

# TIN SEMESTER PROJECT

## ETH Juventus Ecommerce

<b>Introduction</b>	<b>2</b>
<b>User &amp; Admin</b>	<b>2</b>
<b>Database</b>	<b>3</b>
<b>Authentication</b>	<b>4</b>
<b>Backend</b>	<b>6</b>
<b>Frontend</b>	<b>6</b>
API	6
Login	7
Registration	7
Logout	8
Admin panel	8
Shop	9
Cart	9
Checkout	10
Payment	10
Payment Method	11
Feedback	11
<b>Summary</b>	<b>11</b>
<b>To implement</b>	<b>11</b>

# Introduction

The application names 'Juventus Shop' is a small Ecommerce application which accepts ethereum payments and bank transfer, the last payment method has not been fully implemented but it could be easily developed by using external services such as Stripe.

The use cases implemented are the basic ones such as:

- User registration
- User login
- Roles definition
- Authentication with access token and refresh token
- Admin panel with basic functionalities such as displaying,adding,updating and deleting rows from tables product,size,stock,orders
- Assigning a cart to the user
- Add items to cart
- Delete items from cart
- Create a new payment method ( bank name,iban,swift,cvv)
- 

## User & Admin

Users can be created using tools such as postman or directly from the frontend in the /register route and then redirected to the /login route.

Admin can only be created bypassing the user interface and modify the parameter 'roles' into: ['user','admin']. I am aware that this solution should be change into production but I don't see any other solution except updating the database directly.

POST ▼ http://localhost:8080/api/user/register

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings

● none ● form-data ● x-www-form-urlencoded ● raw ● binary ● GraphQL JS

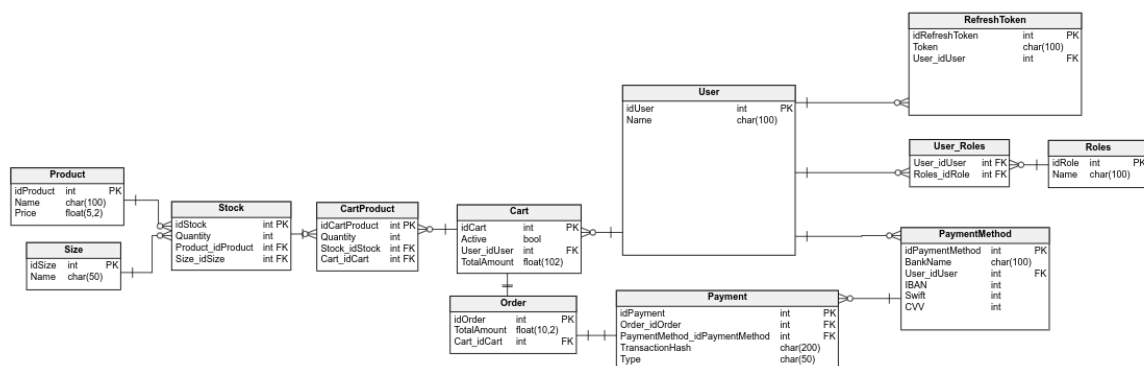
```

1  {
2    "firstname": "admin",
3    "surname": "admin",
4    "email": "admin@gmail.com",
5    "birthDate": "1998-03-17",
6    "password": "Pesca123",
7    "roles": ["user", "admin"]
8  }
9

```

## Database

The database is a localhost instance of MYSQL and it's connected to the backend via Sequelize which allowed me to easily create tables, attributes, associations and queries. The below schema represents the database:



**Stock** table keeps track of the available quantity of **product/size**.

**CartProduct** and **Cart** tables could be replaced by **localStorage** variables but I decided to create dedicated tables in order to allow the product availability management:

when a stock item is added to a cart, the corresponding quantity is deducted from the stock item quantity.

The cart table belongs to many users, why? Here, I tried two different scenario:

1. One to one: It would be correct but I wanted to link the cart with the corresponding order with a one to one association in order to keep track of the purchased items  
(Now that I'm writing this, I realized that I could use a 'OrderProducts' table to do the same, but it persists the problem of managing the available quantity against the items in the cart, what would be the correct approach?)
2. One to many: the user has many carts but only one active, this means that as soon as he logs into the application, we look for any cart which is still open because it means that it was not paid.

