HW1: Mid-term assignment report

Tiago Leon Melo [89005], v2020-04-08

1 Introduction **1**
1.1 Overview of the work 1
1.2 Limitations 2

2 Product specification **2**
2.1 Functional scope and supported interactions 2
2.2 System architecture 2
2.3 API for developers 4

3 Quality assurance **5**
3.1 Overall strategy for testing 5
3.2 Unit and integration testing 5
3.3 Functional testing 6
3.4 Static code analysis 7
3.5 Continuous integration pipeline 8

4 References & resources **9**

1 Introduction

1.1 Overview of the work

In order to develop a maintainable code-base or agile and prone-to-change product, quality coding standards and benchmarks must be ensured and implemented. For this reason, a number of frameworks and libraries has been developed as well as coding practices to help developers.
This project aims to use some of these frameworks and coding practices when developing a real-world small product in order to demonstrate and study how they can be used, what are some of their caveats and so-forth. This report will define the project developed, briefly explain the system architecture behind it and expand on the strategy behind the coding practices used. Furthermore, it is also the goal of this report to justify choices made as a developer when testing, what sorts of tests were performed and how already existing tools were used to develop such. Lastly, there will be an explanation on how Continuous Integration was implemented in the development of this project.
AirQ is a Spring application that allows users to fetch air quality for a specific city given it's name as well as its past air quality values. At the time of writing, AirQ fetches the following air attributes: PM10, CO levels, Ozone levels and the city's air quality index (AQI). The app also supports a caching mechanism which allows it to return recent results on a given city without re-doing a API calls.

1.2 Limitations

At the time of writing, AirQ does not support fetching air quality by coordinates.The app also does not support customization of values returned, but it has been developed with that feature in mind, so it can possibly be implemented in a feature release.

2 Product specification

2.1 Functional scope and supported interactions

"An adult breathes 15,000 liters of air every day. When we breathe polluted air pollutants get into our lungs; they can enter the bloodstream and be carried to our internal organs such as the brain. This can cause severe health problems such as asthma, cardiovascular diseases and even cancer and reduces the quality and number of years of life. (New evidence even suggests that every organ in the human body is harmed.) Vulnerable groups, namely children, people with chronic diseases, and the elderly, are particularly sensitive to the dangerous effects of toxic air pollution." [source]

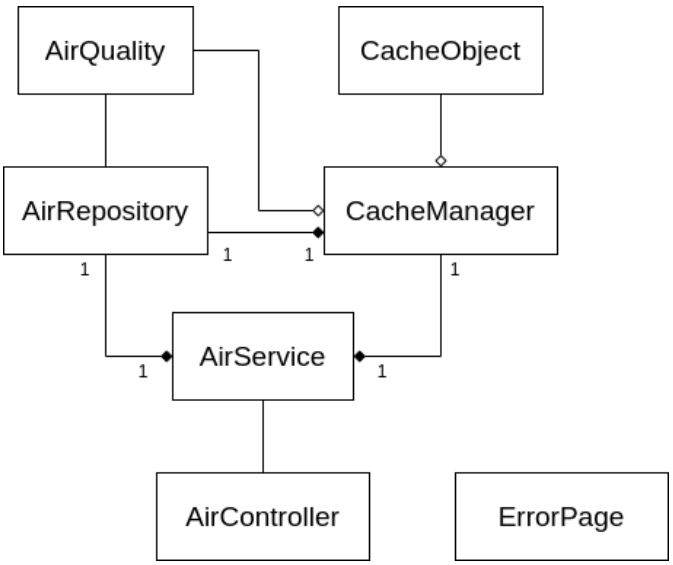
Without a doubt, a city's air quality is a strong factor to have in mind when evaluating it. This application targets any user who wishes to attain information about the air quality of a given city for the most diverse reasons.
From a mainstream user's point of view, some of these might be to perform a well-informed decision on whether or not to move to a new city or to plan the next family vacation. All they need to do is to access the AirQ's website and type in the name of the desired city and the aforementioned air quality data should be displayed within seconds.
To an enterprise, the data made available by AirQ's API can prove to be of extreme value. Companies from areas ranging from Data Science to weather prediction can make use of the information provided by this application, possibly to correlate certain pollutants with a specific disease rate on a given city or for more accurate weather predictions.
That being said, AirQ focuses on serving these two types of users: an occasional, **mainstream client** that needs sporadic access to a city's air quality information and **enterprises** who need access to large amounts of data.

