TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Ivan Tsud 177217IASM

DESIGN AND DEVELOPMENT OF MOBILE-ORIENTED SOCIAL NETWORK. SERVER-SIDE

Master's thesis

Supervisor: Eduard Petlenkov
Professor

Tallinn 2019

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Ivan Tsud 177217IASM

MOBIILSE SOTSIAALVÕRGUSTIKU PROJEKTEERIMINE JA ARENDAMINE. SERVERI POOL

Magistritöö

Juhendaja: Eduard Petlenkov
Professor

Tallinn 2019

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Ivan Tsud

05.03.2019

Abstract

This thesis is written in English language and is 53 pages long, including 8 chapters, 13 figures and 1 table.

Keywords: social network, server-side, node, express, database, programming language.

The object of research: server-side development for mobile platforms.

Theoretical research was implemented by studying the resources provided in the References section.

In this master thesis, the server-side for the social network has been designed and implemented.

It has the following features:

- User Sign In and Sign Up,
- User authentication,
- Password reset,
- File storage setup,
- Routing security.

The practical value of the project is that it can be used as a starter, for any kind of mobile application that needs describing above functionality.

List of abbreviations and terms

MVC	Model View Controller
RAD	Rapid Application Development
I/O	Input / Output
ES6	ECMAScript 6
ES8	ECMAScript 8
RDMS	Relational Database Management Systems
ORM	Object-relational Mapping
ACID	Atomicity, consistency, isolation, and durability.
DBA	Database Administrator
NoSQL	Not only SQL
API	Application Programming Interface
REST	Representational State Transfer
URI	Uniform Resource Identifier
DAO	Data Access Object
npm	Node Package Manager
CTO	Chief Technical Officer

Table of Contents

List of Figures

List of Tables

1 Introduction

Within this master thesis, I want to develop a Back End starter for the mobile application. The main reason for that is the development of new skills and moving to Full Stack development. As working as a Frontend Developer anyway, sometimes I find myself digging into some backend stuff because Backend developer is extra busy and I have to find a quick solution that will work. Anyway, that experience is so negligible, so that can't be related to the experience at all.

That is why I decided to try myself in real backend development.

The idea (Oleksandr is the owner of the idea) of the mobile app is the following:

The main point for a target user is an improvement in the social aspects of everyday life. With the MovieGo app, it is possible to grab your phone and see the most relevant movies in a nearby cinema, choose the one you like, buy tickets and go!

But in contradiction to other similar services – the user does not have to go alone (in case if no friends are available right now). It is possible to see other people who are interested in a particular movie at the same time which is suitable for you in the same cinema.

With just pressing a button, you already have a good plan for the evening, as it is not ending at the cinema as you just made a new friend, so why not to prolong your good time?

Have a walk, discuss the movie you have just watched or do whatever you want.

Before, when people wanted to visit a cinema, they used checking service, buying tickets, watching the movie and that was it.

But it much more interesting to watch the movie with someone as it is becoming not only up to a movie then.

Another important point of the app is to encourage people to do charity. It is not only about the customers but also cinemas as well.

It is possible for cinemas to give the customers an opportunity to buy a ticket a bit cheaper and spend the difference on charity. Thus, it will encourage users to visit the cinemas more and more often for various reasons: tickets are cheaper, now it is much more entertaining than before, and along with all these – they can help others. And everybody wins!

From the description, provided above, I figured out that the following backend parts should be developed and configured:

- Server setup.
- Database setup.
- Media files and their storage.
- User authentication.
- Server routing.
- Basic security.

The minimum plan for this master thesis would be:

- A basic server that can handle requests from the client and sends back responses.
- A simple database configuration to store user's data.
- User authentication with email and password.

2 Choosing Backend Technology Stack

Having no previous backend experience results in a huge problem in choosing the right and reliable backend technology stack for the project as choosing the right one results in a successful launch of the product. Backend development as any other development types covers not only writing code with no bugs, but also optimization, scalability, and security issues. Only the thoroughly done analysis can assure the correctness of the chosen technology stack.

2.1 General Reasoning

As for me, the next reasons drive me when choosing the Tech Stack:

1. My personal sympathy to the stack.
2. Advice from a friend who has experience in the given development direction.
3. Somewhat same stack was used for a similar service, app, website, etc., because of the possibility to use their "best practices" guides.

But there are problems arising from the stated above reasons:

1. Sympathy to the stack is way too subjective. What if the stack is not quite appropriate for the current project or this technology going to be a legacy one?
2. Advice from a friend can be dangerous (except if he is not the CTO in a big corporation, even though it is debatable), as he can’t know all the possible stacks and also has subjective opinions.

3. Similar service may use the stack because of “just because” or “historical” reasons. Maybe the stack has been used only because of popularity, or they used legacy solutions of their own development.

So, nothing from the mentioned above is good reasoning for choosing the Tech Stack and, the first step will be to specify the important, objective criteria.

After the research, I came up with the following criteria for the criteria while choosing the Tech Stack:

1. Size and type of the project.
2. Complexity of the project.
3. Speed of implementing the technological aspects.
4. Development tools that are available at the moment.
5. Availability of already-implemented solutions.
6. Flexibility.
7. Community within the technology.
8. Trend.
9. Reliable and good specifications and documentation.
10. Support price.
11. Load requirements.
12. Security requirements.
13. Cross platform capabilities.
14. Possibility of integration with other solutions.

2.2 Programming Languages and Frameworks

Concerning the programming languages, it can be classified by 2 principal abstractions:

1. Vanilla programming language – let me say that it is a material that can be used to build anything. The only thing that limits a developer – language restrictions. On the top of vanilla languages, such products as Instagram [1] , YouTube [1] , Amazon [1] are built.

2. Framework – is a unique environment for the development that has built-in rules and tools which speed-up and simplifies the development process. But every framework has limitations.

It is obvious that the bigger the project, the bigger the technology stack is going to be used in there. While building huge products as usual developers use several languages as one programming language can be good for solving one specific task, the other one – another task. For example, Google uses [1] a plenty of different languages, like JavaScript, Objective-C, Python, Dart, and others. That is needed because millions of people use Google every day, and there is a need to sustain the stability of the system.

I considered the following languages for backend:

1. Python – modern language that allows quick and easy development [2] . But mainly I have heard about it in the Data Science sphere.

