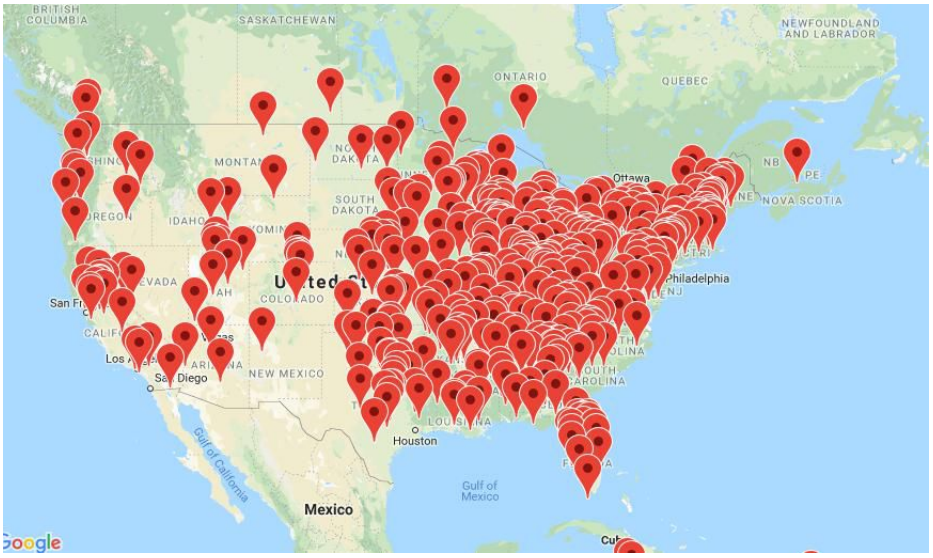
Test Final Assignment

Gutenberg Application

CPHBusiness - 25/05-2018

By: *Marco Blum, David Blum, Alexandar Kraunsøe & Kasper Roland Pagh*.



Alex Kraunsøe, David Blum, Kasper Pagh & Marco Blum

Introduction

This report will go into depth regarding how we tested and developed our platform, which can be found here: http://167.99.237.199/
The code can be found here:
https://github.com/kasperpagh/TestFinalAssignmentAllCodeGroupSix
It should be noted that the map api does not work on the live server. Google map api will not deem it as secure enough (because we do not have ssl) and it will fail. The map api does work on local host, and it will use the live data.

Agile principles

We proceeded with customer satisfaction as our highest priority. We did this using test driven development to make better functional and working software, which we could use as our primary measure of progress. We split up the group in two where some handled the testing and interface creation, where the others took care of the data extraction. This made it so in case a potential customer would oversee the progress they would both have sample data and a sample backend even by the end of the first sprint.

During the sprints we often consulted each other of different areas of the platform, ex. when someone was making the interface and tests for the backend, they would consult the person handling the data extraction to be informed on what data they could expect to stored in the database, even before any database was live.

We mostly communicated by voice chat and teamviewer (a screen sharing program we use). We did this to save time on commuting giving us more work time. But during the entire development we had a chat channel, where everyone was always allowed to speak up so the rest of the group could discuss whatever problem they were facing.

When a section of the work was done, a developer could always consult the our scrum board (hosted on Trello, more on that later) to find other areas that were in need of work, so that nobody got stuck with nothing to do.

By doing this we got through all the agile principles except the two following ones:

2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

We did not change any requirements throughout the project and could therefore not fulfill it even if we wanted too.

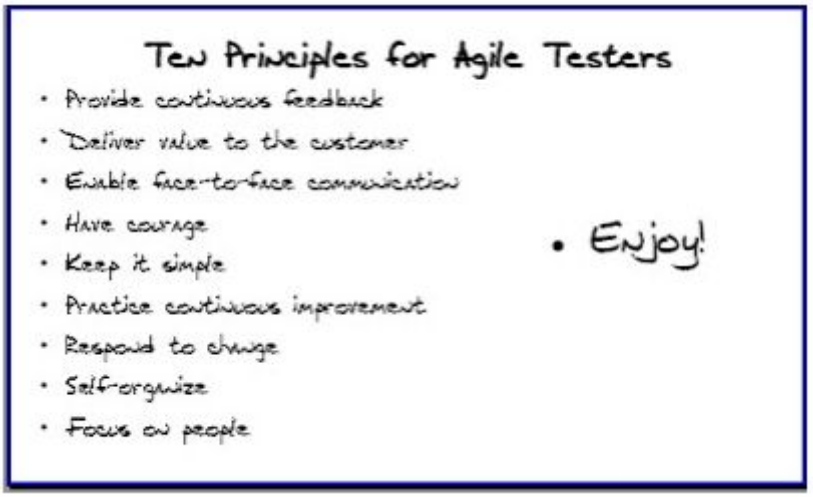
4. Business people and developers must work together on a daily basis throughout the project.

