

SUPER PROJECT

Best Movies Inc.

Students:

Rokas Barasa 285047

Rafał Pierścieniak 279471

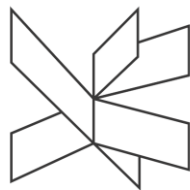
Emmanuel Chukwukodinaka Simon 285152

Supervisors:

Jakob Knop Rasmussen

Laurits Ivar Anesen

Richard Brooks



Bring ideas to life
VIA University College

Best Movie Inc.

34,353 characters

6th semester

17th December 2021

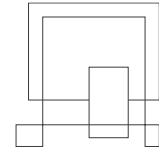
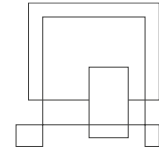
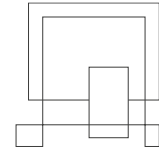


Table of content

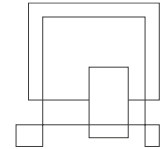
1	Introduction.....	1
2	Analysis	1
2.1	User stories	1
2.2	Use case diagram.....	3
2.3	Use case descriptions.....	4
2.4	Activity Diagram.....	6
2.5	Sequence Diagram	7
2.6	System Sequence Diagram	7
2.7	Domain model	8
3	Design	9
3.1	System Architecture.....	9
3.2	Front-end.....	10
3.2.1	Technologies	10
3.2.2	Architecture	13
3.3	Back-end	14
3.3.1	Technologies	14
3.3.1.5	Express validator.....	16
3.3.2	Third party API's	17
3.3.3	Architecture	18
3.4	Database	20
3.4.1	MySQL Database	20
3.4.2	Architecture	20
3.5	Cloud infrastructure	21



3.5.1	Google cloud	21
3.5.2	GitHub	22
3.5.3	Cloud build	22
3.5.4	Cloud run.....	22
3.5.5	Compute engine	23
3.5.6	Cloud SQL.....	23
3.5.7	Docker	24
3.6	UI design	24
3.6.1	Views.....	25
3.6.2	Layout.....	26
3.6.3	Styling.....	26
4	Implementation	26
5	Test	34
5.1	Back-end	35
5.1.1	Supertest	35
5.1.2	Sinon	35
5.1.3	Back-end test results	35
5.2	Frontend	36
5.3	Test Specifications.....	38
6	Results and discussions	39
6.1	Results	40
7	Conclusions	40
8	Project future	41
9	Website URL	42
10	References.....	42



11	Appendices	1
----	------------------	---



1 Introduction

Peter is the CEO of BestMovies Inc. Peter has requested an application to be made that would fit the needs of movie enthusiasts. Peter wants a social platform where the users can contribute and access information about movies. Peter wants to be able to easily find movies by searching, seeing data about movies and creating his favourite list of movies.

He wants to get some other ideas for this application which must be realized as well, but they must be for movie lovers. Peter also said that he would be pleased if the application had some statistics about movies. Peter underlined that he does not want to store anything stored on BestMovies Inc. premises or take care of any hardware.

The project will mainly focus on the movie lover's experience and user interaction with each other. Only the highest priorities of Peter will be considered for the initial project.

2 Analysis

2.1 User stories

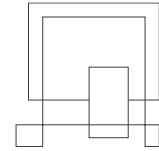
User stories are ordered by importance.

As a user, I want to be able to...

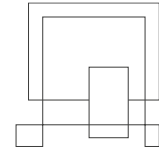
1. See a collection of movies, so that I can find the movies I am interested in.
2. Search for movies, so that I can find the specific movie I am interested in.
3. See data related to a movie, so that I see if the movie match my interest.

Acceptance criteria:

- Be able to see rating.
- Be able to see actors.



- Be able to see directors.
 - Be able to see the description.
4. Login, so that I can authenticate, keep track of my favourite movies and join social activities.
- Acceptance criteria:
- Be able to log in with google services.
5. Manage a list of favourite movies so I can share them with my friends.
- Acceptance criteria:
- “User manages” refers to – User can add or remove a movie from the favourite list and copy the link to his favourite movies
 - The user can manage the list if he is logged in.
 - Link to user favourite movies can be accessible by non-logged-in users.
6. Sort collection while exploring movies so I can find movies that match my interest.
- Acceptance criteria – user can sort by:
- Chosen genre
 - Chosen movie attribute (min. title, year, rating)
 - Order ascending or descending
7. Comment on movies so I can discuss my opinion about the movie with other users.
- Acceptance criteria:
- User can see the comments under in data related to the movie
 - User can make a comment only if is Logged in.
 - User can make a sub comment under the comment.
 - User cannot make a comment under sub comment.
8. Search for the actor, so that I can find information about the actor and movies related to him.
9. See data related to an actor so that I can see what other movies the actor stared in.
- Acceptance criteria:
- Be able to see name.
 - Be able to see age.



- Be able to country of origin.
 - Be able to see movies that they acted in.
10. See how many people viewed the movie on the page, so that I can see how popular the movie is.

2.2 Use case diagram

Analysis of the requirements has been represented by the use case diagram shown below. Following the requirements, the main actions of the user have been described by the direct relationship between the user and use cases. By the outcome of this analysis part, it is stated that movie details would allow for more complex operations. That is why this use case extends some of the main actions as well as is extended by other cases. Logged in, acceptance criteria have been outlined by “include” relationship in “Check favourites movies”, “Comment” and “Manage favourite status”. Use case diagram can be found in Appendix 2 (fig. 1)

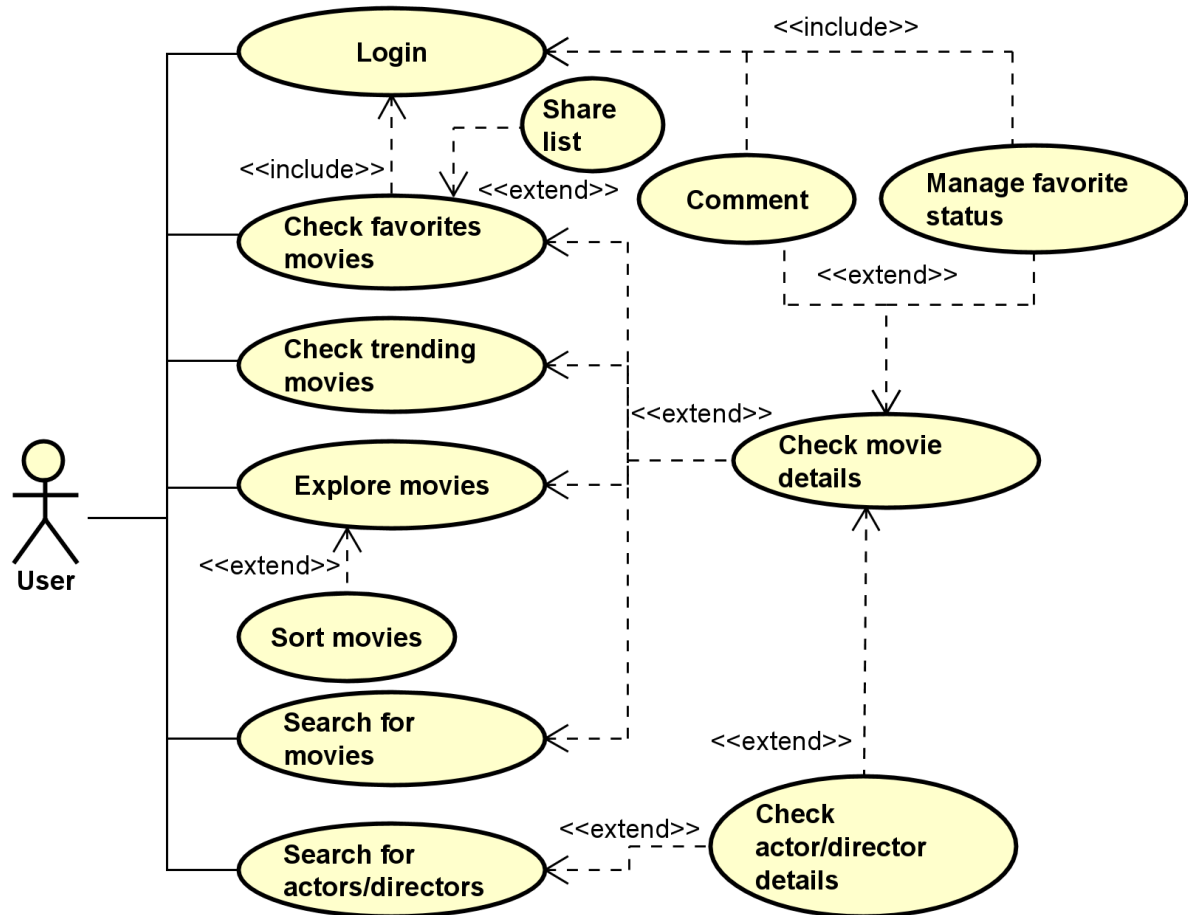
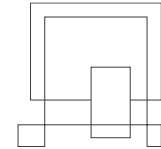
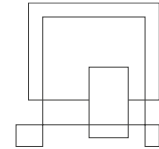


Figure 1 - Use Case Diagram

2.3 Use case descriptions

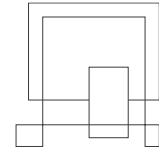
As example, use case description of “Comment”. Following the user stories, some of the acceptance criteria have been outlined in use case descriptions. The “Comment” use case has been divided into two other cases: the user intends to comment on the movie itself or the user intends to reply to the comment under the movie. Base sequence separates those two scenarios. The exception sequence includes the consequences of being not logged in while submitting the comment/reply. Note that besides the notification user is given a chance of not rewriting the comment/reply and he/she should be able to log in during that process. Therefore, there is no strict precondition for log in before the



use case runs. Use case description in fig 2. More use case descriptions can be found in Appendix 3.