2.2 System architecture

<briefly present the software architecture. Include diagrams.>
<explain the supporting data models/data structures, i.e., the entities of your problem>
<detail the specific technologies/frameworks that were used>
Being a Spring application that runs on a single machine, AirQ was developed in accordance to the MVC pattern. That being said, the source code directory is divided into 4 other sub-directories:

- **Controller** – contains the **AirController** class, responsible for mapping most of the endpoints the application uses (/api and /search) and a fallback **ErrorPage**, which is triggered upon server errors. By making use of Spring annotations, such as @GetMapping, HTML pages are pinpointed to specific endpoints. These are rendered in Thymeleaf with the Model object passed on as argument, with which it is possible to "send" backend data, like the AirQuality object.

- **Services** – contains the classes responsible for AirQ's business logic. This includes the module responsible for the external API calls and the caching mechanism as well as the relevant entities for the latter to function properly. Since these aren't exactly entities *per se* in our problem's context, they were not stored in the "entities" directory.
    - The class **AirService** exposes a clean interface for easy endpoint mapping to be done by the controller, with methods that allow fetching the cache's metadata (such as time-to-live, number of hits and misses and requests), getting relevant data for a given city, which returns an AirQuality object, present in the entities directory and fetching historical data on a given city, which is returned as a Map with timestamps as keys and AirQuality instances as values, using Breezometer's API, since Ambee's endpoint for this feature seemed to not work correctly. The application however, does not cache results from the historical endpoint.
    - A **CacheObject**, managed by the CacheManager, consists of a class that encapsulates the aforementioned relevant cache metadata. This is the entity stored in the repository, with a city name as ID. By persistently storing this object, we can fetch and tweak the data at runtime. For instance, when a search is made for city A for the first time, a new CacheObject is instanced for city A, with default values on hits, misses and time to live (30 seconds). From that moment on, every time a request is made for city A, AirQ will update it's fields: if it is a miss, add 1 to this field and expand the time to live (if the CacheObject wasn't available when called, perhaps by extending it's lifespan we can improve future experiences) and if it is a hit, add 1 to this field and reduce the time to live, following the same line of thought as before. By persistently storing them we can ensure these tweaked lifespans "live on" after the cache objects expire, so that the next time a request for city A is made and there is no instanced cache, we need only fetch values.
    - The **CacheManager** class was implemented in a multi-threaded fashion. It contains two Maps, with city names as keys. One maps each city to a CacheObject, in order to have in-memory access to metadata and the other maps the string to an AirQuality object, the actual cached data. Once cache is instanced, a thread is started that will periodically check if the last time the cache object was accessed plus it's time to live are greater than the current time. If they are, cleaning is done on both maps. The rest of the class consists of mostly getters and setters and variations of such, in order to gain easier access to keys and values.
- **Repository** – contains a single class responsible for the interaction with the persistence layer. For such, AirQ makes use of the interface provided by JPA. As explained before, the single entity stored in the repository is the CacheObject class.
- **Entities** – contains the POJO related to the context of the problem: AirQuality. This class encapsulates relevant data to identify the city it refers to, such as the city name and the country. Furthermore, like explained before, this class was developed bearing in mind potential scalability concerns. Hence, instead of containing static attributes for the data it collects, it contains a Map that allows this class to dynamically store different attributes. By mapping each field it collects to the value, we can easily adjust and append new data to gathered air quality. This mechanism was a suggested feature in a previous project, for the subject of IES. (Plant Aware)



2.3 API for developers

AirQ's API exposes two endpoints. The documentation for these can be accessed on the "DOCS" tab on the website, which was developed using the Swagger UI and openapi libraries.

- /api/{cityName} – get the air quality for a city of name passed on as argument. Returns a JSON with fields "country", "city" and "attributes" where "attributes" is a nested JSON containing the gathered labeled data.

