Full-Stack Javascript Developer Test

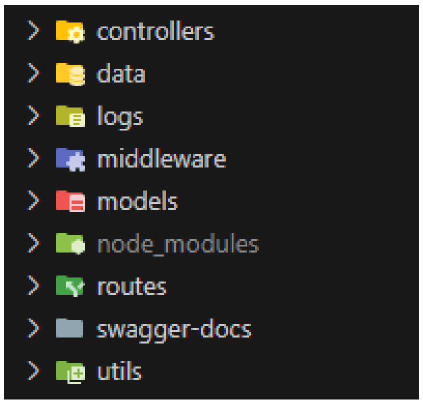
This technical test project showcases a full-stack application built with a Node.js backend and an Angular frontend. The backend, developed using Express.js, includes RESTful APIs with comprehensive unit tests written in Jest to ensure reliability, Winston.js for efficient logging, and Swagger documentation for clear API reference. The Angular frontend, styled with Tailwind CSS and Angular Material, provides a responsive and user-friendly interface, featuring reusable components and animations for enhanced user experience.

Project Links

- Backend Repository: https://github.com/duvan-23/backGamesK
- Frontend Repository: https://github.com/duvan-23/frontGamesK
- API Documentation (Swagger): https://backgamesk-production.up.railway.app/api-docs/
- Deployed Application: https://duvancasino.netlify.app/

Backend

The backend of this project is structured using the MVC (Model-View-Controller) design pattern to ensure maintainability and scalability. The Model handles the data logic. The View (Router) is represented by API responses, serving as the interface between the server and the client. The Controller acts as an intermediary, processing incoming requests, invoking the necessary model logic, and returning appropriate responses. This clear separation of concerns allows for easier debugging, testing, and future enhancements. The project also includes middleware for handling authentication, request validation, and logging.



```
controllers
data
logs
middleware
models
node_modules
routes
swagger-docs
utils
```

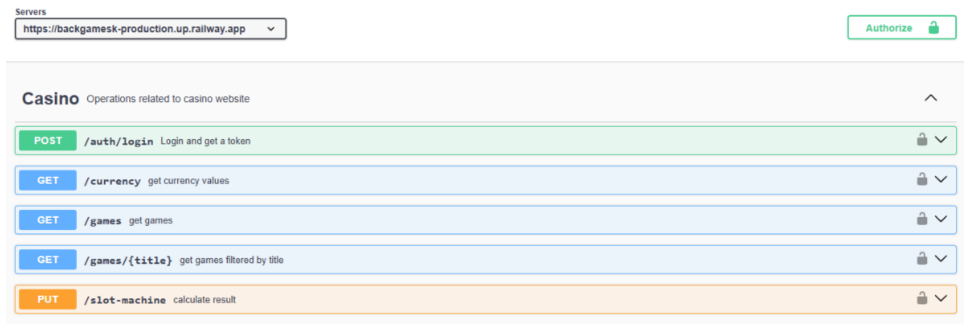
The server for this project is built using Express.js, with several key configurations:

- Swagger is integrated for API documentation, accessible at /api-docs for easy testing and reference.
- CORS middleware is used to allow requests from different origins, enabling frontend-backend integration.
- JSON parsing middleware handles incoming JSON payloads.
- Authentication routes are defined for login and user management (/auth).
- JWT verification middleware secures routes by ensuring only authorized users can access protected resources.
- Main API routes are set up to handle business logic and user requests.
- Fallback handling returns a 404 for undefined routes.
- The server is started with app.listen, and logs are generated to confirm the server's status.

This configuration ensures the server is well-organized, secure, and developer-friendly, with robust documentation and essential middleware for production-ready applications. Additionally, Winston.js is used for logging server activities and errors, while express-validator ensures the validation of incoming request bodies, enhancing the security and reliability of the backend.

The Jest tests for this project ensure the proper functionality and security of various routes. The tests validate token authentication, ensuring that requests without a token, or with an invalid token, are appropriately rejected with a 401 or 403 status. Successful authentication with a valid token results in a 200 status and appropriate data. The login endpoint is also tested for both valid and invalid credentials, checking for correct status codes (200 for success and 401 for failure). Additionally, the /games endpoint is tested for returning game data with the correct structure, and the /slot-machine endpoint is tested for calculating coin winnings based on different game results, ensuring the correct response for valid and invalid requests.

