Exercise: Automated Testing

Exercises for the "Software Engineering and DevOps" course @ SoftUni.

I. Writing Web UI Tests

You are given a **Web application** (SPA) using JavaScript. The application dynamically displays content, based on user interaction and supports user profiles and CRUD operations, using a REST service.

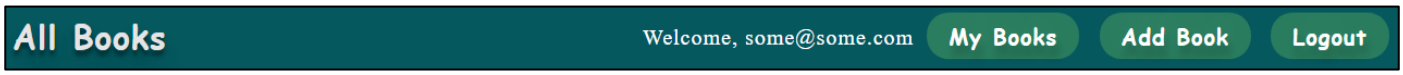
Open the **`Resources`** folder – inside is the **`Library_Catalog`** folder, which contains the necessary files.

1. Application Specifications

Navigation Bar

Guests (un-authenticated visitors) can see the links to the **All Books,** as well as the links to the **Login** and **Register** pages. The logged-in user navbar contains the links to **All Books** page, **My Books** page, the **Add Book** page, **Welcome, {user's email address}**, and a link for the **Logout** action.

User navigation example:



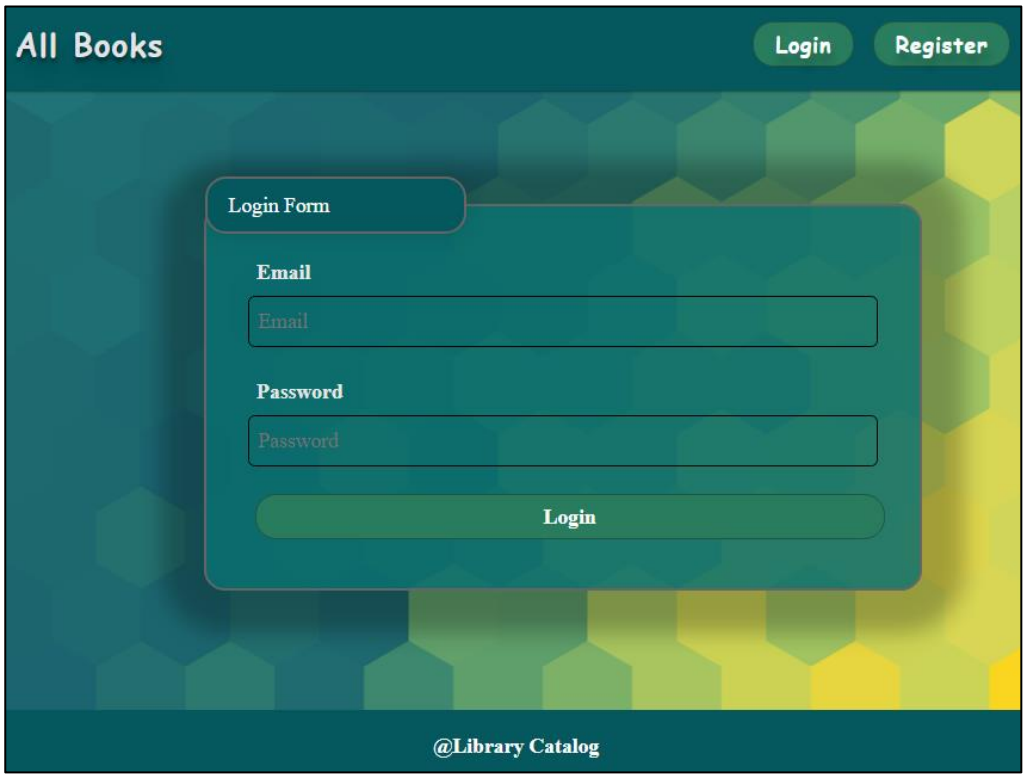
Guest navigation example:

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 logins a user in the system, if there are no empty fields.



The following request is used to perform login:

```
Method: POST
URL: /users/login
```

The service expects a body with the following shape:

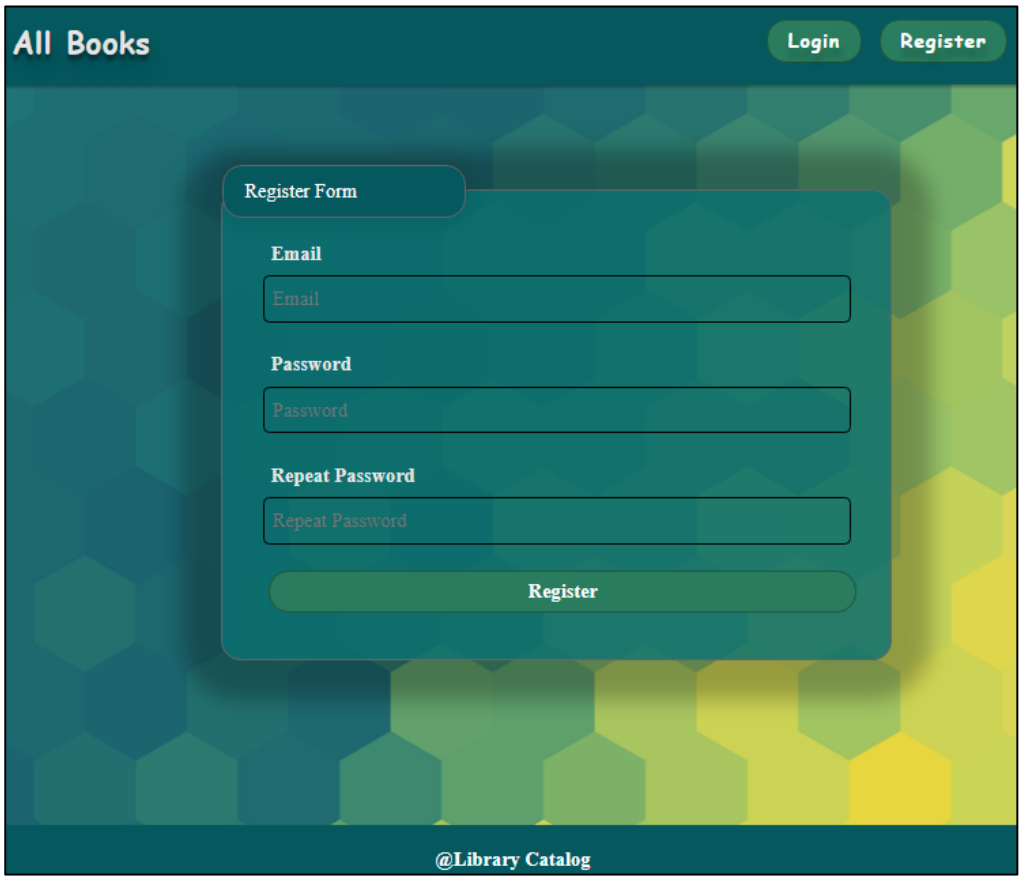
```
{
  email,
  password
}
```

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

If the login is successful, the app **redirects** the user to the **`All Books/Dashboard`** page. If there is an error, or the **validations** don't pass, it displays an error message, using a system dialog (**`window.alert`**).

Register User

By given **email and password**, the app registers a new user in the system. All fields are required – if anyone of them is empty, the app displays an error.



The following request is used to perform login:

```
Method: POST
URL: /users/register
```

The service expects a body with the following shape:

```
{
  email,
  password
}
```

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

If the registration is successful, the app **redirects** the user to the **`All Books/Dashboard`** page. If there is an error, or the **validations** don't pass, it displays an error message, using a system dialog (**`window.alert`**).

Logout

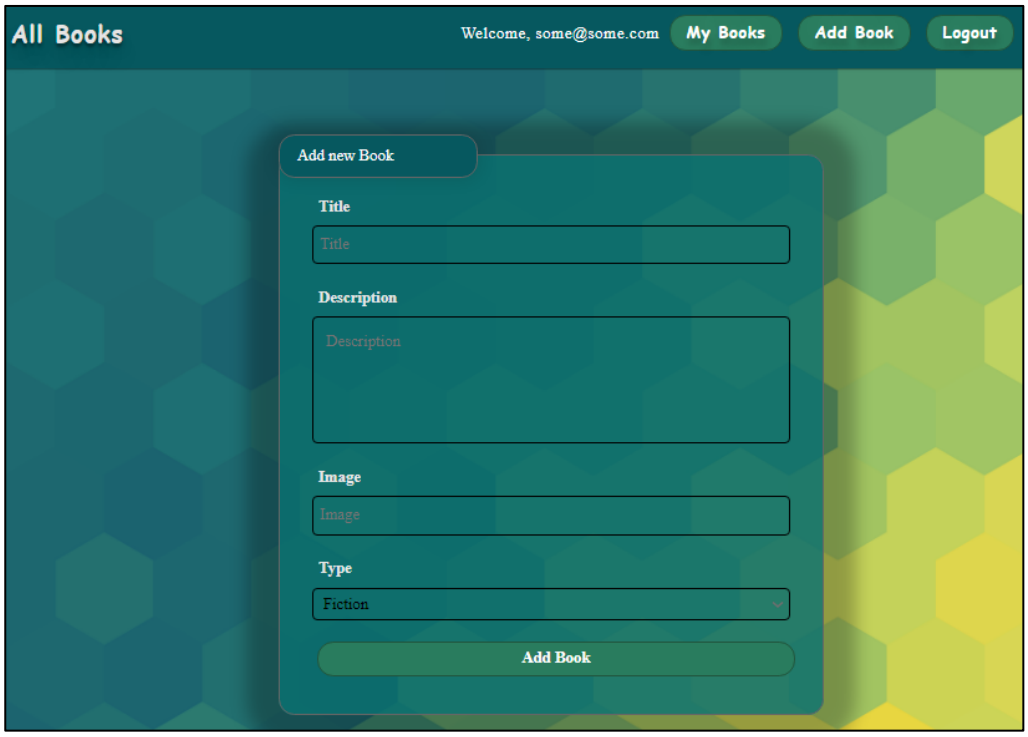
The logout action is available to **logged-in users.** If the logout is successful, the app **redirects** the user to the **`All Books/Dashboard`** page.