**User** can have several payment methods, roles and carts ( only one should be active ).

**Order** must have a cart and a payment, which indicated that a payment was correctly registered.

**Payment** has one **paymentMethod** but the paymentMethod can be assigned to several payments.

Within the payment table I added at the last minute two additional columns 'Type' and 'TransactionHash' to store the payments performed with **ETH ( not the most correct approach )**

**User** can have several roles and a role can have several users so that's why we have a many to many associations between users and roles.

**RefreshToken** stores all the refresh tokens with the corresponding expiration date.

## Authentication

Authentication is implemented using an access Token and a Refresh Token.

### How it works:

- **Registration:** Read data, check if email is already stored, encrypt the password and store the user in the database
- **Login:** Read email and password, decrypt the password, compare the values with the ones in our database, generate an access token and a refresh token, get roles from database, save the refresh token in the corresponding table and return all data to the frontend so that they can be stored in the local storage.

```
if (user && (await bcrypt.compare(password, user.password))) {  
  // Create token  
  const token = jwt.sign({ id: user._id }, config.secret, {  
    expiresIn: config.jwtExpiration,  
  });  
  
  let refreshToken = await RefreshToken.createToken(user);  
  var roles = [];  
  for (var i = 0; i < user.roles.length; i++) {  
    roles.push(user.roles[i].name.toUpperCase());  
  }  
  await User.update(  
    { token: token, refreshToken: refreshToken },  
    { where: { id: user.id } }  
  );  
}
```

- **Frontend:** React uses axios interceptors to intercept each single request and response.
  - When sending using those interceptors, the stored access token is used as 'x-access-token'

```
instance.interceptors.request.use(
  (config) => {
    const token = ApiService.getLocalAccessToken();
    if (token) {
      config.headers["x-access-token"] = token;
    }
    return config;
  },
  (error) => {
    return Promise.reject(error);
  }
);
```

- If the endpoint requires the user to be logged in, the corresponding middlewares check whatever the token is still valid or not

```
router.post("/", [authJwt.verifyToken], cartItem.create);
```

- If valid, it goes directly to the passed (cartItem.create) function and the flow ends
- If not valid, it returns 401
- The frontend interceptor receives the response
- It checks if the response is 401 and it tries to generate a refresh token by sending a request to endpoint /user/refreshToken

```
if (err.response.status === 401 && !originalConfig._retry) {
  originalConfig._retry = true;
  const rs = await instance.post("/api/user/refreshToken", {
    refreshToken: ApiService.getLocalRefreshToken(),
  });

  const { accessToken } = rs.data;
  ApiService.updateLocalAccessToken(accessToken);
  return instance(originalConfig);
}
```

- The backend function 'generateRefreshToken' verifies if the received refresh token is still valid, if so, it returns a new access token which is then used by the interceptor to perform again the initial call but with the new access token

```
if (RefreshToken.verifyExpiration(refreshToken)) {
  console.log("Deleting refresh token");
  RefreshToken.destroy({ where: { id: refreshToken.id } });
  res.status(403).json({
    message: "Refresh token was expired. Please make a new signin request",
  });
}

const user = await db.user.findByPk(refreshToken.userId);

let newAccessToken = jwt.sign({ id: user.id }, config.secret, {
  expiresIn: config.jwtExpiration,
});
```

```
console.log("newAccessToken created ", newAccessToken);

user.token = newAccessToken;
await user.save();
return res.status(200).json({
  accessToken: newAccessToken,
  refreshToken: refreshToken.token,
});
```

## Backend

The backend is implemented in using node and the folder structure is divided into:

- **models:** database tables definition
- **controllers:**

Controllers are in charge of performing all the tasks requested via api.

When needed, I used the sequelize transaction for perform a sequence of functions in order to be able to rollback when an error occurs.

In some cases there are a lot of insert/update/select queries ( adding a stock item into the cart, creating an order ) and it results to be a very long function that could be refactored and divided into several functions.

- **routes**

Route are restricted to users or admins thanks to the help of middlewares that verifies whatever the user is authenticated or if the user is an admin

- **middlewares**

Helper functions that make the entire authentication process possible and easy to utilized.

They're in charge of verifying tokens and user rights.

## Frontend

### API

The 'services' folder contains all the files and functions use while making api calls.

The '**api**' file defined the interceptors as explained in the 'Authentication' section.

