# Practical Project: Creating a Single Page App – Part 2

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.

In the previous workshop, we **implemented** a **front-end application** (**SPA**) to see and manage **the games world**. We implemented features like a **Home** page, a **Login** page, a **Register** page, **Logout** and displaying a **navigation** **bar**. Now, we will continue with implementing features for the SPA, such as **browsing** different types of games, including the latest collections, with different levels of difficulty, **creating** user's own games, **editing** or **deleting** their own posts at any time.

## Technical Details

As skeleton for the workshop, we are going to use the updated with our code skeleton from the previous workshop.

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

No changes here – the requirements for starting the server, sending requests, required headers and server response are the same as in the previous workshop.

### 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:

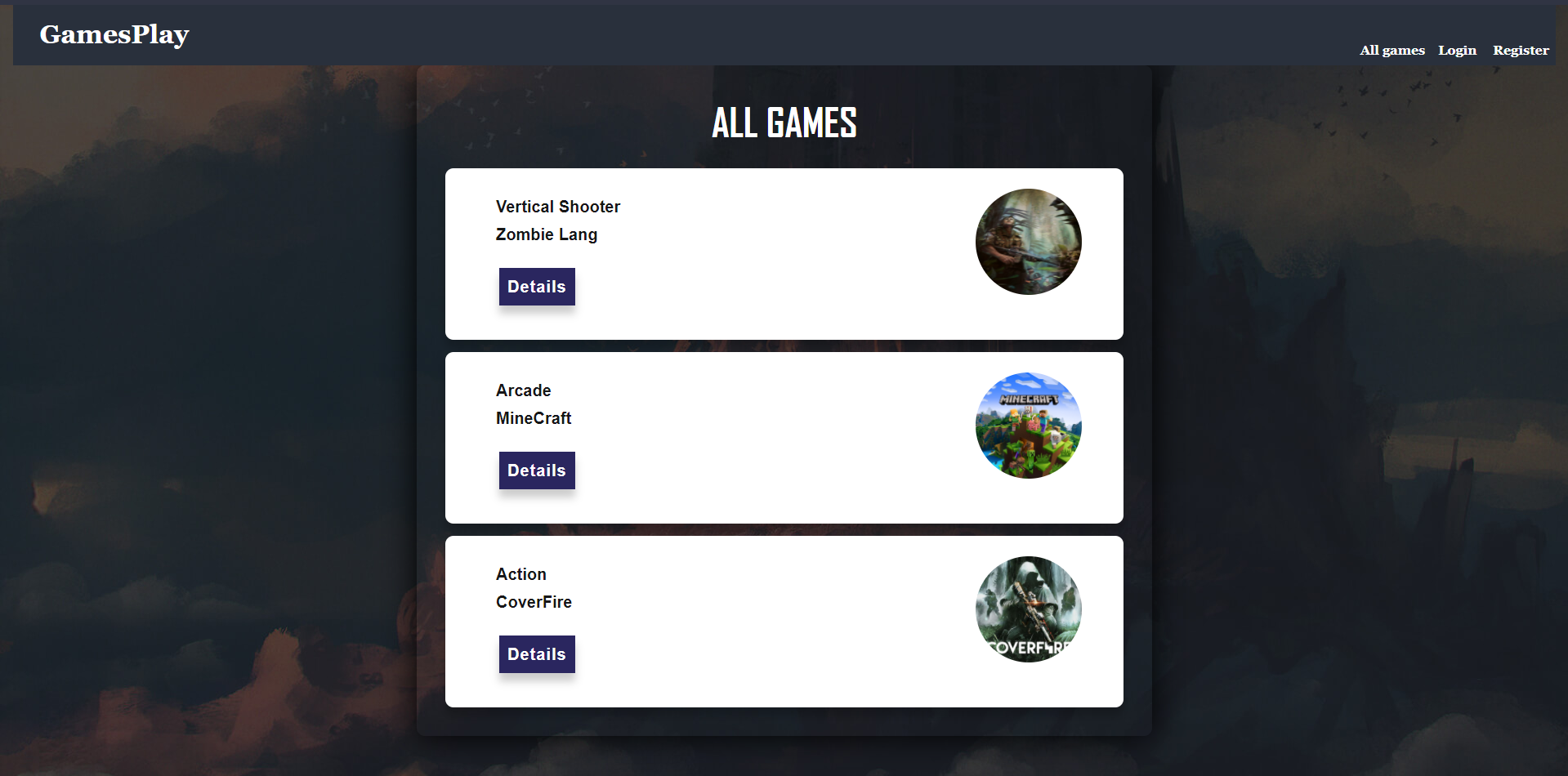
|  |
| --- |
| **HTTP/1.1 204 No Content**  **Access-Contrl-Allow-Origin: \*** |

Error response:

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

## Application Requirements

### All Games Page (Catalogue)

This page displays a list of all games in the system, with their title and category. Clicking on any of the cards leads to the details page for the selected game.

