VILNIAUS KOLEGIJA / UNIVERSITY OF APPLIED SCIENCES
FACULTY OF ELECTRONICS AND INFORMATICS

AUTHORIZED BY
Vice Dean of Electronics and Informatics Faculty

_____dr. Loreta Savulionienė

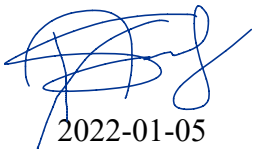
___-January-2022

IMPLEMENTATION OF USER LOYALTY SYSTEM INTO E-COMMERCE APPLICATION

FINAL PROJECT
FP 6531BX028 PI18E

UNDERGRADUATE  DŽIUGAS PEČIULEVIČIUS

2022-01-05

SUPERVISOR  AIRINA SAVICKAITĖ

2022-01-05

REVIEWER

2022-01-__

VILNIAUS KOLEGIJA / UNIVERSITY OF APPLIED SCIENCES
FACULTY OF ELECTRONICS AND INFORMATICS

VERIFIED BY

Vice Dean of Faculty of Electronics and Informatics

_____ dr. Loreta Savulionienė

October 4, 2021

FINAL PROJECT ASSIGNMENT

Given to undergraduate **Džiugas Pečiulevičius** of group **PI18E** on October 4th, 2021.

Final Project Title: Implementation of user loyalty system into e-commerce application.

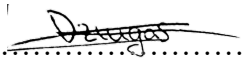
Final project description

The main idea of the project is to implement user loyalty system into an e-commerce web application where user would be able to collect, track and spend loyalty points.

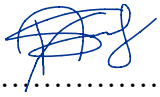
Project tools and components for project would be “React.js” for the client-side functionality of the project, as well as “Node.js” with “Express.js” for the backend where most data are handled and “MongoDB” for database where it is stored. Also, to check if the application works as intended and there are no breaking changes introduced, the application should be tested with “Jest” & “Enzyme”.

Project result would be functional e-commerce loyalty system. User should be able to collect points according to the amount spent on the platform. There should also be a separate page to track and spend their loyalty points.

Final project will be defended in the meeting of Software Development department on January 5, 2022.

Undergraduate…..……  ……Džiugas Pečiulevičius

(signature) **(name, surname)**

Supervisor……  ……Airina Savickaitė

(signature) **(name, surname)**

Verified by:

Head of Software Development Department .....…….....dr. Laura Gžegoževskė

(signature) **(name, surname)**

Advisers for Technical Affairs:

.....…….....Dainius Savulionis

(signature) **(name, surname)**

.....……..... Marius Gžegoževskis

(signature) **(name, surname)**

Vilniaus kolegija/University of Applied Sciences
Faculty of Electronics and Informatics
Department of Software Development

State Code: 6531BX028
Date: 05 January, 2022

Summary of the Final Project of Software Engineering Study Programme

Title of the Final Project: Implementation of user loyalty system into e-commerce application.
Application: e-Commerce application
Undergraduate: Džiugas Pečiulevičius
Supervisor: Airina Savickaitė

Volume of the work – 85 pages of text without annexes, 82 pictures, 13 tables, 0 annexes.

Summary

The main idea of this project was to have an e-commerce application that has been developed from scratch. Users should be able to view products, user cart features, make orders, leave reviews. And, to implement a user loyalty system into e-Commerce web application where users could collect points for their spendings, track how many points they have collected and spend these points in checkout process when paying for products. There is also a separate page where a user can view how many points a user has collected.

The application itself is built using MERN stack. For client side “React.js” has been picked, because it’s easy to scale, especially if there are plans to use React-Native in the future together. “Node.js” and “Express.js” work well together and make it easy to handle backend routing and backend code. And finally, “MongoDB” is a popular NoSQL database used with this stack.

The application is hosted on a free hosting platform called Heroku. It's straightforward to use and has great documentation. It also lets developers host it once and gives the ability to have the application updated as soon as main branch is updated in GitHub, and when the code is updated, Heroku will see it and build and host newest version of the application automatically.

TABLE OF CONTENTS

INDEX OF FIGURES

INDEX OF TABLES

INTRODUCTION

As lockdowns became the new normal, majority of businesses and consumers went online, providing and purchasing more goods and services online. The rise of sales in e-commerce is good news when it comes to revenue reports. During these past few years, the need of e-commerce solutions has skyrocketed. E-commerce is known as electronic or internet commerce. It refers to the buying and selling goods or services using the internet, and the transfer of money and data to execute these transactions. E-commerce is often used to refer to the sale of a physical products online, but these days it can also describe any kind of commercial introduction that is facilitated through the internet such as selling courses to people or selling digital products to customers. Whereas e-business refers to all aspects of operating an online business, ecommerce refers specifically to the transaction of goods and services. (Shopify. Ecommerce Definition. 2021)

Basically, e-commerce is all about shopping online and transferring money from one end to another. There are six types of e-commerce business models which we're going to focus on B2C the most.

- Business-to-Business (B2B) – is when business sells their products to other companies.
- Business-to-Consumer (B2C) – selling products to regular users.
- Consumer-to-Consumer (C2C) – basically when goods or services are being sold from one user to another.
- Consumer-to-Business (C2B) – consumer creates product value and business value in consumer-to-business relationships. This can be a photographer, influencer or a freelancer selling their products or services to a business.
- Business-to-Administration (B2A) – these transactions are conducted between businesses, government agencies, and public administrations.
- Consumer-to-Administration (C2A) – operates between individuals and government authorities via the internet. Consumers can contact their local governments and authorities to submit.

So why this topic has been chosen? Developers really like to challenge themselves and it pushed them to learn new technologies. E-commerce solutions seemed to be one of the most challenging which has a steep learning curve. And in a near future, there are plans to create application that would work similarly to an e-Commerce solution and the new knowledge gained will come in handy.

To create a fully working e-commerce there's lots of things to take into consideration. How will the front-end part look? How and where all the data will go and be handled? How will everything connect to the backend? How will the database look like? How will the payments be handled? And that's just a tip of the iceberg. MERN (MongoDB, Express.js, React.js, Node.js) stack solution looked perfect for it to learn a full stack.

The main project gaol and objectives are to have an e-Commerce application where user could view products, use cart features, make orders, leave reviews and to implement user loyalty system into an e-Commerce web application where users would be able to collect, track and spend loyalty points.

To reach these goals, the following steps need to be taken:

1. Planning and analysing requirements.
2. Create client-side functionality using JavaScript library – "React.js".
3. Create back-end functionality using "Node.js" together with "Express.js".
4. Integrate MongoDB database into the project where all data would be stored.
5. Host the application on Heroku cloud service.

1. PROFESSIONAL COMPETENCES

This table demonstrates a list of professional competences that have been acquired during the years of studying (Table 1).

Table 1 - List of professional competences

General competences		Results of the study program (Student will be able to)		Evidence of the competence
1.	Be able to make decisions. Follow the principle of equal opportunities and tolerance. Evaluate and ensure the quality of work. Adapt to innovation in business and work.	**1.1**	Be able to evaluate changes in the market and make decisions	Summary page provided.
		1.2	Be able to study independently. Improve knowledge. Raise qualification.	Pages 32 - 63
		1.3	Be able to use variety of software systems design and modelling methodologies.	Pages 32 - 63
		1.4	Be able to apply the acquired knowledge: in the software design, development, evaluation and implementation of application systems.	Pages 32 - 63
Subject competences		**Results of the study program (Student will be able to)**		**Evidence of the competence**
2.	Software development	**2.1**	Be able to design and implement algorithms with selected software tools.	Pages 32 - 63
		2.2	Be able to prepare user guide and technical documentation for software solutions.	Pages 64 - 82
		2.4	Be able to design, create new and improve existing software solutions.	Page 83
		2.5	Be able to design, create and improve user interface.	Page 83
3.	Development and management of Internet services (specialization - Internet technologies).	**3.1**	Be able to create and develop online service solutions.	MongoDB (pages 66 – 78) and Heroku (pages 79 – 82) solutions are used in the application

2. TASK FORMULATION

In the use-case diagram (Fig. 1), there are two user groups. One of them are customers and other are administrators. Customer user has limited functionality whereas administrator can do all the actions that basic users can and have access to do administrator-based tasks.

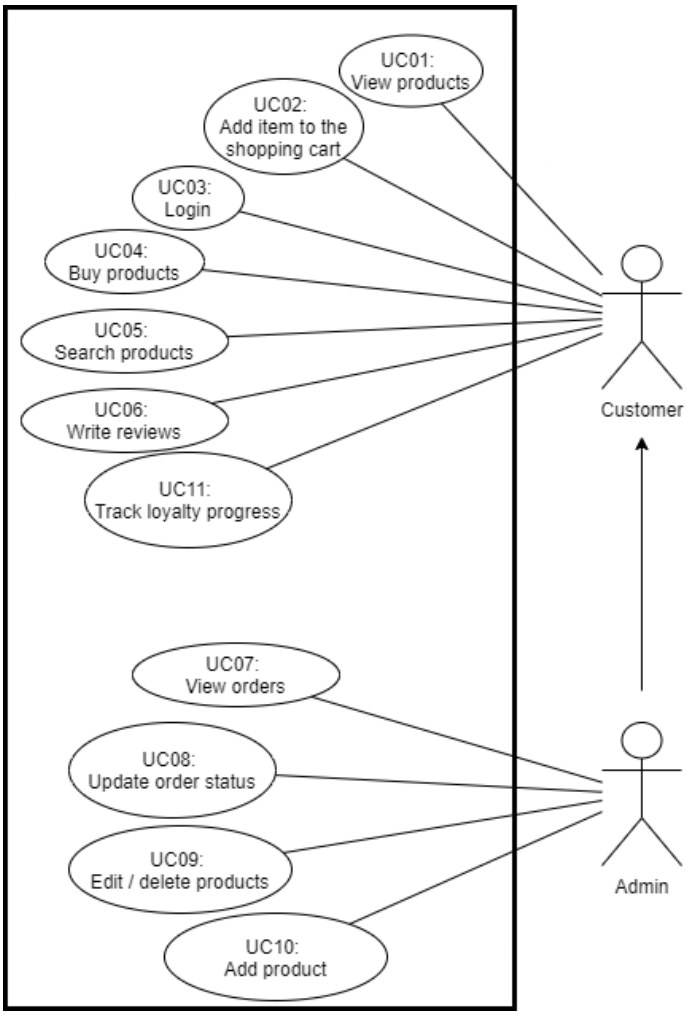


Fig. 1 - Use-case diagram for functional requirements

2.1 Functional requirements

In the functional and user requirement table (Table 2) we can see each requirement id, a what should it do and what kind of users are able to do that specific function. Each requirement also has a use case which it should be related to from the use-case diagram before, and the priority on which requirement had the biggest priority to be developed first.

To consider our application to be Minimum Viable Product (MVP), the application should meet these requirements:

Table 2 - Functional & User requirements

ID	FUNCTIONAL REQUIREMENT	USER REQUIREMENT	USE CASE	PRIORITY
FR1	The system should allow users to view products	Customer, Admin	UC1	1
FR2	The system should allow users to add item to shopping cart	Customer, Admin	UC2	2
FR3	The system should allow users to login into the system using sign-in screen	Customer, Admin	UC3	3
FR4	The system should allow users to check out and buy products	Customer, Admin	UC4	4
FR5	The system should allow users to search for desired products using the search field	Customer, Admin	UC5	9
FR6	The system should allow users to read and leave reviews on product pages	Customer, Admin	UC6	10
FR7	The system should allow admins to view orders and its data that were made by customers	Admin	UC7	6
FR8	The system should allow admins to update order statuses	Admin	UC8	5
FR9	The system should allow admins to edit and delete selected products	Admin	UC9	7
FR10	The system should allow admins to create new products and add them to the database	Admin	UC10	8
FR11	The system should allow users to track loyalty progress	Customer, Admin	UC11	11

2.2 Function analysis

As for the first functional requirement we have a user that can view existing products when user comes onto the website.

2.2.1 View products

When user comes into the website, user is greeted with the main page of the application where user can view and select products.

Table 3 - UC1 - View products

Function	View products
ID	UC 01
Description	Customers and Administrators can view products.
User roles	Customer, Administrator
Pre-conditions	There is an internet connection
Inputs	No inputs required after entering the website.
Outputs	Product list
Action	1. The user enters the website. 2. User should see all listed products.
Post-conditions	After entering the website, the user should be able ot see existing products.

In the activity diagram (Fig. 2), we have a user and system pools. The activity starts with user entering the website. When the user has entered the website, the system will start fetching the data from the backend. If anything goes wrong, it will display an error to the user, on the other hand, if the fetch was successful, the user should be able to see the data fetched from the backend in the user interface.

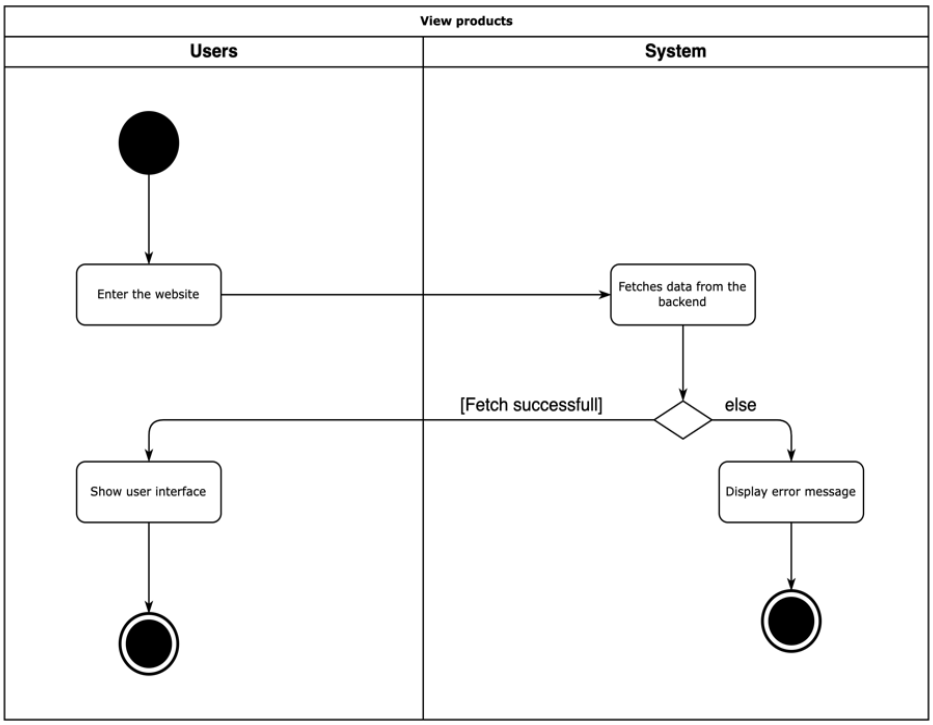


Fig. 2 - Activity Diagram - View products

2.2.2 Add item to shopping cart

In the first functional requirement, we can see a user trying to add a product into a shopping cart, at first user visits our application, and when a customer clicks on the desired product, user gets brought to the product page where user “Add to cart” button and once it’s clicked, the item is going to be added into the shopping cart. If there are no items in stock, then the button is disabled. Otherwise, if the item is added successfully, then user is going to be brought to the cart page. Both regular users and administrators can do this.

We can see the use case table (Table 4) that analyses the requirement:

Table 4 - UC2 - Add item to the shopping cart

Function	Add uten to the shopping cart
ID	UC 02
Description	Customers and Administrators can add products into the shopping cart.
User roles	Customer, Administrator
Pre-conditions	There is an internet connection
Inputs	Add a selected product into the cart by clicking “Add to cart” button.
Outputs	The product is put into the shopping cart.
Action	1. The user enters the website. 2. Users browses products. 3. The user choses a product and visits product page. 4. User clicks „Add to cart“
Post-conditions	The product appears in the shopping cart and user gets redirected into the shopping cart.

And in the activity diagram (Fig. 3) we can the flow for the use case that has been mentioned:

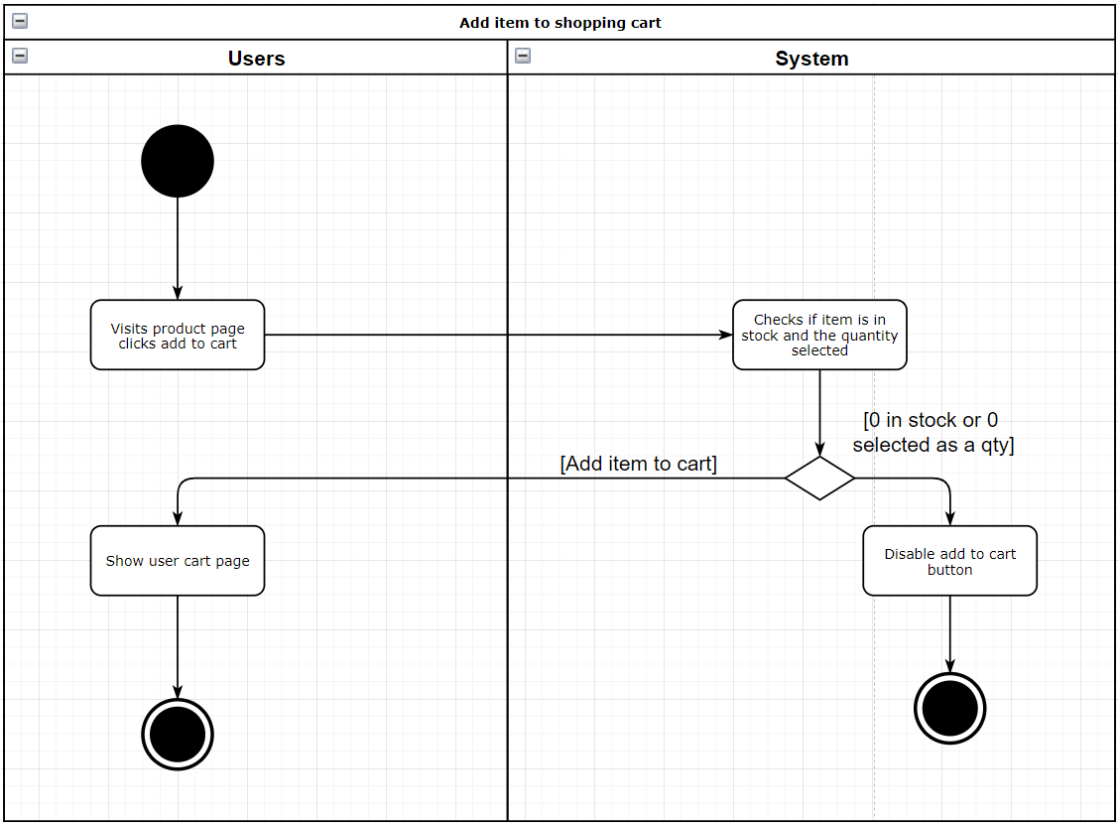


Fig. 3 - Activity Diagram - Add item to shopping cart

2.2.3 Login into the system

When a user is trying to login into the account, user navigates to the login page, enters user data and tries to send the request to the system. If login details are correct, the system responds with success request and logs the user in and navigates to the main page in the user interface. On the other hand, if the user has entered invalid data, the system will deny the request and return an error message that will be displayed on the screen for the user to see.

Below we can see the use case table (Table 5) for this requirement:

Table 5 - UC3 - Login into the system

Function	Login
ID	UC 03
Description	Customers and Administrators can login into the system.
User roles	Customer, Administrator
Pre-conditions	There is an internet connection
Inputs	Email and password must be entered into input fields.
Outputs	User successfully logs into the system.
Action	1. The user enters the website. 2. User clicks Login button. 3. The user enters their email and password. 4. If credentials are correct, user will get logged into the system.
Post-conditions	User is successfully logged in and can now do all the actions that any registered user can.

