# Practical Project: Creating a Single Page App – Part 1

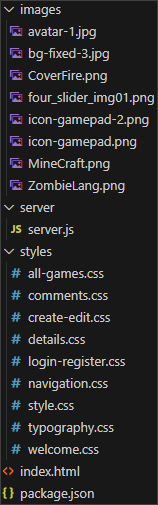
You are assigned to implement a **Web application** (**SPA**) using JavaScript. The application should dynamically display content, based on user interaction and support user profiles and CRUD operations, using a REST service.

The application **must start** from **index.html** file on port **3000**.

## Overview

In this workshop, we will have to **implement** a **front-end application** (**SPA**) to see and manage **the games world**. The app allows visitors to **browse** different types of games, including the latest collections, with different levels of difficulty. Users can **register** with an **email** and **password**, which allows them to create their own games. Game authors can also **edit** or **delete** their own posts at any time.

## Technical Details



We are provided with the following resources:

* **Project scaffold**: A **package.json** file, containing a list of common dependencies.
  + **lit-html**: Allows for efficient HTML rendering;
  + **lite-server**: Provides a lightweight development server;
  + **lru-cache**: Implements a least recently used caching strategy;
  + **page**: Client-sid router for SPAs.
* To initialize the project, we have to execute the following commands via the command-line terminal:

|  |
| --- |
| **npm install**  **npm run server**  **npm run start** |

The **npm run start** command should be run in a new terminal, as the **npm run server** command starts the server and the terminal, in which this command is run, will display the server requests.

* **HTML** **and CSS files**: All views (pages) of the application, including **sample** user-generated **content**, are included in the file **index.html**, which links to CSS and other static files. **Each view is in a separate section** of the file, which can be identified by a **unique class name** or **id** **attribute**. The application may use any preferred method (such as a **templating library** or manual visibility settings) to display only the selected view and to **navigate** between views upon user interaction.
* **Local** **REST service**: A special server, which contains **sample data** and supports **user registration** and **CRUD operations** via REST requests, is included with the project. Each section of this document (where applicable) includes details about the necessary **REST endpoints**, to which **requests** must be sent, and the **shape** of the expected **request body**.

When creating HTML Elements and displaying them on the page, we must **adhere as close as possible to the provided HTML** samples. We may **add attributes** (such as **class** and **dataset**) to any HTML Element, as well as **change "href"** attributes on links and add/change the **method** and **action** attributes of HTML Forms, to facilitate the correct operation of a routing library or another method of abstraction. We may also add hidden elements to help implementing certain parts of the application requirements.

## Using the Local REST Service

### Starting the Service

The REST service is in a folder named **server**, inside the provided resources archive. It has no dependencies and can be started by opening a terminal in its directory and executing:

|  |
| --- |
| **npm run server** |

If everything is initialized correctly, we should see a message about the **host address** and **port** on which the service will respond to requests.

### Sending Requests

To send a request, we will use the **hostname** and **port**, shown in the initialization log and **resource address** and **method** as described in the **application requirements**. If data needs to be included in the request, it must be **JSON-encoded**, and the appropriate **Content-Type** **header** must be added. Similarly, if the service is to return data, it will be JSON-encoded. We must note that **some requests do not return a body** and attempting to parse them will throw an exception.

Read requests, as well as login and register requests do not require authentication. All other requests must be authenticated.

### Required Headers

To send data to the server, we include a **Content-Type** header and encode the body as a JSON-string:

|  |
| --- |
| Content-Type: application/json  {JSON-encoded request body as described in the application requirements} |

To perform an authenticated request, we will include an **X-Authorization** header, set to the value of the **session token**, returned by an earlier login or register request:

|  |
| --- |
| **X-Authorization: {session token}** |

### Server Response

Data response:

|  |
| --- |
| **HTTP/1.1 200 OK**  **Access-Contrl-Allow-Origin: \***  **Content-Type: application/json**  **{JSON-encoded response data}** |

