0ctober – December 2018

340CT - SOFTWARE QUALITY AND PROCESS MANAGEMENT

MY MUSIC APP

OSKARS POZDNAKOVS

1. FUNCTIONALITY

Registration/Login

https://livecoventryac-my.sharepoint.com/:v:/g/personal/pozdnako_uni_coventry_ac_uk/EfvpTLzjz4pIkrkY5oIe_CsBkE9K1eLiu3yA2UVgSffoxA?e=O8uSjY

Playlist Management

https://livecoventryac-my.sharepoint.com/:v:/g/personal/pozdnako_uni_coventry_ac_uk/EdNcPP5fyCBLlyNM92_pa7MBmGd62uBbgdja8aMoOCiLdQ?e=L2R171

Music Streaming

https://livecoventryac-my.sharepoint.com/:v:/g/personal/pozdnako_uni_coventry_ac_uk/EaONmppe-4dMu0QQjdEybukBAh_08-pGt1MsqpNzLsdgng?e=03mjd7

2. DEVELOPMENT PROCESS

System Analysis

Domain Modelling

Having learned the project requirements, the first step taken was to capture the domain knowledge and express it in a model.

A. Knowledge Crunching

In order to make the subsequent work effective, a 'knowledge crunching' session was held with the domain expert (i.e. the lab tutor – Dr Mitchell). This helped establish abstract model of the problem domain.

B. Ubiquitous Language

Also, during the session Dr Mitchell and I worked on domain language extraction. We built the domain vocabulary and split the complexity into smaller contexts.

From this work a learning was gained about the mechanics and dynamics of the domain that the software implementation had to replicate.

The output of it was the domain language description, available in Appendix A.

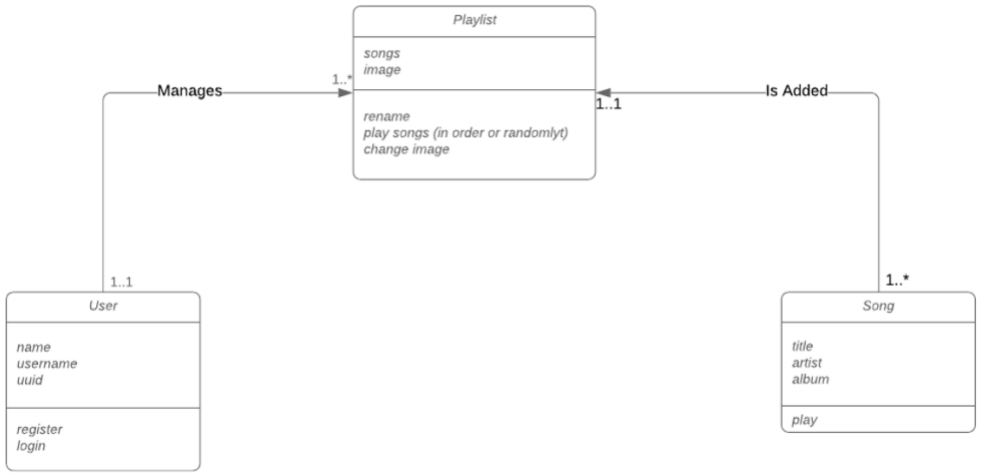
Building domain model and binding it with implementation

Having ascertained the concepts that the domain experts would use to think about problems, next, I needed to organize them explicitly into a model that software could be based on.

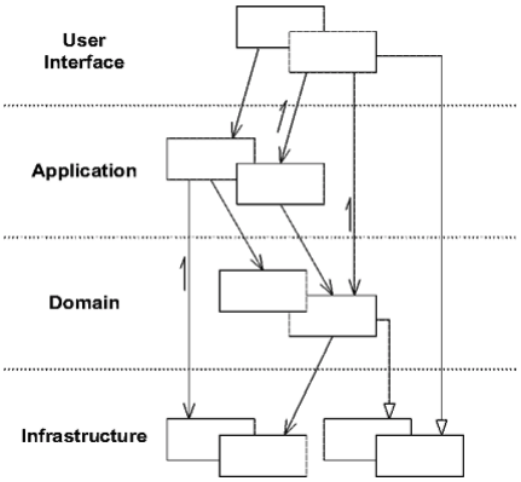
C. Building a domain model

The analysis then was processed to capture concepts from the domain, which could be constructed as components with software development tools, that would correctly solve the application needs.

The software system, reflecting the domain model in a literal way, was designed with very obvious mapping.



Designing the system (Architecture)



Selection of architectural patterns

D. Isolating the domain

Mr Evans (2003) suggests: "we need to be able to look at the elements of our model and see them as a system", that would be decoupled from other functions. The standard approach of a layered architecture was employed to make the domain explicit in the mass of the system.

Model View Controller (MVC) pattern was selected to cultivate the isolation of the domain layer, enabling the conceptual classes (models) to be designed without being concerned about the user interface interacting with them.

Additionally, a modification of adding services to MVC was made to suit the need of keeping the application layer focused on its purpose and facilitated loose coupling.

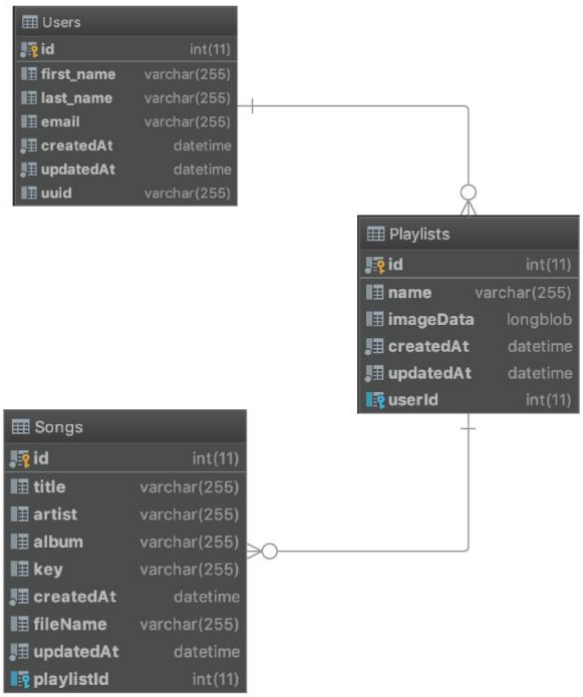
Model Expressed in Software

While producing the code, the guideline was followed that it must be an expression of the model.

i. Information System

The first step taken in creating the system design, which would reflect the domain model, was to map the conceptual classes on domain classes used in the software.