UseCase	Comment
Summary	User can comment the movie. User can reply to the comment under the movie.
Actor	
Precondition	User see movie details with comment section
Postcondition	Created comment/reply is visible on the comment section
Base Sequence	<p>A - User intend to comment the movie:</p> <ol style="list-style-type: none"> 1) User enters the comment text in comment section 2) User submits the comment to send 3) System updates the comment section by added comment <p>B - User intend to reply to the comment:</p> <ol style="list-style-type: none"> 1) User choose "reply" option next to the comment 2) System updates the comment section by displaying the reply field under it 3) User enters the reply in reply field 4) User submits the reply to send 5) System updates the comment section by added reply under the comment
Branch Sequence	
Exception Sequence	<p>1) User is not logged in the system:</p> <p>A:2-3 - System notify the user about the exception and user continue from step 2</p> <p>B:4-5 - System notify the user about the exception and user continue from step 4</p>
Sub UseCase	Login
Note	User can login during the base sequence. The process of sending the comment or reply to comment can be stopped before performing step A2 or B4.

Figure 2 - Use Case Description: Comment



2.4 Activity Diagram

All of Activity Diagrams made can be found in Appendix 2. Example can be seen in fig.

3

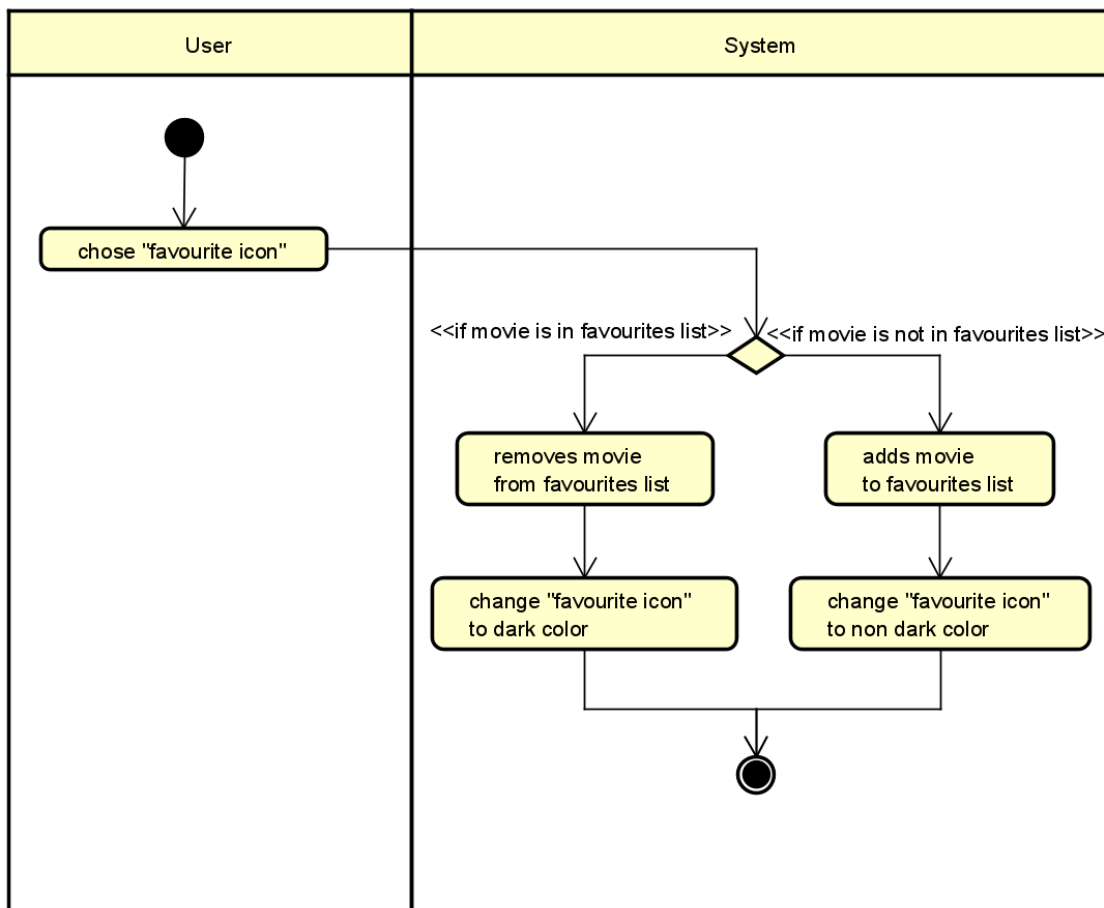
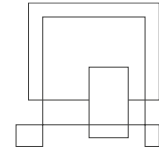


Figure 3 - Activity Diagram: Manage favourites list



2.5 Sequence Diagram

All made Sequence Diagrams can be found in Appendix 2. Example can be seen in fig.

4

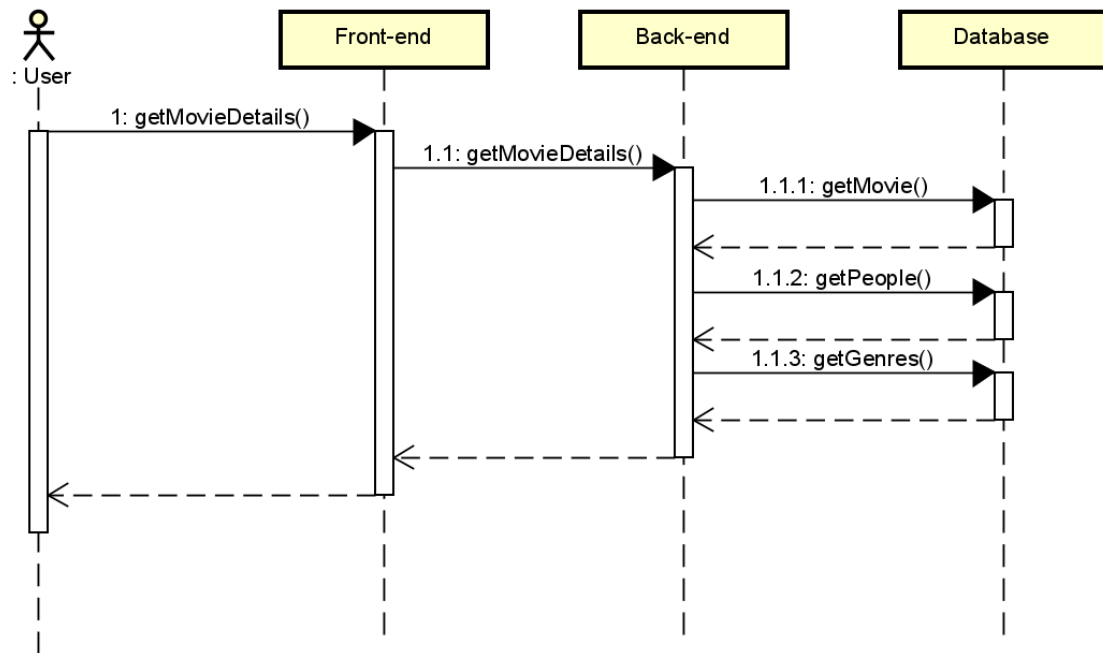


Figure 4 - Sequence Diagram: Movie Details

2.6 System Sequence Diagram

All made System Sequence Diagrams can be found in Appendix 2. Example can be seen in fig. 5.

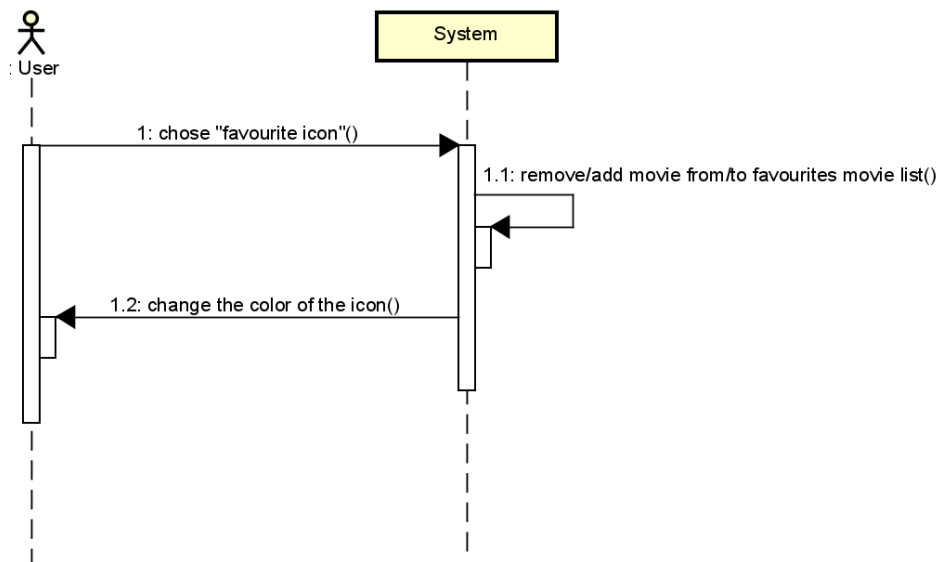
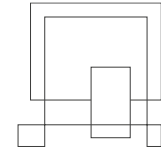


Figure 5 - System Sequence Diagram: Manage favourites

2.7 Domain model

The domain model (fig.6) shows the relationships between entities in the system. The main entity in the system is the movie, without it, there would be no point to the system. The movie has added information of people related to the movie and genres, which can be assigned to multiple movies. The secondary functionality is social media. The user makes comments that attach to the movies and are seen publicly. The user is also able to add a movie to his favourite list, which is also publicly visible. The domain model can be found in Appendix 2.