And below the use case requirement we can see the activity diagram (Fig. 4) of the login flow:

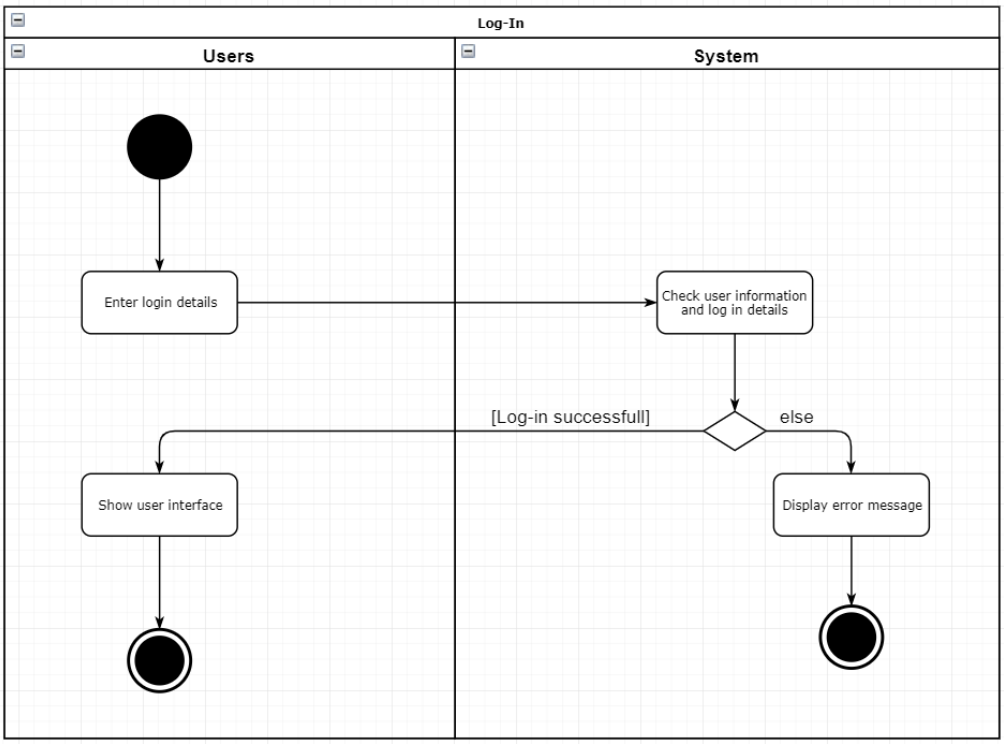


Fig. 4 - Activity Diagram – Login into the system

2.2.4 Buy products

If a user has made up their mind to buy a product, they add it to cart and then click checkout. First what happens is that the system checks if the user is signed in. If user is signed the system will display shipping page where user enters shipping details. If user is not signed in, the system will redirect user to the login page, where user enters login credentials and all the details entered are correct, then user gets redirected to the shipping page.

Once shipping information is entered, the system checks if all fields have been filled, if not it will display an error. If fields are filled, user will be shown a payment page where user is able to see a selection on what kind of payment user wants to proceed with and then system shows the payment method for the customer. Once user has entered details and clicked to proceed with the order, the application will place all data into the database and check if it has been paid.

In this case, the order will be already stored in the database, but it will show if it has been paid or not.

Table 6 - UC4 - Buy product

Function	Buy products
ID	UC 04
Description	Customers and Administrators can buy chosen products.
User roles	Registered customer, Administrator
Pre-conditions	There is an internet connection and user must be registered into the system
Inputs	User must provide shipping information and pay for the order.
Outputs	Once paid, user will see order window.
Action	1. The user checks out items in the shopping cart. 2. Enters shipping details in the checkout flow. 3. Clicks place order. 4. Order gets placed into the database and shown as not paid. 5. User pays for the product.
Post-conditions	Users order is stored into the database and administrators can process the order once they see that it has been paid.

In the activity diagram (Fig.5) the flow of user checkout process and buying a product is shown:

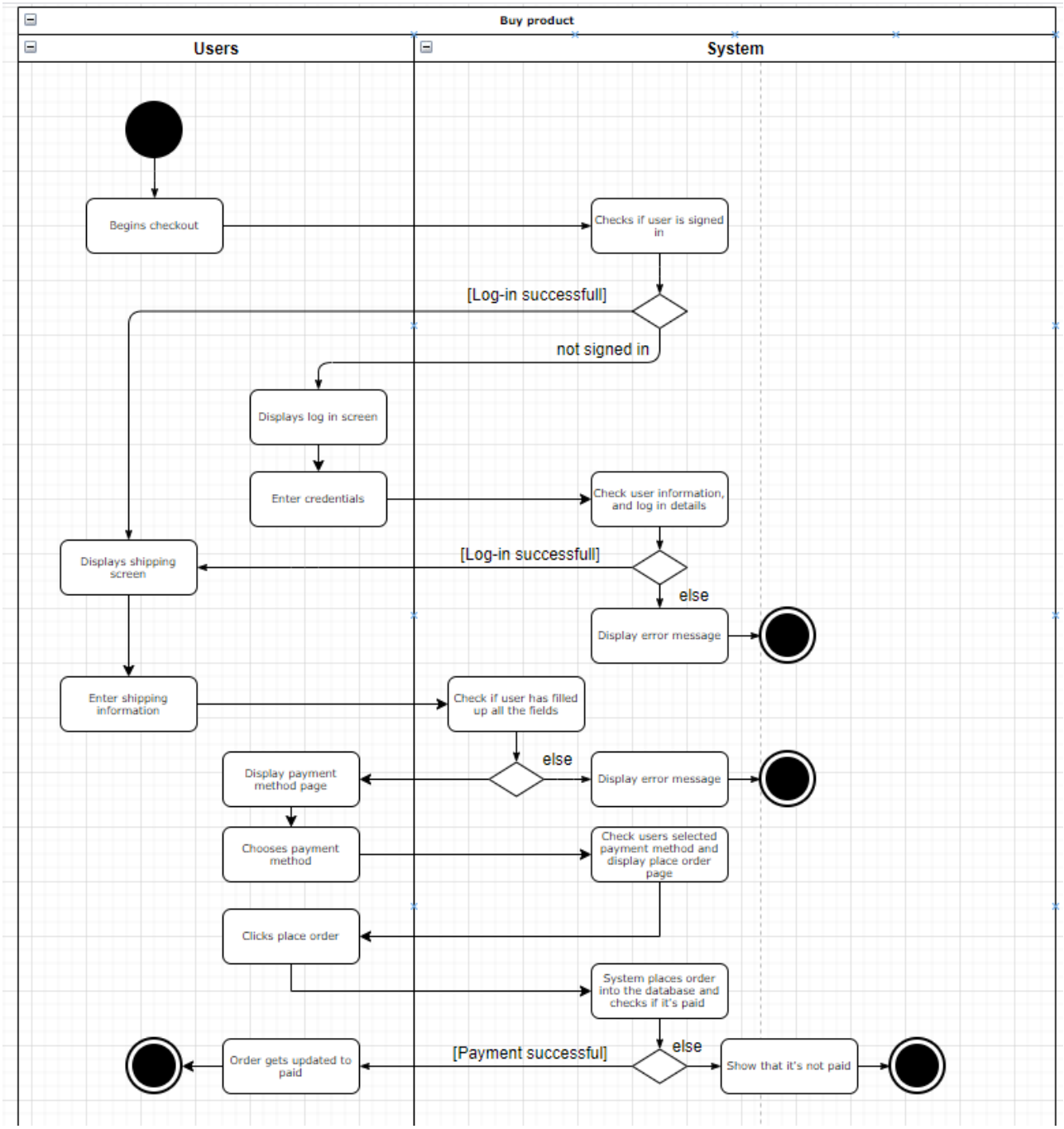


Fig. 5 - Activity Diagram - Buy product

2.2.5 Search products

A user should be able to search for the product. When a user enters the application, the products are fetched from the backend and a user is shown products available. In the navigation bar a user can find a search field and search for existing products. When a user is doing a search, the system filters out the product that user has searched for.

Table 7 - UC5 - Search products

Function	Search products
ID	UC 05
Description	Customers and Administrators can search products using the search field.
User roles	Customer, Administrator
Pre-conditions	There is an internet connection
Inputs	Product title
Outputs	System will filter and display items that match the serach field.
Action	1. User enters the website. 2. User enters the product info. 3. System displays the products
Post-conditions	The system will filter and show the items that user has searched for and user can select and

And in this example (Fig. 6), we can see the flow of a user entering the website and searching for a product where the system then filters out the products available.

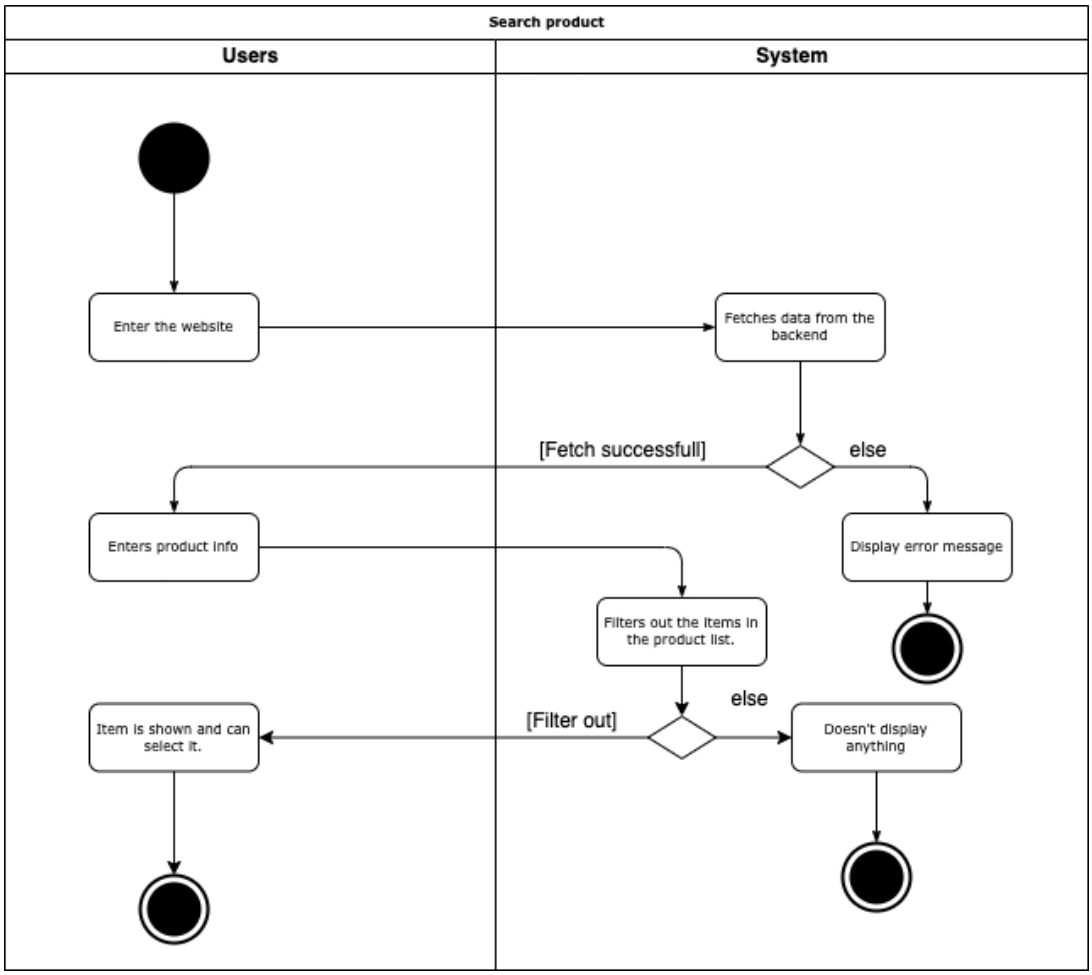


Fig. 6 - Activity Diagram - Search product

2.2.6 Write reviews

Users can leave reviews on the product pages below the product itself. Once a user visits the product page, he can scroll down and click reviews tab. If user is not signed in, the application will not show any form that user could leave the review and instead it will show a message that user needs to log in. If user is signed in, the application will show the form which user can fill out. Once user fills out the fields, the system checks if all fields have been filled. If not, it will display a message that not all fields have been filled and need to fill them out before continuing with the review. If all fields have been filled on the other hand, it will post a review on the product page.

Table 8 - UC6 - Write reviews

Function	Write reviews
ID	UC 06
Description	Customers and Administrators can leave reviews for the products.
User roles	Registered customer, Administrator
Pre-conditions	There is an internet connection and user should be logged in.
Inputs	A rating and a comment that user wants to leave.
Outputs	Message appears in the reviews section.
Action	1. The user enters the website. 2. Clicks on a chosen product. 3. Scrolls down below the product and clicks “reviews”. 4. Leaves message and rating. 5. The review appears in the product page.
Post-conditions	Review appears in the review section of the product page.

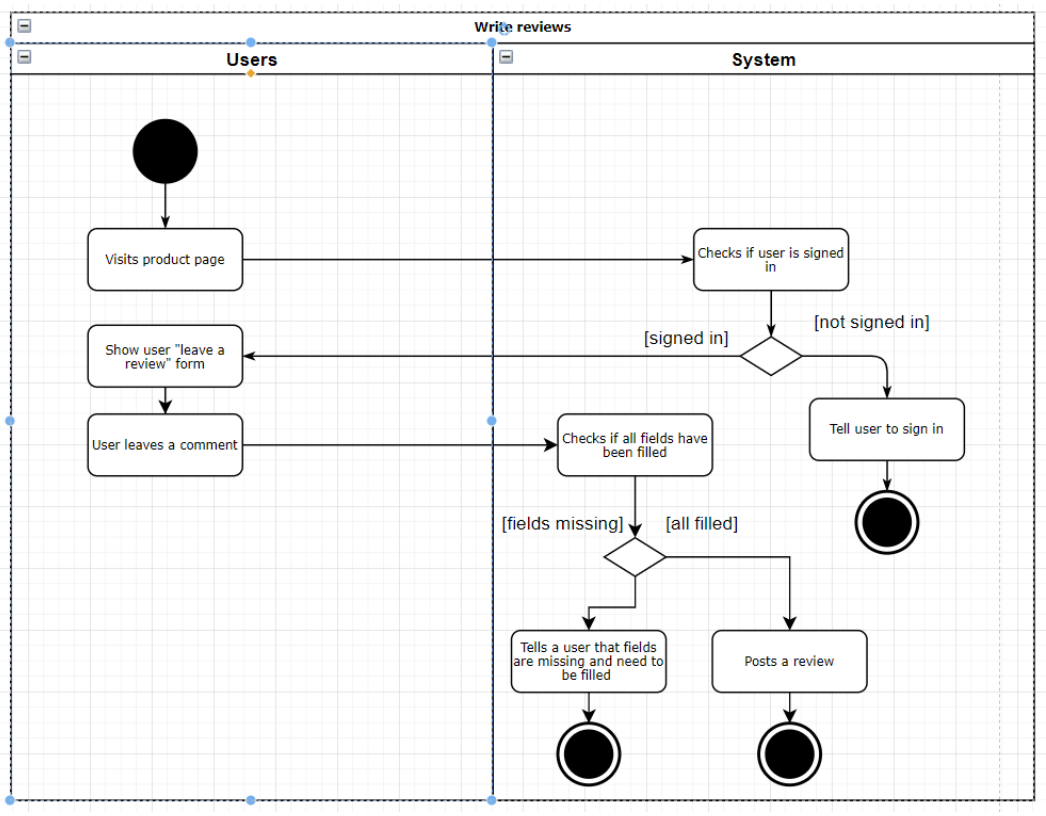


Fig. 7 - Activity Diagram - Write reviews

2.2.7 View orders

Administrators can view existing orders that customers have made. When a user is logged into the system and has administrator privileges, administrator dashboard is available. There a user can find orders as one of the selections available. Once clicked, a user can view all the orders that have been made within the application.

Table 9 - UC7 - View orders

Function	View orders
ID	UC 07
Description	Administrators can view existing orders made by customers.
User roles	Administrator
Pre-conditions	There is an internet connection and user must be an administrator.
Inputs	Visits admin dashboard and looks at orders panel.
Outputs	Message appears in the reviews section.
Action	1. The user enters the website. 2. System checks if the user is logged in and if admin. 3. User goes to the administrator dashboard. 4. In the dashboard clicks on “Orders”. 5. All orders appear on the screen.
Post-conditions	Orders are now shown to the administrator and admin can now view and confirm orders.

Below we can see an activity diagram (Fig. 8) of user viewing the orders in their profile screen.

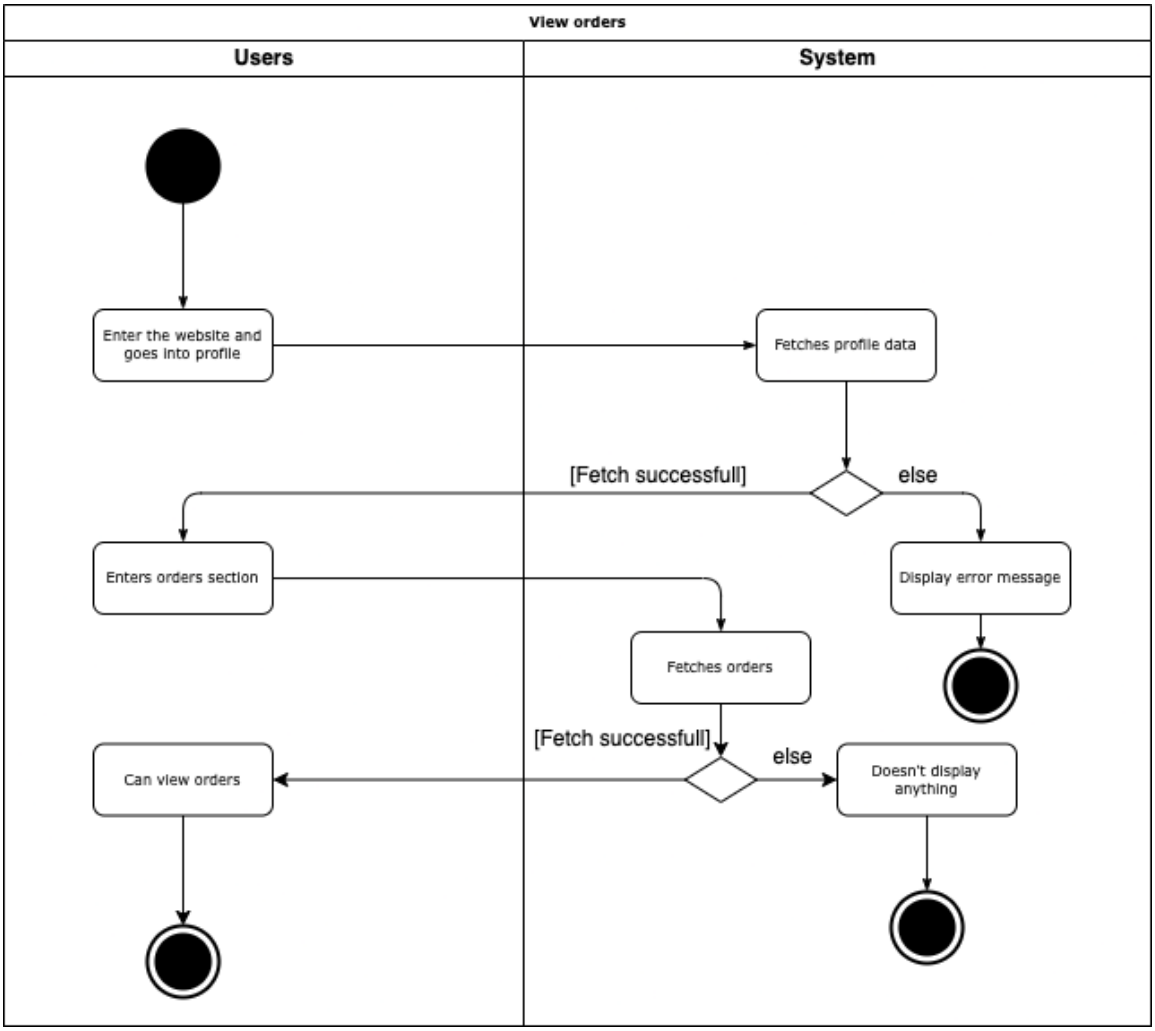


Fig. 8 - Activity Diagram - View orders

2.2.8 Update order status

In the application, administrators can change order statuses. Once an administrator logs into the system the system checks if the user has administrator privileges, if not, it will not show the administrator dashboard, on the other hand, the user will show a user dashboard option and can click on it. In the dashboard user can select orders and he would see all the orders that have been made. If there are no orders the system would not show anything. Once a user sees all the orders in the table, administrator can select the order he/she wants to view and change the status of, and once the system displays the order, the user can then change the status.