Empty response:

|  |
| --- |
| **HTTP/1.1 204 No Content**  **Access-Contrl-Allow-Origin: \*** |

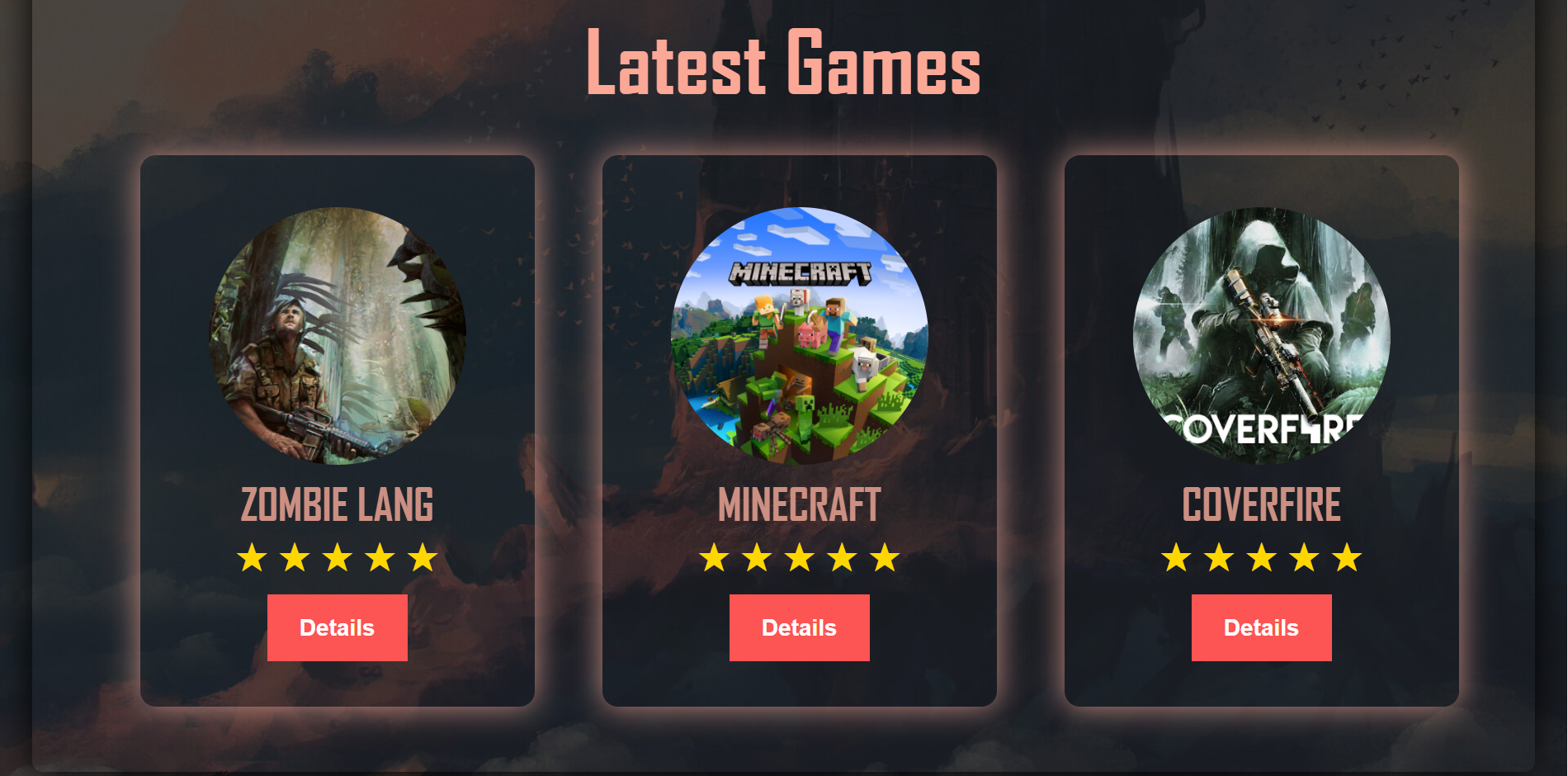
Error response:

|  |
| --- |
| **HTTP/1.1 400 Request Error**  **Access-Contrl-Allow-Origin: \***  **Content-Type: application/json**  **{JSON-encoded error message}** |

## Application Requirements

### Home Page

All users should be greeted from the homepage, where they should be able to see the three most recently added games. Clicking on the details links leads to the details page for the selected game.



If no games **have been added** yet, show the text **"No games yet"** instead.



We will send the following **request** to read the new games:

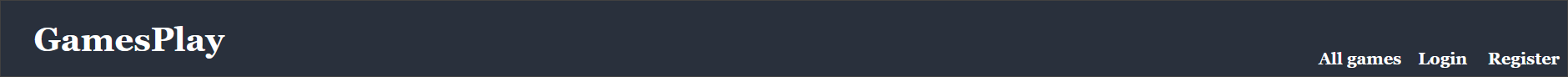
|  |
| --- |
| **Method: GET**  **URL: /data/games?sortBy=\_createdOn%20desc&distinct=category** |

Required **headers** are described in the documentation. The service will return an array of games.