The app sends the following **request** to perform logout:

```
Method: GET
URL: /users/logout
```

Add Book

The **Add Book** page is available to **logged-in users**. It contains a form for adding a new book. Checks if all the fields are filled before you send the request.



The following request is being sent:

```
Method: POST
URL: /data/books
```

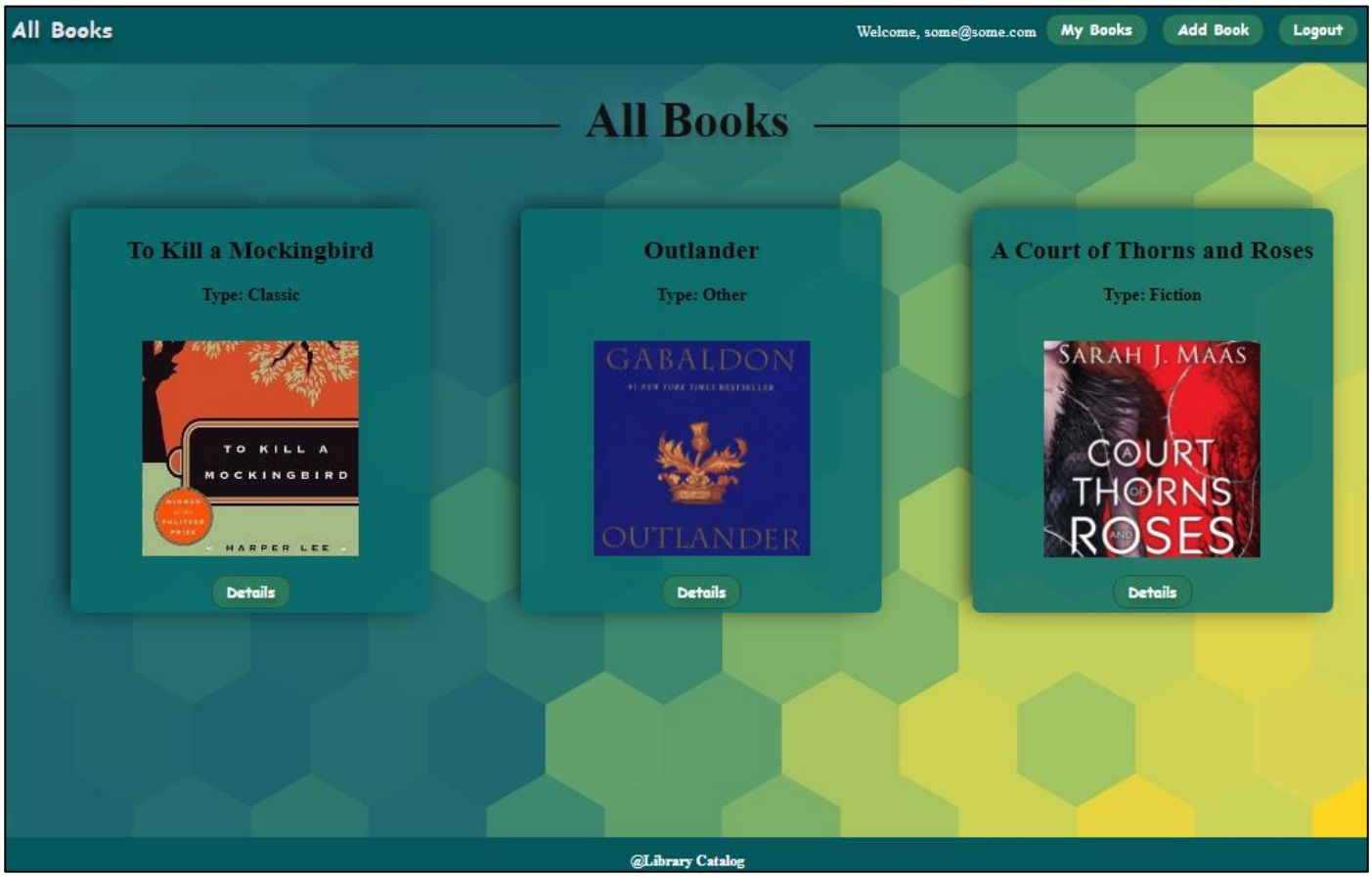
The service expects a body with the following shape:

```
{
  title,
  description,
  imageUrl,
  type
}
```

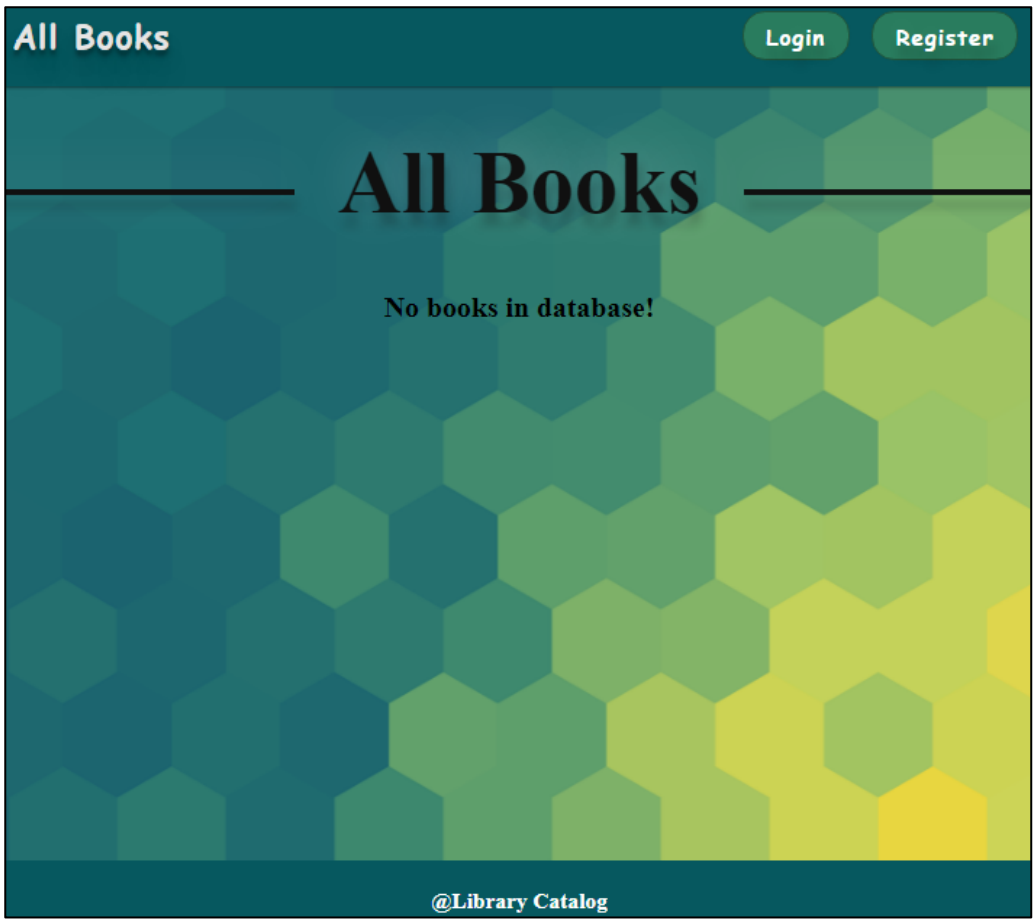
The service returns the newly created record. Upon success, it **redirects** the user to the **`All Books/Dashboard`** page.

All Books

This page **displays** a list of **all books** in the system. Clicking on the details button in the cards leads to the details page for the selected book. This page should be visible to **guests and logged-in users**.



If there are **no books**, the following view is be displayed:



The following **request** is used to read the list of books:

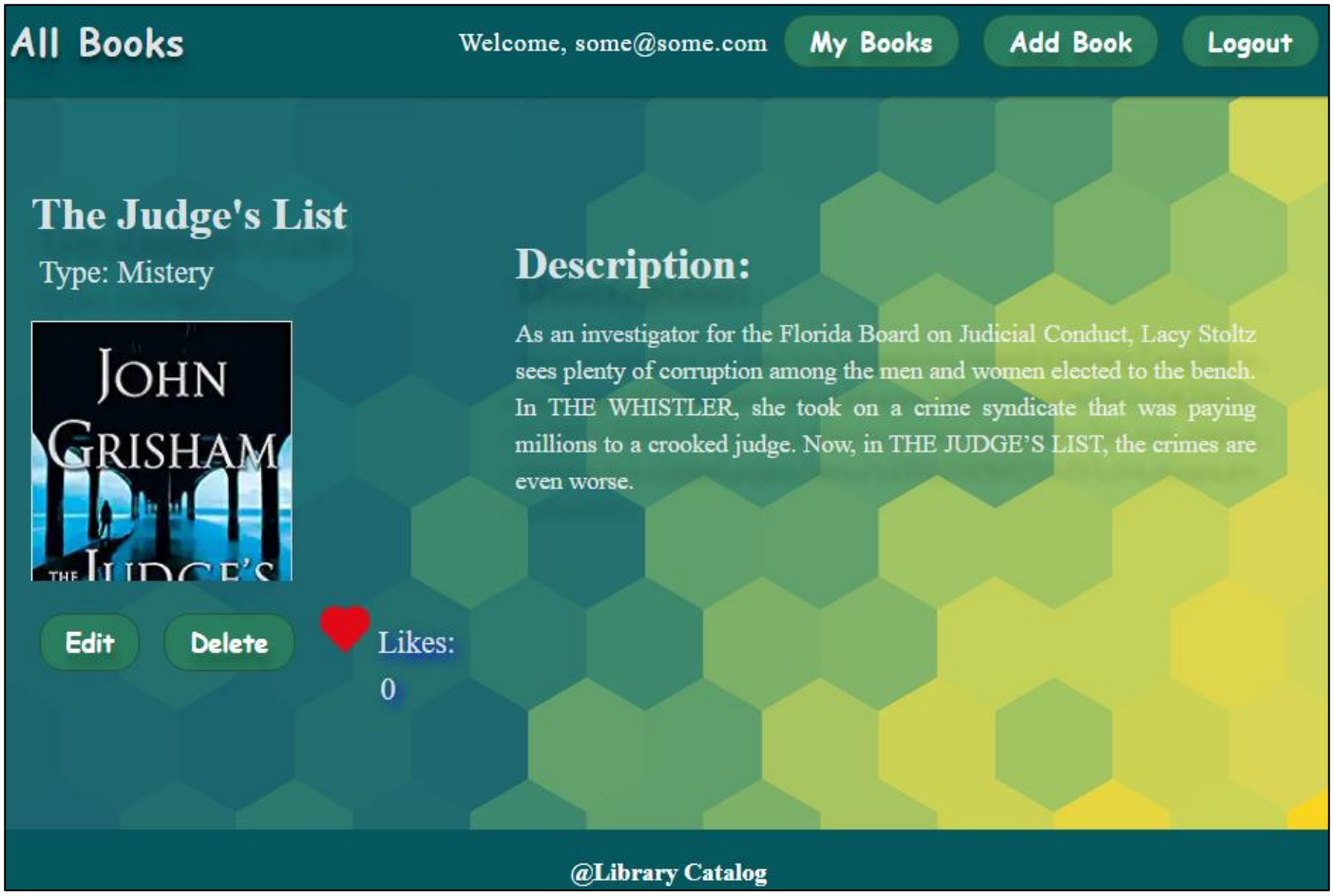
```
Method: GET
URL: /data/books?sortBy=_createdOn%20desc
```

The service returns an **array of books**.

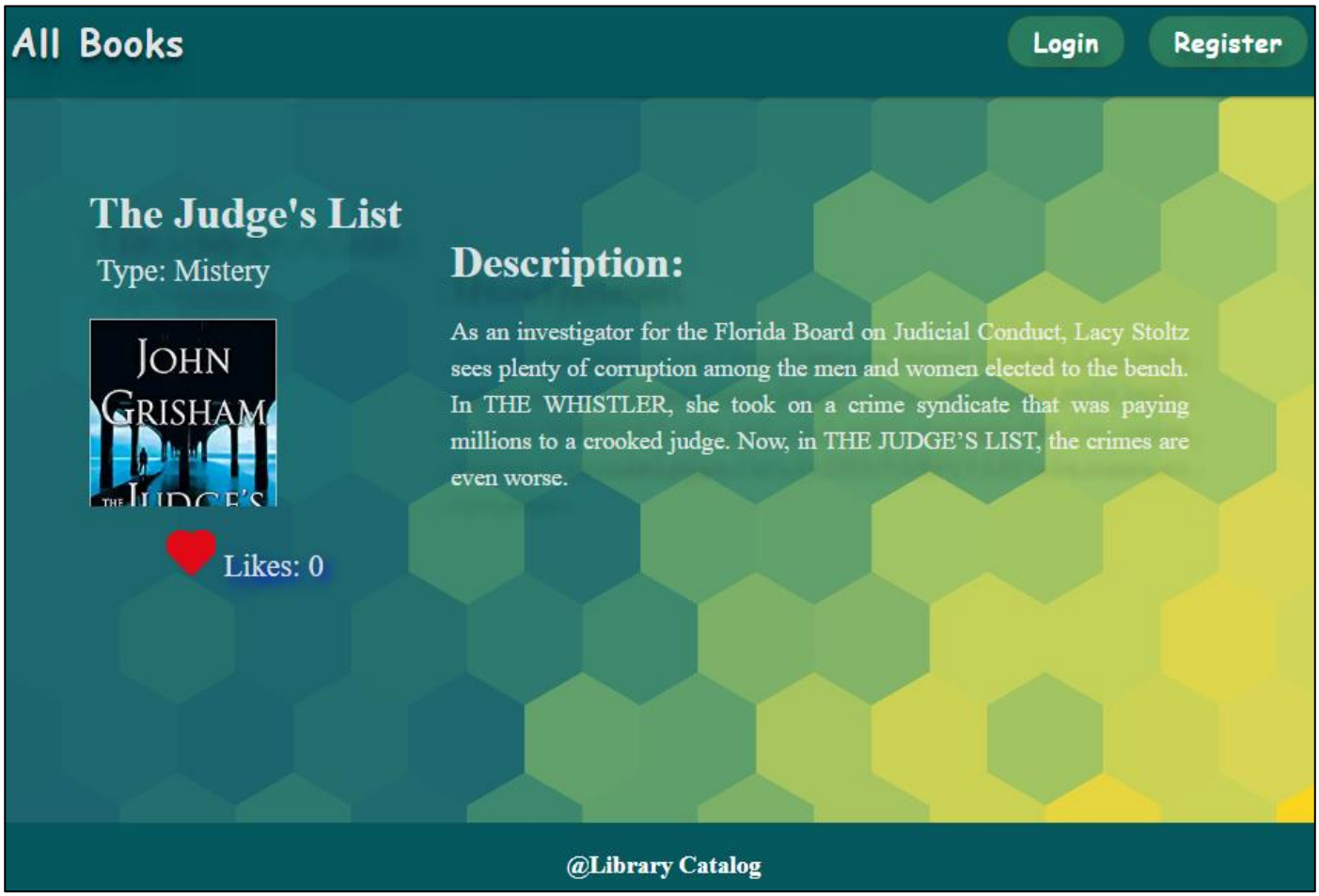
Book Details

All users are able to **view details** about books. Clicking the **Details** link in of a **book card displays** the **Details** page. If the currently **logged-in user** is the **creator**, the [`Edit`] and [`Delete`] buttons are displayed, otherwise they are not available.

The **view for** the **creators** looks like this:



The **view for** the **non-creators** looks like this:



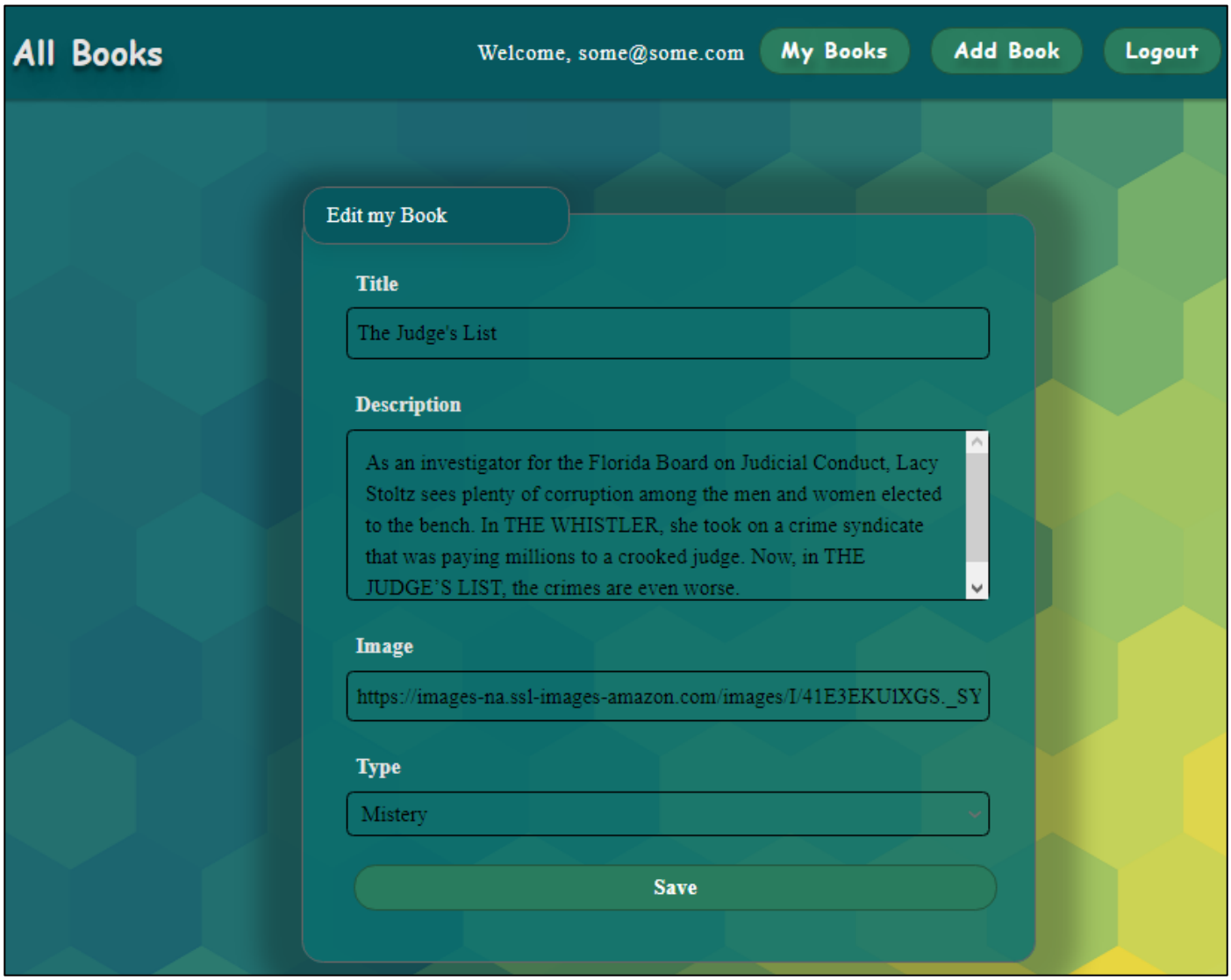
The following **request** is used to read a single book:

```
Method: GET
URL: /data/books/:id
```

id is the **id** of the desired book. The service returns a single object.

