Exercise: Automated Testing

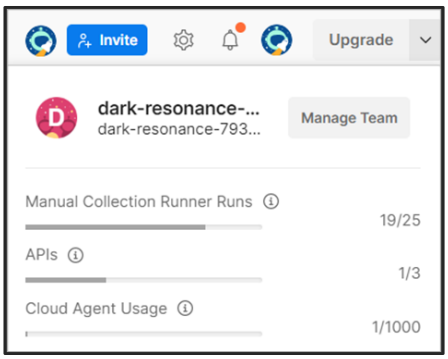
Exercises for the "Software Engineering and DevOps" module @ SoftUni

I. Postman and Newman

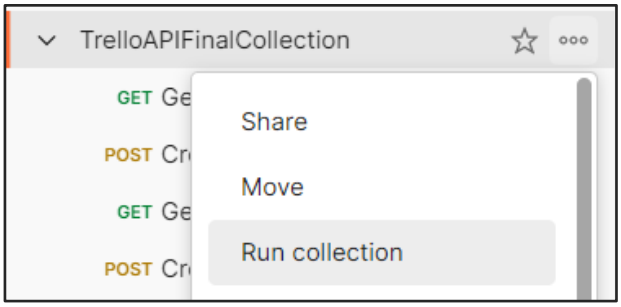
1. Postman Runner

For a long time, the Collection Runner was a free tool with no usage limits. However, starting in March 2023, Postman introduced significant restrictions **on how many times you can use the Collection Runner each month**. On **the free plan**, you're **limited** to **25 runs per month**. A run counts each time you press the **[Run]** button, regardless of how many requests your collection has. As you approach this limit, you'll see a warning, and eventually, you'll reach a soft limit. If you keep using the tool beyond this point, you'll hit a hard limit. To avoid reaching the hard limit and making the Collection Runner unusable, try to minimize how often you execute your collections.

You can check how many runs do you have left from the [Upgrade] button:



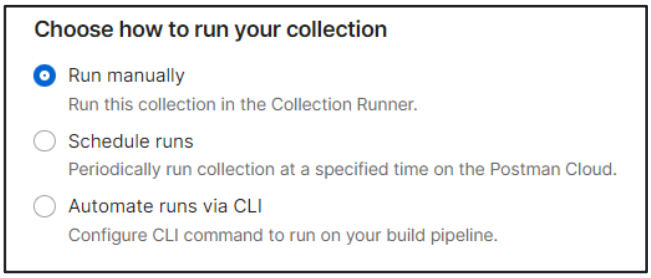
The **Collection Runner** is a tool that is **built into Postman** and allows us to **execute the entire collection with just a single click** instead of manually going over each individual request and clicking the send button. You can execute your API test collections within Postman by **selecting the Run collection option from the collection menu.**



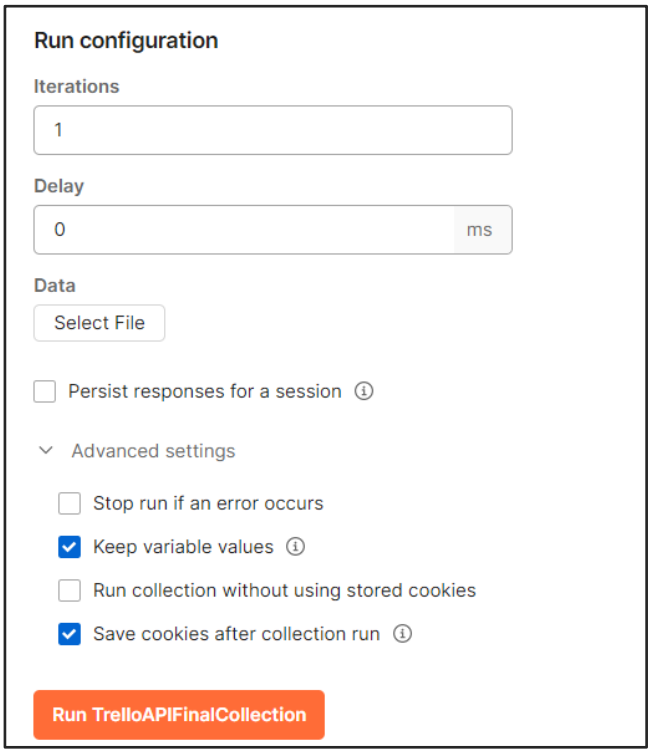
The Collection Runner opens in a separate tab. Notice the run order first, which shows how requests will be executed in the same sequence as they appear in the collection. If there's a request you prefer not to run, simply disable it. If you're not satisfied with the execution order, you can easily change it by dragging the request to a new spot. In our case, the current order is perfectly fine.

We have few options here:

- **Run Manually**
- **Schedule runs**
- **Automate runs via CLI**



1.1. Running the Collection Manually



Iterations

Determines how many times the collection runs. If you input "2", the collection will run twice. Usually, one iteration sufficient.

Delay

Manages the timing between requests, like allowing time for a board to become available after creation, add a delay. This can prevent failures in subsequent requests.

Persist responses for a session

This saves the responses from your API requests until you close the session, helping you to check them later for any issues.

Keep variable values

This keeps the new values of any variables you change during the run for use later in the same session.

Run collection without using stored cookies

This runs your requests without any cookies saved before, making sure your tests are clean.

Save cookies after collection run

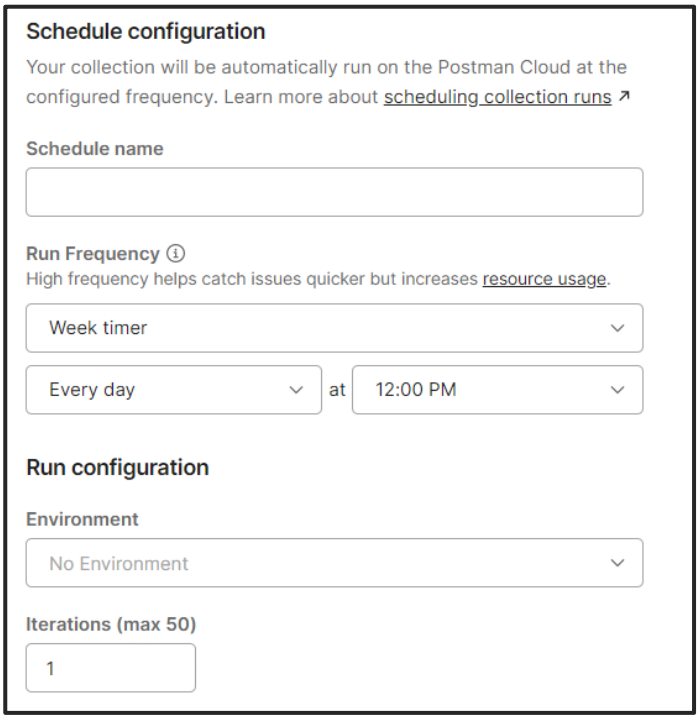
This saves new cookies from your run to use in future sessions or tests.

When you are running the collection:

- **Utilize Postman Console for Troubleshooting**
  - Before running the collection, clear any existing logs in the Postman console to streamline troubleshooting. Console logs are invaluable for debugging, showing executed requests and logged statements.
- **Inspect Request Details**
  - After running the collection, you can examine each request for additional insights, such as the request URL and response body. If certain details are missing, like the request body, refer to the Postman console for a complete overview.
- **Review and Debug**
  - The Collection Runner provides a preliminary step towards automation by offering a semi-automatic way to run collections. It generates a report detailing which requests succeeded or failed, aiding in troubleshooting.
- **Ensure Collection Runs Smoothly Before Automating Further**
  - Verify that your collection runs correctly with the Collection Runner before moving on to more advanced automation tools. If there are issues at this stage, it's easier to debug within Postman.

1.2. Schedule runs

To set up a scheduled run for a collection in Postman, which will run automatically without you needing to click anything:



In Postman, the 'Schedule runs' feature is designed for automating collections to run at specified intervals:

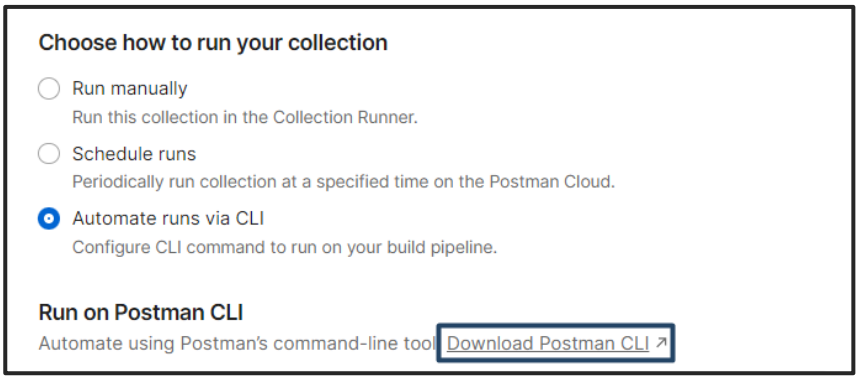
- You can set up automation by choosing 'Schedule runs' in the collection options.
- This allows you to name your schedule, like 'Daily API Check', and pick the frequency, such as daily at noon.
- The automation executes on Postman's cloud, so your local machine's status won't affect it.
- To view or manage these runs, you'd navigate to the 'Runs' tab in the collection's settings.
- It's important that the scheduled runs are configured with the correct initial variable values since they depend on these for proper execution.