If there are nogames, we will display the following view:



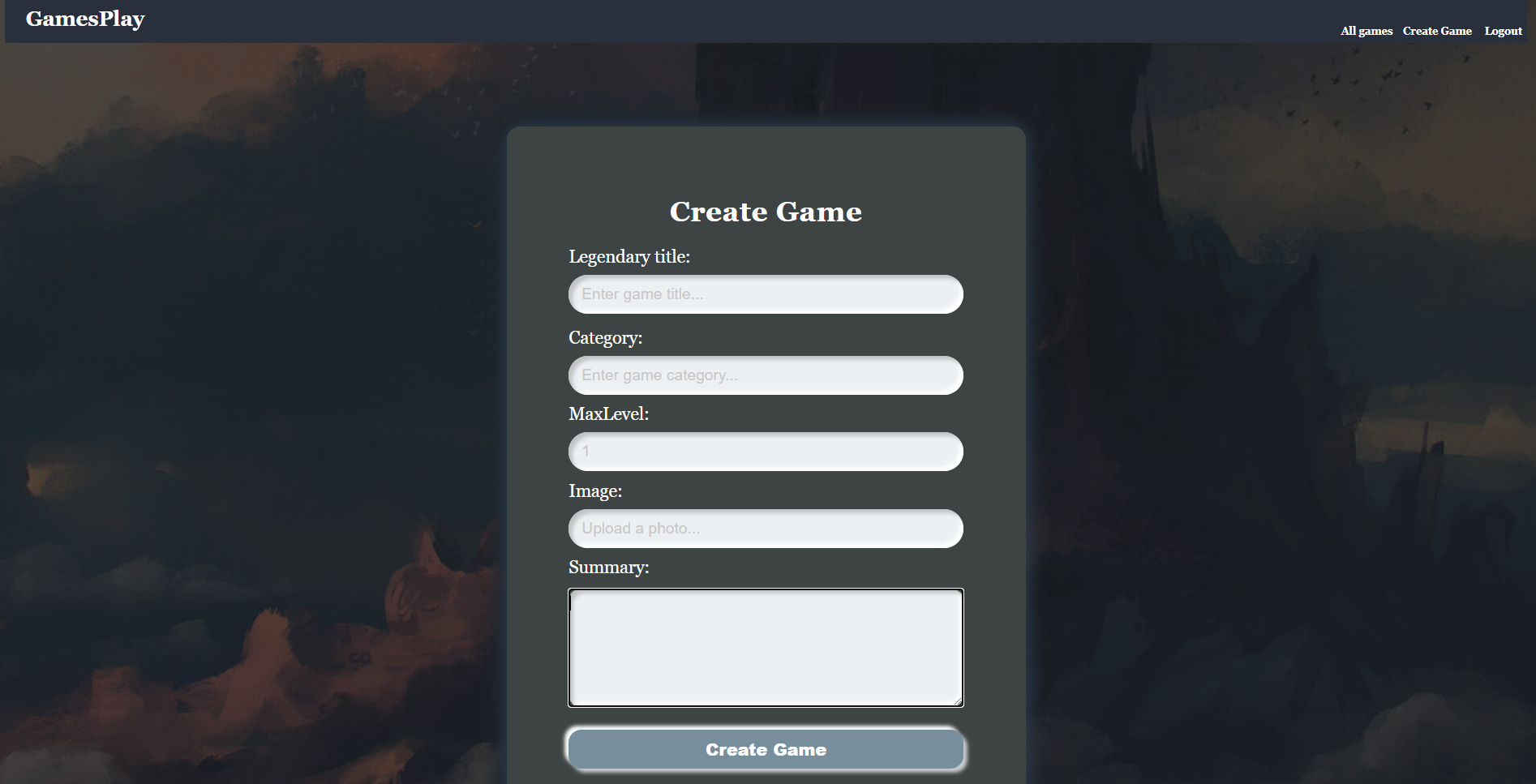
Send the following **request** to read the list of games:

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

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

### Create Game

The Create page is available to **logged-in** users. It contains a form for creating new games. We will check if all the fields are filled before we send the request.