Edit Book

The **Edit** page is **available to logged-in users** and it allows **authors** to **edit** their **own** book. Clicking the **Edit** link of a particular book on the **Details** page displays the **Edit** page, with all fields filled with the data for the book. It contains a form with input fields for all relevant properties. Checks if all the fields are filled before you send the request.



To edit a post, the app uses the following **request**:

```
Method: PUT
URL: /data/books/:id
```

id is the **id** of the desired card.

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

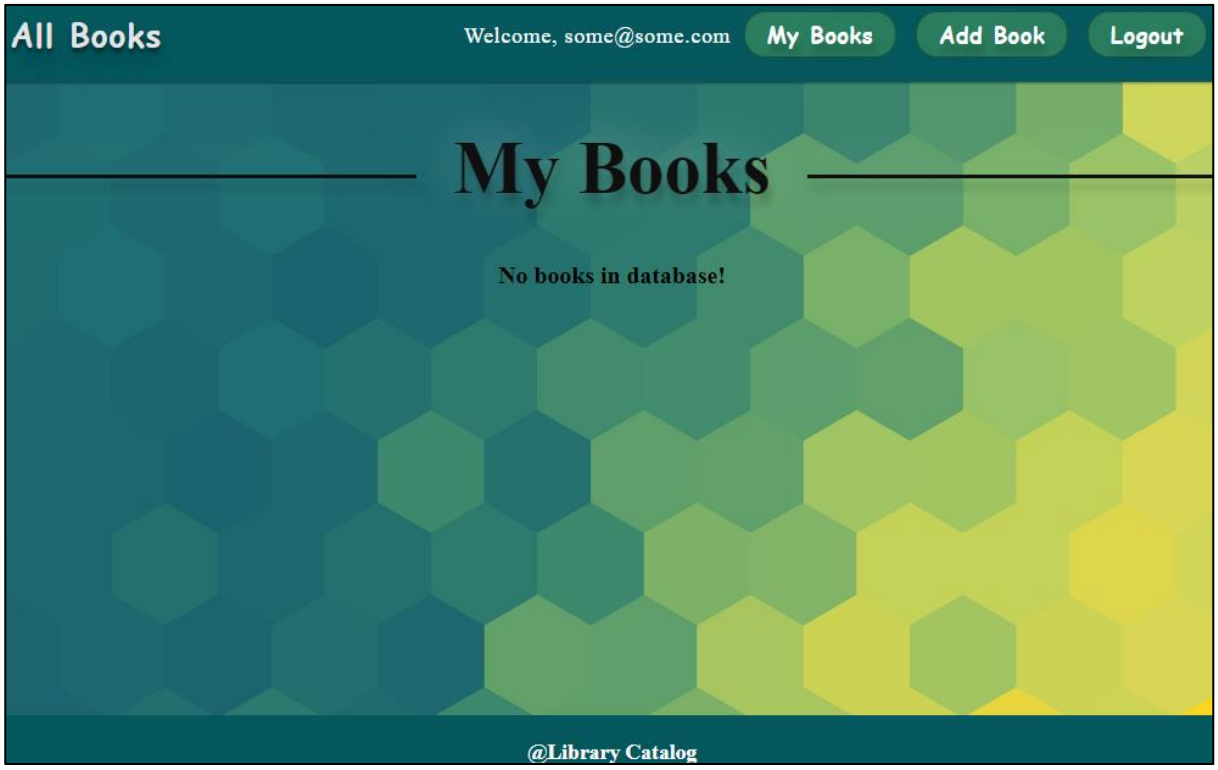
```
{
  title,
  description,
  imageUrl,
  type
}
```

The service returns the modified record. Note that **PUT** requests **do not** merge properties and instead **replace** the entire record. Upon success, it **redirects** the user to the **Details** page for the current book.

Delete Book

The delete action is available to **logged-in users**, **for books they have created**. When the author clicks on the **Delete** action on any of their book, a confirmation dialog is displayed, and upon confirming this dialog, the book is **deleted** from the system and the user is **redirect** to the **All Books/Dashboard** page.

If there are **no books**, the following view is displayed:



To delete a book, the following **request** is being used:

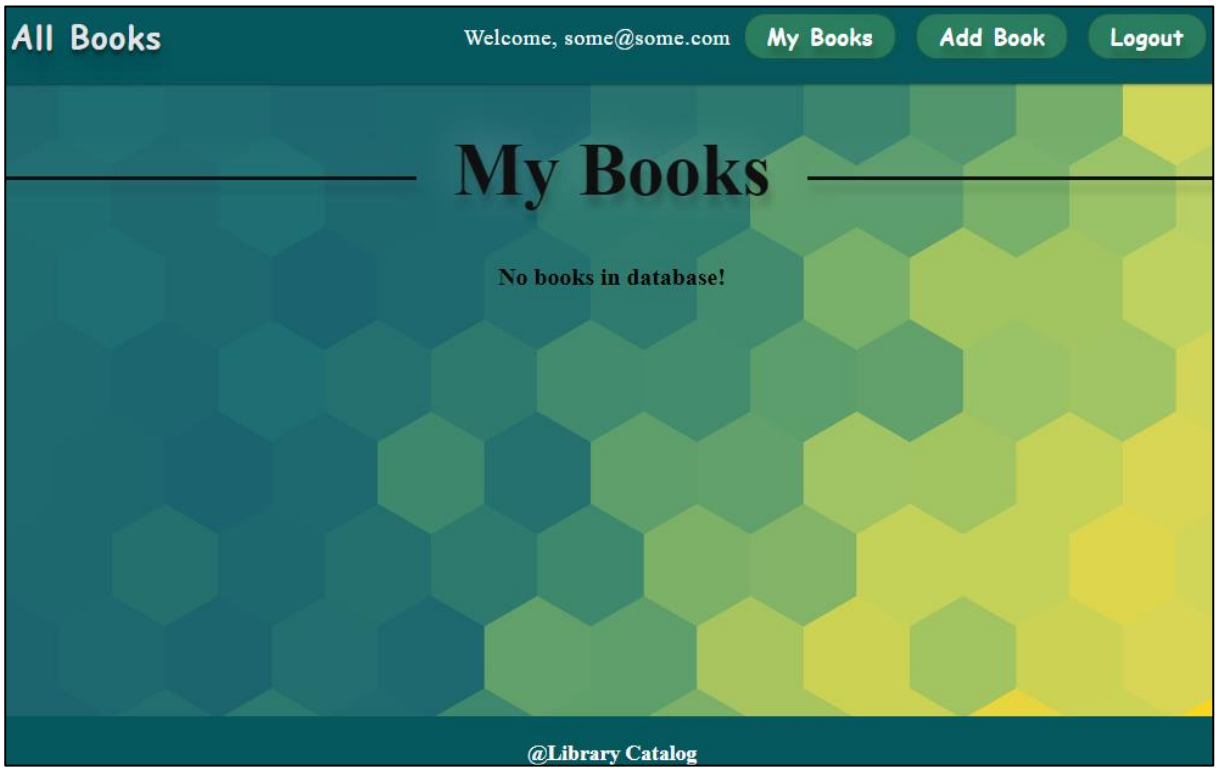
```
Method: DELETE
URL: /data/books/:id
```

id is the **id** of the desired book. The service returns an object, containing the deletion time. Upon success, it **redirects** the user to the **All Books/Dashboard** page.

My Books

Each **logged-in user is** able to **view their own books** by clicking **[My Books]**. Here are listed all books added by the current user.

