3d shop – project documentation & project build process

Created & Documented by Filip Vojnovski

1 CONTENTS

2 ABOUT THE PROJECT

The goal of this project is to create a web application where users can trade 3d models as products. Each user is able to upload their own product, as well as buy products from other users. The other important feature of this project is that the users are able to view the models in their web browser.

The goal of this document is to give the reader an understanding of how to use this web application, how this web application works in the background, the research that was done in order to create this application and inform the reader of how this application can be upgraded or improved.

Instead of a standard approach of writing a documentation where the reader is informed of the existing code and its functionalities in a very professional manner, this document aims to help the user understand the creation process of this application, as well as how to upgrade it or find some advice if they intend to build additional functionalities. This is done by explaining and exploring the development process of the project as well as providing some insight into the decisions that went in certain crossroads when deciding of the project's future.

The reader should have a basic understanding of the pHp programming language, the Laravel framework and the React library.

Refer to the table of contents if you are interested only in information about only certain aspects of the project.

3 DEVELOPER ENVIORNMENT

This project was built on a Windows 10 operating system, using pHp storm as an IDE for creating the server-side functionalities and VSCode for building the client-side functionalities.

Although during the entire development process the operating system used was Windows 10, I advise that other developers who want to work on this project to use Linux as it makes using pHp and Laravel much easier, especially if the developer is looking for solutions online to configuration related problems.

The application Postman has been used for sending data for the endpoints, although it is not necessary, using an application such as Postman or Insomnia which allows for sending requests to a server makes testing endpoints and visualizing responses much easier.

4 TECHNOLOGY STACK

The choice of database is SQLite which is a **relational database**. Other alternatives were possible, such as using a MySQL server for the database or using PostgreSQL. Keep in mind that the project is

configured in such a way that switching to a different database is entirely possible and is only a matter of changing some configuration settings and migrating the existing data.

Another consideration for this project was to use a non-relational database, the reasons for these were that a 3d object may be provided in more formats and may contain a varying number of resources, but the decision was to go with using a non-relational database since it requires less documentation, is mostly easier to use alongside Laravel and is better for the “shop” aspect of the project.

The API is created using Laravel. Using another technology such as Java Spring or ASP.NET was possible and would yield the same results if done properly. The Laravel project holds the entire API of the project and its function is to process client requests.

The client side has been created using React. This was an important choice to make, and the reason is as follows: since rendering 3d geometry on the browser is not supported by a wide variety of frameworks / libraries in a convenient manner, the one chosen for this project was React because by choosing React, access to the three.js functionalities and features is made easier by the React Three renderer.

Further details on why this technology was chosen and how this library is used will be provided in the other sections of this document.

5 DATABASE STRUCTURE

Note: I have been using DB Browser for SQLite for viewing and editing data in the database, this tool is simple to use and I recommend it.

This project has a simple database, with only three main tables: users, products and sales.

The users table has been created with help from the Sanctum, which is the authentication system of choice for this project. Each user has an unique identifier, stored in the ID field. Other data stored for each user which holds importance to the project includes: a username, an email address and a hashed password.

The products table holds information for the models the users intend on selling, the fields are:

- ID: Incrementing number which uniquely identifies each product in the table
- Name: a display name for the product chosen by the user
- Description: a description of the product
- Price: a decimal number indicating the purchasing price of a given model
- Obj_file_path: Link to the product .obj file in storage
- Gltf_file_path: Link to the product .gltf file in storage
- Thumbnail_path: Link to the thumbnail image file in storage
- User_id: Foreign key which leads to the user that uploaded the product
- Created_at & Updated_at columns, which do not hold much significance for the project in its current state

The sales table holds information for each sale of products that was made, and its fields are:

- ID: Incrementing number which uniquely identifies each sale that was made on the app
- Buyer_id: User who purchased the product
- Proudct_id: The product that was sold
- Seller_id: User who owned the product at the time of selling. Note that although this field can be accessed by going through the product_id to the user_id, the decision to have this field was made for the following reason: If a requirement arises for this project where there is need to transfer product ownership, having the seller_id field here would make this process much easier. There are also other scenarios where this can be useful, such as if we want to allow the users to sell their products through another person or business.
- Price: The revenue which the seller has made from this specific sale. This field is also accessible by going to the product, although by having it here, it is possible to update the product price without affecting the already existing sale.