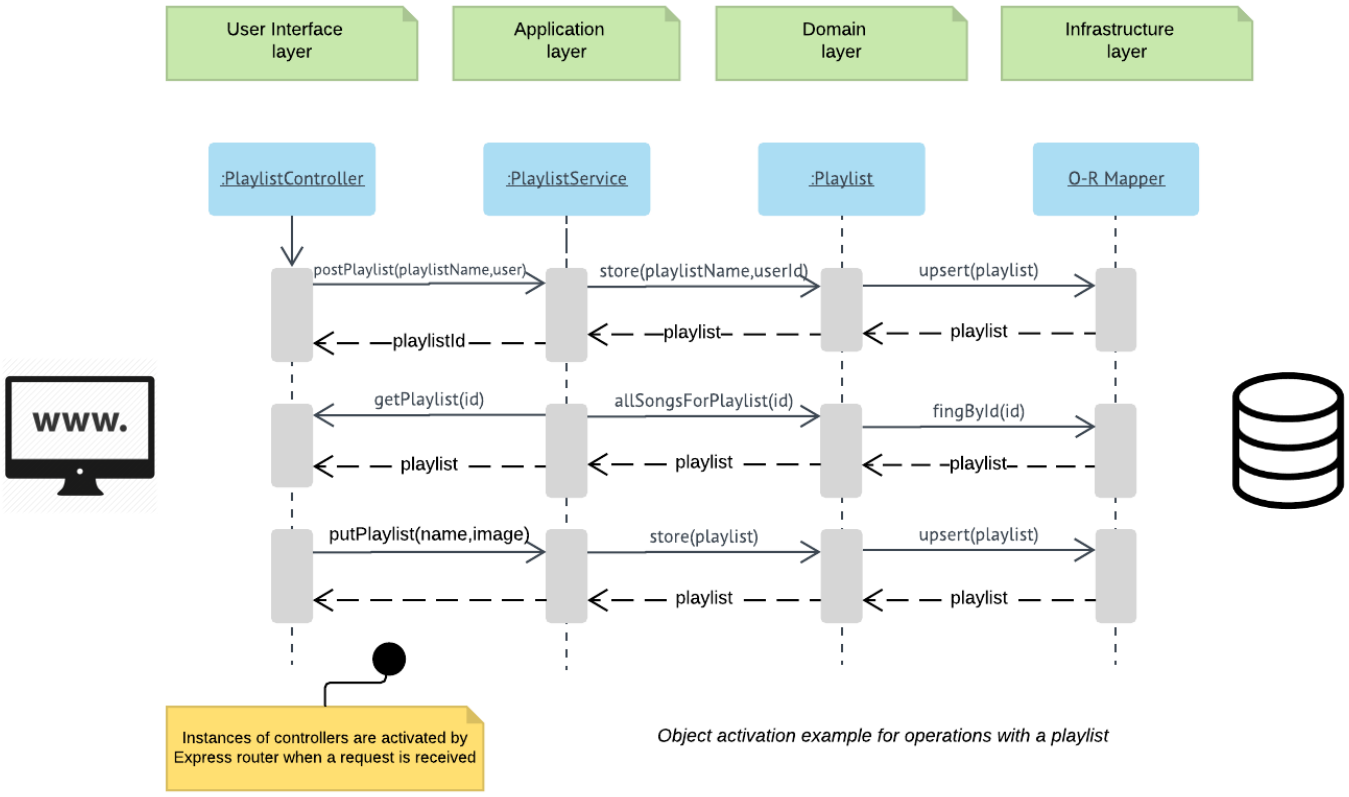
This was done by creating an ER Diagram.



ii. Object interaction

Next, to visualise how objects would interact arranged in time sequence according to the chosen MVC architectural patter, a sequence diagram was produced.

An example of object interaction and message passing in playlist related operations is provided below.