2. Ruby – modern language that allows quick development with the focus on simplicity [3] .

3. Java – old language, with the postulate on hard and long development. Usually used for huge project with specific business logic [4] .

4. JavaScript – quite old, but rapidly develops, is the most trending one with huge community, and used for any kind of project [5] .

The frameworks and platforms related to the mentioned languages:

1. Django — open-source framework for backend web applications based on Python. Its main goals are simplicity, flexibility, reliability, and scalability. Django has its own naming system for all functions and components. It also has an admin panel, and other technical features, including:
   - Simple syntax.
   - Its own web server.
   - MVC core architecture.
   - HTTP libraries.
   - Middleware support.
   - A Python unit test framework. [6]
2. Ruby on Rails.
   - The process of programming is much faster than with other frameworks and languages, partly because of the object-oriented nature of Ruby and the vast collection of open source code available within the Rails community.
   - The Rails conventions also make it easy for developers to move between different Rails projects, as each project will tend to follow the same structure and coding practices.
   - Rails are good for rapid application development (RAD), as the framework makes it easy to accommodate changes.
   - Ruby code is very readable and mostly self-documenting. This increases productivity, as there is less need to write out separate documentation, making it easier for other developers to pick up existing projects.
   - Rails have developed a strong focus on testing and have good testing frameworks. [7]

3. Spring.

   The Spring Framework provides a comprehensive programming and configuration model for modern Java-based enterprise applications — on any kind of deployment platform.

   A key element of Spring is infrastructural support at the application level: Spring focuses on the "plumbing" of enterprise applications so that teams can focus on application-level business logic, without unnecessary ties to specific deployment environments. [8]

4. Node.

   Actually Node is not a framework, but platform, on top of which the following frameworks are built:

   1. Express – is the most popular framework in Node community. It is minimal and flexible. Doesn't enforce developer in using any pattern. It is also one of the best performing frameworks. [9]

   2. Koa – is even more minimalistic framework than Express. It also has more of recent features from JavaScript. [10]

During the last few years Node.js became extremely popular and because of that has had explosive growth, mainly because of its top performance and the benefit that allows you to build End-to-End JavaScript applications (both server and client), making the systems more scalable and maintainable as a whole.

The major factor that makes Node performance better over other frameworks is because of its non-blocking, event-driven I/O with a single thread paradigm, making servers handle all requests in a single thread, allowing much better scalability than traditional multithreaded servers.

The main drawback of the single thread paradigm is that the processes that need a huge amount of calculations, could in the end block the thread for unreasonably long time that resulting in overall performance of the server. [11]

2.3 TIOBE Index

Table 1. TIOBE Index for March 2019. [12]

March 2019	March 2018	Programming language	Rating	Change
1	1	Java	14.880%	-0.06%
3	4	Python	8.262%	+2.39%
7	8	JavaScript	2.426%	-1.49%
15	9	Ruby	1.202%	-1.54%

From the Table 2 it is clearly seen that the leaders are Java and Python. Nevertheless, JavaScript gained a lot of popularity during last years due to introducing new features (like ES6 to ES8).

2.4 Conclusions

After all the research and comparing the benchmarks [13] JavaScript, particularly Node.js with Express.js framework seem to be the best choice, due to the performance, huge community, popularity (the stack will be trendy for several years from now), flexibility, scalability, development tools available, possibility to integrate with other solutions, and ability to use the same codebase on the client and server.

3 Choosing the Database

Choosing the right database is one of the most important key things developers should consider while starting a new project.

3.1 Database Considerations and Requirements

The database should be capable of doing the following with the entries stored there: create, read, update and delete.

But there are some other functional and non-functional system considerations and requirements [14] :

1. Consistency, availability, and partition tolerance.
2. Robustness and availability.
3. Scalability.
4. Performance and speed.
5. Distributability.
6. In-database analytics and monitoring.
7. Operational and querying capabilities.
8. Storage management.
9. Data model flexibility.
10. Database security.

3.2 Relational Database Management Systems (RDMS)

Relational database management systems are used for storing and querying structured relational data. Relational data is the data stored in different tables of the database and often each table has a relation to a different table. Relations could be one to another, one to many or many to many. Each table consists of rows and columns, with a unique identifier per row.

Data often needs to be stored and accessed across multiple tables due to its relational nature, and this creates a mismatch between applications and database data representations that is usually referred to as the object-relational impedance mismatch.

This mismatch occurs more specifically due to the need to map software objects to database tables created from relational schemas. In order to address this mismatch, various architectural patterns and software applications have been created to map software objects to database tables and vice versa. This is known as object-relational mapping (ORM).

RDMS are generally focused on providing strong consistency and ACID[1] transactions, stability, and strong reliability. They are also very well-suiting for complex querying and non-real time analytics applications. Atomicity indicates that a transaction is all or nothing, which means that any failure in a transaction causes the entire transaction to fail and therefore leaves the database unchanged. Consistency ensures that all transactions result in a valid state of the database, and that all validation rules and constraints are met, and required actions are carried out. Isolation means that the database is able to perform concurrent transactions that result in the same database state as if the transactions had occurred one after the other. This is a form of concurrency control, which can be implemented in different ways, with serializability referring to the highest level of isolation. Durability ensures that a committed transaction persists, or is permanently stored under any and all conditions.

[1] ACID stands for database transaction properties and includes atomicity, consistency, isolation, and durability.

Lastly, RDMS systems can be complex to install, manage, take full advantage of (e.g., complex queries, stored procedures, triggers, …), and optimize. As a result, these systems have traditionally been worked on by highly specialized people with roles such as Database Administrator (DBA), and less so by software engineers. [14]

3.3 NoSQL Databases

NoSQL database systems were created for, and have gained widespread popularity primarily due to benefits relating to scalability and high availability. They favour the eventual consistency model, and these systems typically model and store data in ways other than the traditional tabular relations of relational databases.

These systems are also characterized as being modern web-scale databases that are typically schema-free (dynamic schemas, and do not suffer from schema rigidity), provide easy replication, have simple APIs.

There are multiple types of NoSQL databases, with document, key-value, graph, and wide-column being the most prevalent. The different types refer mainly to how the data is stored and the characteristics of the database system itself.

Each type involves tradeoffs between preferred data model, simplicity, querying and operational capabilities, partition ability, consistency, availability, performance, tolerance for object impedance mismatch, and so on.