Even though the developers worked together at multiple parts of the platform and at different areas, We did not have any business people to work with (only our “mock product owner”).

Difference in Testing Methodology

In the regular development such as the waterfall model, testing is often done once, and often very late in the development cycle.
In Agile development, testing is a much more active part of the development plan. Testing is done at the beginning of the project and with Test Driven Development you even write most of the tests before the actual logic of the code. When something new is added regression testing will always follow

In essence agile testing differs from regular development testing, that often occur after the build process, whereas the agile testing uses it as a driving/exploratory force during the entire development.



Team Logistics

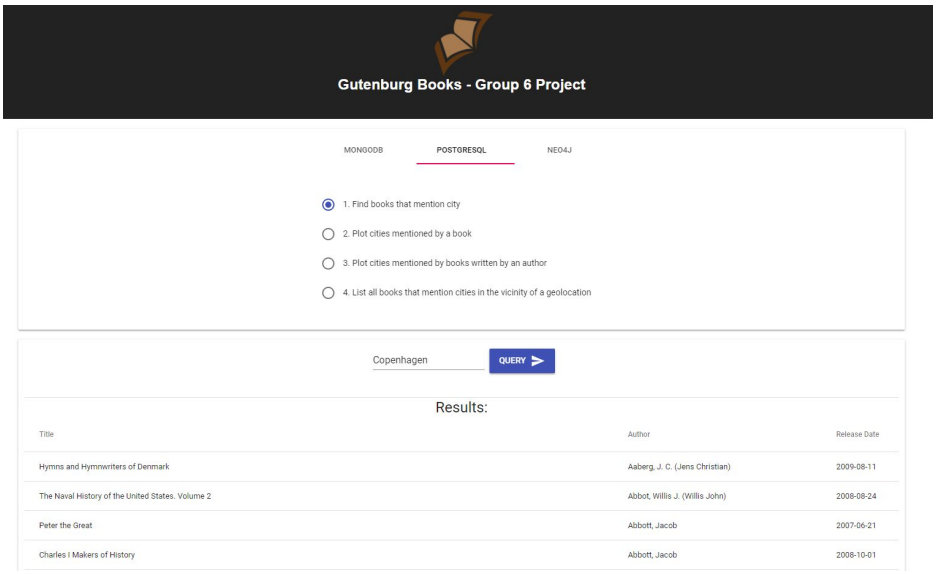
Our team has worked together on many projects over the course of our education, making us very familiar with each others strengths and weaknesses. This meant when we delegated the workload we knew who wanted to do what.
Marco and David both wanted to work with the domain, setting up docker images, setting up the databases, and setting up a jenkins server for the project.
Kasper and Alex mainly worked on the backend writing the code and tests. This however did not mean that is all those people did.
Everyone in the group helped in all aspects. Everyone helped setting up the domain whether it be a digitalocean (our cloud host) droplet (digital oceans name for a VM in the cloud), jenkins server or the databases. And everyone helped working on the code whether it be frontend or backend.