To create a game, send the following **request**:

|  |
| --- |
| **Method: POST**  **URL: /data/games** |

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

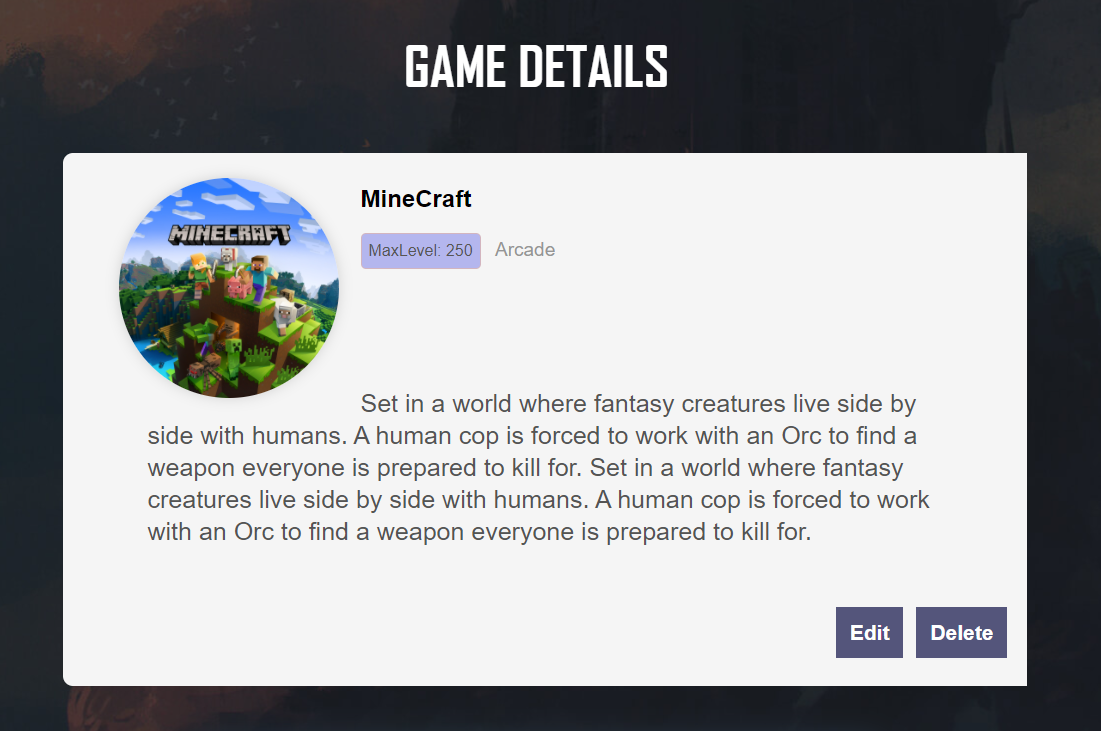
|  |
| --- |
| **{**  **title,**  **category,**  **maxLevel,**  **imageUrl,**  **summary**  **}** |

Required **headers** are described in the documentation. The service will return the newly created record. Upon success, we will **redirect** the user to the **Home** page.