Benefits of NoSQL databases are simplicity, scalability, flexibility, speed, low latency, high availability, high throughput, and partition tolerance.

Unlike RDMS systems, many NoSQL databases can be easily installed, managed, and almost fully utilized by software engineers without requiring a DBA background. This is important since they do not require multiple highly specialized roles, and since having the same people write software and database queries/operations is highly efficient, more agile, faster, and less error-prone.

The simplified database API and querying language used by many of these systems are also very software engineer-friendly and familiar, with JSON as a primary example. This allows for common and developer-friendly language usage between database and application coding. [14]

3.4 Conclusions

Due to the described above facts, I would choose the NoSQL database because of the following reasons:

1. Simplicity.
2. Availability to have flexible schema.
3. No need to write any complex queries.
4. High speed of the operation.
5. High scalability.
6. JSON, XML data formats.

4 REST API vs GraphQL API

4.1 REST API

REST is an architecture for building web services. It follows principal directives, making users ask for actions and resources through a URI and an HTTP method (GET, POST, PUT, DELETE), and making transactions stateless.

REST actually does a pretty good job in most of the cases but following its conventions usually results in getting more data than needed or getting less data, forcing doing more requests. [11]

4.2 GraphQL API

GraphQL API has a data schema and a set of manipulations that client requests. A client accesses the API with requests to a single URI, and a JSON like a body that specifies which data wants to be received and which manipulations have to be executed, allowing them to specify completely what fields they want to receive and how.

This new paradigm allows much more flexible interactions between client and server, reducing the overall amount of data and requests that need to go through the network. [11]

4.3 Conclusions

GraphQL seems really a good replacement for REST, but due to the fact that the technique is a new one, there are no good reference and patterns.

So, for now, I would use REST and after some time will migrate to GraphQL if it still be alive.

5 Choosing the Architecture

5.1 Monolithic Architecture

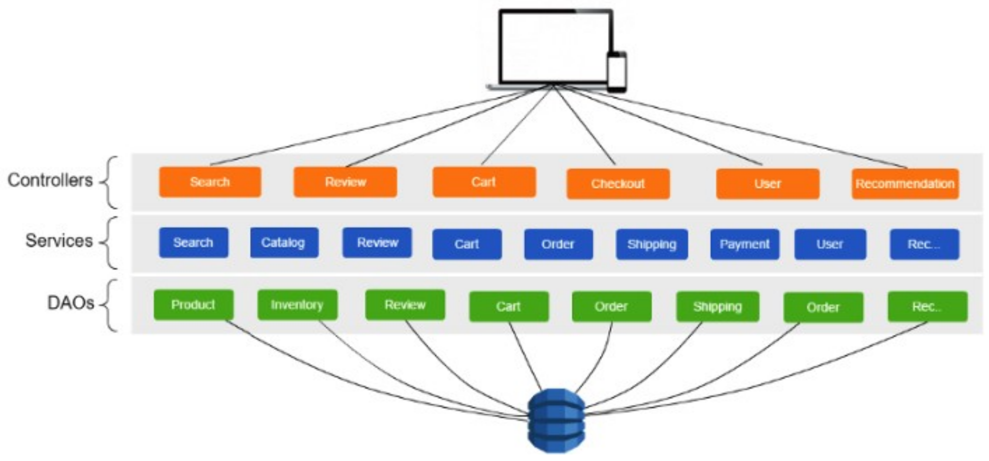


Figure 1. A typical monolithic architecture. [15]

The top layer will generally handle the client requests and after doing some validations it will forward the request to service layer where all the business logic is implemented. A service will make use of various adapters like database access components in DAO layer, messaging components, external APIs or other services in the same layer to prepare the result and return it back to the controller which intern returns it to the client.

Such applications are generally packaged and deployed as a monolith. Applications like these are pretty common and have many advantages, they are easy to comprehend, manage, develop, test and deploy.

But it has drawbacks:

1. Language / Framework lock: No way in experimenting and injection of new technologies as the entire application is written in a single tech stack.
2. Difficult to understand: Once the app becomes larger it becomes difficult to understand such a large codebase.
3. Deployment of a single feature: No way to deploy a single feature independently. [15]

5.2 Microservices Architecture

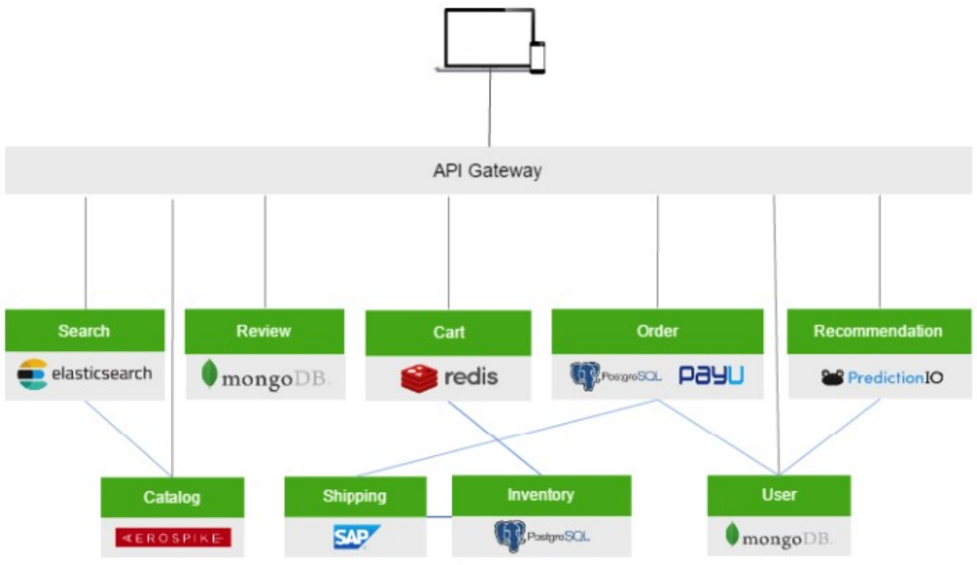


Figure 2. A typical microservices architecture. [15]

Microservices architecture solves a problem by dividing it into smaller sub-problems. Each microservice implements its own function. Each service can be written in any language/framework that suits the use case.

