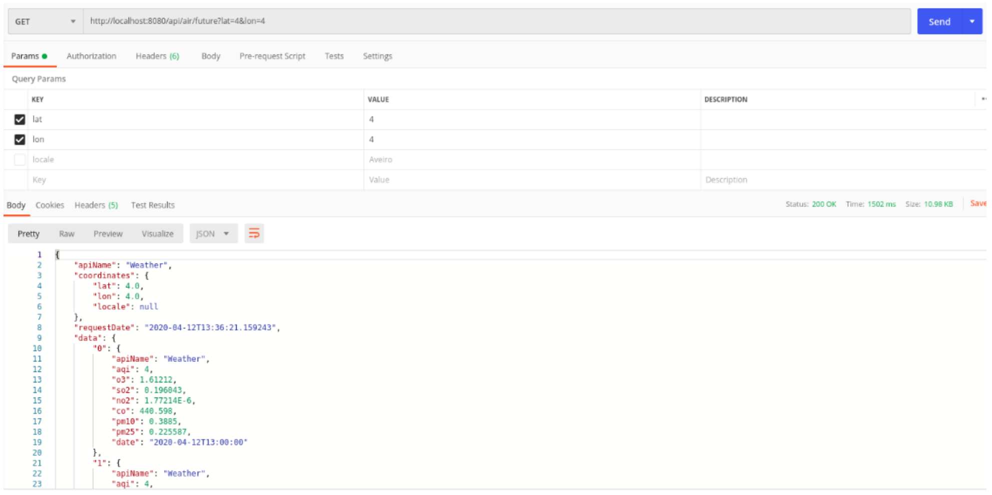
Status: 200 OK Time: 1625 ms Size: 1.98 KB

```
{
    "apiName": "Breeze",
    "coordinates": {
        "lat": 9.6538,
        "lon": 40.6405,
        "locale": "Aveiro"
    },
    "requestDate": "2020-04-12T13:35:35.071297",
    "data": {
        "0": {
            "apiName": "Breeze",
            "baqi": 56,
            "pollutants": {
                "NO2": {
                    "name": "Nitrogen dioxide",
                    "concentration": 4.22,
                    "unit": "ppb",
                    "aqi": 97
                },
                "O3": {
                    "name": "Ozone",
                    "concentration": 22.44,
```

Figure 12. Example of usage of the api/air/past endpoint by locale

- **/api/air/future** *(by coordinates)*
  - Used to get the future air quality at given coordinates

- **Type**:
    - GET
- **Parameters**:
    - lat (String)
    - lon (String)
    - api (String) (OPTIONAL)
- **Response**:
    - AirResponse Object (converted to JSON)



http://localhost:8080/api/air/future?lat=4&lon=4

Query Params

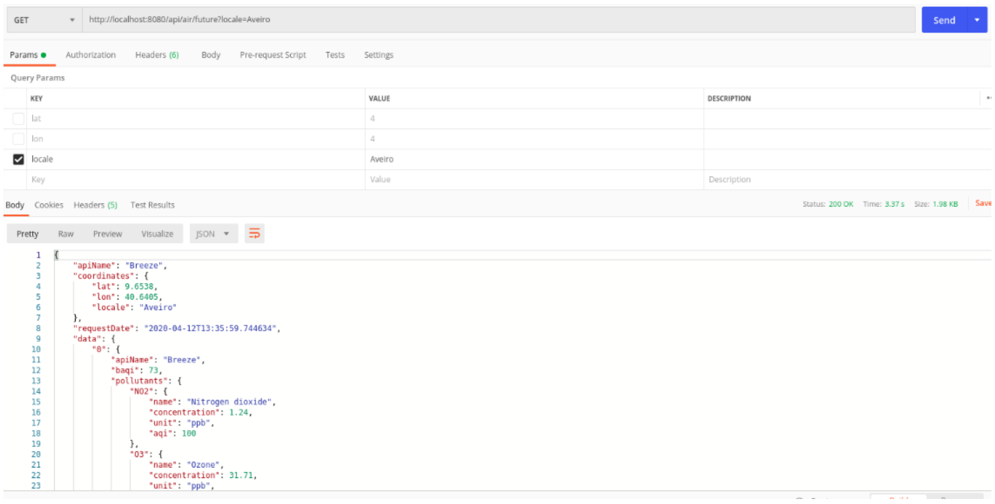
	KEY	VALUE	DESCRIPTION
✓	lat	4	
✓	lon	4	
	locale	Aveiro	
	Key	Value	Description

Status: 200 OK Time: 1502 ms Size: 10.98 KB

```
{
    "apiName": "Weather",
    "coordinates": {
        "lat": 4.0,
        "lon": 4.0,
        "locale": null
    },
    "requestDate": "2020-04-12T13:36:21.159243",
    "data": {
        "0": {
            "apiName": "Weather",
            "aqi": 4,
            "o3": 1.61212,
            "so2": 0.196043,
            "no2": 1.77214E-6,
            "co": 440.598,
            "pm10": 0.3885,
            "pm25": 0.225587,
            "date": "2020-04-12T13:00:00"
        },
        "1": {
            "apiName": "Weather",
            "aqi": 4,
```

Figure 13. Example of usage of the api/air/future endpoint by coordinates

- **/api/air/future** *(by locale)*
    - Used to get the past future quality at given locale
    - **Type**:
        - GET
    - **Parameters**:
        - locale (String)
        - api (String) (OPTIONAL)
    - **Response**:
        - AirResponse Object (converted to JSON)



http://localhost:8080/api/air/future?locale=Aveiro

Query Params

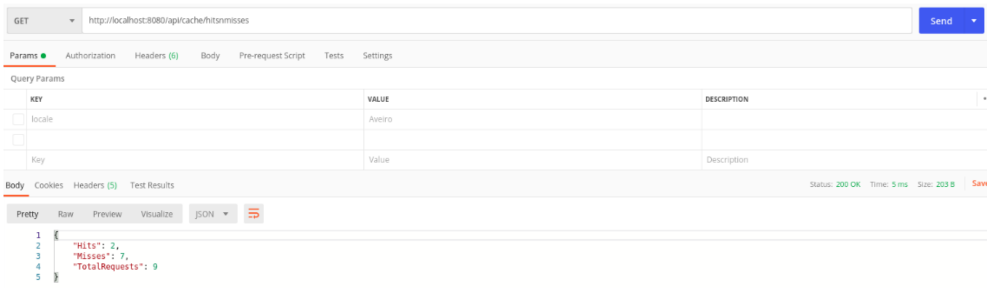
	KEY	VALUE	DESCRIPTION
	lat	4	
	lon	4	
✓	locale	Aveiro	
	Key	Value	Description

Status: 200 OK Time: 3.37 s Size: 1.98 KB

```
{
    "apiName": "Breeze",
    "coordinates": {
        "lat": 9.6538,
        "lon": 40.6405,
        "locale": "Aveiro"
    },
    "requestDate": "2020-04-12T13:35:59.744634",
    "data": {
        "0": {
            "apiName": "Breeze",
            "baqi": 73,
            "pollutants": {
                "NO2": {
                    "name": "Nitrogen dioxide",
                    "concentration": 1.24,
                    "unit": "ppb",
                    "aqi": 100
                },
                "O3": {
                    "name": "Ozone",
                    "concentration": 31.71,
                    "unit": "ppb",
```

Figure 14. Example of usage of the api/air/past endpoint by locale

- **/api/cache/hitsnmisses**
  - Used to get the current cache's hits, misses and total number of requests made to it
  - **Type**:
    - GET
  - **Response**:
    - {"Hits" : int, "Misses" : int, "TotalRequests" : int}



GET http://localhost:8080/api/cache/hitsnmisses Send

Params ● Authorization Headers (6) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE	DESCRIPTION
locale	Aveiro	
Key	Value	Description

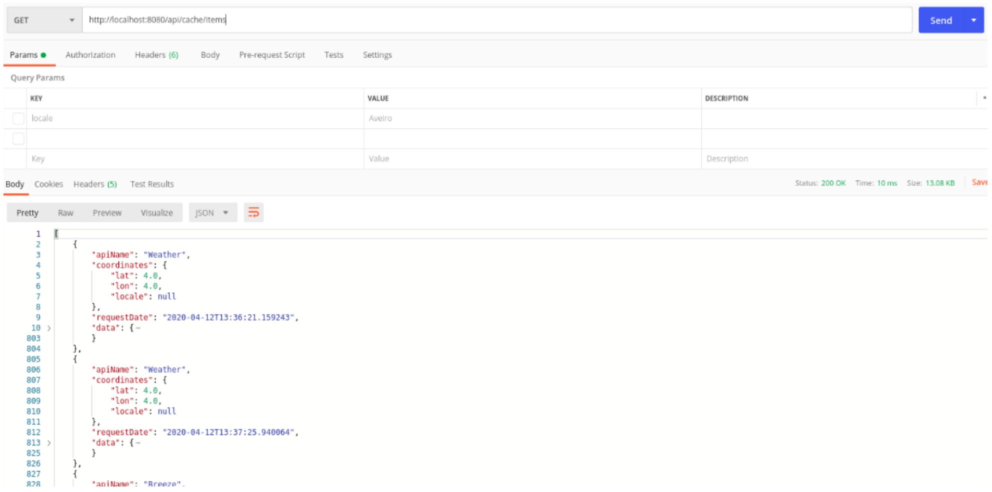
Body Cookies Headers (5) Test Results — Status: 200 OK Time: 5 ms Size: 203 B

Pretty Raw Preview Visualize JSON

```
{
    "Hits": 2,
    "Misses": 7,
    "TotalRequests": 9
}
```

Figure 15. Example of usage of the api/cache/hitsnmisses endpoint

- **/api/cache/items**
  - Get all of the items currently stored within the cache
  - Type:
    - GET
  - Response:
    - Array of AirResponse Objects (converted to JSON)



GET http://localhost:8080/api/cache/items Send

Params ● Authorization Headers (6) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE	DESCRIPTION
locale	Aveiro	
Key	Value	Description

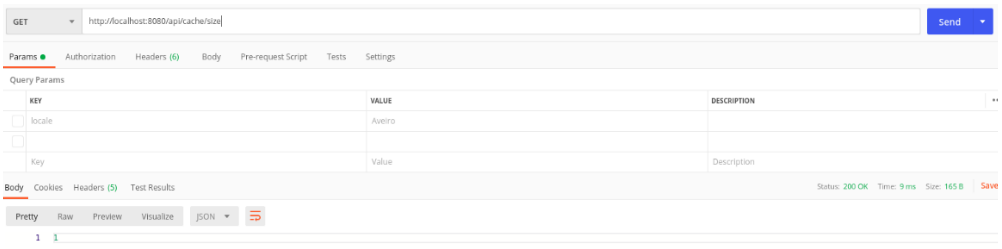
Body Cookies Headers (5) Test Results — Status: 200 OK Time: 10 ms Size: 13.08 KB

Pretty Raw Preview Visualize JSON

```
[
    {
        "apiName": "Weather",
        "coordinates": {
            "lat": 4.0,
            "lon": 4.0,
            "locale": null
        },
        "requestDate": "2020-04-12T13:36:21.159243",
        "data": {…
        }
    },
    {
        "apiName": "Weather",
        "coordinates": {
            "lat": 4.0,
            "lon": 4.0,
            "locale": null
        },
        "requestDate": "2020-04-12T13:37:25.940064",
        "data": {…
        }
    },
    {
        "apiName": "Breeze",
```

Figure 16. Example of usage of the api/cache/items endpoint

- **/api/cache/size**
  - Get how many items are currently stored within the cache
  - Type:
    - GET
  - Response:
    - Int



GET http://localhost:8080/api/cache/size

Query Params

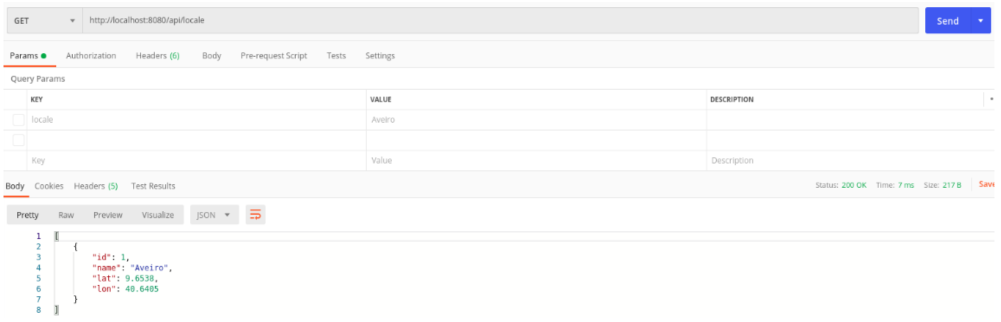
KEY	VALUE	DESCRIPTION
locale	Aveiro	

Status: 200 OK Time: 9 ms Size: 165 B

```
1
```

Figure 17. Example of usage of the api/cache/size endpoint

- **/api/locale**
  - Get a list with all of the locales in our database
  - Type:
    - GET
  - Response:
    - Array of Locale Objects (converted to JSON)