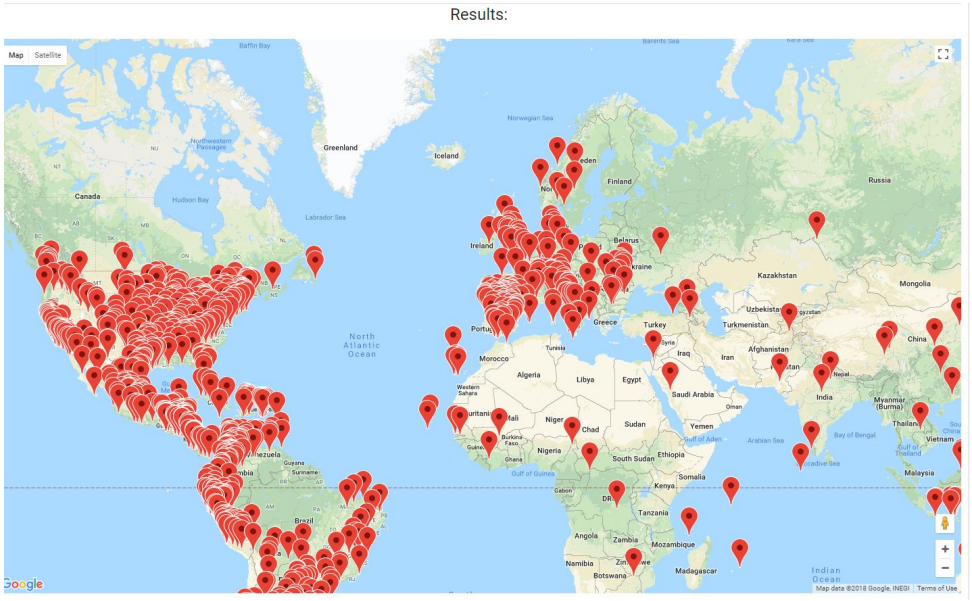
One way we handled bug tracking was using a jenkins server. As the jenkins server utilizes continuous integration and doesn’t push anything to the production server unless all tests pass and no bugs are found.

This allowed us to work as we pleased and once we were done jenkins could take over and push to production. On top of this we have a telegram chat. This telegram chat is an easy way for us to contact everyone incase we find bugs or other issues with our code. Telegram helped us significantly whenever and issue arose as we could get everyone involved with it quickly thus fixing issues as they came up.

Nonfunctional Requirements

As for the nonfunctional requirements for the project we set up a React.js frontend using the Google Javascript Map API to load the maps. For the frontend we made simple component tests for the individual components.





We also setup a Jenkins server for continuous integration and delivery. The Jenkins server was setup with two build tasks one for the backend and one for the frontend. The tasks built the application, ran all the tests and deployed it to the production server if all tests passed. This pipeline is initiated by the master branch on github being pushed to. For development

we would work on branches, when the feature was done we could merge to master and the build and deployment would happen automatically.

Project planning

Before we even wrote the application's first line of code, there was a myriad of things we had to agree on. Specifically we wanted to establish the following before we started:

- A shared project methodology (such as Scrum, Waterfall etc.)
- A shared communication platform/board for posting stories, tasks and general information.
- The roles of different team members (described in *Team Logistics*)

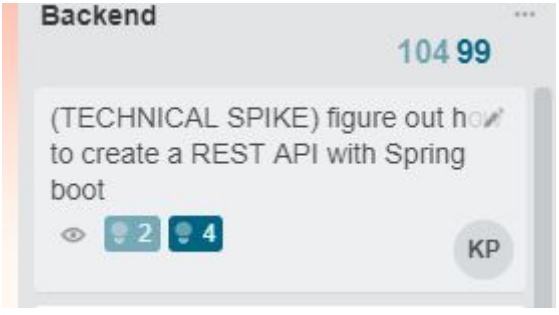
In the end, as per usual, we ended up using a variation of scrum, with a couple of XP practices sprinkled in for good measure.

Scrum Variation

In our version of Scrum we decided to do away with some of the more wasteful artifacts and practices usually associated with an agile scrum process.
In particular we did not see the need to formally include the following:

- We did not use daily Scrum meeting
- We did not do regular spring planning, instead we decided to consider the whole project as one big spring for convenience (largely due to the scope of the assignment).

Besides the above points our process was pretty much standard Scrum, with individual user stories broken down into manageable tasks, a shared sprint board where we can track each others progress etc.
As mentioned above we also used a number of XP practices during the development of the Gutenberg application. Namely we really liked the use of ***technical spikes*** since, for us anyways, this project had a lot of unknowns in the beginning.