Table 10 - UC8 - Update order status

Function	Update order status
ID	UC 08
Description	Administrators can update order statuses.
User roles	Administrator
Pre-conditions	There is an internet connection and user must be an administrator.
Inputs	Visits admin dashboard, clicks on the order and changes the status to delivered or not.
Outputs	Displays order information and displays if the order status has been changed.
Action	1. The admin enters the website. 2. System checks if the user is logged in and if admin. 3. User goes to the administrator dashboard. 4. In the dashboard clicks on “Orders”. 5. All orders appear on the screen. 6. Clicks to view order details. 7. Clicks to mark the order as delivered.
Post-conditions	The application now displays the order as delivered.

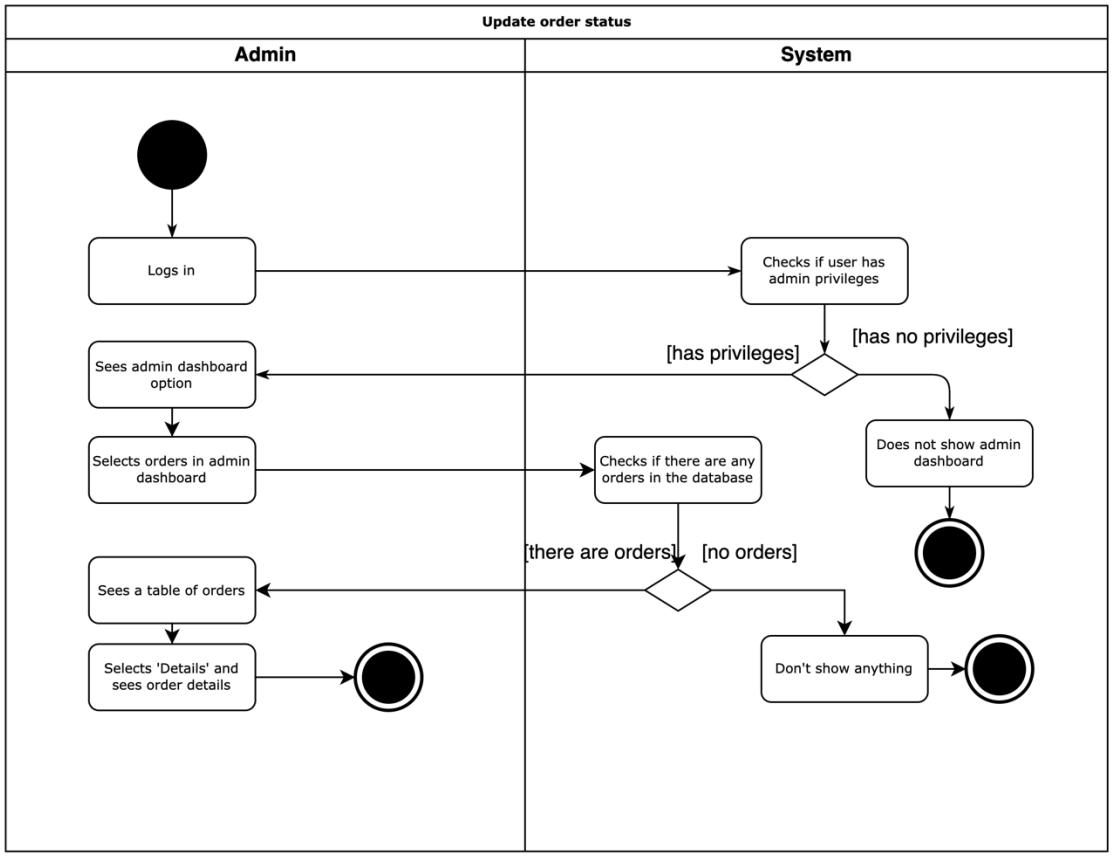


Fig. 9 - Activity Diagram - Update order status

2.2.9 Edit / delete products

Administrators can also delete and edit existing products. Once again, user logs in and system checks for admin privileges. If user doesn't have any, then they will not see an administrator dashboard menu option. But when administrator goes to the dashboard and selects products tab, they will see all the existing products. Then user can select the product and edit the data of it.

Table 11 - UC9 - Edit / delete product listings

Function	Edit / delete products
ID	UC 09
Description	Administrators can edit or delete products.
User roles	Administrator
Pre-conditions	There is an internet connection and user must be an administrator.
Inputs	Clicks on edit or delete button in product list page in the administrator panel.
Outputs	Shows the edit screen if the edit option was selected or removes the product completely.
Action	1. The admin enters the website. 2. System checks if the user is logged in and if admin. 3. User goes to the administrator dashboard. 4. In the dashboard clicks on “Products”. 5. All products appear on the screen. 6. Clicks to edit or delete the product.
Post-conditions	The item data is changed or deleted depending on the action.

And below, we can see the flow represented in the activity diagram (Fig. 10), how a user tries to edit or delete products and how the system responds.

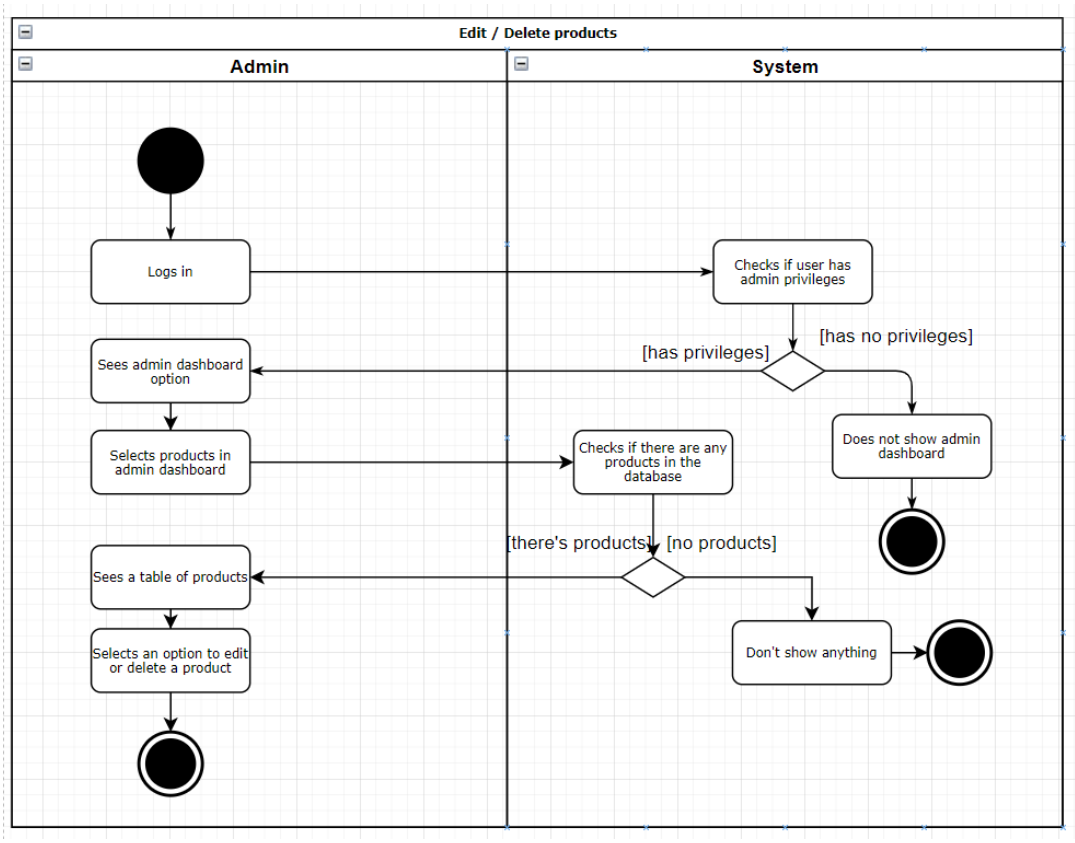


Fig. 10 - Activity Diagram - Edit / delete products

2.2.10 Add product

Once again, when administrator gets logged in and heads to the administrator panel and selects the products dashboard, user then can click on “Add a product” and the system will create a sample product that is put into the database, and then redirects user to the edit page of that product. And once administrator saves the item, user will be brought back into the administrator dashboard.

Table 12 - UC10 - Add product

Function	Add product
ID	UC 10
Description	Administrators canadd product to the system.
User roles	Administrator
Pre-conditions	There is an internet connection and user must be an administrator.
Inputs	Admin needs to enter product information like product name, description, quantity and price.
Outputs	Displays the product in the product list.
Action	1. The admin enters the website. 2. System checks if the user is logged in and if admin. 3. User goes to the administrator dashboard. 4. In the dashboard clicks on “Products”. 5. All orders appear on the screen. 6. Clicks on “Add product”. 7. Enters product details.
Post-conditions	The item is now added into the database and displayed in the product list.

In this activity diagram below (Fig. 11) we can see a user adding a product into the system.

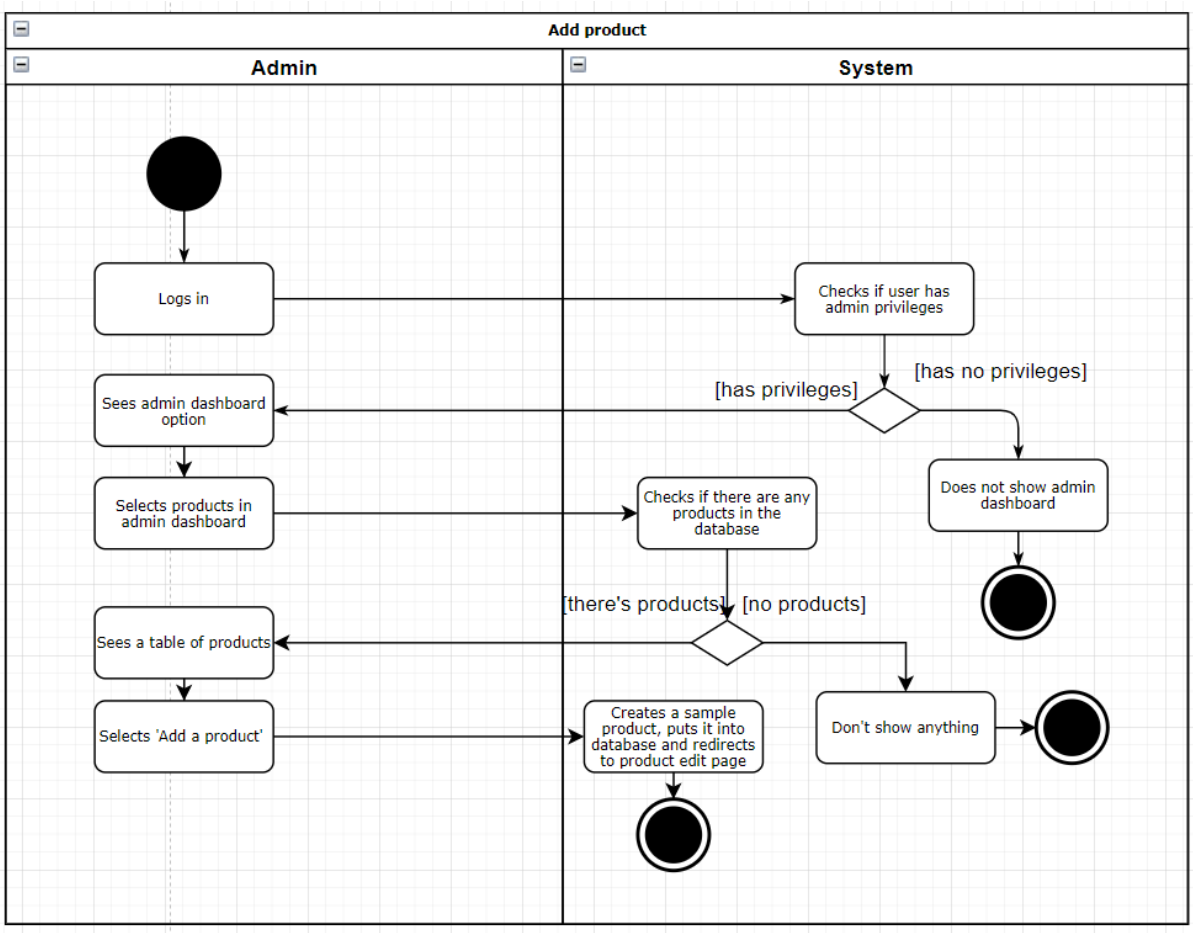


Fig. 11 - Activity Diagram - Add product

2.2.11 Track loyalty progress

Logged in users can track loyalty progress where they can see how many points they have collected. When they click on their name in the header and then click profile button, they will see a whole profile dashboard, after that will be able to see a tab named “Loyalty”. When they click on that tab, users will be able to see all the points they have collected that can be spent.

Table 13 - UC11 - Track loyalty progress

Function	Track loyalty progress
ID	UC 11
Description	Customer / Administrators can track loyalty points collected.
User roles	Customer / Administrator
Pre-conditions	There is an internet connection.
Inputs	Visits profile and select “Loyalty”
Outputs	User should see loyalty page come up.
Action	1. The admin enters the website. 2. System checks if the user is logged in. 3. User goes to the profile. 4. User clicks on loyalty tab. 5. All loyalty data is displayed on the screen.
Post-conditions	The user is able to see and track points.

In the activity diagram which is shown below (Fig. 12), a progress of tracking loyalty points is shown. A user logs into their account and goes into the profile page. When they have reached the profile page, then they enter loyalty tab and the system then shows the loyalty progress of a user.

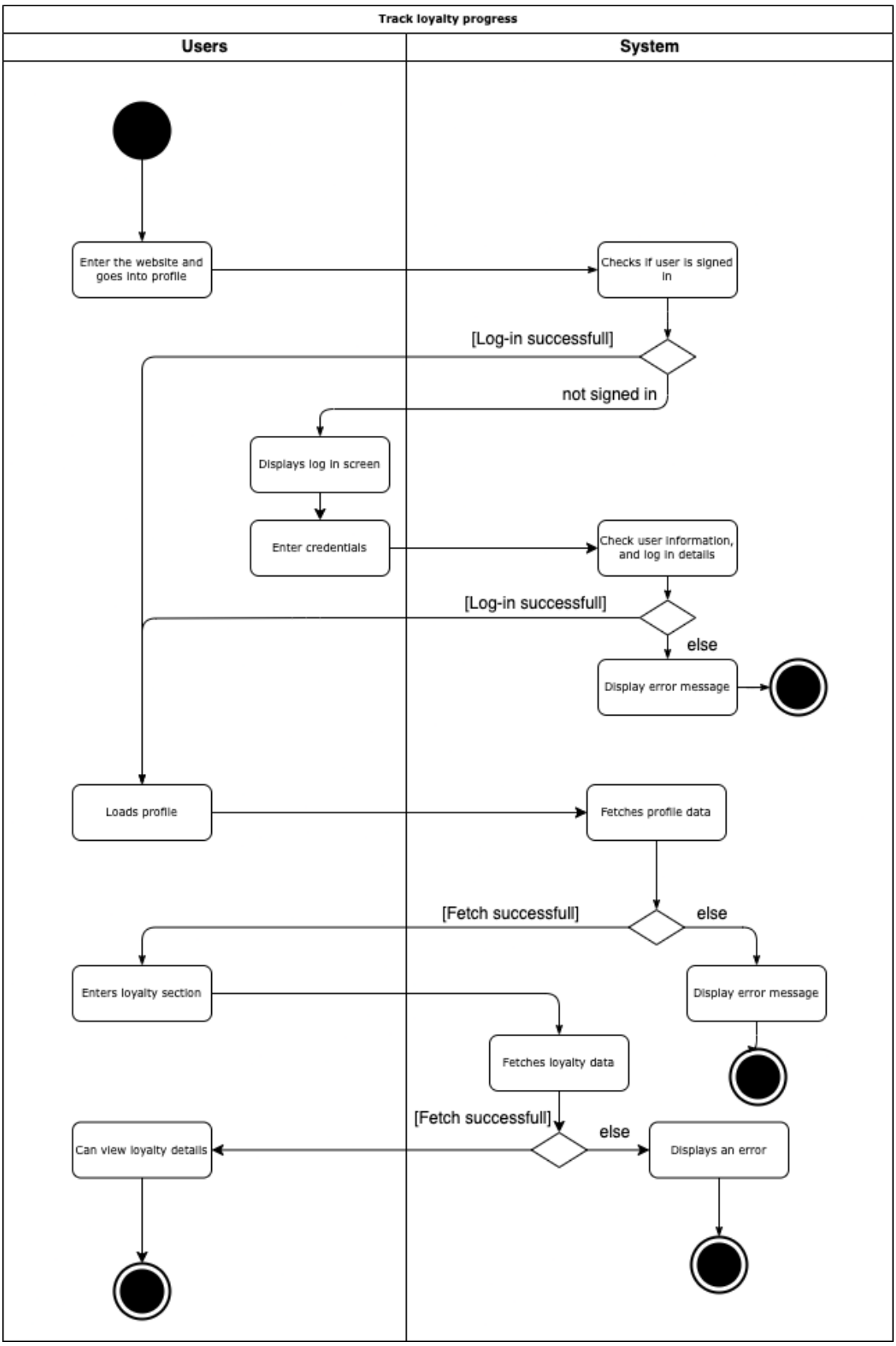


Fig. 12 - Activity Diagram - Track loyalty progress

2.3 Non-functional requirements

As we have covered functional requirements, we also need to cover non-functional requirements:

1. Application should be written in JavaScript.
2. Application should use React.js framework for frontend code.
3. The application should use Express.js together with Node.js to handle backend data.
4. The application data should be stored on MongoDB database.
5. The system should allow users to pay for their goods using Stripe or PayPal.

3. TASK ANALYSIS

3.1 Entity Relationship diagram

The MongoDB database that is used within this project is a popular object-oriented, dynamic and scalable NoSQL database. MongoDB is basically a document-oriented database. It uses JSON-like documents with optional schemas. The objects of data are stored as documents inside a collection. Unlike a traditional relational database of storing the data into columns and rows.

This is how MongoDB NoSQL database Entity Relationship diagram looks like.