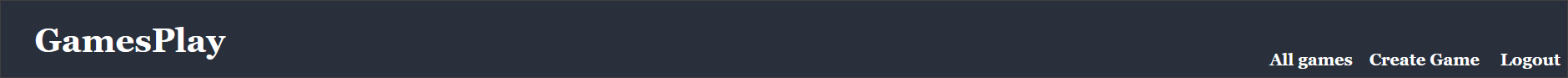
### Navigation Bar

Navigation links should correctly change the current page (view). **GamesPlay** link should redirect to the **Home** page. **Guests** (un-authenticated visitors) can see the links to the **All Games (Catalogue)** page, as well as the links to the **Login** and **Register** pages. The logged-in user navbar should contain the links to **All Games** **(Catalogue)** page, the **Create** page and a link for the **Logout** action.

**Guest** navigation example:



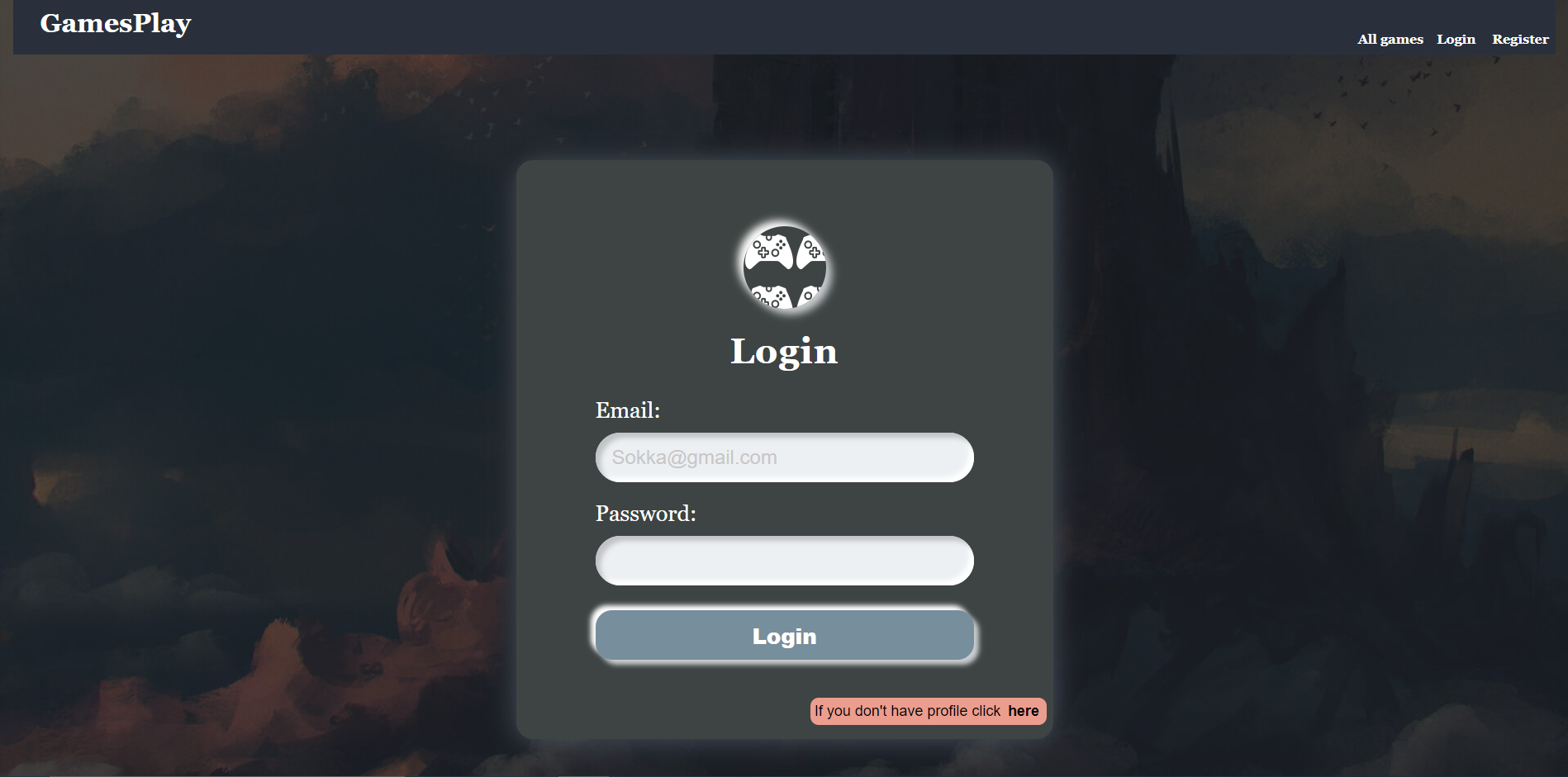
**User** navigation example:



### Login User

|  |
| --- |
| The included REST service comes with the following premade user accounts, which you may use for development:  { "email": "peter@abv.bg", "password": "123456" }  { "email": "john@abv.bg", "password": "123456" } |

The **Login** page contains a form for existing user authentication. By providing an **email** and **password,** the app should login a user in the system if there are no empty fields.



We will send the following **request** to perform login:

|  |
| --- |
| **Method: POST**  **URL: /users/login** |

Required **headers** are described in the documentation. The service expects a body with the following shape:

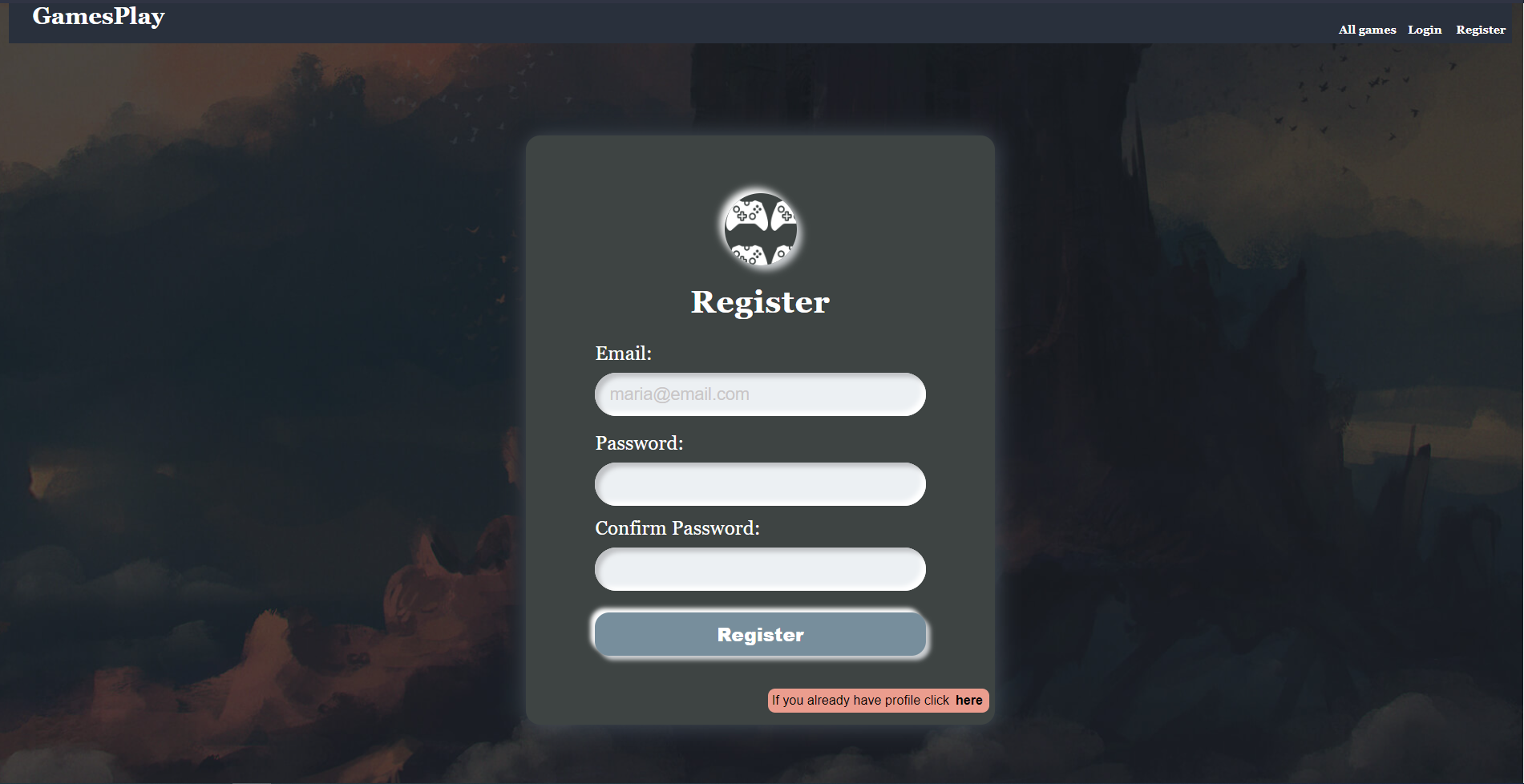
|  |
| --- |
| {  email,  password  } |

Upon success, the **REST service** will return information about the existing user along with a property **accessToken**, which contains the **session token** for the user **–** you need to store this information using **sessionStorage** or **localStorage,** in order to be able to perform authenticated requests.

If the login was successful, we have to **redirect** the user to the **Home** page. If there is an error, we have to display an appropriate error message, using a system dialog (window.alert).

### Register User

The **Register** page contains a form for new user registration. By providing an **email** and **password,** the app should register a new user in the system if there are no empty fields.



We will send the following **request** to perform registration:

|  |
| --- |
| **Method: POST**  **URL: /users/register** |

Required **headers** are described in the documentation. The service expects a body with the following shape:

|  |
| --- |
| **{**  **email,**  **password**  **}** |

Upon success, the **REST service** will return the newly created object with an automatically generated **\_id** and a property **accessToken**, which contains the **session token** for the user **–** you need to store this information using **sessionStorage** or **localStorage,** in order to be able to perform authenticated requests.

If the registration is successful, we will **redirect** the user to the **Home** page. If there is an error, or the **validations** don't pass, we will display an appropriate error message, using a system dialog (window.alert).

### Logout

The logout action is available to logged-in users. We will send the following **request** to perform logout:

|  |
| --- |
| **Method: GET**  **URL: /users/logout** |

Required **headers** are described in the documentation. Upon success, the **REST service** will return an **empty response**. Clear any session information you’ve stored in browser storage.

If the logout is successful, we will **redirect** the user to the **Home** page.