All requests from clients first go through the API Gateway. It then routes requests to the appropriate microservice. The API Gateway will often handle a request by invoking multiple microservices and aggregating the results. It might have other responsibilities such as authentication, monitoring, load balancing, caching and static response handling. Since this gateway provides client-specific APIs it reduces the number of round-trips between the client and application which reduces network latency and it also simplifies the client code.

Microservices also brings some complex challenges:

1. Distributed Computing Challenges: Since different microservices will need to run in a distributed environment it will be needed to take care of the behavior and the locations of the components of the system will constantly change.
2. Handling Service Unavailability: It is needed to design the system to handle unavailability or slowness of services. [15]

5.3 Conclusions

It is understandable that monolithic architecture has scalability and maintainability problems, but as of now, I see no point in setting up the microservices architecture.

Probably, with the future development of the product, there will be a reason to move to the microservices architecture.

6 Project Structure

In this section I will go deeper into planning of the project structure and main components.

In general, project should consists of 3 main components:

1) React Native UI library. It should be deployed as a package to `npm` [16] because of the following reason: it will allows us to use the `Leuchtturm` for any future React Native project and have a ready solution for the UI. We also will be writing documentation for the package as we go. The documentation should has description for all implemented UI components. For now, we considering to use `docsify.js` as a documentation site generator [17] . We also considering of usage Storybook, as it will allow write self-documenting components [18] . But it will consume quite a lot of time to start the project, so it will be implemented later. The static website with documentation will be deployed to the Netlify, because it is one of the easiest and fastest ways to host static websites for free [19] . It has all features we need: it automatically deploys pushes to the master branch, allows custom domain names, allows modify build preferences, deploys everything in an ease. All the UI components part will be explained by Oleksandr in more details.

2) React Native Client App “MovieGO”, is going to be built with React Native framework for the development of cross-platform apps for iOS and Android [20] . We also will write documentation as we go, to keep all main aspects in one place. We will write client part together. Oleksandr will be focusing on UI part, while I will be covering functional part, that is related to the server (making request, upload files).

3) And last, but not least, Express Node Server. This is the core of the application as it handles all the requests and responses for the Client App. Node has been taken as a base for the server [21] . Express has been taken as a core framework for Node [9] . The main reasoning of using Node and React Native – the one codebase. For a database for the app, I have chosen MongoDB [22] . The database will be hosted on mLab [23] . The main reason of choosing MongoDB and mLab because there is a minimal setup needed, it is free for the beginning and as a whole it has Database as a Service features, that simplifies the development and configuration. And finally, for now, the server will be deployed to the Heroku [24] . Once there will be a need in several separate microservices, probably the server will be migrated to the Docker [25] .

   The reasoning of choosing the stack was in the corresponding sections above, dedicated to the comparison of all considered services, products, languages and technologies.

All these 3 projects going to live on the Bitbucket [26] . The main reason why we have chosen Bitbucket – because of ability to have a free private project for collaboration for a team of less than 5 people.

7 Server Setup

The server will be configured with the following stack:

- Node.js as a runtime environment.
- Express.js as a Node.js Framework.
- MongoDB as a NoSQL database.
- Passport.js as a middleware for User Authentication.

Project structure:

- **./src/** - is the main folder with the source files.
    - **/avatar-upload/** - folder that will contain all the configuration for the helper that will handle uploading of the user's avatar to the server.
    - **/controllers/** - folder that will contain all the controllers (core things that will manipulate accordingly and trigger the Model or make actions to interact with View).
    - **/db/** - folder with the config of the database. See Figure 1. Part of the source code of ./src/db/index.js for reference.
    - **/mail/** - folder that will contain config for reset password emailing flow.
    - **/middlewares/** - folder that will contain middlewares that will serve as helpers for the core controllers.
    - **/models/** - folder that will contain defined schemas of the database instances, like User structure, structure of the Film, etc.

    - **/passport/** - folder that will contain configuration files for the PassportJS middleware.
    - **/routes/** - folder that will contain logic of the application routing.
    - **/app.js** – there will be initialized Express server, routes, and all other stuff related to the Express, authentication middlewares and their configs. See Figure 2. Part of the source code of ./src/app.js for source code example.
- **./start.js** - entry point of the whole application. It runs server, connects to the database, links environment variables, etc. See Figure 3. Part of the source code of ./start.js for more details.

7.1 User Authentication

Before creating core microservices and doing any server logic side, the first thing, that should be implemented – User Authentication. It allows verify the user identity and protects the routes if the access to the user is not granted.

7.1.1 User Sign Up Flow

When a new user signs up, he/she sends a bunch of personal information from the React Native App to the Server. All the data is being sent by a single POST request, that comes to the https://moviego-express-server.herokuapp.com/user/register. The information that is being sent (fields) presented below:

1. Username – is a unique combination of the characters that will identify user among others in chats.
2. Email,
3. Password,
4. Confirm password,

5. Avatar Uri,
6. First Name,
7. Last Name,
8. Birth Date,
9. Latitude,
10. Longitude,
11. Gender.

For more information about the fields, please see Chapter 7.2 where all the information about Database Schema set up will be placed.

The logic of Sign Up Flow:

The Route Handler verifies whether the endpoint is valid. Basically, the route handler check whether there is a set-up for a route the client trying to reach. At the time, I have the following route setups:

- `/` - home route, that just will display a dummy information about the server.
- `/avatar-upload` - route that handles all the necessary steps to upload avatar to Amazon AWS S3 [27] and is described in more details in section Avatar Field.
- `/user/register` - route that handles registration for a new user. If the `/user/register` route can be handled, the data is passed to the Data Validation Middleware. See Figure 4. User Middleware to Validate data for more information.

  This middleware checks whether all the needed data is present and data has correct formatting. This middleware runs sanitization for major fields, so we are sure that fields does not include script tags in the username, firstName and lastName. `sanitizeBody()` method can be accessed on `req` because in the `app.js` I inject `app.use(expressValidator())`, so it adds a bunch of methods available on 'req' for validating data. And if I have a request, I can just call different validation methods that live on the top of it, without importing a library again and again.

As a result, all the data is being standardised and then passed to the Registration Controller. See Figure 5. Register Controller for more information.

The register() controller creates a new user instance of off the obtained fields from request. Then by calling a `register()` method on the User model, where internal parameters are: created user instance, password and callback function. The password is sent to the Passport.js middleware in a plain form and then it is saved to the database in a form of a hash. It is possible to call register() method on User model because in while setting up User model, the passport-local-mongoose [28] plugin was added there.