Other tables exist in this project, but understanding how they work is not essential to understanding this application, therefor they are not documented here. These include the tables generated by Sanctum for resetting passwords & managing user tokens, as well as the tables provided by Laravel for failed jobs, migrations and the sqlite_sequence table. We interact with these tables indirectly through code, not through accessing their fields directly, and as such I don't find it necessary for their purpose to be documented here. Explanations on how these tables work can be found in the Sanctum and Laravel documentations.

6 SERVER-SIDE

In this section the endpoints of the project will be explained, how they work and what their purpose is, as well as some other details of the back-end side of things. The reader should have basic understanding on how to interact with the server through a client app, as well as information on how to update the API after reading this section.

6.1 ELOQUENT

This project uses Eloquent to perform Database queries. This aspect is better explored by looking at the code and reading the Eloquent documentation.

6.2 AUTHENTICATION

This application uses Sanctum for providing authentication. Sanctum is featherweight and really easy to use alongside Laravel, and in my opinion the best choice for Laravel apps that do not intend to use OAuth2 authentication such as this one.

6.3 ENDPOINTS

All of the endpoints can be found in the api.php file, which is standard for any Laravel project. By looking at the code here it is possible to determine which controller is responsible for which endpoint, and as such the existing controllers in the project will not be documented individually, but by the endpoint.

Complete details on how these endpoints perform the explained functionalities and the type of data they return is not provided here as it is easier to visualize through looking at the code.

Unprotected endpoints: Endpoints which do not require an access token to interact with and are not guarded by Sanctum

6.3.1 Getting products endpoint

By sending a **GET request** to to the **api/products** endpoint, the user will get a paginated result of all of the products which currently exist in the project, ordered by id. This is implemented with the default **paginate(number)** method provided by Laravel, which makes accessing a specific page of the products easy and it also makes the code easy to maintain since we use only one line of code to get the results. Accessing a specific page is simple, we just need to pass the desired page in the URL as a parameter: **/api/products?page=2**. The endpoint will return all of the products on a specific page with the required data.

6.3.2 Getting product by ID endpoint (show)

Sending a **GET request** to the **api/products/{id}** will return only the product whose id corresponds to the one given in database. The **show** function in the product controller also has a different functionality if the user is authenticated, this is further documented below.

6.3.3 Getting a product by searching

Sending a **GET request** to **/products/search/{name}** will return all products which name includes part of the **name** string.

6.3.4 Getting all products by a specific user

It is possible to get all of the products by a given user if we know their id, this is done by sending a **GET request** to **api/products-by-user/{userId}.** In the background, this endpoint will run an Eloquent query which will return a paginated result of all products which were uploaded by a specific user. To get results from a second page we send a request to this endpoint in the same manner as described in the **Getting products endpoint**.

6.3.5 Registering a user

To create a new user, we send a **POST request** to **api/auth/register**. The request should contain the following fields:

- **Name**: The name which the user has chosen
- **Email:** The email for the user
- **Password:** A password for the user which will later be hashed
- **Password_confirmation:** A password confirmation field which ensures that the user typed in the same password twice

Sending a request to this endpoint will cause all of the fields to be validated, and if successful, the server will encrypt the password using the **bCrypt** function and return a 201 status code indicating that a new user has been created. The response will contain the details for the given user as well as a token which can be used for interacting with protected endpoints.

6.3.6 Login endpoint

Sending a **POST request** to **api/auth/login** with the fields:

- **Name**: An existing name in the database
- **Password**: The password corresponding to the given username

will cause the server to check the database for an existing user and if one is found, the server will check to see if the provided password is valid using the **Hash.check** function. If these checks are okay, the server will return a response containing the details for the given user and a token for accessing protected endpoints, same as the register endpoint.