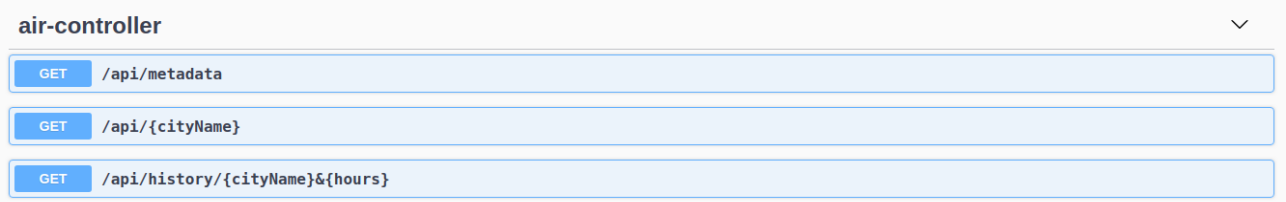
```
{
  "country": "PT",
  "city": "Tondela",
  "attributes": {
    "OZONE": "52.2",
    "PM10": "14.12",
    "AQI": "48.0",
    "CO": "0.63"
  }
}
```

- /api/metadata – get AirQ's cache data for each city queried during current runtime.

```
[
  {
    "city": "Tondela",
    "hits": 0,
    "misses": 1,
    "requests": 1,
    "ttl": 30,
    "lastAccess": 1586338396388
  }
]
```

- /api/historical/{cityName}&{hours} – get historical air quality data obtained in the last number of hours (must be between 0 and 168)


```
{
   "2020-04-08T12:00:00Z": {
     "country": "PT",
     "city": "Tondela",
     "attributes": {
       "OZONE": "31.29",
       "PM10": "9.4",
       "AQI": "75",
       "CO": "122.86"
     }
   },
   "2020-04-08T14:00:00Z": {
     "country": "PT",
     "city": "Tondela",
     "attributes": {
       "OZONE": "32.36",
       "PM10": "10.38",
       "AQI": "74",
       "CO": "177.03"
     }
   },
   "2020-04-08T13:00:00Z": {
     "country": "PT",
     "city": "Tondela",
     "attributes": {
       "OZONE": "33.45",
       "PM10": "13.63",
       "AQI": "74",
       "CO": "155.3"
     }
   }
}
```



Screenshot 1: Swagger UI generated endpoint documentation

3 Quality assurance

3.1 Overall strategy for testing

The followed test development strategy for this project was TDD. By writing the tests before developing classes, we can have an unbiased opinion of what the class should do, rather than what it does, which can easily happen when no approach is followed.
Once again, the test directory is split in three sub-directories, one for each "class" of tests performed. A total of 19 tests was developed, and it was ensured they all passed.

```
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 19, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  01:36 min
[INFO] Finished at: 2020-04-08T17:49:47+01:00
```

Screenshot 2: Test results

3.2 Unit and integration testing

The "unit" tests directory contains 4 unitary test classes on some of the objects developed, such as the AirController, AirRepository, AirService and CacheManager. AirQuality and CacheObject were not tested, since they consisted mostly of the automatically generated getters, setters, equals and hashcode methods and contained little to no relevant functionalities.
In the **AirRepositoryTest** no mock ups were needed. Two unit tests were developed to test if CacheObjects were being saved and returned. In order to do this, a TestEntityManager provided by the JPA library saved persistently a CacheObject on the repository which was then retrieved by the subject under test.
On the second test, both operations were performed by the subject under test, the repository.

```
CacheObject savedCache = entityManager.persistAndFlush(new CacheObject("Viseu", 0, 0,
```

Snippet 1: Making use of the entity manager to save objects on the repository

In order to test the cache managing mechanism 5 unit tests were developed on the **CacheManagerTest** class:

- `whenAddToCache_cacheSizeIsOne()` - test if values are being inserted and if size is being updated properly

- `whenAddToCache_cacheContainsAdded()` - test if the inserted value corresponds to the value being inserted and can be identified as "contained" in cache
- `whenAddToCache_addedElementIsCorrect()` - test if the retrieved value corresponds to the value inserted
- `whenTimePasses_cacheExpires()` - this test was originally performed with "Thread.sleep()", however SonarQube identified this as a major code smell. As such, this test forces the cleanup of the cache and checks if the inserted cache object is removed.
- `whenSameKeyInsert_updateOccurs()` - test if when the same city is stored twice within the cache, the values associated with it are updated. This can be useful in possible future version in case an API call should be forced.