And then, all the data composed to a single User object and passed to the MongoDB with the help of Mongoose. The flow diagram is presented below in Figure 3.

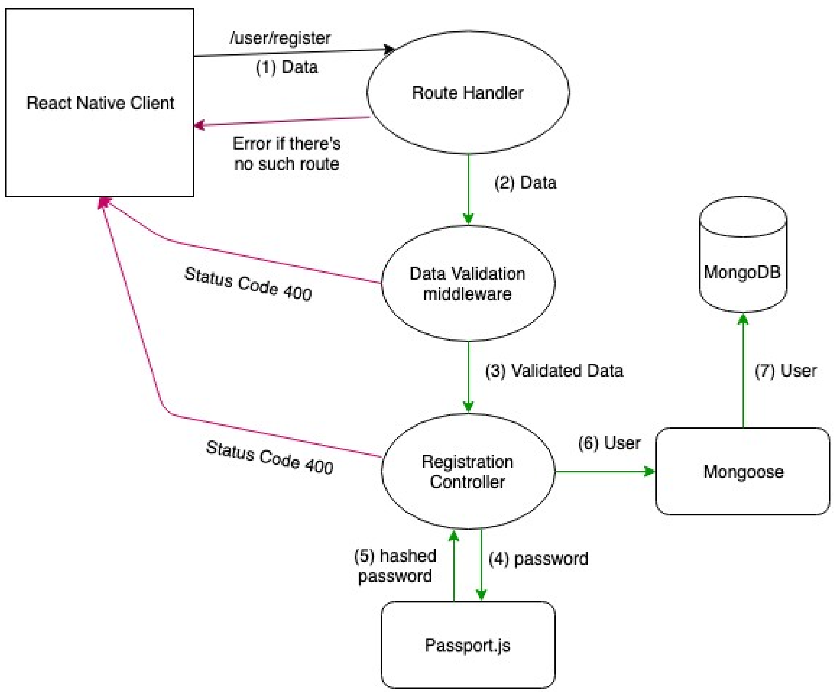


Figure 3. New User Sign Up Flow Diagram

7.2 User Schema Set Up

Although MongoDB allowing pushing data to the database without any schemas, it is a good thing to set up basic data validation before pushing it to the database.

7.2.1 Username Field

```
username: {
    type: String,
    trim: true,
    minlength: 3,
    unique: true,
    required: 'Username is required!'
  },
```

Figure 4. Username Field Schema

- **type: String** defines the type of the field. Basically, I expect a string for the username field.
- **trim: true** removes the spaces in the beginning and the end of the field as sometimes users register with something like that: "   myUsername   ", but to the database this field will be stored as "myUsername".
- **minlength: 3** specifies that it should be at least 3 characters in the username.
- **unique: true** will throw an error if somebody signing up with the username that already in use.
- **required: "Username is required!"** specifies the field as required and it's not possible not to specify it.

7.2.2 Email Field

```
email: {
    type: String,
    trim: true,
    minlength: 5,
    unique: true,
    required: 'Email is required!',
    lowercase: true,
    validate: [validator.isEmail, 'Invalid email address!']
  },
```

Figure 5. Username Field Schema

- **type**, **trim**, **minlength**, **unique**, **required** purposes are the same as in **username** field. See Username Field for more details.
- **lowercase: true** will unify the emails to avoid something like this "Ivan.TSUd@GmaIL.cOm" as it is absolutely not readable.
- **validate: [validator.isEmail, 'Invalid email address!']** checks whether provided email structure is not just some random text. It is done with 'validator' package. [29]

7.2.3 Password Field

The password field is not specified in the Schema as PassportJS [30] will take care of the password by itself. It will add hashed password to the database instead of the plain password. It increases the security of users' accounts. As in case if the database will leak, nobody will be able to use the accounts.

7.2.4 Avatar Field

```
avatarUri: {
    type: String,
    unique: true,
    required: 'Avatar is required!'
  },
```

Figure 6. Avatar Field Schema

The avatar is stored in a database as a URI link to the file hosting. The only validation here only for making sure that the “**avatarUri**” is **unique** and is a **String**.

Amazon AWS S3 [27] is used as a hosting for images.

The process user avatar saving and upload is the following:

1. Setup image upload permissions, so client app could request access to the file system.

    a) For “**ios**”, permissions are defined in “**Info.plist**” file.

    b) For “**android**”, permissions are defined in “**AndroidManifest.xml**” file.

2. Select the photo from photo gallery or take a new one using the camera on the Client App using “**react-native-image-picker**” module. [31]

3. Link “react-native-image-picker” module to the React Native app. The reason of linking the module is that this module has native dependencies for both “**ios**” and “**android**”. So, there is a need to link that with native code to have full access and being able to use setted permissions.

4. Store photo it in the app’s state.

5. Once photo has been successfully uploaded to the app’s state, form a request and using POST method send it to the server route “**/avatar-upload**”.

6. Using the **AWS S3 SDK** [32] for **Node.js**, configure endpoint and upload the photo to the file hosting. See Figure 6. Upload Avatar to the Amazon AWS S3 for the source code.

7. AWS sends back a link, it then it is back to the client app.

7.2.5 FirstName and LastName fields

```
firstName: {
    type: String,
    trim: true,
    minlength: 2,
    required: 'Please provide your First Name!'
  },
  lastName: {
    type: String,
    trim: true,
    minlength: 2,
    required: 'Please provide your Last Name!'
  },
```

Figure 7. FirstName and LastName fields Schema.

type, **trim**, **minlength**, **required** purposes are the same as in **username** field. See Username Field for more details.

7.2.6 Birth Date Field

```
birthDate: {
    type: String,
    required: 'Please provide the birth date!'
  },
```

Figure 8. Birth Date field Schema.

Birth Date is passed to the server as a formatted string that looks in the following way: “Sep 30, 1995”. The field is required as we need to be sure that the person is eligible to use the service.

7.2.7 Position Coordinates Fields

```
    latitude: {
      type: Number
    },
    longitude: {
      type: Number
    },
```

Figure 9. Geo-position fields Schema.