However, keep in mind that on the free plan, Postman limits you to 25 collection runs per month, so while setting this up, you should be cautious not to exhaust your allowed runs.

2. Postman CLI

The Postman CLI is an **integrated** tool within Postman that enables the **automation** of collection runs through the command line.

2.1. Installation and Basic Usage



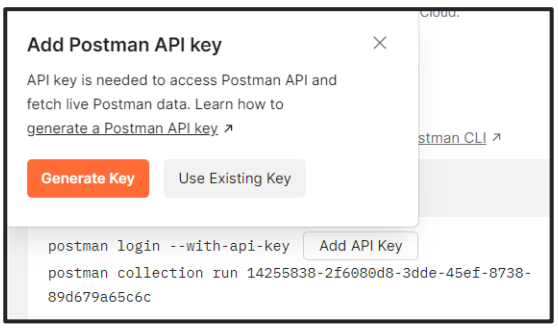
This will open a new tab. Choose your operation system and copy the command for your operating system in your terminal and this will install Postman CLI:

```
PS C:\Users\mddim> powershell.exe -NoProfile -InputFormat None -ExecutionPolicy AllSigned
-Command "[System.Net.ServicePointManager]::SecurityProtocol = 3072; iex ((New-Object Syst
em.Net.WebClient).DownloadString('https://dl-cli.pstmn.io/install/win64.ps1'))"
The Postman CLI has been installed
PS C:\Users\mddim>
```

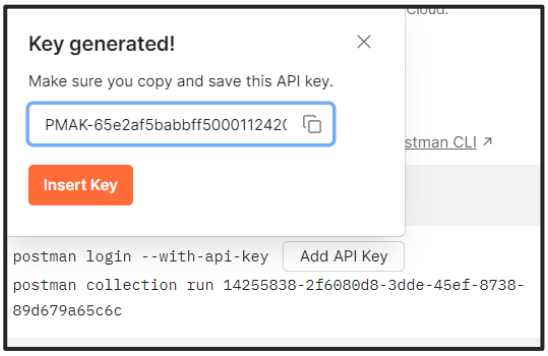
After the installation, you can simply type **`postman`** in your terminal and it will give you some ideas of how we can use Postman:

```
PS C:\Users\mddim> postman
Usage: postman [options] [command]

Options:
  -v, --version      output the version number
  -h, --help         display help for command

Commands:
  collection         Run and test your Postman collections directly from the command line.
  api                Publish and test your APIs directly from the command line
  login [options]    Store a Postman API Key to access your Postman resources
  logout [options]   Delete the stored Postman API key.

To get available options for a command:
  postman <command> -h
```

After installing Postman CLI, you'll need an API key to access your Postman collections. So, let's create one:



Name your API key, after that **copy** and **paste it somewhere safe**. It is **shown** only **once**:



When you click the **`[Insert]`** button, it will add the API key to the command, so you need to copy the commands and run them in your local terminal:

The first command logs you in with the API key and the second one runs the collection:

```
PS C:\Users\mddim> postman login --with-api-key PMAK-65e2af5babbff5000112420f-1dca04356fd
Logged in using api key of user: avionics-cosmonaut-38
Logged in successfully.
```

```
PS C:\Users\mddim> postman collection run 14255838-2f6080d8-3dde-45ef-8738-8
postman
```

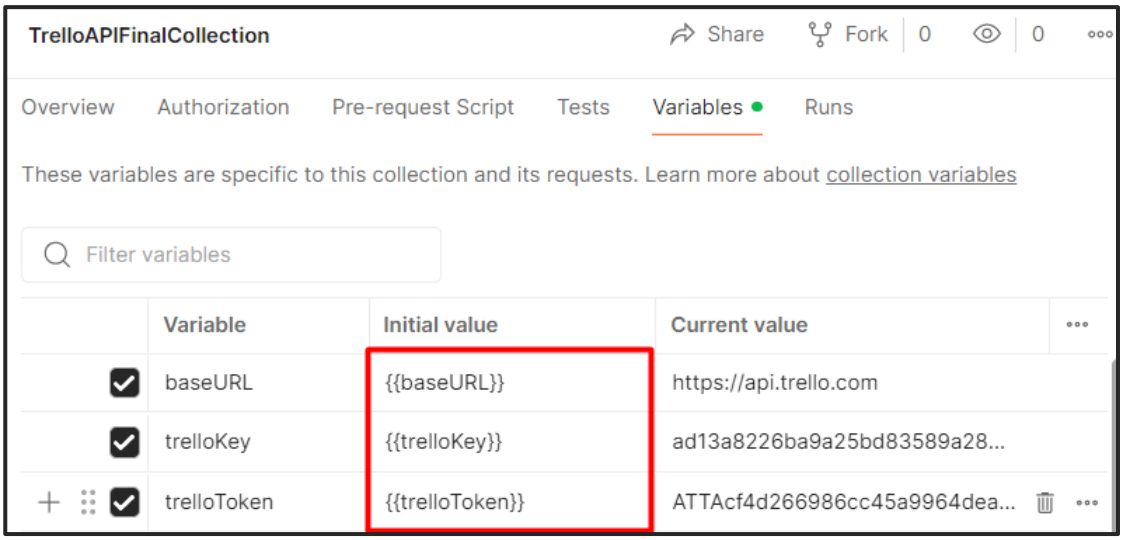
But all of our tests are failing. Why is that?

	executed	failed
iterations	1	0
requests	10	9
test-scripts	20	8
prerequest-scripts	12	0
assertions	15	15
total run duration: 1509ms		
total data received: 63B (approx)		
average response time: 32ms [min: 296ms, max: 296ms, s.d.: 93ms]		

We can see here on our first request that Postman doesn't recognize what **`{{baseURL}}`** and doesn't have access to our **`trelloKey`** and **`trelloToken`**.

```
TrelloAPIFinalCollection

→ Get All Boards
  GET {{baseURL}}/1/members/me/boards?key={{trelloKey}}&token={{trelloToken}} [errored]
     getaddrinfo ENOTFOUND {{baseurl}}
  2. Status code is 200
  3. Response time is less than 30 seconds
  4. Response body is not empty
```

In general, Postman can't resolve our variables, because their initial value is hidden.



So go ahead to your collection and navigate to **`Edit` → `Variables`** and "**unhide**" the initial value (copy and paste the current value to in the initial value). Don't forget to save the changes!

Go to the CLI and since you're already logged in, just run the second command:

```
PS C:\Users\mddim> postman collection run 14255838-2f6080d8-3dde-45ef-8738-89[redacted]
```

And this time all our tests are passing. 😊

Some useful commands for Postman CLI:

- **`postman login`**
- **`postman logout`**
- **`postman collection run {collection id}`**
- **`postman collection run {collection URL}`**
- **`postman collection run {JSON file}`**

Some additional options for the **`postman collection run`** command

- **`--bail option`** → **stops the collection from running immediately if any test fails** (works or collection id, collection URL and JSON file)
- **`--delay-request [number]`** → specifies a **delay (in milliseconds) between requests**
- **`-i [requestUID]`** or **`[folderUID]`** → **Runs only the specified folder UID or request UID** from the collection

3. Newman

Postman CLI lacks custom reporters and integrations with third party tools and services. This is where his older brother **Newman** comes to help. It is also created by Postman, but under an open-source model, which makes it easier to extend and create more advanced integrations.

3.1. Installation

Newman runs inside the **Node.js** runtime environment. So, this is why **Node.js** needs to be installed locally on your computer. To check if you have **Node.js** installed run the following command on your terminal:

```
node -v
```

It will show you the current version installed on your computer.

If you don't have Node.js installed, go to **https://nodejs.org/en** and install the LTS version.

Now, to install Newman, you need to run this command:

```
npm install -g newman
```

```
PS C:\Users\mddim\onedrive\desktop> npm install -g newman
npm WARN deprecated har-validator@5.1.5: this library is no longer supported
npm WARN deprecated uuid@3.4.0: Please upgrade  to version 7 or higher.  Older versions may
use Math.random() in certain circumstances, which is known to be problematic.  See https://v
8.dev/blog/math-random for details.

added 117 packages in 23s

9 packages are looking for funding
  run `npm fund` for details
```

3.2. Running a Collection

The command we need to write to run a Postman collection in Newman is:

newman run (followed by a location where Newman can fetch that collection – this can be a local file or a remote URL)

You already have the JSON file with your collection, stored in your computer and you also have the URL of the collection, both from the exercises with Postman CLI. Go ahead and run the tests from the terminal. Pretty much it is the same as with Postman CLI, so what is so special about Newman? Newman has multiple reporters. Let's see what a reporter is.