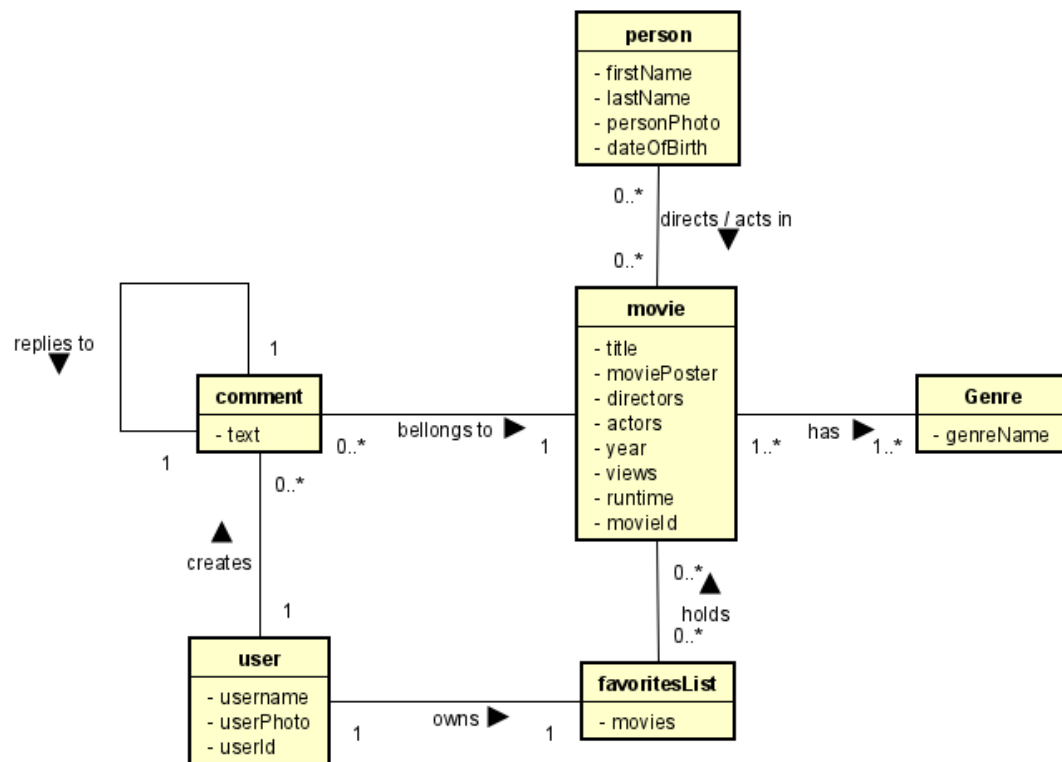
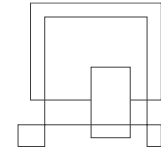


Figure 6 - Domain model

3 Design

3.1 System Architecture

The overall system architecture is standard three tier architecture with Database, Backend and Frontend. The system is supported by third party APIs and several key other technologies, what has been visualized on (fig. 7).

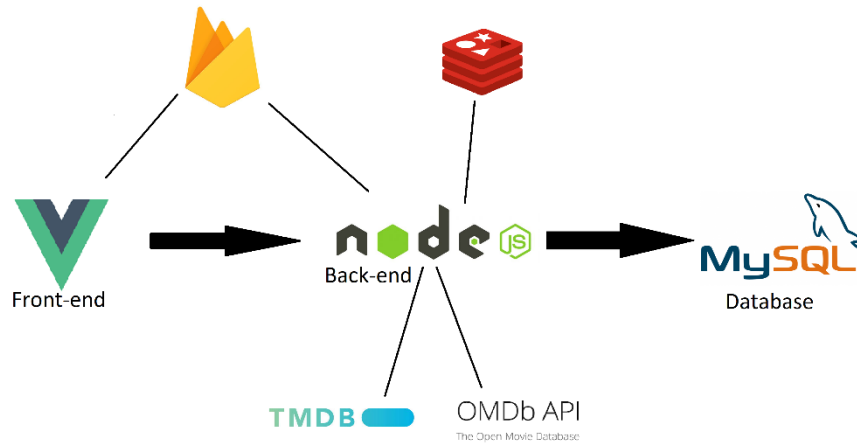
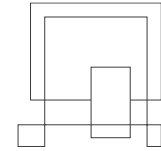


Figure 7 – Main technologies for each tier with supporting third parties and technologies

3.2 Front-end

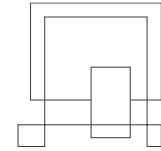
3.2.1 Technologies

The technologies have been chosen based on the popularity – as this aspect speed up process of solving potential issues by reaching the target community. The other factor is a common experience among the developers. The last and most important factor for making decisions in this section is the overall advantages, disadvantages, and comparison of the technologies.

All the technologies have been chosen following the news versions. The reason for that is to ensure long-term support and give a possibility to use the newest features from them.

3.2.1.1 Vue.js v3.x

Vue was picked as a framework for developing the front-end. It is a component-based framework, which means that it gives the possibility to work on the App component by another component. This helps to scope the different parts of the project and simplify the workload management. Another reason to pick this framework over other



component-based ones is its lightweight size and good performance. Additionally, Vue is a progressive framework, meaning it can be gradually introduced into the code and its core can be extended by other libraries.

3.2.1.2 Vuex v4.x

Vuex is a state management pattern (fig. 8) and library for Vue.js applications. It provides a global store for components in an application. Also, it ensures that the state can only be mutated in a predictable fashion.

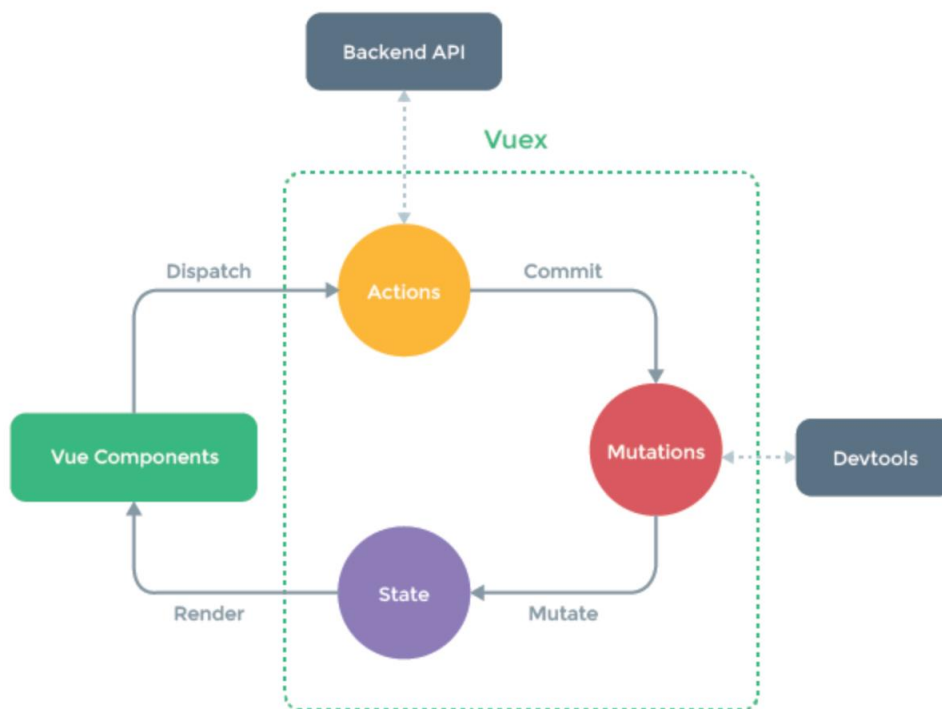
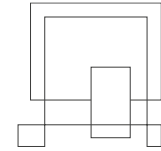


Figure 8 - state management pattern in Vuex

3.2.1.3 VueRouter

The vue-router library is the officially supported library for Vue.js. Dynamic route matching in vue-router allows for constant reacting to parameters changes. This supports well the experience of reactive programming.

Navigation - The Vue Router allows for both HTML and programmatic JS navigation.



Syntax of the Vue Router has been inspired from express. Therefore, it supports many advanced matching patterns like optional parameters, zero/more parameters or custom regex patterns.

3.2.1.4 Vuetify

Vuetify is a material design framework design on top of Vue.js. It provides the developers with ready to use components that allow building engaging user experience UI.

Mobile-first approach – the ready components are well supported on mobile devices. This allows to follow common practice in front-end development – first for mobiles, then for desktops.






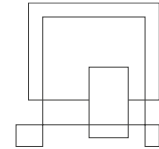
Vue Framework Comparison 2021					
Features	 Vuetify	 BootstrapVue	 Buefy	 Element UI	 Quasar
Accessibility and section 508 support	●	●	●		
Business and enterprise support	●				
Long-term Support	●				
Release cadence**	Weekly	Bi-Weekly	Bi-Monthly	Bi-Weekly	Bi-Weekly
RTL support	●	●		●	●
Premium themes	●	●			
Treeshaking	Automatic	Manual	Manual	Manual	Automatic
**Based on average of all Major/Minor/Patch releases over the last 12 months.					

Figure 9 - Vue Design Framework Comparison 2021 (<https://vuetifyjs.com/en/introduction/why-vuetify/#comparison>)



As shown in the figure above (fig 9) the argument of this choice over others is the frameworks frequency of releases. The long-term support would ensure the stability of the application over time.

3.2.1.5 FirebaseUI for Web - Auth

FirebaseUI for Web was chosen to provide simple customizable UI bindings on top of Firebase SDKs. Since it is built on top of Firebase Auth, it benefits the project by bringing auth solution that handles Identity providers Sign in by all most popular social media platforms including Google. Therefore, this solution would allow a user to log in to the system without creating an account – just as requested in the Analysis section.