Once the user allowed to track geolocation, the "latitude" and "longitude" are going to be tracked and send to the server in the following form: "latitude": 37.785834, longitude: -122.406417.

As the client app was initialized with `*react-native-cli*` [33] , it is possible to track location using `*navigator.geolocation.getCurrentPosition()*`. See code for more details Figure 7. Get User Location

The Geolocation API exists as a global navigator object in React Native, just like on the web. It's accessible via navigator.geolocation in our source code and there is no need to import it.

The `*getCurrentPosition*` method from the Geolocation API is used. This method allows a mobile app to request a user's location and accepts three parameters: a success callback, an error callback, and a configuration object. [34]

7.2.8 Gender Field

```
    gender: {
      type: String
    },
```

Figure 10. Gender field Schema.

Gender field is not necessary, but for those who would like to set it, there will be an option to do it. The only checking here: checking for type of the field. It should be a String.

7.2.9 User Model Wrap-up

So, in general, there are 9 fields that are specified in the User Model Schema for the MongoDB. They describe the User and what properties user model may have.

7.3 User Sign In Flow

Sign In process is not a straightforward one. I have been searching for the strategies that will be well-suitable for both mobile and web so later there won't be a need to change Auth Flow for the Web client.

Among those cases, JSON Web Tokens is the strategy that is well-used to mobile development and web development, where the server is a RESTful API.

7.3.1 JSON Web Tokens

JSON Web Token is a token that is obtained by the client from the server and then saved in a device local storage and then JWT is placed to the header of every request sent to the server. See the Figure 11. JWT Signing Process. below for more details:

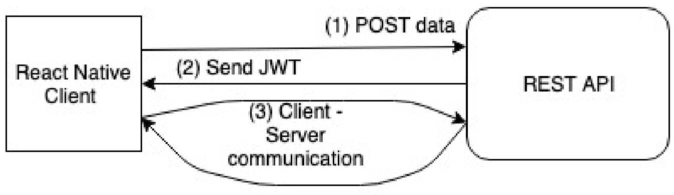


Figure 11. JWT Signing Process.

1. The user sends data as a `*POST*` request to the server.
2. The server authenticate the data received from the server, creates JWT and sends it to the client. The client saves it to the device local storage.
3. The server decodes the JWT for every request and verifies whether it is a valid token, and it can be used for identification and authentication purposes.

Pros of using JWT for Authentication:

- Stateless: While using JSON Web Tokens, a Server REST API never need to store user sessions or any other related information. Here, the token may contain all needed information for user authentication. The token itself may also contain information, like User Role and different permissions.
- Scalable: As far as JSON Web Tokens assume it to be stateless, all the systems pretending to use JWT could be scaled easily with respect to the state-dependent systems. As a result, I no longer need to think about the right user sessions management across multiple servers and databases – instead, the server can respond

to requests and authenticate users based only on the secret key used to sign the tokens.

- Unpredictable behavior of cookies: As far as traditional sessions rely on the browser to store a cookie, it may be a big problem to use it on mobile devices due to a variety of browsers and different permissions used on mobile platforms. In some cases, cookies even can't be sent to the server and request cannot be authorized.

7.3.2 Authentication Flow

The User Sign In flow is implemented in `userController.js` file. This file contains both `login` and `register` controllers. See Figure 8. Login Controller and Figure 5. Register Controller for code samples.

The `authController.js` file deals with all of the heavy lifting for signing in, making sure that the user logged in, JSON Web tokens are valid and the user has access to the account. See Figure 9. JWT Auth Controller for code samples.

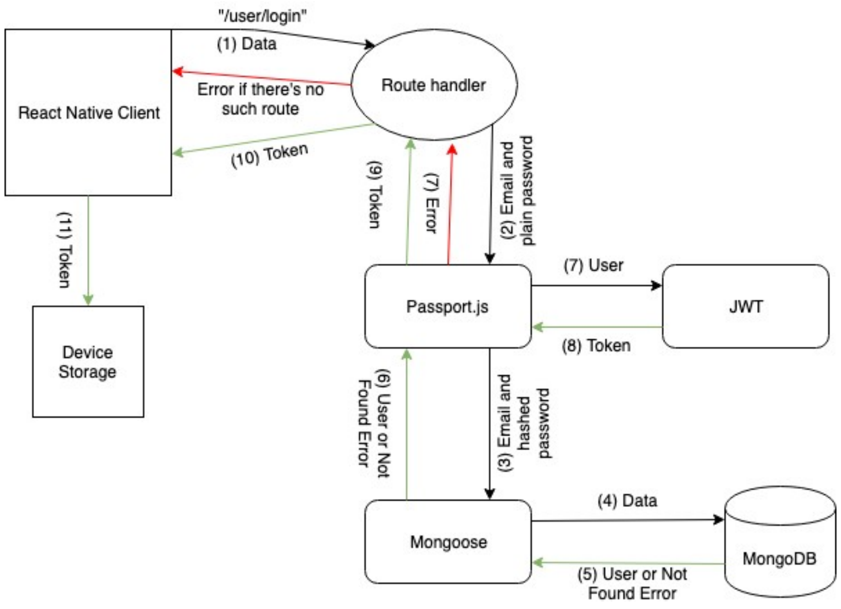


Figure 12. Authentication Flow Diagram.

According to the Figure 12. Authentication Flow Diagram., the following steps are applied for user to be signed in:

1. `POST` request is sent to the `/user/login` route. The request body has email and password in it. If server does not have such route, the error will be thrown.

2. Route handler will receive that data and pass data as a `*req.body*` to the PassportJS which will hash the password and send it to the Mongoose.
3. Mongoose receives the user credentials and makes an access to the MongoDB.
4. MongoDB returns back to the Mongoose a User object or error if the User instance is not found.
5. If PassportJS will receive a User instance, it will sign the JSON Web token with a secret code, stored in `.env` file.
6. PassportJS takes token and sends back to the Route handler which sends back it to the client.
7. Client takes the token and stores it in the device local storage (async storage).
8. From there, for every request will contain a header with token which will be checked on the server and if token check shows that the token is valid, user will be authorized (while user trying to access private routes), otherwise the user access will be rejected.

7.4 Password Reset Flow

We are all humans and all we tend to forget passwords we use for different services, especially if all the passwords are different.

The password reset flow is minimalistic here:

1. User is prompted to enter an email address he used to register an account in a special screen and press submit button.
2. Check email and find there a temporary password.
3. Log into his account with the temporary password and change it.

