**Web Application Project Assignment**

***Part I, Backend***

Table of Contents

[Project Description 2](#_Toc51942817)

[Functional Requirements 2](#_Toc51942818)

[Public Part 2](#_Toc51942819)

[Register 2](#_Toc51942820)

[Login 2](#_Toc51942821)

[Logout 3](#_Toc51942822)

[Private Part 3](#_Toc51942823)

[Retrieve all books 3](#_Toc51942824)

[View individual book 3](#_Toc51942825)

[Borrow a book 3](#_Toc51942826)

[Return a book 3](#_Toc51942827)

[Read book reviews 4](#_Toc51942828)

[Create book review 4](#_Toc51942829)

[Update book review 4](#_Toc51942830)

[Delete book review 4](#_Toc51942831)

[Rate book 5](#_Toc51942832)

[Like reviews 5](#_Toc51942833)

[Reading points / Gamification 5](#_Toc51942834)

[Administration Part 5](#_Toc51942835)

[CRUD any books/reviews 5](#_Toc51942836)

[Ban users 5](#_Toc51942837)

[Delete users 6](#_Toc51942838)

[General Requirements 6](#_Toc51942839)

[Backend Requirements 6](#_Toc51942840)

[Teamwork Guidelines 6](#_Toc51942841)

[Projects Defenses 6](#_Toc51942842)

[Give Feedback about Your Teammate 7](#_Toc51942843)

# Project Description

Design and implement the **Backend** of the new **Telerik** **Academy 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 are best 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:

### Register

A register endpoint must exist with at least a **username** field and a **password** field, which is **server-side validated**. Two users with the same username cannot exist. 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. **Do not store the password as a plain text! Use hashing.**

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

**Example responses: The user’s data (without the password), or an error message**

User Validation:

* number – auto, unique
* username – string - [3 - 30], no special characters, unique
* password – string – [7 - 20], one uppercase, one number
* firstName – string – [1 - 20]
* lastName – string – [1 - 20]
* age – number > 0 || null
* gender – string (list)
* e-mail – string [6 - 40], include ‘@’, no unique
* isAdmin – Boolean
* isDeleted – Boolean
* isBanned – Boolean

### Login

A login functionality must exist with at least **username** field and a **password** field. 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 or an error message**

### Logout

A logout functionality must exist. You should listen for an authenticated request with a valid token and somehow invalidate that token.

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

**Example responses: A success/error message**

## 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 book is unlisted (cannot be retrieved or borrowed) and if the book if free and can be borrowed. 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**

### View individual book

A view individual book functionality must exist that will allow the user to see individual books. You should get the book from the database by something unique (like id) and return it as a response.

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

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

### Borrow a book

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

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

### Return a book

A return a book functionality must exist that will allow the user to return a book they have borrowed. 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: DELETE /api/books/:id**

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

### Read book reviews

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

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

**Example responses: The reviews’ data**

### Create book review

A create book review functionality must exist with at least a **content** field. 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/:id/reviews,**

**Example responses: The review’s data, or an error message**

Review validation:

* id – number – auto, unique

### 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**. 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/:id/reviews/:reviewId**

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

### 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**. 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 raise the flag instead.

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

**Example responses: The deleted review’s data or an error message**

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

**Example requests: PUT /api/books/:id/rate**

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

### Like reviews

The reviews **should** have **votes** as **additional property** and the user should be able to like or dislike a review. You should find the review in the database and check if it has a vote from the authenticated user’s id or username already. If it has, update it, if it does not, add a new vote to the review and update it.

**Example requests: PUT /api/reviews/:id/votes**

**Example responses: The review’s data, or an error message**

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

### Reading points / Gamification

The users could get **reading points** for every book they have borrowed **and** returned, for writing a review or for having a review liked. Additionally, a leveling system could be introduced, and users could gain levels depending on how many reading points they have. Users should lose some percentage of points of banned. (depending on how long they are banned for)

## 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. You can add an **/api/admin/books** route and **/api/admin/reviews** route if you like.

Book:

* number – auto, unique
* Title – string [1 - 256]
* Author – string [3 - 256]
* Genre – string (list)
* Age\_recommendation – number >0
* ISBN – string [10]
* PublicationDate – date
* Language – string (list)
* printLength – number [1 - 10000]
* isDeleted – Boolean
* isBorrowed – Boolean
* bookReservation – number
* bookCount - number

### 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. 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/admin/users/:id/banstatus**

**Example responses: The user’s data, or an error message**

### Delete users

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

**Example requests: DELETE /api/admin/users/:id**

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

# 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 **Git Kanban** for project management.
  + You **must** use **ESLint** to write consistently styled code.
* You **must** follow **OOP** **principles** and other principles you **have heard of** such as **Single Responsibility**.
* You **must** use correct naming and write clean, **self-documenting code**.

# Backend Requirements

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

* + You **must** use **Express**.
  + You **must** follow the **REST** architectural principles.
* You **must** use **MySQL/MariaDB**.
  + You **must** use **services** for the data access and for the business logic.
* You **must** have at least five types of **database tables.**
* 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.**
* If you feel like going an extra mile – **research** and write **unit tests** for the majority of your application's features. All tests **must** be isolated to be valid.

# Teamwork Guidelines

Refer to the teamwork guidelines document found along with the project requirements.

# Projects Defenses

When you finish both parts of the web application you will present it.

# Give Feedback about Your Teammate

You will be invited to provide feedback about all your teammates, their attitude about this project, their technical skills, their team working skills, their contributions to the project, etc. **The feedback is important part of the project evaluation so take it seriously and be honest.**