Protected endpoints: These endpoints require a sanctum token provided in the request headers to access

6.3.7 Uploading a product

By sending a **POST request** to **api/products** we can add a new product. The body of the request should contain:

- **Name:** The name given to the new product by the user
- **Price:** The price for which the product will be listed in the marketplace
- **Description:** A description for the product
- **Thumbnail:** An image file which will represent the product
- **objModel:** A file which contains the geometry for the 3d model in .obj format
- **gltfModel:** A file which contains the geometry for the 3d model in .gltf format

Keep in mind that a product may have only an objModel or only a gltfModel, but cannot be uploaded by not providing either. When the request is validated, the server will store each of the files in an appropriate folder, create a symbolic link to the files it had just stored and save that symbolic file in database. As you can notice, there is a more lightweight approach here where we can just use a naming convention where we store the products with a name such as **{id}.obj** in our file system, however, this approach would lead to more difficulties down the road if the project is to be updated and does not answer many questions, one such being: what if in the future we want to allow a user to store more files of the same type for one product?

If this request is successful, the new product will be stored in database, the files will be stored in the file system and the returned response will be the details of the newly uploaded product.

6.3.8 Updating a product

A product can be updated by sending a **PUT request** to **api/products/{id}**. For now, this endpoint is not used by the UI and will not be further discussed, although if you browse the code you will see that the functionality is there.

6.3.9 Getting products by current user

Sending a **GET request** to **api/current-user-products** will return a list of products owned by the current authenticated user. This endpoint functions in the same way as the previously mentioned endpoint which returns a list of products for a given user. You might ask yourself: Why does this endpoint need to exist then? Consider this scenario: What if we are to upgrade the application in such a way that a user

may declare a specific product they own as **unlisted**. Although this can be achieved in one function in the products by user endpoint, having them as separate will help keep the code clean and maintainable.

6.3.10 Owned products endpoint

Sending a **GET request** to **api/owned-products** will return one page of all of the products which the currently authenticated user has purchased, this works the same as the other documented endpoints for getting products and discussing it further is not necessary.

6.3.11 Getting products for an authenticated user

Sending a **GET request** to **api/products-authenticated/{id}** will get data for a given product in the same manner as the **Getting product by ID endpoint**, with the difference that this "version" of the endpoint will also query the database to see if the user owns the product or has it purchased. This comes useful when creating the UI for the application.

6.3.12 Loging out a user

Sending a **POST request** to **api/auth/logout** will cause the token for the currently logged in user to be deleted in database. This endpoint is not really necessary for logging out, and logging out can be safely performed if the user deletes the token information for their local storage.

6.3.13 Buying products

By sending a **POST request** to **api/sales/buy** an authenticated user can purchase a list of products. The request should contain an array named **products**, and each member of the array should contain:

- **Product id**: the id of the product we are trying to purchase

One important aspect to mention here: If the business logic for the project is updated in a manner where the users can update the price of their products, each member of this array should also contain a **price** field. This is important, as having such a field would ensure that the pricing for each product has not updated between the time when the user has **seen** the prices for his transaction and the time the user has clicked the **Purchase** button.

The function for making a sale is transactional, and if one purchase fails the others will fail as well. If successful, details for the newly made sales will be provided to the user.

6.3.14 Getting a list of all sales

By sending a **GET request** to **api/sales**, an authenticated user is able to view details for all of the sales they have made so far.

7 FRONT END

By looking at the app description, you might have guessed that this project has more work on the client side than on the server side, and you'd be correct.

In this section, the used npm packages, libraries and everything else will be documented in the order they were added in the project.

7.1 REACT

The front end application uses the React library, again, the main reason for this is the fact that React has great support for the **three.js** library which we use for displaying three dimensional geometry to the user with the **React Three** package.

7.2 REACT REDUX & REDUX TOOLKIT

This project uses redux to maintain state of the application. The choice for using redux was simple: I want the state of the application to be maintained and updated separately from where my logic for displaying interface for the user is. Redux also comes with many other advantages, and further details of why this choice was made can be found in the articles linked in the sources section.

7.3 BOOTSTRAP

Although the bootstrap components might look outdated, the reason for using bootstrap is not how their components look, but how easy it is to create a responsive web application using it. Even if how the components look is to be entirely remodeled from scratch, the responsive layout can still be maintained since the html code for the project is written with the help of the Bootstrap grid system.