An example of a spike card on our shared board

Besides that we also tried to embrace the XP practice of *Collective code ownership*, which means that every member of the team must have at least some insight into the entire application, even if they only actually worked on the API (for example).

Furthermore we also wholeheartedly embraced the practices of *continuous integration* (CI) and *Test Driven Development* (TDD).
The TDD part was relatively straight forward, in the sense that we simply started out by documenting all “important” functionality with unit tests

```
mongoConnectorTestObject = mock(I_MONGOConnector.class);
psqlConnectorTestObject = mock(I_PSQLConnector.class);
neoConnectorTestObject = mock(I_NEO4JConnector.class);
```

An example of a TDD test, where we mocked the individual database connectors

In terms of the CI part we achieved this by using a combination of *Maven* and *Jenkins* where, everytime we pushed a change in the code, the jenkins server would pull the code and then use Maven to ensure all tests are running properly.

If the test did indeed pass, Jenkins would then proceed to automatically push the new code to the production server and construct and deploy a docker image of the backend.

Test Strategy

We started the project with the one goal of wanting to do the whole thing using TDD as much as we possibly could. None of us are very experienced using tdd as we much prefer other means of development but we agreed that we would want to try tdd to get the practice. We also wanted to have all levels of tests ranging from unit tests, through integration tests, system tests and of course acceptance tests.

We initially started writing tests for the backend however we very quickly ran into problems. Our main problem was that although we were setting up tests for the database connectors, controllers and for the api we had no idea how the data would look like in the database. This meant that all our initial tests were done on mock data to see if the tests would work if we just gave it some hard coded values. Once real data was introduced we quickly realized that some of the tests had to be rewritten as the hard coded properties were tested another way than what the data actually needed. Tests were then refactored and used every time we made a change in the code to see if it would pass or fail.

Another issue with the tests was that some of the queries would give several thousand results. The issue with the many results was that each database sent the data back differently. Although each database sent back the same number of data points as well as the same data, however the order they sent was vastly different. As we can’t test for all one

thousand or more pieces of data we just checked for the first returns which were different for all of the databases for some of the queries. This again meant we had to refactor our tests so that they were checking on the right values depending on which database we used to get the data.

Of all the tests we did, the functional tests seemed the most useful. Functional tests helped us test if the functionality actually worked. Did we send the data the right way, did we get the right data, etc. This was very useful when creating the different queries as we had to create queries for 3 different databases. Each database being vastly different from the next the queries also are quite different but it was imperative they give the same results. As soon as we had queries for one database to work it was very easy to test the remaining two, as we were expecting the same results as the first one. It made the process of making the queries for the second and third database significantly easier as we already knew what to look for and had a way to test it.

Once we had the right data and we knew what data we needed to look for we didn't find one test level that much more complicated than the others. But if we were to chose one out of necessity it would probably be the api tests. We had a lot of issues getting the test to return the right data. We could easily get it to return the right length of data however we kept getting a list of null values. It took us some time to figure out how to work around this and get actually data in a nice format to appear. However once we figured that out writing those tests were actually a lot easier than the original tests we made as rest assured made it very easy for us to look for what we wanted to.

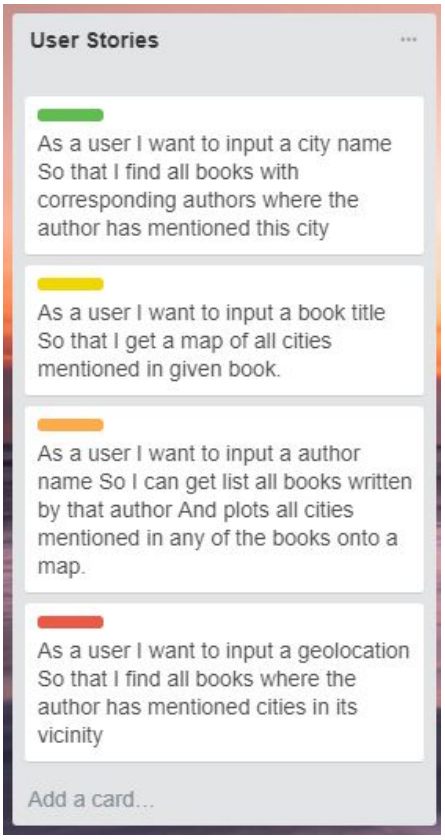
Iteration Metrics

During this project we have been subject to a number of reviews where we were expected to present some sort of metrics to document our work and progress.