In the **AirControllerTest** test, thanks to the @WebMvcTest annotation we were able to Mock a servlet that simulated requests to our application. Additionally, since this class depends on the AirService, the latter was also mocked using Mockito and then "programmed" to return a preset of AirQuality. Afterwards, the test performed a GET request on the target API endpoint and compared the obtained city value with the one on the preset. Another developed test on this class was meant to test historical data fetching. For such, the mocked AirService was programmed to return a single timestamped entry when queried for "Coimbra" and the timestamp values were compared with the expected and the actual. (View Snippet 3)

```
@WebMvcTest
Run Test | Debug Test
public class AirControllerTest {

    @Autowired
    MockMvc servlet;

    @MockBean
    AirService airService;

    @Test
    Run Test | Debug Test
    public void whenGetAirForCity_thenReturnAirQuality() throws Exception {

        // "Programming" the mock
        given(airService.getAirForCity("Coimbra")).willReturn(new AirQuality("PT", "Coimbra")
                .putAttr("PM10", "18.97").putAttr("CO", "0.63").putAttr("OZONE", "69").putAttr("AQI", "

        // Testing a GET on a given API endpoint
        servlet.perform(MockMvcRequestBuilders.get("/api/coimbra")).andExpect(status().isOk())
                .andExpect(jsonPath("city").value("Coimbra"));

    }
}
```

Snippet 2: Using WebMvc to isolate dependencies and mock behaviour

```
@Test
Run Test | Debug Test | ✓
public void whenGetHistoricalDataForCity_thenReturnHistoricalData() throws Exception {

    // Programming the mock
    HashMap<String, AirQuality> hm = new HashMap<>();
    hm.put("123", new AirQuality("PT", "Coimbra")
    .putAttr("PM10", "18.97").putAttr("CO", "0.63").putAttr("OZONE", "69").putAttr("AQI", "91"));

    given(airService.getAirHistoryForCity("Coimbra", 1)).willReturn(hm);

    // Testing a GET on a given API endpoint
    servlet.perform(MockMvcRequestBuilders.get("/api/history/coimbra&1")).andExpect(status().isOk(
            .andExpect(jsonPath("123").exists());
}
```

Snippet 3: Comparing timestamps in order to verify returned historical data

Lastly, on the **AirServiceTest** it is tested if when a request is made to a random city, the city queried on the third party API and the results correspond to the original city. Since AirService depends on the AirRepository, mocks were created and then injected on the subject under test. Furthermore, it is also tested an utility function that retrieves coordinates and country given only a city name. This is done with a set of hardcoded values compared with a String, since the return value of the getCoordinates method is a String formatted to fit "lat/long/countryCode". Additionally, it is tested if when retrieving historical data the number of entries obtained is lesser than or equals the number of hours. This is due to the fact that some times data is unavailable, but I chose to display the, although incomplete, results. Lastly, a test to check if each entry corresponds to the same city.

```
@Mock
AirRepository airRepository;

@InjectMocks
AirService sutAirServiceSut;

@Test
Run Test | Debug Test
public void whenGetCoimbraStats_returnCoimbraStats() throws UnirestExcept
{
```

Snippet 4: Using mocks to instance an AirService object

As to service level tests, the ServiceCacheRepoIntegrationTest class was developed in order to test the interactions between the Service, Cache and Repository.

- whenHit_hitIsRecorded() - test if hits are working and being updated for cache objects
- whenMiss_missIsRecorded() - test if misses are working and being updated for cache objects

3.3 Functional testing

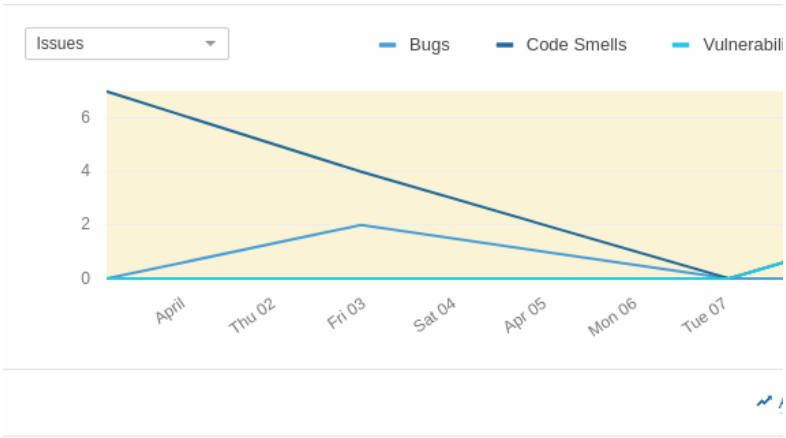
Web interface functional testing was implemented using the ChromeDriver for Selenium. Furthermore, for better code readability, a Page Object was developed to provide a clean interface for testing. The tests themselves consisted of testing if results existed when the page was first loaded (which they should not), test if a results exist when a known city is queried and test if the right results are displayed. For the latter test, an AirService object was instanced and an API call forced, to compare results.

```
public void search(String query)
{

    driver.findElement(By.id("citySearch")).click();
    driver.findElement(By.id("citySearch")).sendKeys(query

    driver.findElement(By.id("discoverButton")).click();

}
```