If there are **no books**, the following view is displayed:



The following request is used to read the list of books:

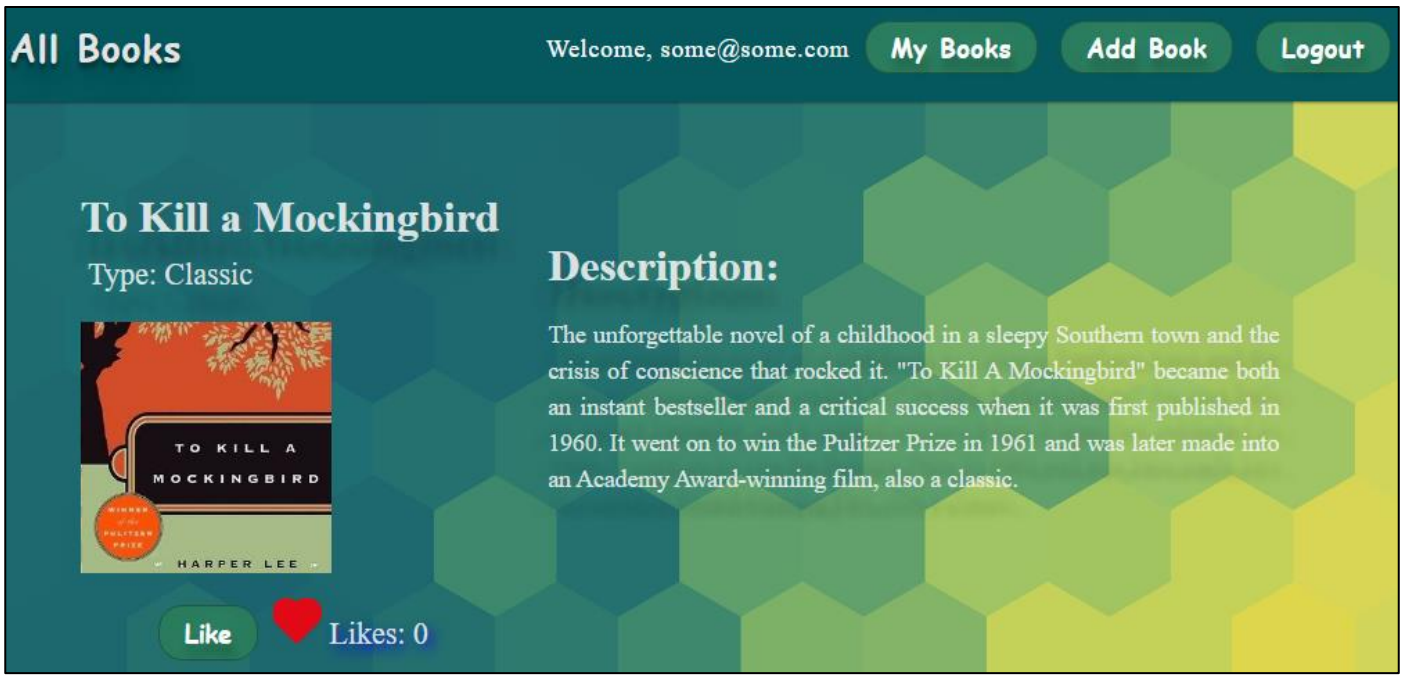
```
Method: GET
URL: /data/books?where=_ownerId%3D%22{userId}%22&sortBy=_createdOn%20desc
```

Where **`{userId}`** is the **id** of the **currently logged-in user**. The service returns an array of books.

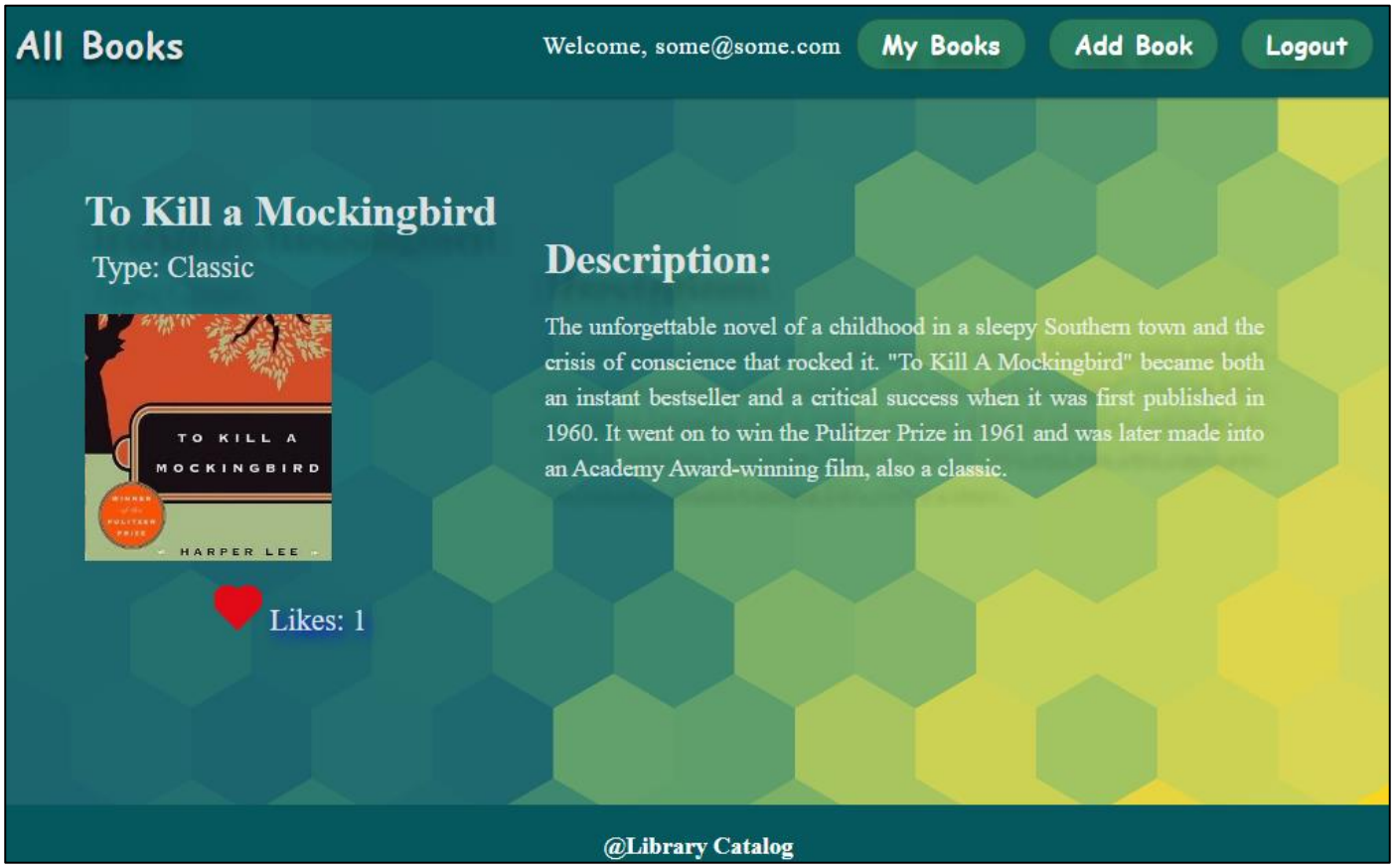
Like a Book

Every **logged-in user** is able to **like other books**, but **not his own**. By clicking on the **`[Like]`** button, **the counter of each book increases by 1**.

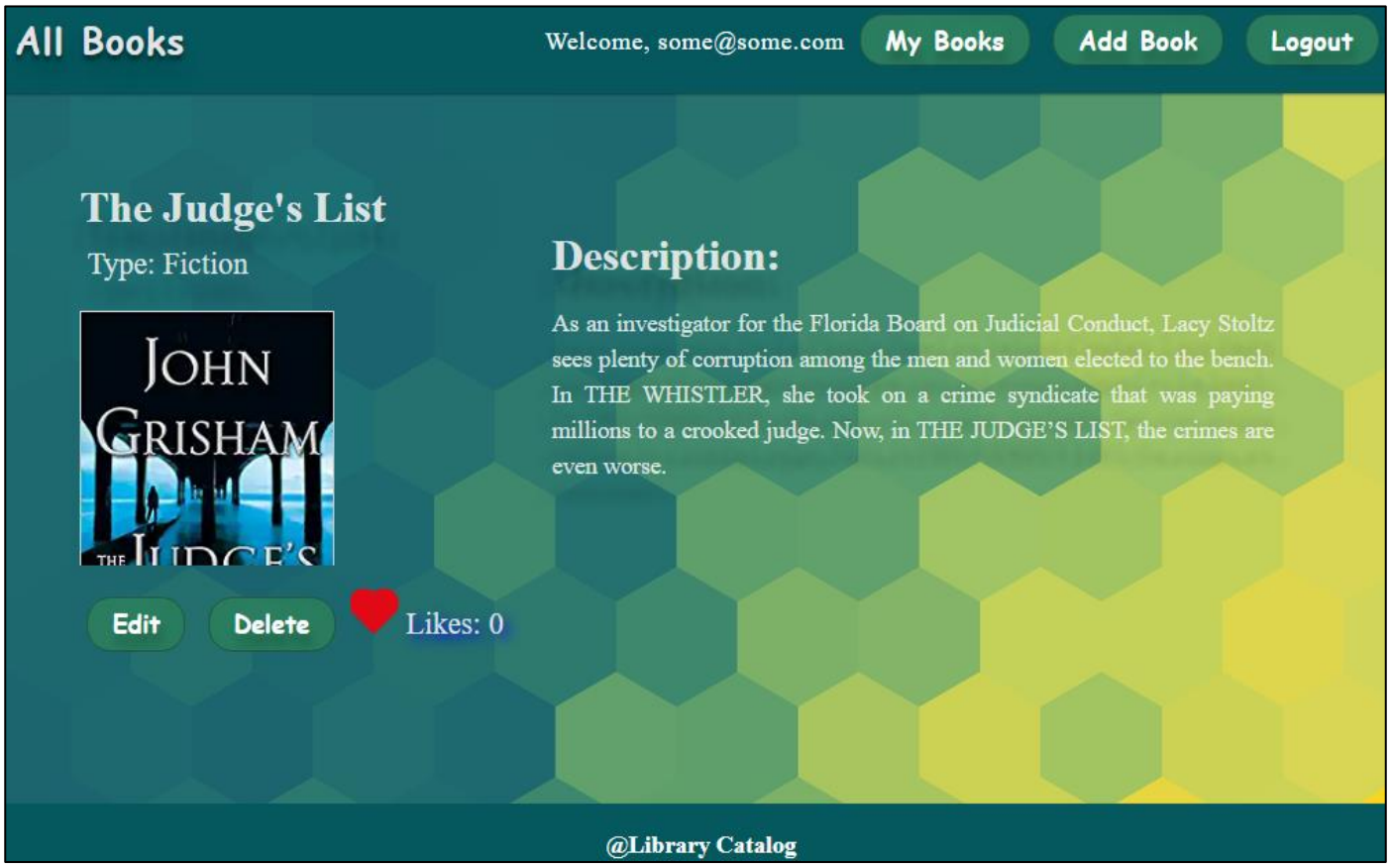
The view when the user did not press **`[Like]`** button looks like:



When the user **liked** the book, the **[Like]** button is **not** available and the counter is **increased** by **1**.