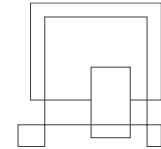
3.2.1.6 Cypress

Cypress is a tool made for browser-based test automation in modern web development. The primary argument for choosing it was that it is widely used in industry and is with all modern JavaScript frameworks. What is more, it enables several kinds of tests as: end-to-end tests, integration tests, unit tests.

Cypress ecosystem is the other advantage. Besides providing a great environment for TDD, it can be easily integrated into most of CI Providers. Additionally, it has the advantage of Dashboard Service that record test runs and simplifies failed test inspections for developers.

3.2.2 Architecture

The frontend architecture is defined by the structure of the view and Component, The components act as re-usable elements which supply data into a view. The component is used in the view to render the interface to users. A class diagram that represents each individual component as a class categorized by View and Components has been made (refer to Appendix_ Class Diagram Frontend).The class diagram only contains components that are created in the project and not imported. The data and props are identified as fields, while the computed properties are defined by a “/” before name of properties. Watch mounted and methods are all represented as methods in the class diagram. Below is the Toolbar and Toolbar Menu components



which comprise a part of the class diagram. (fig.10) Full front-end class diagram can be found in Appendix 2.

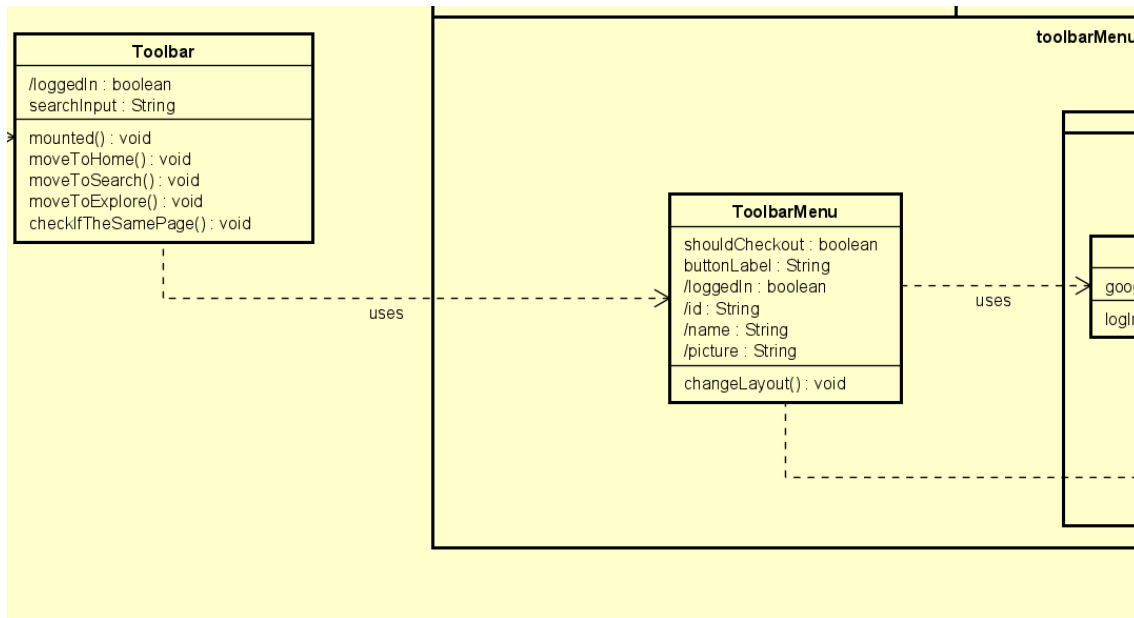


Figure 10 - Toolbar and Toolbar components on Class Diagram

3.3 Back-end

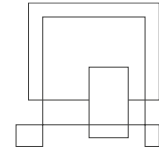
The back-end communicates with the database and third-party services to serve data to the front-end.

3.3.1 Technologies

3.3.1.1 Node.js

Node.js with JavaScript was picked as the language of choice for the back-end. Node.js is well supported by cloud service providers and its executed language, JavaScript, is very popular among the developer community. (StackOverflow, 2020) (Google cloud, n.d.) (Microsoft Azure, n.d.)

Node.js is beneficial to have in the back-end as it uses the same language as the frontend. A developer does not have to switch to a different syntax. Code, npm libraries and functions can be reused between the front-end and back-end.



Node.js is single-threaded and non-blocking. When node.js waits for a response, database or HTTP request its single thread is handling requests from other users instead of waiting for a response. This makes Node.js very scalable and perfect for the cloud environment.

Node.js is often used with a powerful and effective library management system – npm. Npm hosts open-source libraries. Open-source libraries are constantly monitored for security issues which are then reported to the users of the libraries when they are installed. Due to the abundance of well-made libraries, if a library is malfunctioning or unsafe it can be easily swapped out with one that is of equally good quality. Using the same language in back-end as the front-end means that it is easier to find people who can work with the system. Front-end developers can develop in the back-end as well.

3.3.1.2 Express.js

Express.js was chosen as the framework to handle HTTP REST requests. It is designed to build web applications and API's. Its lightweight, simplicity and extendibility make it very easy to construct the ideal API without any unneeded features.

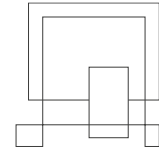
Middleware is what makes the framework so customizable. Middleware is code that is specified to run before the endpoint code. The currently used middleware is:

- input validation
- user authorization
- error handling
- caching services.

Middleware functions can be provided by libraries or created.

3.3.1.3 MySQL Driver

A low-level driver was chosen to handle the connection to the MySQL database. This choice was made as ORM's get complicated when they need to handle more complex



queries. Writing regular SQL in a low-level driver is less space-efficient, but what it lacks in abstraction is quickly made up by consistency, familiarity, and dependability.

SQL is something that most programmers know already therefore it is easy for even the least experienced developers to work on the back-end.

Queries that use written SQL instead of a language-dependent ORM framework can be easily transferred to another language in case of changed priorities in the back-end.

3.3.1.4 Redis

Redis is used to provide caching to the back-end. Redis is a key-value database. Caching saves the result from an endpoint request and returns the same data for all requests to that endpoint afterwards. The cached data expires after a set amount of time so that data does not get outdated. A response from an endpoint is cached only if it is not returning data for authorized user functionality.

Redis speeds up the back-end response rate and saves money on the cloud by eliminating CPU time that would be used for querying and constructing the response. CPU usage is the main driver of costs in cloud services. (Google cloud, n.d.)

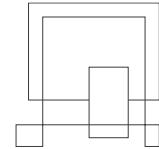
3.3.1.5 Express validator

Express validator is used to check the validity of parameters being passed to the back-end. Express validator provides some middleware functions that constrain the data and provide valuable feedback to the users calling the API by saying what parameters are wrong.

3.3.1.6 Firebase admin

Firebase admin library is used for user authorization. Firebase admin library enables access to Firebase services from privileged environments like the back-end.

This can be used because the back-end does not send provide its source code to the users of the system as the front-end does. Meaning secrets can be stored in the back-



end environment and functionality that requires privacy can be accessed from the back-end.

Authorization functions with the libraries JWT validation functions are used in middleware and are only enabled for specific endpoints. Interacting with favourite list, registering, and commenting.

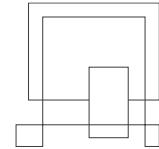
When the user makes a request to an endpoint that needs authorization, he passes an authorization header that contains a JWT token obtained after he logs in using firebase auth on the front-end. The firebase admin library validates the JWT token and returns the user information contained in the token if it is valid, otherwise the request is rejected. The user id from firebase is then compared to the user id passed in endpoint parameters. If the id's match the endpoint is accessed, if not it is rejected.

This authorization protects the user's data in the system from manipulation by other users.

3.3.2 Third-party API's

Third-party API's like OMDB and TMDB are used to get missing data about the movies in the database. These API's hold data about movies and things related to them. OMDB is a very simple API where as TMDB has a lot more information in it.

The OMDB is used to get the data about movies. It gives a good overview of the movie in text form. It does not have all the data that is needed for our service. Actor and director photos are not included and cannot be obtained by the API. Some old movie posters are not available. That is why the data is supplemented from the TMDB API.



3.3.3 Architecture

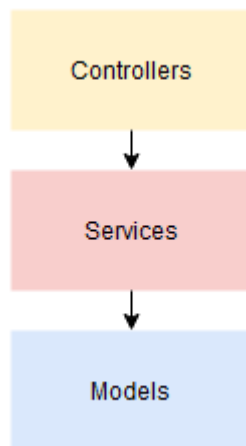


Figure 11 - layered architecture

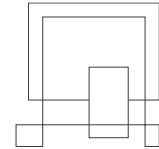
The back-end is organized into layers (fig.11):

- Controllers – Endpoints and input validation, response to the user, response status, response data.
- Services – Business logic
- Models – Connection to third party API's. databases and serverless functions.

This architecture helps separate the concerns of different functionalities in the back-end. Testing is made easier as a layer can be stubbed to isolate the functions that are being tested.

An overview of the architecture can be seen below (fig 12). The Backend-end class diagram can be found in Appendix 2.

The server is started by running the `index.js` file. The file initializes the routes and middleware that the endpoints use. The connections to services are established by the models and middleware when they obtain the connection modules.



3.4 Database

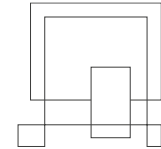
3.4.1 MySQL Database

MySQL was chosen as the database management system. PostgreSQL or MSSQL would have been equally good choices. MySQL is however more popular than the other database systems, meaning that finding people who work with it is easier.