3.3. Generating HTML reports

Step 1: Install HTMLEXTRA

The **htmlextra** Reporter is a **plugin** for **Newman** that generates comprehensive **HTML reports** for Postman collection runs. This tool is designed to provide an enhanced visualization of API test runs for better analysis and debugging. You can find it here:

https://www.npmjs.com/package/newman-reporter-htmlextra

In order to install it you should run the following command:

```
npm install -g newman-reporter-htmlextra
```

Step 2: Using Newman with HTMLEXTRA

In order to use this reporter, specify **`htmlextra`** in Newman's **`-r`** or **`--reporters`** option. The following command will create a new report in the **`./newman`** directory. If the directory does not exist, it will be created as part of the Newman run.

```
newman run {collection.json} -r htmlextra
```

```
PS C:\Users\mddim> newman run .\TrelloAPIFinalCollection.postman_collection.json -r htmlextra
PS C:\Users\mddim>
```

For a few seconds, nothing happens and you might think that this doesn't work, but if you navigate to where your collection is stored in your computer you will find a folder called **`newman`** was created. In that folder, you'll have the HTML report. Go ahead and open it. It will open in your browser and it has pretty awesome features. Go ahead and take a look. Explore it. Click on Summary, Total Requests, Failed Tests, Skipped Tests. It contains a lot:

- A summary dashboard with the overview of the run
- Tabs for detailed request and response data
- Visual indicators for passed and failed tests
- Buttons or links for navigation within the report
- Options to view logs, headers, body data, and variables used during the run

But what happened with the CLI report? Whenever we are writing **`-r`** in Newman, we are specifying a reporter and only that reporter will be used. You can specify as many reporters as you like. So, in order to see the CLI report on the console and create an HTML report via **`htmlextra`**, we can specify that, like this:

```
newman run {collection.json} -r 'cli,htmlextra'
```

```
PS C:\Users\mddim> newman run .\TrelloAPIFinalCollection.postman_collection.json -r 'cli,htmlextra'
newman

TrelloAPIFinalCollection

→ Get All Boards
```

`htmlextra` is highly customizable, through command-line options, so we suggest to explore its documentation.

https://www.npmjs.com/package/newman-reporter-htmlextra

II. Selenium IDE

1. Selenium IDE

We have the "**SeleniumIDE**" **solution** in the **resources which has some test projects already**.

This Selenium test project automates the testing of a login process for the **Sauce Demo** website, focusing on the scenario where an invalid username is entered.

The test **TC01IfUserIsInvalidTryAgainTest** automates the process of testing if a user with invalid credentials is prompted to try again and then successfully logs in with valid credentials.

Key Steps

Setup

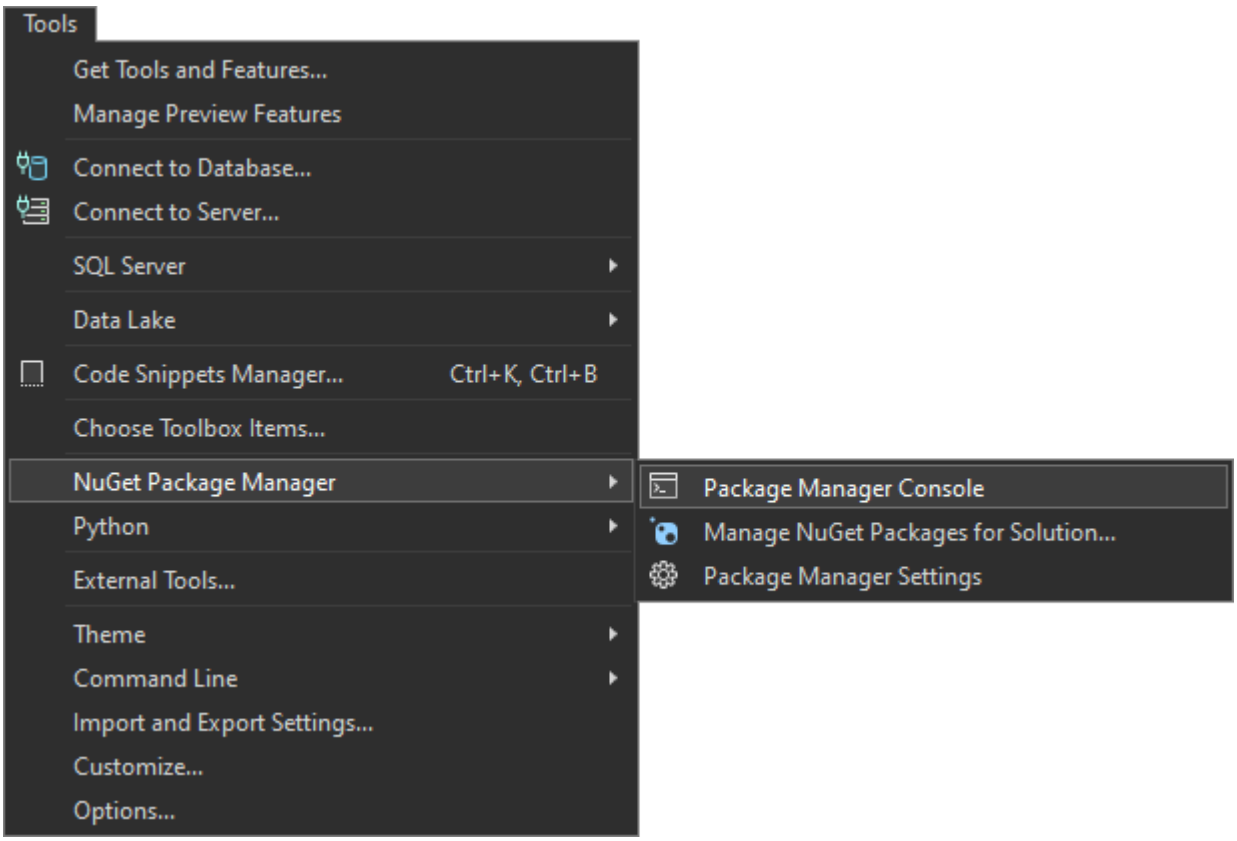
A Chrome browser instance is created (**ChromeDriver**) and the browser window is resized to specific dimensions. A JavaScript executor (**IJavaScriptExecutor**) is initialized to execute JavaScript within the test.

Test Logic

The test navigates to the Sauce Demo website (https://www.saucedemo.com/) and it simulates entering an invalid username (**user123**) and the correct password (**secret_sauce**). After submitting the form, it captures the error message shown on the page and checks if it matches the expected error.

In general, the test demonstrates how to handle login attempts with incorrect credentials, capture error messages, and retry with the correct information. It provides a good example for automated testing in cases where login functionality needs to handle both invalid and valid input.

Open **Visual Studio** and from there navigate to the **Tools** menu. Select **NuGet Package Manager** and select **Package Manager Console**:



Let's first build the application by using the following command:

```
dotnet build
```

After you have **ensured** that the **build** was **successful**, you can **run** the **tests**, too, by using the command below or just by clicking on the **`[Run All Tests in View]`** button in the **`Text Explorer`**.

```
dotnet test
```

III. Selenium WebDriver

1. Chromedriver

Before exploring the solution, there is something very important that we must acknowledge ourselves with – **the Chromedriver**.

What is Chromedriver?

Chromedriver is a tool that enables automated testing of web applications in Google Chrome. It acts as a bridge between the Selenium automation framework and the Chrome browser. When running a Selenium test, **Chromedriver** communicates with the browser, sending commands (like navigating to a URL, clicking elements, or filling forms) and retrieving results (like checking if an element is visible or validating text on a page).

Why is Chromedriver important?

When automating browser interactions using Selenium, each browser requires its own driver to communicate with Selenium. For example, Firefox uses geckodriver, while Chrome uses chromedriver. Without it, Selenium cannot control Chrome or run any automated tests.

When Selenium wants to control a browser, it needs a driver that translates commands from Selenium to the browser. In this case, for Google Chrome, we need Chromedriver. It's like a remote control that makes sure Selenium can interact with the browser efficiently.

How to Download Chromedriver?

Step 1: Check Chrome Version

Before downloading **Chromedriver**, you need to know your installed Chrome version. To do this, open Chrome, click on the three-dot menu in the top-right corner, go to **Help** > **About Google Chrome**, and note the version number (e.g., 116.x.x).

Step 2: Download Chromedriver

Go to the official **Chromedriver** page: https://sites.google.com/chromium.org/driver. Find the **Chromedriver** version that matches **your installed Chrome browser version**. Download the appropriate version based on your operating system (Windows, Mac, Linux).

Step 3: Add Chromedriver to Your Project

Extract the downloaded file and place it in a directory accessible to your project (for example, a folder called drivers within your project structure). Explicitly provide the Chromedriver path when initializing the WebDriver in the project (as seen in the tests).

2. Selenium WebDriver

These tests automate real-world interactions with live web applications and validates that the system works as expected across different scenarios.

You can run the tests the same way you ran them for **SeleniumIDE**.