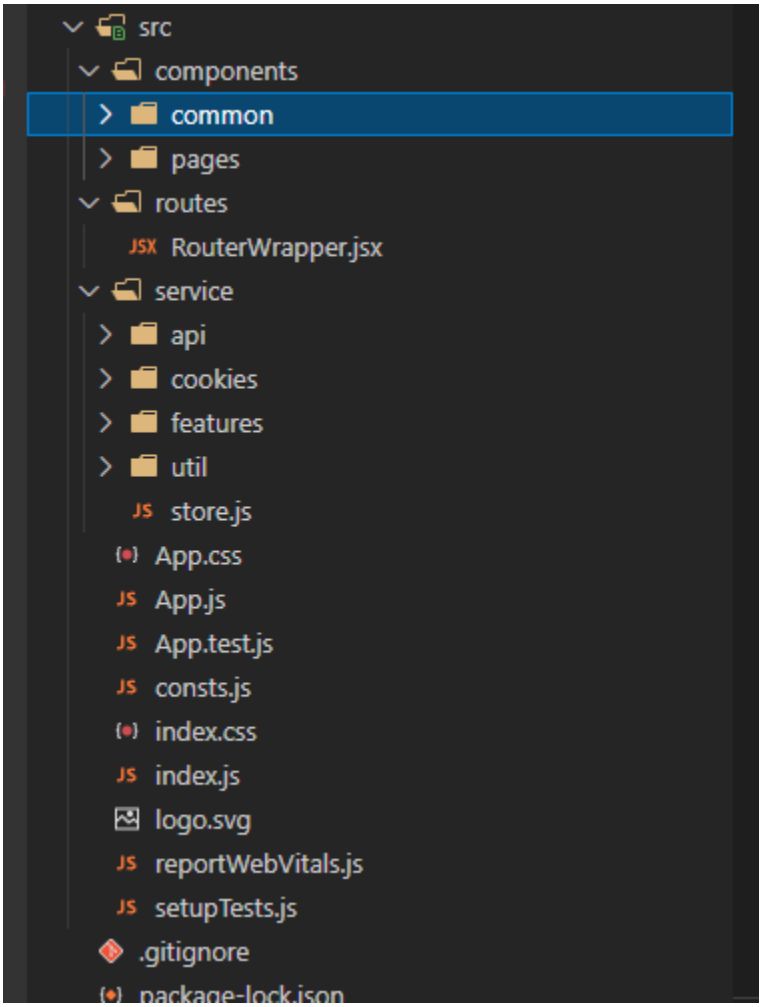
Besides, the components that Bootstraps provides do not look that bad.

7.4 OTHER PACKAGES NOTE

The other packages & other things used for this project will not be separately discussed as they do not hold as much importance for the project. Expect other packages of the project to be mentioned when discussing the appropriate place in the UI where they are being used.

7.5 CLIENT-SIDE PROJECT STRUCTURE

This React project structure can be seen on the picture below. Here, what the importance of each folder is will be discussed only briefly and more details will be provided as the documentation explores each aspect of the client-side project.



- Components: This folder holds every component that will be rendered on the page, and there are two subfolders: common – where we have information for reusable components which are then used by components in the pages folder – this is where each specific page is stored.
- Routes: This folder contains only a RouterWrapper which will be further discussed later.
- Service: This folder contains code that has no relation to rendering stuff on the web application, and in its subfolders we can find:
    - Api: This is a folder dedicated to functionalities which directly interact with the server.
    - Cookies: A folder dedicated to functionalities related to cookies
    - Fetures: A folder which holds information for the Redux reducers.
    - Util: A folder for utility functions which are called at more places in the code.

7.6 SERVER INTERACTION

We will start by exploring the **api** folder. You will notice that this folder contains two files, the **client.js** and **axios.js** files.

7.6.1 client.js

The client.js contains code which is practically "left-over" from a previous attempt to avoid using axios. During development, this has proven to be difficult, as axios really simplifies requests towards the server, especially when you consider that we need to add header. The code that depends on client.js has been refactored in all of the places in the application and this file can be safely deleted.