The '**apiService**' defines all the POST/GET/PUT/DELETE methods which are used in the application, the simply return the api instance (with interceptors) and the required endpoint

## Login

### Login form

Email

Password

Sing in

Need an account? [Sign up!](#)

## Registration


### Registration form

Firstname

Surname

Email

Birthdate



Password

Repeat Password

Singup

Already have an account? [Sign in!](#)

# Logout

Component which simply logs out the user and redirects to the home page

```
import React from "react";
import { Navigate } from "react-router-dom";

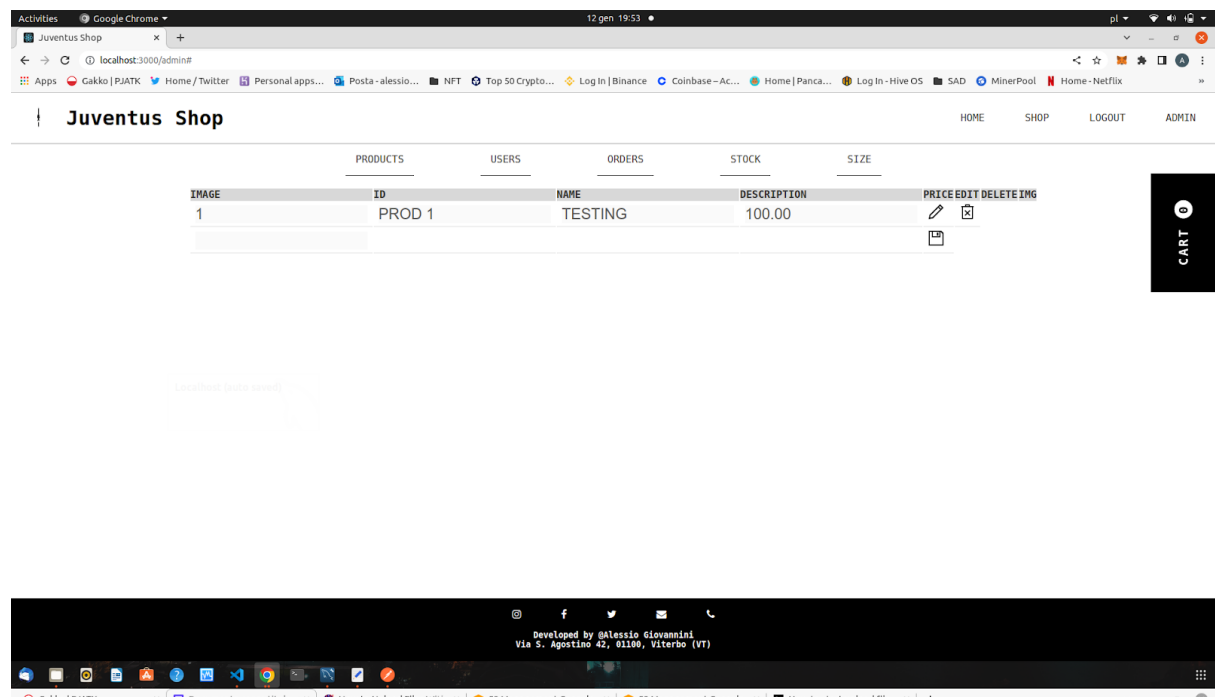
function Logout(props) {
  props.logout();
  return <Navigate to="/" />;
}

export default Logout;
```

# Admin panel

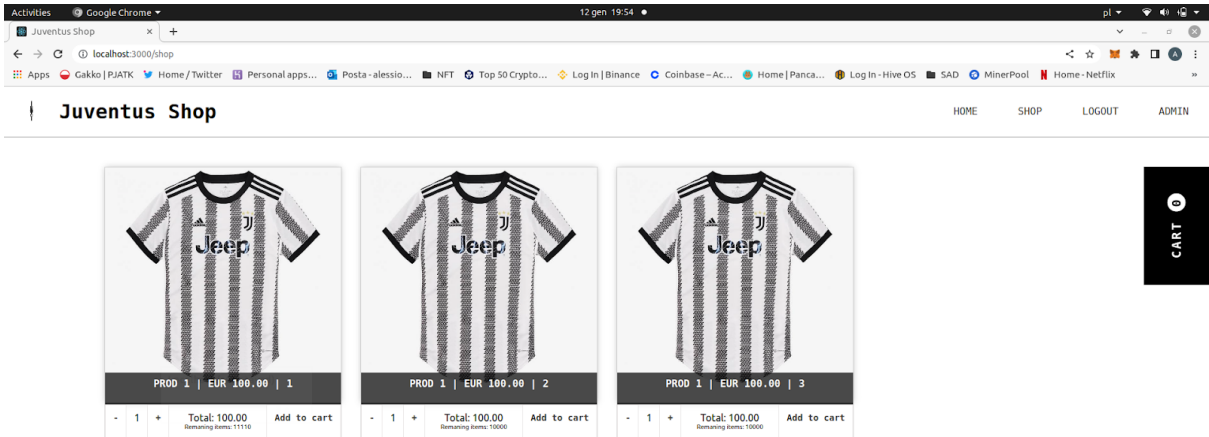
You can add,update,delete rows.