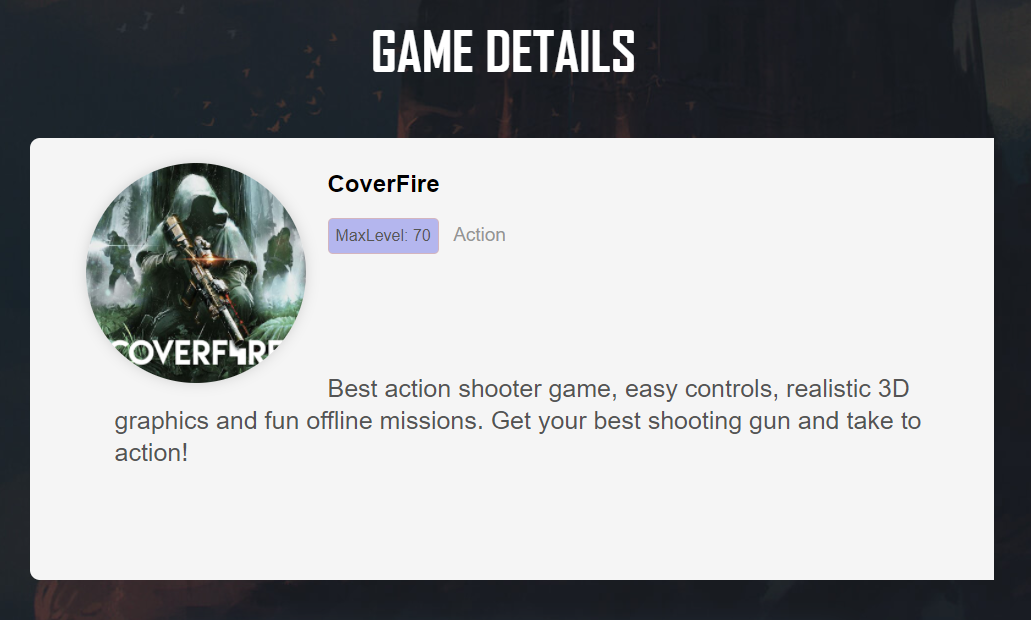
### Details

All users should be able to **view details** aboutgames. Clicking the **Details** link in of a **game** should **display** the **Details** page:

* If the currently logged in user is the **creator** of the game, the **Edit** and **Delete** buttons should be displayed, otherwise they should not be available.
* The view for the **creators** should look like:



The view for **guests** and **logged-in users** should look like:



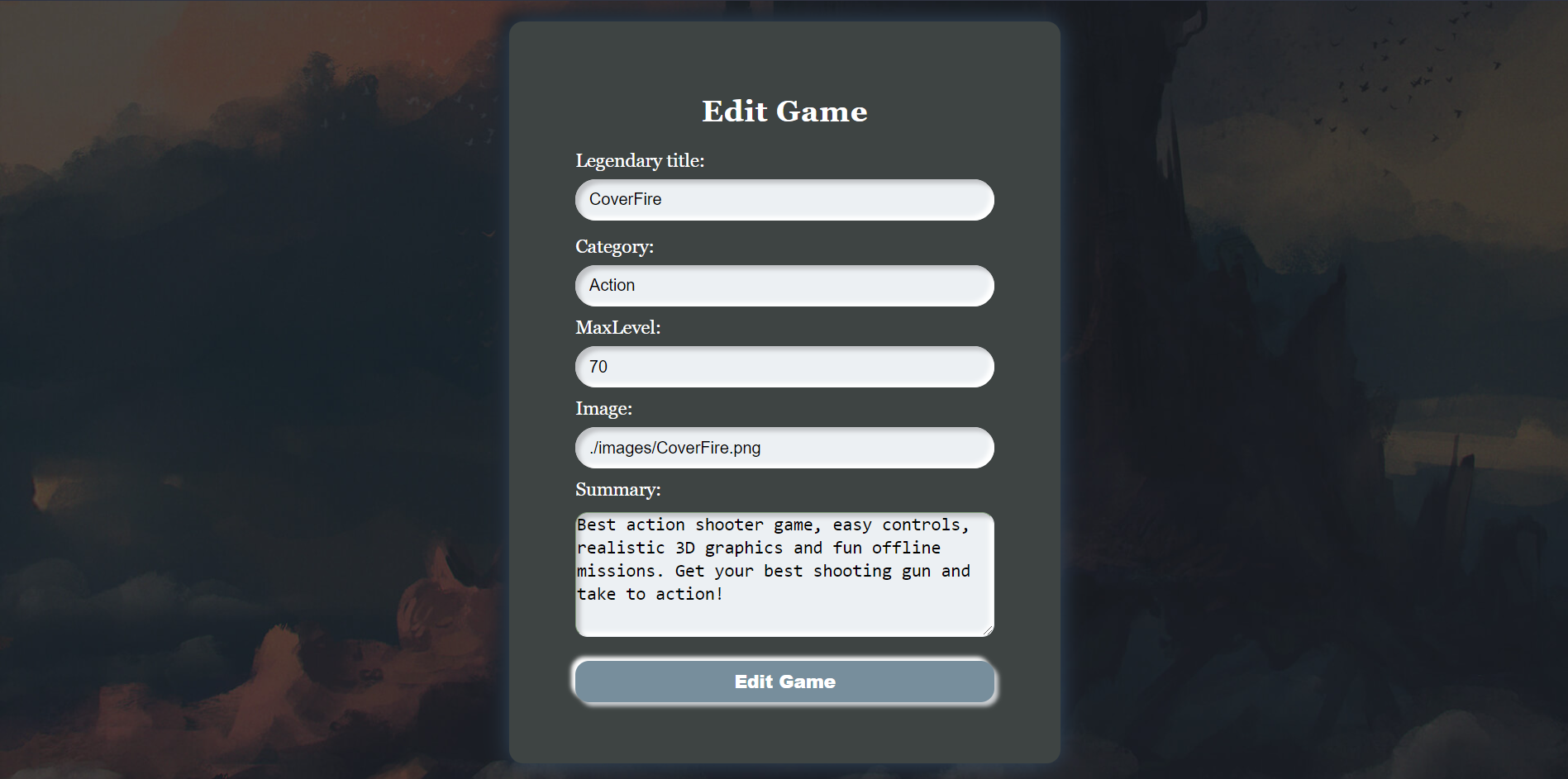
We will send the following **request** to read a single game:

|  |
| --- |
| **Method: GET**  **URL: /data/games/:id** |

Where **:id** is the **id** of the desired game. Required **headers** are described in the documentation. The service will return a single object.