The database when it is initialized stores id's, titles and year's of movies. The back-end gradually updates the data for movies by adding posters, descriptions, genres and actors obtained from third party API's. This speeds data delivery and independence over time.

The database's primary functions are to speed up data delivery to the users, reduce reliance on third party API's over time and relate the users to movies in form of comments or favourites.

3.4.2 Architecture



The database is made using normalized tables that capture all content of a movie and the users interaction with the movie. The enhanced entity-relationship diagram can be seen in (fig. 13). EER diagram can be found in Appendix 2.

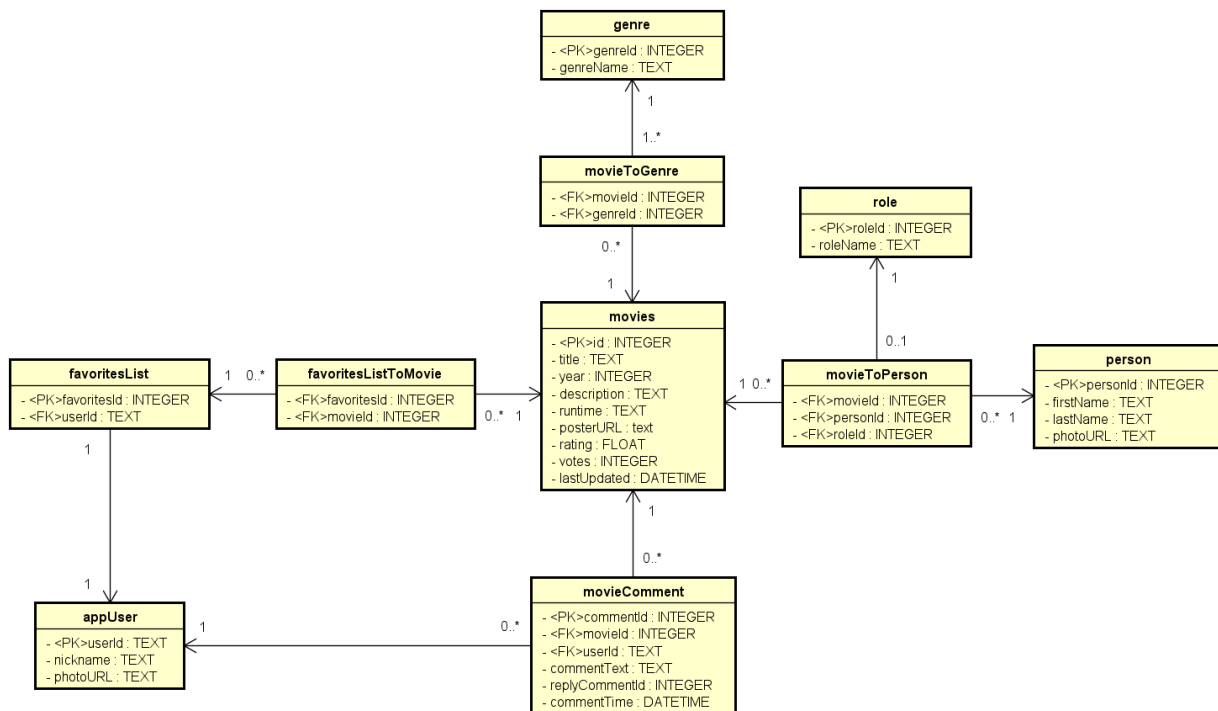


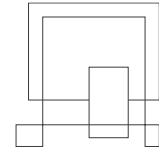
Figure 13 - EER Diagram

3.5 Cloud infrastructure

Instructions on how to set up cloud infrastructure can be found in Appendix 1.

3.5.1 Google cloud

Google cloud was chosen as the cloud service of choice due to it's native support of Firebase Auth and simple setup for continuous deployment of web applications and API's.



3.5.2 GitHub

GitHub is used to store the source code online in a public repository. Google cloud is connected to the repository and uses this connection to make deployments of the latest features in the system.

Frontend is using GitHub actions for continuous integration for running the tests. Therefore, it is recommended that before each merge to the main branch, developer will make pull request at first place. This will allow to run all the tests to ensure everything is working as it should. Additionally, this solution in DevOps brings the advantage of assigning the reviewers to the code while doing the pull request, which gives better control in the development team over code quality.

3.5.3 Cloud build

Cloud build is a google cloud service that builds docker images out of source code specified in a repository. It uses the “Dockerfile” specified by a path to build the application and push it to the cloud environment.

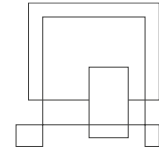
The project’s cloud build service is specified to deploy the image to the container runner service Cloud run after it finished the build process.

Cloud build was chosen as it does not necessarily need a “.yaml” specification to build the application and can instead use the regular “Dockerfile” specification for the build process.

The cloud build service is used to build the front-end application as well as the back-end application

3.5.4 Cloud run

Cloud run is a google cloud service that deploys docker containers form docker images in google cloud. It has additional features of error logging and revision handling. A new



revision gets created if a new source code commit gets successfully built and successfully deployed on the cloud run service. The revision is then made the primary revision, directing all traffic to that revision instead of the old ones. This makes sure the API or web application does not go down.

Cloud run was chosen from multiple services available on google cloud due to it's simplicity and container usage. It does not need manual configuration as does compute engine and it can host larger applications than serverless functions. All of this while not being too tied to one cloud platform due to containerization.

The cloud run service is used to deploy the front-end application as well as the back-end application

3.5.5 Compute engine

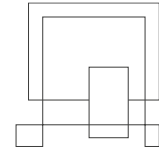
Compute engine is a google cloud service that hosts a virtual machine. Which is basically a Linux computer running in the cloud that can be accessed by ssh command line to install the applications. If an application can be accessed from outside the virtual machine it needs to have the applications port forwarded.

The compute engine service runs the Redis key value database that is used for caching of request responses in the back-end application.

3.5.6 Cloud SQL

Cloud sql is googles relational database service. It hosts a number popular of database management systems and makes them available to connect to using credentials.

The project has a MySQL database initialized on the service. The back-end communicates to this database using the MySQL library in Node.js



3.5.7 Docker

Docker is used for deployment to cloud services. Docker is used to create images which contain all libraries, dependencies and files needed for an application to run. Docker then uses these images to create a container, which is a standardized virtual Linux run environment for the application.

The containers can be very easily deployed to any cloud service and scaled depending on load on the application. A docker container can run anywhere and is independent from the users OS environment.

The back-end application image is built and deployed to the cloud after each successful build in the develop branch. The build process for the back-end runs tests that are defined in the source code to make sure everything is working. If a test fails, the build process is cancelled.

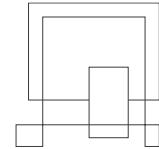
The build process itself is defined in a “Dockerfile”, the file uses a special syntax to describe what happens in the build process in steps. has it’s own syntax of t.

Just like the back-end, front-end docker application image is also build and deployed to the cloud after each successful build in the main branch. It is important to outlined that build process in frontend does not run tests that are defined in the source code. However, the tests are run during the pull request to the main branch (GitHub).

To ensure the same environment as on production – it is recommended that docker image is also used in local development environment.

3.6 UI design

Due to the luck of UI/UX designer in the team, the UI design section focuses mainly on the layouts.



3.6.1 Views

Views describe the main elements in the UI which will allow to fulfill the user stories and use cases from the analysis section.

This section outcome is the list of the views which will be used for the design of layouts.

- 1) Toolbar
- 2) Movie Details

The movie details will contain of subsections:

- Movie detailed information
- Actors from the movie
- Directors from the movie
- Comment section

- 3) Movie Collections

There will be different types of movie collection:

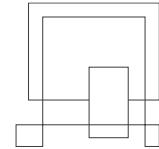
- Home Page – collection of trending movies and some carousel
- Search Page – collection of movies containing result of search only
- Explore Page – collection of explore movies with additional sorting options
- Favorites Page – collection of favorites movies of the user

All of them will have similar layout and will be based on basic collection of movies (grid of photos)

- 4) Sign in
- 5) Account
- 6) Actor/Director Details

Person details will contain of subsections:

- Person detailed information
- Basic collection of movies in which he/she participate



3.6.2 Layout

UI layout design was started at the early stage of the project and continuously updated whenever needed. The purpose of it is to ensure the same view on the project layout among the developers. The example can be seen in (fig.14).

The layouts are designed originally for desktop. Before accepting each layout, the possibility of implementing a mobile version was taken into consideration among the developers. The mobile version layouts are not documented.

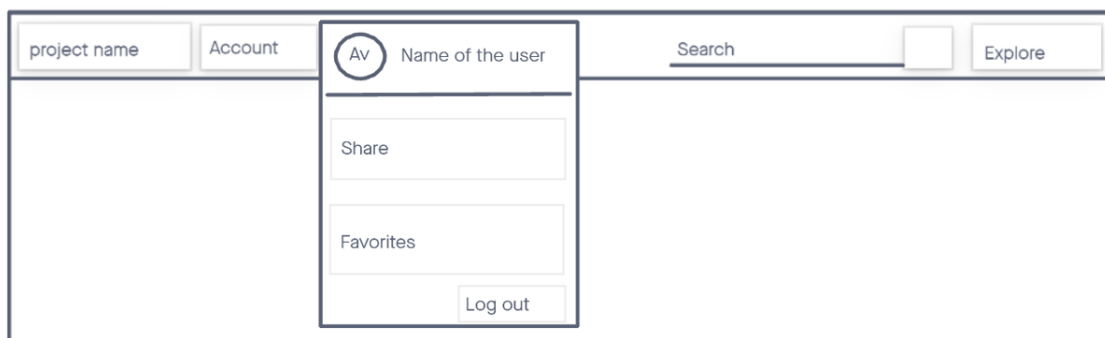


Figure 11 - Layout for "Account"

All layouts can be found in Appendix 7.