7.6.2 axiosClient.js

Axios is a promise-based http client, and it is a very good and time tested one compared to the client.js code that this project tried to use. The decision to use axios for requests towards the server is a no-brainer and is self-explanatory.

This file holds our axios instance. I will link articles discussing why using only one instance of axios in the project is a good idea. How you use the functionalities provided here is simple: you look at the API, you look at the type of request you need to send and then you look at one of the exported functions which suits your needs.

Each function then returns a promise which the project uses for rendering appropriate data for the user.

You will notice that the **get** function and **post** function also have a version that takes a **token** as a parameter. The reason for this is that during development, I ran into a lot of issues when trying to set a token for my axios instance, and all of the solutions provided online did not work. This led to the solution which you can see in code, which is not ideal as the code now contains more boilerplate, but was necessary as some of the axios functionalities were not working as expected.

7.7 THE REDUX STATE

If you open up the **index.js** file, you will notice that the entire app is wrapped in a **Provider** component. The app is further wrapped in a **PersistGate** component. These two components are necessary for redux and persisting the redux state, this aspect will become clearer by continuing to read the documentation.

The main configuration for the Redux store is in the **store.js** file. Here, in the **persistConfig** variable you can find all of the information that the state contains that should be persisted in the user's local storage. This aspect of the application uses the **react-persist npm package.**

The other configuration that is applied here is standard to how a redux toolkit configuration looks like, and can be better understood by reading the redux toolkit documentation.

7.7.1 Auth slice

The **authSlice.js** file contains information about the login and register state of the application. By accessing the data here, our rendered components are able to tell if the user is currently authenticated, or we can dispatch actions for registering, creating a new user or logging out the current user. In the

postLoginData method you can notice that we are also getting a **XSRF-TOKEN** as a cookie for the server. This cookie is needed by Sanctum and is necessary for communicating with the protected endpoints.

Also, this reducer holds information about the token of the currently logged in user. We use this token to access authenticated endpoints in other reducers.

7.7.2 Cart slice

The cart reducer's purpose is to track the products that the user has decided to purchased, and it contains a function which can map a list of products and tell the server that the user has decided to buy them.

7.7.3 Product slice

The product reducer contains data only about one product, this is always the product that the user has selected by navigating through the UI. Using the products reducer (documented below) is not enough, as we need to call an endpoint to see if the user can download the product they are currently viewing.

7.7.4 Products slice

This reducer holds data for the list of products we are displaying. You might notice that it has a lot of code, this is only due to the fact that here we dispatch requests to the API to different endpoints which list a different category of products (owned, purchased or all).

7.7.5 Product upload slice

This reducer simply manages the state of the upload page, which is used when a user is trying to upload a product.

7.7.6 Sales slice

This reducer is responsible for displaying the sales a user has made.

7.8 ADDITIONAL FILES

7.8.1 Cookies consent

Since this application **must** use cookies, we use the **react-cookie** npm package to handle this for us. This is a really useful package which will solve our UI problems related to asking our user if they consent to use cookies. This component is implemented in the **CookiesConsentWrapper** component, and is used in the **App.js** file. More information on how this can be modified can be found in the react-cookie docs.

7.8.2 The util folder

This folder contains only one file for now, which is the **fileToData.js**, a file which contains only one function which basically allows us to temporary store a file in the client's RAM memory. We need this because we are trying to display 3d models to the user before they decide to upload them, and reading this directly from the user's computer is not possible on modern web browsers as it is a security threat.

7.8.3 Routes wrapper

There is a **RoutesWrapper.jsx** file in the **routes** directory. Since this project, as most React projects uses the **React Router** package, having one place where all of the routes in the application are visible comes in really handy. All of the routes should be updated from **this place in the code only.**

7.9 COMMON COMPONENTS

7.9.1 CookiesConsentWrapper.jsx

This simply holds the UI that is to displayed if the user has not consented to using cookies, details on how this works and why it is here can be found in this document by scrolling above.

7.9.2 DownloadButton.jsx

A simple container for a download button. There is a high probability that this component might need to be upgraded if this project reaches production, therefor it is contained separately.