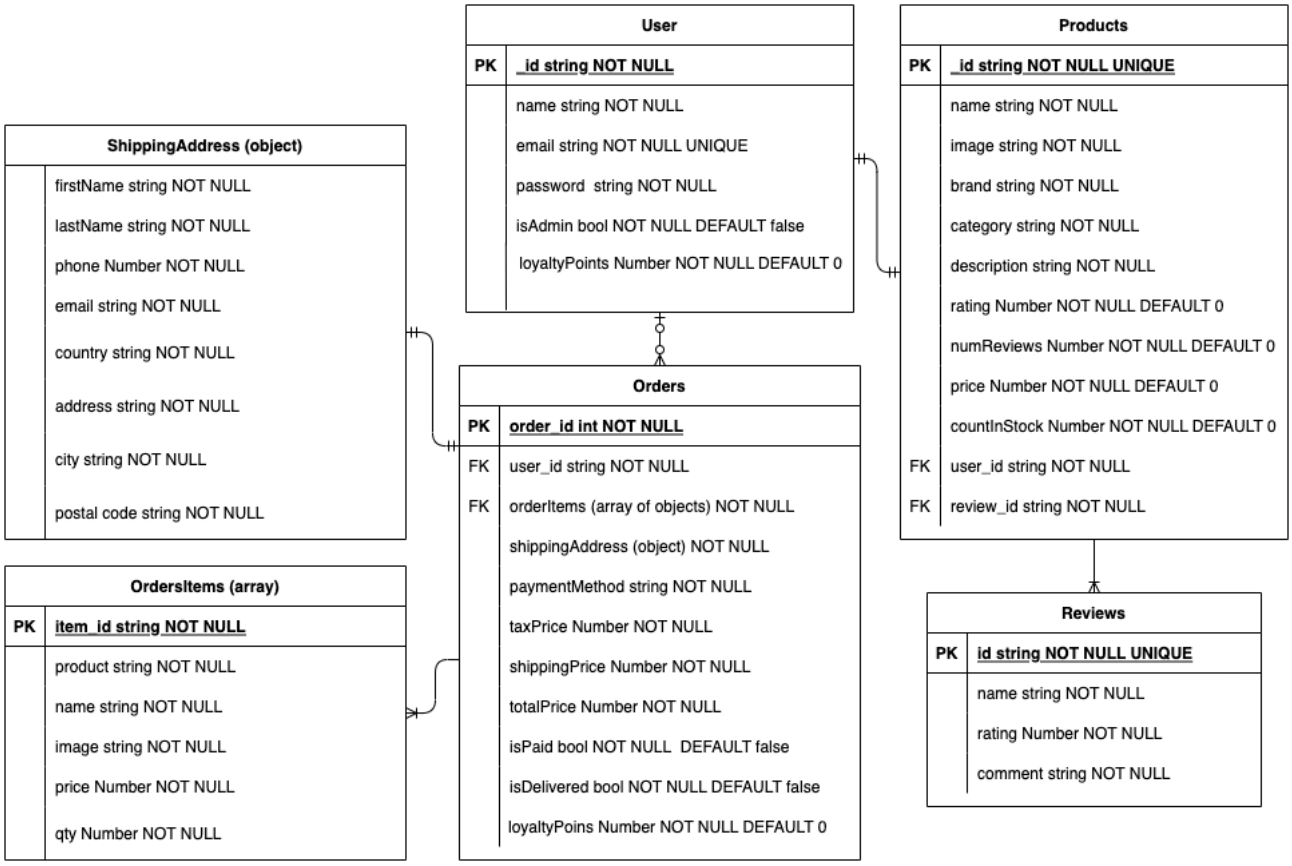


Fig. 13 - Entity Relationship diagram

All products in the application are stored as documents as JSON format in the database. JSON (JavaScript Object Notation) is a simple human readable data format. Basically, JavaScript objects are simple containers where they have a key which is an ID in this case and other values related to that object.

And this is an example on how one of the order documents looks with some test data. We can see that order has an id, prices, statuses, how many loyalty points a user is awarded and a user that it

belongs to, how it was paid, and other related data like ordered items list and user shipping address data.

Fig. 14 - Order schema example

The products have data such the rating, number of reviews, price, stock count, product descriptions. Also, all the reviews that are made by user, they are stored as objects into reviews array.

Fig. 15 - Product schema example

And the last one probable is the simplest one, are users. User model has a property that separates regular users from administrators. All users have a field that tracks how many points a user has collected, name, email and a password that is hashed and salted.

Fig. 16 - User schema example

3.2 Class diagram

In the class diagram we can see that we have two main actors for the application that are going to be using the application. One of them are regular users like customers, and other actors are administrators – people that are managing products, orders and all other sensitive data.

Unlike regular users, admins have some extra functionality that they can do in the application, like they can manage users that are in the system. They can remove, update and make other users administrators. Also, admins can change order statuses, update and delete orders. Unlike other users, administrators can also add, delete and edit products.

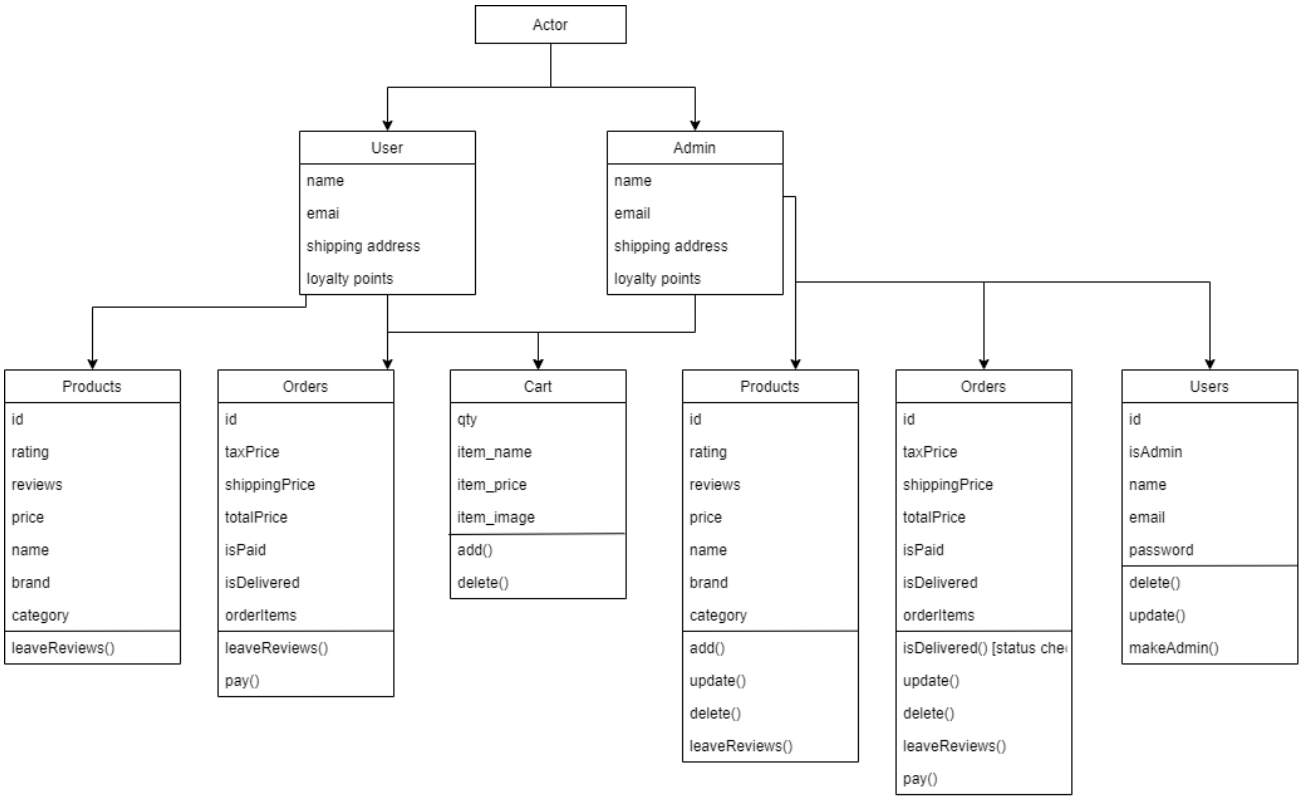


Fig. 17 - Application class diagram

3.3 Use case and requirement traceability table

As we can see the table below and as we talked previously, unlike administrator, a customer user has limited functionality. Administrator can do everything that a regular user can and has some extra functionality to view and manipulate sensitive data. Regular users can look around search and view products, add them to shopping cart, buy them, leave reviews and track their loyalty progress within the profile page. Administrators on the other hand, can do all that including view all the existing orders, update orders statuses, change product data and delete them and create new products.

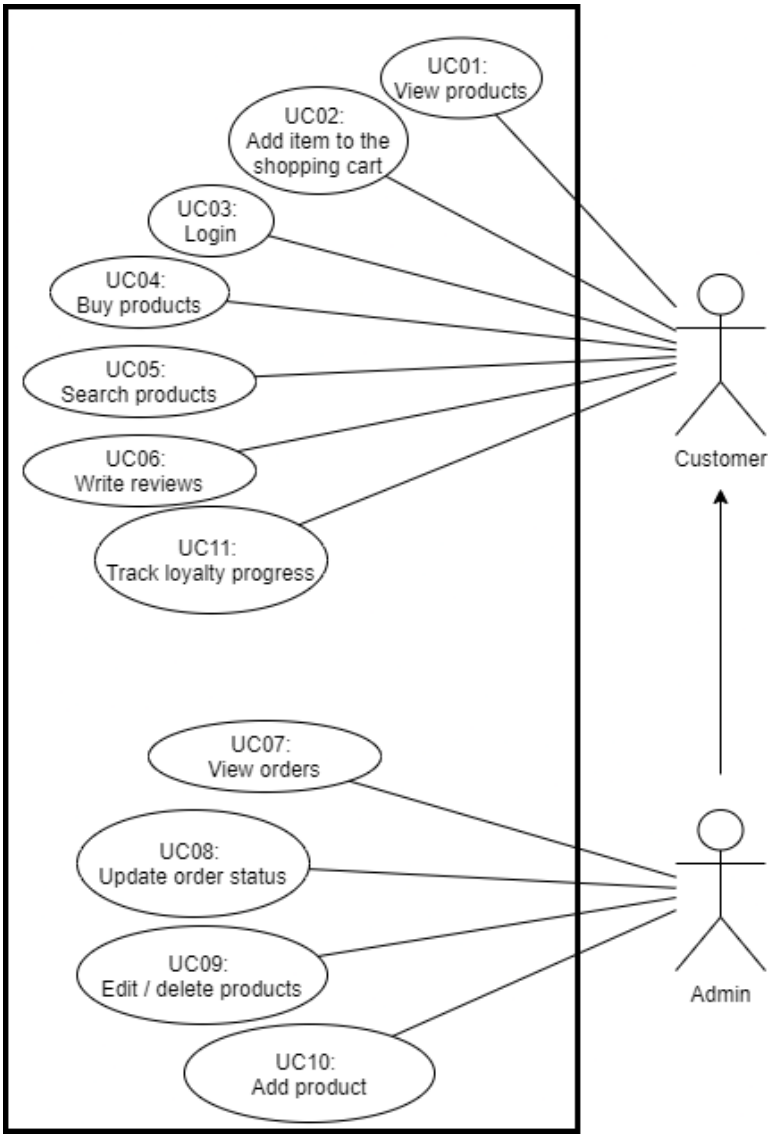


Fig. 18 - Use case diagram

Just at the beginning, we have specified our functional requirements on what user should be able to do within the application. And in the use-case diagram we can see our users and what use-cases they are able to do. In the requirements traceability table, we can see that each of the functional requirement matched our use-case, and this table shows exactly that.

		USE CASES										
		UC1	UC2	UC3	UC4	UC5	UC6	UC7	UC8	UC9	UC10	UC11
REQUIREMENTS	R1	X										
	R2		X									
	R3			X								
	R4				X							
	R5					X						
	R6						X					
	R7							X				
	R8								X			
	R9									X		
	R10										X	
	R11											X

Fig. 19 - Requirement traceability table

4. SOFTWARE IMPLEMENTATION

Fig. 20 - MERN stack

MERN stack is a popular stack for creating Single Page Applications (SPAs). MongoDB in this stack is a NoSQL database that keeps data in collections and documents. Express is a Node.js web application framework that helps the Node application development easier and a whole lot faster. React in this stack is a frontend framework. It helps developers to build SPAs which allows users to browse through the application without needing to reload the application. It changes only relevant parts of the page with new pieces of information. And Node.js itself is one of the most widely used server-side frameworks for running JavaScript code directly onto the web server itself.

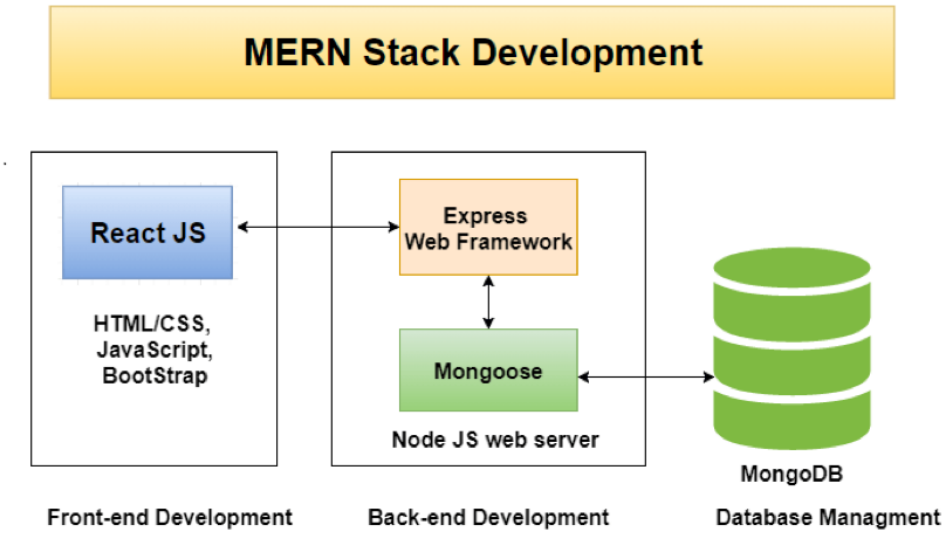


Fig. 21 - MERN stack diagram

4.1 Code structure

The is practically divided into two parts. One of them is for User-Interface (front-end) part of the project and the other is back-end. Front-end part is a react project with pages and reusable components. The other part is back-end. It contains various files, such as database configuration, models, middleware files and route handling.

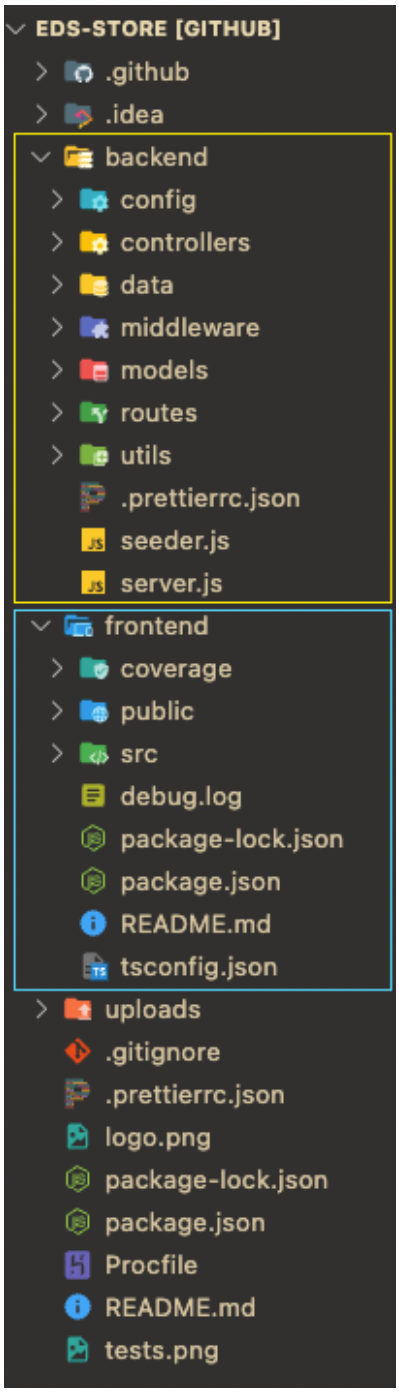


Fig. 22 - Code structure

4.2 Backend

If we dive deeper into the backend folder, we can see lots more. Backend consists of controllers, dummy data, middleware's, models, routes, configuration and server files. We're going to go more in-depth in upcoming sections.

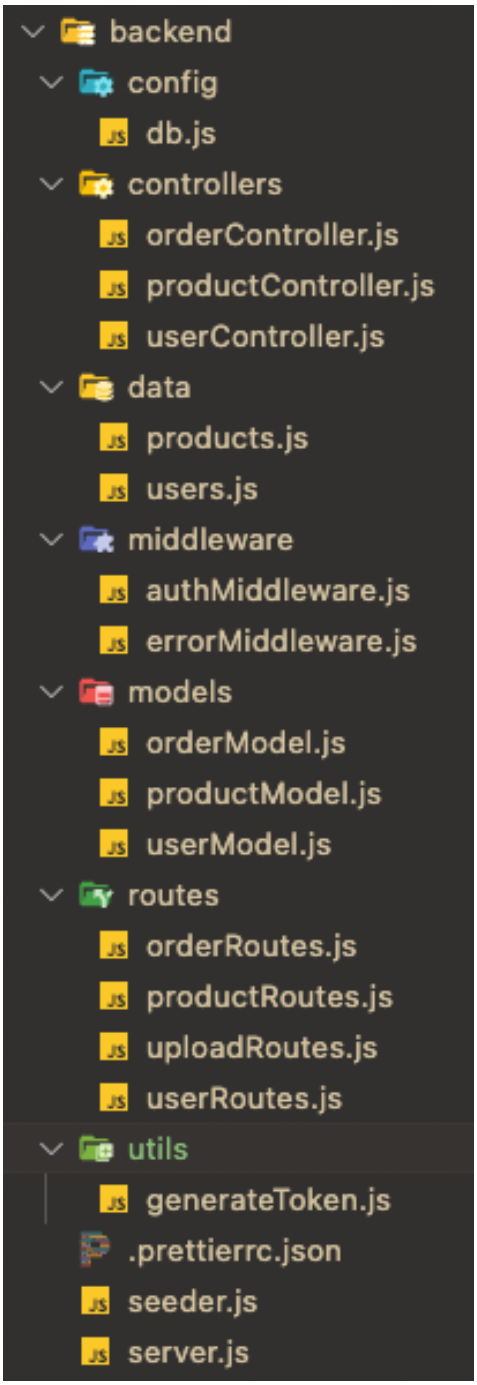


Fig. 23 - Backend code structure

4.2.1 Backend – Controllers

Backend controllers are responsible for handling request. In this example (Fig. 24), we can see a user controller handling registering new user. When we try to register a new user, it will first check if the user exits with the entered email and if user exists, the system will return a code of 400 and throw an error we specified. If the user doesn't exist, then a user is created. And finally, we add a check if a user exists, we respond HTTP response code 201 – Created and pass user data. On the other hand, if the user isn't found, then we respond with status code 400 – Bad Request and return an error that data is invalid.

Fig. 24 - Backend user controller example

4.2.2 Backend – Models

In backend model files we define how our models will look like when we connect our application to the database. We define a schema, in this example a user schema and a user will have

a name, emails, password, a field that shows if a user is an admin, loyalty points field and a field to show when a user was created. All the fields have a type, Boolean value if the field is required and a unique Boolean value if the field must be unique.

/backend/models/userModel.js

```
1  import mongoose from "mongoose";
2
3  const userSchema = mongoose.Schema(
4    {
5      name: {
6        type: String,
7        required: true,
8      },
9      email: {
10       type: String,
11       required: true,
12       unique: true,
13     },
14     password: {
15       type: String,
16       required: true,
17     },
18     isAdmin: {
19       type: Boolean,
20       required: true,
21       default: false,
22     },
23     loyaltyPoints: {
24       type: Number,
25       required: true,
26       default: 0,
27     },
28   },
29   {
30     timestamps: true,
31   }
32 );
33
```

Fig. 25 - Backend user model example

A user model also has methods to check if password match in case we want to update user passwords and before saving a password into the database we want to add salt to user passwords, so they are hashed and salted before saving.

/backend/models/userModel.js

```
import mongoose from "mongoose";
import bcrypt from "bcryptjs";

const userSchema = mongoose.Schema(
...
);

userSchema.methods.matchPassword = async function (enteredPassword) {
  return await bcrypt.compare(enteredPassword, this.password);
};

userSchema.pre("save", async function (next) {
  if (!this.isModified("password")) {
    next();
  }

  const salt = await bcrypt.genSalt(10);
  this.password = await bcrypt.hash(this.password, salt);
});

const User = mongoose.model("User", userSchema);

export default User;

```

Fig. 26 - Backend user model methods example

4.2.3 Backend – Config

Backend configuration folder has a database configuration file. We can add more methods if we wish to include more databases. For example, if we had one database for testing and another for production, we could add another method which would use a production database API key and just add a flag which would check if the system were in development environment or production and would use required method accordingly.