For this project we chose to record and maintain the following metrics:

- Trello board with tracking of time estimated versus time spent.
- A burndown condensing the above point into an "easy-to-read" format.
- A breakdown of our test coverage (as in percent of methods tested)

The trello board allows us to communicate user stories and tasks in a concise and easy manner



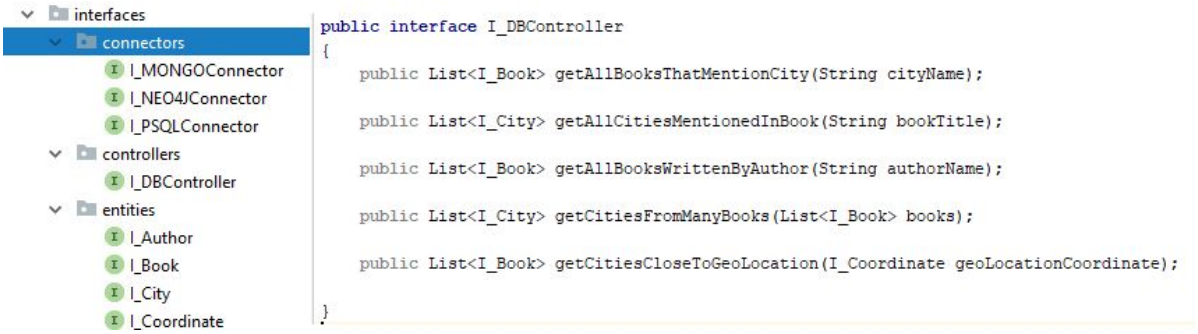
A screenshot of a part of the Trello board, detailing the extrapolated user stories.

Code Coverage

From the very start of the project we chose to document all of our “important” classes and methods through interfaces.

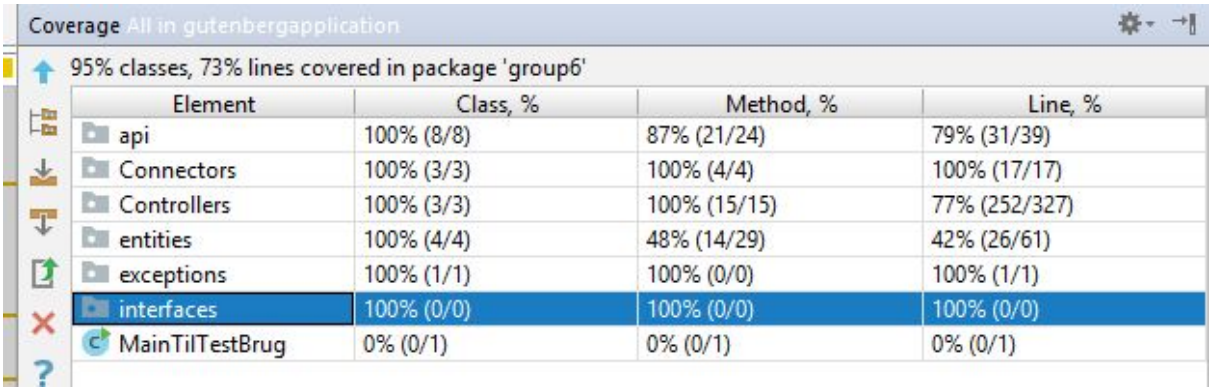
We limited our definition of important to be the following subsystems of the application:

- Database connection
- Database controllers
- REST API
- Data entities (in this domain this is things such as: Authors, Books etc)



The interface package and an example of an interface

From the very start our goal was to cover every single method described in the interfaces in our JUnit tests (with the exception of the entities since they only have *setters and getters*, which we consider trivial).



The test coverage of our project, as you can see almost all methods in the API, Controllers and Connectors are tested.