Every action is followed by a feedback.

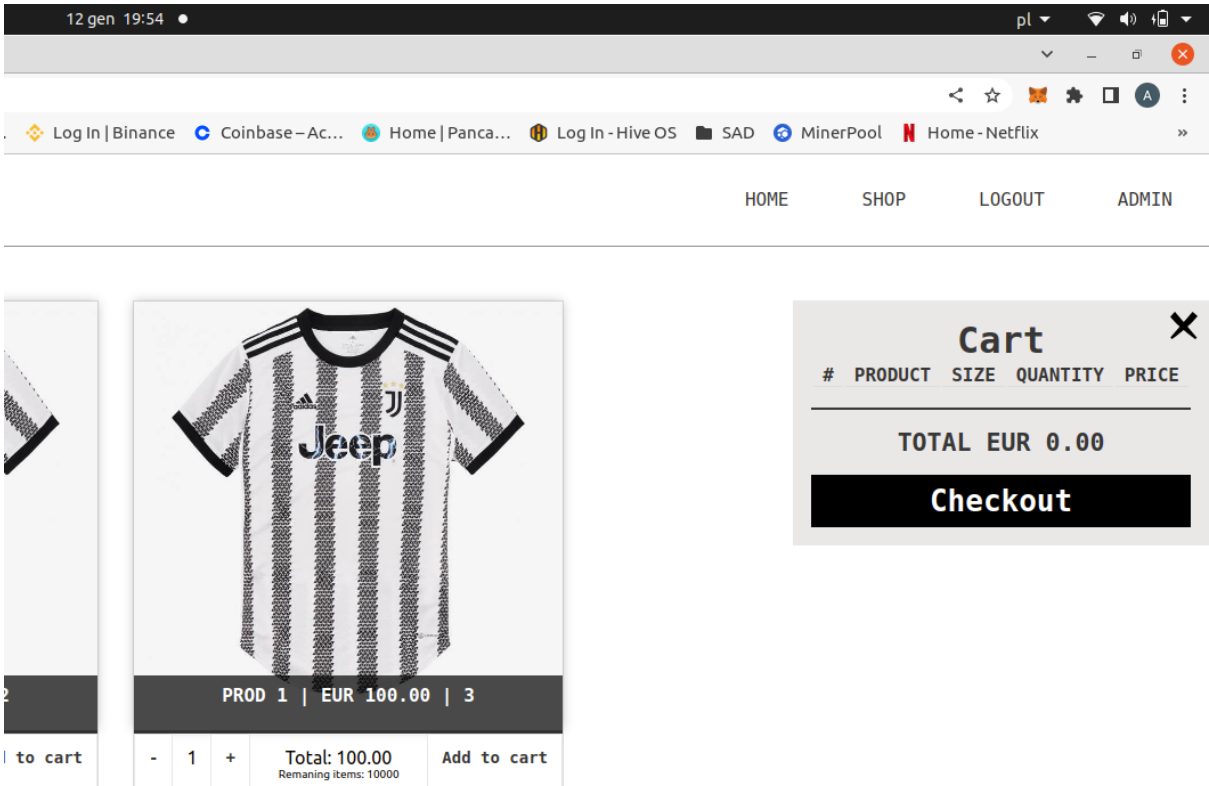




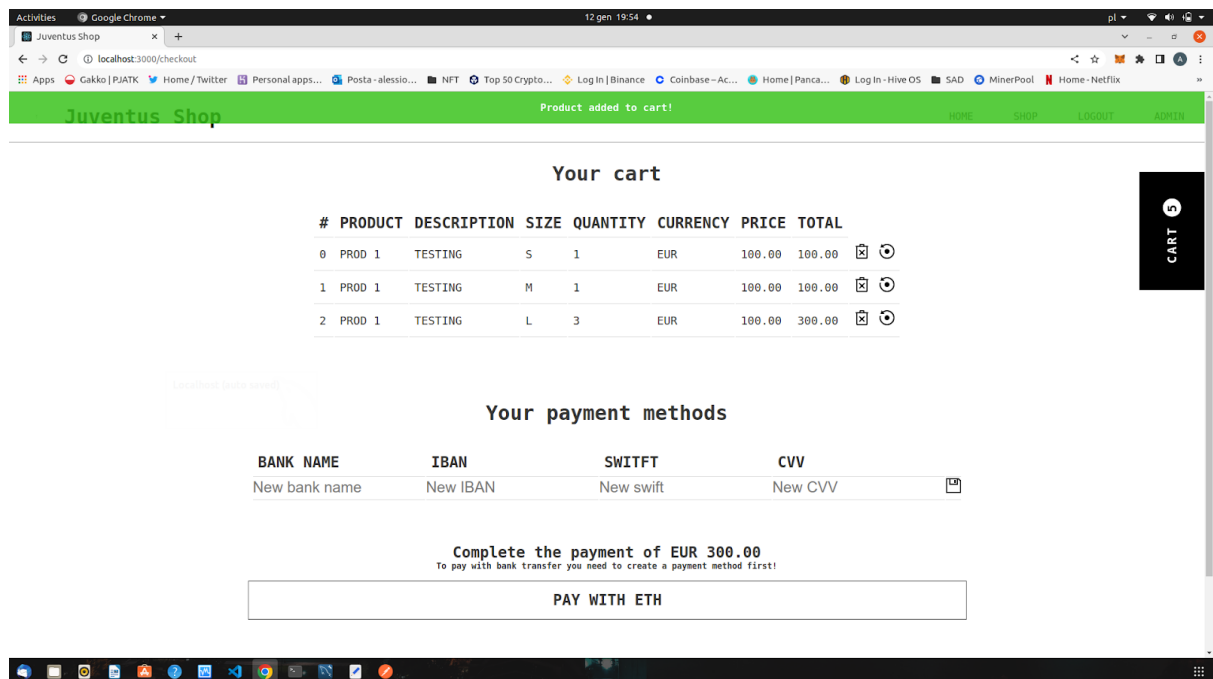
# Shop



# Cart



# Checkout



## Payment

Ethereum payments are allowed thanks to a smart contract written in solidity:

<https://goerli.etherscan.io/address/0xf9C4870f3F59e3dD99Ce0184A788DF8a9903c15f>

The contract accepts any amount for the moment and there is no check when receiving the sendEth request.

In order to test the payment with eth you need to install the Metamask extension, create a wallet and get some Goerli ETH <https://goerlifaucet.com/>.

For the purpose of the test, the part of sending the request to the smart contract is commented, in production the below code can be uncommented:

To charge the real price in eth the parseEther argument must be replaced with the const ethToPay.

file: EthPayment.js lines 44...53.

```
//UNCOMMENT AND CHANGE ETH PRICE IN PROD
//    let nftTxt = await nftContract.sendEth({
//      value: ethers.utils.parseEther("0.001").toString(),
//    });
//    await nftTxt.wait();
//UNCOMMENT IN PROD
//    let transaction = nftTxt["hash"];
// COMMENT IN PROD
```

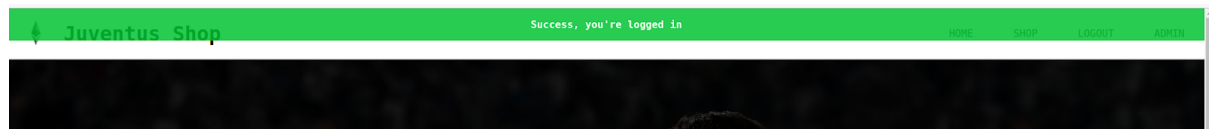
Bank transfer payment is not implemented, the api creates a payment and an order without really charging the credit card ( only for testing).

## Payment Method

Payments methods are stored in the database for the project implementation, they should probably be handled by third parties services or at least encrypted as passwords. They can be edited in the checkout form by the user directly.

## Feedback

Feedback are using a context at the application level to globally change state, they close only when clicked.



## Summary

The application works as expected, I am fully aware aware that some items have not been implemented but for luck of resources I had to choose the most appropriate solution in order to submit on time the project source code.

The idea of the ethereum payment comes from an old NFT which I implemented and the smart contract was used to receive ethers and send back a specific token generated by the super classes ( predefined 'contracts') such as ERC721.

For the purpose of this project I only created few lines of cose that allow to receive a payment and store it in the contract balance.

## To implement

There are some important functionalities and components which I could not developer for luck of resources:

- Update user data from profile
- Add images to products
- Group stocks by products and use the available sizes as <select> element
- Responsive design