GET http://localhost:8080/api/locale

Query Params

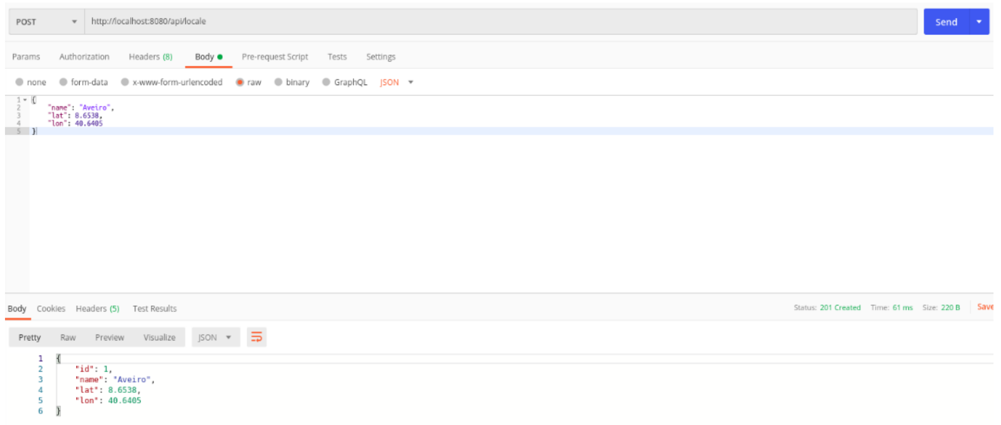
KEY	VALUE	DESCRIPTION
locale	Aveiro	

Status: 200 OK Time: 7 ms Size: 217 B

```
[
    {
        "id": 1,
        "name": "Aveiro",
        "lat": 9.6538,
        "lon": 40.6405
    }
]
```

Figure 18. Example of usage of the api/locale endpoint with GET

- **/api/locale**
  - Insert a new locale into our database
  - Type:
    - POST
  - Request Body:
    - {“name” : String, “lat” : double, “lon” : double}
  - Response:
    - The created Locale Object (converted to JSON)



POST http://localhost:8080/api/locale

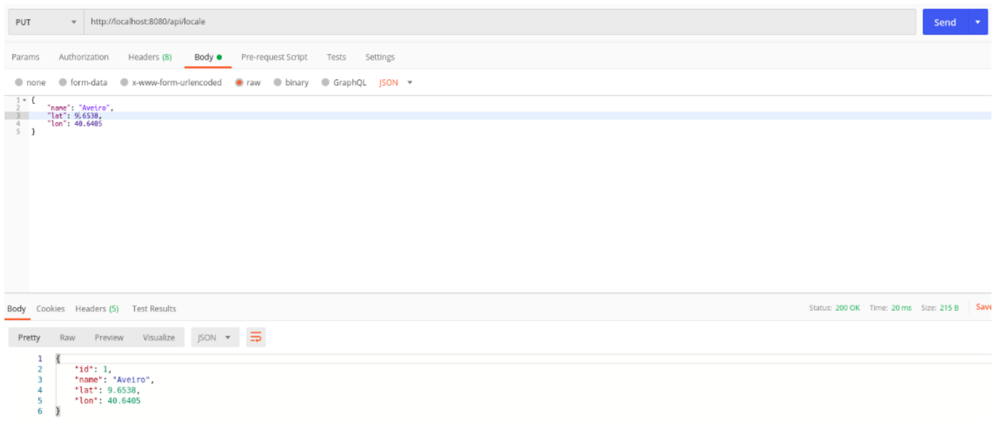
```
{
    "name": "Aveiro",
    "lat": 8.6538,
    "lon": 40.6405
}
```

Status: 201 Created Time: 61 ms Size: 220 B

```
{
    "id": 1,
    "name": "Aveiro",
    "lat": 8.6538,
    "lon": 40.6405
}
```

Figure 19. Example of usage of the api/locale endpoint with POST

- **/api/locale**
    - Update a given locale in our database
    - Type:
        - PUT
    - Request Body:
        - {“name” : String, “lat” : double, “lon” : double}
    - Response:
        - The updated Locale Object (converted to JSON)


PUT http://localhost:8080/api/locale

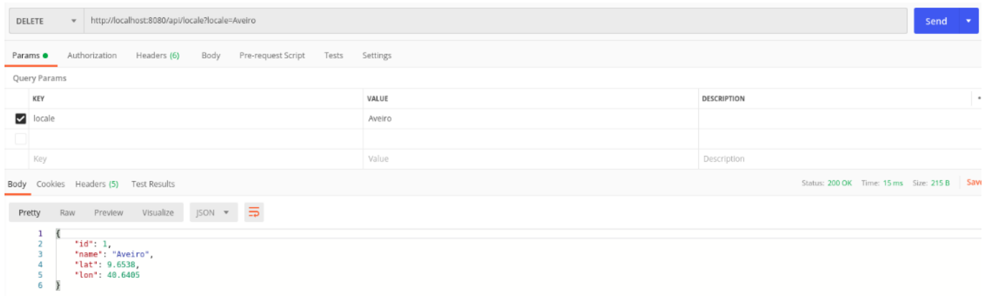
```
{
    "name": "Aveiro",
    "lat": 9.6538,
    "lon": 40.6405
}
```

Status: 200 OK Time: 20 ms Size: 215 B

```
{
    "id": 1,
    "name": "Aveiro",
    "lat": 9.6538,
    "lon": 40.6405
}
```

Figure 20. Example of usage of the api/locale endpoint with PUT

- **/api/locale**
    - Delete a given locale from our database
    - Type:
        - DELETE
    - Parameters:
        - locale (String)
    - Response:
        - The deleted Locale Object (converted to JSON)


DELETE http://localhost:8080/api/locale?locale=Aveiro

Query Params

	KEY	VALUE	DESCRIPTION
✓	locale	Aveiro	

Status: 200 OK Time: 15 ms Size: 215 B

```
{
    "id": 1,
    "name": "Aveiro",
    "lat": 9.6538,
    "lon": 40.6405
}
```

Figure 21. Example of usage of the api/locale endpoint with DELETE

3 Quality assurance

3.1 Overall strategy for testing

Right from the bat we decided to adopt a Test Driven Development (TDD) where the tests were written before actually implementing the methods. More specifically, the approach we took was to start off by writing down a list of features that we wanted to be included, choosing the external APIs (and properly testing and ascertaining their response bodies), then designing (i.e thinking about what methods and parameters would be necessary) the classes needed to accomplish what we wanted, followed by writing the tests and only then would we write the actual code to implement each method. Tests were written in a top-down fashion, as in, we started by creating tests for our controller, then the service, repository and all other support classes. The one exception to this rule was the frontend web-page which was only functionally tested after having been built. Funnily enough this proved to be an issue as we will explain on section 3.3. That being said, obviously during the process of development new functionalities and changes, that weren't initially thought of, were implemented, and of course, new tests were added and old ones adjusted as needed. All classes were thoroughly tested excepting those which had none or no meaningful methods, by which I mean automatically generated methods by Lombok.

The tools utilized consisted in **Junit 5** as a base testing framework, **Hamcrest** as a support library to make our test's code more readable and for their useful matchers and **Mockito** to create our mocks and simulate some of the classes' dependencies. All of these packages were included within the *springboot-starter-test* dependency which includes all of those libraries. Besides these we also utilized the **Sellenium Web Driver** to write functional tests on our frontend interface and **Awaitility** to test our local cache's scheduled method. For integration tests on our API we utilized **Spring Boot MockMvc** to simulate the interaction between a user and our REST API.

In total over 130 tests were written for our backend and more than 30 functional tests were implemented for our frontend. All backend tests were included in the implemented CI pipeline so that at every push to our master branch they would be ran, as we'll explore further in section 3.5.

One note that I would like to add is that some of the tests rely on certain responses from the external API's be given. As such, when the access tokens expire or the daily query limit for those tokens is exceeded (especially problematic for BreezeOMeter), some of the tests may fail. If this happens I kindly ask the reader to replace the necessary variables present in the **application.properties** file and the AetherService initializer line on the **AetherServiceUT.java** test class (line 45).

3.2 Unit and integration testing

We'll begin by discussing the Unit tests written for our main logic and support classes – AetherService.java, LocaleRepository.java, Coordinates.java and LocaleCache.java. Besides all of these, we also considered making tests for classes, such as BreezeAirResponse, which were causing our code coverage report to be lowered (more on this on section 3.4) but since in most cases these contained no methods, automatically generated methods, or simple add methods (as was the case with the aforementioned class which contained a single add method to put a new item in it's pollutants hashmap), we opted out of it since they wouldn't be meaningful enough tests, besides the fact that these classes represent domain entities rather than classes with deep logic behind them. The overall thought behind all tests performed was "test everything that might happen, and assure that when something goes wrong, we're prepared to answer". As such we didn't limit ourselves to creating a single test per method and call it a day, rather we thought about all logical paths our algorithms may take, both when returning successful or when throwing errors, and test what happens when such occasions happen. This ideology applies not only to the unit tests but to all created tests for the project. With that said, the following are all our Unit test classes, their purpose, what methods they include and some snippets of the type of tests written.

AetherServiceUT.java

- Unit test for our main business logic class, AetherService.java. This test mocks our service's dependencies – the LocaleRepository and the LocaleCache. It also initializes the service class (*aetherService*) with a given set of parameters, specifying the external API's request URLs and keys. This was done for two main reasons, the first being so that we could include tests for the situations when our API keys expire, initializing a specific service object for this purpose (*aetherServiceInvalidKeys*) and the other being because there was no intuitive way to fetch the environment variables, included in the *application.properties* file.

- It should be noted that, in the eventuality the API keys expire or the URLs are no longer available, some of the tests which expect certain successes or failures from the APIs (to test how our own service reacts) will fail. To counteract this please change the service initializer described on the last paragraph.

```
    @Mock(lenient = true)
    private LocaleRepository localeRepository;

    @Mock(lenient = true)
    private LocalCache localCache;

    @InjectMocks
    private AetherService aetherService = new AetherService("https://api.breezometer.com/air-
quality/v2", "a42e3e4b1d8a47ef86c39275523c6491", "https://api.weatherbit.io/v2.0", "8f8936746fad495999afcd2c2
42b0bf0");

    @InjectMocks
    private AetherService aetherServiceInvalidKeys = new AetherService("https://api.breezometer.com/air-
quality/v2", "0", "https://api.weatherbit.io/v2.0", "0");
```

Figure 22. Initialization of the service entities and creation of the Mocks. Note the usage of lenient to bypass strict stubbing validation rules allowing us to load some expectations that might not be used in all tests

- As stated previously, this class includes tests for just about anything that might happen within the AetherService class. Two setups are included, one to initialize our repository with 3 locales, and one for our cache to have it include 4 items with varying parameters. This setup is ran every time before each and every test so that we're using a fresh instance of the mock for every test.
- There are tests such as *whenCreatingNonExistingLocale_thenReturnLocale()* and *whenUpdatingNonExistingLocale_thenReturnNull()* that assert the proper interaction between our service and the LocaleRepository

```
    @Test
    public void whenDeletingExistingLocale_thenReturnDeletedLocale() {
        Locale aveiro = new Locale("Aveiro", 8.6538, 40.6405);
        Locale deleted = aetherService.deleteLocale(aveiro.getName());

        assertThat(deleted.getName()).isEqualTo(aveiro.getName());

        assertThat(deleted.getLon()).isEqualTo(aveiro.getLon());

        assertThat(deleted.getLat()).isEqualTo(aveiro.getLat());

        Locale lisboa = new Locale("Lisboa", 9.1393, 38.7223);

        deleted = aetherService.deleteLocale(lisboa.getName());

        assertThat(deleted.getName()).isEqualTo(lisboa.getName());
        assertThat(deleted.getLon()).isEqualTo(lisboa.getLon());
        assertThat(deleted.getLat()).isEqualTo(lisboa.getLat());

        Mockito.verify(localeRepository, VerificationModeFactory.times(2)).delete(Mockito.any(Locale.class));
        Mockito.reset(localeRepository);
    }
```

Figure 23. A test verifying the response given when attempting to delete a locale that exists within the LocaleRepository. Note how we start by defining first the Locale we're trying to delete, then calling the deleteLocale() method and then verifying that the responded locale is the same as the one we attempted to delete. This is tested for two different locales, and afterwards we also verify that the correct LocaleRepository method was called, as well as resetting it's mock.

```
    @Test
    public void whenCreatingExistingLocale_thenReturnNull() {
        LocalePOJO aveiroPOJO = new LocalePOJO("Aveiro", 8.6538, 40.6405);
        Locale created = aetherService.createLocale(aveiroPOJO);

        Mockito.verify(localeRepository, VerificationModeFactory.times(0)).save(Mockito.any(Locale.class));
        Mockito.reset(localeRepository);

        assertThat(created).isNull();
    }
```

Figure 24. A test verifying the response given when attempting to create a locale that already exists. Note how we start by defining the LocalePojo object that is passed to the service's createLocale() method. Afterwards we verify that our service effectively didn't call the save method on the LocaleRepository (since it verifies that there already exists a locale with the specified parameters), and that the response given is a null

- Tests to the service's direct interaction with the cache's statistical methods were also made, such as *whenGettingCacheHitsNMisses_thenReturnMapWithHitsNMisses()*

```
    @Test
    public void whenGettingCacheHitsNMisses_thenReturnMapWithHitsNMisses() {
        Map<String, Integer> hitsnmisses = aetherService.getHitsNMisses();
        int expectedHits = 5;
        int expectedMisses = 4;
        int expectedTotalRequests = 9;

        Mockito.verify(localCache, VerificationModeFactory.times(1)).getHits();
        Mockito.verify(localCache, VerificationModeFactory.times(1)).getMisses();
        Mockito.verify(localCache, VerificationModeFactory.times(1)).getTotalRequests();
        Mockito.reset(localCache);

        assertThat(hitsnmisses).hasSize(3);
        assertThat(hitsnmisses.get("Hits")).isEqualTo(expectedHits);
        assertThat(hitsnmisses.get("Misses")).isEqualTo(expectedMisses);
        assertThat(hitsnmisses.get("TotalRequests")).isEqualTo(expectedTotalRequests);
    }
```

Figure 25. A test verifying what happens when we our service tries to get the stats from the cache. Note how we define what the expected responses are, verify that the appropriate cache's methods were called and that the response has both the appropriate format (a map) and values.