IV. Playwright

1. Application Specifications

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

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:

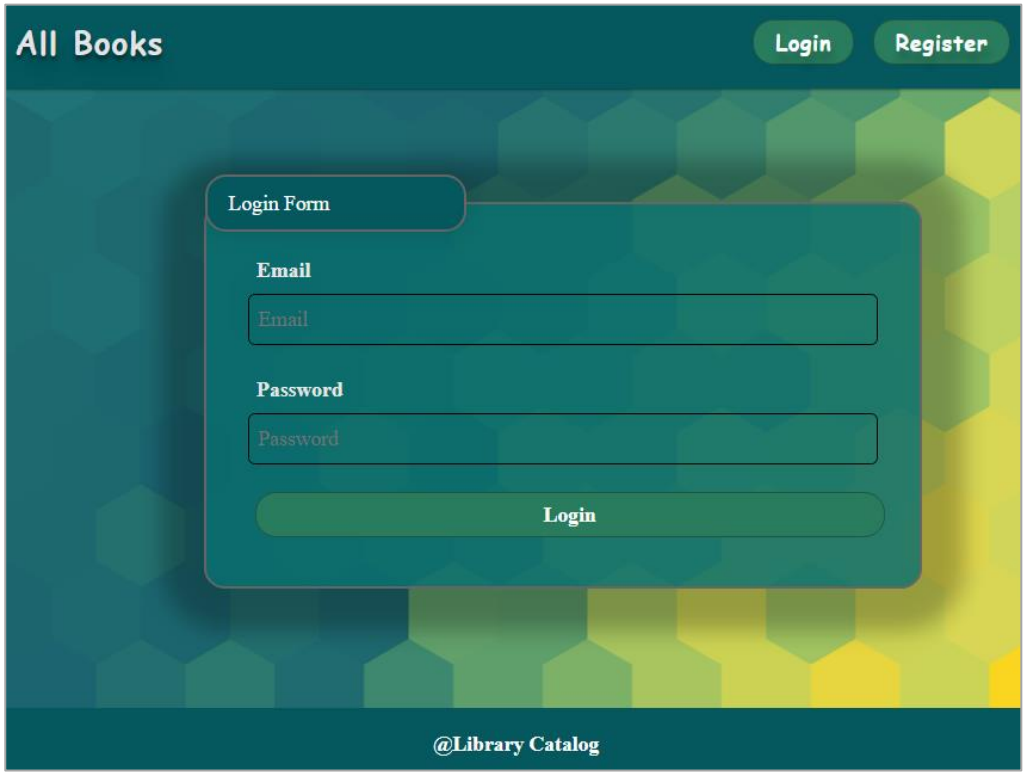
Guest navigation example:

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:

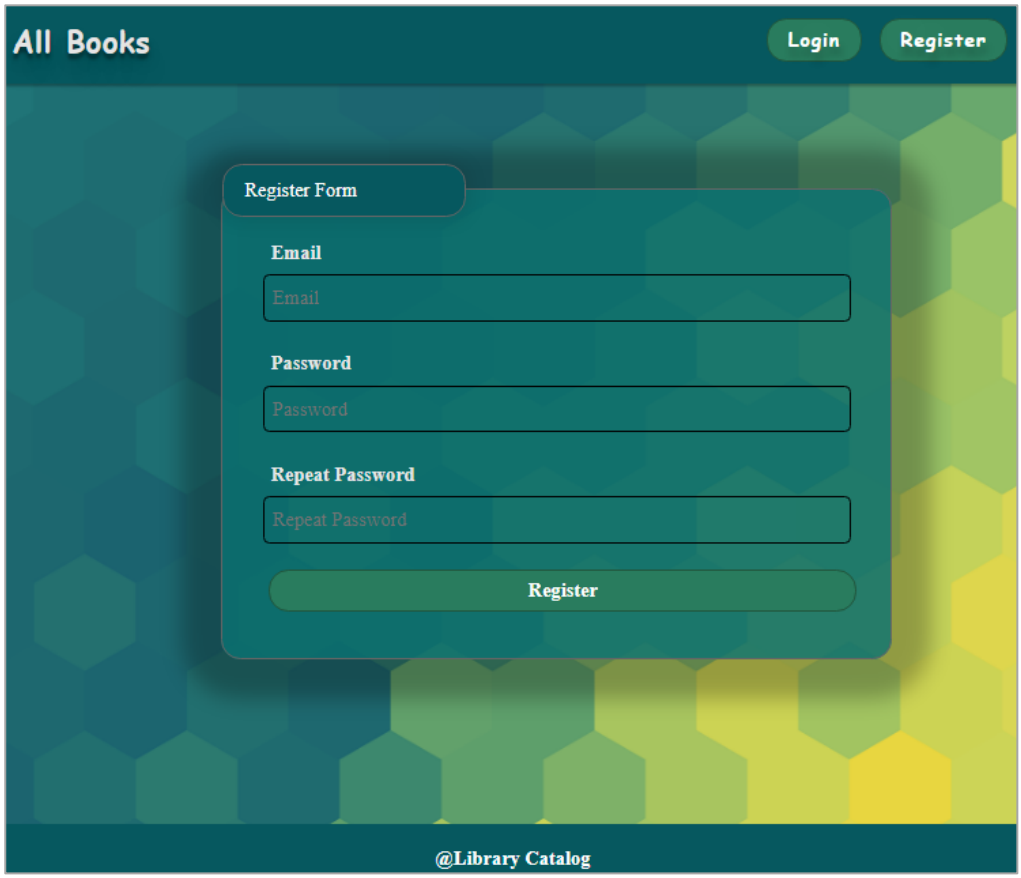
```
{
  email,
  password
}
```

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

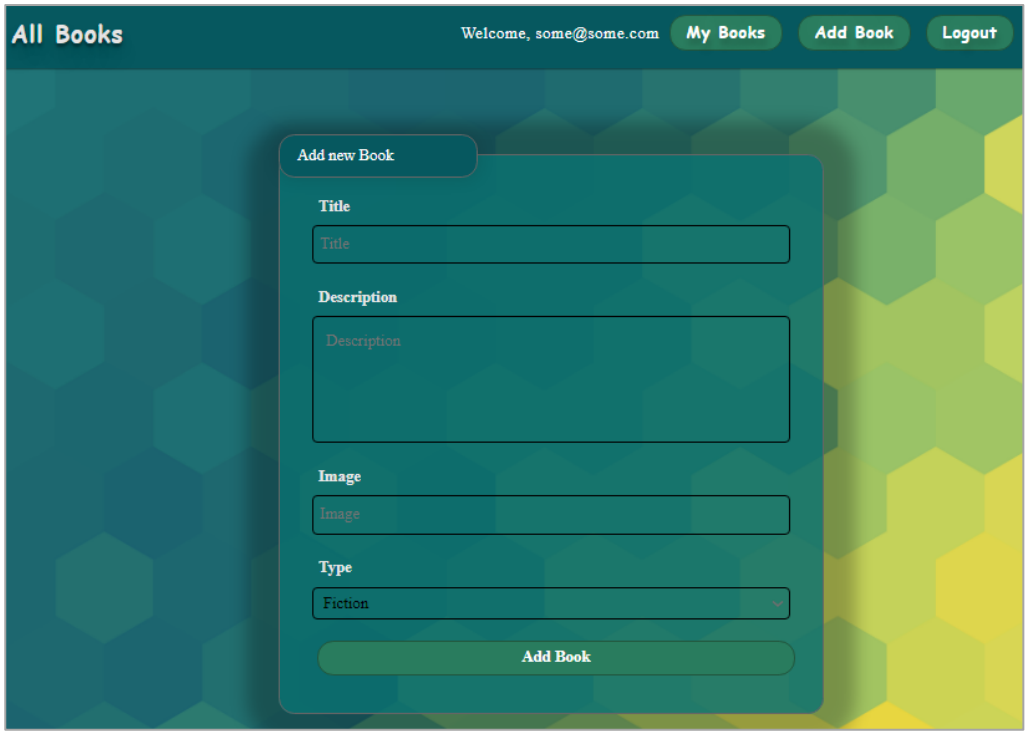
The logout action is available to **logged-in users.** If the logout is successful, the app **redirects** the user to the **`All Books/Dashboard`** page.

