**Web Applications Project Assignment**

Table of Contents

[Project Description 2](#_Toc15907143)

[Functional Requirements 2](#_Toc15907144)

[Public Part 2](#_Toc15907145)

[Homepage 2](#_Toc15907146)

[Register 2](#_Toc15907147)

[Login 3](#_Toc15907148)

[Logout 3](#_Toc15907149)

[Private Part 3](#_Toc15907150)

[Retrieve all books 3](#_Toc15907151)

[View individual book 4](#_Toc15907152)

[Borrow a book 4](#_Toc15907153)

[Return a book 4](#_Toc15907154)

[Read book reviews 5](#_Toc15907155)

[Create book review 5](#_Toc15907156)

[Update book review 5](#_Toc15907157)

[Delete book review 6](#_Toc15907158)

[Rate book 6](#_Toc15907159)

[Like reviews 7](#_Toc15907160)

[Flag reviews 7](#_Toc15907161)

[Administration Part 7](#_Toc15907162)

[CRUD any books/reviews 7](#_Toc15907163)

[Unlist books 7](#_Toc15907164)

[Ban users 8](#_Toc15907165)

[Delete users 8](#_Toc15907166)

[Delete review 8](#_Toc15907167)

[General Requirements 9](#_Toc15907168)

[Backend Requirements 9](#_Toc15907169)

[Frontend Requirements 9](#_Toc15907170)

[Projects Defenses 10](#_Toc15907171)

# Project Description

Design and implement the new **Telerik Library System**, where the users can borrow and return books, rate them, write a review about a book they have borrowed, read all the reviews about a book and rate the reviews.

# Functional Requirements

Each requirement is categorized in one of three categories – **must**, **should** or **could**.

* Must requirements have the highest priority and should be addressed first.
* Should requirements have medium priority and should be addressed after all or most must requirements have been implemented and tested.
* Could requirements should be left for the last.

The application should have:

* **public part** (accessible without authentication)
* **private part** (available for registered users)
* **administration part** (available for admin users only)

Any additional features are welcome, if you have covered all the listed requirements below.

## Public Part

The **public part** of your library system should be **accessible** **without authentication**.

This section **must** support the following functionalities:

### Homepage

The homepage must be the landing page for the library where the user can get acquainted with the system is some way along with corresponding functionality for user register and user login.

* **Frontend**

The homepage is the face of your system. It should have something engaging for the user (public books, video, carousel widget, etc.) and should show some kind of navigation through the application (navbar, side menu, etc.) as well as points to the register/login functionalities or pages.

### Register

A register functionality must exist with at least a **username** field and a **password** field, which are both **client-side** **and** **server-side validated**. Two users with the same username cannot exist.

* **Backend**

You should get the user from the request body, validate the properties, check if a user with such username already exists and store it in the database. **Don’t store the password as a plain text! Use hashing.**

**Example requests: POST /api/users**

**Example responses: The user’s data (without the password), a success/error message**

* **Frontend**

The register functionality may be done on a separate route or within a widget (like a modal). It should contain the desired fields with the desired validations that can tell the user immediately that his field is invalid and a button that will make a request to the API with the user’s input. After the response you can redirect or show a notification to the user.

### Login

A login functionality must exist with at least **username** field and a **password** field.

* **Backend**

You should get the user from the request body, validate the properties, check if user with such username exists and if so, compare the password from the request body with the password in the database. If the login passes send a token (JWT), that will be used to authenticate the user, as a response.

**Example requests: POST /api/session, POST /api/users/login, POST /api/login**

**Example responses: A token (JWT) for the authenticated user, a success/error message**

* **Frontend**

The login functionality may be done on a separate route or within a widget (like modal). It should contain the desired fields with the desired validations that can tell the user immediately that his field is invalid and a button that will make a request to the API with the user’s input. If the server returns a successful response with token, you should save the token in the local/session/cookie storage and use it to access the private API resources. You can redirect or show a notification to the user.

### Logout

A logout functionality must exist.

* **Backend**

You should listen for an authenticated request with a valid token.

**Example requests: DELETE /api/session, GET /api/users/logout, GET /api/logout**

**Example responses: The user data, a success/error message**

* **Frontend**

The login functionality may be done with just a button that sends a request to the API and on successful response deletes the token from the storage. You can redirect or show a notification to the user.

## Private Part

The **private part** of your library system should be **accessible** **for registered users**.

This section **must** support the following functionality:

### Retrieve all books

A retrieve all books functionality must exist that will allow the user to see all the books in the library. Each book should have a status that indicates if the book is currently borrowed, if the books is unlisted (cannot be retrieved or borrowed) and if the books if free and can be borrowed.

* **Backend**

You should get the books from the database and return them as a response. You can add some additional features like server-side pagination, filtering or sorting using query parameters.

**Example requests: GET /api/books**

**Example responses: The books data**

* **Frontend**

The view all books functionality should make a call to the API and visualize the returned books in some way. You can add some additional features like client-side pagination, filtering or sorting. You can display a “borrow” button on books that are available (not borrowed and not unlisted).