- Finally, we also did thorough testing on our service's behavior when told to retrieve the air quality information (as these were the methods with the most amount of logic behind them). We tested what happened when the service was told any given combination of getting the information by coordinate or by locale, by specific API or none, by current, past or future and even for when the item was already stored, or not in the cache. Obviously for every success case test, failure state tests were also included such as *whenGettingQualityByCoordBreeze_AndNotInCache_returnInfo_andAddToCache()* vs *whenGettingQualityByCoordBreeze_andInvalidCoord_throwException()*.

```
    @Test
    public void whenGettingQualityByCoordWeather_AndNotInCache_returnInfo_andAddToCache() throws JsonProcessi
ngException {
        double lat = 8.4043;
        double lon = 39.6054;
        Coordinates coords = new Coordinates(lat, lon);

        String api = "Weather";

        // PAST
        AirResponse resultCurrent = aetherService.getQualityByCoord(lat, lon, "PAST", api, null);

        // Verify we added to the cache at the end
        Mockito.verify(localCache, VerificationModeFactory.times(1)).addToCache(resultCurrent, "PAST");
        assertThat(resultCurrent.getCoordinates().getLat()).isEqualTo(lat);
        assertThat(resultCurrent.getCoordinates().getLon()).isEqualTo(lon);
        assertThat(resultCurrent.getData().size()).isGreaterThanOrEqualTo(72);

        // CURRENT
        resultCurrent = aetherService.getQualityByCoord(lat, lon, "CURRENT", api, null);

        // Verify we added to the cache at the end
        Mockito.verify(localCache, VerificationModeFactory.times(1)).addToCache(resultCurrent, "CURRENT");
        assertThat(resultCurrent.getCoordinates().getLat()).isEqualTo(lat);
        assertThat(resultCurrent.getCoordinates().getLon()).isEqualTo(lon);
        assertThat(resultCurrent.getData().size()).isEqualTo(1);

        // FUTURE
        resultCurrent = aetherService.getQualityByCoord(lat, lon, "FUTURE", api, null);

        // Verify we added to the cache at the end
        Mockito.verify(localCache, VerificationModeFactory.times(1)).addToCache(resultCurrent, "FUTURE");
        assertThat(resultCurrent.getCoordinates().getLat()).isEqualTo(lat);
        assertThat(resultCurrent.getCoordinates().getLon()).isEqualTo(lon);
        assertThat(resultCurrent.getData().size()).isGreaterThanOrEqualTo(72);

        // Verify we tried to get from the cache
        Mockito.verify(localCache, VerificationModeFactory.times(1)).getFromCache(coords, api, "CURRENT");
        Mockito.verify(localCache, VerificationModeFactory.times(1)).getFromCache(coords, api, "PAST");
        Mockito.verify(localCache, VerificationModeFactory.times(1)).getFromCache(coords, api, "FUTURE");
        Mockito.reset(localCache);
    }
```

Figure 26. Example of a test following the logical path of what happens when attempting to get the air quality given a coordinate, specifying the weather API and when the specific item is still not in the cache. Note how the test is applied for all types of requests – current, past or future, and how we verify both the response aswell as the interactions with the cache.

```
    @Test
    public void whenGettingQualityByCoordNoAPI_andBothAPIIsError_throwException() {
        String api = null;
        Coordinates coords = new Coordinates(40, 2);

        RestClientResponseException exception = assertThrows(RestClientResponseException.class, () -> {
            aetherServiceInvalidKeys.getQualityByCoord(coords.getLat(), coords.getLon(), "CURRENT", api, null
);
        });
        assertThat(exception.getResponseBodyAsString()).contains("{\"error\": \"Both APIs responded with erro
rs - ");
        assertThat(exception.getResponseBodyAsString()).contains("Breeze: ");
        assertThat(exception.getResponseBodyAsString()).contains("WeatherBit: ");
    }
```
Figure 27. A test that ascertains what happens when the service is working with invalid keys. Observe how we verify that a correct error with the appropriate message is thrown

LocaleRepositoryUT.java

- Unit test to verify the data access interactions given by the LocaleRepository component. This test class is overall pretty simple due to the fact that our LocaleRepository contains only two methods worth testing (since all other methods, such as save and delete are automatically included due to the JPARepository inheritance).
- This class utilizes a TestEntityManager used to directly access the database and limits the context of the test to focus only on the repository component. The usage of the @DataJPATest annotations makes it so our H2 database is set up (as long as it's included in our POM.xml) Only 4 tests are included within:
    - *whenFindByName_thenReturnLocale()* – Tests that when searching for a locale by name, and when there exists a locale with said name, it is returned
    - *whenInvalidName_thenReturnNull()* – Tests that when searching for a locale by a name that no locale in the H2 database possesses, a null is returned
    - *givenSetOfLocales_whenFindAll_thenReturnAllLocales()* – Tests that when the database possesses a set of locales stored, when we try to get all locales using our repository that set is returned in the form of a list.
    - *givenNoLocales_whenFindAll_thenReturnEmptyList()* – Verifies that when there are no entities stored within the database, then an empty list is returned by our repository.

```
    @Test
    public void whenFindByName_thenReturnLocale() {
        Locale aveiro = new Locale("Aveiro", 8.6538, 40.6405);
        entityManager.persistAndFlush(aveiro);

        Locale found = localeRepository.findByName(aveiro.getName());
        assertThat(found.getName()).isEqualTo(aveiro.getName());
    }
```
Figure 28. A test verifying that when trying to get a Locale by name and there indeed exists a locale with the searched for name in the database, then it gets returned

CoordinatesUT.java

- A very simple and fast unit test for a very simple class. Coordinates.java is a utility class that contains a custom equals, hashcode and toString function, and as such these are the three methods whose behaviors are tested.

```
    @Test
    void testToString() {
        double aLat = 5;
        double aLon = 5;
        double bLat = 10;
        double bLon = -4;
        String locale = "Aveiro";
        Coordinates coordA = new Coordinates(aLat,aLon);
        Coordinates coordB = new Coordinates(bLat,bLon);

        coordA.setLocale(locale);

        assertThat(coordA.toString()).isEqualTo("(lat=" + aLat + ", lon=" + aLon +")");
        assertThat(coordB.toString()).isEqualTo("(lat=" + bLat + ", lon=" + bLon +")");
    }
```

Figure 29. A test verifying that the Coordinate.java toString function returns a string with the expected template

LocalCacheUT.java

- This was the second most complex unit test class included and it's aimed at verifying the correct functioning of our cache. It verifies all main methods and logical paths of the LocalCache.java class with the exception of the scheduled method which could only be tested in an integration test since otherwise the @Scheduled annotation that makes it work wouldn't be automatically triggered at every timeout.
- This class instantiates only a LocaleCache object since our cache doesn't really depend on any other component (it utilizes only other domain and utility classes such as CacheKey.java). Before each test is run a setup function is called which starts by resetting the current cache, deleting all of the items it's holding and resetting it's stats back to 0, followed by inserting 3 items into the cache for the purposes of testing.
- Like with our service class, we aimed at being as thorough as possible, testing all possibilities that might occur within the inner workings of our class. For purposes of illustrating the type of tests included, only a couple will be illustrated below in the following code snippets.

```
    @Test
    public void whenInvalidItem_andGivenNoAPI_thenReturnNull__andIncrementMissesAndRequests() {
        Coordinates nonExistantCoord = new Coordinates(8.6410, 41.0072);
        String api = null;
        String reqType = "PAST";
        int startingHits = localCache.getHits();
        int startingMisses = localCache.getMisses();
        int startingRequests = localCache.getTotalRequests();

        AirResponse response = localCache.getFromCache(nonExistantCoord, api, reqType);
        assertThat(response).isNull();

        nonExistantCoord = new Coordinates(7.6410, 45.0072);
        reqType = "CURRENT";

        response = localCache.getFromCache(nonExistantCoord, api, reqType);
        assertThat(response).isNull();
        assertThat(localCache.getHits()).isEqualTo(startingHits);
        assertThat(localCache.getMisses()).isEqualTo(startingMisses + 2);
        assertThat(localCache.getTotalRequests()).isEqualTo(startingRequests + 2);
    }
```

Figure 30. A test that asserts that, when looking for an item that's not included in our cache and no API is speicified, we return a null and properly increment the misses and request statistical variables. Note how we test for different types of request types (past and current). The reason as to

having specified that no API is given is due to the fact that internally, when given an API our cache follows a different logical path when retrieving an item, comparatively to when no API is specified, hence the need to test both paths.

```java
    @Test
    public void whenCacheFull_removeOldest_andAddNew(){
        AirResponse aveiroQuality = new AirResponse();

        Coordinates aveiroCoord = new Coordinates(8.6538, 40.6405);

        Map<Integer, AirQualityItem> aveiroData = new HashMap<>();

        aveiroQuality.setApiName("Weather");
        aveiroCoord.setLocale("Aveiro");
        aveiroQuality.setCoordinates(aveiroCoord);
        aveiroQuality.setRequestDate(LocalDateTime.now());   // A date that is way over 2 minutes from the co
rrent one

        aveiroQuality.setData(aveiroData);

        Coordinates espinhoCoord = new Coordinates(8.6410, 41.0072); // Coordinates of the oldest item

        int startingSize = localCache.getSize();

        // Fill Cache
        int maxSize = localCache.getMaxCacheSize(); // Max cache size is 150
        for(int i = startingSize ; i < maxSize ; i++){
            AirResponse inc = new AirResponse();
            Coordinates coor = new Coordinates(i,i);

            inc.setCoordinates(coor);
            localCache.addToCache(inc, "CURRENT");
        }

        assertThat(localCache.getSize()).isEqualTo(maxSize);

        localCache.addToCache(aveiroQuality, "CURRENT");

        assertThat(localCache.getCache().values()).hasSize(maxSize).extracting(AirResponse::getCoordinates).d
oesNotContain(espinhoCoord);

        assertThat(localCache.getCache().values()).contains(aveiroQuality);

        assertThat(localCache.getCacheQueue().peek().getCoords()).isNotEqualTo(espinhoCoord);
    }
```

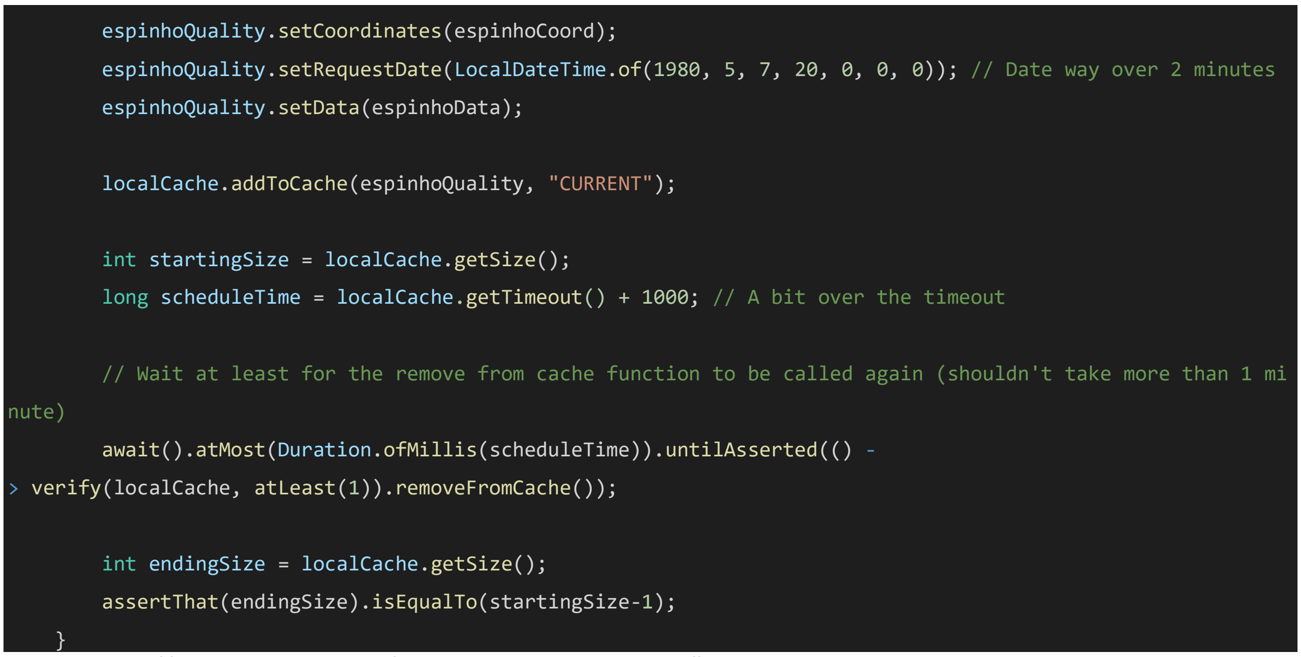
Figure 31. A test that verifies that when we're trying to add a new item to our cache but it's full we remove the oldest item and only then add the new one. Note how we first defined the item we were trying to add, then artificially filled the cache's size so that it would contain 150 (the default max size for our cache) items, and then verified what would happen when adding the new item.