/backend/config/db.js

```
import mongoose from "mongoose";

const connectDB = async () => {
  try {
    const conn = await mongoose.connect(process.env.MONGO_URI, {
      useUnifiedTopology: true,
      useNewUrlParser: true,
      useCreateIndex: true,
    });
    console.log(`MongoDB connected: ${conn.connection.host}`.cyan.underline);
  } catch (error) {
    console.error(`Error: ${error.message}`.red.underline.bold);
    process.exit(1);
  }
};

export { connectDB };

```

Fig. 27 - Backend database connection

4.2.4 Back end – Routes

In routes files we can specify how to routes can be reaches and we can also add middleware's what kind of users can reach those files which I'm going to talk about in middleware section. Route can have a protect and admin middleware. If protect is used, that means that exactly that route can only be reached by registered users. And if admin is used, that route can only be reached by application administrator users.

/backend/routes/orderRoutes.js

```
1 import express from "express";
2 const router = express.Router();
3 import { ... } from "../controllers/orderController.js";
4 import { protect, admin } from "../middleware/authMiddleware.js";
5
6 router.route("/").post(protect, addOrderItems).get(protect, admin, getOrders);
7 router.route("/myorders").get(protect, getMyOrders);
8 router.route("/:id/pay").put(protect, updateOrderToPaid);
9 router.route("/:id/deliver").put(protect, admin, updateOrderToDelivered);
10 router.route("/:id").get(protect, getOrderById);
11
12 export default router;
13
```

Fig. 28 - Backend order routes example

4.2.5 Backend – Middleware's

As it has been talked previously, we can add middleware's to limit user access to specific routes. If we use protect, the system will limit users that are not authorized and don't have an active JWT token (a package used that helped regarding handling authentication [6]).

Authentication middleware will check if user has administrator privileges and will limit access to users who don't have such access. This method has a name of "protect". It first checks if user is authorized and has an authentication token (lines 8-11). The program then will try and get the token and verify it with the JWT secret that is used to generate the token. If a user is authorized, we will get all user information except for the password. If a user does not have a token, that means that user isn't logged in and is not able to go to specific routes and do any actions.

The other method, called "admin", checks if user is an administrator. The application basically checks if a user has administrator role set to true. If user is not an administrator, then an error is thrown to show that a user is not authorized as an administrator.

/backend/middleware/authMiddleware.js

```
1  import jwt from "jsonwebtoken";
2  import asyncHandler from "express-async-handler";
3  import User from "../models/userModel.js";
4
5  const protect = asyncHandler(async (req, res, next) => {
6    let token;
7
8    if (
9      req.headers.authorization &&
10     req.headers.authorization.startsWith("Bearer")
11   ) {
12     try {
13       token = req.headers.authorization.split(" ")[1];
14
15       const decoded = jwt.verify(token, process.env.JWT_SECRET);
16
17       req.user = await User.findById(decoded.id).select("-password");
18
19       console.log(decoded);
20       next();
21     } catch (error) {
22       console.error(error);
23       res.status(401);
24       throw new Error(
25         "Not authorized, token failed, please logout and try logging in again"
26       );
27     }
28   }
29
30   if (!token) {
31     res.status(401);
32     throw new Error("Not authorized, no token found");
33   }
34 });
35
36 const admin = (req, res, next) => {
37   if (req.user && req.user.isAdmin) {
38     next();
39   } else {
40     res.status(401);
41     throw new Error("Not authorized as an admin");
42   }
43 };
44
45 export { protect, admin };
46
```

Fig. 29 - Backend authentication middleware example

4.2.6 Backend – Server file

Server file is responsible to connect all the backend code together. First, we import all the necessary stuff, including our routes, we load “dotenv” configuration which will allow us to use environmental variables in our server, and assign app variable to express which we will use to connect everything.[2]

```
import path from "path";
import express from "express";
import dotenv from "dotenv";
import colors from "colors";
import morgan from "morgan";
import { notFound, errorHandler } from "./middleware/errorMiddleware.js";
import { connectDB, connectDBProd } from "./config/db.js";

// import routes
import productRoutes from "./routes/productRoutes.js";
import userRoutes from "./routes/userRoutes.js";
import orderRoutes from "./routes/orderRoutes.js";
import uploadRoutes from "./routes/uploadRoutes.js";

dotenv.config();

const app = express();

```

Fig. 30 - Backend server file example 1

After we have loaded our imports, we want to specify which environment the application currently is active on. If it’s the development environment, we will connect to our development database. If the application is in the production environment, then we connect to our production database, make our build folder static so we can access it and load “index.html”.

```
// imports

dotenv.config();

const app = express();

if (process.env.NODE_ENV === "development") {
  connectDB();
  app.use(morgan("dev"));
}

if (process.env.NODE_ENV === "production") {
  connectDBProd();

  // we want to set frontend/build folder as our static folder
  // so we can directly access it and load index.html
  app.use(express.static(path.join(__dirname, "/frontend/build")));

  // * - anything that isnt any of the above routes
  app.get("*", (req, res) =>
    res.sendFile(path.resolve(__dirname, "frontend", "build",
"index.html"))
  );
} else {
  app.get("/", (req, res) => {
    res.send("API RUNNING...");
  });
}

```

Fig. 31 - Backend server file example 2

And then we include the rest of information, all the routes, PayPal configuration route with our PayPal client API key (PayPal documentation that helped implementing into the project can be found here [7]). Also make our upload folder static so it gets loaded into the browser.

```
...

app.use(express.json());

// mounting routes
app.use("/api/products", productRoutes);
app.use("/api/users", userRoutes);
app.use("/api/orders", orderRoutes);
app.use("/api/upload", uploadRoutes);

// paypal config route
app.get("/api/config/paypal", (req, res) =>
  res.send(process.env.PAYPAL_CLIENT_ID)
);

// making uploads folder static so it can get loaded in the browser
const __dirname = path.resolve();
app.use("/uploads", express.static(path.join(__dirname, "/uploads")));

app.use(notFound);
app.use(errorHandler);

const PORT = process.env.PORT || 4000;

app.listen(
  PORT,
  console.log(
    `Server running in ${process.env.NODE_ENV} mode on port ${PORT}`.yellow.bold
  )
);
```

Fig. 32 - Backend server file example 3

4.3 Frontend

4.3.1 Frontend – Components

As we're using React.js framework for our front-end code, the main and most important part of the code are components. React uses component-based architecture. Each component represents specific element in the page. Here we can see our components of the application.

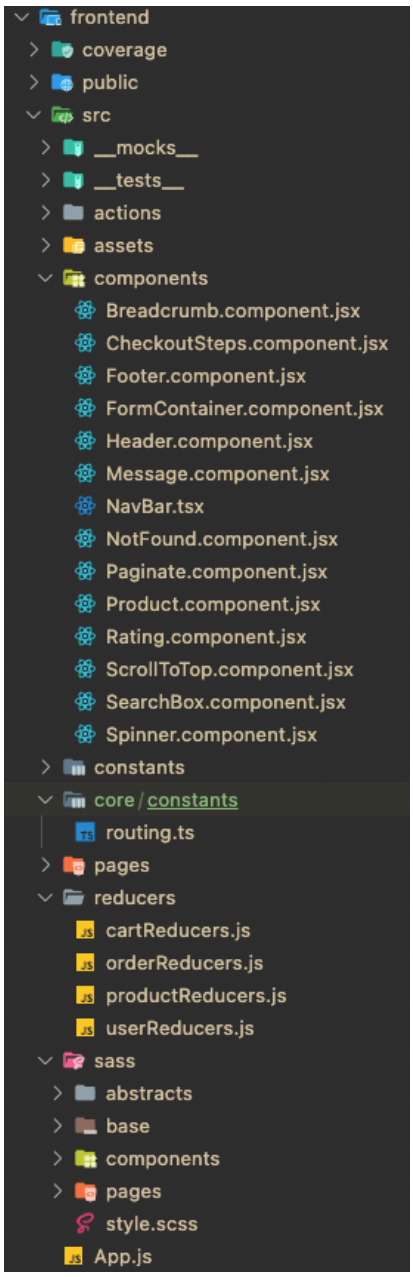


Fig. 33 - Frontend code structure

For example, we have a not found component which we want to reuse whenever we cannot find an existing page or a product. In that case we don't need to rewrite and handle the same piece of code. We can write it once, store it into a component and reuse it whenever we need to show it. Here we have a code example of that exact component:

/frontend/src/components/NotFound.jsx

```jsx
import React from "react";

const NotFound = () => {
  return (
    <section className="cart-section section-b-space">
      <div className="container">
        <div className="row">
          <div className="col-sm-12">
            <div>
              <div className="col-sm-12 empty-cart-cls text-center">
                <img
                  src={require(`${process.env.PUBLIC_URL}../assets/images/icon-page-not-found.jpg`)}
                  className="img-fluid mb-4"
                  alt="empty-cart-img"
                />
              </div>
            </div>
          </div>
        </div>
      </div>
    </section>
  );
};

export default NotFound;

```

Fig. 34 - Not found component example

We can see our component is written and ready to be used. To use this component is straightforward, we just need to add a route and attach it as a component, it will automatically render this not found component if any other of the components have not been found and rendered.

Fig. 35 - Frontend not found component usage

4.3.2 Frontend – Pages

The second most important thing after component in react is pages. Pages are basically more complete components. It can have multiple components withing a page. In the example, we can see couple of components being used. If the page is supposed to still be loading, then it will render loading spinner component, if it's loaded, it will loop through products and will render product components with paginate component.

frontend/src/pages/Homepage.js

```
...
  return (
    <>
        {loading ? (
          <div
            style={{
              position: "absolute",
              top: "50%",
              left: "50%",
              transform: "translate(-50%, -50%)",
            }}
          >
            <Spinner />
          </div>
        ) : error ? (
          <Message variant="danger"> {error} </Message>
        ) : (
          <div className="grid-products">
            {products.map((product) => (
              <div className="item">
                <Product product={product} />
              </div>
            ))}
          </div>
        )}
        <Paginate pages={pages} page={page} keyword={keyword ? keyword : ""} />
      </div>
    </>
  );
};

export default HomePage;

```

Fig. 36 - Frontend homepage example

4.3.3 Frontend – Redux

Redux has been used to manage application state. Redux is a JavaScript library to manage application state.

Once we click a button, we trigger (dispatch) an action. An action in redux returns an object and gives it to a reducer to handle it. Those reducers are put into one big store – combined reducer. And then we re-render our components depending on that state change. There are also effects that are used when we must make requests and fetch information from our backend. Basically, effects help us handle side effects.

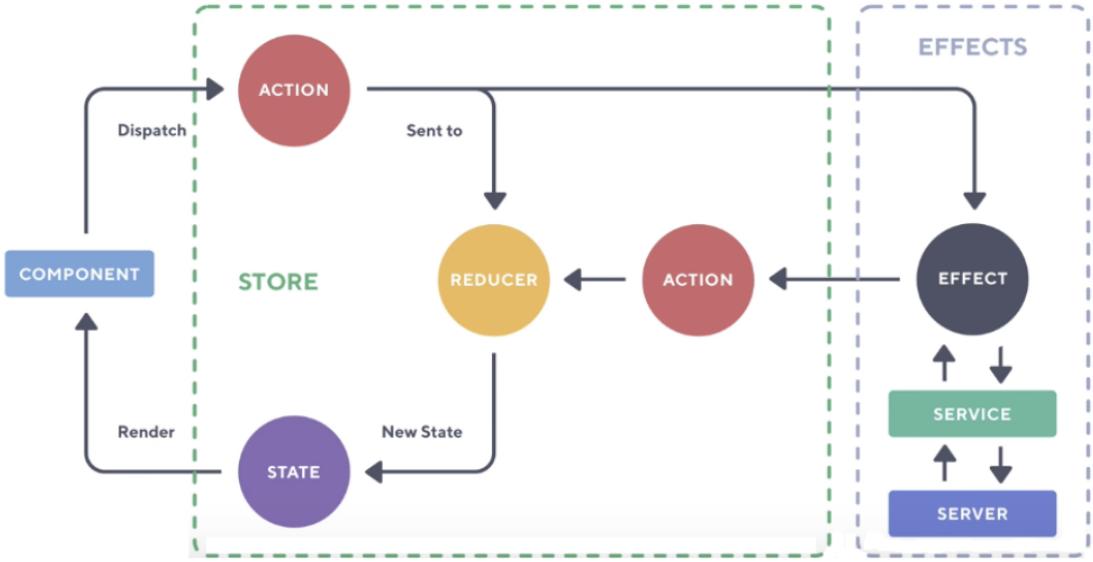


Fig. 37 - Redux example

Here's a simpler graph without any effects taken into consideration:

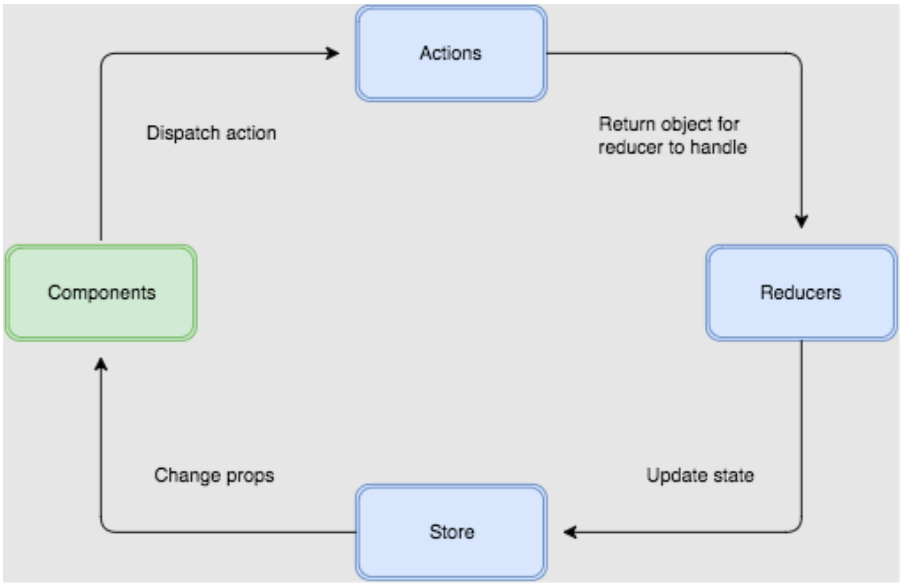


Fig. 38 - Redux example simplified

4.3.3.1 Redux - setup

In order to use redux in our react project, we need to install few dependencies to work with redux. It's quite a few actually (Documentation for Redux helped setting things up [5]):

- **Redux** – we need to install redux package itself. It's not part of react and can be used for any language.

- **React-redux** – this package helps us to make react work with redux.
- **Redux-thunk** – allows us to make asynchronous requests in our action creators which is basically a middleware.
- **Redux-devtools-extension** – in our browser we can install an extension (Redux DevTools), but t requires this package to be installed for it to work in the browser.

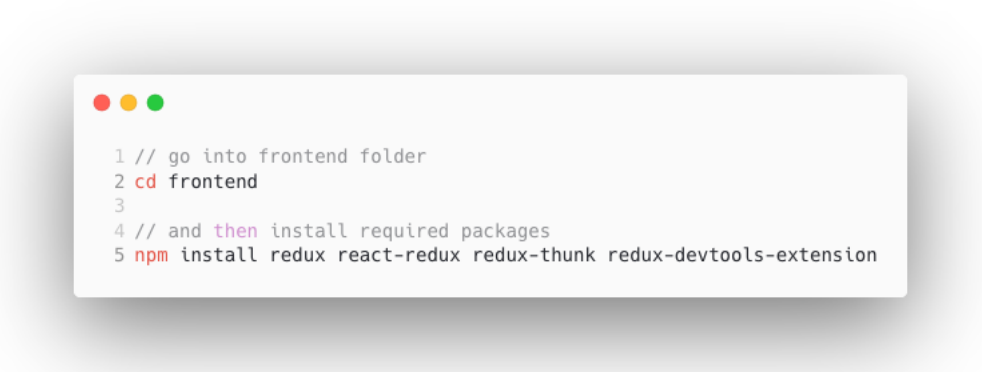


Fig. 39 - Redux installation

After installing all these dependencies, we want to create redux store file. In our "frontend/src" directory we want to create "store.js" file. This is where we connect our reducers, any middleware's and stuff like that.

/frontend/src/store.js

```
import { createStore, combineReducers, applyMiddleware } from "redux";
import thunk from "redux-thunk";
import { composeWithDevTools } from "redux-devtools-extension";

// reducer (we will bring reducers here)
const reducer = combineReducers({});

// if we want something to be loaded when redux-store loads,initially we can put it here
const initialState = {};

// array of all middlewares
const middleware = [thunk];

// creating store (first thing we need to pass is our reducer, second initialState and third
middleware)
const store = createStore(
  reducer,
  initialState,
  composeWithDevTools(applyMiddleware(...middleware))
);

export default store;
```

Fig. 40 - Redux store example

And the way we use our store in our application, we just use it through a provider. In our “index.js” we want to import the store and the wrap the Application component inside the provider, so we provide our state to the application.

/frontend/src/index.js

```
1 import React from 'react';
2 import ReactDOM from 'react-dom';
3 import { Provider } from 'react-redux';
4 import store from './store';
5 {...}
6 ReactDOM.render(
7   <Provider store={store}>
8     <BrowserRouter>
9         <App />
10    </BrowserRouter>
11    </Provider>,
12  document.getElementById('root')
13 );
14
15 serviceWorker.unregister();
```

Fig. 41 - Redux store wrapper

4.3.3.2 Redux - reducer

To begin using redux, we want to create reducers folder in our “frontend/src” directory. Each resource of our application will have a reducer file such as products, orders or users.

So now in our reducers folder we can create “productReducers.js”. We're going to have bunch of different reducer functions in here. But they are all going to be related to products.

First is going to be product list reducer and it’s going to handle state for product list which we see on the home page. Reducer takes in two things, initial state and an action.

Now when we create an action reducer, we're going to dispatch an action to this reducer (productListReducer for example), And this will be an object that has a type. The type we're going to evaluate in the function. Action might also contain a payload. In here we're getting products, so it will have a payload with those products in it that we fetch from the server.

/src/reducers/productReducers.js

```
// in the initial state we want to add products and set it to empty array.
export const productListReducer = (state = { products: [] }, action) => {
  // we want to evaluate the type in the action so we use switch for that
  switch (action.type) {
    case "PRODUCT_LIST_REQUEST":
      return { loading: true, products: [] };
    case "PRODUCT_LIST_SUCCESS":
      return { loading: false, product: action.payload };
    case "PRODUCT_LIST_FAIL":
      return { loading: false, error: action.payload };
    default:
      return state;
  }
};
```

Fig. 42 - Redux reducer example

Now to use this reducer we must add it to our store. So, in store.js we must import our product reducers.

/frontend/src/store.js

```
import { createStore, combineReducers, applyMiddleware } from "redux";
import thunk from "redux-thunk";
import { composeWithDevTools } from "redux-devtools-extension";
import { productListReducer } from "./reducers/productReducers";

const reducer = combineReducers({
  productList: productListReducer,
});

{...}
export default store;
```

Fig. 43 - redux store reducer example

Now typically we want to put these strings such as "PRODUCT_LIST_REQUEST", "PRODUCT_LIST_SUCCESS" and so on into a file called constants.js because they never change.

So now we can create a new folder called constants in our “frontend/src/” and create a file “productConstants.js”.