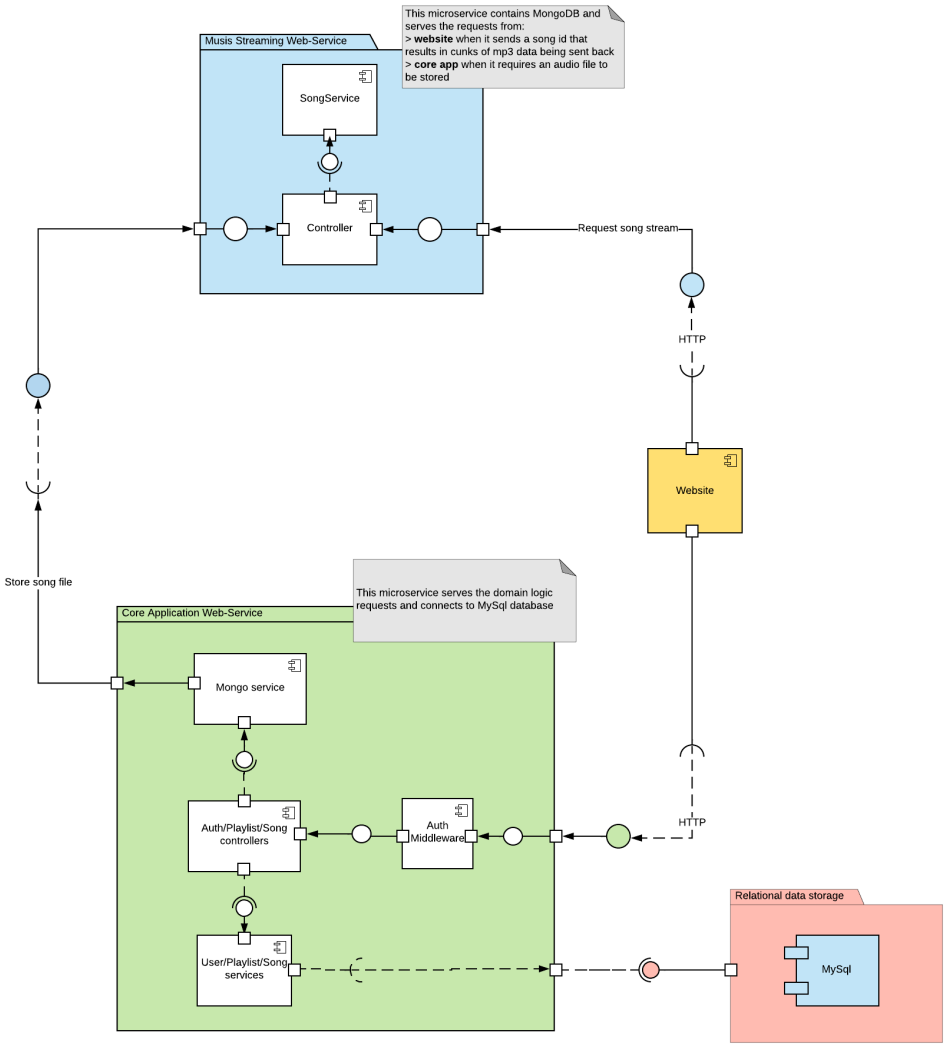


Object activation example for operations with a playlist

iii. High level architecture

Having gained fundamental appreciation of lower level processes, work was started on the high-level design of the system.

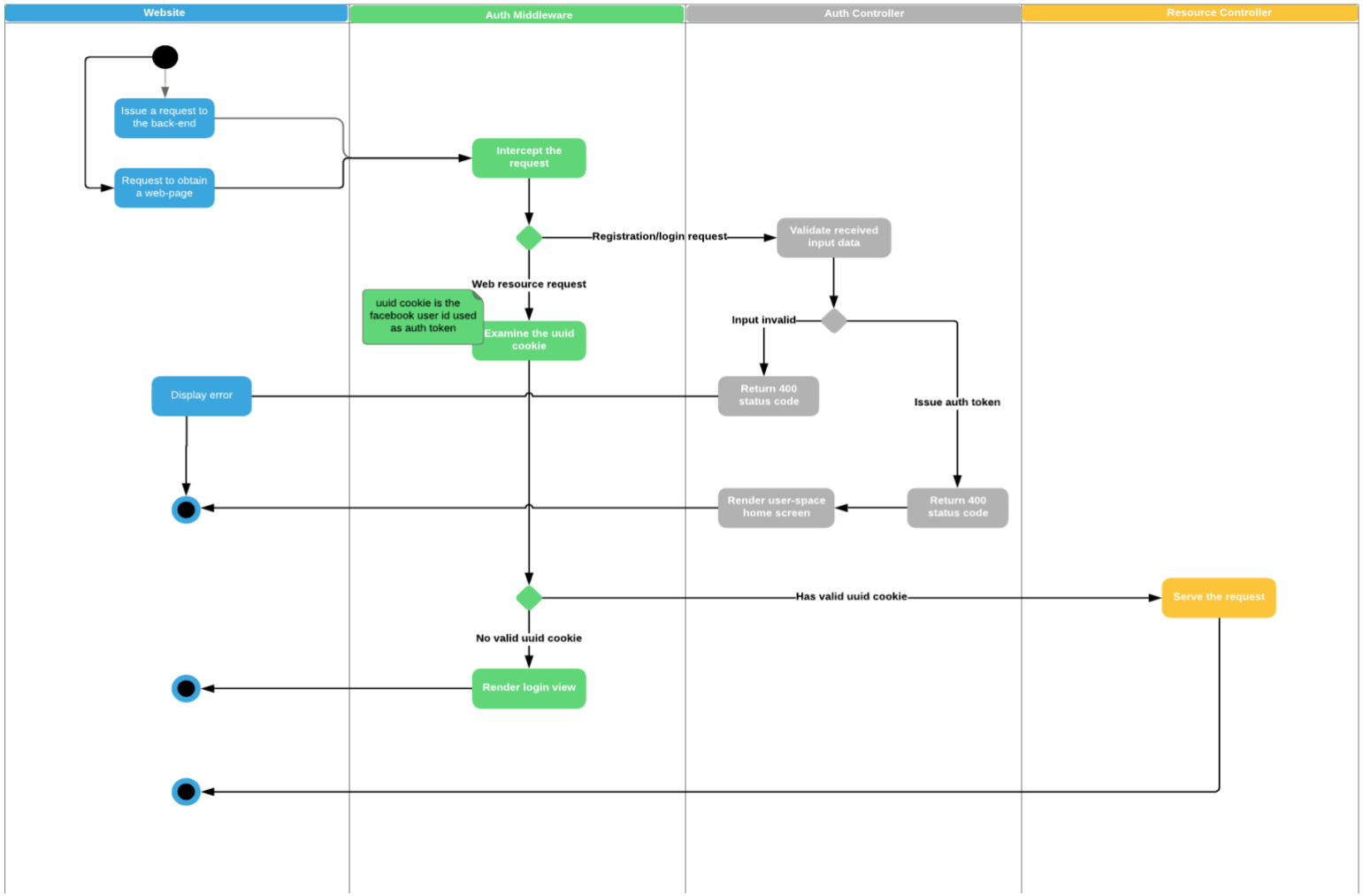
It was depicted in a component diagram which was reiterated several times. This helped the thought process of wiring several components together and ensure that all the requirements were covered.



Based on the fact that music storage and streaming are resource intensive processes, the system scalability became of a major concern. This was addressed by having dedicated databases and isolate them together with the infrastructure and functionality they served for. This also meant that the system would benefit from increased resilience and easier maintenance.

iv. Authentication & Authorization

During the development, the question arose about the authentication and authorization matters. It was resolved using the application level middleware. Its implementation is depicted in the activity diagram below.