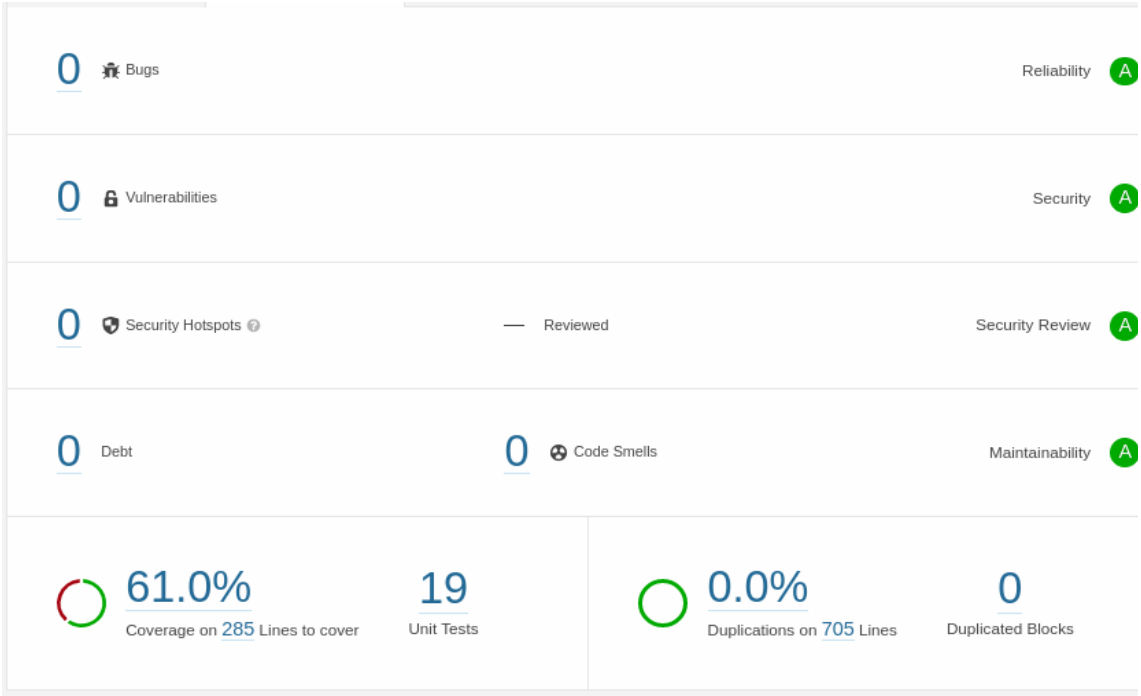
Snippet 5: One of the methods exposed by the interface made available by following the Page Object pattern.

3.4 Static code analysis

The two main tools used to perform static code analysis were SonarQube and JaCoCo. Throughout the development of the project, at stable releases static code analysis was performed by invoking the *sonar:sonar* maven goal. By reducing code smells, bugs and security hotspots on these iterations, we prevented expensive debt time and kept the code maintainable and prone to change.



Screenshot 3: Evolution of bugs, code smells and security hotspots throughout development



Screenshot 4: SonarQube dashboard on the last iteration

airq

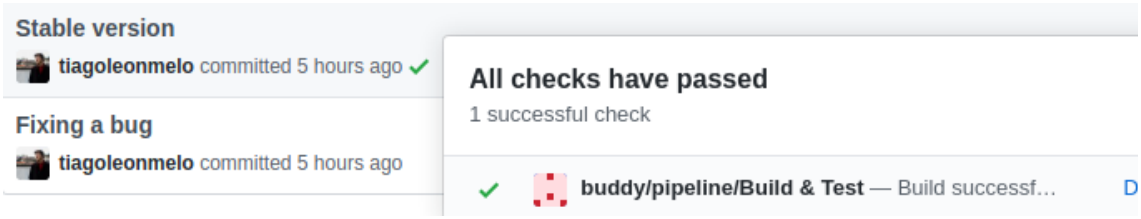
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed
deti.tqs.airq.services		80%		58%	25	62	39	209	7	38	0
deti.tqs.airq.entities		20%		3%	20	26	34	46	5	11	0
deti.tqs.airq.controller		52%		33%	10	15	16	27	8	12	1
deti.tqs.airq		33%		n/a	1	2	2	3	1	2	0
Total	407 of 1,301	68%	53 of 84	36%	56	105	91	285	21	63	1