Creator is not able to see the **[Like]** button. The view looks like:



Guest are not able to see the **[Like]** button. The view for **guests** looks like:

To add a like to a book, the following **request** is being used:

```
Method: POST
URL: /data/likes
```

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

```
{
  bookId
}
```

The service returns the newly created record.

The app uses the following **request to get total likes count for a book**:

```
Method: GET
URL: /data/likes?where=bookId%3D%22{bookId}%22&distinct=_ownerId&count
```

Where **{bookId}** is the **id** of the desired book. Required **headers** are described in the documentation. The service will return the **total likes** count.

The app uses the following **request to get like for a book from specific user**:

```
Method: GET
URL:/data/likes?where=bookId%3D%22{bookId}%22%20and%20_ownerId%3D%22{userId}%22&count
```

Where **{bookId}** is the **id** of the desired book and **{userId}** is the **id** of the **currently logged-in user**.
The service returns either **0** or **1**. Depends on that result the **[Like]** button should be displayed or not.

2. Preparing the Environment

To initialize the SPA, execute the following commands in the Visual Studio Code terminal:

```
npm install
```

To install the **playwright/test** framework, write the following command:

```
npm install -D @playwright/test
```

Then, start the server by using the following command:

```
npm run start
```

After that, a web browser should open and you should be able to see the home page of the app.

Then go to the **server folder** in the project directory, open a CLI there and type this command:

```
node server.js
```

After that, create a new folder, named "**tests**" and in it a new file, named "**ui.test.js**". This file will contain the UI tests that we'll write using **Playwright** and **playwright/test**. Create another **api.test.js** file that will contain our **Mocha** tests.

NOTE: To execute the command for running the Playwright tests, just open a new terminal in Visual Studio code and write the following:

```
npx playwright test tests/ui.test.js
```

3. Write Web UI Tests

Your task is to write Web UI tests, using Playwright, in order to test the UI of the given app.

Navigation Bar for Guest Users

Let's write test for the navigation bar that should be displayed for guest users. The tests are listed below.

Verify That the "All Books" Link Is Visible

We have to verify that the "**All Books**" link is visible. To do that, we have to import the **test** and **expect** functions from the **@playwright/test** package. These functions are used for defining tests and making assertions within the tests:

```
const { test, expect } = require('@playwright/test');
```

Then, we have to define a test case using the **test** function. The test case is named "**Verify 'All Books' link is visible**". It is an asynchronous function that takes an object destructuring argument with **page** as one of the properties. **page** represents the browser page that will be used for the test:

```
test('Verify "All Books" link is visible', async ({ page }) => {
```

After that, we have to navigate to the **page** to the specified URL, which is **http://localhost:3000**. It opens the web application in the browser:

```
  await page.goto('http://localhost:3000');
```

We have to wait for the navigation bar to load on the web page. Playwright waits until an element matching the CSS selector **nav.navbar** becomes visible on the page:

```
  await page.waitForSelector('nav.navbar');
```

Then, we have to find the element representing the "**All Books**" link on the page. We use the **$** function of the **page** object along with the CSS selector '**a[href="/catalog"]**' to select the link element:

```
  const allBooksLink = await page.$('a[href="/catalog"]');
```

Then, we check whether the "**All Books**" link is visible on the page. We use the **isVisible** function on the **allBooksLink** element to determine its visibility:

```
  const isLinkVisible = await allBooksLink.isVisible();
```

Finally, we make an assertion using the **expect** function. It expects the value of **isLinkVisible** to be **true**. If the value is not **true**, the test will fail:

```
  expect(isLinkVisible).toBe(true);
});
```

Verify That the "Login" Button Is Visible

First, let's define a test case titled "Verify 'Login' button is visible":

```
test('Verify "Login" button is visible', async ({ page }) => {
```

Then, navigate to the specified URL that holds the html element we want to test:

```
  await page.goto('http://localhost:3000');
```

After that, wait for the **nav.navbar** selector to appear on the page and ensure that the navigation bar has loaded before proceeding with the test:

```
  await page.waitForSelector('nav.navbar');
```

Then, we use the **$** function on the page to select the **[Login]** button, based on its selector (**a[href="/login"]**):

```
  const loginButton = await page.$('a[href="/login"]');
```

After that, we use the **isVisible** function to check if the **[Login]** button is visible in the browser. Let's not forget that the **isVisible** function returns a boolean value indicating the visibility of the element:

```
  const isLoginButtonVisible = await loginButton.isVisible();
```

Finally, use the **expect** function to assert that the **[Login]** button is visible. If the expectations are not met, the test will fail:

```
  expect(isLoginButtonVisible).toBe(true);
});
```

Verify That the "Register" Button Is Visible

Try and write a test by yourself which verifies that the **[Register]** button is visible. The test should be similar to the one verifying the visibility of the **[Login]** button.

Navigation Bar for Logged-In Users

Now let's write some tests for the navigation bar that should be displayed for logged-in users. The tests are listed below.

Verify That the "All Books" Link Is Visible

To verify that the "**All Books**" links is visible after login, we should first define the test case:

```
test('Verify "All Books" link is visible after user login', async ({ page }) => {
```

As we are testing the visibility of links in the navigation bar that is displayed to a logged-in user, we should first navigate to the login URL **http://localhost:3000/login**, so that we login:

```
  await page.goto('http://localhost:3000/login');
```

Then, we fill the input fields with the predefined credentials and click on the **[Submit]** button. This way Playwright submits the form with the filled in credentials:

```
  await page.fill('input[name="email"]', 'peter@abv.bg');
  await page.fill('input[name="password"]', '123456');
  await page.click('input[type="submit"]');
```

After that, we retrieve the element, that represents the "**All Books**" link using a selector and check its visibility, using the **isVisibile()** function:

```
  const allBooksLink = await page.$('a[href="/catalog"]');
  const isAllBooksLinkVisible = await allBooksLink.isVisible();
```

Finally, we use the **expect** assertion to compare the visibility of the "**All Books**" link with the expected value **true**:

```
  expect(isAllBooksLinkVisible).toBe(true);
});
```

Verify That the "My Books" Link Is Visible

Try and write a test by yourself which verifies that the "My Books" button is visible after user login. The test should be similar to the one verifying the "**All Books**" link.

Verify That the "Add Book" Link Is Visible

Try and write a test by yourself which verifies that the "**Add Book**" button is visible after user login. The test should be similar to the one verifying the "**All Books**" link.

Verify That the User's Email Address Is Visible

Try and write a test by yourself which verifies that the user's email address is visible. The test should be similar to the one verifying the "**All Books**" link.

Login Page

Now, let's test the login page. The tests are listed below.

Submit the Form with Valid Credentials

Let's write a test that fills the form with valid credentials and submits the form.

We already know how to define the test case and navigate to the login page. After that, we should start filling in the input fields with the predefined credentials, described in the beginning of this document:

```
test('Login with valid credentials', async ({ page }) => {
  await page.goto('http://localhost:3000/login');

  await page.fill('input[name="email"]', 'peter@abv.bg');
  await page.fill('input[name="password"]', '123456');
```

After we're ready with filling the form, we click on the **[Submit]** button in order to submit the form:

```
  await page.click('input[type="submit"]');
```

Finally, we should check if we have been directed to the correct page:

```
  await page.$('a[href="/catalog"]');
  expect(page.url()).toBe('http://localhost:3000/catalog');
});
```

Submit the Form with Empty Input Fields

Now, let's write a test that submits an empty form.

First, define the test case. Then, as in the previous tests, we should first navigate to the login URL and click on the **`[Login]`** button:

```
  await page.goto('http://localhost:3000/login');
  await page.click('input[type="submit"]');
```

Then, we should listen for the **`dialog`** event, which would visualize an alert popup window with the warning message. After that we, should verify that the dialog type is indeed **`alert`** and check if the message is the same as in the **`login.js`** file. Finally, we should click on the **`[OK]`** button in order to dismiss the alert window:

```
  page.on('dialog', async dialog => {
      expect(dialog.type()).toContain('alert');
      expect(dialog.message()).toContain('All fields are required!');
      await dialog.accept();
    });
```

We should also check whether we have been redirected to the right page or not:

```
    await page.$('a[href="/login"]');
    expect(page.url()).toBe('http://localhost:3000/login');
```

Submit the Form with Empty Email Input Field

Try and write a test by yourself which submits the form with an **empty email** field and a **valid password** field.

Submit the Form with Empty Password Input Field

Try and write a test by yourself which submits the form with a **valid email** field and an **empty password field**.

Register Page

Try writing some test by yourself for the "**`Register`**" page. They should be similar to the tests regarding the **`Login`** form. The tests are described below.

Submit the Form with Valid Values

Write a test that submits the form with **valid values for the email and the passwords**. Check if you're redirected to the correct page.

Submit the Form with Empty Values

Write a test that submits the form with **empty input fields for the email and the passwords**. Check if the alert message is the correct and whether you have been redirected or not.