SYSTEM IMPLEMENTATION

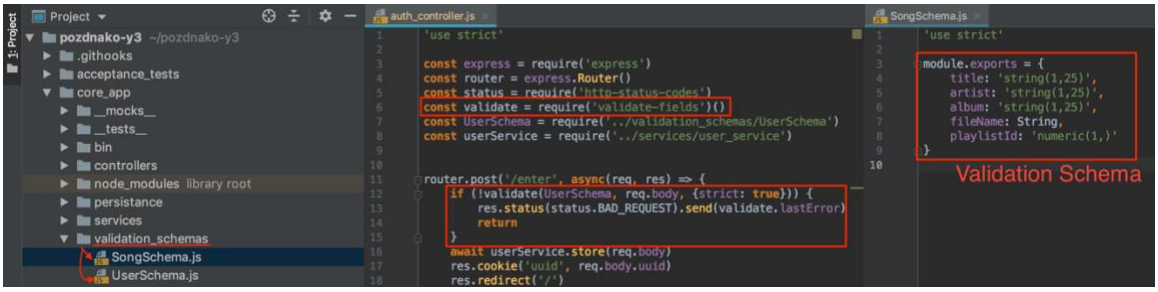
Templating

Handlebars temple engine was used to render dynamic elements on the views.

The user interface was built employing Bootstrap, jQuery and other libraries to enhance user experience.

Input validation

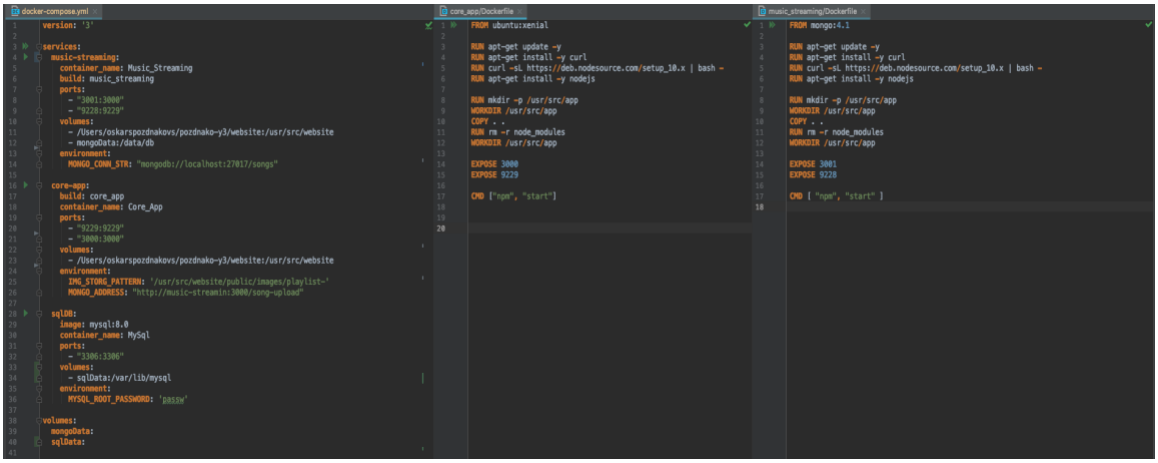
An npm library was added for input validation.



Microservices

Docker was used to implement the Microservice architecture and the following services were defined:

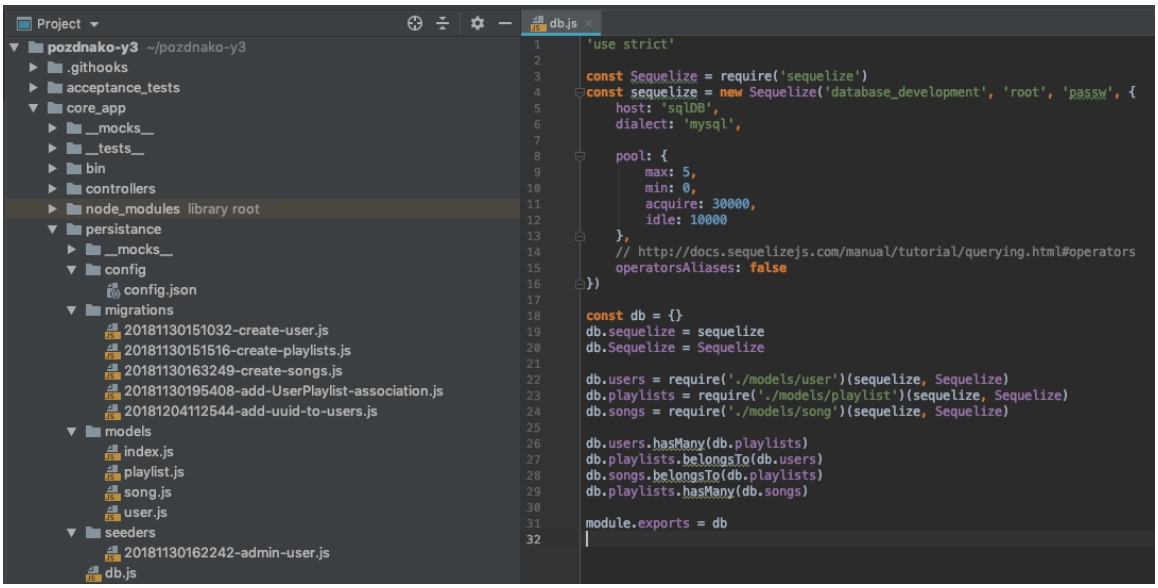
1. **Core App** serves the domain logic requests.
2. **Music Streaming** uses MondoDB to store and stream audio files and serve corresponding requests.
3. **SQL Database** to store the relational data. This was encapsulated in a different container because it was intended to also add Auth middleware to **Music Streaming** that would require access to it. Due to lack of time it was not done.



Object Relational Mapping (ORM)

The system has an integrated ORM library: sequelize.

Its features that were utilised are as follows: migrations, seeds and object mapping. Also, a centralised model access object (db.js) was created and exported.