3.6.3 Styling

Main styling technologies are described in Front-end - 3.2.1 Technologies.

There is no strict design made for styling the UI. The main rules applies that the text should readable, elements visible and not overlap each other.

4 Implementation

The implementation example is about: "User can see movie details".

In this scenario, the user clicks on any movie as shown below (fig.15).

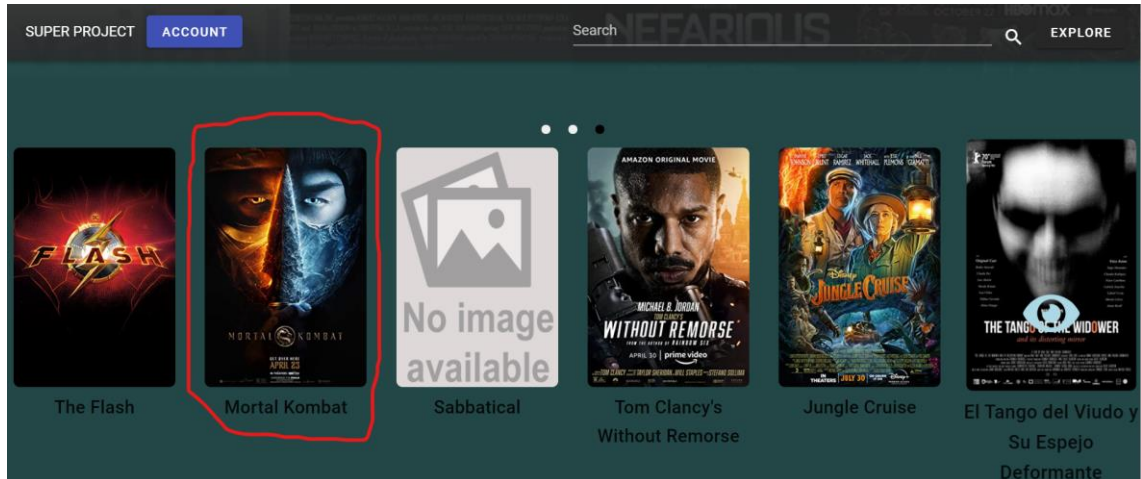
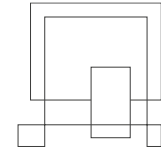


Figure 12 - Home page

For this functionality the following code is called “@click” calls a method “moveToMovie” (fig.16)

```
<v-col class="movie_item col-lg-2 col-md-4 col-6" v-model="movieList"
  v-for="(movie, _key) in movieList" v-on:click="openModal(movie)" @mouseover="activeOver(_key)"
  @mouseleave="removeOver(_key)" :key="_key" @click="moveToMovie(movie)">
  <div class="img_container">
    
    
  </div>
  <div>
    {{ movie.title }}
  </div>
  <div>
    {{ movie.year }}
  </div>
  <font-awesome-icon v-if="showId==_key" class="icon_movie" icon="eye"/>
</v-col>
```

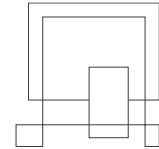
Figure 13 – HTML element from MovieList Component

The “moveToMovie” method routes the user to the movie page with the movie id as a query to the page URL (fig.17).

```
moveToMovie(movie){
  this.$router.push({name:'moviePage',query:{searchQuery: movie.id.toString() }})
}
```

Figure 14 - "moveToMovie" method from MovieList Component

When the movie page is mounted, it calls dispatch to the store which gets the movie details for the page depending on the state of the user. (fig.18)



```
mounted() {
  this.$store.dispatch( type: "clearFirstOrderComments")
  this.loadMoreComments()
  if (this.$store.state.user.loggedIn) {
    this.$store.dispatch( type: "getMovieDetails", payload: {userId: this.$store.state.user.data.uid, movieId: parseInt(this.searchQuery), num: 1})
  } else {
    this.$store.dispatch( type: "getMovieDetails", payload: {userId: 'none', movieId: parseInt(this.searchQuery), num: 0})
  }

  if (this.movie.rating !== null && this.movie.rating !== "N/A" && this.movie.rating !== undefined) {
    this.interval = setInterval( handler: () => {
      while (this.value !== this.movie.rating * 10) {
        this.value += 1
      }
    }, timeout: 1000)
  }
}
```

Figure 15 - "mounted" in MoviePage Component

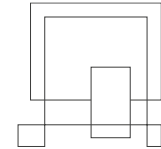
The dispatch from the previous calls the "getMovieDetails" method in the store which uses axios to make an asynchronous call to the backend server. (fig.19)

```
getMovieDetails({commit},{userId,movieId,num}) {
  let url = `${backendUrl}/movies/details/${movieId}/${num}/${userId}`
  console.log(url)
  axios.get(url)
    .then(response => {
      commit('SET_MOVIE_DETAILS', response.data)
    })
},
```

Figure 16 - "getMovieDetails" in sstore/index.js

Back-end gets the request on the specified endpoint which is created by "express.Router()". Firstly, all the parameters are validated by the middleware. Note that there is "userId" parameter in case user is logged in.

Express endpoint is using anonymous asynchronous function with request and response. Request is used to pass parameters to the "movieService.getMovieDetailsAndFavorites" where it also waits for the returned data before it sends response with it. (fig.20)



```

47 router.get( path: "/details/:movieId/:checkFavorites/:userId",
48   param( fields: "movieId").isInt( options: {min:1,max:9999999}),
49   param( fields: "checkFavorites").isInt( options: {min:0,max:1}),
50   param( fields: "userId").custom( validator: (value, {req: Request}) => checkUserIdOrNull(value, req.params.checkFavorites)),
51   validate,
52   async (req: Request<[[p: string]: any], any, any, [[p: string]: any], Record<string, any>>, res: Response<any, Record<string, any>> ) => {
53     const data = await moviesService.getMovieDetailsAndFavorites(
54       parseInt(req.params.movieId),
55       parseInt(req.params.checkFavorites),
56       req.params.userId
57     )
58
59     res.send(data)
60   })

```

Figure 17 - Movie Details endpoint in apis/movies.js

The “getMovieDetailsAndFavorites” from services/moviesService.js delegates the job of getting movie details to other function “getMovieDetails” from the same file. (fig.21)

```

226 module.exports.getMovieDetailsAndFavorites = async (movieId, checkFavorites, userId) => {
227   const data = await module.exports.getMovieDetails(movieId)
228
229   if(!data.hasOwnProperty( v: "error") && checkFavorites == 1){
230     const favorite = await favoritesService.isMovieInUserFavorites(
231       userId,
232       movieId
233     )

```

Figure 18 - “getMovieDetailsAndFavorites” function in MoviesService.js

Next, the model function “getMovieByMovieId” is invoked from models/movieModel.js. (fig.22)

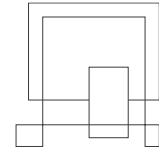
```

243 module.exports.getMovieDetails = async (movieId) => {
244   let movie = await moviesModel.getMovieByMovieId(movieId)
245   if(movie.length > 0){
246     movie = movie[0]
247

```

Figure 19 - “getMovieDetails” in MoviesService.js

This function creates a SQL prepared statement query abstraction which escapes user parameters and passes it to the lower level function “mysql.query” from models/connections/mysqlConnection.js. (fig.23)



```
134 module.exports.getMovieById = async (movieId) => {  
135   return mysql.query(  
136     queryString: "SELECT * FROM movies " +  
137     "WHERE id = ? ",  
138     values: [movieId]  
139   )  
140 }
```

Figure 20 - getMovieById function in models/moviesModel.js

The query function creates new Promise to all the awaiting functions and it passes the parameters it got to connection.query() from MySQL library.

Additionally, it passes call-back function so Promise can get status of rejected if error or Promise can be resolved and pass query result to function caller (fig.24).

```
34 module.exports.query = async (queryString, values) => {  
35   return new Promise( executor: (resolve, reject) => {  
36     connection.query(queryString, values, function (error, elements) {  
37       if (error) {  
38         return reject(error)  
39       }  
40       return resolve(elements);  
41     });  
42   });  
43 }
```

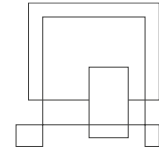
Figure 21 - query function in models/connections/mysqlConnection.js

Thus, the awaiting service function “getMovieById()” can continue due to resolved promise with movie object.

Once “getMovieById” gets the result back it can be returned back to “getMovieDetails” from services/moviesService.js.

Service checks the photo of the movie (fig.25).

First it checks if the movie object is not empty. If its null, it will invoke other asynchronous function from service to get more data from third party APIs. Once it has returned data, it will create new movie object and ask for photos of actors/directors as well. What is more, it will invoke the update of the movie entity in database via other function from the service before it returns the new object of the movie.



```

243 module.exports.getMovieDetails = async (movieId) => {
244   let movie = await moviesModel.getMovieByMovieId(movieId)
245   if(movie.length > 0){
246     movie = movie[0]
247
248     if(movie.posterURL == null){
249       const otherData = await module.exports.getMoreDataForMovieFromThirdParty(movie["id"])
250
251       let newMovie = {
252         ...movie,
253         ...otherData
254       }
255
256       newMovie = await module.exports.getPhotosForPersons(newMovie)
257
258       module.exports.updateDatabaseMovie(newMovie)
259
260       return newMovie
261     }else{

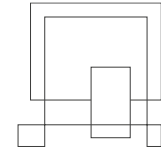
```

Figure 22 - getMovieDetails function in service/moviesService.js

In case the photo of the movie is already in the received object (fig.26). It will still invoke the function for actors and directors photos but this time directly from the database. Additionally, it will ask for the genres of the movie. That is because the movie details needs to be constructed from multiple tables (In first case it gets all the info from third parties).