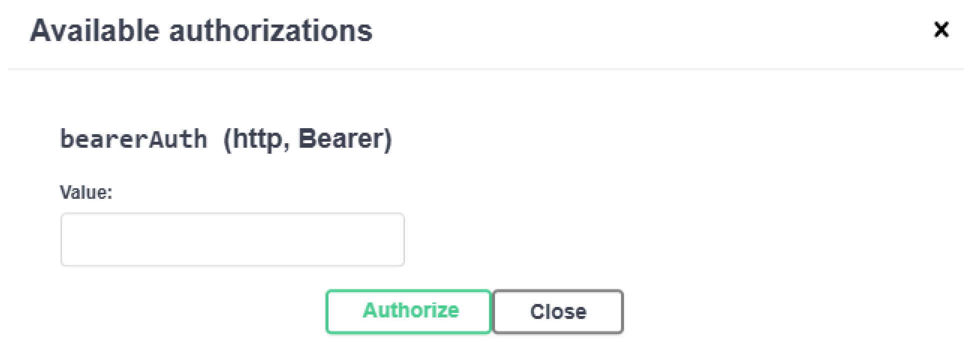
The Swagger documentation for this project includes a POST /auth/login endpoint that allows users to authenticate using a default username and password. This is useful for testing purposes. Once you successfully authenticate, you'll receive a token in the response. You can then set this token in the "Authorize" section of the Swagger UI. After authorization, you will be able to use the other protected routes, as they require a valid token to access. This setup ensures a seamless testing experience for users exploring the API endpoints.



Servers
https://backgamesk-production.up.railway.app
Authorize

Casino Operations related to casino website

POST /auth/login Login and get a token
GET /currency get currency values
GET /games get games
GET /games/{title} get games filtered by title
PUT /slot-machine calculate result



Available authorizations

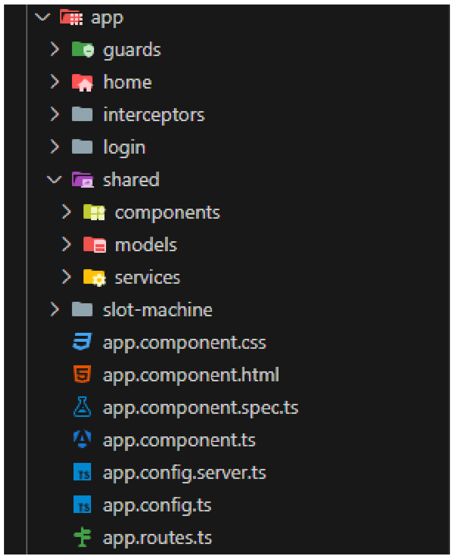
bearerAuth (http, Bearer)

Value:

Authorize
Close

Frontend

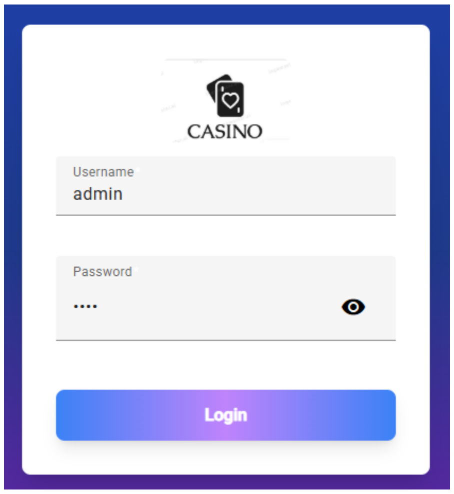
The frontend of this project is built with Angular, utilizing a clean and structured architecture. It features guards for authentication, ensuring that users are redirected to the login page if they don’t have a valid token. The project also includes interceptors that automatically attach the token to HTTP requests, allowing seamless authentication with the backend. Services are created to interact with the APIs via HTTP calls, and models and interfaces are used throughout the project to ensure the use of typed data. The project is styled with Tailwind CSS and Angular Material, providing a responsive and modern user interface. It follows a modular structure, where each page has its own folder containing /components, /models, /pages, and /services to keep the codebase organized, legible, and maintainable. Additionally, there is a /shared folder that contains reusable components, models, and services that are used across multiple pages. The login interface uses a default username and password for authentication, and once logged in, users can access other routes protected by the authentication guard. The token is saved in cookies and is automatically included in the HTTP requests through the interceptor.