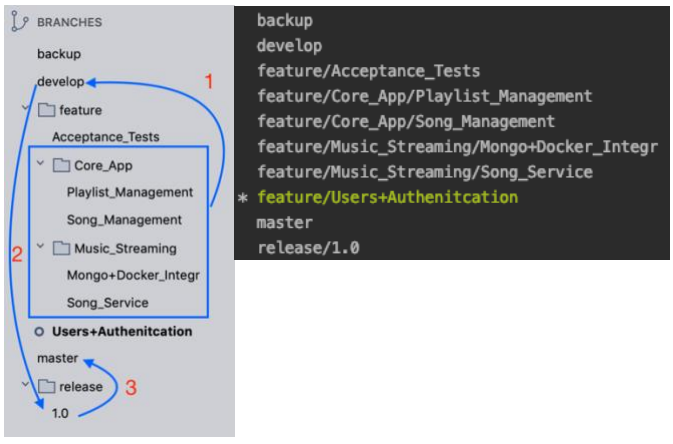


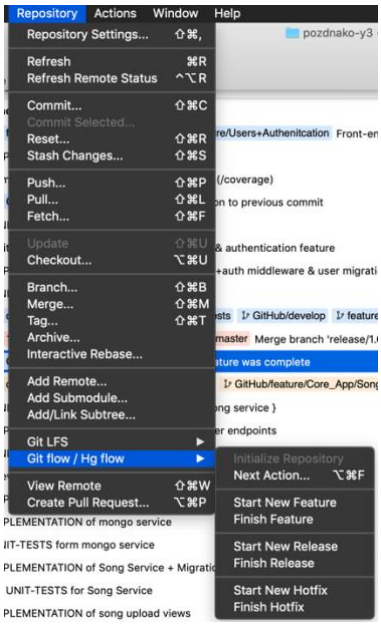
The migrations and the seed are executed when the package script is run in the **Core App**.

Code Quality

A static code analysis tool ESLint was used to enforce rules ensuring cleaner syntax and the subset of JavaScript used that does not compromise the code quality.

Version Control

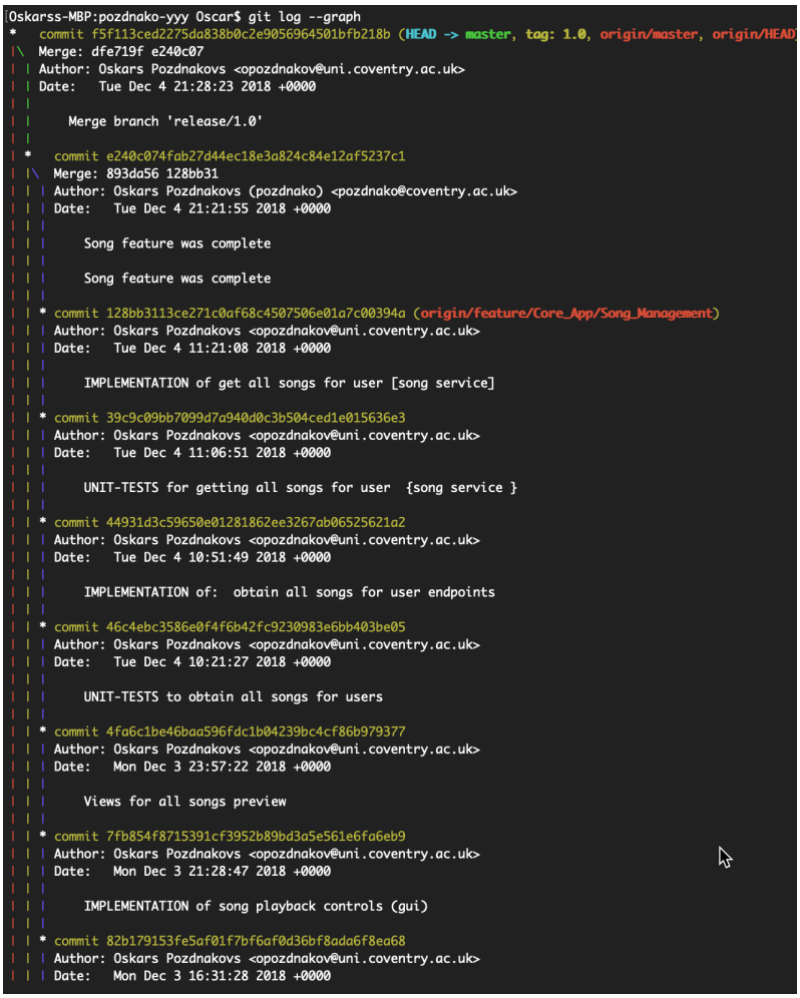




Git Flow branching model was used for the version control. It was done with the help of the tools integrated in my git GUI after Git Flow was initialised in the repository.

The system was developed by creating feature branches that upon completion were pulled into *develop* via PRs. When all the *Core_App* and *Music_Streaming* related branches were in *develop,* the *release/1.0* were branched off and subsequently a PR was completed to merge it into *master.*

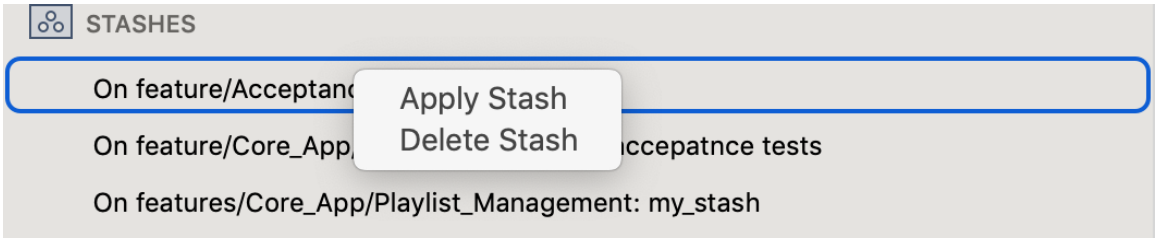
Git log --graph was used to visualise dependency tree for all commits



Git tag, occasionally annotated tags were added to commits that were followed by pull requests.