Now to the technical aspects:

On the client app there is a special navigation flow for the password reset. On the `Sign In` screen there is a `Forgot password?` button which on click redirects to the `ForgotPassword` screen where user can enter an email and press on `Reset

`Password` button. Pressing the mentioned button triggers the `_handlePasswordReset()` function which takes `email` as an incoming value, makes a `POST` request to the `/user/account/forgot-password` route.

`forgotPassword` controller accepts the request and do the following:

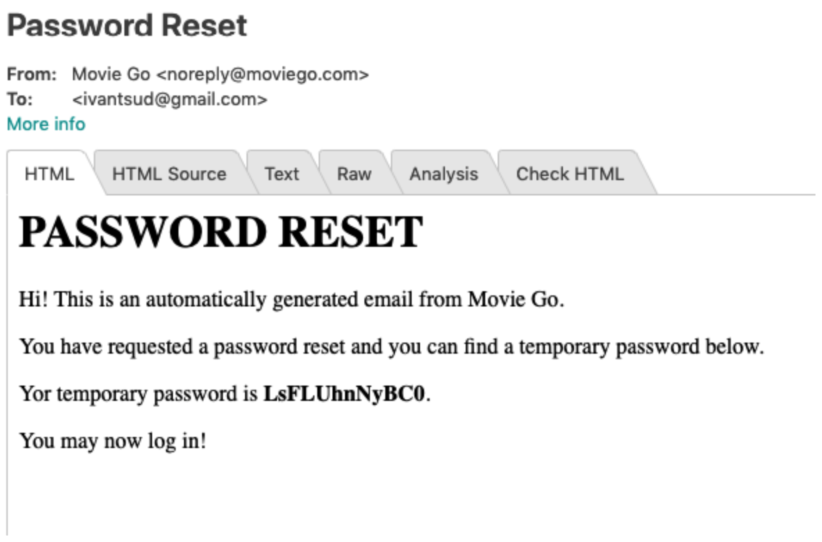
1. Check if there is a record in the database for a user with the provided email. If not, the error is thrown to the client, so it can notify the user that the email is not correct.

2. Use of password-generator [35] module to generate a secure password that will be sent to the user.

3. Update the password in the database and save the changes made on user model.

4. Send an email with the temporary password to the user. Nodemailer [36] was used as a middleware for sending email and Mailtrap [37] was used as a "*fake*" SMTP server.

And as a result, the Password Reset Emails looks like following:


Password Reset

From: Movie Go <noreply@moviego.com>
To: <ivantsud@gmail.com>
More info

PASSWORD RESET

Hi! This is an automatically generated email from Movie Go.

You have requested a password reset and you can find a temporary password below.

Yor temporary password is **LsFLUhnNyBC0**.

You may now log in!

Figure 13. Password Reset Email.

See Figure 10. Forgot Password Controller for the code reference.

8 Conclusions

As of this Master Thesis I have been doing back-end development for a mobile application. And up tho this point I have implemented the following:

1. Has been constantly deploying npm package `Leuchtturm` while Oleksandr has been developing it. [38]

2. Have configured the basic Node server with Express.

3. Have deployed the server to the Heroku. [39]

4. Have implemented user registration and sign in with the PassportJS.

5. Have implemented handling of the user authorization with the JSON Web Tokens and access flow for the private routes.

6. Have implemented password reset flow.

7. Have configured mail server for sending emails.

8. Partially done Facebook Authentication, meaning that the user entry can be created, but the user can't access the account yet.

During the development I have been using WesBos course on Node, StackOverflow, Spencer Carli guides and plenty of Medium articles that are mentioned in the references section.

Still, there's plenty of work left and still many improvements need to be done. There is a need to create a database of films and a microservice that will be pushing films to the database. Chat function for the registered users, and so on.

During this small period of time I’ve done as many as I could and happy about results. This project gave me new knowledge about databases, back-end stacks, servers, types of architecture. Within this project, I have increased my level as a developer and now I see things on a bigger scale, not only what end users see, but also the part that is hidden from them.

References

[1] Siftery [Electronic resource]

[2] Python. Official Python Documentation [Electronic resource]

[3] Ruby. Official Ruby FAQ [Electronic resource]

[4] "Why Should You Use Java for Your Backend Infrastructure?", Aran Davies [Electronic resource]

[5] "About JavaScript", Mozilla Documentation [Electronic resource] - Mar 23, 2019

[6] "Why We Use Django Framework & What Is Django Used For", Djangostars [Electronic resource]

[7] "Ruby on Rails: What it is and why you should use it for your web application", Bitzesty 01 [Electronic resource] - March 03, 2014

[8] "Overview of Spring Framework", Spring [Electronic resource]

[9] "Overview of the framework", ExpressJS [Electronic resource]

[10] "Overview of the framework", KoaJS [Electronic resource]

[11] "Choosing Back-end Tech in 2018", Andrés Cádiz Vidal [Electronic resource] - Aug 11, 2018

[12] "TIOBE Index for May 2019", TIOBE [Electronic resource] - May, 2019

[13] The Computer LanguageBenchmarks Game [Electronic resource]

[14] "How to Choose the Right Database System: RDBMS vs. NoSql vs. NewSQL", Alex Castrounis [Electronic reference]

[15] "Designing scalable backend infrastructures from scratch", Anshul Chauhan [Electronic resource] - Apr 30, 2017

[16] NPM. Official page [Electronic resource]

[17] DocsifyJS. Official Page [Electronic resource]

[18] StorybookJS. Official page [Electronic resource]

[19] Netlify. Official page [Electronic resource]

[20] React-native framework, Facebook [Electronic resource]

[21] NodeJS. Official documentation [Electronic resource]

[22] Mongodb. Official page [Electronic source]

[23] Mlab. Official documentation [Electronic resource]

[24] Heroku. Official page [Electronic resource]

[25] Docker. Official page [Electronic resource]

[26] Bitbucket. Official page [Electronic resource]

[27] Amazon S3 Overview [Electronic resource]

[28] passport-local-mongoose, NPM package, Christoph Walcher [Electronic resource]

[29] Validator, NPM package, Chris O'Hara [Electronic resource]

[30] PassportJS. Official documentation [Electronic reference]