Submit the Form with Empty Email

Write a test that submits the form with **an empty email input field and valid password input fields.** Check if an alert appears and if it has the expected text message.

Submit the Form with Empty Password

Write a test that submits the form with **an empty password input field and valid email input fields.** Check if a popup alert appears and if it has the expected text message.

Submit the Form with Empty Confirm Password

Write a test that submits the form with **an empty confirm password input field and valid email input field and valid password input field**. Check if a popup alert appears and if it has the expected text message.

Submit the Form with Different Passwords

Write a test that submits the form with **valid email, but different passwords in the two password input fields**. Check if a popup alert appears and if it has the expected text message.

"Add Book" Page

Now let's test the "**Add Book**" page.

Submit the Form with Correct Data

Let's write a test that submits the form for adding a book with correct data.

First, we have to go to the "**Login**" page and fill in the predefined email and password credentials and click on the **[Submit]** button in order to be logged-in, as only logged-in users can add books:

```
test('Add book with correct data', async ({ page }) => {
  await page.goto('http://localhost:3000/login');

  await page.fill('input[name="email"]', 'peter@abv.bg');
  await page.fill('input[name="password"]', '123456');

  await Promise.all([
    page.click('input[type="submit"]'),
    page.waitForURL('http://localhost:3000/catalog')
  ]);
```

As you can see from the code above, we also wait for the navigation to the "**Catalog**" page, as all users should be redirected to that page after login.

Then, we have to go to the "**Add Book**" page via the navigation menu link:

```
  await page.click('a[href="/create"]');
```

We then wait for the form to load:

```
  await page.waitForSelector('#create-form');
```

After the form has been loaded, we start filling in the book details with some dummy data:

```
  await page.fill('#title', 'Test Book');
  await page.fill('#description', 'This is a test book description');
  await page.fill('#image', 'https://example.com/book-image.jpg');
  await page.selectOption('#type', 'Fiction');
```

Keep in mind that the Type field is a dropdown menu and you should input only the predefined types – "**Fiction**", "**Romance**", "**Mistery**", "**Classic**" or "**Other**".

After we are finished with filling in the book information, we click on the **[Submit]** button:

```
  await page.click('#create-form input[type="submit"]');
```

Finally, we verify that we're being redirected to the correct page – this is our indicator that the book has been successfully added:

```
  await page.waitForURL('http://localhost:3000/catalog');

  expect(page.url()).toBe('http://localhost:3000/catalog');
});
```

Submit the Form with Empty Title Field

Now let's write a test that verifies the app behavior when we send the form with an empty title field.

We should again first navigate to the Login page, then login with the predefined credentials and go to the **"Add Book"** page:

```
test('Add book with empty title field', async ({ page }) => {
  await

  await
  await

  await Promise.all([
    page.
    page.
  ]);

  await page.

  await page.
```

We start filling the book details, but we leave the title field empty:

```
  await page.fill('#description', 'This is a test book description');
  await page.fill('#image', 'https://example.com/book-image.jpg');
  await page.selectOption('#type', 'Fiction');
```

This time, we have to check if an alert popup window appears, that has the "**`All fields are required!`**" text message. We already know how to do that:

```
  page.on('dialog', async dialog => {
    expect(dialog.type()).toContain('alert');
    expect(dialog.message()).toContain('All fields are required!');
    await dialog.accept();
  });
```

Finally, we check if we're redirected to some other page or if we're navigated to the "**Add Book**" page:

```
  await page.$('a[href="/create"]');
  expect(page.url()).toBe('http://localhost:3000/create');
});
```

Submit the Form with Empty Description Field

Write a similar test, but this time leave the **description** field empty.

Submit the Form with Empty Image URL Field

Write a similar test, but this time leave the **image URL** field empty.

"All Books" Page

Now let's write some tests for the "**`All Books`**" Page.

Verify That All Books Are Displayed

Let's first verify that all added books are being displayed. Of course, we have to first login with one of the predefined users and then navigate to the "**`All Books`**" page:

```
test('Login and verify all books are displayed', async ({ page }) => {
  await page.goto('http://localhost:3000/login');

  await page.fill('input[name="email"]', 'peter@abv.bg');
  await page.fill('input[name="password"]', '123456');

  await Promise.all([
    page.click('input[type="submit"]'),
    page.waitForURL('http://localhost:3000/catalog')
  ]);
```

In order to get the book list, we have to wait for it to load and get it via the CSS class that is assigned to it:

```
  await page.waitForSelector('.dashboard');
```

We then get all of the book elements and verify that they are displayed by checking whether their count is greater than 0:

```
  const bookElements = await page.$$('.other-books-list li');

  expect(bookElements.length).toBeGreaterThan(0);
});
```

Verify That No Books Are Displayed

Now, let's write a test that verifies that when there are no books in the database, a message that says "**`No books in database!`**" is displayed.

First, delete all books – you can do that manually. Then, we'll use the code from the previous test and adjust to check if there is a message with **`no-books`** class that says "**`No Books in the Database!`**":

```
  const noBooksMessage = await page.textContent('.no-books');
  expect(noBooksMessage).toBe('No books in database!');
```

Note that if you delete all books, you should either start the project again following the steps from the beginning of this document, or add the same books after that in order for the next tests to be working properly.

"Details" Page

Now let's focus on the "**Details**" page.

Verify That Logged-In User Sees Details Button and Button Works Correctly

Let's write a test that clicks on the **`[Details]`** button of a book and verifies that the book details page is displayed. **Delete manually all of the test books that you created so far.**

Let's start with logging in the app with the john@abv.bg user.

As you have already guessed, we should navigate to the "**All Books**" page and get the first book (as user john@abv.bg) is its creator:

```
test('Login and navigate to Details page', async ({ page }) => {
  await page.goto('http://localhost:3000/login');

  await page.fill('input[name="email"]', 'peter@abv.bg');
  await page.fill('input[name="password"]', '123456');

  await Promise.all([
    page.click('input[type="submit"]'),
    page.waitForURL('http://localhost:3000/catalog')
  ]);

  await page.click('a[href="/catalog"]');

  await page.waitForSelector('.otherBooks');
```

After we have found the first book, we click on its **[Details]** button:

```
  await page.click('.otherBooks a.button');
```

Then, we have to wait for the "**Details**" page to load:

```
  await page.waitForSelector('.book-information');
```

Finally, we have to verify that the correct "**Details**" page has been opened:

```
  const detailsPageTitle = await page.textContent('.book-information h3');
  expect(detailsPageTitle).toBe('Test Book');
});
```

Verify That Guest User Sees Details Button and Button Works Correctly

Now write a similar test but instead of checking if a logged-in user sees the **[Details]** button, you should check if a guest user sees it.

Verify That All Info Is Displayed Correctly

You should write a similar test (it doesn't matter if you use a Guest user or a Logged-In User). This test should check if all of the book information is correct.

Verify If Edit and Delete Buttons Are Visible for Creator

Only the creator of a book must be able to see the **[Edit]** and **[Delete]** buttons. Think how to write a test to verify the app logic.

As you can guess, we have to first sign in with a user. User peter@abv.bg is the creator of book "**Outlander**". Check if this user sees the **[Edit]** and **[Delete]** buttons.

Verify If Edit and Delete Buttons Are Not Visible for Non-Creator

Only the creator of a book must be able to see the **[Edit]** and **[Delete]** buttons. Think how to write a test to verify the app logic.

As you can guess, we have to first sign in with a user. User john@abv.bg is **NOT** the creator of book "**Outlander**". Check if this user sees the **[Edit]** and **[Delete]** buttons.

Verify If Like Button Is Not Visible for Creator

Only the non-creator of a book must be able to see the **[Like]** button. Think how to write a test to verify the app logic.

Apply the same logic from the previous two tests.

Verify If Like Button Is Visible for Non-Creator

Only the non-creator of a book must be able to see the **[Like]** button. Think how to write a test to verify the app logic.

Apply the same logic from the previous three tests.

"Logout" Functionality

Let's write some tests to check if the logout functionality works correctly.

Verify That the "Logout" Button Is Visible

The logic for this test is similar to the logic for the tests that check whether the "**All Books**" and "**Add Book**" links are visible upon login.

The only difference is that we have to check if we have been directed to the correct page after successful logout:

```
  const logoutLink = await page.$('a[href="javascript:void(0)"]');
```

Verify That the "Logout" Button Redirects Correctly

The last test that we should execute is to verify that the **[Logout]** button redirects us to http://localhost:3000/catalog.

We should login with the predefined credentials, find the **[Logout]** button and retrieve the URL of the page that we were redirected to:

```
test('Verify redirection of Logout link after user login', async ({ page }) => {
  await page.goto('http://localhost:3000/login');

  await page.fill('input[name="email"]', 'peter@abv.bg');
  await page.fill('input[name="password"]', '123456');
  await page.click('input[type="submit"]');

  const logoutLink = await page.$('a[href="javascript:void(0)"]');
  await logoutLink.click();

  const redirectedURL = page.url();
  expect(redirectedURL).toBe('http://localhost:3000/catalog');
});
```

Finally, we can verify that all tests pass using the command, described in the beginning of this document:

```
PS C:\Users\      \Desktop\09-Exercises-Automated-Testing-Resources\Library-Catalog> npx playwright test tests/ui.test.js

Running 11 tests using 1 worker

  ✓  1 tests\ui.test.js:3:1 › Verify "All Books" link is visible (1.6s)
  ✓  2 tests\ui.test.js:14:1 › Verify "Login" button is visible (163ms)
  ✓  3 tests\ui.test.js:26:1 › Verify "All Books" link is visible after user login (242ms)
  ✓  4 tests\ui.test.js:39:1 › Login with valid credentials (216ms)
  ✓  5 tests\ui.test.js:51:1 › Login with empty input fields (198ms)
  ✓  6 tests\ui.test.js:65:1 › Add book with correct data (344ms)
  ✓  7 tests\ui.test.js:92:1 › Add book with empty title field (331ms)
  ✓  8 tests\ui.test.js:123:1 › Login and verify all books are displayed (249ms)
  ✓  9 tests\ui.test.js:141:1 › Login and navigate to Details page (348ms)
  ✓  10 tests\ui.test.js:164:1 › Verify visibility of Logout button after user login (239ms)
  ✓  11 tests\ui.test.js:178:1 › Verify redirection of Logout link after user login (260ms)

  11 passed (5.6s)
```

Feel free to try and think of other tests scenarios and cases and try writing tests for them.