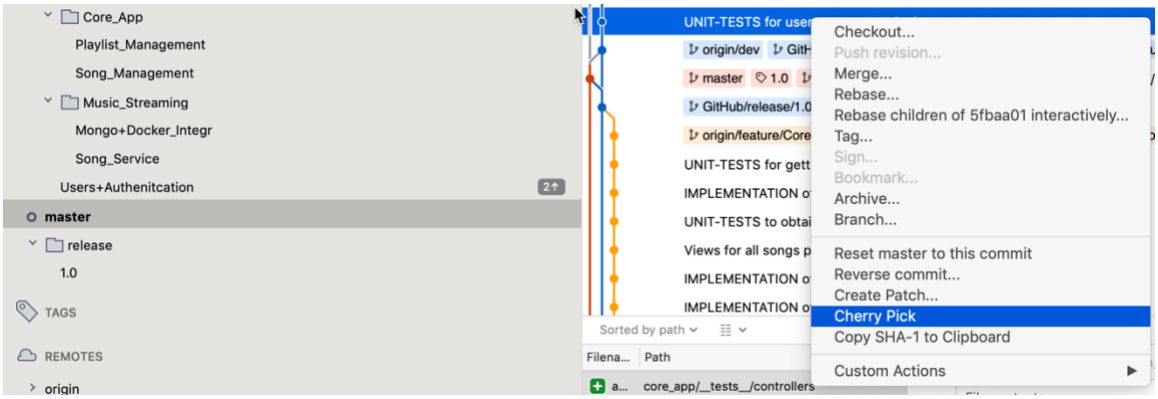
Git stash

Stashing was occasionally performed when I needed to switch between incompatible branches, but I was constrained by the recent commits.



Cherry picking

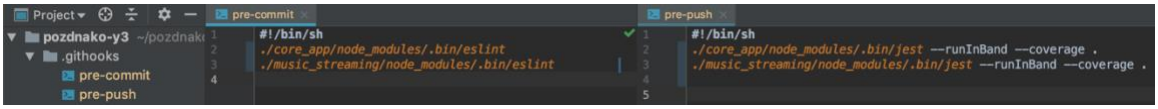
It was useful when I needed to obtain a particular commit from a different branch that didn't have it.



Continuous Integration

GitHooks were also used to ensure continuous quality of the software.

- pre-commit – run ESLinter
- pre-push – run the Jest test suites



The configuration for successful test completion can be seen here -->

3. TESTING

TDD

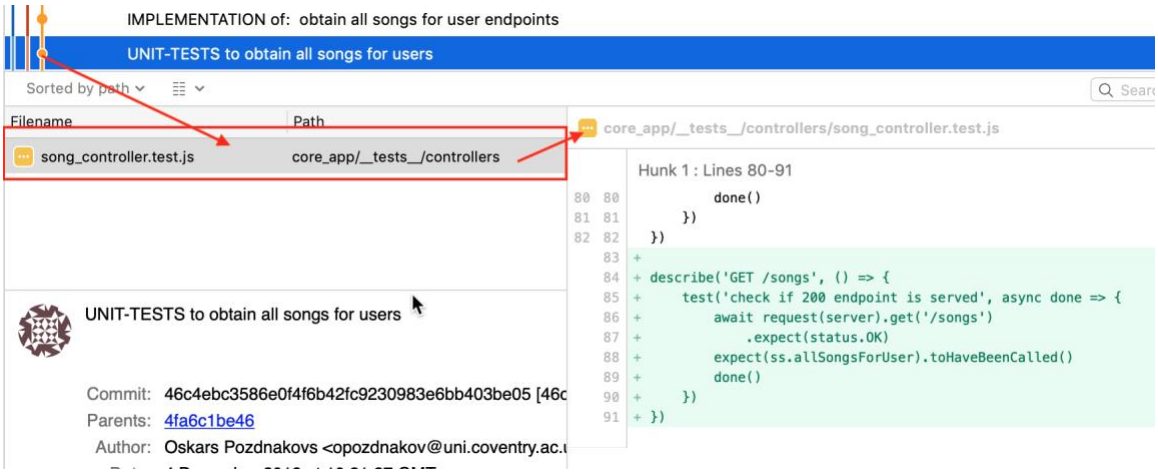
Test Driven Development approach was applied most of the time working on this project.

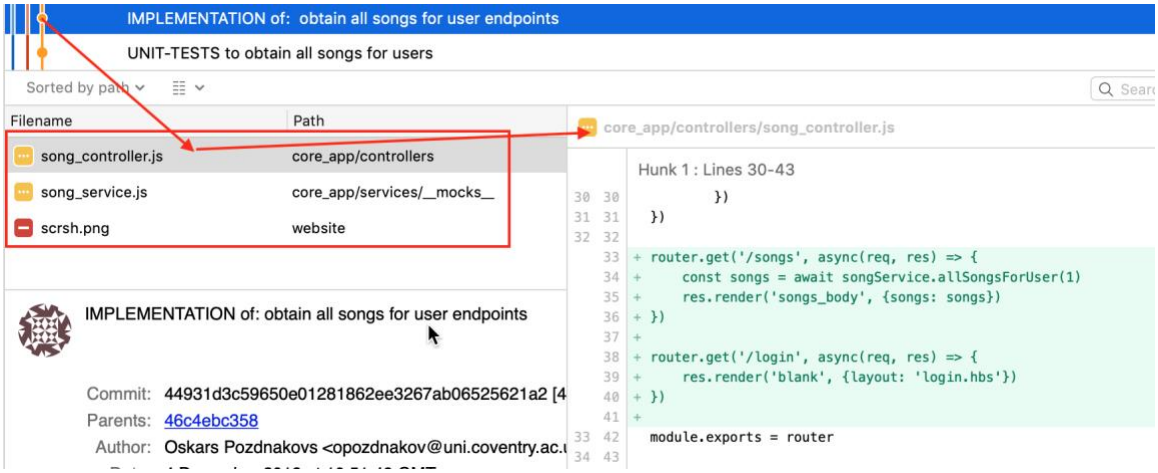
It was used in the following way:

1. an automated test case was written that defined a desired improvement or functionality
2. the entire test suite was run to ensure the test case failed
3. then minimum amount of code was written to pass that test case(s)
4. the entire test-suite was re-run to see if the new test case passed

Because the architecture had a service layer, when a controller unit test was written a mocked service was created and committed along.