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
```

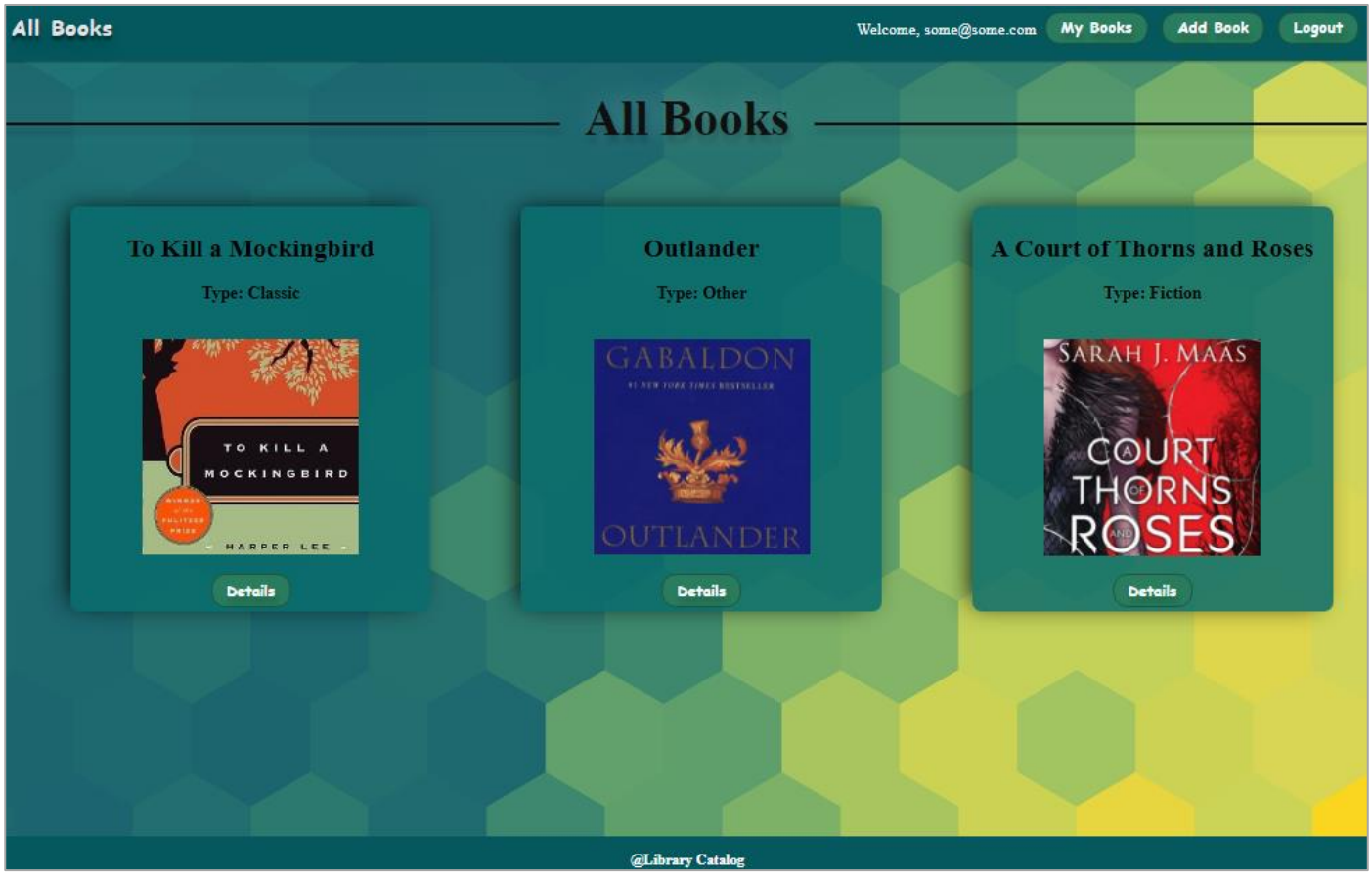
The service expects a body with the following shape:

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

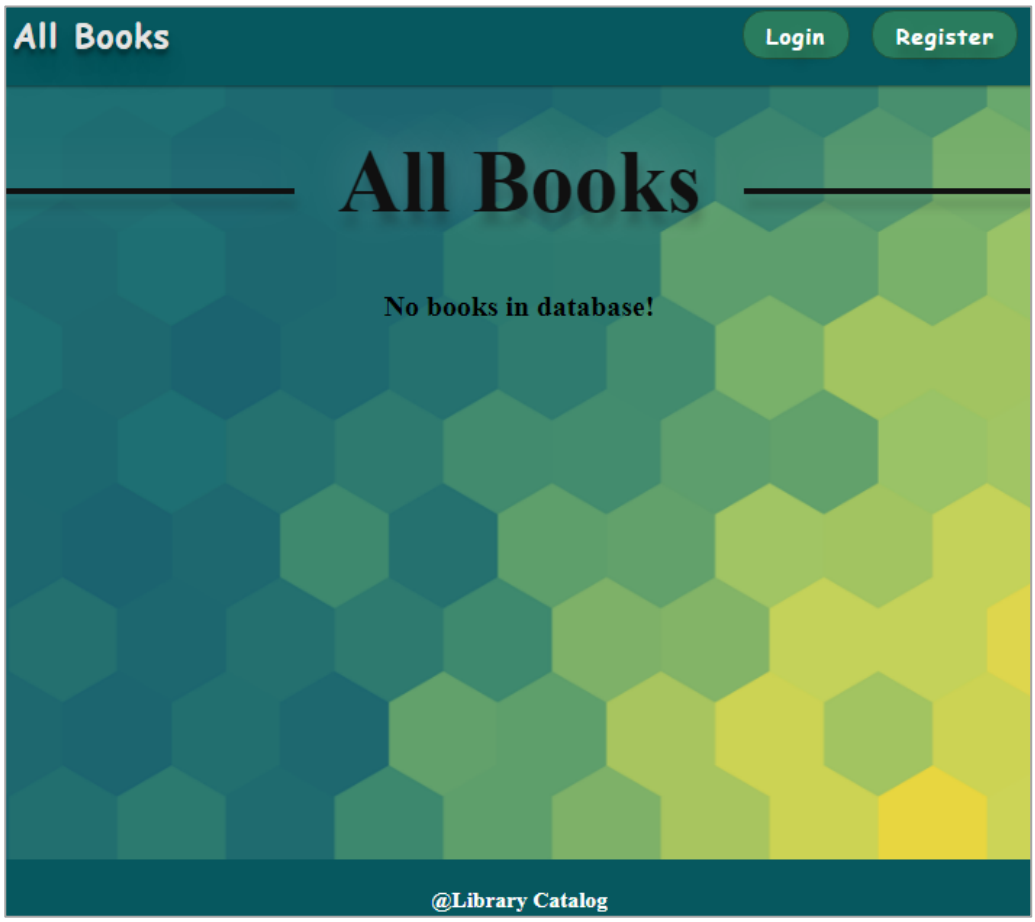
The service returns the newly created record. Upon success, it **redirects** the user to the **`All Books/Dashboard`** page.

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:

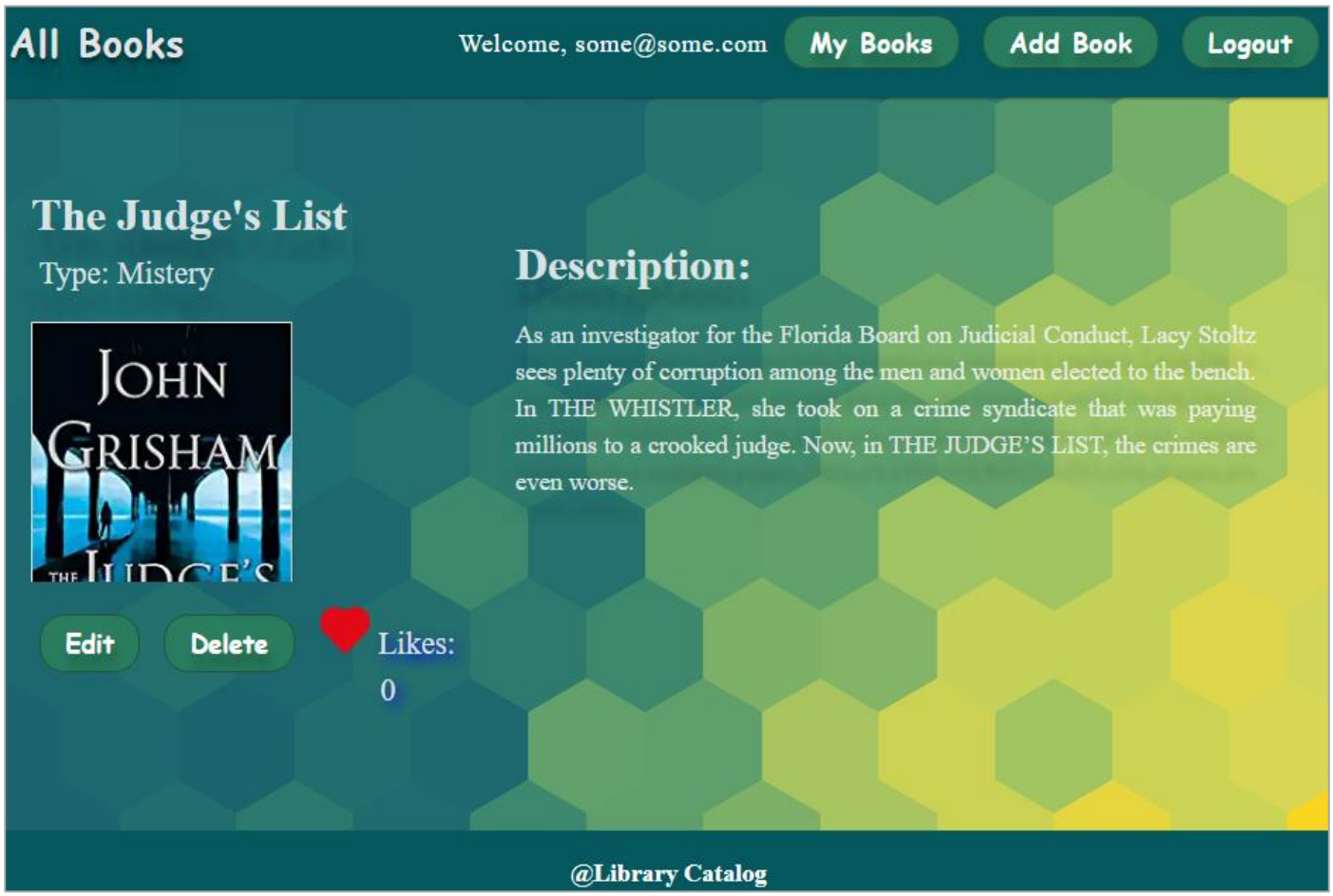
```
Method: GET
URL: /data/books?sortBy=_createdOn%20desc
```

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

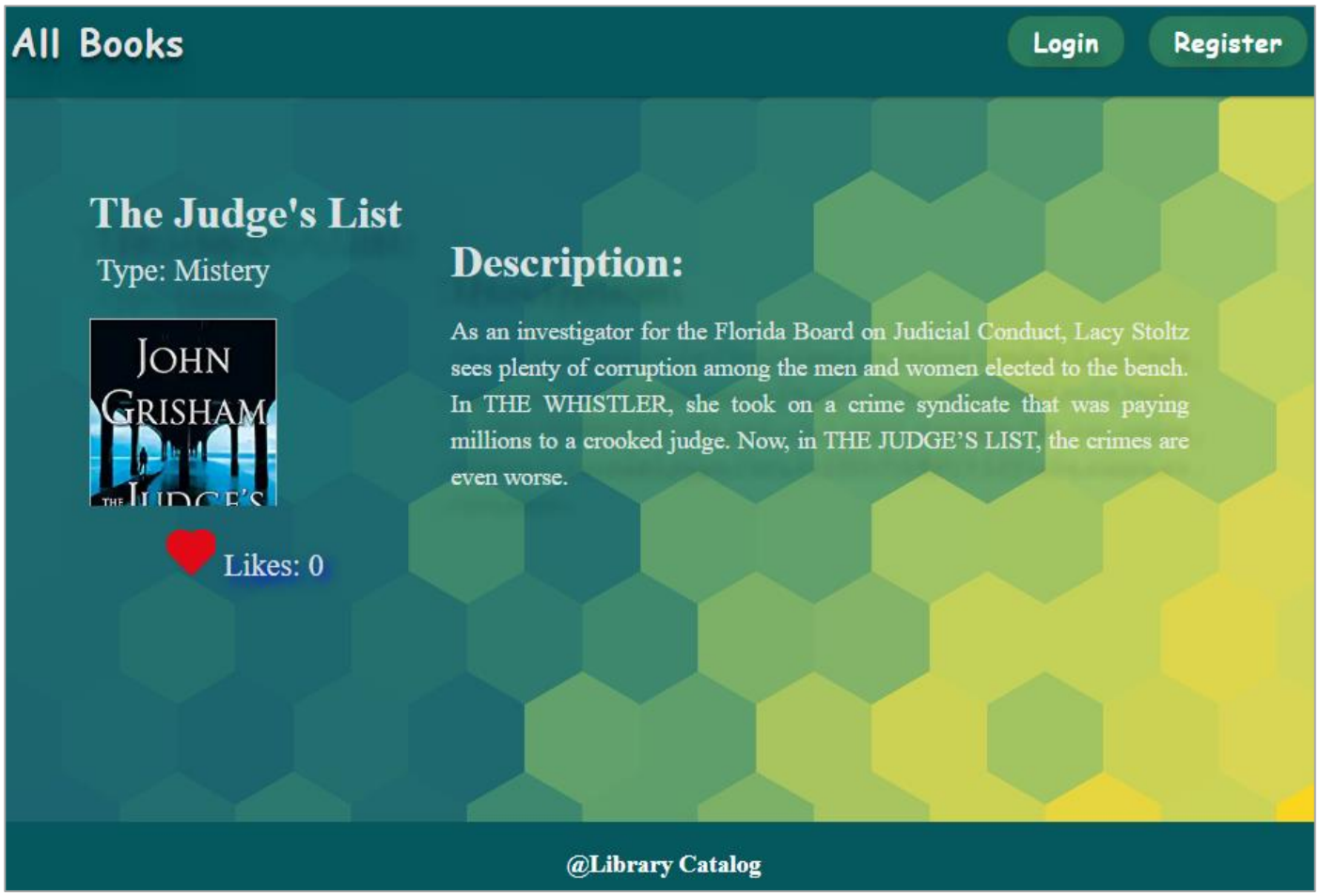
Book Details

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,
```