### Edit Game

The **Edit page** is accessible to logged-in users and allows the author to **edit** their own games. Clicking the **Edit** a specific game link on the **details page** should display the Edit page. It contains a form with input fields for all relevant properties. Make sure **all fields** are filled in before submitting the request. The fields **must be filled** in when the page is first loaded.



To edit a game, send the following **request**:

|  |
| --- |
| **Method: PUT**  **URL: /data/games/:id** |

Where **:id** is the **id** of the desired game.

The service expects a body with the following shape:

|  |
| --- |
| **{**  **title,**  **category,**  **maxLevel,**  **imageUrl,**  **summary**  **}** |

Required **headers** are described in the documentation. The service will return the modified record. Note that **PUT** request **do not** merge properties and will instead **replace** the entire record. Upon success, we will **redirect** the user to the **Details** page for the current game.

### Delete Game

The delete action is available to logged-in users, for game they have created. When the author clicks on the Delete action on any of their games, a confirmation dialog should be displayed, and upon confirming this dialog, the game should be deleted from the system.

To delete a game, we will send the following **request**:

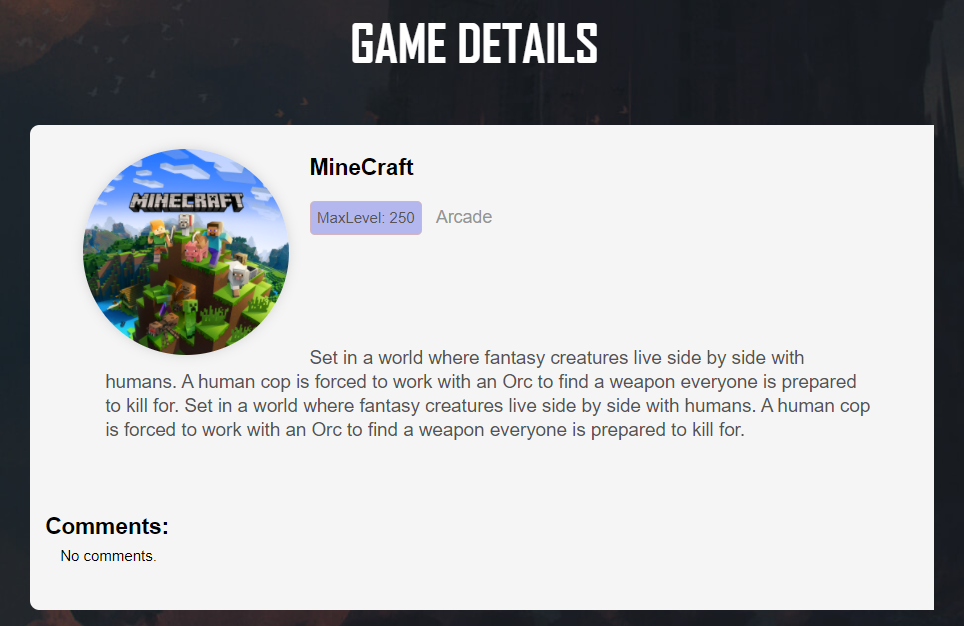
|  |
| --- |
| **Method: DELETE**  **URL: /data/games/:id** |

Where **:id** is the **id** of the desired game. Required **headers** are described in the documentation. The service will return an object, containing the deletion time. Upon success, **redirect** the user to the **Home** page.

### Comments

Every logged-in user should be able to **comments other games**, but **not his own**.