Screenshot 5: JaCoCo coverage report

From the SonarQube dashboard and JaCoCo's coverage report, we can easily spot the greatest problem in our project would be from lack of code coverage. However, JaCoCo inspects coverage on automatically generated methods such as getters and setters which, as it has already been discussed, were not tested.
A particularly interesting security hotspot found when developing would be the use of RequestMapping with an unspecified HTTP method. According to SonarQube, "a method you intended only to be POST-ed to could also be called by a GET, thereby allowing hackers to call the method inappropriately." which, albeit sounding obvious, I had not realized.

3.5 Continuous integration pipeline

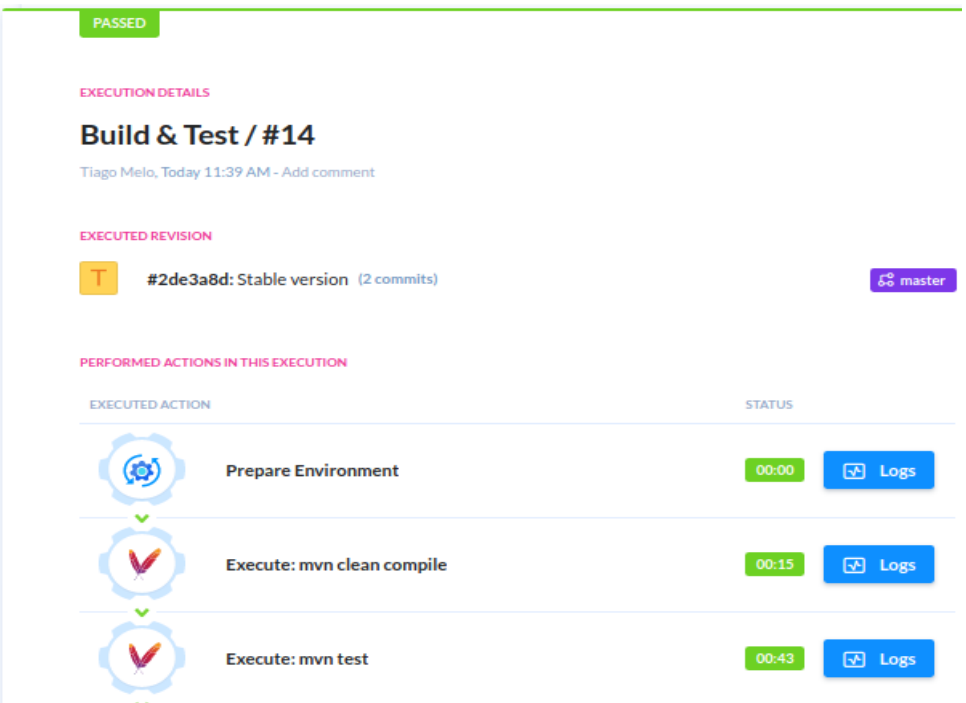
The continuous integration framework used to implement a CI pipeline in this project was Buddy. Albeit being a paid service, Buddy offers a 7 day free-trial, which was enough to set up a pipeline and experience continuous integration.
This framework offers integration with Github, the SCM platform used. This allowed to set up the pipeline to be triggered on every push and display the build results on Github, which also helped in keeping track of integration status.



Screenshot 6: Buddy integration with GitHub

The pipeline itself consisted of 4 actions:

- Preparing the environment: this was an implicit default action, that was built in from the moment a "Maven" pipeline was chosen. Essentially, Buddy is building an environment that supports Maven's dependencies and the ones declared in the pom.xml file.
- Executing `mvn clean compile`
- Executing `mvn test`
- Executing `mvn install`
  By separating each Maven goal in distinct actions, we can easily understand what went wrong and where. It is worth mentioning, however, the pipeline was not configured to work with Selenium, which made it impossible to execute functional tests. So, recent builds are displayed as failing in Buddy despite working locally.



4 References & resources

Project resources

- Git repository: https://github.com/tiagoleonmelo/airq
- Video demo: /demo.mp4

Reference materials

- APIs:
  - Ambee
  - Breezometer
- Libraries and frameworks: JUnit, Selenium, Mockito, JaCoCo, Swagger UI, openapi, SonarQube, Spring, Thymeleaf, Buddy