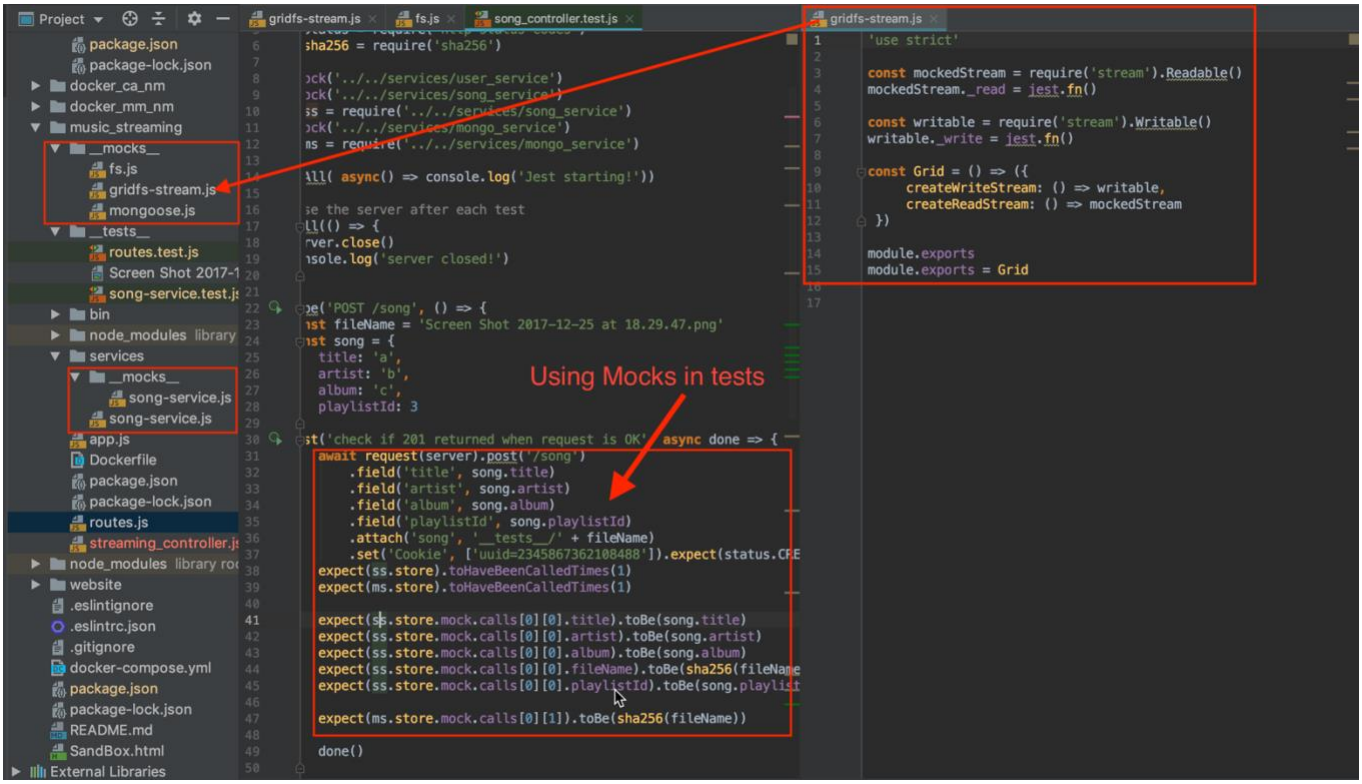
In the example below, when the implementation of the controller was committed, it was using mocked song_service.js until the actual one was implemented subsequently.





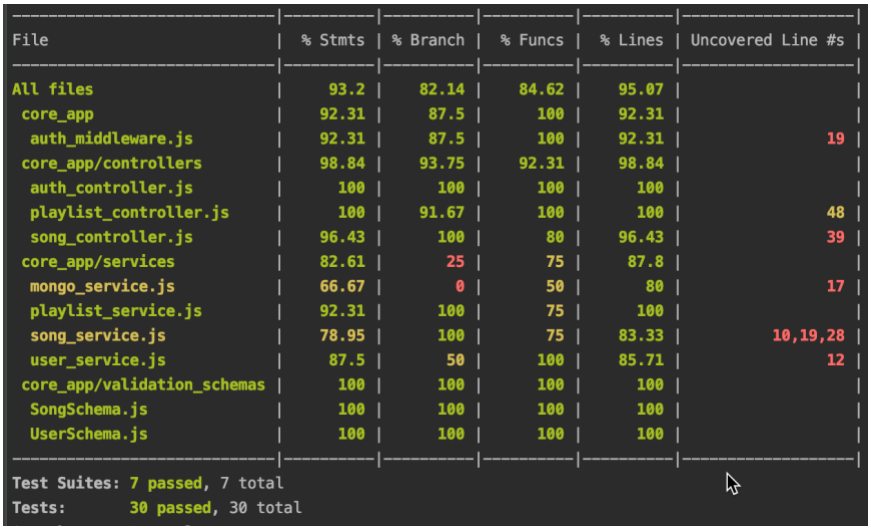
Mocks

The system has a complete set of mocks for external and internal dependencies. Some mocks are manual, and some are automatic. Altogether 9 dependencies were mocked for testing and TDD purposes.