**Guest** should not be able to see the section **Add new comment**, but should be able to see the section **Comments** The view for **guests** should look like:



If there are **no comments all** users should see:



If there are **comments all** users should see:

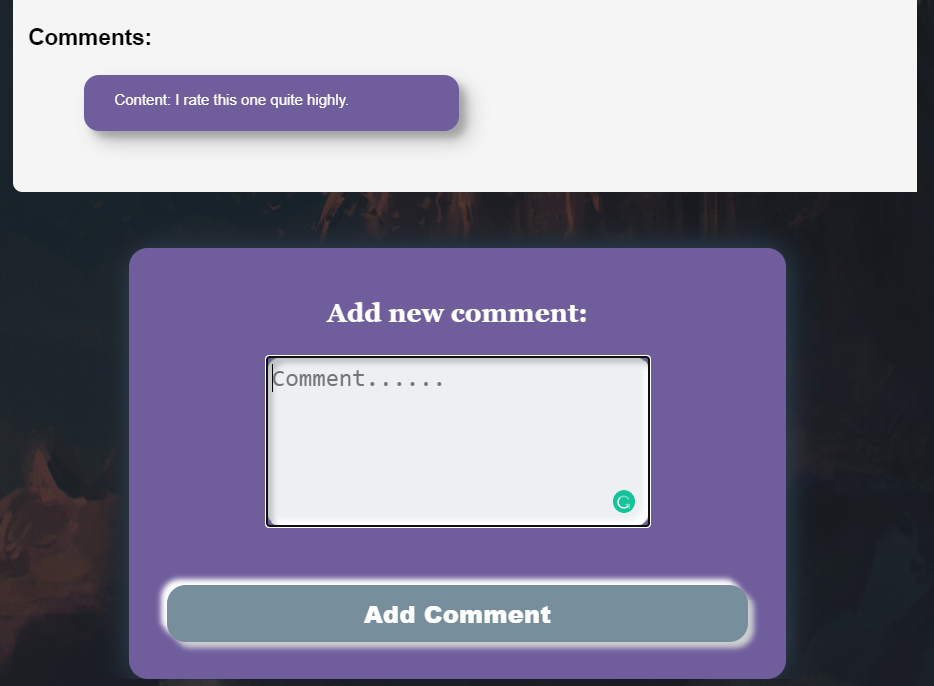


Logged-in users see a **form** for adding a new comment. Every **registered** user can leave a **comment** under any games. **Authors** can't comment on their **own** games.

The view when there are no comments yet and the user did not press **[Add Comment]** button should look like:



The view when the logged-in user add **Comment** to the game should look like:



Upon success, we will **clear** the content from the textarea field.

To **load** all comments for game, we will send the following request:

|  |
| --- |
| **Method: GET**  **URL: /data/comments?where=gameId%3D%22{gameId}%22** |

Where **{gameId}** is the **id** of the desired game

To **create** a new comment, we will send the following request:

|  |
| --- |
| **Method: POST**  **URL: /data/comments** |

The **service** expects a body with the following shape:

|  |
| --- |
| **{**  **gameId,**  **comment**  **}** |

Where gameId is the **id** of the game, which the comment is associated with, and comment is the text content. Upon success, we will **redirect** the user to the **same** page.

## Development

No, let's start with implementing the SPA.

### All Games Page (Catalogue)

Let's start with the **Catalogue** for the **All Games** Page. As you remember, first, we need the template JS file, which will hold the logic for rendering the code for this section in our SPA.

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:

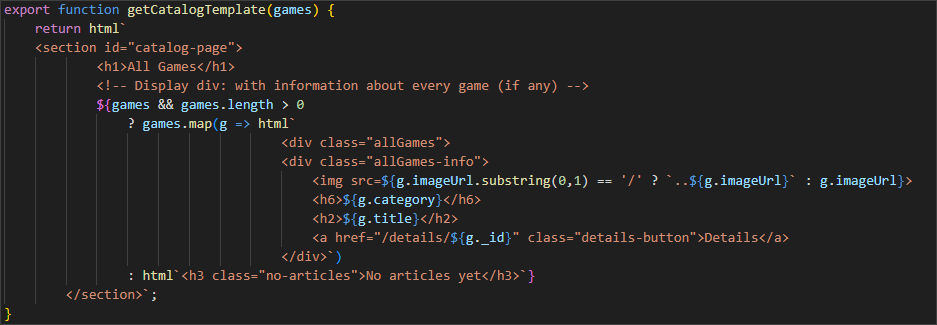


Once we have declared the function, within it, we have to define the structure of the HTML that has to be rendered, using an **html** literal. We can take the **html** code from the **index.html** file and use it in the function body. Look for the section with comment

|  |
| --- |
| **<!—Catalogue -->** |

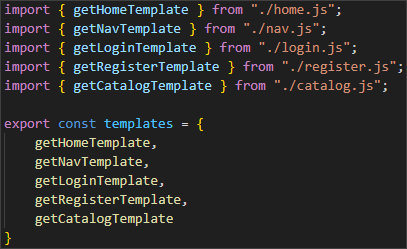
Do not forget to delete the used code from the **index.html** file.