7.9.3 Header component

In the **Header** directory where the common components are, you can see the implementation for the header. We display two separate versions of the header, one for logged in users and one for unauthenticated users. The logic for these two separate scenarios is contained in two separate components.

A logged in user is able to access the **Upload Page, My Uploads Page, My Purchases page, My Sales Page,** as well as the **log out** functionality from the header.

A user that is not logged in can access the **Login** and **Register** pages.

There is a link to navigate to the index page by clicking the title of the header.

7.9.4 Model Displayer

The ModelDisplayer folder contains information for two of the most vital components in this application: Which are the **GltfModelDisplayer.jsx** & **ObjModelDisplayer.jsx** components. Each of these two components renders the appropriate file type, and in the **props** field they can be provided with a **fileUrl**, which tells the loaders where they should look for the geometry they are trying to render and a **isLocalFile** variable, which tells the components if they should refer to the **server** when fetching a file.

You will notice that these two files almost work in the exact same manner, and that they both use the **react-three/drei** library in order to simplify the process of creating a scene and rendering geometry. This library also provides as with an **OrbitControls** component, which allows the user to view the model from all angles without additional code required from the developer side.

7.9.5 Product Overview

This component is used for displaying one component in a paginated grid which can be provided in the props field.

7.9.6 Product Listing Grid

This is a component can take a list of products and display them in a responsive grid. It depends on an npm package called **react-paginate.** This package displays a simple pagination based on parameters we provide, and is used for rendering multiple pages if it is required.

7.9.7 Shopping cart

This component is responsible for displaying the shopping cart menu, it uses the cart reducer and shows UI for currently owned products. It also supports the feature of clearing the cart.

7.9.8 Spinner

This is a loading spinner which is used everywhere in the application where we are loading data without knowing how long it will take. It is wrapped in a separate component since it allows changing the design with ease and without affecting any of the other code.

7.10 PAGES

7.10.1 Checkout page

This page will display all of the products the user has added to their cart, and from here they can decide to pay and check out. If they do so, the products will be added to their purchased products list.

7.10.2 Current user products list

This page uses the Product Listing Grid to display the products the current user has uploaded.

7.10.3 Landing page

A simple container for the landing page.

7.10.4 Login & Register pages

Pages where the user can interact with the appropriate login and register endpoints.

7.10.5 Product upload page

A page where the user can upload and preview .gltf, .obj files and a product thumbnail. Interacts with the upload product endpoint on the API and uses the uploadProduct reducer.

7.10.6 Products View

Contains ModelLoaderErrorFallback, a page that should be displayed if a model cannot be loader. Note: This should be moved in the common folder.

Contains AddToCart, a responsive button which the user can interact with if they purchase a product. This button communicates with the single product reducer and displays different interface depending if the user owns the product, has it purchased or does not own the product.

Contains SingleProductView.jsx, a component which reads the information of the current product and uses a neat styling trick to display a responsive box of the product. If the product contains both a .obj and .gltf file, this component will allow the user to view them separately.

7.10.7 Products list

You can find the **ModelListPage.jsx** in this folder, a component responsible for listing all of the products available in store.

7.10.8 Purchased products page

This page will display only the products the user has purchased.

7.10.9 Sales page

This page will display all of the products the user has purchased.

8 PRODUCT USAGE

Start the client-side by running **npm start.**

Start the api by running **php artisan serve.**

8.1 APPLICATION FLOW:

When first accessing the site, the site should be in the background and the user should not be able to interact with anything until they consent to using cookies by clicking the **Accept Cookies button.**

Once they have consented to using cookies, the user should be taken to the landing page, which is simple for now.

By clicking the Browse Products button, the user should get a list of the products currently on the marketplace. All of the products which cannot load a thumbnail should look like follows:

Each page of products should show the thumbnails of 16 different products alongside their name and their price, and on the bottom of the page the navigation should be shown.

By clicking on a product, the user should be able to view the 3d model and view the details.

The user should be able to toggle between the .gltf and .obj preview of the model if they exist. The user should also be able to add the model to cart, however, he will not be able to check out unless they log in.

The user can choose to register by navigating to the Register page through the header, and through here they can create a new account.

Upon registering, the user should automatically be logged in and the Header should update. The user can now see the products in their shopping cart added from before.