Now, moving on to some more complex Integration Tests, namely ones focused on verifying the way our controller class was behaving and how an interaction with our API would go down. That, however, wasn't all that was tested, and as such, below are all of the integration tests written for our application.

LocalCacheIT.java

- This is an integration test on the LocaleCache.java class, required due to it's removeFromCache() method, annoted with Spring's @Scheduled annotation. The reason as to why an integration was needed was due to the necessity of starting the application context in order for the annotation to take effect and actually start calling the method on the set schedule. The @SpringJUnitConfig annotation is used for this purpose and to also boot start our beans in the testing environment.
- Besides that, this class also injects the @SpyBean annotation on the LocalCache bean used for testing. This annotation is used to check the number of times that the scheduled method is called. We also utilized Awaitility to test the behaviour of our method as time passes in a more declarative way. It should be stated that originally we were using a simple *Thread.sleep()* to force our test to wait for the passage of time, but after parsing our code through the SonarCloud analysis, it was heavily suggested that this be removed and replaced.
- With all of this said, the tests within this class are aimed solely at testing the functioning of the removeFromCache() method, asserting it's behaviour as time passes. The 4 tests included are:
  - *whenTimeout_andEmpty_doNotRemove()* – Tests that when the set timeout passes and there are no items stored in the cache, our function gets called, the code doesn't break unexpectedly and the size remains 0.
  - *everyMinute_removeGetsCalled()* – Asserts that our method gets called every minute (which is the default timeout)
  - *whenTimeout_andNotEmpty_andOldestItemOlderThanTimeout_RemoveFromCache()* – Verifies that, when the timeout passes, and there's an item in the cache that is older than one minute, it gets removed from it.
  - *whenTimeout_andNotEmpty_andOldestItemYoungerThan2Mins_DoNotRemove()* – This test verifies that, when the timeout passes and there's an item in the cache that's been there for less than the set timeout, it doesn't actually get removed from the cache (as it hasn't been in there for over a minute yet). Note that the name of the function says "2Mins" because originally that was what the cache's timeout was set for, however, for the purpose of speeding up the execution of tests and more easy verification of possible errors, the timeout was decreased internally within the LocaleCache.java class to 1 minute.
- It should also be said that this class contains a setup function that is called before each test that resets the entire cache bean. This is done so that the timer on the scheduled method is reset. Note also that the testing timeout is set to be the timeout + 1000 milliseconds. This is done because Awaitality's await method requires us to wait for the full execution of the method, so we add the 1000 ms as an error window to allow for completion of the function's logic.

```
    @Test
    public void whenTimeout_andNotEmpty_andOldestItemOlderThanTimeout_RemoveFromCache() throws InterruptedExc
eption {
        AirResponse espinhoQuality = new AirResponse();
        Coordinates espinhoCoord = new Coordinates(8.6410, 41.0072);
        Map<Integer, AirQualityItem> espinhoData = new HashMap<>();
        BreezeAirQuality testEspinho = new BreezeAirQuality();
        testEspinho.setBaqi(69);
        espinhoData.put(0, testEspinho);

        espinhoQuality.setApiName("Breeze");
```



```
        espinhoQuality.setCoordinates(espinhoCoord);
        espinhoQuality.setRequestDate(LocalDateTime.of(1980, 5, 7, 20, 0, 0, 0)); // Date way over 2 minutes
        espinhoQuality.setData(espinhoData);

        localCache.addToCache(espinhoQuality, "CURRENT");

        int startingSize = localCache.getSize();
        long scheduleTime = localCache.getTimeout() + 1000; // A bit over the timeout

        // Wait at least for the remove from cache function to be called again (shouldn't take more than 1 mi
nute)
        await().atMost(Duration.ofMillis(scheduleTime)).untilAsserted(() -
> verify(localCache, atLeast(1)).removeFromCache());

        int endingSize = localCache.getSize();
        assertThat(endingSize).isEqualTo(startingSize-1);
    }
```

Figure 32. A test that assures that after the timeout passes, we try and effectively remove the oldest item in the cache, since it has a set request date that is older than the timeout. We do this by first inserting a new item into the cache with a set request date of way over a minute. Then we wait, using Awaitility's functions, that our removeFromCache method is called at least once. Finally we check that the item has been removing by verifying that the final size is one smaller than the starter one.

AetherControllerIT.java

- This is a simple Integration Test set that is aimed at verifying the functioning and top-level logic of our AetherController.java class. We do this by recursing to the @WebMVCTest annotation, that simulates the behavior of the application server, utilizing a MockMVC bean as a reference to the server context and mocking our service class, which the controller mostly interacts with. All of this creates a very light testing environment, perfect for assuring that our controller is written and working as intended.
- It should be noted that most of our controller's logic has the goal of defining all of our API's endpoints, calling the appropriate service method and returning it's response. It also contains methods to handle possible errors thrown by our service. With this in mind, tests were written to ascertain the responses retrieved after calling each of the possible endpoints, both with good and bad arguments, as well as testing what happens when our service returns a possible exception.
- It contains over 25 different tests, such as whenAPIError_throwError, that explores what we respond when our service returns an error related to the external API (which might happen, for example, if our tokens expire) and givenLocale_andAPI_getAirQuality_andAssureAPIIsCorrect_Future(), which tests if, when we search for the future forecast of air quality by locale, and specify what API we want be used, a response is generated and it was created utilizing the specified external API. A plethora of other tests are also included, verifying all possible logical path our controller's code might take.
- This test class, as well as some others, utilizes the JsonUtil test support class which helps converting into json format the necessary request bodies for some endpoints (namely our create and update locale endpoints). This class is courtesy of professor Ilídio C. Oliveira and was given in the context of the TQS practical classes.

```
     @Test
    // Tests that when the weatherbit api returns an error, we get returned a ResponseStatusException
    public void givenWeatherBitError_getErrorMessage_Past() throws Exception {
        given(service.getQualityByCoord(Mockito.anyDouble(), Mockito.anyDouble(), Mockito.anyString(), Mockit
o.any(), Mockito.any())).willThrow(new RestClientResponseException("Mock", 400, "Mock", HttpHeaders.EMPTY, "{
\"error\": \"Mock Error raised by WeatherBit API\"}".getBytes(), Charset.defaultCharset()));

        double lat = 8.6538;
        double lon = 40.6405;

        String reqType = "PAST";

        String expectedErrorMessage = "Mock Error raised by WeatherBit API";

        // Past
        mvc.perform(get("/api/air/past").contentType(MediaType.APPLICATION_JSON).param("lat", lat + "").param
("lon", lon + ""))
                .andExpect(status().is4xxClientError())
                .andExpect(status().reason(is(expectedErrorMessage)));

        verify(service, VerificationModeFactory.times(1)).getQualityByCoord(lat, lon, reqType, null, null);
        reset(service);
    }
```

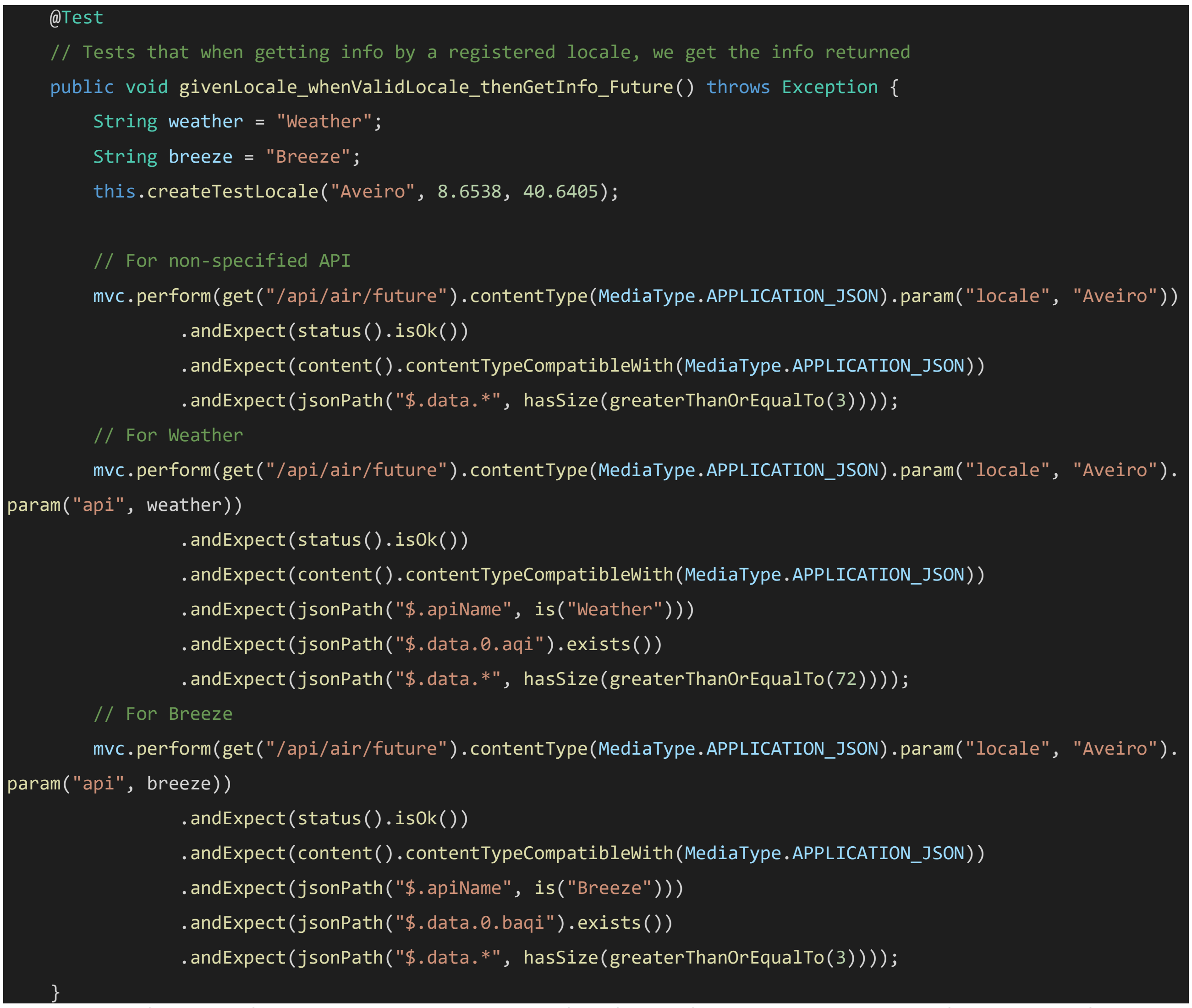
Figure 33. A test that assures that when we search for the past history of air quality using the WeatherBit API, and it returns an error, we respond with the error retrieved from that external API. We do this by first mocking the behaviour of our service and then performing a MockMVC call to the /api/air/past endpoint, telling it what to expect the response to be. We also verify that the correct service method was called and finish by resetting it

```
    @Test
    // Tests that when fetching cache we get returned the cache
    public void givenCacheItems_whenGetCache_thenReturnCache() throws Exception {
        AirResponse aveiroQuality = new AirResponse();
        aveiroQuality.setApiName("Test1");
        AirResponse espinhoQuality = new AirResponse();
        aveiroQuality.setApiName("Test2");
        List<AirResponse> cache = Arrays.asList(espinhoQuality, aveiroQuality);

        given(service.getEntireCache()).willReturn(cache);

        mvc.perform(get("/api/cache/items").contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$", hasSize(2)))
                .andExpect(jsonPath("$[0].apiName", is(espinhoQuality.getApiName())))
                .andExpect(jsonPath("$[1].apiName", is(aveiroQuality.getApiName())));


        verify(service, VerificationModeFactory.times(1)).getEntireCache();
        reset(service);
    }
```

Figure 34. This test is used to verify that when trying to get the cache's items our controller returns a json with a list of AirResponse objects (obviously, converted to json)