## What Is a SPA?

No, SPA is not a relaxing spa procedure. It is something much more exciting.

SPA stands for Single Page Application and is a widely used application architecture in the modern programming world.

Now, a little bit of theory in order for us to get acquainted with the concept of a SPA.

### Overview

A Single Page Application (SPA) is a web application or website that interacts with the user by dynamically rewriting the current page rather than loading entire new pages from the server. This approach avoids interruption of the user experience between successive pages, making the application behave more like a desktop application.

### How It Works

#### Initial Load

A SPA loads a single HTML page and dynamically updates the content as the user interacts with the app.

When a user first visits the SPA, the server sends only the initial HTML, CSS, and JavaScript files necessary to load the application. Unlike traditional multi-page applications, the entire webpage is not reloaded during user interaction.

#### User Interaction

User actions in the application trigger JS functions instead of server requests for different pages. This includes actions like clicking navigation links, submitting forms, etc.

#### AJAX Requests

SPAs primarily use AJAX calls to fetch data from the server using requests and then use JavaScript to update the DOM based on the data received, without reloading the page, allowing the application to update dynamically.

#### Client-Side Routing

SPAs use client-side routing. When a user clicks a link, the URL can change but the browser does not make a request to the server for the new page. Instead, the SPA handles this change, fetching only the data that needs to be updated.

#### DOM Manipulation

Once the data is fetched, JavaScript manipulates the DOM to reflect changes (like displaying a list of games or a user profile).

#### Data Binding and State Management

The application maintains a front-end "state" that tracks user interaction and data. Changes to the state trigger updates to the view, ensuring the UI stays consistent with the underlying data.

#### Frameworks and Libraries

Most SPAs are built using frameworks or libraries, which provide structured ways of updating the view and syncing with the server.

## Development Workflow

Now, after all that boring theory, let's examine the provided skeleton and see what other files we should add, so that our application can match the concepts for the SPA architecture.

#### Initial Load

These are the files that are loaded when the SPA initially starts up.

##### Existing files

* **index.html**

##### Files to create

* **app.js:**
  + This file will serve as the **entry** point of the application.

#### User Interaction

These are the files that contain handlers, which respond to user actions.