By navigating to the checkout page and proceeding to payment, the user should get the products in their collection.

Now, by navigating to My purchases from the header, the user can see their purchased product.

Download buttons should appear for the appropriate file types, since the user now owns the product.

The user can now try to upload a product. After filling the form with the appropriate data and clicking Upload, the user should now be redirected to the newly uploaded product.

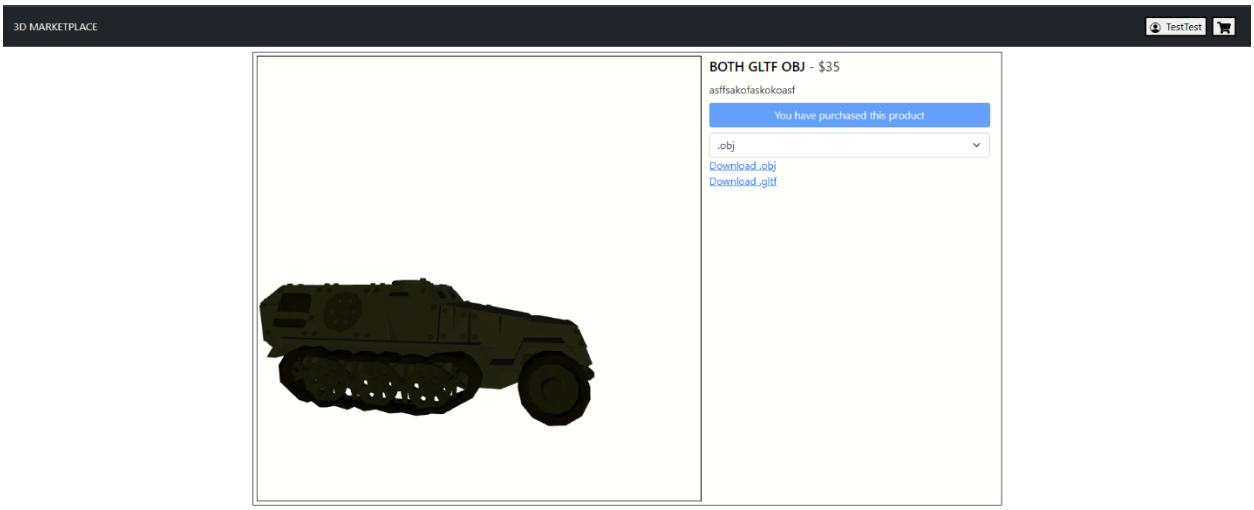
The user will not be able to add this product to the cart since they own it, but the download buttons should appear.

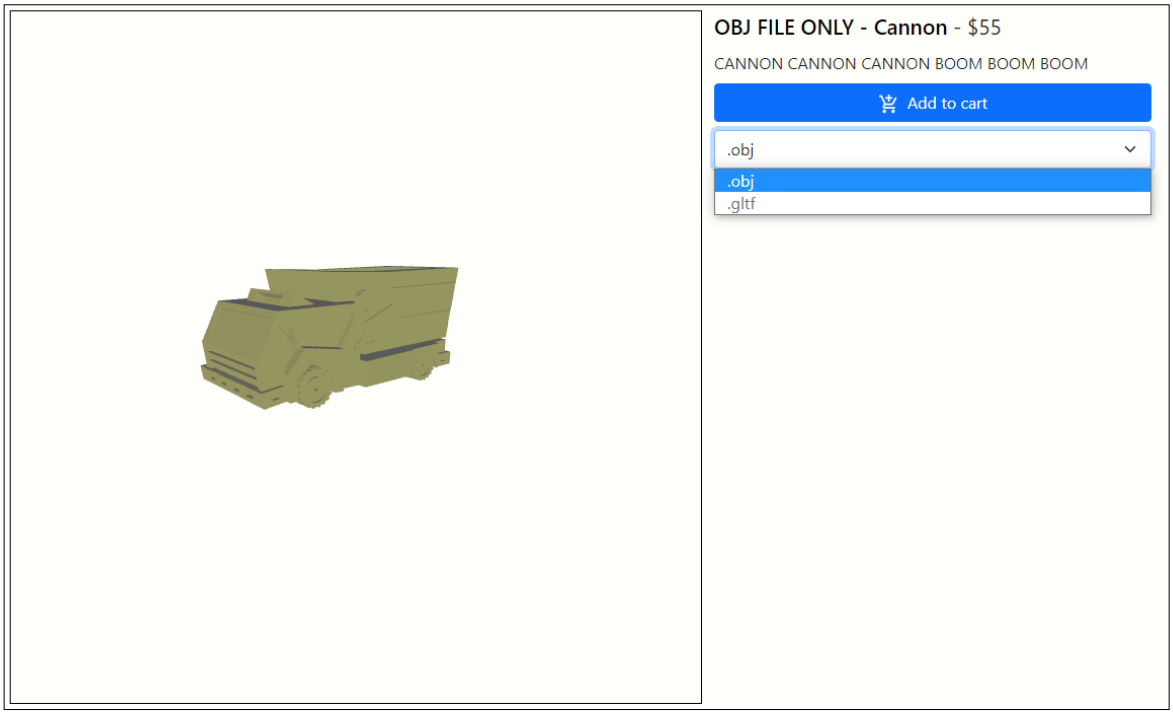
The user should now test the log out functionality.