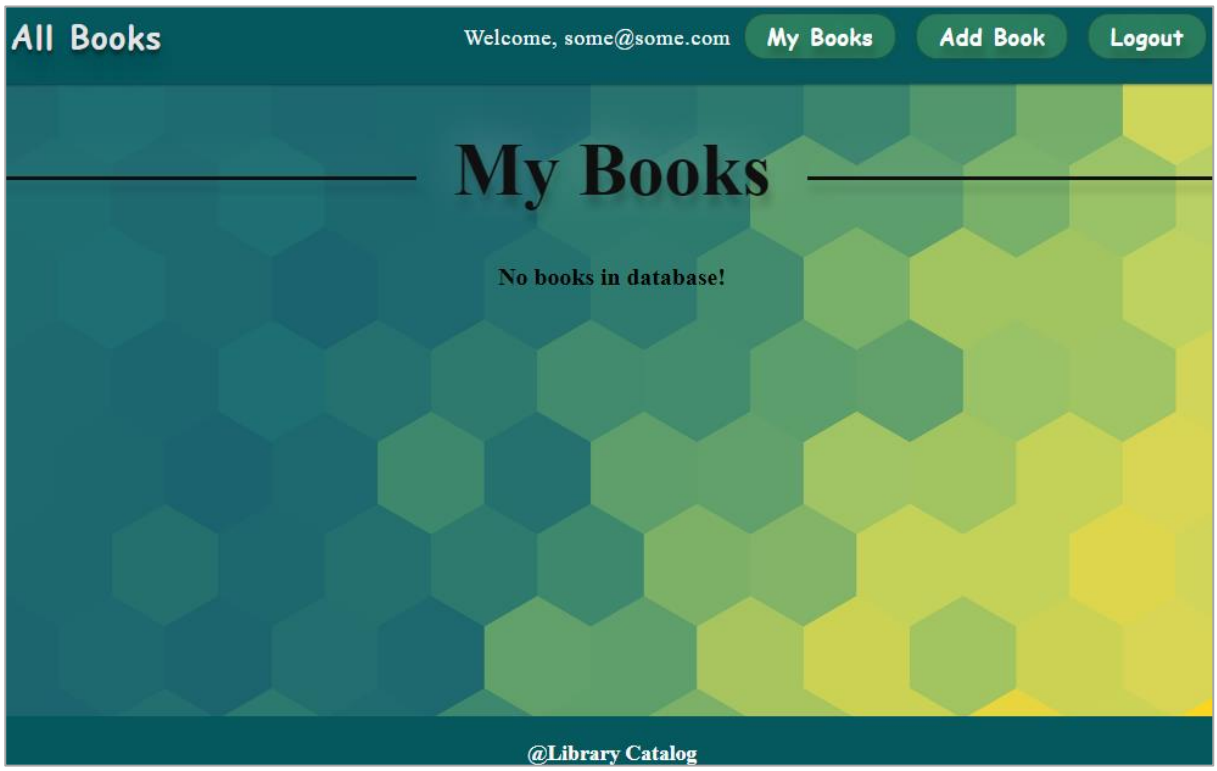
```
  description,
  imageUrl,
  type
}
```

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:

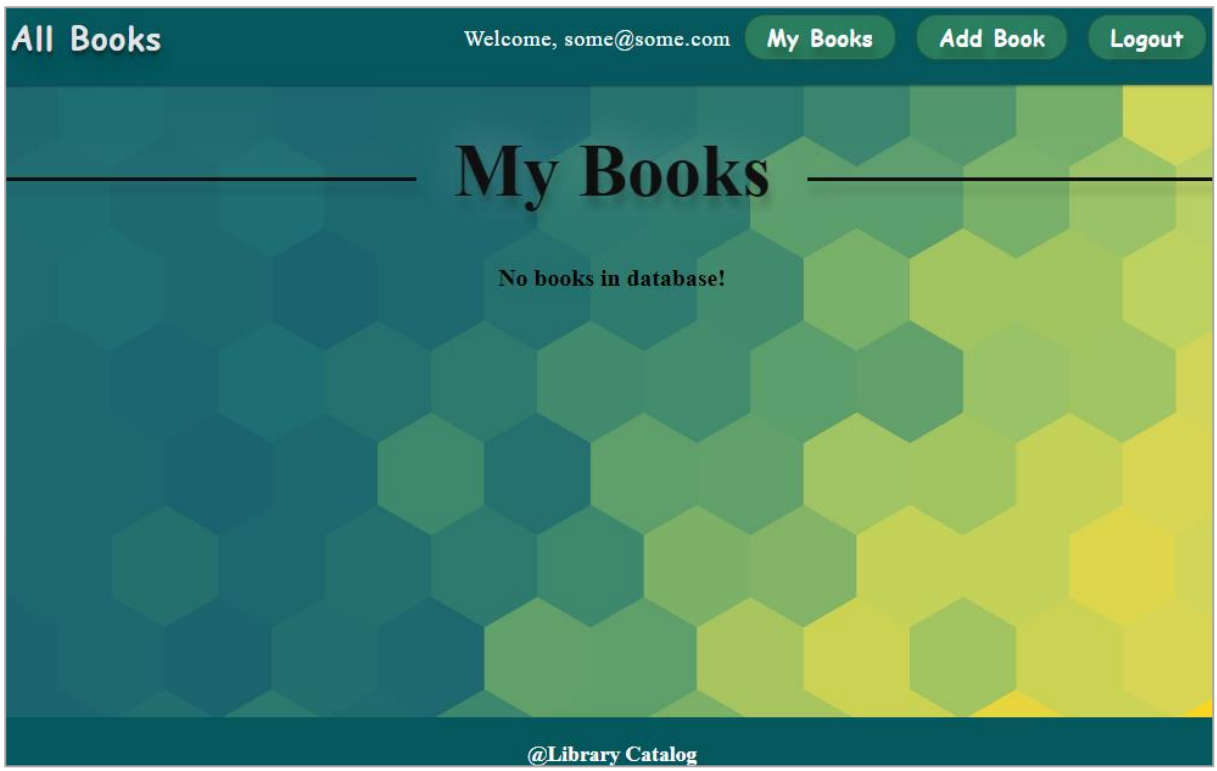
```
Method: DELETE
URL: /data/books/:id
```

`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:

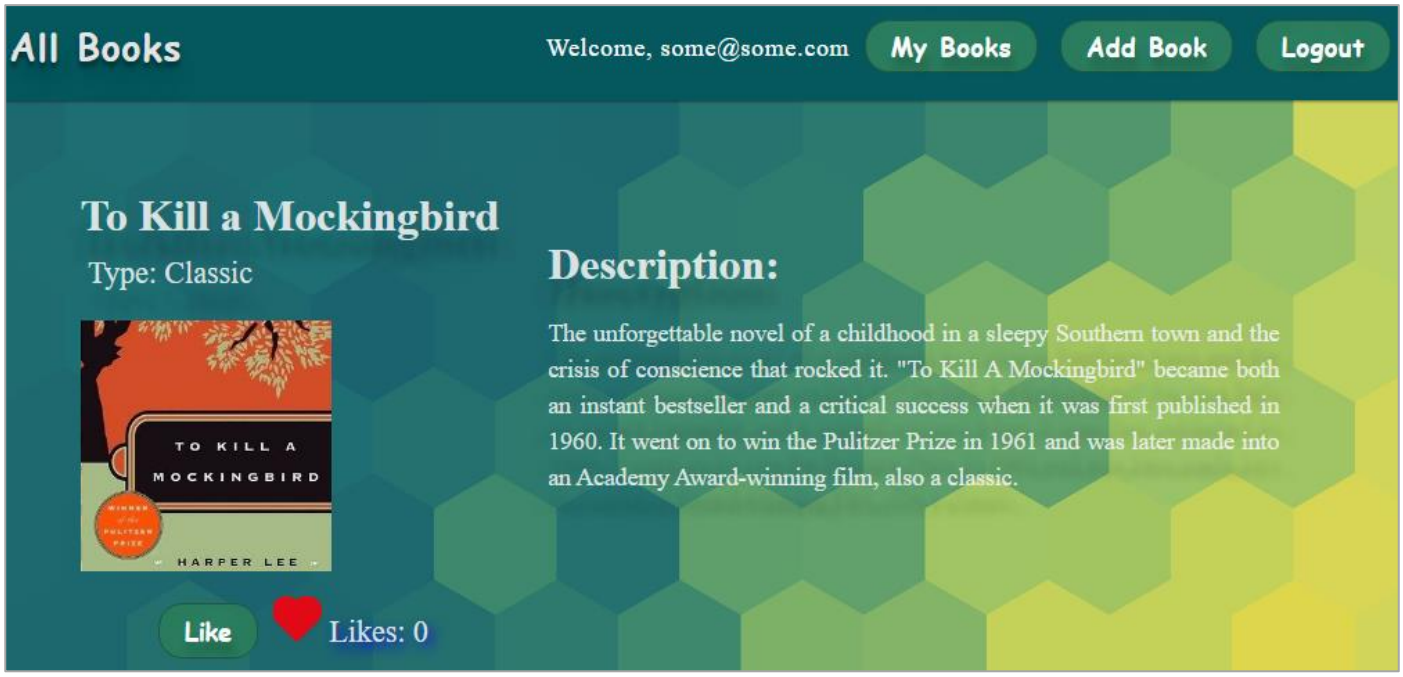
```
Method: GET
URL: /data/books?where=_ownerId%3D%22{userId}%22&sortBy=_createdOn%20desc
```

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

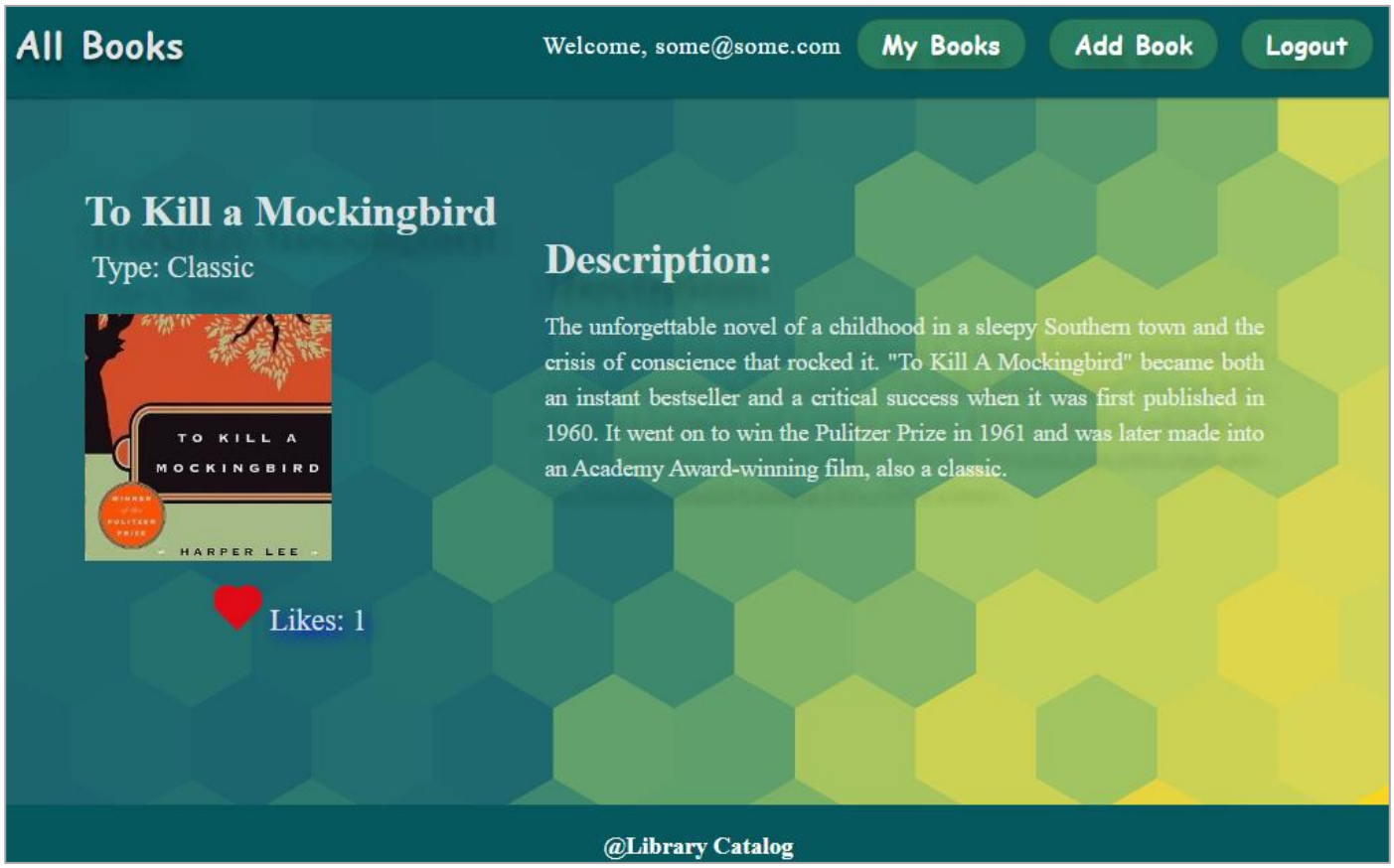
Like a Book

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**.