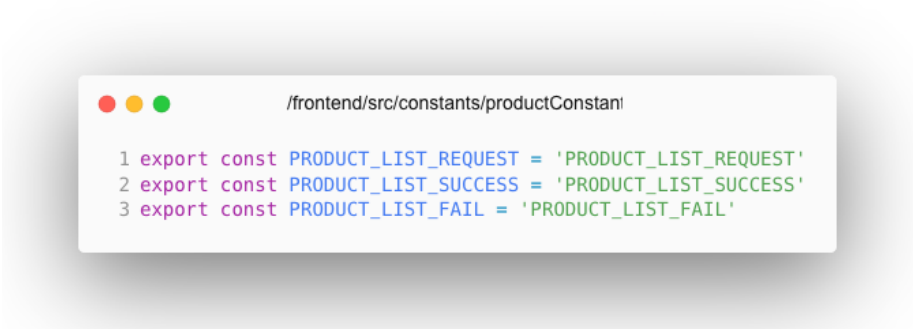


Fig. 44 - Redux constants

And now we can bring these constants into our product list reducer. This reducer has initial state of empty products array since we want to load them here. When we dispatch a request to load products, it will set loading state to true and have an empty products array. When it finished loading, it would dispatch an action depending on the action if it succeeded or failed. If it succeeded, it would dispatch an action with products. On the other hand, if it failed, it would dispatch an error.

Fig. 45 - Redux full reducer example

4.3.3.3 Redux - actions

Redux actions are where we make requests and other logic depending on the state. So now that we have our reducer set up, we want to add actions. So, in "/frontend/src" we want to create a folder "actions" and then we want to create a new file "productActions.js". We also want to bring constants we wrote earlier into an action as well.

Fig. 46 - Redux action example

And to finally dispatch this action we wrote and list our products into the browser, we want to add all product list state and then dispatch that action that we created in a "useEffect". This method allows us to perform side effects in our components. When we dispatch our action, we can finally loop through our products, and it will display each product in a product component.

/frontend/src/pages/HomePage.compon

```
1 ...
2 import Product from "../components/Product.component";
3 import { listProducts } from "../actions/productActions";
4
5 const HomePage = () => {
6   const dispatch = useDispatch();
7
8   const productList = useSelector((state) => state.productList);
9   const { loading, error, products } = productList;
10
11  useEffect(() => {
12    dispatch(listProducts());
13  }, [dispatch]);
14
15  return (
16    <>
17      {loading ? (
18        <h1>Loading</h1>
19      ) : error ? (
20        <h3>{error}</h3>
21      ) : (
22        <Row>
23          {products.map((product) => (
24            <Col key={product._id} sm={12} md={6} lg={4} xl={3}>
25              <Product product={product} />
26            </Col>
27          ))}
28        </Row>
29      )}
30    </>
31  );
32 };
33
34 export default HomePage;
```

Fig. 47 - Redux action usage example

5. USER MANUAL

5.1 Serving & fetching data from Express

React.js is our frontend framework which we're viewing in the browser. To create a full stack application, we need to have a backend. Our backend is created with node.js which is a JavaScript runtime which allows us to run JavaScript on a server. Express is a backend framework which allows us to create routes and where we communicate with our database which is MongoDB.

To communicate with our database, we use "object data mapper" – mongoose. It has methods like find, "findById". They make it easy to interact with our database. To get data from the backend we need to make HTTP request. If we want to fetch products from our backend, we will do a GET request. And we will send back the data in JSON to our frontend where we will display the information.

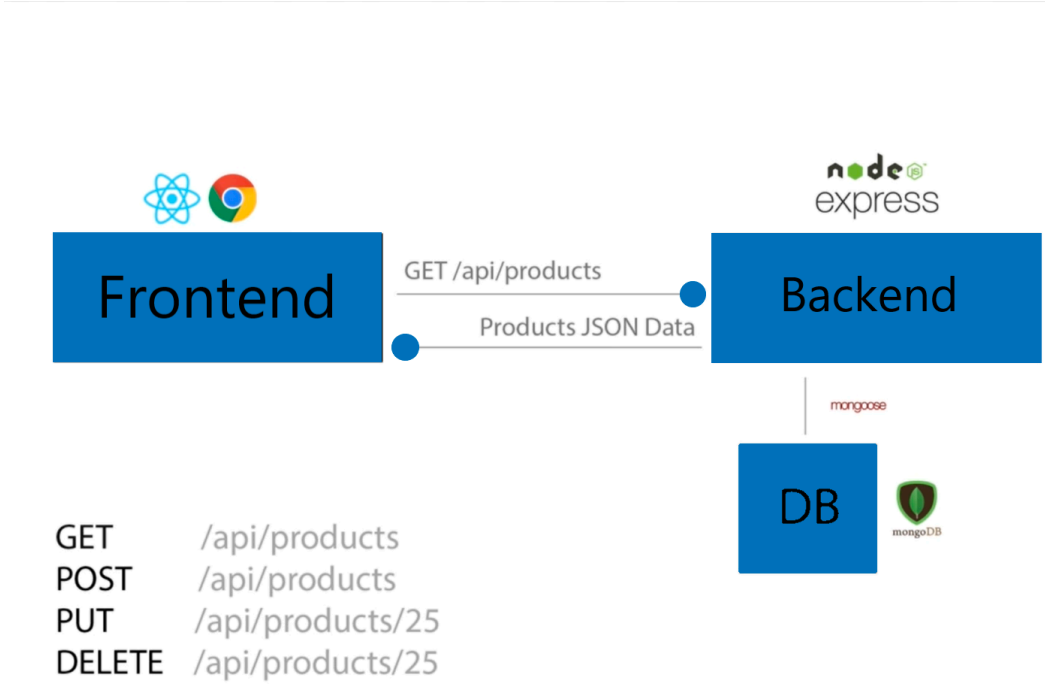


Fig. 48 - Express.js example diagram

We basically want our controllers top handle the functionality and routes should only point to the controller methods. So, in backend we create a folder called "controllers" and create a file "productController.js".

/backend/controllers/productController.js

```
1 import asyncHandler from "express-async-handler";
2 import Product from "../models/productModel.js";
3
4 // @desc Fetch all proucts
5 // @route GET /api/products
6 // @access Public
7 const getProducts = asyncHandler(async (req, res) => {
8   const products = await Product.find({});
9   res.json(products);
10 });
11
12 // @desc Fetch single prouct
13 // @route GET /api/products/:id
14 // @access Public
15 const getProductById = asyncHandler(async (req, res) => {
16   const product = await Product.findById(req.params.id);
17
18   if (product) {
19     res.json(product);
20   } else {
21     res.status(404);
22     throw new Error("Product not found");
23   }
24 });
25
26 export { getProducts, getProductById };
```

Fig. 49 - Product controller code example

When we have logic setup in our controllers, we can use these methods in routing files. If a user visits root page, the application will fetch all products and if a user visits a page that has an id, it will fetch a product according to the id that is specified.

/backend/routes/productRoutes.js

```
1 import express from "express";
2 const router = express.Router();
3 import { getProducts, getProductById } from "../controllers/productController.js";
4
5 router.route("/").get(getProducts);
6
7 router.route("/:id").get(getProductById);
8
9 export default router;
```

Fig. 50 - Product routes code example

5.2 Getting started with MongoDB

5.2.1 MongoDB Atlas setup

Firstly, to use MongoDB, we need to login into their website. When we come into their website, we head to the login page and enter our credentials.[3] If we don't have an account, then we need to create one.

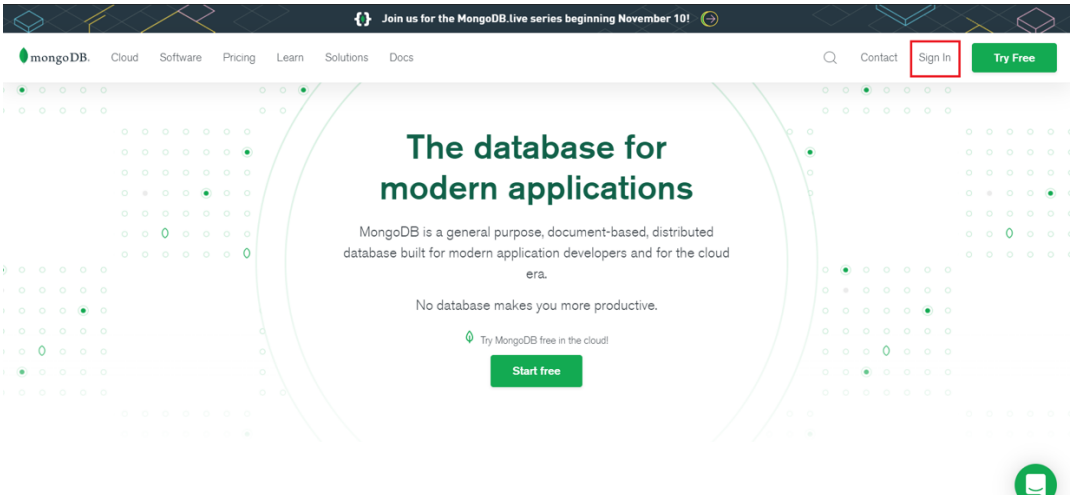


Fig. 51 - MongoDB homepage

Once we have logged in into MongoDB, we're going to be met with this page. We want to create an organization that we're going to use, so we click this the big green button called "Create an organization" (To set up the database, MongoDB documentation came in real handy [4]).

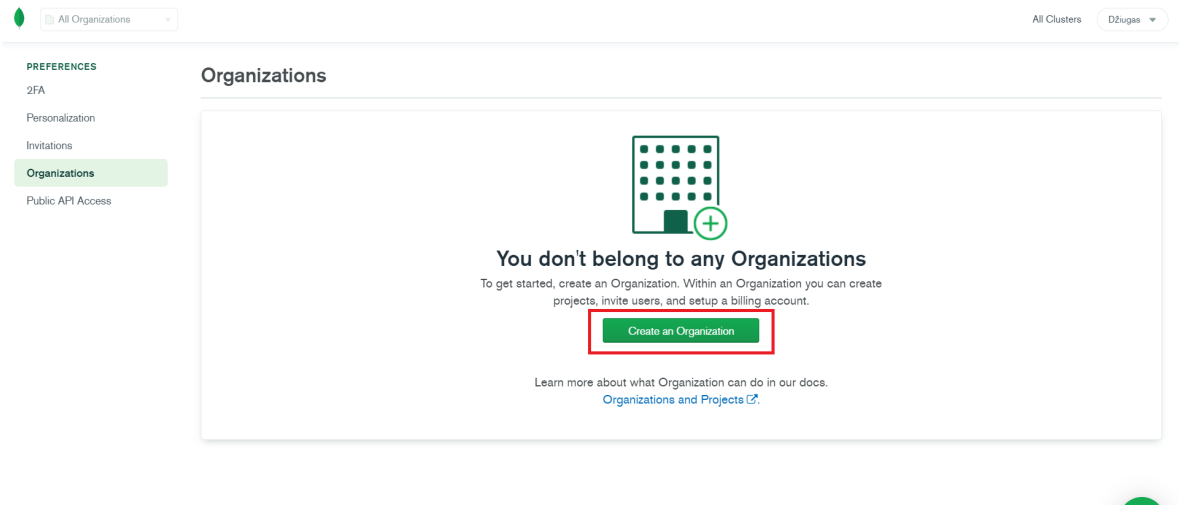


Fig. 52 - MongoDB create organization step 1

Then a name must be given to the organization and select a service that is going to be used. In this case MongoDB Atlas is used.

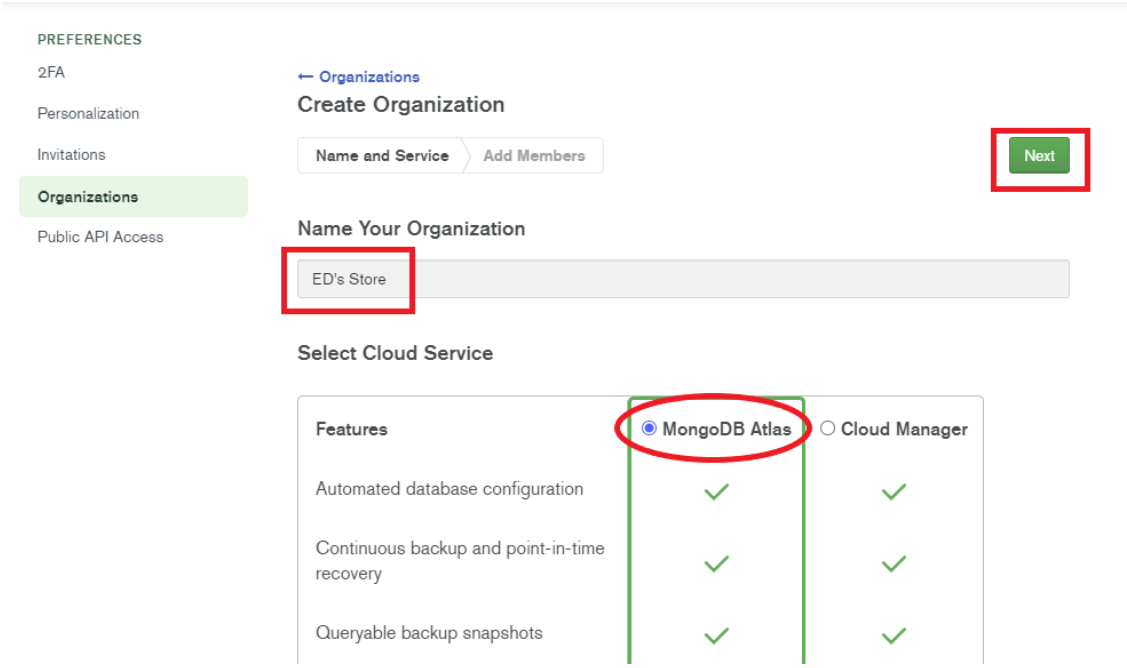


Fig. 53 - MongoDB create organization step 2

And a final step to create an organization is just to add any members if there are any and finish the process.

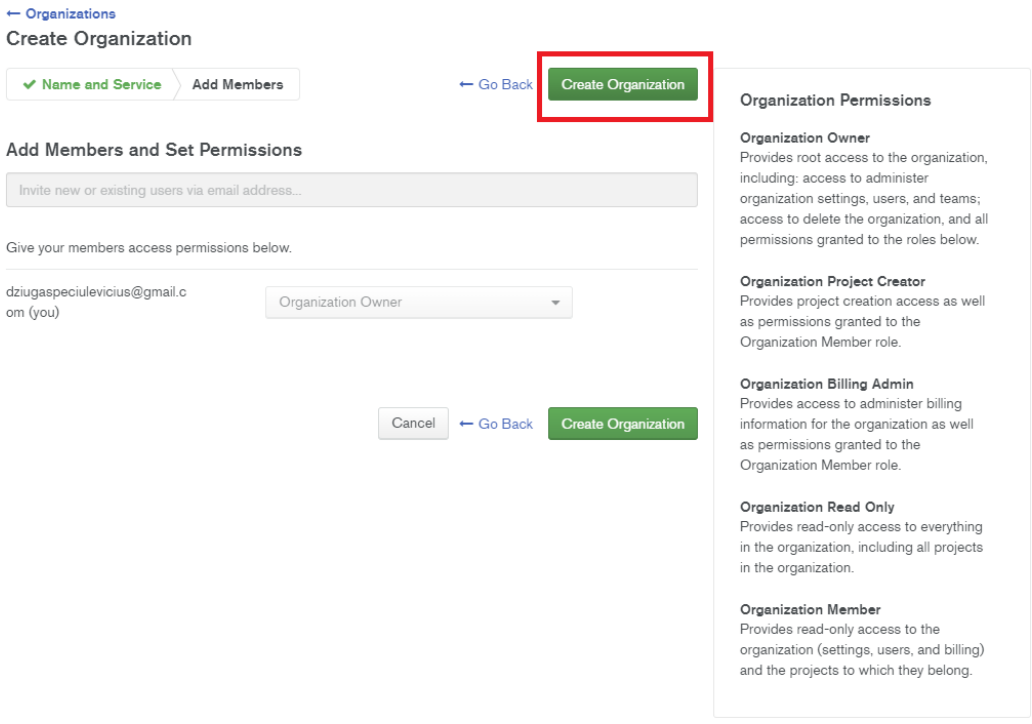


Fig. 54 - MongoDB create organization step 3

Once we have created our organization, we can create a project that will have our database within that project. So, on the left-hand side we select “Projects” if that has not been yet selected, and then on the right-hand side we click the “New Project” button.

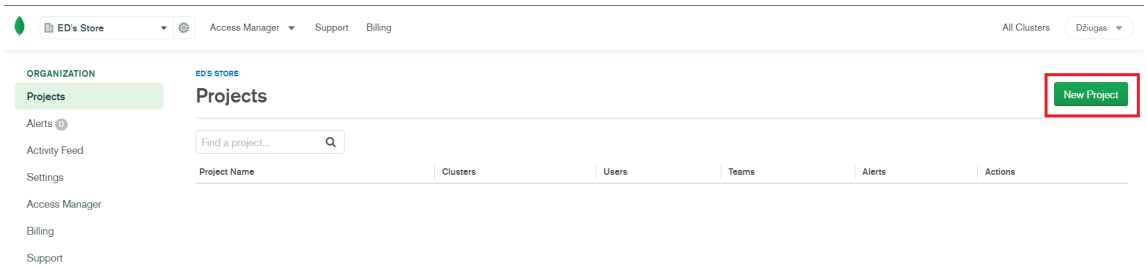


Fig. 55 - MongoDB create project step 1

Then a name must be given for the project and after that we can proceed to the next step of the process.

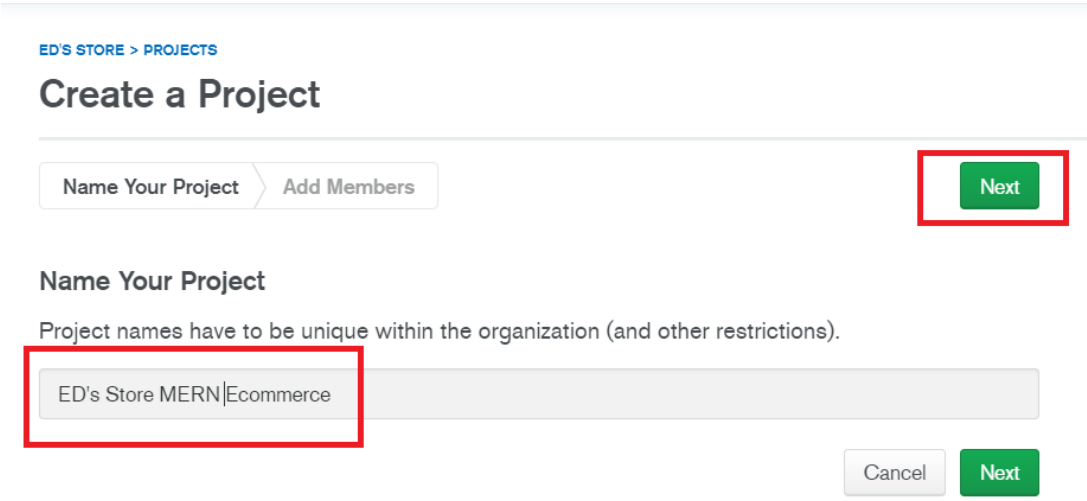


Fig. 56 - MongoDB create project step 2

Then the setup would ask us again to add members and give them permissions. So that would have to be done before creating a project. In this case, there are no other members were included so then we finish creating a project by clicking “Create Project”.