AetherRestControllerIT.java

- This test set has the goal of verifying, once again, the correct behavior of our controller class, but this time also testing the functioning of our entire REST API service but without involving an API user. We do this by utilizing the @SpringBootTest to start the web context. Once again we'll be using the MockMVC entity to access our endpoints. It should be noted that during this test our application will be running in the a normal SpringBoot context.
- In this test class we utilize a local cache and locale repository bean that is reset after each of our tests (all locales are deleted from our repository and all items from our cache, alongside it's statistical variables being reset to 0).
- With over 30 tests, this class utilizes no mocks, unlike the last one discussed which mocked the service's behavior. Our tests are, once again, focused on both success and expected failure states as we didn't limit ourselves to just making a single test for each endpoint. On the contrary, we attempted to fully test every single decision and path that might be taken by our algorithms making it so our controller calls every single possible variation of our service's methods. Below are some examples of some of the included tests, which pass a request and the necessary parameters (or not, in order to test what happens when the request is lacking in the necessary parameters or request body) and assert if the generated response is as expected.



```java
    @Test
    // Tests that when getting info by a registered locale, we get the info returned
    public void givenLocale_whenValidLocale_thenGetInfo_Future() throws Exception {
        String weather = "Weather";
        String breeze = "Breeze";
        this.createTestLocale("Aveiro", 8.6538, 40.6405);

        // For non-specified API
        mvc.perform(get("/api/air/future").contentType(MediaType.APPLICATION_JSON).param("locale", "Aveiro"))
                .andExpect(status().isOk())
                .andExpect(content().contentTypeCompatibleWith(MediaType.APPLICATION_JSON))
                .andExpect(jsonPath("$.data.*", hasSize(greaterThanOrEqualTo(3))));
        // For Weather
        mvc.perform(get("/api/air/future").contentType(MediaType.APPLICATION_JSON).param("locale", "Aveiro").
param("api", weather))
                .andExpect(status().isOk())
                .andExpect(content().contentTypeCompatibleWith(MediaType.APPLICATION_JSON))
                .andExpect(jsonPath("$.apiName", is("Weather")))
                .andExpect(jsonPath("$.data.0.aqi").exists())
                .andExpect(jsonPath("$.data.*", hasSize(greaterThanOrEqualTo(72))));
        // For Breeze
        mvc.perform(get("/api/air/future").contentType(MediaType.APPLICATION_JSON).param("locale", "Aveiro").
param("api", breeze))
                .andExpect(status().isOk())
                .andExpect(content().contentTypeCompatibleWith(MediaType.APPLICATION_JSON))
                .andExpect(jsonPath("$.apiName", is("Breeze")))
                .andExpect(jsonPath("$.data.0.baqi").exists())
                .andExpect(jsonPath("$.data.*", hasSize(greaterThanOrEqualTo(3))));
    }
```

Figure 35. This specific test asserts that when trying to get the future forecast of air quality by locale we get the info in the expected format. We apply this test without specifying an API, and also for the cases where we specify either Weather or Breeze. The first thing we do is call our createTestLocale function (a support function included in this file that inserts a new locale into our repository) and then we proceed to use our MockMVC to call the respective endpoint, passing the locale parameter for the locale we just created and analysing the response's parameters.

```
    @Test
    // Tests that when updating an existing locale it gets updated
    public void whenValidInput_thenUpdateLocale() throws Exception {
        this.createTestLocale("Espinho", 8.6538, 40.6405);
        double updatedLon = 41.0072;
        double updatedLat = 8.6410;
        LocalePOJO espinho = new LocalePOJO("Espinho", updatedLat, updatedLon);

        mvc.perform(put("/api/locale").contentType(MediaType.APPLICATION_JSON).content(JsonUtil.toJson(espinh
o)))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.name", is("Espinho")))
                .andExpect(jsonPath("$.lat", is(updatedLat)))
                .andExpect(jsonPath("$.lon", is(updatedLon)));

        Locale foundLocale = localeRepository.findByName("Espinho");
        assertThat(foundLocale.getName()).isEqualTo(espinho.getName());
        assertThat(foundLocale.getLat()).isEqualTo(espinho.getLat());
        assertThat(foundLocale.getLon()).isEqualTo(espinho.getLon());
    }
```

Figure 36. This test is used to verify that when we provide a valid request body, calling our endpoint to update a locale (that actually exists), the response is as expected (i.e equal to the updated locale) and ascertaining that the item was actually updated on our repository.

```
    @Test
    // Tests that when getting info by a non registered locale, we get an error
    public void givenLocale_whenInvalidLocale_thenGetError_Past() throws Exception {
        String invalidLocale = "Espinho";
        mvc.perform(get("/api/air/past").contentType(MediaType.APPLICATION_JSON).param("locale", invalidLocal
e))
                .andExpect(status().isNotFound())
                .andExpect(status().reason(is("Error! Couldn't find any locale with the specified name")));


    }
```

Figure 37. Verify that when attempting to get the past history of air quality when searching by a locale that's not registered in our repository, we get returned an error 404 with the reason being the one expected.

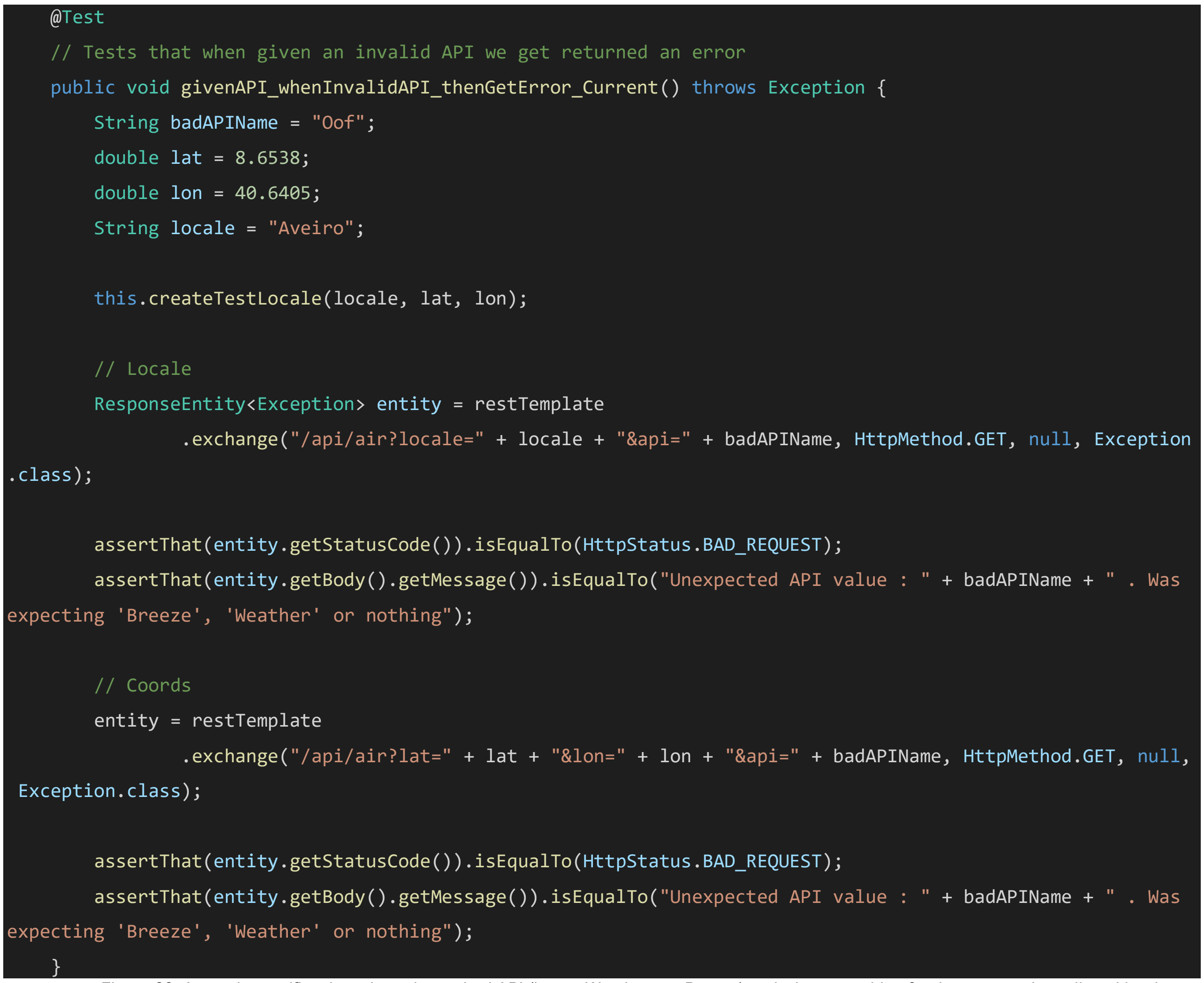
AetherRestControllerTemplateIT.java

- This test set is in many ways like the previous one. They both test our entire REST API and integrate our multiple classes at play when processing an API request and most test cases made on the previous class are also repeated on this one. The difference between these sets, however, is that, unlike the AetherRestControllerIT.java test class, this one actually explicitly involves an HTTP client, testing what the behavior of our application is when processing actual requests coming from a user of our API.
- Like before, we start by starting the full web context utilizing the @SpringBootTest annotation and our API is once again put working under the normal SpringBoot context. This time, however, we're using a REST client, by utilizing an instance of TestRestTemplate, to generate realistic requests. We also use the @LocalServerPort annotation applied to a parameter to inject the HTTP port that gets allocated to our app on runtime.
- As stated, the logic and cases we're testing are similar to the ones from the aforementioned class, but this time, instead of using the MockMVC to call our API, we're using our

TestRestTemplate instance to create our calls. Included below are some examples of tests included in this set. Note that this class also has a reset function that is called after each of the tests and resets both the repository and the cache.

```java
    @Test
    // Tests that when getting info by a registered locale, we get the info returned
    public void givenLocale_whenValidLocale_thenGetInfo_Current() throws Exception {
        String weather = "Weather";
        String breeze = "Breeze";

        double lat = 8.6538;
        double lon = 40.6405;
        String locale = "Aveiro";

        this.createTestLocale(locale, lat, lon);

        // For non-specified API
        ResponseEntity<AirResponse> entity = restTemplate
                .exchange("/api/air?locale=" + locale, HttpMethod.GET, null, AirResponse.class);

        assertThat(entity.getStatusCode()).isEqualTo(HttpStatus.OK);
        assertThat(entity.getBody().getData().size()).isEqualTo(1);

        // For Weather
        entity = restTemplate
                .exchange("/api/air?locale=" + locale + "&api=" + weather, HttpMethod.GET, null, AirResponse.
class);

        assertThat(entity.getStatusCode()).isEqualTo(HttpStatus.OK);
        assertThat(entity.getBody().getApiName()).isEqualTo(weather);
        assertThat(entity.getBody().getData().size()).isEqualTo(1);
        assertThat(entity.getBody().getData().get(0)).isInstanceOf(WeatherAirQuality.class);
        assertThat(entity.getBody().getData().get(0)).isNotInstanceOf(BreezeAirQuality.class);

        // For Breeze
        entity = restTemplate
                .exchange("/api/air?locale=" + locale + "&api=" + breeze, HttpMethod.GET, null, AirResponse.c
lass);

        assertThat(entity.getStatusCode()).isEqualTo(HttpStatus.OK);
        assertThat(entity.getBody().getApiName()).isEqualTo(breeze);
        assertThat(entity.getBody().getData().size()).isEqualTo(1);
        assertThat(entity.getBody().getData().get(0)).isInstanceOf(BreezeAirQuality.class);
        assertThat(entity.getBody().getData().get(0)).isNotInstanceOf(WeatherAirQuality.class);
    }
```

Figure 38. A test that verifies that when given a locale that's in our repository, and we try to get the current air quality information for that locale we get returned a code 200 and the response body contains the right values. Note the usage of the restTemplate.exchange() method to call our API.



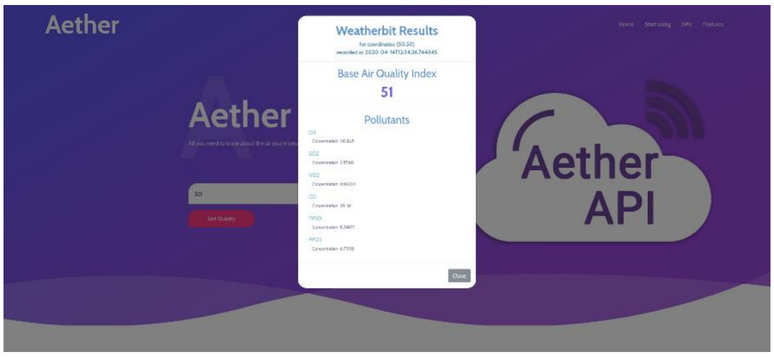
```
    @Test
    // Tests that when given an invalid API we get returned an error
    public void givenAPI_whenInvalidAPI_thenGetError_Current() throws Exception {
        String badAPIName = "Oof";
        double lat = 8.6538;
        double lon = 40.6405;
        String locale = "Aveiro";

        this.createTestLocale(locale, lat, lon);

        // Locale
        ResponseEntity<Exception> entity = restTemplate
                .exchange("/api/air?locale=" + locale + "&api=" + badAPIName, HttpMethod.GET, null, Exception
.class);

        assertThat(entity.getStatusCode()).isEqualTo(HttpStatus.BAD_REQUEST);
        assertThat(entity.getBody().getMessage()).isEqualTo("Unexpected API value : " + badAPIName + " . Was
expecting 'Breeze', 'Weather' or nothing");

        // Coords
        entity = restTemplate
                .exchange("/api/air?lat=" + lat + "&lon=" + lon + "&api=" + badAPIName, HttpMethod.GET, null,
 Exception.class);

        assertThat(entity.getStatusCode()).isEqualTo(HttpStatus.BAD_REQUEST);
        assertThat(entity.getBody().getMessage()).isEqualTo("Unexpected API value : " + badAPIName + " . Was
expecting 'Breeze', 'Weather' or nothing");
    }
```

Figure 39. A test that verifies that when given a bad API (i.e not Weather nor Breeze) and when searching for the current air quality, either by locale or by coordinates we get returned the correct error code and error message.

Before moving on to the next section there's one more test within our set of tests that has, so far, gone unmentioned. That is the **AetherApplicationTests.java**. This is a file that was automatically generated when creating the Spring Boot app and that, at the start, we did not know what to do with. After some research however, most discussions found reached the consensus that the our AetherApplication.java class, the one we use to actually start our app, doesn't really require much testing since it has no logic other than the one automatically generated when creating a new project using the Spring Initializr. That being said, we still included a small test named contextLoads() that verifies that our controller bean, upon starting the Spring Boot context, is not null.