Then it will place all the extra data to the right properties. At the end it will return a new object of fetched movie parameters and all sorted information.

If the model returned the empty object of the movie to the service at the beginning of the function, function would return the object with the error instead of any real movie data.



```

261     }else{
262
263         const people = await moviesModel.getPeopleByMovieId(movieId)
264
265         const genres = await moviesModel.getGenresByMovieId(movieId)
266
267         let directors = []
268         let actors = []
269         let genresArray = []
270         genres.map((genre) => genresArray.push(genre.genreName))
271
272         for(let i = 0; i < people.length; i++){
273             if (people[i].roleName == "Director"){
274                 directors.push({name: people[i].name, photoURL: people[i].photoURL})
275             }else{
276                 actors.push({name: people[i].name, photoURL: people[i].photoURL})
277             }
278         }
279
280         return {
281             ...movie,
282             directors: directors,
283             actors: actors,
284             genres: genresArray
285         }
286     }
287 }else{
288     return {
289         error: "Movie not found"
290     }
291 }
292 }

```

Figure 23 - getMovieDetails function in service/moviesService.js

Function “getMovieDetailsAndFavorites” from the service has some movie data now (fig.27). Next, it needs to get information if a movie was in a user favourites list. However, it will do that only if “checkFavorites” was set and the received movie data is not an error. In another case, it just delegates the data back to the REST endpoint handler function.

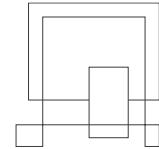
```

226 module.exports.getMovieDetailsAndFavorites = async (movieId, checkFavorites, userId) => {
227     const data = await module.exports.getMovieDetails(movieId)
228
229     if(!data.hasOwnProperty('error') && checkFavorites == 1){
230         const favorite = await favoritesService.isMovieInUserFavorites(
231             userId,
232             movieId
233         )
234         return {
235             ...data,
236             favorites: favorite.exists
237         }
238     }else{
239         return data
240     }
241 }

```

Figure 24 – getMovieDetailsAndFavorites function in services/moviesService.js

Finally, the handler in GetMovieDetails endpoint can send a response with any received data from the service.



The data is then received in the “response.data” as shown below (fig.28).

```
getMovieDetails({commit},{userId, movieId, num}) {
  let url = `${backendUrl}/movies/details/${movieId}/${num}/${userId}`
  console.log(url)
  axios.get(url)
    .then(response => {
      commit('SET_MOVIE_DETAILS', response.data)
    })
},
```

Figure 25 - getMoviesDetails in store/index.js

The “SET_MOVIE_DETAILS” then sets the state of the “movieDetails” properties with the response from the server (fig.29).

```
SET_MOVIE_DETAILS(state, movieDetails){
  console.log(movieDetails)
  state.movieDetails = movieDetails
},
```

Figure 26 - "SET_MOVIE_DETAILS" in store/index.js

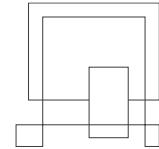
The “movieDetails” will then trigger an update on the computed properties of the movie page such as if the movie is a favourite of the user, the details of the movie and the directors (fig.30).

```
computed: {
  isFav() {
    return this.$store.state.movieDetails.favorites
  },
  movie() {
    return this.$store.state.movieDetails
  }
},
```

Figure 27 - computed in MoviePage.vue

Then the computed properties are used in the template to fill in the values of the movie page dynamically.

The figure below is a simple use case of how the computed properties are used around the component as the movie is used to get the poster URL which is used to display the image of the Movie (fig.31).



```
<div class="col-md-5 col-12 image_favorite">
  
  

  <font-awesome-icon @click="removeFav" v-if="isFav===true" class="icon_movie" style="color:hotpink"
    icon="heart"/>
  <font-awesome-icon @click="addFav" v-else class="icon_movie" style="color:gray" icon="heart"/>

</div>
```

Figure 28 - HTML element in MoviePage.vue

The resulting view of this will be a movie Page view with all details well placed and functional is shown below (fig.32).

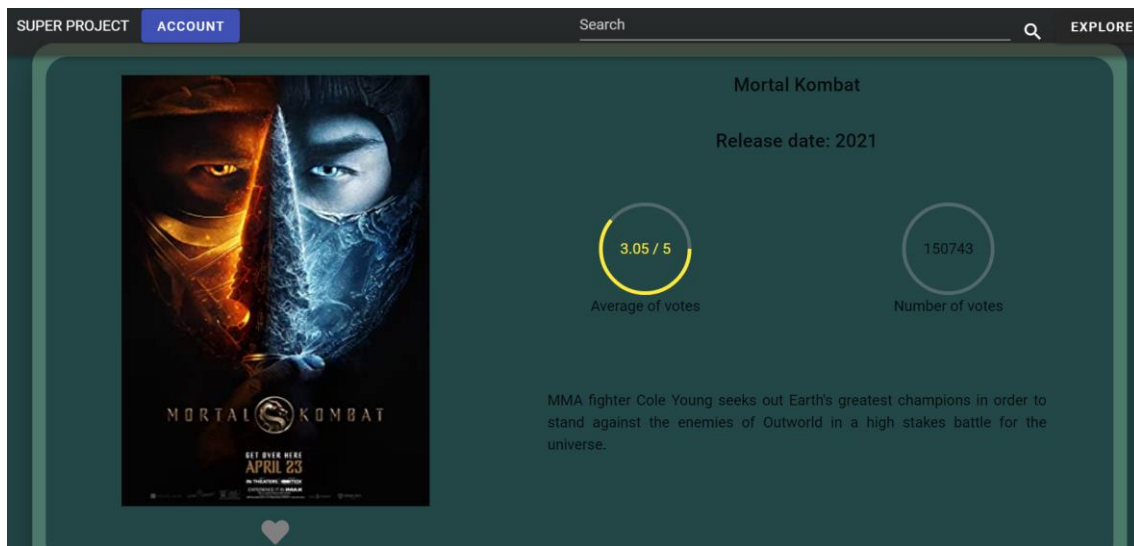
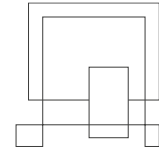


Figure 29 - movie page with details

All source code for both front-end and back-end can be found in Appendix 4

5 Test

Tests were used to verify that the functionality made in a user story worked. The applications have been tested using a combination of white box and black box unit testing.



5.1 Back-end

White box testing and a variety of testing techniques were used to test the back-end.

5.1.1 Supertest

Integration tests were performed on the API endpoints. A library called supertest was used to simulate calls to the API and test one endpoint at a time. The test is an integration test because it tests not only the ability of the API to respond to calls, but also the functionality of the input validation that the express validate library provides. Boundary value analysis is performed on each of the parameters in the endpoints to make sure the validation is working.

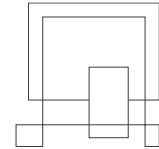
5.1.2 Sinon

Unit tests with a strategy of branch testing were performed on the services layer which is the business logic of the back-end. A testing library called Sinon was used to stub and return specific values from all external functions that a function that is being tested calls. Specifying the output of the stubbed functions allowed to test the specific scenarios that occur in the back-end.

5.1.3 Back-end test results

The amount of tests for each user story can be seen in the table below (table.1):

User stories	Tests done
See list of movies	7 integration, 20 unit
Search for movies	6 integration, 1 unit
See data related to movies	6 integration, 6 unit
Login	10 integration, 3 unit



Create list of their favourite movies	20 integration, 11 unit
Choose category of movies to look at	1 integration, 2 unit
Comment on movies	16 integration, 3 unit
Middleware that affects all user stories	16 unit
Total:	66 integration, 62 unit

Table 1 - Back-end test results

5.2 Frontend

Cypress is used in the project to perform end-to-end testing. We take real user scenarios and execute as would be in real life. However, due to the drawbacks of cypress which includes the inability to test third-party services. Therefore, there was an inability to test the login and favourites test cases however, we have performed this test manually and are represented in the test case specifications.

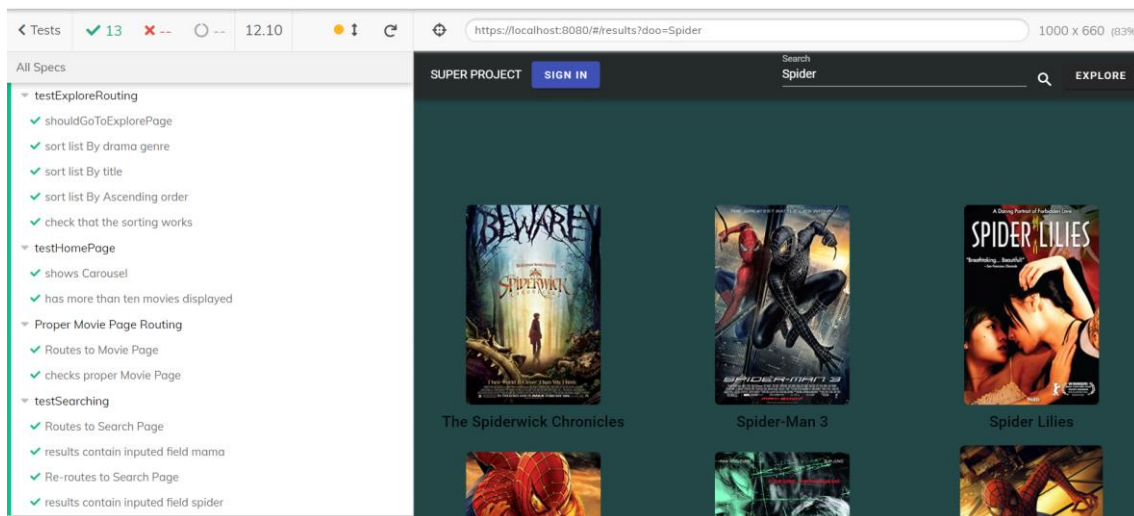
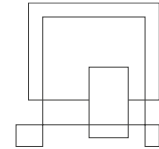


Figure 30 – E2E test results shown in Cypress Dashboard

In the figure above (fig.34), we perform four e2e testing which includes test Explore Routing where we ensure 5 test cases; The user properly routes to explore page, then we sort the list by genre, title, order and then we ensure that the list is correctly sorted as such.