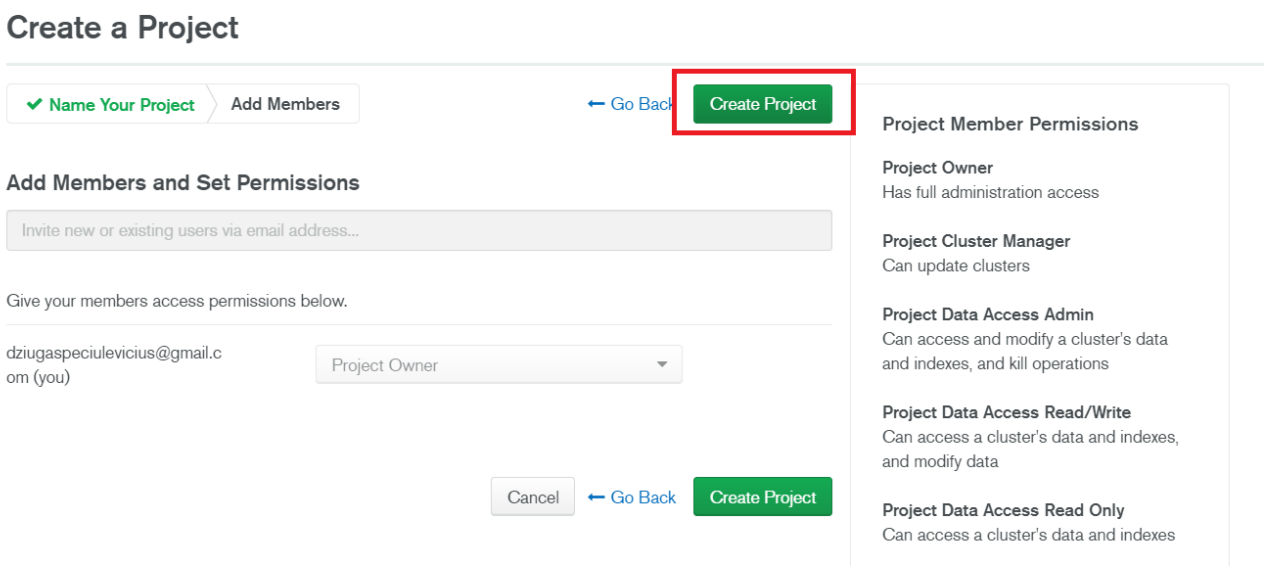


Fig. 57 - MongoDB create project step 3

We now have our project created and we can see it in our projects panel.

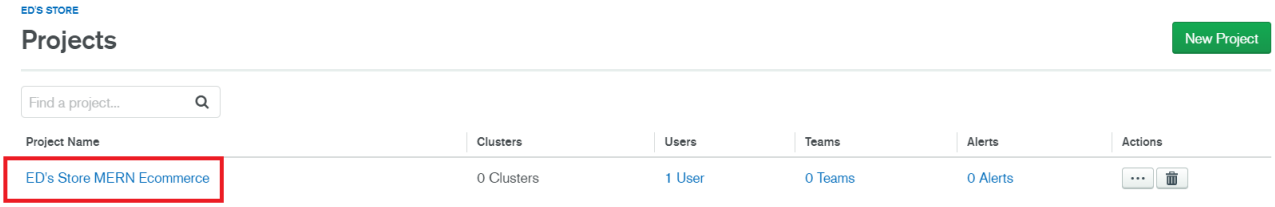


Fig. 58 - MongoDB create cluster step 1

We can now access our project, and we need to create a cluster within it. MongoDB cluster allows us to scale our database across many servers.

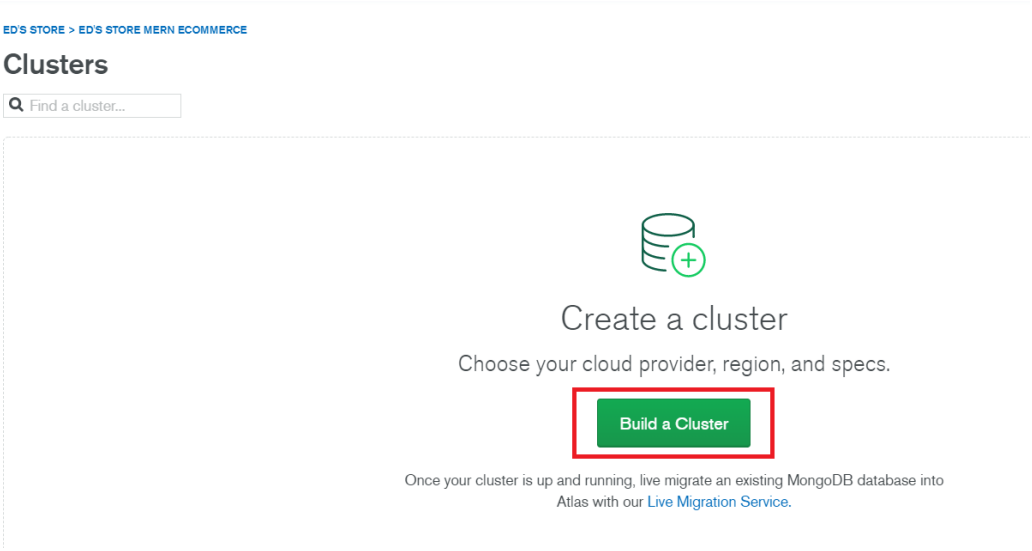


Fig. 59 - MongoDB create cluster step 2

Now select type of clusters needed, we're going to stick to free ones. Because we're not using such a huge application that we would need a bigger server capacity. If you are planning to have more users or seeing a slowdown in the performance or just want to see real-time performance metrics, then you might think about upgrading your cluster.

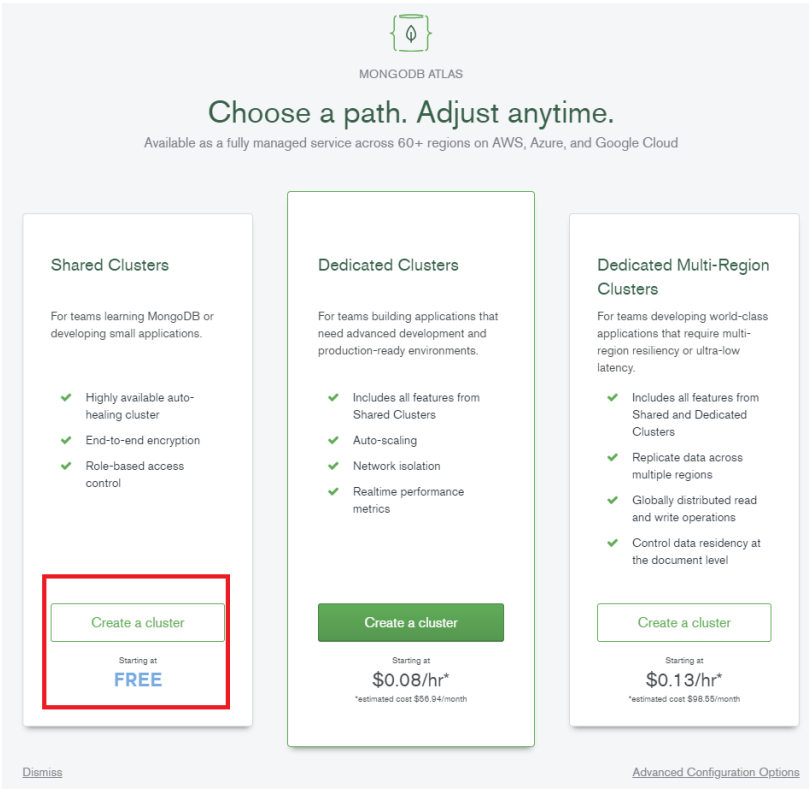


Fig. 60 - MongoDB create cluster step 3

Now we chose a preferred provider, in this case AWS was chosen, and a for a region, it would be best to consider where most of the traffic would be coming from and chose the region around that location.

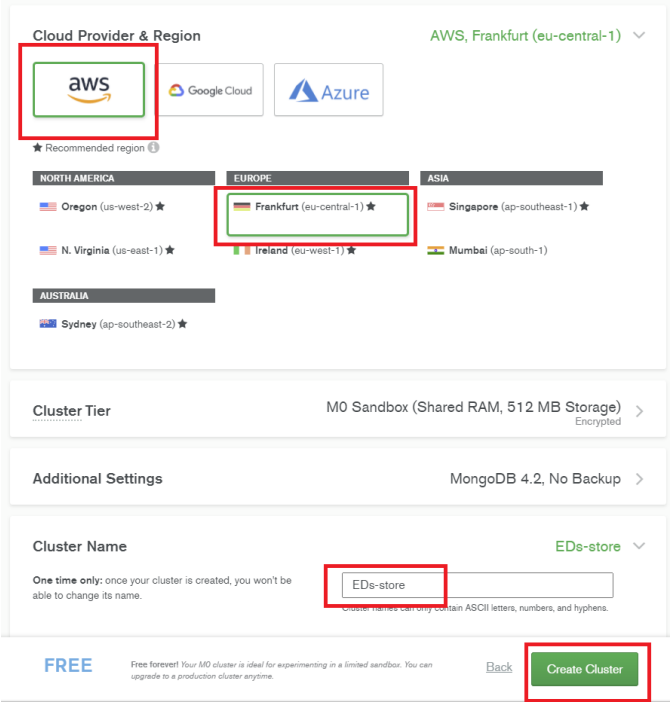


Fig. 61 - MongoDB create cluster step 4

After finishing up with the last step, we can see a new cluster being created. This process takes up to couple of minutes.

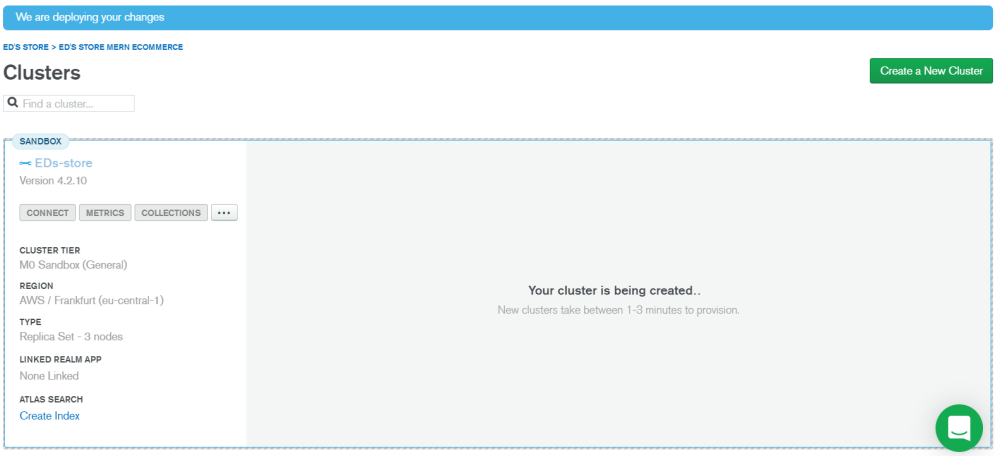


Fig. 62 - MongoDB create cluster step 5

Once our cluster is created, we need to add a user that would have access to the database. We can create users and limit their access to read-only or however we want it. We can also set it, so it automatically deletes the user after a specified amount of time has passed.

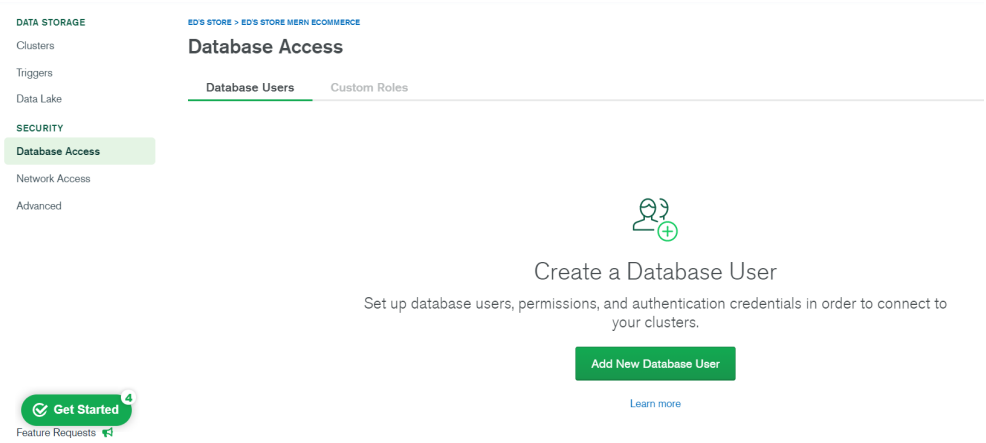


Fig. 63 - MongoDB create access for a user step 1

The password we're using here, we will need to use in API key to connect it to our application, so it's important not to forget the password.

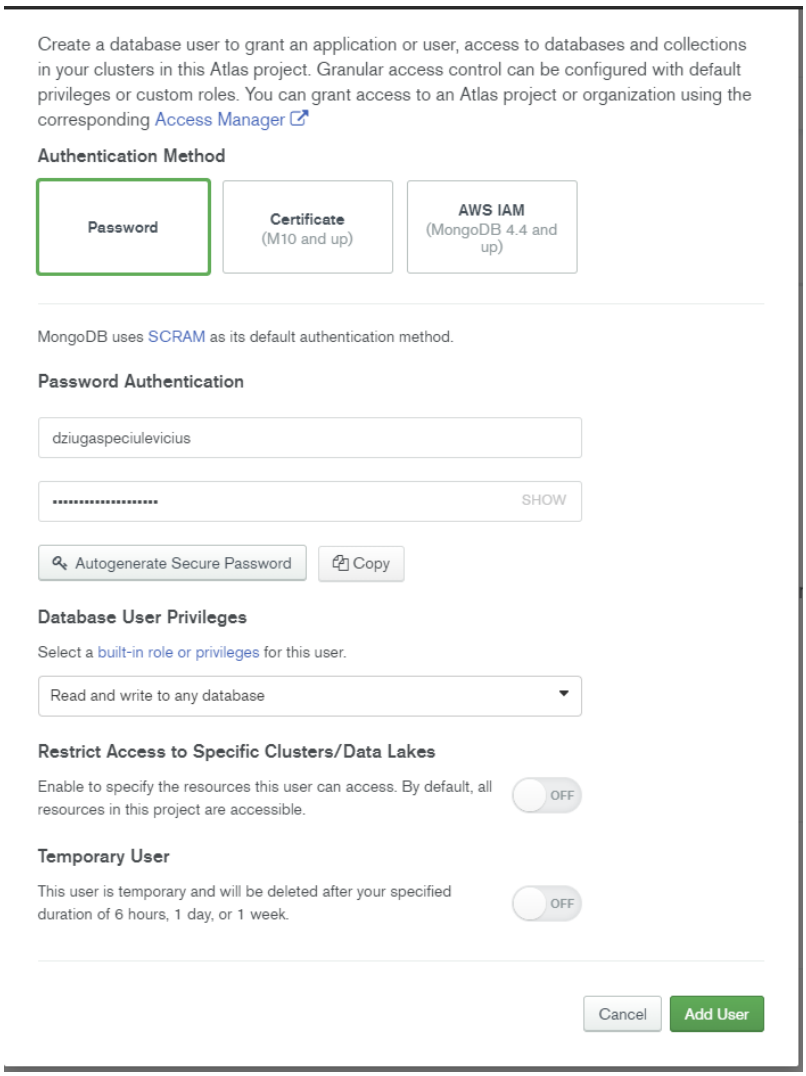


Fig. 64 - MongoDB create access for a user step 2

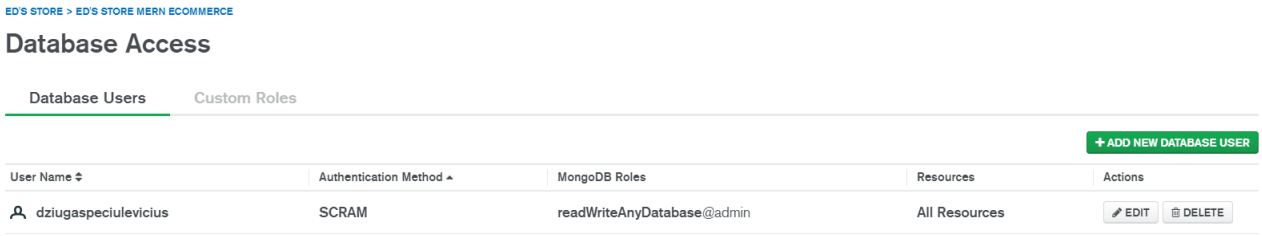


Fig. 65 - MongoDB create access for a user step 3

And finally, we can create collections inside our clusters.

Fig. 66 - MongoDB create collections step 1

Inside the cluster, we can head to collections where we should see a pop-up.

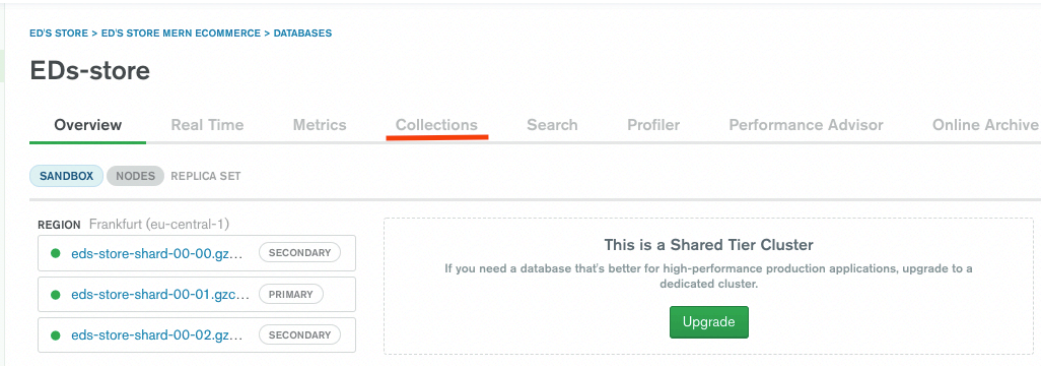


Fig. 67 - MongoDB create collections step 2

Since there are no collections created within a database for this cluster, a user will be asked to create one.

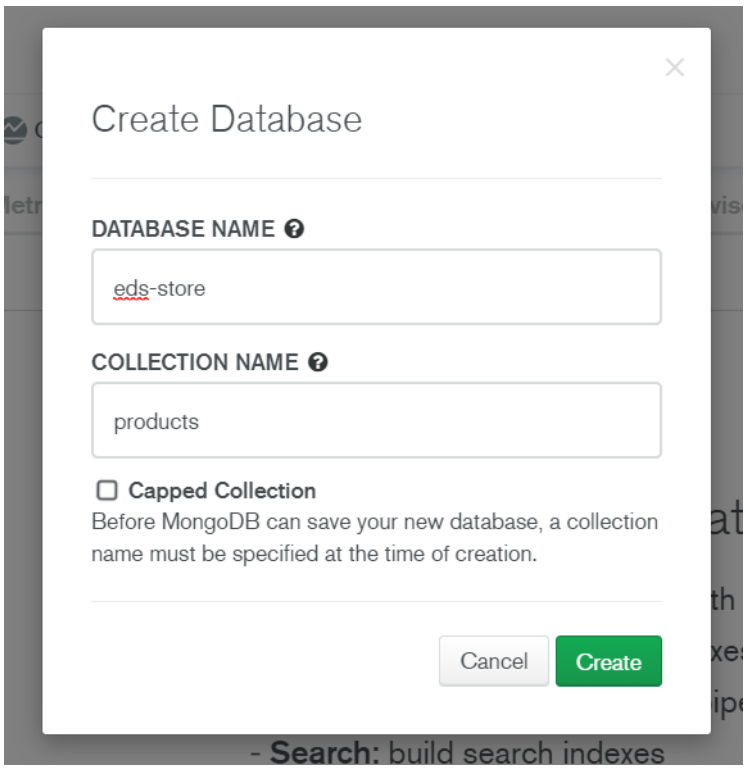


Fig. 68 - MongoDB create collections step 3

And now when we have all our collections created, we can now connect it to our application. All we need to do, is save the API key that MongoDB gives us.