### View individual book

A view individual book functionality must exist that will allow the user to see individual books.

* **Backend**

You should get the book from the database by something unique (like id) and return it as a response.

**Example requests: GET /api/books/:bookId**

**Example responses: The book’s data**

* **Frontend**

The view individual book functionality should make a call to the API and visualize the returned book in some way.

### Borrow a book

A borrow a book functionality must exist that will allow the user to borrow a book that is free.

* **Backend**
* You should get the book (or only the changed properties) from the request body, validate the properties, find the same book in the database by something unique (like id), check if it can be borrowed and update its status.

**Example requests: POST /api/books/:bookId**

**Example responses: The book’s data**

* **Frontend**

The borrow a book functionality should make a call to the API and visualize the returned book in some way.

### Return a book

A return a book functionality must exist that will allow the user to return a book they have borrowed.

* **Backend**
* You should get the book (or only the changed properties) from the request body, validate the properties, find the same book in the database by something unique (like id), check if it has been borrowed by the user and update its status.

**Example requests: POST /api/books/:bookId**

**Example responses: The book’s data**

* **Frontend**

The return a book functionality should make a call to the API and visualize the returned book in some way.

### Read book reviews

The book must have **reviews** as **additional property**.

* **Backend**

Each individual book should hold its reviews. You can consider each user to hold its reviews also. **The review should not exist as a separate resource, unless you have a good reason to expose it (e.g. GET /api/reviews)!**

* **Frontend**

The ‘view individual book’ view should be extended to feature the book’s reviews.

### Create book review

A create book review functionality must exist with at least a **content** field.

* **Backend**

You should get the review from the request body, validate the properties and add the review in the authenticated user’s book reviews in the database.

**Example requests: POST /api/books/:bookId/reviews,**

**POST /api/users/:userId/ reviews,**

**POST /api/users/:userId/ books /: bookId / reviews**

**Example responses: The review’s data, the book’s reviews, a success/error message**

* **Frontend**

The create book review functionality can be done in the ‘view individual book’ view or in the ‘view all books’ view with a button somewhere in the book. The feature should enable the user to create a review for the desired book with a form that contains the review’s fields with the desired validations that can tell the user immediately that his field is invalid and a button that will make a request to the API with the review’s data. You can add some additional features that will allow the user to make the font of the text bold/italic or create a hyperlink. On response you can redirect or show a notification to the user.

### Update book review

An update book review functionality must exist that will allow the user to edit reviews. The user should be able to edit **only his own reviews**.

* **Backend**

You should get the review (or only the changed properties) from the request body and validate the properties. Find the same review by first finding the book or the user, that it belongs to, and then searching inside the entity’s reviews property by something unique (like id) and update it **if it belongs to the authenticated user**.

**Example requests: PUT /api/books/:bookId/reviews/:reviewtId,**

**PUT /api/users/:userId/reviews/:reviewId**

**Example responses: The updated review’s data, a success/error message**

* **Frontend**

The update book review functionality can be done in the ‘view individual book’ view or in the ‘view all books’ view (where you show the reviews) with a button somewhere in the review. The feature should enable the user to edit the review’s content with a form that contains the desired fields with the desired validations that can tell the user immediately that his field is invalid and a button that will make a request to the API with the updated review’s data. On response you can redirect or show a notification to the user.

### Delete book review

A delete book review functionality must exist that will allow the user to remove their book reviews. The user should be able to remove **only his own reviews**.

* **Backend**

You should find the reviews by first finding the book or the user, that it belongs to, and then searching inside the entity’s reviews property by something unique (like id) and delete it **if it belongs to the authenticated user**. Instead of deleting anything you can consider featuring an **isDeleted: boolean** property to your db entities and rise the flag instead.

**Example requests: DELETE /api/books/:bookId/reviews/:reviewId,**

**DELETE /api/users/:userId/reviews/:reviewId**

**Example responses: The deleted review’s data, a success/error message**

* **Frontend**

The delete book review functionality can be done in the ‘view individual book’ view or in the ‘view all books’ view (where you show the reviews) with a button somewhere in the book. You can consider showing a popup to the user to confirm if he really wants to delete his review. The delete review feature should make a request to the API and on response you can redirect or show a notification.

This section **should** support the following functionality:

### Rate book

A rate book functionality should exist for each book the user has borrowed and returned. Rating could be a number between 1 and 5, or whatever point system you decide to implement.

* **Backend**

You should get the rating from the request body, validate the properties, validate if the user posting the rating has read the book and create a rating for the book (or update it if they have rated the book before).

**Example requests: PUT /api/books/:bookId/rating**

**Example responses: The book’s data, a success/error message**

* **Frontend**

The rate book functionality should exist on the book view, it should show if the user has rated the book or not, and it should make a call to the API when the used selects a rating. Upon successful return of the API call the overall rating of the book should be updated (the average of all ratings of the book by different users).

This section **could** support the following functionality:

### Like reviews

The reviews could have **votes** as **additional property** and the user should be able to like or dislike a review. A **vote** model must exist with at least a **username** field and a **like** field (boolean).