Code Coverage

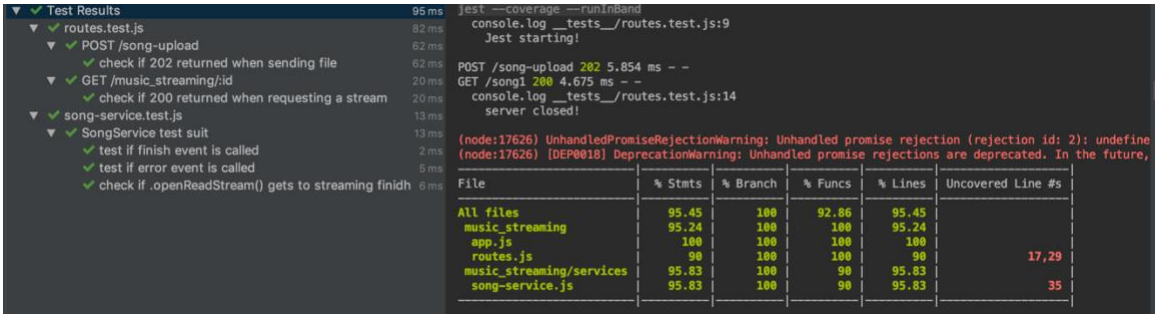
Core App



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	93.2	82.14	84.62	95.07	
core_app	92.31	87.5	100	92.31	
auth_middleware.js	92.31	87.5	100	92.31	19
core_app/controllers	98.84	93.75	92.31	98.84	
auth_controller.js	100	100	100	100	
playlist_controller.js	100	91.67	100	100	48
song_controller.js	96.43	100	80	96.43	39
core_app/services	82.61	25	75	87.8	
mongo_service.js	66.67	0	50	80	17
playlist_service.js	92.31	100	75	100	
song_service.js	78.95	100	75	83.33	10,19,28
user_service.js	87.5	50	100	85.71	12
core_app/validation_schemas	100	100	100	100	
SongSchema.js	100	100	100	100	
UserSchema.js	100	100	100	100	

Test Suites: 7 passed, 7 total
Tests: 30 passed, 30 total

Music Streaming

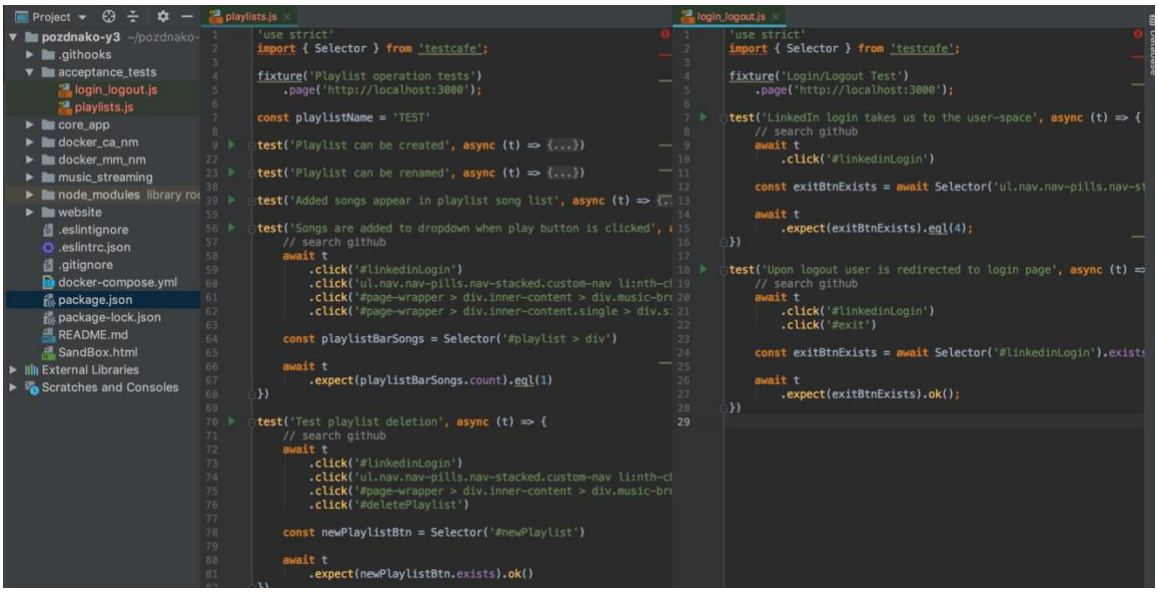


Acceptance tests

Testcafe framework was used to perform front-end testing.

Two features have such test suites:

- Sign-in/Sign-up (2 test cases)
- Playlist management (4 test cases)




```
Oskarss-MBP:pozdnako-y3 Oscar$ testcafe safari ./acceptance_tests
 Running tests in:
 - Safari 12.0.1 / Mac OS X 10.14.1

 Login/Logout Test
 ✓ LinkedIn login takes us to the user-space
 ✓ Upon logout user is redirected to login page

 Playlist operation tests
 ✓ Playlist can be created
 ✓ Playlist can be renamed
 ✓ Added songs appear in playlist song list
 ✓ Test playlist deletion


 6 passed (46s)
```

4. SUMMARY

While doing the system analysis phase with the DDD approach, I gained a learning about its usefulness when it's put in practice.

Ubiquitous language of the domain could potentially leverage clearer communication between stakeholders by eliminating all the jargon used by each party.

As DDD complements the object-oriented paradigm (OOP) with its expression of the domain model, my belief is that it would also make a system more adaptable to change. This would be because the model is almost directly mapped on objects.

Also, I have learned how to use a range of tools to ensure code quality. Among such tools are ESLint, Git hooks and CI.

I managed to familiarise with more advanced branching model than that I had used before, namely GitHub-flow.

Github-flow with its simple branching model is perfectly suitable for single developer projects with one version in production.

On the other hand, Git-flow is better suitable for projects with several versions in production and CI/CD pipelines configured.

Another powerful practice was TDD. I realised that it acts as a safety net when changes are made to the source code. Not to mention the effect it makes to force you to think of a better design in advance. However, a downside of it is that you are left with a massive number of tests to maintain, which takes a lot of time.

I could possibly have done more work on the acceptance testing making snapshots in headless browsers and add to CI. This would massively improve the software quality.

APPENDIX A

Users can sign in, or register if they don't have an account.

Once signed in, they can manage their list of playlists.

Every playlist will have an image and an option to change it.

Users can manage song files for a selected playlist.

Songs can be played either in order or randomly.

Registration

The users will be required to provide:

- Full name
- Phone number
- Email
- Facebook

Sign-in

Users can sign in using

- Facebook

Playlist Management

Users will be able to carry out the following actions with playlists:

- Create
- Delete
- Rename

Playlist image

The following operations will be allowed

- Upload from PNG file

Manage songs

- Upload from mp3 file

Song playback

- Change volume

- Fast forward
- Rewind
- Play/ Pause