[31] React-native-image-picker, NPM package, React Native Community [Electronic resource]

[32] Aws-sdk, NPM package, Amazon Web Services [Electronic resource]

[33] React Native documentation for react-native-cli [Electronic resource]

[34] "Using Geolocation in React Native", Aman Mittal [Electronic resource] - Sep 30, 2018

[35] Password-generator, NPM package, Bermi Ferrer [Electronic resource] - 2017

[36] Nodemailer. Official documentation [Electronic resource]

[37] Mailtrap. Official page [Electronic resource]

[38] Leuchtturm, NPM package, O. Shabala, I.Tsud [Electronic resource] - Apr 4, 2019

[39] "Moviego Server", Ivan Tsud [Electronic resource] - May 6, 2019

Appendix 1 – Code Samples

```
// Connect to Database and if there errors - throw to the terminal
mongoose
  .connect(process.env.DATABASE, { useNewUrlParser: true,
useCreateIndex: true })
  .then(() => console.log('Successfully connected to the
DATABASE.'))
  .catch(error => {
    console.error(`There was an error while connecting to the
DATABASE: ${error.message}`);
  });
```

Figure 1. Part of the source code of **./src/db/index.js**

```
// Create Express app
const app = express();
// Add properties to requests to use it as `req.body`
app.use(bodyParser.json());
app.use(cookieParser());
// Add validation methods for single `req`
app.use(expressValidator());
app.use(passport.initialize());
// Apply Passport Config
require('./passport/passportConfig');
// Handle routes
app.use('/', routes);
```

Figure 2. Part of the source code of **./src/app.js**

```
// Make sure that host uses Node 7.6+
const [major, minor] =
process.versions.node.split('.').map(parseFloat);
if (major < 7 || (major === 7 && minor <= 5)) {
  console.log(
    'To run the project. Node 7.6+ has to be installed. Go to
https://nodejs.org/ and download version 7.6 or greater.'
  );
  process.exit();
}
// Import environmental variables from .env file
require('dotenv').config({ path: '.env' });
// Connect to the DB
require('./src/db/index');
// Connect models
require('./src/models/User');
// Start Server
const app = require('./src/app');
// Set server PORT
app.set('port', process.env.PORT || 4444);
// Run Server
const server = app.listen(app.get('port'), () => {
  console.log(`Server started on http://localhost:$
{server.address().port}`);
});
```

Figure 3. Part of the source code of **./start.js**

```
exports.validateRegisterData = (req, res, next) => {
  // sanitization of username
  req.sanitizeBody('username').trim();
  req.checkBody('username', 'Username length should be at least 3
characters').isLength({ min: 3 });
  req.checkBody('username', 'Username should be
supplied!').notEmpty();
  // sanitization of email
  req.checkBody('email', 'The email is not valid!').isEmail();
  req.sanitizeBody('email').normalizeEmail({
    remove_dots: false,
    remove_extension: false,
    gmail_remove_subaddress: false
  });
  // sanitization of password
  req.checkBody('password', 'Password cannot be
empty!').notEmpty();
  req.checkBody('password', 'Password length should be at least 8
characters').isLength({ min: 8 });
  // sanitization of confirm password
  req.checkBody('confirmPassword', 'Confirm password cannot be
empty!').notEmpty();
  req.checkBody('confirmPassword', 'Passwords should
match!').equals(req.body.password);
  // sanitization of firstName and lastName
  req.checkBody('firstName', 'First Name field cannot be
empty!').notEmpty();
  req.checkBody('lastName', 'Last Name field cannot be
empty!').notEmpty();
  // if there are errors, send status of the error and error
message
  const errors = req.validationErrors();
  if (errors) {
    errors.map(error => res.status(400).send(error.msg));
    return;
  }
  next();
};
```

Figure 4. User Middleware to Validate data

```
  // Register user and hash the password
  User.register(newUser, password, (error, user) => {
    if (error) {
      return res.status(400).send(error);
    }
    if (!user) {
      return res.status(400).send();
    }
    return res.status(200).send(user);
  });
};
```

Figure 5. Register Controller

```
 const avatarUpload = multer({
  storage: multerS3({
    s3: new aws.S3(),
    bucket: 'moviego-avatar-storage',
    acl: 'public-read',
    ContentType: 'image/jpeg',
    metadata(req, file, cb) {
      cb(null, { fieldName: file.fieldname });
    },
    key(req, file, cb) {
      cb(null, `${file.originalname}_avatar.$
{file.mimetype.split('/')[1]}`);
    }
  })
});
```

Figure 6. Upload Avatar to the Amazon AWS S3

```
getLocation = () => {
    navigator.geolocation.getCurrentPosition(
      position => {
        this.setState({
          latitude: position.coords.latitude,
          longitude: position.coords.longitude
        });
      },
      error => console.log(error.message),
      { enableHighAccuracy: true, timeout: 20000, maximumAge:
1000 }
    );
  };
```

Figure 7. Get User Location

```
passport.authenticate('local', function(err, user) {
    if (err) {
      return res.status(400).json({ error: err });
    }
    if (user) {
      const token = jwt.sign({ id: user._id, email: user.email },
secret);
      return res
        .status(200)
        .header('x-auth', token)
        .send('Successfully authenticated!');
    }
    return res.status(401).json({ error: 'Invalid credentials.' });
  })(req, res);
```

Figure 8. Login Controller

```
passport.authenticate('jwt', function(err, user) {
    if (err) {
      // internal server error occurred
      return req.status(400).json({ error: err });
    }
    if (user) {
      // successful auth
      return res.status(200).send(user);
    }
    // no JWT or user found
    return res.status(401).json({ error: 'Invalid credentials.' });
  })(req, res);
```

Figure 9. JWT Auth Controller

```
exports.forgotPassword = async (req, res) => {
  // check if the user exists
  const user = await User.findOne({ email: req.body.email });
  if (!user) {
    return res.status(401).json({ error: 'Invalid credentials.' });
  }
  // generate new password
  const tempPassword = generatePassword(12, false, /[\w\d-]/);
  // update the password in the DB
  await user.setPassword(tempPassword);
  await user.save();
  // send email with temporary password
  mail.send({
    user,
    subject: 'Password Reset',
    tempPassword,
    filename: 'password-reset'
  });
  return res.status(200).send();
};
```

Figure 10. Forgot Password Controller