* **Backend**

You should find the review in the database and check if it has a vote from the authenticated user’s username already. If it has, update it, if it doesn’t, add a new vote to the review and update it.

**Example requests: PUT /api/reviews/:reviewId/votes,**

**PUT /api/books/:bookId/ reviews /: reviewId /votes**

**Example responses: The review’s data, the review’s likes, a success/error message**

* **Frontend**

The user should be able see the votes count and click a button to like/dislike a review in the view.

### Flag reviews

The reviews could have **flagged** as an **additional property** and the user should be able to flag reviews that he sees as inappropriate. Additionally, you can send a notification to the admin users or to the user that the reviews is belonging to.

* **Backend**

You could get the review’s id from the request and find it in the db. Then update its fagged property.

* **Frontend**

You could update the review’s view to show if the review has been flagged and to allow for the user to flag it.

## Administration Part

The **administration part** of your library system should be **accessible** **only for admin users**.

This section **should** support the following functionality:

### CRUD any books/reviews

An admin user should be able to **create, read, update and delete** any book or review.

* **Backend**

Extend your book/review routes to check if the authenticated user has admin rights.

* **Frontend**

Extend your existing views to check if the authenticated user has admin rights. (you may store this information and pass it in the JWT)

### Unlist books

An admin user should be able to unlist books. If the book is unlisted, it should not be retrievable by users who don’t have admin rights. Unlisted books cannot be borrowed.

* **Backend**

You should find the book in the database and update its status.

**Example requests: PUT /api/books/:bookId**

**Example responses: The book’s data, a success/error message**

* **Frontend**

An admin user should have a “unlist” button available in the book’s view.

### Ban users

An admin user should be able to ban users. The users should have **banstatus** as **additional property** and the admin should be able to ban users from all library activities except reading. A **banstatus** model must exist with at least an **isBanned** field and a **description** field. Additionally, it could feature a ban expiration date.

* **Backend**

You should find the user in the database and update his ban status. A banned user should be restricted from every operation in the library (borrowing books, writing reviews, etc.), except reading.

**Example requests: PUT /api/users/:userId/banstatus,**

**POST/DELETE /api/users/:userId/banstatus**

**Example responses: The user’s data, a success/error message**

* **Frontend**

An admin user should have a “ban” button available in the user’s view. The buttons and actions for the modifying operations should be disabled/hidden for the banned users.

### Delete users

An admin user should be able to delete users from the library system.

* **Backend**

You should find the user in the database and delete him. Instead of deleting anything you can consider featuring an **isDeleted: boolean** property to your db entities and rise the flag instead.

**Example requests: DELETE /api/users/:userId**

**Example responses: The deleted user’s data, a success/error message**

* **Frontend**

An admin user should have a “delete” button available in the user’s view or you can consider creating a view for all users and enable the removing there.

This section **could** support the following functionality:

### Delete review

An admin user should be able to delete reviews from the library system.

* **Backend**

You should find the review in the database and delete it. Instead of deleting anything you can consider featuring an **isDeleted: boolean** property to your db entities and rise the flag instead.

**Example requests: DELETE /api/reviews/:reviewId**

**Example responses: The deleted review’s data, a success/error message**

* **Frontend**

An admin user should have a “delete” button available in the review’s view or you can consider creating a view for all reviews by a book and enable the removing there.

# General Requirements

This section **must** support the following functionality:

* + You **must** use **Git** to keep your source code and for team collaboration.
  + You **must** use **Trello** for project management.
  + You **must** use **TSLint** to write consistently styled code.
* You **must** follow the **SOLID** **principles** and the **OOP** **principles**. The lack of **SRP** or **DI** will be punished by **death**.
* You **must** use correct naming and write clean, **self-documenting code**.

# Backend Requirements

This section **must** support the following functionality:

* + You **must** use **NestJS**.
  + You **must** follow the **REST** architectural principles.
* You **must** use **TypeORM.**
* You **must** use **MySQL/MariaDB**.
  + You **must** use **services and repositories** for the data access and for the business logic.
* You **must** have at least five types of **database entities.**
* You **must** provide at least two type of **relations in the database.**
* You **must** apply proper **data validation.** All data received from the client should be validated through validation pipes.
  + You **must** apply proper **error handling.**
* You **must** write **unit tests** for the majority of your application's features. Non-isolated tests are not considered valid.

# Frontend Requirements

This section **must** support the following functionality:

* + You **must** use **Angular**.
  + You **must** create **usable and responsive UI** (use **Bootstrap/Material**).
  + You **must** use **modules and routing** to split your application logic.
* You **must** use **route guards.**
* You **must** apply proper **data validation.**
  + You **must** apply proper **error handling.**
* You **must** write **unit tests** for the majority of your application's features. Non-isolated tests are not considered valid.

# Projects Defenses

You will have around **90 minutes** to:

* Present the project overall
* Explain how you’ve created the project
* Answer some theoretical questions related to the entire Alpha program