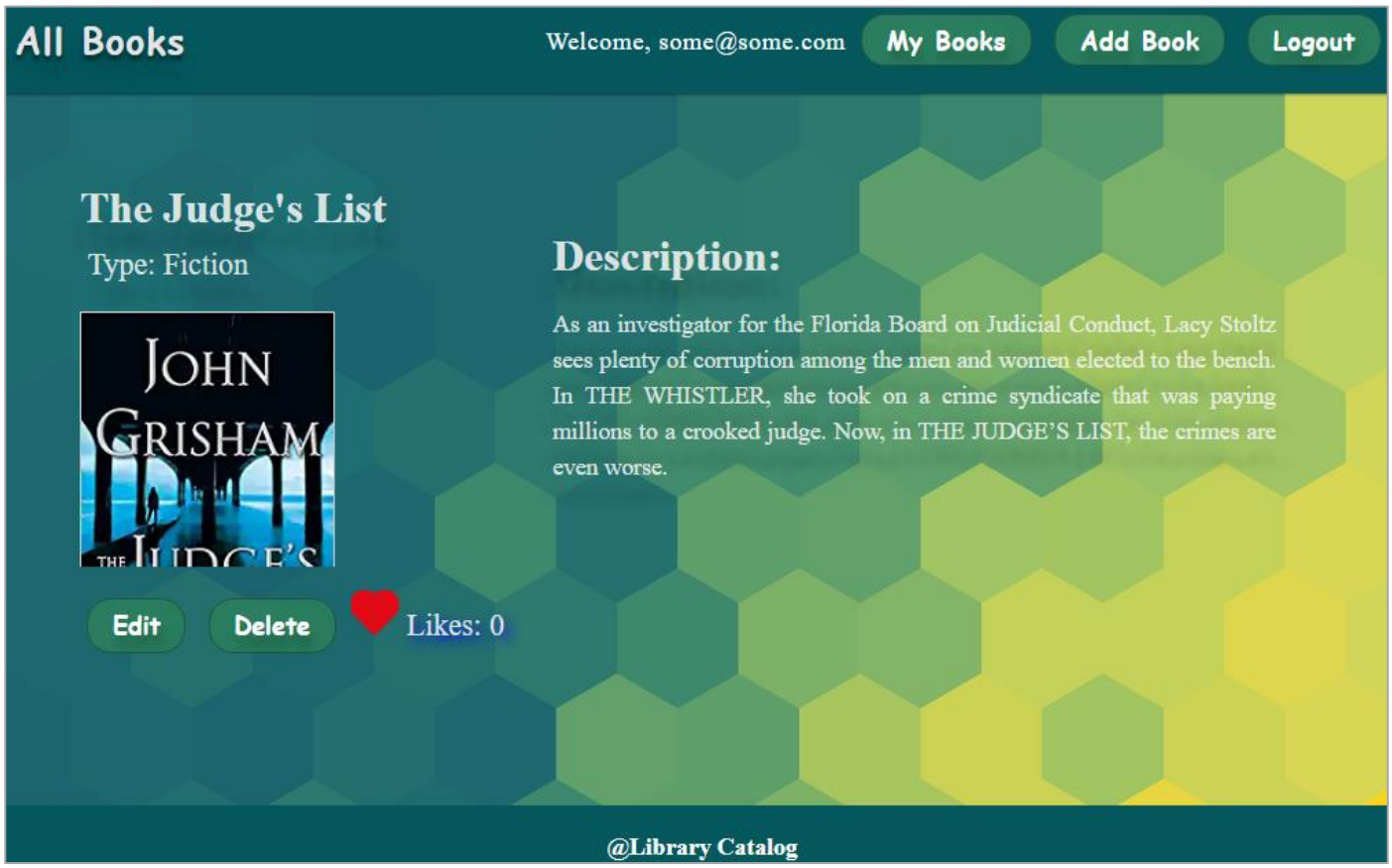
The view when the user did not press **`[Like]`** button looks like:



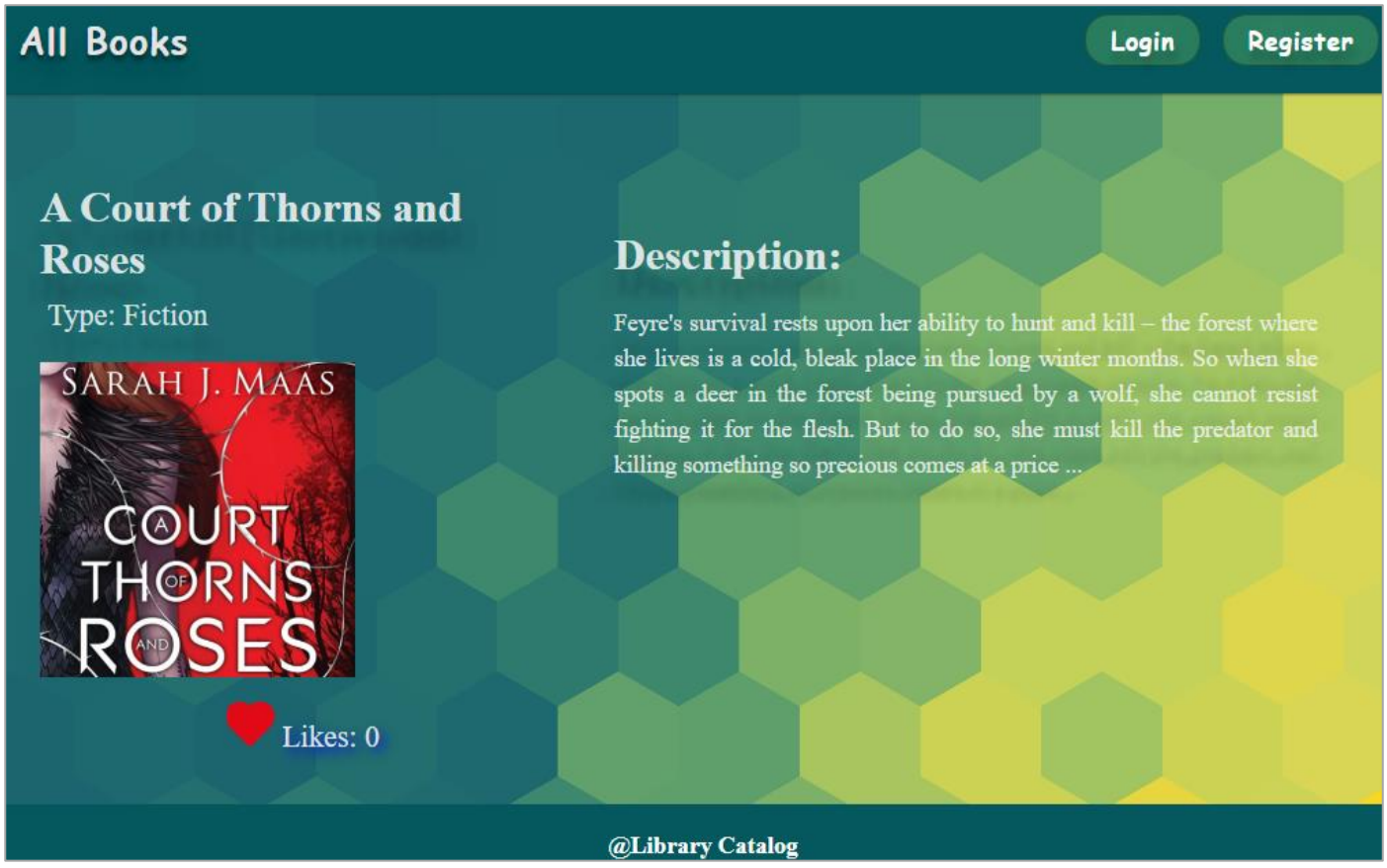
When the user **liked** the book, the **`[Like]`** button is **not** available and the counter is **increased** by **1**.



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. Running the Tests

First, inspect the **package.json** file and take a look at the **scripts** and **dependencies** sections.

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
npx playwright install
```

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:

```
npm run server
```

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

```
npm run test
```

```
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)
```

V. API Testing

You are given a simple API which executes simple CRUD operations over a collection of books. The application is a JS app and the API tests are written in **Mocha** and **Chai**.

Inspect the tests and run them by executing the command

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

```
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)
```

VI. Integration Testing

LibroConsoleAPI is a console-based application built using ***.Net Core Framework*** that manages a collection of books.

It allows users to perform various operations, such as **creating**, **reading**, **updating**, **deleting** books in database in **SQL Server**.

The application has some **xUnit** and **nUnit** integration tests already included in the solution. Try running them separately and observe the results.