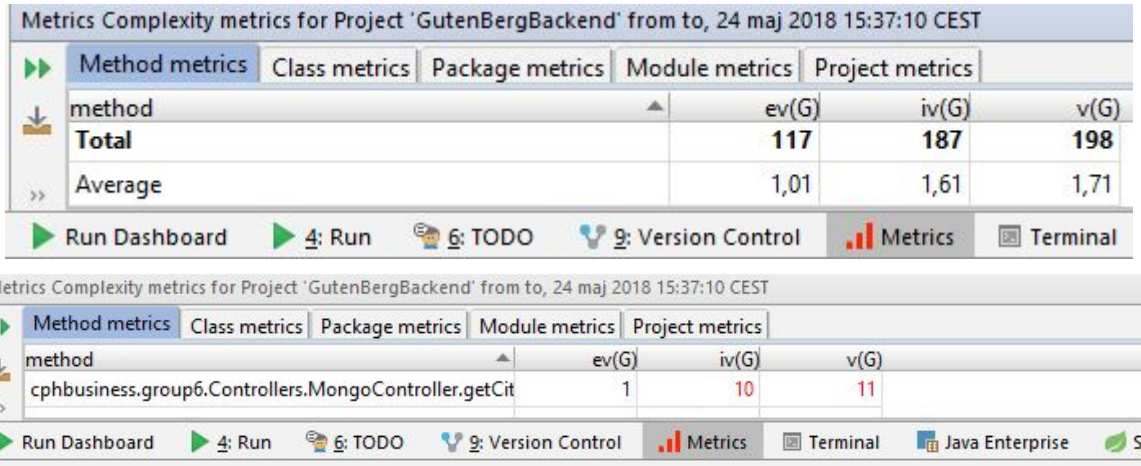
Internal Code Reviews

During the project we did a lot of “informal” reviewing of each others work, this is due to several facts: Often the person who wrote the test cases (for TDD purposes) was not the final person to write the actual code. This means that we all catch each other's mistakes, since any dissonance in our thought processes would be illuminated by the tests failing or some such.

This also meant that we did not see any added value in holding “formal” code reviews, since we did it in increments as we developed.

Cyclomatic Complexity (CC)

During the entire project we strived to keep the complexity of our code “manageable”, however we did not make any “set-in-stone” rules about the maximum allowed CC, which means that in a couple of spot we allowed ourselves to increase the CC a bit if it made sense.



A total breakdown of the CC (topmost) and a case where we allowed ourselves a little extra CC

Documentation, Rationale and Reflection

Requirements

There were 4 overall requirements to the application. which we made into our only 4 functional requirements:

1. As a user I want to input a city name So that I find all books with corresponding authors where the author has mentioned this city
2. As a user I want to input a book title So that I get a map of all cities mentioned in given book.
3. As a user I want to input a author name So I can get list all books written by that author And plots all cities mentioned in any of the books onto a map.
4. As a user I want to input a geolocation So that I find all books where the author has mentioned cities in its vicinity

All the user stories serve a specific requirement from the problem description. We did not however give them a time limit, as all of them need the same to function properly. They all need the data to be extracted, database to active with data loaded, a backend with connections to the database, an api that connects the frontend to the backend, and a frontend. All 5 need to be completed before any of the user stories can be completed. the only difference really is the controllers that use different logic to get data from the database.

From the start we implemented some rules, such as a book always has a title, release date and an author, or a city always has a name and coordinates.
This means that when a user uses the first api call stated in the user stories, it should return a list of books all of which has a title, a release date and an author, without exception. Likewise the second will should return a list of cities all of which has a name, a latitude and a longitude.

All of them affect performance, as there is a ton of data in the backend, and it takes a while to get all the relevant ones to the end user. Now each additional feature will slow down parallel api calls as the backend gets overloaded with data handling. This being the major reason that there are no more api calls than the essential ones, to minimize the potential workload a user would give to the backend.

Security is however pretty easy as none of the requirements need any writing permission and will read data from the database, making it invulnerable to cross site scripting. The only security issue would be that a potential attacker could disrupt the backend rather easily by ddos, as it would probably not take a lot to start causing trouble as we have only invested the bare minimum in the backend droplet.