II. Write API Tests for RESTful API

1. Books API

Preparing the Environment

It's now time to write some API tests. Open folder named "**Books**" in Visual Studio. There is a "**server.js**" file that represents a simple server, which handles different HTTP methods (**GET**, **POST**, **PUT**, **DELETE**) and implements basic CRUD operations for books.

Now run the following commands:

```
npm install express
npm install body-parser
npm install mocha
npm install chai
npm install chai-http
node server.js
```

- **Express.js** is a minimal and flexible **Node.js** web application framework that provides a robust set of features for web and mobile applications. It is used to build APIs, handle HTTP requests, and manage routes in your application.
- **body-parser**: Body-parser is a middleware that helps to parse incoming request bodies before your handlers. It provides various functionalities to parse text, JSON data, raw data, and even multipart data.
- **mocha**: Mocha is a JavaScript test framework running on **Node.js**, which makes asynchronous testing simple and fun. It provides functions to structure your tests (like **describe**, **it**, etc.) and a command-line tool to run them.
- **chai**: Chai is a BDD (Behavior-Driven Development) / TDD (Test-Driven Development) assertion library for **Node.js** and the browser that can be paired with any JavaScript testing framework. It provides functions to assert that certain conditions are met in your code (like **expect**, **should**, etc.).
- **chai-http**: Chai-HTTP is a plugin for Chai, which provides an interface for HTTP assertions. It's useful for testing APIs: you can send HTTP requests to your server from your tests and use Chai assertions to check the results.

NOTE: To execute the command for running the Mocha tests, just open a new terminal in Visual Studio code and write the following:

```
npx mocha api.test.js
```

Now, create a new file "**api.test.js**" in the folder. It will hold our API tests.

Write API Tests

We'll start by setting up the necessary dependencies and configurations like importing the **Chai** library, the **Chai HTTP plugin** and the **server** module.

```
const chai = require('chai');
const chaiHttp = require('chai-http');
const server = require('./server');
const expect = chai.expect;

chai.use(chaiHttp);
```

Verify Posting a Book

Our first test is the test for successful creation of a book by making a **POST** request to the **/books** endpoint of the API server and checking the response for expected properties and values. The **bookId** is also captured for potential use in subsequent test cases.

First, we set a test suite with the **describe** function from the Mocha framework and then we declare a variable **bookId** to store the id of the book created during the test:

```
describe('Books API', () => {
    let bookId;
```

Then, we define our first test case with the **it** function from Mocha. It represents an individual test scenario. In this case, it tests the functionality of posting a book to the API.

Then, we define a **book** object with properties such as **id**, **title**, and **author**. It represents the book that will be sent as a **POST** request payload to the API:

```
    it('should POST a book', (done) => {
        const book = { id: "1", title: "Test Book", author: "Test Author" };
```

After that, we use Chai HTTP to make a **POST** request to the **/books** endpoint of the API server. It sends the **book** object as the request payload:

```
        chai.request(server)
            .post('/books')
            .send(book)
            .end((err, res) => {




            });
```

Then, we start making assertions that the response has a status code of 201, indicating a successful creation of the book resource, that the response body is an object, that the response body has properties called **id**, **title** and

author. Finally, we assign the value of the **id** property from the response body to the **bookId** variable that we declared earlier and we signal the completion of the test case by calling **done()**:

```
        chai.request(server)
            .post('/books')
            .send(book)
            .end((err, res) => {
                expect(res).to.have.status(201);
                expect(res.body).to.be.a('object');
                expect(res.body).to.have.property('id');
                expect(res.body).to.have.property('title');
                expect(res.body).to.have.property('author');
                bookId = res.body.id;
                done();
            });
```

You can run the test and check if it is successful.

Verify Getting All Books

Now, let's write a test that verifies the successful getting of books.

First, we should define the test case using the **it** function from Mocha. Then, using Chai HTTP we should make a **GET** request to the **/books** endpoint. After we receive a response, we should assert that the response has a status code of **200** (indicates successful request), that the response body is an array. Finally, we should call the **done()** function to signal the completion of the test:

```
    it('should GET all books', (done) => {
        chai.request(server)
            .get('/books')
            .end((err, res) => {
                expect(res).to.have.status(200);
                expect(res.body).to.be.a('array');
                done();
            });
    });
```

Verify Getting a Single Books

Now, let's write a test that verifies the successful getting of one single book.

We start with defining the test case, using the **it** function and declare a **bookId** variable that we'll use later. Then, we use Chai HTTP to make a **GET** request to the **/books/{bookId}** endpoint of the API server, where **{bookId}** is the value stored in the **bookId** variable.

When we receive a response from the GET request, we start asserting that:

- the response body is an object;
- the response body has a property **id**;
- the response body has a property **title**;
- the response body has a property **author**.

Finally, we should call the **done()** function to signal the completion of the test:

```
    it('should GET a single book', (done) => {
        const bookId = 1;

        chai.request(server)
            .get(`/books/${bookId}`)
            .end((err, res) => {
                expect(res).to.have.status(200);
                expect(res.body).to.be.a('object');
                expect(res.body).to.have.property('id');
                expect(res.body).to.have.property('title');
                expect(res.body).to.have.property('author');
                done();
            });
    });
```

Verify Updating a Book

Let's write a test that verifies updating a book. To update a book, we use the **PUT** request.

We begin by defining the test, using the **`it`** function. Inside the test case, we declare a **`const`** variable called **`updatedBook`**, representing the updated book data.

Using Chai HTTP, we make a PUT request to the **`/books/{bookId}`** endpoint of the API server, where **`{bookId}`** is the specific book's ID being updated. The **`updatedBook`** object is sent as the payload of the request.

When the **PUT** request receives a response, we perform a series of assertions to verify the expected results:

- The response is expected to have a status code of **`200`**, indicating a successful update;
- The response body is expected to be an object;
- The **`title`** property in the response body is expected to be equal to the updated title specified in the **`updatedBook`** object;
- The **`author`** property in the response body is expected to be equal to the updated author specified in the **`updatedBook`** object;

Finally, we call the **`done()`** function to signal the completion of the asynchronous test case:

```
    it('should PUT an existing book', (done) => {
        const bookId = 1;
        const updatedBook = { id: bookId, title: "Updated Test Book", author: "Updated Test Author" };
        chai.request(server)
            .put(`/books/${bookId}`)
            .send(updatedBook)
            .end((err, res) => {
                expect(res).to.have.status(200);
                expect(res.body).to.be.a('object');
                expect(res.body.title).to.equal('Updated Test Book');
                expect(res.body.author).to.equal('Updated Test Author');
                done();
            });
    });
```

Verify Deleting a Book

Try writing a test by yourself, that verifies successful deletion of a book. The expected status is **`204`**.

Verify Non-Existing Book

Now, let's write a test for the test case that checks the expected behavior when attempting to perform **`GET`**, **`PUT`**, or **`DELETE`** operations on a non-existing book in the API.

The test starts by using the **`it`** function to define the test case description.

Within the test case, we make three separate requests to the API server, each representing a different operation: GET, PUT, and DELETE. In each request, we use a non-existing **`bookId`** (**`9999`**).

For the first request, we perform a **`GET`** operation by making a request to the **`/books/9999`** endpoint. We then check the response, using the expect function to ensure that the status code is **`404`**, indicating that the book was not found.

Next, we perform a **`PUT`** operation by making a request to the **`/books/9999`** endpoint. We send an object representing the non-existing book data as the payload. Similar to the previous request, we check the response to confirm that the status code is **`404`**, indicating that the book could not be found for updating.

Finally, we perform a **`DELETE`** operation is performed by making a request to the **`/books/9999`** endpoint. Again, we check the response to ensure that the status code is **`404`**, indicating that the book could not be found for deletion.

Finally, we call the the **`done()`** function is called to signal the completion of the test:

```
    it('should return 404 when trying to GET, PUT or DELETE a non-existing book', (done) => {
        chai.request(server)
            .get('/books/9999')
            .end((err, res) => {
                expect(res).to.have.status(404);
            });

        chai.request(server)
            .put('/books/9999')
            .send({ id: "9999", title: "Non-existing Book", author: "Non-existing Author" })
            .end((err, res) => {
                expect(res).to.have.status(404);
            });

        chai.request(server)
            .delete('/books/9999')
            .end((err, res) => {
                expect(res).to.have.status(404);
                done();
            });
    });
});
```

If we run the tests, they should all pass successfully:

```
PS C:\Users\     \Desktop\Books> npx mocha api.test.js
Server is up and running


  Books API
    ✓ should POST a book
    ✓ should GET all books
    ✓ should GET a single book
    ✓ should PUT an existing book
    ✓ should return 404 when trying to GET, PUT or DELETE a non-existing book


  5 passing (48ms)
```