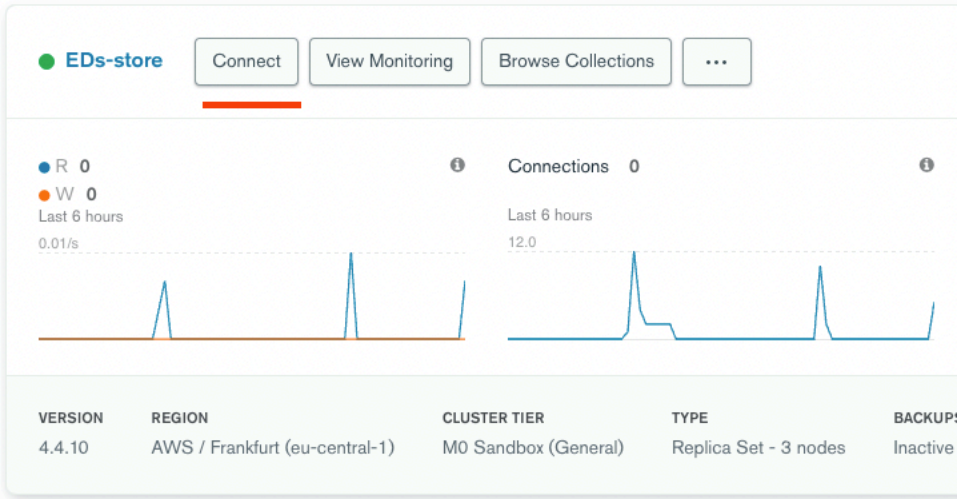


Fig. 69 - MongoDB connect to application step 1

When connecting the application, we're given a chance to connect via other methods, but we're going to use plain API keys to connect to the database.

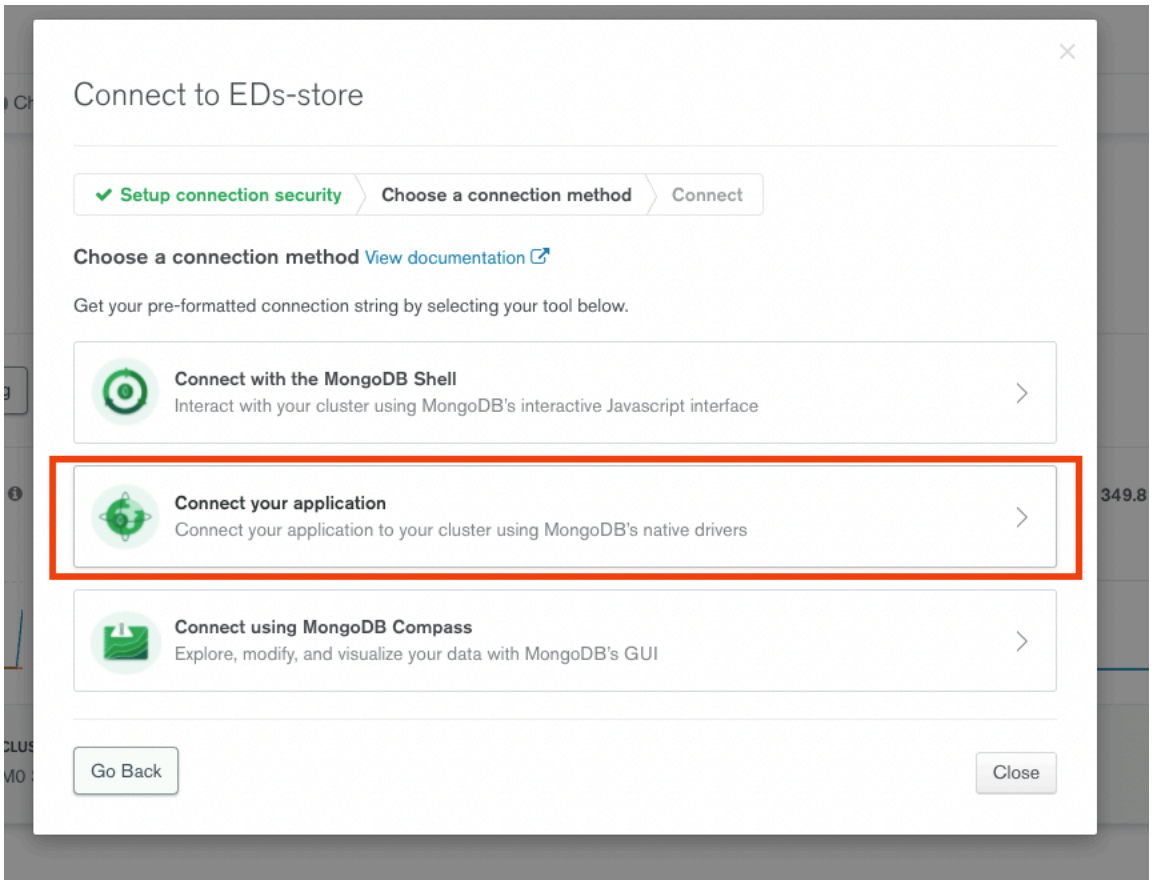


Fig. 70 - MongoDB connect to application step 2

After choosing a method on how we want to connect, we're able to choose what kind of application we want to connect to, and the API key is given to us. To use this API key, we need to replace the password bit with the password that we created a user with and include our database name where needed.

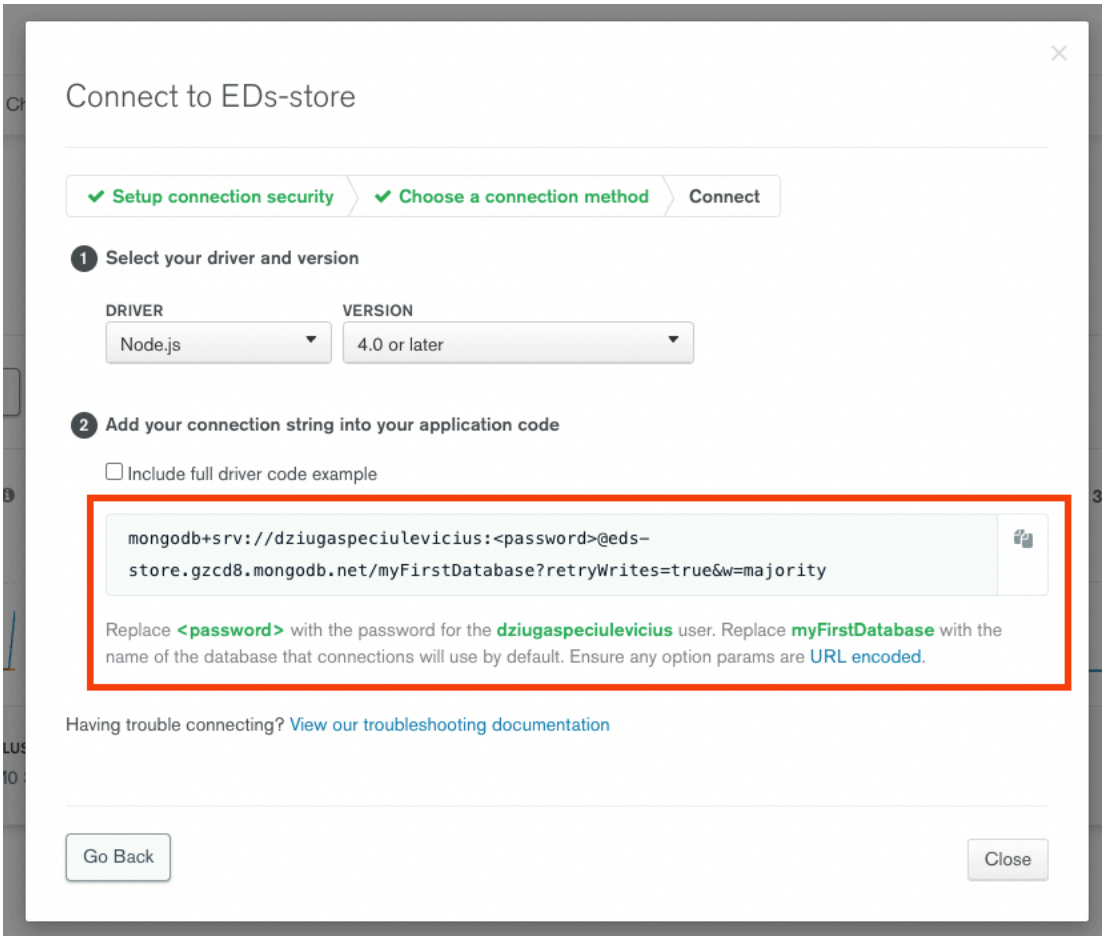


Fig. 71 - MongoDB connect to application step 3

5.2.2 Connecting to database

We use mongoose to connect to our database. Mongoose is object modelling for node.js. It allows us to create a model and a schema for different resources for our database like products, users and so on.

In the root of our project, we want to install mongoose:

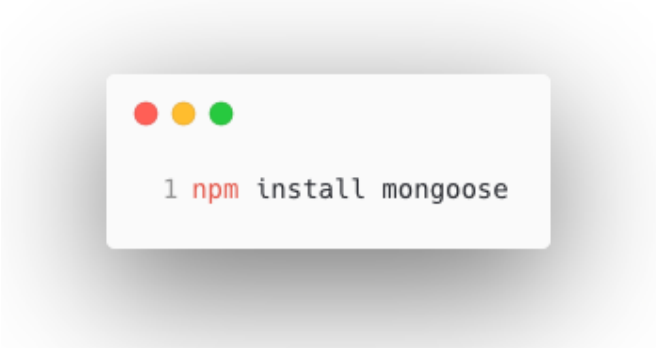


Fig. 72 - Installing mongoose

After installing mongoose, we can connect to our MongoDB database using our database configuration file. The method takes a Mongo URI as the first argument, and a set of options as a second argument. With the current version of mongoose, the application fails if we don't pass them through.

/backend/config/db.js

```
1 import mongoose from "mongoose";
2
3 const connectDB = async () => {
4   try {
5     const conn = await mongoose.connect(process.env.MONGO_URI, {
6       useUnifiedTopology: true,
7       useNewUrlParser: true,
8       useCreateIndex: true,
9     });
10    console.log(`MongoDB connected: ${conn.connection.host}`.cyan.underline);
11  } catch (error) {
12    console.error(`Error: ${error.message}`.red.underline.bold);
13    process.exit(1);
14  }
15 };
16
17 export { connectDB };
18
```

Fig. 73 - backend database connection configuration

5.3 Deployment on Heroku

When we build our application, there's one thing left and it's to host the application somewhere where a user could access the application, which is usually a cloud service or other hosting platform. (Documentation that helped hosting the website on Heroku [8]).

5.3.1 Preparation for deployment

To prepare our application for deployment to Heroku, first, we want to create a production build.

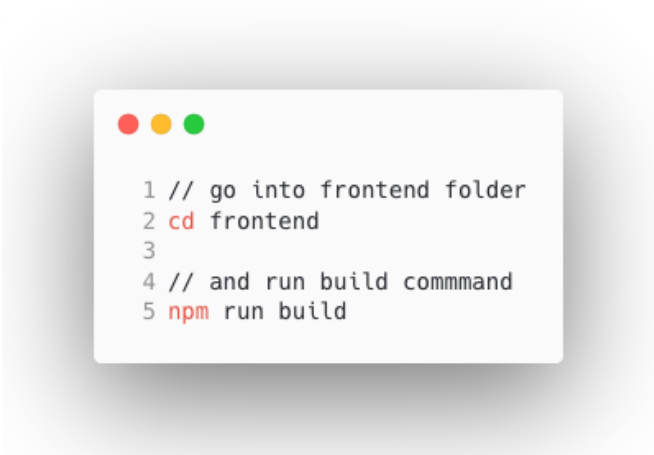


Fig. 74 - Building a project via command line

If we look in our folder html file in react project, this is the page we want to load in production. So, in our backend server file, under routes, we want to add statement if we're in production, then it should use our build.

Fig. 75 - server.js file example for project build

5.3.2 Deploy to Heroku

So, for deployment we want to set our “NODE_ENV” to development, and we need to make sure we have our “.env” in “.gitignore so we don’t push any of our API keys to public.

First, we need to set up our Heroku account. It will show a message to press anywhere to log in and it will open a browser.

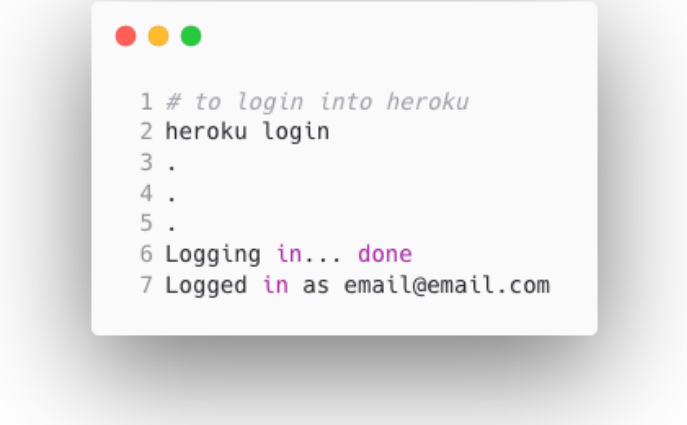


Fig. 76 - Login into Heroku account

After we have logged into the account, we want to create a new application in Heroku.

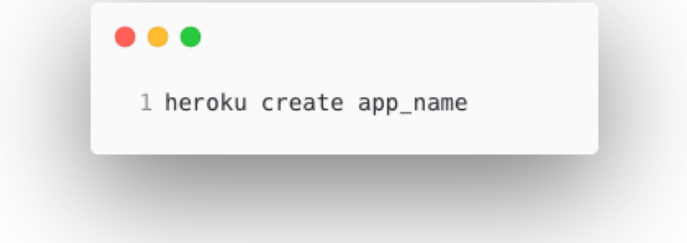


Fig. 77 - Create Heroku application

This will give us the git and the URL to the application. Before we push to Heroku we need to add a Procfile in the root. It basically tells Heroku what to run to run the application.

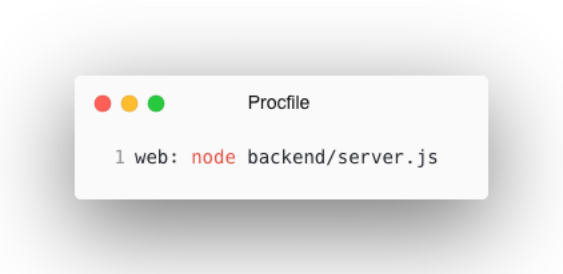


Fig. 78 - Procfile

Then we want to add a postbuild script into our package file in our root directory. This script basically will temporarily set “NPM_CONFIG_PRODUCTION” to false. Then it’s going to go into the frontend folder and run “npm install”. It’s also going to run “npm run build” which will create that build folder that has the “index.html” file in it.

Fig. 79 - package.json with postbuild script

Now when we have our project setup, we can push everything to Heroku.

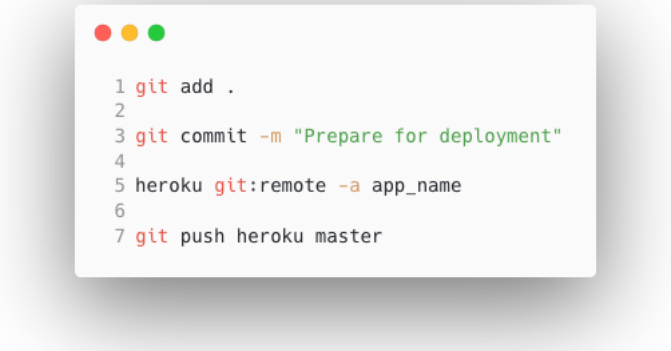


Fig. 80 - Committing to Heroku

When we have all project files pushed to Heroku, just like in GitHub, all we have left to do is just add our API keys into the Heroku environment. Because while developing our application, we can keep our API keys in a file, since we don't want to share them with anybody - we don't push them into GitHub together with our project. But for a hosted application to continue using other online services like MongoDB or PayPal, we somehow need to share our API keys. Heroku has a simple solution for that. In their website, on the project dashboard, they have a section where we can store all the API keys. And once saved, there's no need to do anything else.

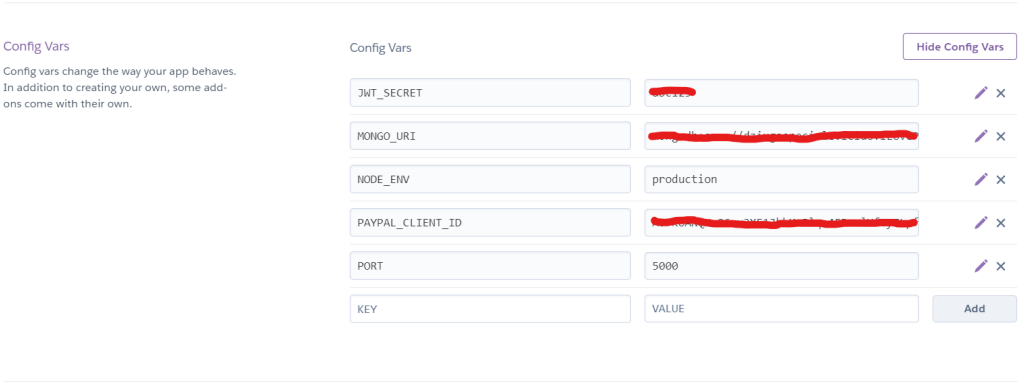


Fig. 81 - Heroku configuration variables

CONCLUSIONS AND RECOMMENDATIONS

For the conclusions we can say that all the functional requirements were completed successfully.

1. First project task was to plan and analyse what kind of functionality will be implemented and how the design for the application would look like. All requirements were fulfilled during the development process.
2. Second project task was to create a front-end part of the application. User Interface part was programmed using React. Chunks of the user-interface were divided into smaller components for more reusable code. When it came to managing user state, Redux came into help.
3. Third task was to create the backend. For the backend, Node.js together with Express.js was used. Backend was implemented successfully short after the need to have routes and a ready API to fetch data from a database.
4. Fourth task was to create a database where all user and product data would be stored. At first only development database has been created, but then a production database was created to help separate test data with the data that was planned to use in production.
5. The final task was to host the application in a cloud service. Heroku. Heroku provides a great developer experience which speeds up developers work. Heroku also provides a Command Line Interface (CLI) which makes for easy management.

The application is still under development and will have a separate mobile application for users to browse on their phones. There are some things that will be changed and / or implemented in the future:

1. As accessibility standards are improving as times goes on, there are plans to make the application accessible for users with disabilities, so users can browse the page using a screen reader.
2. There are also plans to replace JavaScript with TypeScript since the application is going to be scaled and improved, so TypeScript would help us reduce number of bugs in the future.
3. Also, there are plans to store and move repositories into a single monorepo since there are plans to have a separate phone application, so we would be able to have and share components between applications.

LIST OF INFORMATION SOURCES

1. Shopify. 2021. Ecommerce Definition - What Is Ecommerce. [online] Available at: <https://www.shopify.com/encyclopedia/what-is-ecommerce> [Accessed 12 November 2021].

2. Express. 2021. Express.js writing middleware's. [online] Available at:

<https://expressjs.com/> [Accessed 6 August 2021]

3. MongoDB. 2021. MongoDB home website. [online] Available at:

<https://www.mongodb.com/> [Accessed 6 August 2021]

4. MongoDB documentation. 2021. MongoDB documentation website. [online] Available at:

<https://docs.mongodb.com/> [Accessed 6 August 2021]

5. Redux documentation. 2021. Redux documentation website. [online] Available at:

<https://redux.js.org/introduction/getting-started> [Accessed 28 July 2021]

6. JSON Web Token. 2021. JWT package. [online] Available at:

<https://www.npmjs.com/package/jsonwebtoken> [Accessed 1 August 2021]

7. PayPal documentation. 2021. PayPal documentation website. [online] Available at:

<https://developer.paypal.com/docs/checkout/> [Accessed 10 August 2021]

8. Heroku documentation. 2021. Heroku getting started. [online] Available:

<https://devcenter.heroku.com/articles/getting-started-with-nodejs#set-up> [Accessed 17 December 2021]