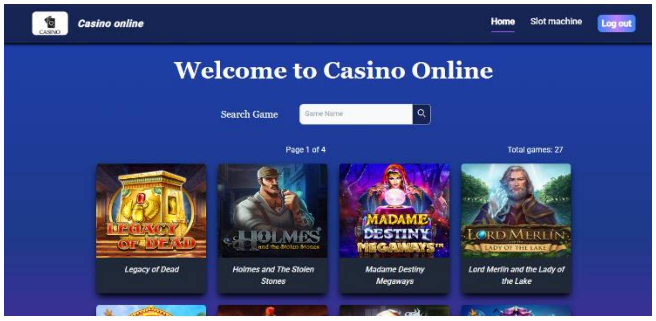
```
app
  guards
  home
  interceptors
  login
  shared
    components
    models
    services
  slot-machine
  app.component.css
  app.component.html
  app.component.spec.ts
  app.component.ts
  app.config.server.ts
  app.config.ts
  app.routes.ts
```

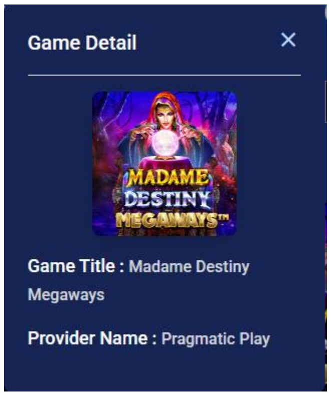
Pages:

Login page contains a form for user authentication. If the user provides incorrect credentials, an alert is displayed informing them of the error.



Home page features a grid displaying cards for various games. Each card includes a game picture, and clicking on the picture opens a dialog with more details about the game. The grid supports pagination, and there is a search input that filters games by title. the home page include a navigation bar that allows users to navigate between pages or log out of the application.

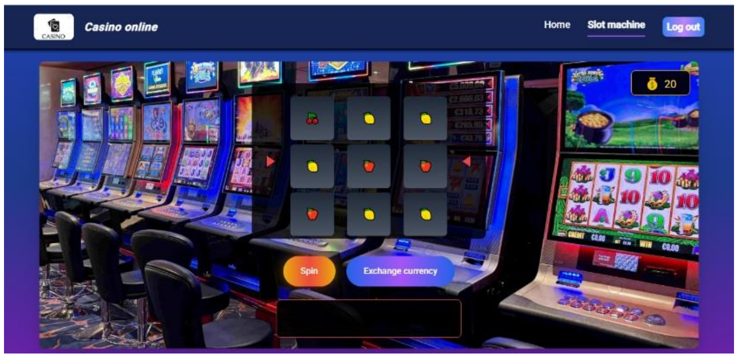


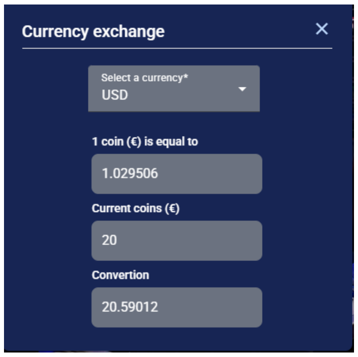


Game Title : Madame Destiny Megaways

Provider Name : Pragmatic Play

Slot Machine page lets users play a slot game. It includes a "Spin" button that activates three moving reels, with results determining whether the user wins coins. The current coin balance is displayed in the top-right corner, and the coin count is saved in cookies to persist across pages. Logging out resets the coin balance to the default value of 20. There's an "Exchange Currency" button that opens a dialog where users can choose a currency and see how much 1 euro is worth in that currency, along with their current coin balance and the corresponding total value in the selected currency. Winning a spin increases the coin balance, with an animation displayed when the user wins. Each spin costs 1 coin. If the coin balance reaches 0, the buttons are disabled. A message box below the buttons displays information on wins, losses, and the amount of coins gained or lost. The page also includes a navigation bar to help users navigate between pages or log out.