Testing the user stories is a fairly easy task which we have done in several stages. There are regression/integration tests that test the controllers and connectors to see if the database is both connected and returns the correct data in the right format that the frontend needs. We have api tests that check if the api handles requests properly. We have

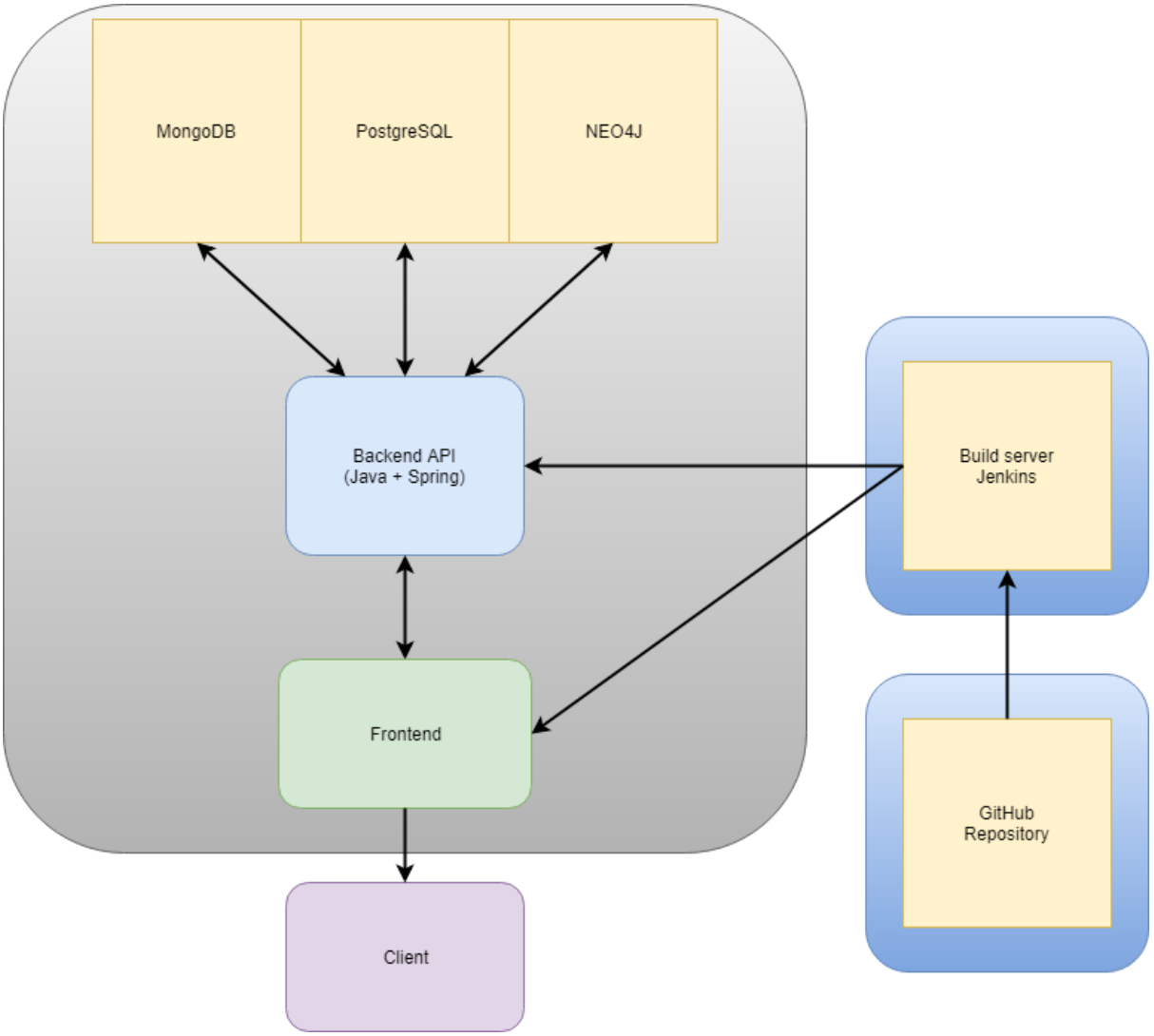
component tests that check if the frontend components is running as it should we have selenium tests that tests if the end user can see the correct information via the GUI.