Another user should log in and buy the uploaded product and log out.

The previous user should once again log in.

If the now logged in user accesses the Sales page from the header, they should see that they have made a sale.





3D MARKETPLACE

TestTest

Total: $55

Checkout

In cart:

$55 - OBJ FILE ONLY - Cannon

× Clear cart

Thumbnail not availbale

x

OBJ FILE ONLY - Cannon

$55

Proceed to payment

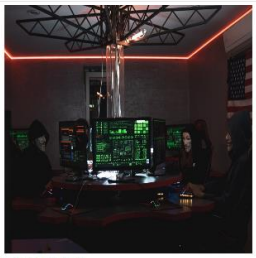
3D MARKETPLACE

OBJ FILE ONLY - Cannon

$55

ONLY GLTF

$24



BOTH GLTF OBJ

$35

9 USED SOURCES

https://laravel.com/docs/9.x - The Laravel documentation which is absolutely necessary for understanding and modifying this project.

https://laravel.com/docs/9.x/sanctum - The documentation for Sanctum, which this project uses for authentication.

https://laravel.com/docs/9.x/filesystem - The documentation for the Laravel filesystem which this project heavily relies on.

https://laravel.com/docs/9.x/eloquent - The documentation for Eloquent which this project uses for its database querying.

https://getbootstrap.com/docs/5.0/getting-started/introduction/ - Link to the Bootstrap documentation which this project uses for displaying responsive pages, as well as in some components.

https://react-bootstrap.github.io/ - React Bootstrap, which just documents how to implement Bootstrap into React.

https://redux-toolkit.js.org/introduction/getting-started - Information about React Redux toolkit, which this project uses for maintaining state.

https://github.com/rt2zz/redux-persist - Information about Redux Persist, which this project uses to persist redux state between sessions.

https://github.com/pmndrs/react-three-fiber - React Three Fiber, which this project uses for displaying three-dimensional geometry in the web browser.

https://github.com/pmndrs/drei - React Three Drei, a helper library which this project uses to make working with React Three easier.

https://axios-http.com/docs/intro - Axios, which this project uses for making requests.

https://github.com/szhsin/react-menu - React Menu, which this project uses for displaying menus.

https://www.npmjs.com/package/react-cookie-consent - React Cookies Consent, which this projto ask the user for their permission on using cookies.

https://github.com/axios/axios/issues/2474 - Explanation on why using an Axios Instance may be a good idea in larger projects.

https://chrome.google.com/webstore/detail/cors-unblock/lfhmikememgdcahcdlaciloancbhjino - An extension which is useful for testing when problems from CORS arise.

https://www.jetbrains.com/phpstorm/ - Link to the phpStorm website.

https://code.visualstudio.com/ - Link to the Visual Studio Code website.

https://www.postman.com/ - Link to the Postman application.

https://docs.pmnd.rs/react-three-fiber/tutorials/loading-models - How to load models with React Three.