3.3 Functional testing

Moving on to functional testing, we kept the same ideology that we've been showcasing up until now, test every single possible interaction with our frontend in the hopes of covering all ways a user might utilize our frontend page. As aforementioned, we utilized the Sellenium Web Driver (connecting to a Firefox Driver) to create our tests which actually caused some problems. Initially, the first iteration of our webpage heavily utilized modals to show the results of calling our API. It was only after starting the testing procedure, and asking the professor in charge, that we noticed this testing tool has a lot of problems recognizing the active modals. This lead us to having to swap most of our interface's jQuery logic and page layout in order to be able to embed the obtained results within the page itself. The original, modal based interface solution is still included in the source code and is dubbed index-with-modals.html for the curious who might want to check all changes that had to be hastily implemented.

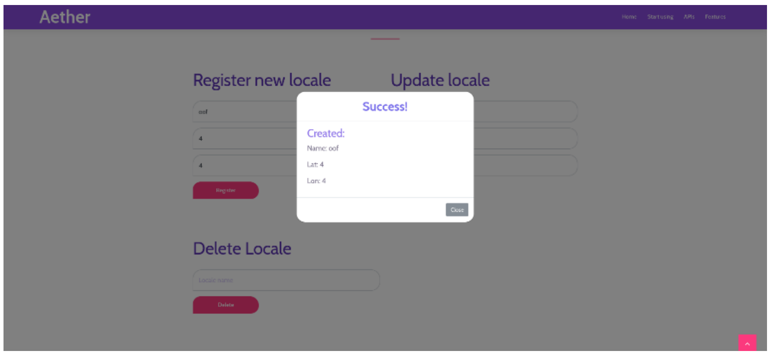


Weatherbit Results
for coordinates: (50,50)
recorded in: 2020-04-14T12:34:36.744845

Base Air Quality Index
51

Pollutants
O3
Concentration: 110.845
SO2
Concentration: 2.57410
NO2
Concentration: 0.90241
CO
Concentration: 35.31
PM10
Concentration: 8.24877
PM25
Concentration: 6.71938

Figure 40. The old way of showing our API’s results, through the usage of modals



Success!

Created:
Name: oof
Lat: 4
Lon: 4

Figure 41. A success modal being shown after registering a new locale

It should also be noted that, due to complications and lack of extra time, the functional test was the only one that could not be properly integrated into the CI/CD pipeline. This is caused due to the fact that Gitlab’s runners aren’t prepared to instantiate and use the required web driver to perform the tests. As such, these tests can be ran, either by individually running the **InterfaceTests.java** class or by using the **mvn test** command (which will run all tests in the project).

Turning to our implementation of the functional test set, we decided to utilize the Page Object design pattern in order to make our actual tests more readable and understandable, keeping all logic to actually control the web driver within the **PageObject.java** class. By doing this, we gave our InterfaceTests.java class a high level interface to interact with, migrating all extensive logic of utilizing the web driver into the page object class, assuring a clean and readable test class.

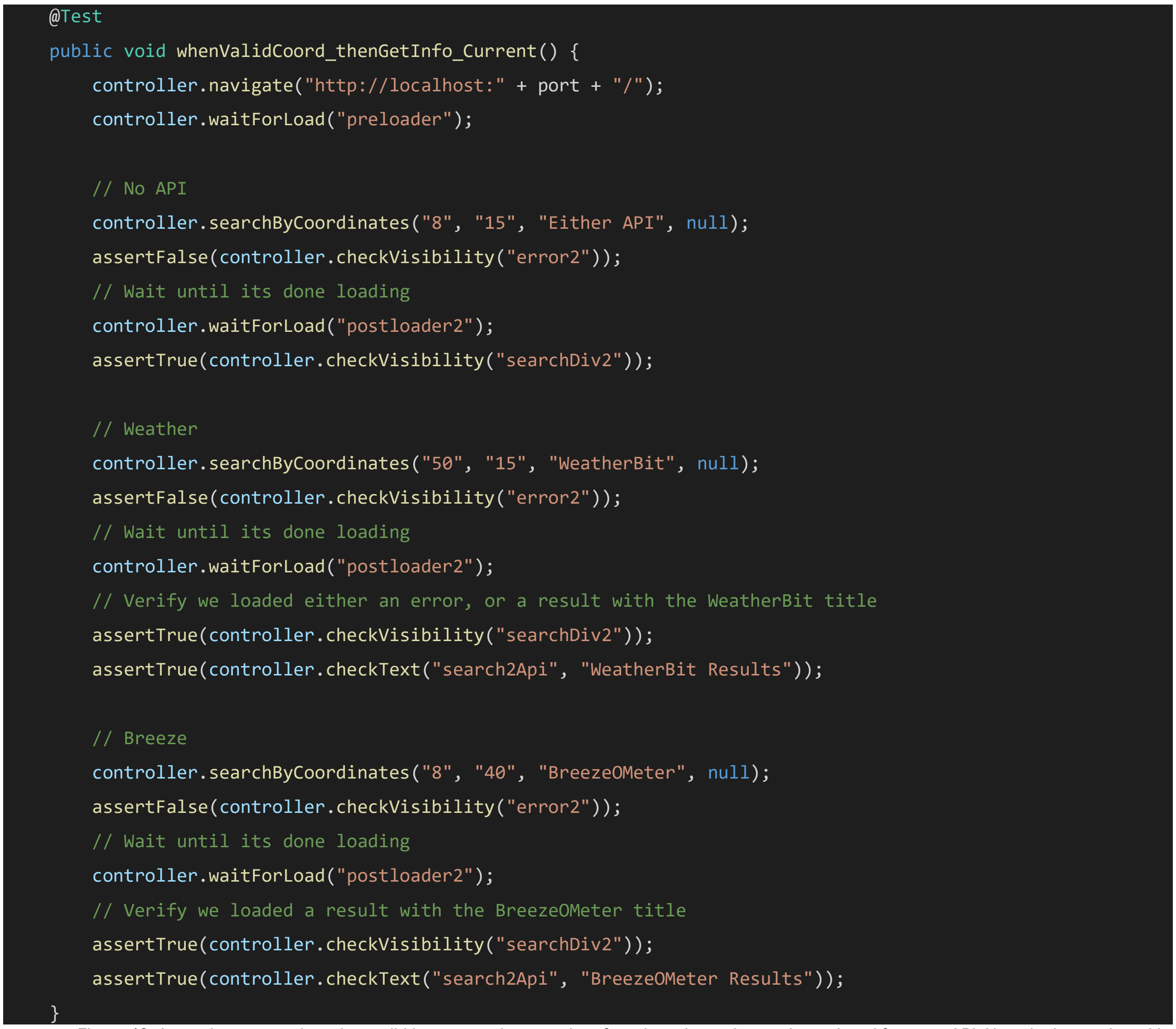
Starting with breaking down our **PageObject.java**, this class, when instantiated, starts a Firefox web driver and contains several methods that allow for the easy interaction with our webpage, aswell as some support methods that assert certain conditions such as an element with a given id being visible, or verifying that a certain element contains the given text.

```
    public void searchByCoordinates (@Nullable String lat, @Nullable String lon,
@Nullable String api, @Nullable String reqType){
        if (lat != null) {
            driver.findElement(By.id("lat2")).click();
            if (lat.equals("")) {
                driver.findElement(By.id("lat2")).clear();
            } else {
                driver.findElement(By.id("lat2")).clear();
            driver.findElement(By.id("lat2")).sendKeys(lat);
        }
    }

    if (lon != null) {
        driver.findElement(By.id("lon2")).click();
        if (lon.equals("")) {
            driver.findElement(By.id("lon2")).clear();
        } else {
            driver.findElement(By.id("lon2")).clear();
            driver.findElement(By.id("lon2")).sendKeys(lon);
        }
    }

    if (api != null) {
        Select apiSelect = new Select(driver.findElement(By.id("api2")));
        apiSelect.selectByVisibleText(api);
    }

    if (reqType != null) {
        Select reqSelect = new Select(driver.findElement(By.id("reqType2")));
        reqSelect.selectByVisibleText(reqType);
    }

    driver.findElement(By.id("search2")).sendKeys(Keys.SPACE);
}
```

Figure 42. A method from our PageObject class that is utilized to interact with the search by coordinate form. Note how it takes as arguments the intended latitude, longitude, api and request type (past, current or future) as these are all of the fields present in this form. If any of these parameters is passed as null then the driver simply wont change the current value of the element, else if the text is empty ("") it'll delete the current text, otherwise it'll swap the current text of the specific field with the new one. For the API and ReqType parameters, the approach is slightly different since these elements are select dropboxes.

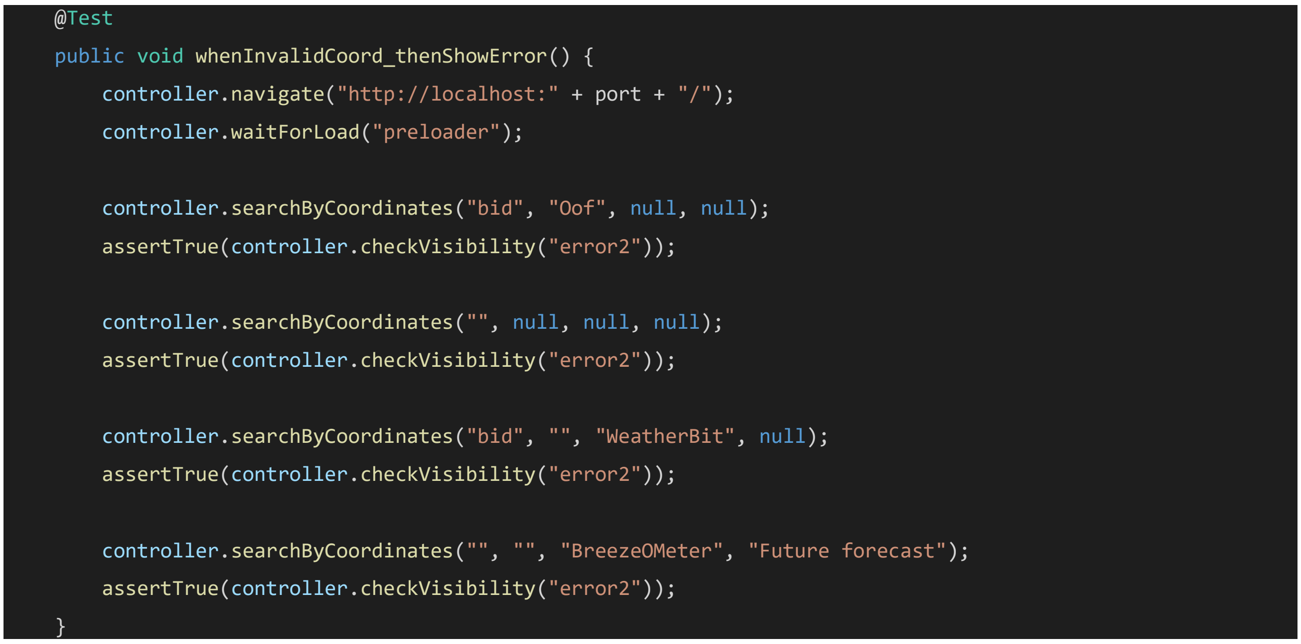
Now, as for the **InterfaceTests.java** class, it, technically consists in an integration test class ince it starts the SpringBoot context, initiating a web environment to boot start our app into. Originally we opted for a different approach, making it so no context was initiated by our test itself and opting by, first fully starting our application (running the AetherApplication.java class), and then, while it was running in the background, we ran the tests. This wasn't optimal however, as with this methodology our test wouldn't be runnable alongside the other tests since during the **mvn test** phase, unless specifically stated by the test itself, no context is initiated. So we changed the class to include the @SpringBootTest annotation, specifying the AetherApplication.class and SpringBootTest's web environment (using a random port). This random port is stored into the internal *port* variable which is then used when specifying the URL our PageObject controller should connect to (http://localhost: + *port*). This class has a routine that is used before the start of each test where we instantiate a new PageObject object in order to reset our driver's browser and it's elements, and a routine that is ran after every test is complete where we tear the current PageObject's driver connection.

Test set wise, we have over 25 tests that attempt to assert all possible behaviours and actions a user may perform on our website, be them actions that result in the correct interaction with our API or ones that cause an error to be presented, such as trying to insert non numeric values into the latitude and longitude form fields, or trying to add a new locale without specifying it's name. We also assert some other inner workings such as verifying that when there are no locales the locale selection dropdown (used to get the air quality by locale) is disabled and not clickable, as should be seen by the user.



```java
    @Test
    public void whenValidCoord_thenGetInfo_Current() {
        controller.navigate("http://localhost:" + port + "/");
        controller.waitForLoad("preloader");

        // No API
        controller.searchByCoordinates("8", "15", "Either API", null);
        assertFalse(controller.checkVisibility("error2"));
        // Wait until its done loading
        controller.waitForLoad("postloader2");
        assertTrue(controller.checkVisibility("searchDiv2"));

        // Weather
        controller.searchByCoordinates("50", "15", "WeatherBit", null);
        assertFalse(controller.checkVisibility("error2"));
        // Wait until its done loading
        controller.waitForLoad("postloader2");
        // Verify we loaded either an error, or a result with the WeatherBit title
        assertTrue(controller.checkVisibility("searchDiv2"));
        assertTrue(controller.checkText("search2Api", "WeatherBit Results"));

        // Breeze
        controller.searchByCoordinates("8", "40", "BreezeOMeter", null);
        assertFalse(controller.checkVisibility("error2"));
        // Wait until its done loading
        controller.waitForLoad("postloader2");
        // Verify we loaded a result with the BreezeOMeter title
        assertTrue(controller.checkVisibility("searchDiv2"));
        assertTrue(controller.checkText("search2Api", "BreezeOMeter Results"));
    }
```

Figure 43. A test that asserts that when valid inputs are given, our interface then shows the results retrieved from our API. Note the interaction with our PageObject instance, as well as how we verify that the interface's error message (displayed when bad inputs are given) is not visible, and that we wait until our response div is visible to assert it's contents.