As we are natural born agile developers, we use exploratory testing as naturally as breathing. We used it to write the majority of all the code that is now functioning, as it is a very productive and informative way to gather the necessary tools to solve whatever issue and write whatever feature you are working on. e.g with the controllers, specifically the mongo controller is very different to use in java than javascript or with the mongoshell for that matter. We just tried a bunch of different strategies to see what we could make it do and what doesn't work, from here it just to muddle around until you have all the necessary parts and the feature is done.

Architecture and Design

Architecture

For our project we chose an architecture that we were experienced and confident in using, such that we could put the highest possible amount of effort into designing the database- and testing parts of the assignment.



A diagram detailing the architecture of the entire system

In the end, this choice of architecture turned out to be well-suited for the purpose and scope of this assignment, since it is relatively efficient in it's operation, while at the same time it provides plenty of opportunity to write interesting tests.

Design

In terms of design, we set out to accomplish making the project as easily testable as possible.
To that end we took a number of decisions in regards to how we did things (could be considered our *coding standards*), in this part we will try to evaluate their effectiveness in achieving the stated goal.

In short the steps we took were the following:

- Create interfaces for all "relevant" classes and implement them.
- Keep a clear package structure such that there's never any doubt about what goes where.
- Keep the cyclomatic complexity low (where viable), such that hidden dependencies etc. are easily discovered.
- If you do anything more "advanced" than *getter/setter* level stuff, you need to write the tests for the thing you are building before you start coding.

Of the above the first point (about interfaces) was the most valuable of the four, since it allowed us to define an "idiot proof" solution to communication the design between team members. For example, we were basically able to code the REST API before we coded the database controllers, which is amazing since the API relies very heavily on the methods from the controllers (since they retrieve all the data that the API needs to return).

In regards to the other points mentioned above, they provided somewhat more dubious results. The CC part is very easy to implement, since it usually only requires you to split you large methods into many smaller ones, however, we would argue that, for an application of this scale, the rewards for doing such are rather negligible.

In regards to the package structure, a small positive effect was achieved in the sense that none of the group members were ever in doubt as to where to put their code. This in turn saved us a couple of refactorings.

The very last point, Test Driven Development, was the hardest to adhere to. This is due to the fact that all of us are used to an approach of "making it up as we go along" (so to speak), which means that it is quite challenging to suddenly have to "formulate" your entire mental picture of the system in terms of unit tests (as opposed to writing the code, and then considering the testing which, at least to us, is a more natural approach).

However TDD did allow us to catch a couple of mistakes at an earlier point than we normally would have done, in particular in regards to meaningful exception throwing/handling, which is easy forget when everything is working.

A second benefit to all the tests, written using TDD, is that we can run them all through our Jenkins CI chain. Here we can stop “unstable” code from being deployed to our production server.

Test

We also have a selenium test that automatically tests all the functionality and ideally goes through all the user stories. We wanted to setup selenium on the Jenkins server but could not get it to work as React doesn’t work well with the headless browser for selenium. Instead we have it as a kind of full system test. The tests test all the functionality works, it will fail if any of the databases go down, it will fail if the backend doesn’t return the expected results and lastly it will also fail if the frontend isn’t behaving as expected.

Organisation

Throughout the entire project we have had impeccable team work. Even though we split up the workload, if there ever came a time where one member was in a pinch the others would swarm to him to fix whatever was an issue.
This combined with splitting the group in testing and data handling from the start made it for an incredibly smooth development process. Where any given developer got a deeper understanding of the entire platform for each problem they helped another developer with. Making for an increasingly stable and robust platform.
We did hold sprint meetings, but it was more the comradery and problem solving from every team member, that made the agile process so much better. As e.g. when someone was making the interface and the tests for the api, even though there was not much of a backend nor database yet, could discuss with the developer doing the data extraction, to find out what data could be expected from the api when it was made, along with all the other areas where something similar occured made for better testing and coding overall.

Discussion

So the teamwork in general was outstanding. What could have been better would be earlier implementation of the Spring api/maven and the jenkins, as the development process further along we forgot to do it. Where it only got finished in the last part of the development process. Even though it was not a requirement to implement a CI/CD chain, we thought it could tie up the entire testing course well, to have the tests implemented through it. Even more so when we consider that we did Test Driven Development wherever it could be implemented, even though we forgot at times as it is a hard mindset to get used to. The code in general is therefore both robust and easily testable.