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.

As you remember, we are holding all of the templates file in one common template file, so our next step is to import the **catalog.js** file into the **templates.js** file.

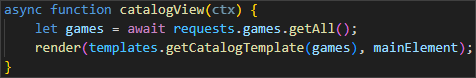
****

Now, it's time to go to the **games.js** file. We will implement an asynchronous function that fetches all of the games from the server.

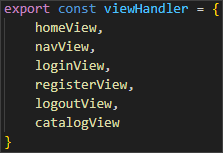


As you can see, we are using the Fetch API to make a GET request to the server.

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

The functions should look like this:  


Don't fortget to export it:



Let's not forget the **engine.js** file, where we have mapped the path and the view:



Now, refresh the application in the browser.

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

### Create Game

Ok, now, let's implement the more existing features. We are going to start with creating a new game. You already know the procedure. Let's start with the template create.js file.

Take the html from the **index.html** file. Look for the section

|  |
| --- |
| **<!-- Create Page ( Only for logged-in users ) -->** |

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

The template file should look like this:



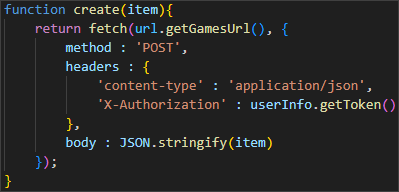
Next step is to import the file to the **templates.js** file:

****

After we have done this, in the **viewHandler.js**, we have to update the **main** DOM element:



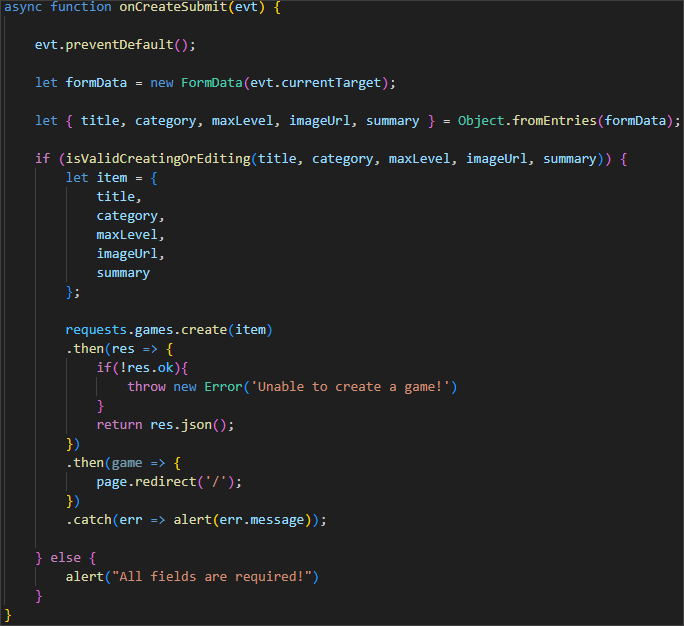
Now, in the **api/games.js** file we will create a new function, which will send new game data to the server. It will construct and send the POST request to the API endpoint, using the Fetch API:



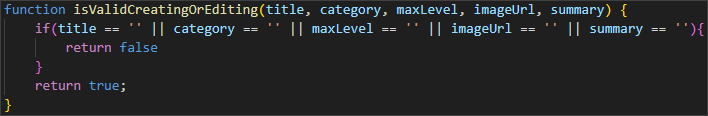
Don't forget to import the **userInfo**:



After that, we have to update the **eventHandler.js** file with the following code:



We need one additional function, which will validate the input data for the game that is being created:



We are doing this, because we need an event handler, which will manage the submission of a form, validation of the input data, as well as managing the user navigation.

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



Try reloading the page. Login as a registered user and navigate to the **Create Game** menu. Fill the form and try saving the game.

Congratulations! You were able to create a new game!

### Details & Comments

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

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

|  |
| --- |
| **<!—Details Page -->** |

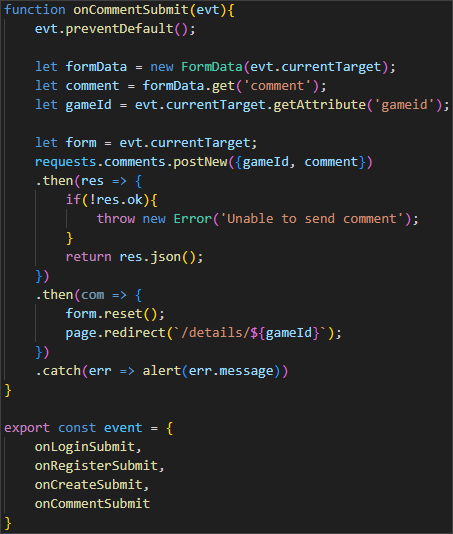
Don't forget to delete the code from the **index.html** file. The **details.js** file should look like this:

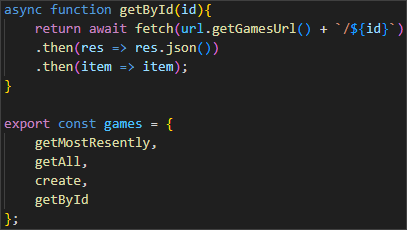


It's a bit tricky part here, because here we implement the logic for the comments as well.

Don't forget to modify the **templates.js** file as well.

As you can see, we have a function **onCommentSubmit**, which we will have to add in our **util/eventHandler.js** file:

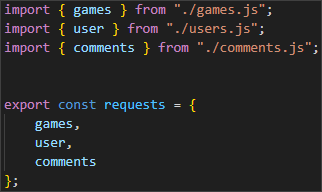


One additional change is adding an asynchronous function in the **api/games.js** file, which will fetch the game by id:  


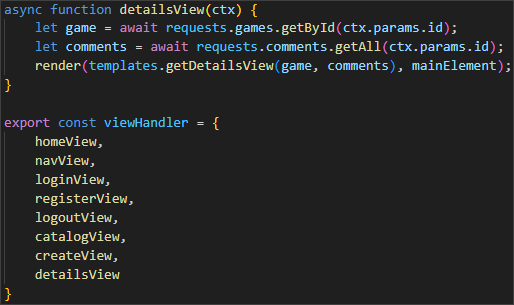
Finally, we have to create a new **comments.js** file in the **api** directory, which will handle the API requests, related to the comments:



After we have created the file, of course, we have to import it in the **requests.js** file:



Now, let's go to the **viewHandler.js** and add a new asynchronous function, which will handle the **Details** view for each game:



Of course, let's not forget to update the **engine.js** file as well.

Good job! 😊

### Edit Game

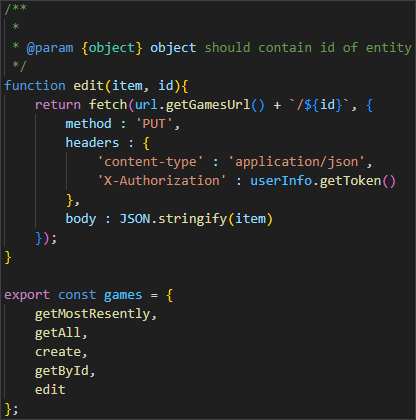
Now, it's time to extend the functionality of our application and add the **Edit** feature.

You know the procedure. Let's start with the template **edit.js** file:



Import it in the **templates.js** file accordingly.

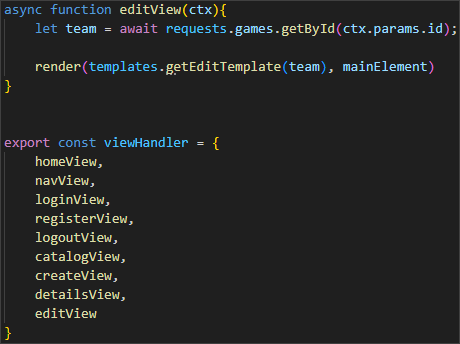
Now, go to the **api/games.js** file and add the following code:



We are also going to update the **eventHandler.js** with the following code:



Now, update the **viewHandler.js** accordingly:



One of the last steps is to add the following function in the **eventHandler.js** file:



Finally, modify the **engine.js** file:



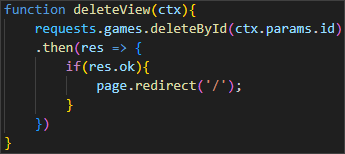
Reload the application and try editing the game you added in the previous section.

Great! Keep up the good work!

### Delete Game

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

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

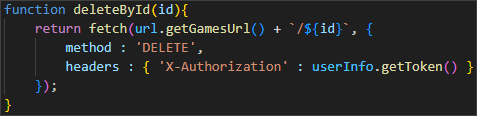


Also, import the **page.js** library, because we are going to need it for the routing after the deleting of the game:



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

Then, we will add a **deleteById(id)** function in the **api/games.js** file:



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



After you have made these changes, try deleting the game that we created and edited.

Congratulations! You managed to successfully implement all of the features for our SPA application! 😊