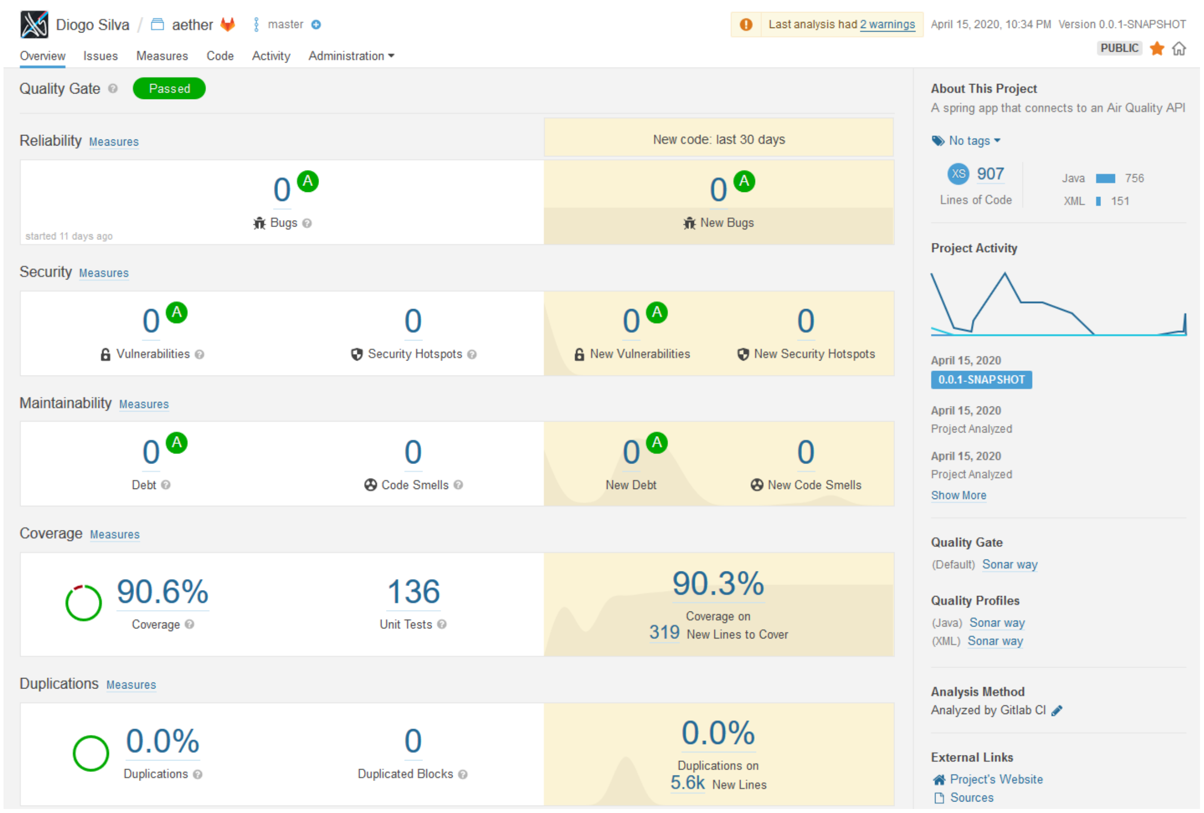


```java
    @Test
    public void whenInvalidCoord_thenShowError() {
        controller.navigate("http://localhost:" + port + "/");
        controller.waitForLoad("preloader");

        controller.searchByCoordinates("bid", "Oof", null, null);
        assertTrue(controller.checkVisibility("error2"));

        controller.searchByCoordinates("", null, null, null);
        assertTrue(controller.checkVisibility("error2"));

        controller.searchByCoordinates("bid", "", "WeatherBit", null);
        assertTrue(controller.checkVisibility("error2"));

        controller.searchByCoordinates("", "", "BreezeOMeter", "Future forecast");
        assertTrue(controller.checkVisibility("error2"));
    }
```

Figure 44. A test that checks that, when non numerical coordinates are given as input when searching using coordinates, an error message is shown in our interface

3.4 Static code analysis

For code analysis we integrated our GitLab repository with the **SonarCloud** service and included a stage in our CI/CD pipeline that would automatically call this static code analysis tool to generate a report on the newly pushed commit. It should also be stated that SonarCloud doesn't actually create code coverage reports by itself, instead these stats are generated by **JaCoCo**, which SonarCloud then gathers and displays in the overall report. It should also be specified that we assured that the overall grade of our code was at least passable by including the necessity to pass SonarCloud's Quality Gate when running the code analysis pipeline stage. If by any chance our new code didn't pass this requirement, the pipeline would actually automatically fail, forcing us to go see what the problem with our code was.


Diogo Silva / aether · master
Last analysis had 2 warnings · April 15, 2020, 10:34 PM Version 0.0.1-SNAPSHOT · PUBLIC
Overview · Issues · Measures · Code · Activity · Administration

Quality Gate: Passed

		New code: last 30 days	
Reliability			
0 A Bugs (started 11 days ago)		0 A New Bugs	
Security			
0 A Vulnerabilities	0 Security Hotspots	0 A New Vulnerabilities	0 New Security Hotspots
Maintainability			
0 A Debt	0 Code Smells	0 A New Debt	0 New Code Smells
Coverage			
90.6% Coverage	136 Unit Tests	90.3% Coverage on 319 New Lines to Cover	
Duplications			
0.0% Duplications	0 Duplicated Blocks	0.0% Duplications on 5.6k New Lines	

About This Project
A spring app that connects to an Air Quality API
No tags
XS 907 Lines of Code · Java 756 · XML 151

Project Activity
April 15, 2020 — 0.0.1-SNAPSHOT
April 15, 2020 — Project Analyzed
April 15, 2020 — Project Analyzed
Show More

Quality Gate: (Default) Sonar way
Quality Profiles: (Java) Sonar way; (XML) Sonar way
Analysis Method: Analyzed by Gitlab CI
External Links: Project's Website; Sources

Figure 45. One of the last reports generated by SonarCloud. Note the lack of code smells, duplicated code, bugs, duplicate lines and the over 90% code coverage. Note also how we passed the default Quality Gate and how the analysis method utilized was by integrating the toll with our GitLab CI pipeline

The importance of this tool to assure our code was at the very least decent cannot be overstated. Most pushes we did contained certain minor oversights that would have gone unnoticed if they hadn't been reported by SonarCloud in the form of Code Smells. Most of these were pretty harmless, but without this tool they would have kept pilling up and making our overall project a weaker product with bad or dangerous practices.

Another important measure we cared for was the code coverage. This was something that vexed us for a while since at the start, no matter what we did our coverage never seemed to actually improve by much. After analyzing the coverage report more in depth, however, we noticed that, whilst most of our classes had either 100% or over 90% coverage, some were actually sitting at 0%. These were the classes that had Lombok auto generated methods, which JaCoCo was picking up as not tested. After some configurations we managed to get around this for the Getter and Setter methods, but there was no feasible way to make it so the coverage report ignored Lombok's HashCode and Equals functions. Still, by checking out which logical paths weren't being tested in our main classes, such as the AetherRestController.java, we managed to effectively write tests that would explore every possibility and managed to increase our overall coverage to over 90% in the end.

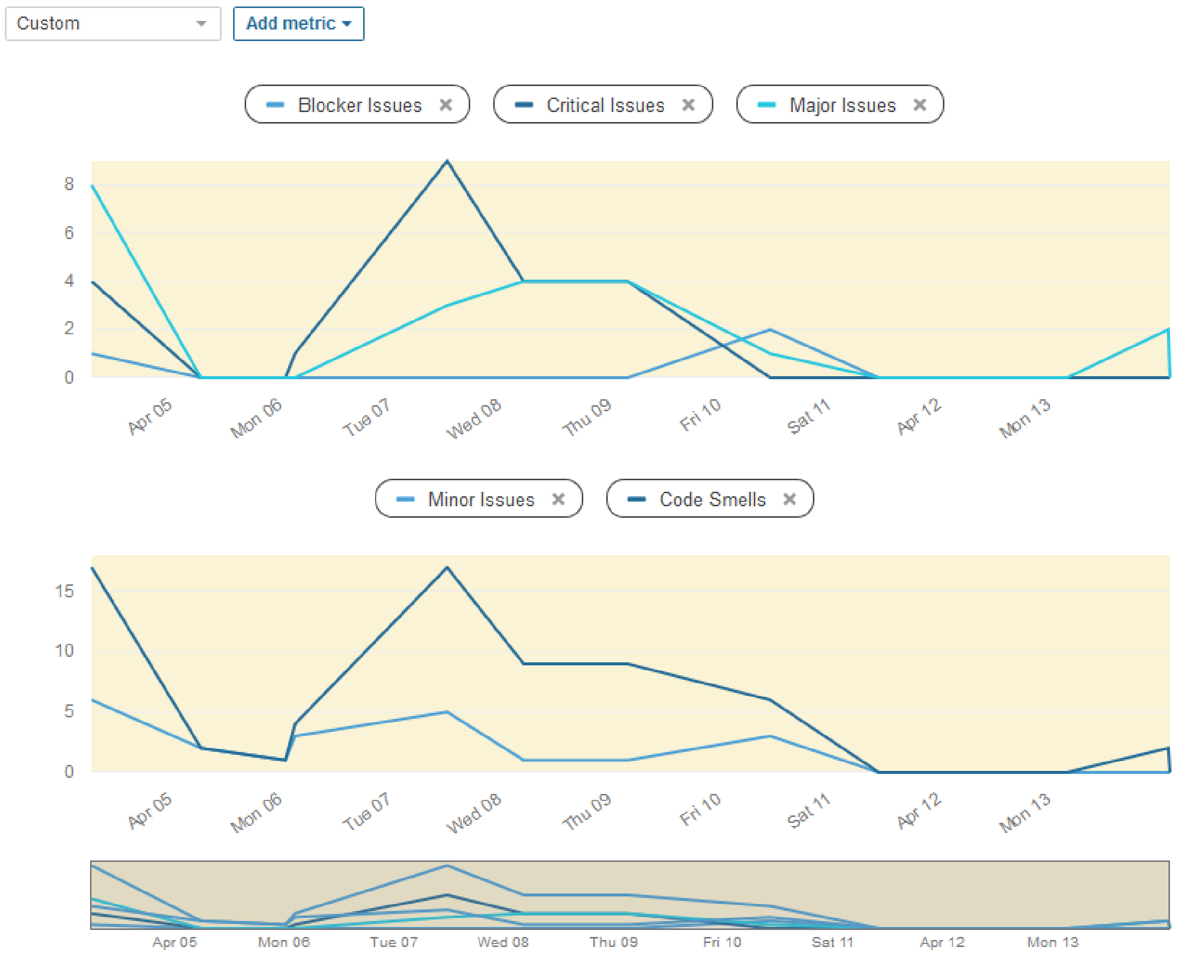
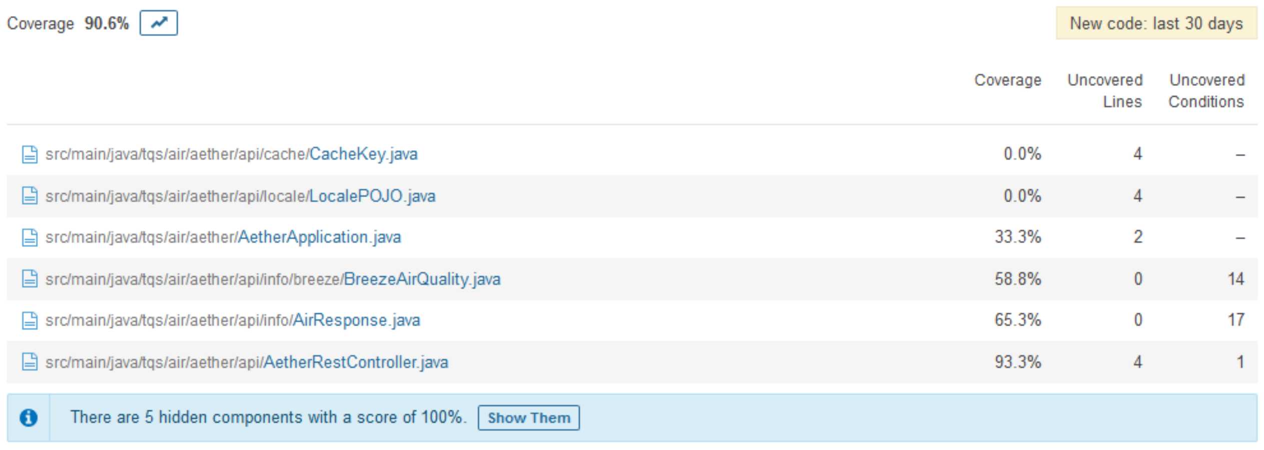