Currency exchange

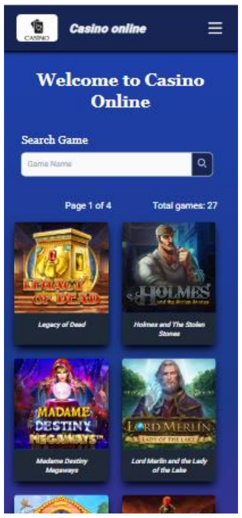
Select a currency*
USD

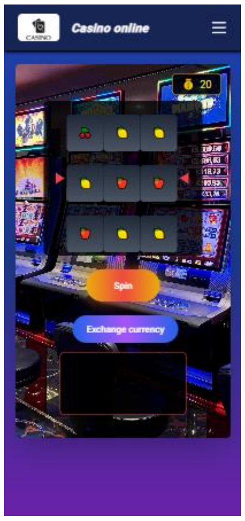
1 coin (€) is equal to
1.029506

Current coins (€)
20

Convertion
20.59012

For mobile devices, the design adapts to ensure a seamless user experience. The navigation bar transforms into a button that reveals a vertical menu, providing easy access to navigation options or logout. On the Home page, the game grid shifts from a 4-column layout to a 2-column format for better readability and usability on smaller screens. Similarly, the Slot Machine page adjusts its buttons from a horizontal row to a vertical column layout. Additionally, all styles are dynamically adjusted based on screen size using responsive design principles with Tailwind CSS, ensuring that the interface remains user-friendly and visually appealing across various devices.

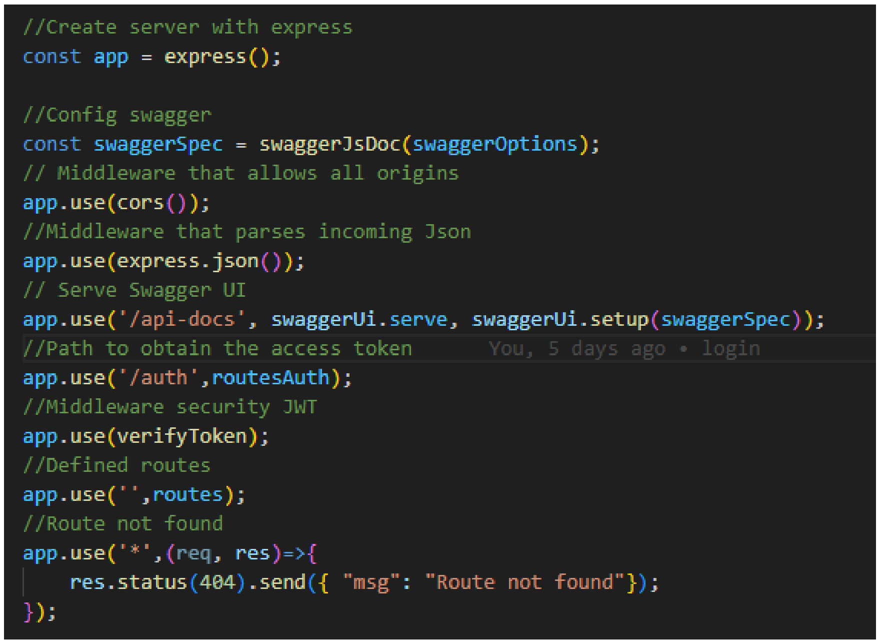




Important points

- Response to 2.1.4. Question 4:

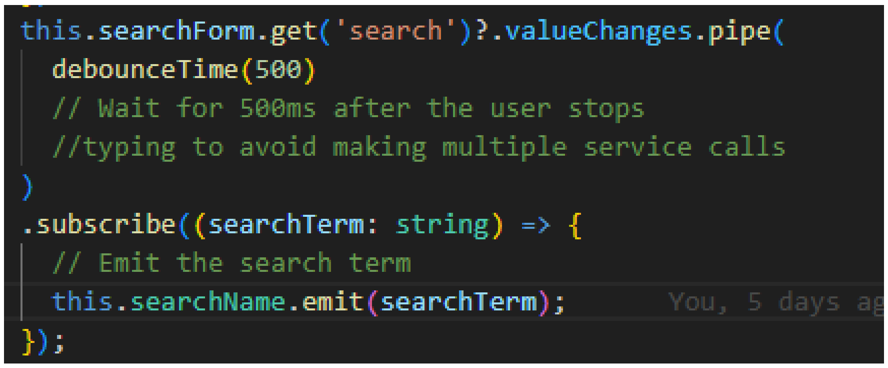
  Middlewares used: Cors, Express.json, JWT verification, Swagger and Express-validator



```
//Create server with express
const app = express();

//Config swagger
const swaggerSpec = swaggerJsDoc(swaggerOptions);
// Middleware that allows all origins
app.use(cors());
//Middleware that parses incoming Json
app.use(express.json());
// Serve Swagger UI
app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(swaggerSpec));
//Path to obtain the access token       You, 5 days ago • login
app.use('/auth',routesAuth);
//Middleware security JWT
app.use(verifyToken);
//Defined routes
app.use('',routes);
//Route not found
app.use('*',(req, res)=>{
    res.status(404).send({ "msg": "Route not found"});
});
```