##### Files to create

* **/util/eventHandler.js**
  + This file will hold the logic for managing the user interactions, connecting the UI events with the API calls in order for the basic CRUD operations to be made. In other words, the logic here will connect the front-end actions with the back-end operations, using the requests.js API calls and the page.js library.

#### AJAX Requests

These files are responsible for making AJAX requests to the server to fetch or send data.

##### Files to create

* **/api/requests.js**
  + This file will hold the logic for exporting APIs from separate modules into a single requests object. This way, we will allow the SPA to import and use these APIs as one consolidated object.
* **/api/games.js**
  + This file will hold the logic for handling API requests that are related to the **All Games Page (Catalogue)** requirement. The file will define a function, interfacing with a backend server using fetch API calls.
* **/api/users.js**
  + This file will hold the logic for handling API requests that are related to the users requirements. The file will define functions, interfacing with a backend server using fetch API calls.
* **/util/urls.js**
  + This file will hold the logic for managing and building the URLs used for the API requests.

#### Client-Side Routing

These files manage the routing within the SPA, ensuring that different views are loaded based on the URL without a full-page refresh.

##### Files to create

* **/router/engine.js**
  + This is the **core** of the **routing** **system**, which allows the SPA to handle different routes (URL paths) by rendering different views, without the need of reloading the entire page.
* **/router/viewHandler.js**
  + This file will hold the logic for managing the rendering of different parts of the application. We will use the **lit-html** library for rendering HTML templates dynamically and it will interact with the **requests.js** module to fetch the necessary data for the different views.

#### DOM Manipulation

These files handle the rendering and updating of HTML elements on the page based on the application's state and data.

##### Files to create

* **/templates/catalog.js**
* **/templates/home.js**
* **/templates/login.js**
* **/templates/nav.js**
* **/templates/register.js**
* **/templates/templates.js**

All of these files will handle the rendering and updating of the HTML elements on the page.

#### Data Binding and State Management

This file manages user-related data, handling tasks like storing and retrieving user session information.

##### Files to create

* **/util/userInfo.js**
  + This file will hold the user-related data, which is stored in the browser's **sessionStorage**.

## Development

Finally, after so much theory, let's start with implementing the SPA.

### Home Page

First of all, we need to start with the **Home** page. In order to do that, our first step is to create the **JavaScript** file which will hold the logic for **rendering** the code for the **Home** section of our SPA.

Create a new folder called **template** in the root directory of the project and add a new **home.js** file in it.

Let's start by importing the **html** templating function from the **lit-html** library:



Then, we need to declare a function which will render the array, holding the game objects:



Once we have declared the function, within it, we have to define the structure of the HTML that has to be rendered, using an **html** literal. We can take the **html** code from the **index.html** file and use it in the function body. Look for the section with comment

|  |
| --- |
| **<!--Home Page-->** |

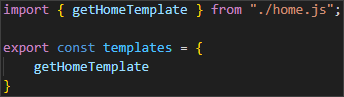
Do not forget to delete the used code from the **index.html** file.

Inside the template, we will dynamically generate content, based on the **length** of the **array**, which holds the games objects. The function must look like the image below:



It is a good practice to hold all of the template files in one common **templates.js** file. This practice is considered good, because this way the application is easily maintainable, the reusability and scalability are optimized, which leads to a higher efficiency of the development workflow.

Having this in mind, we will add an additional **templates.js** file, which will look like this for now:



Once we have done this, it's time for the **/router/viewHandler.js** file. It will hold the logic for rendering the **template** file, holding the **html** code for the **Home** page.

Let's start by importing the **render** function from the **lit-html.js** file and after that let's import the **templates.js** file:

****

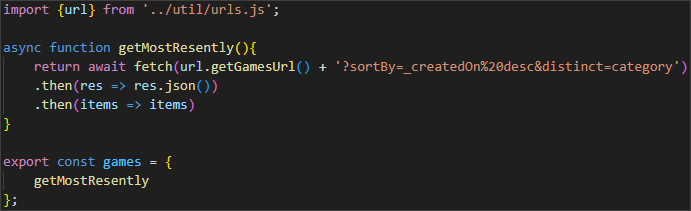
In this file, we have to implement a function, which will handle the asynchronous data fetching via an API module.

So, as we still don't have the API module, we must first create it and then import it in this file. But, as this is a SPA, we first need logic for managing and building the URLs used for the API requests.