Figure 46. The evolution of amount of code smells (subcategorized into their several issue classes) in our code. If not for this tool's foresight we wouldn't have corrected or noticed practically any of these. Note how in the end we finished with 0 code smells and issues



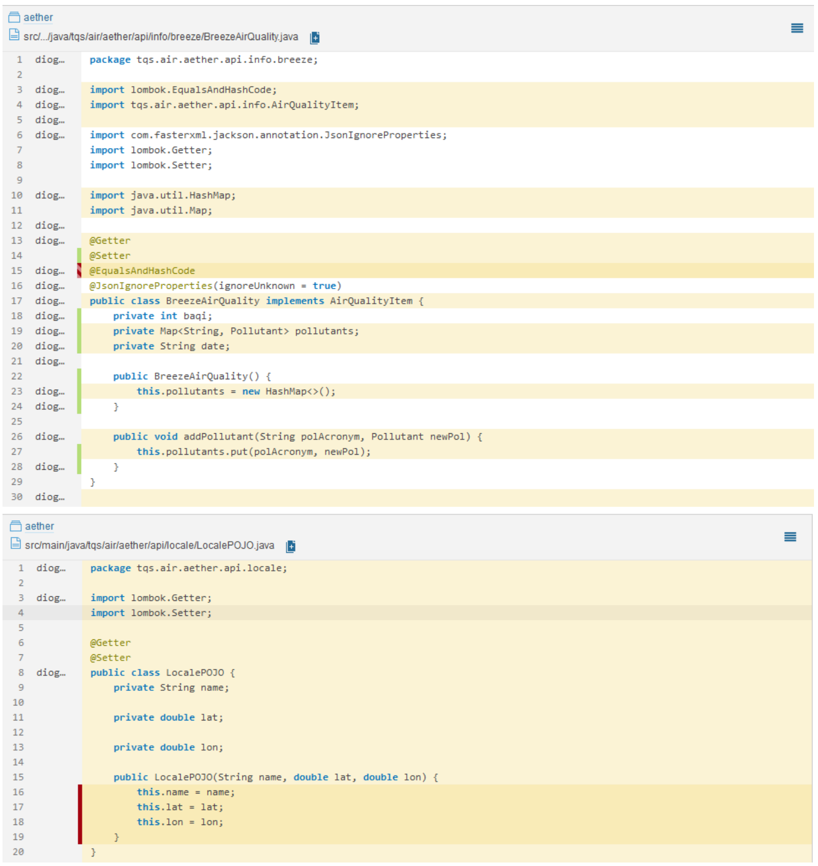
Coverage 90.6%

New code: last 30 days

	Coverage	Uncovered Lines	Uncovered Conditions
src/main/java/tqs/air/aether/api/cache/CacheKey.java	0.0%	4	–
src/main/java/tqs/air/aether/api/locale/LocalePOJO.java	0.0%	4	–
src/main/java/tqs/air/aether/AetherApplication.java	33.3%	2	–
src/main/java/tqs/air/aether/api/info/breeze/BreezeAirQuality.java	58.8%	0	14
src/main/java/tqs/air/aether/api/info/AirResponse.java	65.3%	0	17
src/main/java/tqs/air/aether/api/AetherRestController.java	93.3%	4	1

There are 5 hidden components with a score of 100%. Show Them

Figure 47. The line coverage per class on our project. The CacheKey and LocalPOJO classes show a 0% coverage, but this is because they possess no methods other than their constructor. We found that it would be redundant to make a test class exclusively for this as these classes are already instantiated in other tests, and if we did write them, we wouldn't be ascertaining nothing new, and would just be doing it to increase the overall coverage metric, hence defeating the purpose for tests. Other than that note how most of our classes possess a coverage of 100%



aether
src/.../java/tqs/air/aether/api/info/breeze/BreezeAirQuality.java

```java
package tqs.air.aether.api.info.breeze;

import lombok.EqualsAndHashCode;
import tqs.air.aether.api.info.AirQualityItem;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import lombok.Getter;
import lombok.Setter;

import java.util.HashMap;
import java.util.Map;

@Getter
@Setter
@EqualsAndHashCode
@JsonIgnoreProperties(ignoreUnknown = true)
public class BreezeAirQuality implements AirQualityItem {
    private int baqi;
    private Map<String, Pollutant> pollutants;
    private String date;

    public BreezeAirQuality() {
        this.pollutants = new HashMap<>();
    }

    public void addPollutant(String polAcronym, Pollutant newPol) {
        this.pollutants.put(polAcronym, newPol);
    }
}
```

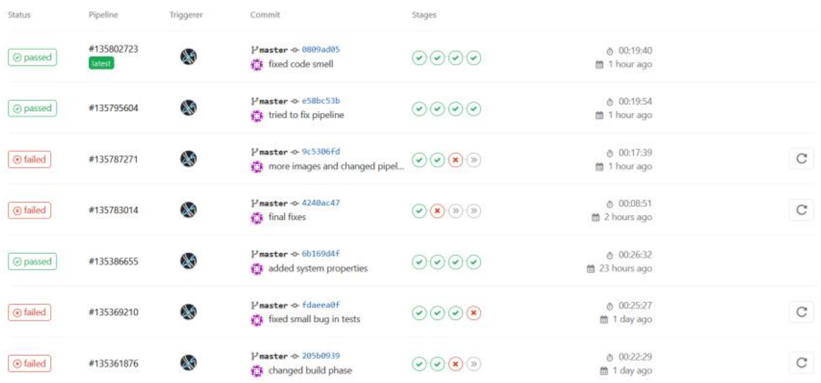
aether
src/main/java/tqs/air/aether/api/locale/LocalePOJO.java

```java
package tqs.air.aether.api.locale;

import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class LocalePOJO {
    private String name;

    private double lat;

    private double lon;

    public LocalePOJO(String name, double lat, double lon) {
        this.name = name;
        this.lat = lat;
        this.lon = lon;
    }
}
```

Figure 48. This figure illustrates an issue where the code coverage analysis doesn't detect that the hash code and equals methods have been generated by Lombok and counts them as not having been tested. It also shows how the LocalPOJO class (which was reported as having 0% coverage) contains only one constructor method.

3.5 Continuous integration pipeline

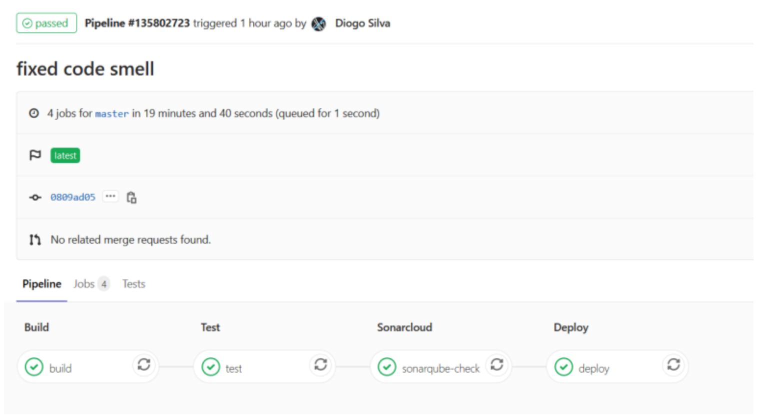
One of the first things we did after starting our project was define a pipeline that would automatically apply all of our tests and assure our code's quality after every push. The tool utilized to define our CI/CD pipeline was **GitLab CI**. At the start of it's creation, our **.gitlab-ci.yml** file (which is a markdown-like file used to define the pipeline's stages and jobs) defined a build, test and run stage where we, in this order, packaged, tested and then ran our application. After realizing that it wouldn't make much sense to run the application in the pipeline, we decided to remove that stage. We also changed the first Build stage from running the **mvn package** to using **mvn compile**, which is much more lightweight since all it does is compile our code, verifying that there aren't any build errors, without actually generating a .jar file like the package phase does. When it came time to integrate a static code analysis tool we also added a new stage that would automatically integrate our project with SonarCloud right after all tests had finished running.



Status	Pipeline	Triggerer	Commit	Stages	
passed	#135802723 latest		master 0809ad05 fixed code smell	✓ ✓ ✓ ✓	00:19:40 1 hour ago
passed	#135795604		master e58bc53b tried to fix pipeline	✓ ✓ ✓ ✓	00:19:54 1 hour ago
failed	#135787271		master 9c5306fd more images and changed pipel...	✓ ✓ ✗ »	00:17:39 1 hour ago
failed	#135783014		master 4240ac47 final fixes	✓ ✗ » »	00:08:51 2 hours ago
passed	#135386655		master 6b169d4f added system properties	✓ ✓ ✓ ✓	00:26:32 23 hours ago
failed	#135369210		master fdaeea0f fixed small bug in tests	✓ ✓ ✓ ✗	00:25:27 1 day ago
failed	#135361876		master 205b0939 changed build phase	✓ ✓ ✗ »	00:22:29 1 day ago

Figure 49. A showcase of our pipeline being ran, failing and succeeding

Towards the end of the project we also added a deployment phase to automatically deploy our application to an **Heroku** server when both the tests and the sonar code verification stages passed. This way our application would be accessible online and out for the world to use.


passed Pipeline #135802723 triggered 1 hour ago by Diogo Silva

fixed code smell

4 jobs for master in 19 minutes and 40 seconds (queued for 1 second)

latest

0809ad05

No related merge requests found.

Figure 50. An example of a successful pipeline where all stages succeeded

The final flow and stages of our pipeline are, then:

- **build**
  - Has 1 “build” job.
  - Runs the **mvn compile** command to ascertain if there are any build errors (i.e if our project can be correctly compiled)
- **test**
  - Has 1 “test” job.
  - Runs the **mvn test -Dtest=!InterfaceTests** command that automatically runs all of our written tests. The -Dtest flag is used to ignore the functional test present in the InterfaceTests class.
- **sonarcloud**
  - Has 1 “sonarqube-check” job.
  - Runs the **mvn verify sonar:sonar -Dtest=!InterfaceTests -Dsonar.qualitygate.wait=true** command. Note, once again the usage of the -Dtest flag to ignore the functional test, aswell as the Dsonar.qualitygate.wait flag that makes it so, if for some reason our code analysis’ quality gate fails, the stage will also fail and notify us.
  - Also includes a specification that makes it so this is only ran when our code is sent to our master branch.
- **deploy**
  - Has 1 “deploy” job.
  - Runs a bunch of commands that push our project into our Heroku repository, allowing it to be deployed.
  - Like the last job it also has a flag that makes it so this job is only ran when our code is merged to our master branch.

```
image: maven:3.6.1-jdk-11


variables:
    SONAR_TOKEN: "0cf615d408d5ff477cd673021ee872556bd255bc"
    SONAR_HOST_URL: "https://sonarcloud.io"
```

```yaml
before_script:
    - cd ./aether

stages:
  - build
  - test
  - sonarcloud
  - deploy

cache:
  paths:
    - target/

build:
  stage: build
  script:
    - mvn compile

test:
  stage: test
  script:
    - mvn test -Dtest=!InterfaceTests #Exclude functional tests

sonarqube-check:
  stage: sonarcloud
  script:
    - mvn verify sonar:sonar -Dtest=!InterfaceTests -Dsonar.qualitygate.wait=true
  only:
    - master

deploy:
  stage: deploy
  image: ruby:latest
  script:
    - apt-get update -qy
    - apt-get install -y ruby-dev
    - gem install dpl
    - dpl --provider=heroku --app=aether-tqs --api-key=8713231f-eb7b-435b-9786-000da5536d39
  only:
    - master

#run:
#  stage: run
#  script:
#    - mvn package
#    - mvn spring-boot:run
```

Figure 51. Our pipeline. Note the usage of a before script to move from our current directory to the one containing our code. That is necessary since the .gitlabe-ci.yml file has to be placed in the root of our project (rather than the root of our code). Note also how we used a maven image since that was all that was necessary to have our project run in (except for the deploy stage).

4 References & resources

Project resources

- Git repository: https://gitlab.com/HerouFenix/aether
- Video demo using the frontend page has been placed in .mp4 format in the Demo (Demo/demo.mp4) folder within the repository. It can also be accessed within the page itself on the video section.
- Ready to use application: https://aether-tqs.herokuapp.com/

Reference materials

For this project, most questions that appeared during development were answered by the tutorials present in https://www.baeldung.com/. More concretely, some of the most important and used ones were:

- https://www.baeldung.com/spring-scheduled-tasks
- https://www.baeldung.com/building-a-restful-web-service-with-spring-and-java-based-configuration
- https://www.baeldung.com/spring-testing-scheduled-annotation
- https://www.baeldung.com/maven-java-version
- https://www.baeldung.com/spring-data-derived-queries
- https://www.baeldung.com/maven-goals-phases

As for the external APIs, the documentation for both BreezeOMeter and WeatherBit can be found at https://docs.breezometer.com/air-quality-api/v2/ and https://www.weatherbit.io/api respectively.

Other documentation sites were utilized such as:

- https://docs.spring.io/spring/docs/current/spring-framework-reference/index.html
  - For the Spring framework and some of it's annotations
- https://github.com/awaitility/awaitility/wiki/Usage
  - For the Awaitility tool
- https://docs.sonarqube.org/latest/analysis/gitlab-cicd/
  - For integrating SonarCloud with the Gitlab CI
- https://docs.gitlab.com/ee/ci/
  - For writing the CI/CD pipeline
- https://devcenter.heroku.com/categories/reference
  - For deploying our app into Heroku