The code below shows the e2e testing fully Implemented (fig 35).



```
describe('testExploreRouting', fn: ()=>{

  let text =[]

  it('title: 'shouldGoToExplorePage', config: ()=>{
    cy.visit('https://sep6-front-end-an6w7okvaa-uc.a.run.app/#/')
    cy.get('#explore_button').click()
  })

  it('title: 'sort list By drama genre', config: ()=>{
    cy.get('[data-cy=genre_select]').parent().click( options: {force: true})
    cy.get(".v-list-item__title").contains( content: "Drama").click()
  })

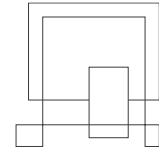
  it('title: 'sort list By title', config: ()=>{
    cy.get('[data-cy=sortBy_select]').parent().click( options: {force: true})
    cy.get(".v-list-item__title").contains( content: "title").click()
  })

  it('title: 'sort list By Ascending order', config: ()=>{
    cy.get('[data-cy=order_select]').parent().click( options: {force: true})
    cy.get(".v-list-item__title").contains( content: "ASC").click()
    cy.get('.movie_item').first().get( selector: '.title').
    each( fn: (el : JQuery<HTMLElement> )=>{
      let title = el[0].innerHTML.trim()
      // || /\d/.test(title) lets forget about numbers cause they are not
      if(/^([A-Z])/\.test(title)|| /\d/.test(title) ){
        text.push(el[0].innerHTML.trim().toLowerCase())
      }
    })
  })

  it('title: 'check that the sorting works', config: ()=>{
    assert.isTrue(!text.reduce((n, item) => n !== false && item
      >= n && item), message: 'this val is true')
  })

})
```

Figure 31 – E2E test in Cypress



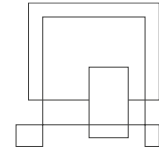
5.3 Test Specifications

Test specifications were made based on the user stories. Therefore, each test case Id is unique and is combination of user story id it relates to and test id (i.e., U03T01). Test steps were made to follow Black-box testing approach.

What is more, test cases consist of test case description, test data, postcondition, precondition and expected result.

Additionally, the result section helps to outline the actual result with following test result status – passed or failed.

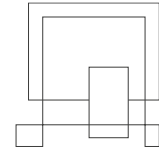
Due to limited space in portrait orientation of the document, the Test Specifications table can be found in Appendix 6.



6 Results and discussions

Note that User Stories in table below are without acceptance criteria. However, they are still taken into consideration and included in test cases (table 2). All test cases can be found in Appendix 6.

No.	User Story	Fully Implemented / Partially Implemented / Not Implemented	Reference To Test Cases	Discussion
1	See a collection of movies, so that I can find the movies I am interested in.	Fully Implemented	U01T01	
2	Search for movies, so that I can find the specific movie I am interested in.	Partially Implemented	U02T01, U02T02, U02T03, U02T04, U02T05, U02T06	The test case U02T05 fails on the “%” search input – this causes a crash on the back-end.
3	See data related to a movie, so that I see if the movie match my interest.	Fully Implemented	U03T01	Additionally, extra data is displayed: number of votes, picture, year of release (see acceptance criteria in User Stories).
4	Login, so that I can authenticate, keep track of my favourite movies and join social activities.	Fully Implemented	U04T01	Google Sign-In works. Additionally, Facebook Sign-In was implemented and it works with the issue of not always displaying a picture of the user.
5	Manage a list of favourite movies so I can share them with my friends.	Fully Implemented	U05T01, U05T02, U05T03, U05T04, U05T05, U05T06	
6	Sort collection while exploring movies so I can find movies that match my interest.	Fully Implemented	U06T01, U06T02, U06T03, U06T04	
7	Comment on movies so I can discuss my opinion about the movie with other users.	Partially Implemented	U07T01, U07T02, U07T03, U07T04, U07T05, U07T06,	5 out of 11 test cases are passed.



			U07T07, U07T08, U07T09, U07T10, U07T11	General comments work as they should with the issue of not validating the max size of comment. Reply comments are not implemented.
8	Search for the actor, so that I can find information about the actor and movies related to him.	Not Implemented	None	
9	See data related to an actor so that I can see what other movies the actor stared in.	Not Implemented	None	
10	See how many people viewed the movie on the page, so that I can see how popular the movie is.	Not Implemented	None	

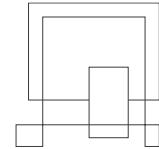
Table 2 - Table of fulfilling the requirements

6.1 Results

As the Table of fulfilling requirements shows, five out of ten user stories were fully implemented (some with additional features). Two out of ten user stories were partially implemented. Three of the user stories were not implemented at all.

7 Conclusions

At the end of this project, the project has been mostly successfully implemented as previously discussed, 50% of user stories had been fully implemented. The Project has implemented what has been identified as a priority. However, there are a lot of weaknesses which can be found in the system as previously discussed. The system had a very low styling rate which can be seen. The theme is not the best, as well as the carousel cannot navigate to a movie these are few of minor stylings that should have been implemented better in the system. The team focused fully on the functionality more so than the general looks of the system. Even though the team focused on functionality than styling, there are lots of functionalities that were not implemented and could be done in the future. The search has a flaw as it is possible to search for special characters yet it breaks after the request returns an error. The user should have been able to reply to comments to give the app a more social app but due to time constraint



this was not implemented. Another minor weakness in the system is the fact that when two users are commenting on the same movie. The users do not see each other change until reload. The team thinks that this should probably be fixed with reactive programming but should be noted for the future. The favourites page and sorting on the explore page work flawlessly and can be seen as a great strength of the project. The other requirements could be implemented further in the future. All in all, the project has had its strength and weakness and would be much better if more time were to be put towards the project.

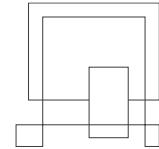
8 Project future

UI design - From the among aspects to improve, better UI styling would bring a big improvement to the project. What is more, improved layout design could bring a better user experience.

Modules for Vuex state – Split of the state into different modules and files would ensure the separation of concerns. Therefore, the front-end code would be better structured. A future task could be easier to distribute among the developers.

Reactive programming – Any reactive observer library (i.e., RxJs) would be a big advantage for the project. Vue can update the components dynamically i.e., after sending the comment it is shown right away in the comment section without refreshing the page. However, if there is someone else sent the comment, it is required to refresh the page. Reactive programming would solve the issue.

Terraform – The process of initializing the cloud infrastructure is very time consuming and complicated. Having an infrastructure as a code platform could benefit the project by allowing to reproduce the cloud infrastructure at a click of a button. Changes to the infrastructure can be made safely and in a more constructive way as opposed a messy google cloud UI. There was an attempt to use Terraform earlier in the project with failed results. It would be good to attempt it again.



9 Website URL

<https://sep6-front-end-an6w7okvaa-uc.a.run.app/#/>

10 References

Anon., n.d. *Agile Modeling*. [Online]

Available at: <http://agilemodeling.com/artifacts>

[Accessed 2021].

Anon., n.d. *Medium*. [Online]

Available at: <https://medium.com/front-end-weekly/why-choose-vue-js-over-any-other-frontend-framework-for-better-ui-16a8de5567d2>

[Accessed 2021].

Anon., n.d. *Vuex - state management pattern*. [Online]

Available at: <https://next.vuex.vuejs.org/#what-is-a-state-management-pattern>

[Accessed 2021].

Google cloud, n.d. *Node.js on google cloud*. [Online]

Available at: <https://cloud.google.com/nodejs>

[Accessed 2021].

Google cloud, n.d. *Price of cloud compute*. [Online]

Available at: <https://cloud.google.com/compute/all-pricing#disk>

[Accessed 2021].

Microsoft Azure, n.d. *Node.js on Azure*. [Online]

Available at: <https://azure.microsoft.com/en-us/develop/nodejs/>

[Accessed 2021].

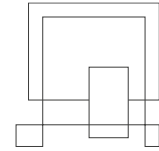
NISO, 2010. *Scientific and Technical Reports -*, Baltimore: National Information Standards Organization.

StackOverflow, 2020. *2020 Developer survey*. [Online]

Available at: <https://insights.stackoverflow.com/survey/2020#technology-programming-scripting-and-markup-languages-professional-developers>

[Accessed 2021].

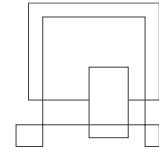
VIA Engineering, in preparation. *Confidential Student Reports*, s.l.: s.n.



Vuetify, n.d. *Vuetify*. [Online]

Available at: <https://vuetifyjs.com/en/introduction/why-vuetify/#comparison>

[Accessed 2021].



11 Appendices

Appendix 1 – Cloud User Guide

Appendix 2 – UML Diagrams

Appendix 3 – Use Case Descriptions

Appendix 4 – Source code

Appendix 5 – Architecture

Appendix 6 – Use Case Testing

Appendix 7 – UI Layout