- Response to 2.1.5. Question 5:

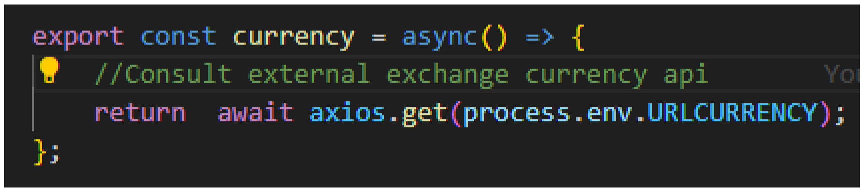


```
this.searchForm.get('search')?.valueChanges.pipe(
  debounceTime(500)
  // Wait for 500ms after the user stops
  //typing to avoid making multiple service calls
)
.subscribe((searchTerm: string) => {
  // Emit the search term
  this.searchName.emit(searchTerm);       You, 5 days ag
});
```

In Angular, debounceTime is an operator from the RxJS library used to delay the emission of values from an observable by a specified amount of time. When applied to an input field, it helps prevent unnecessary HTTP requests or actions from being triggered every time the user types a character.

For example, without debounceTime, every keystroke would emit an event, leading to multiple HTTP requests. However, when debounceTime is used, it ensures that the observable waits for a set period after the last keystroke before emitting the result. This means that as the user types, the HTTP request will only be made after they stop typing for the defined duration, reducing the number of calls to the backend and improving performance, especially when dealing with costly or frequent network requests.

- Response to 2.1.6. Question 6

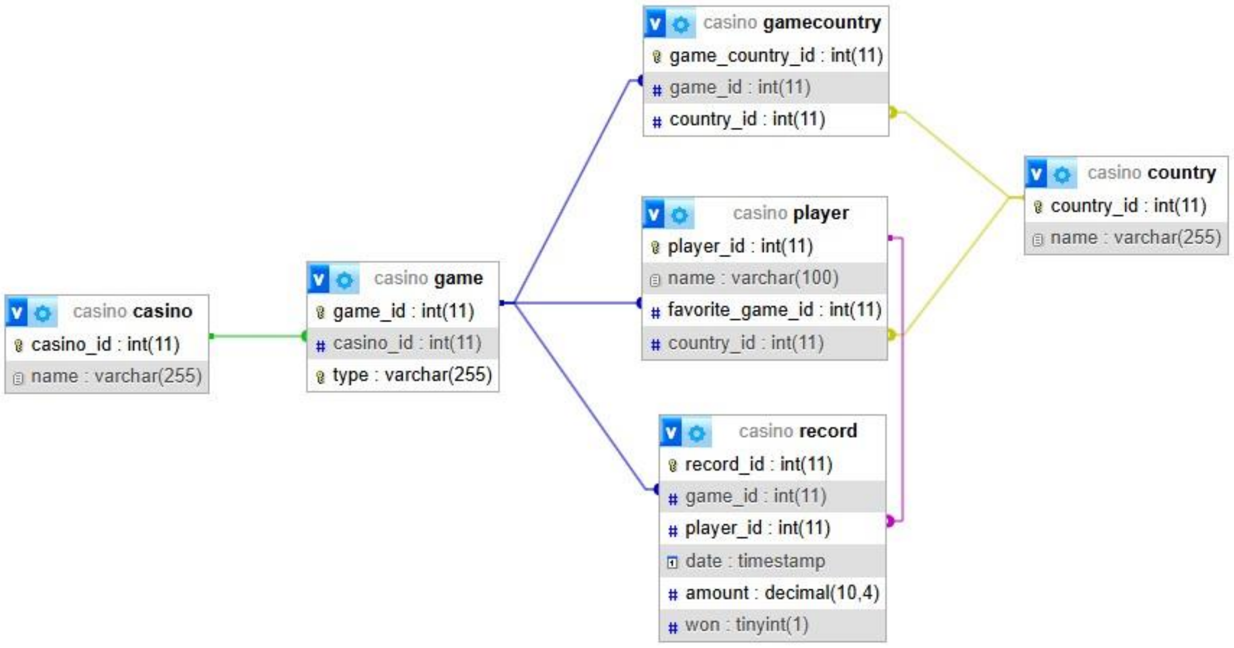


```
export const currency = async() => {
    //Consult external exchange currency api        Yo
    return  await axios.get(process.env.URLCURRENCY);
};
```

In the backend, **Axios** is used to make HTTP requests to an external API to retrieve the currency list.