In order to achieve this, go to the **root** directory of the project and create a new **util** folder. In it, create a **urls.js** file. The file should look like this for now:



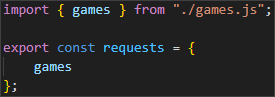
Once we have this logic, we can go back to the **root** directory of the project and create a new **api** folder. In it, we will create a **games.js** file. The file should implement an asynchronous function that fetches the most recently added or updated games from the server. We must not forget to import the **urls.js** file so that we can use the **getGamesUrl()** function from it:



As you can see, we are using the Fetch API to make a GET request to the server.

Like we did with the **templates** files, we will combine all of the **api** files in one common file, holding references to the individual ones.

Create a new **requests.js** file, which should look like this:



Now, we are ready to go back to the **viewHandler.js** file. We already imported the **lit-html** library, as well as the **templates.js** file, so it's time to import the **requests.js** file:

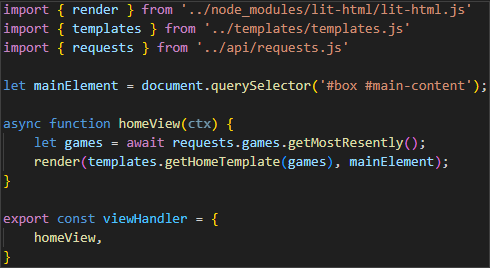


Then, we have to select the DOM element, which will act as the container for the main content of the application:



Once we have done this, we can implement an asynchronous function, which will fetch the data for the needed games, as an API request. Once the data is received, it will update the **mainElement** dynamically with the content, generated by the template.

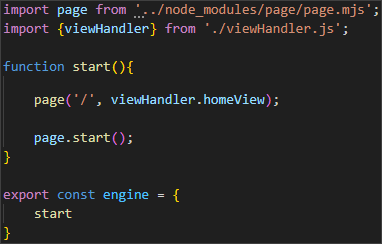
Finally, the file should look like this:



Now, it's time to create the **/router/engine.js** file. As you remember, this file is responsible for the handling of the client-side routing.

We will use the **viewHandler.js** and the **page.js** library in order to do that.

The file will hold coupled path and view, which will be a function itself, as well as some utility functions and it should look like this:



Finally (for now), we need to create the **/app.js** file. As the **app.js** is the entry point for the SPA, it will only implement the start logic for the engine of the application:

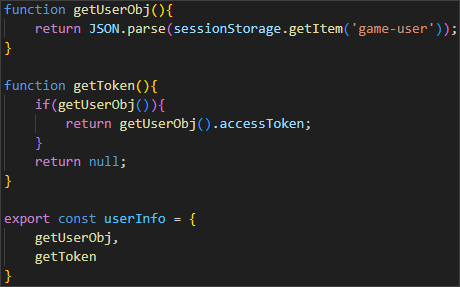


Now, refresh the application in the browser.

Congratulations! You should be able to see the Home Page fully rendered and loaded.

### Navigation Bar

Ok, now, let's implement the navigation bar. The steps that we must follow are pretty much the same, but we have one additional step here – we have to create a userInfo.js file, which will hold the logic for abstracting the details of reading from **sessionStorage**, providing a clean and reusable interface for accessing user session data.

The file should look like this:  


Once we have done that, we can continue with the template for the navigation bar, where we will import the **userInfo.js**. Name the file **nav.js**.

Again, take the html from the index.html file. Look for the section

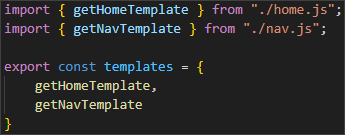
|  |
| --- |
| **<!-- Navigation -->** |

Don't forget to delete the html code from the **index.html** file.

The template file should look like this:

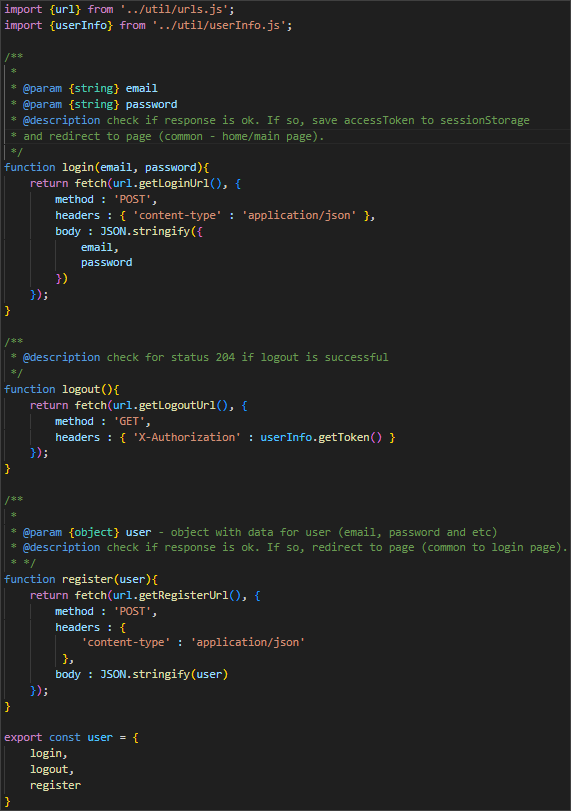


Next step is to import the file to the **templates.js** file:

****

Now, in the **api** folder, we will add the file for the users and name it **user.js**. This file will handle the user authentication and registration. It will interact with the server using HTTP requests.

The file should look like this:



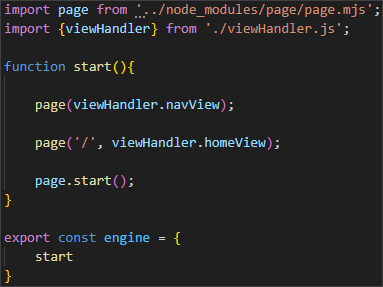
After we have ensured that we have the necessary logic in the **user.js** file, don't forget to import it to the **requests.js** file:



Now, in the **viewHandler.js**, we have to find the header DOM element and update it:



Now, we have to go to the engine.js file and update it accordingly:



Try reloading the page. You should be able to see the navigation bar. Good job!

### Login User

After our success, it's time to dive into the login functionality of our SPA.

You know the drill – first, create the login.js file, using this section from the **index.html** file:

|  |
| --- |
| **<!-- Login Page ( Only for Guest users ) -->** |

Don't forget to delete the code from the **index.html** file.

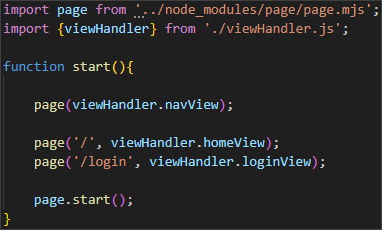
Now, we have to create an additional **eventHandler.js** file in the **/util** directory. Its purpose is to provide a secure process for logging in the SPA, while handling form submissions for user login. It processes the form and sends an API request to the server, sending the email and password to it. It also stores the user data in the **sessionStorage** after the successful login in order to maintain the user session state while using the SPA.



After we are ready with this file, we have to import it in the login.js template file, which should look like this:



After that you know the steps – import the template file into the **templates.js** and update the **viewHandler.js** file. Finally, update the **engine.js** file:



Refresh the page and click on the **[Login]** button. You should be able to see the Login page. Try logging with the provided credentials:

|  |
| --- |
| **{ "email": "peter@abv.bg", "password": "123456" }**  **{ "email": "john@abv.bg", "password": "123456" }** |

Good job! 😊

### Register User

Now, it's time to extend the functionality of our application and add the **Register** feature.

Before creating the template file, let's expand our **eventHandler.js** file. Add the following code to the file:



Once we are ready with this, let's create a **register.js** file:



Import it in the **templates.js** file and update the **viewHandler.js** and **engine.js** accordingly.

Now restart the server and try registering.

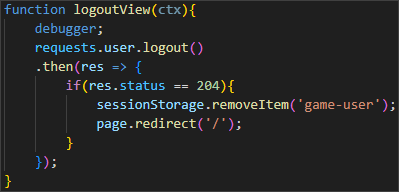
You should be able to register a new user and automatically login after that.

Great! Keep up the good work!

### Logout

Finally, for this workshop, we will implement the **Logout** functionality of our SPA.

This time it is a little bit different. We won't have a template file, because we don't need a logout view. All you have to do is update the **viewHandler.js** with the following function:



Don't forget to export the function at the end.

And, of course, let's not forget the **engine.js** file.

After you have made these changes, login in the application and click on the **[Logout]** button.

You should be able to logout and see the required page.

Good work! In this workshop, we were able to create a single page application, which provides functionalities for a Home page, a Login page, a Register page, Logout and displaying a navigation bar. 😊