- Response to 2.1.7. Question 7

Schema



SQL statements:

```
CREATE DATABASE casino;

CREATE TABLE casino.Casino (
  casino_id int NOT NULL AUTO_INCREMENT,
  name varchar(255) NOT NULL,
  PRIMARY KEY (casino_id)
);

CREATE TABLE casino.Game (
  game_id int NOT NULL AUTO_INCREMENT,
  casino_id INT NOT NULL,
  type VARCHAR(255) UNIQUE NOT NULL,
  PRIMARY KEY (game_id),
  FOREIGN KEY (casino_id) REFERENCES Casino (casino_id) ON DELETE CASCADE
);

CREATE TABLE casino.Country (
  country_id int NOT NULL AUTO_INCREMENT,
  name varchar(255) NOT NULL,
  PRIMARY KEY (country_id)
);
```

```
CREATE TABLE casino.GameCountry (
  game_country_id int NOT NULL AUTO_INCREMENT,
  game_id INT NOT NULL,
  country_id INT NOT NULL,
  PRIMARY KEY (game_country_id),
  FOREIGN KEY (game_id) REFERENCES Game(game_id) ON DELETE CASCADE,
  FOREIGN KEY (country_id) REFERENCES Country(country_id) ON DELETE CASCADE
);
CREATE TABLE casino.Player (
  player_id INT AUTO_INCREMENT,
  name VARCHAR(100) NOT NULL,
  favorite_game_id INT,
  country_id INT NOT NULL,
  PRIMARY KEY (player_id),
  FOREIGN KEY (favorite_game_id) REFERENCES Game(game_id) ON DELETE SET
NULL,
  FOREIGN KEY (country_id) REFERENCES Country(country_id) ON DELETE CASCADE
);
CREATE TABLE casino.Record(
  record_id INT AUTO_INCREMENT,
  game_id INT NOT NULL,
  player_id INT NOT NULL,
  date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  amount DECIMAL(10, 4) NOT NULL,
  won TINYINT(1) NOT NULL,
  PRIMARY KEY (record_id),
  FOREIGN KEY (game_id) REFERENCES Game(game_id) ON DELETE CASCADE,

  FOREIGN KEY (player_id) REFERENCES Player(player_id) ON DELETE CASCADE

);
```

- Response to 2.1.8. Question 8
  During the development of this project, AI code assistants such as ChatGPT and GitHub Copilot (Blackbox) were used to help with specific tasks, particularly to enhance the development process. Here's a detailed description of how they were used:
  Animation Libraries: AI tools were utilized to understand how to implement animation libraries, specifically canvas-confetti for creating celebratory effects and GSAP (GreenSock Animation Platform) for smooth, performant animations. The AI assisted with code snippets and guidance on how to properly use these libraries within the project.
  Debounce Implementation: ChatGPT was used to understand the best practices for implementing debounceTime with RxJS in Angular. The AI provided valuable insights into how to optimize user input handling by preventing excessive API calls during typing, which was critical for improving performance in specific features of the project.
  The AI tools were used as a supplementary resource for understanding implementation details and learning how to integrate these libraries and functionalities into the project efficiently.

  Instructions for executing projects:
  Backend :

```
    npm install
    Create .env
            PORT = 3000
            SECRET_KEY = "testApi"
            USERLOGIN = "admin"
            PASSWORDLOGIN = 5678
            CHERRY = '🍒'
            LEMON = '🍋'
            APPLE = '🍎'
            BANANA = '🍌'
            COINS = 20
```

```
URLCURRENCY = 'https://open.er-api.com/v6/latest/EUR'
URLSERVER = 'http://localhost:3000'
```

Run server: npm run start

Run unit test: npm run test

Angular:

npm install

Run project: ng serve

Test done by,

Duvan Mendivelso
Software developer
duvamendi2@hotmail.com