Telerik Academy

Forum

Forum management System

Project Description

Design and implement a **Forum System**, where the users can create posts, add comments, and upvote/downvote the things that they like or dislike the most. Choose what your forum will be about and stick with it. Examples: an automotive fanbase forum, java technical learning forum, crypto trading forum, fashion forum, etc. The forum should revolve around one or two general topics.

Functional Requirements

Entities

- Each **user** must have a first and last name, email, username, and password.
  - First name and last name must be between 4 and 32 symbols.
  - Email must be a valid email and unique in the system.
- Each **admin** must have a first and last name, email, and may have a phone number.
  - First name and last name must be between 4 and 32 symbols.
  - Email must be a valid email and unique in the system.
- Each **post** must have a user who created it, a title, content, comments and how many likes it has received.
  - The title must be between 16 and 64 symbols.
  - The content must be between 32 symbols and 8192 symbols.
  - The post must have a user who created it.
  - Other users must be able to post replies.

Public Part

The public part must be accessible without authentication.

On the home page, anonymous users must be presented with the core features of the platform as well as how many people are using it and how many posts have been created so far.

Anonymous users must be able to register and log in.

Anonymous users should be able to see a list of the top 10 most commented posts and a list of the 10 most recently created posts.

Private part

Accessible only if the user is authenticated.

The user must be able to log in and log out.

Users must be able to browse posts created by the other users with an option to sort and filter them.

Users must be able to view a single post including its title, content, comments, likes, etc. The details of the post and any available user actions (comment/like/edit) should be presented on the same page.

Users must be able to update their profile information. Users should not be able to change their username once registered. Users could upload a profile photo.

Users must be able to create a new post with at least a title and content.

Each user must be able to edit only personal posts or comments.

Each user must be able to view all their or any other user's posts and comments (with the option to filter and sort them).

Each user must be able to remove one or more of their own posts. Deleting a post should be available while reading the details of an individual post or when browsing the list of all posts.

Each user must be able to comment/reply to any other forum post.

Administrative part

Accessible to users with administrative privileges.

Admin must be able to search for a user by their username, email, first name.

Admin must be able to block or unblock individual users. A blocked user must not be able to create posts or comments.

Admin must be able to delete any post.

Admin must be able to view a list of all posts with an option to filter and sort them.

Optional feature

Post Tags – In order for the users to navigate easier and find certain topics faster, you can implement tags. A tag is additional information that can be put under each post after creating the post. The process of adding a tag/s to a post is as follows: The user creates a post, then proceeds to edit the post, once on the edit post page, the user can add a tag/s under it. If the tag does not exist, a new one must be added to the database. If the tag exists, a new one must not be created, the one already in the database must be reused. All tags should be lowercase only.

After the post is tagged, other users can find it by typing in the tag in the search bar. For example: your forum is about cars, a post is marked with the tags: “mercedes” and “tuning,” when “mercedes” is typed in the search bar, all posts that are tagged with “mercedes” should appear in the front-end.

A User must be able to add/remove/edit tags only on its own posts. Admins must be able to add/remove/edit tags on all posts.

REST API

To provide other developers with your service, you need to develop a REST API. It should leverage HTTP as a transport protocol and clear text JSON for the request and response payloads.

A great API is nothing without great documentation. The documentation holds the information that is required to successfully consume and integrate with an API. You must use Swagger to document yours.
The REST API provides the following capabilities:

1. Users
   - CRUD operations (must)
   - Search by username, email, or first name (must)
   - Filter and sort certain users’ posts (must)
   - Filter all posts by tag/s (could)

2. Admin
   - Make other users admin (must)
   - Delete posts (must)
   - Block/unblock user (must)

3. Posts
   - CRUD operations (must)
   - Comment (must)
   - List and edit user’s own posts (must)
   - Comment on and like other users' posts. (must)

4. Tags – optional
   - CRUD operations (could)

Use Cases

Scenario 1

A friend of Pavel's told him about this amazing forum, where lots of people share their ideas and perspectives on the crypto/stock market. Pavel enters the website and sees a feed of posts. He can sort them by most liked or newest. He can also filter them by a certain word/s. He is an anonymous user so he cannot create a post yet. He registers and then logs in to the forum. He can now start sharing his ideas with his buddy crypto "hodlers."

Scenario 2

Your forum has accumulated thousands of new users. Most of them are proactively helpful and positive, but some of them started posting spam or/and irrelevant information to the forum. You hire a moderator. You instruct the moderator to enter the forum and create a first-time registration. You as an admin give admin rights to your moderator through the forum. They can now start deleting posts and ban users that do not follow the forum rules!

Technical Requirements

General

- Follow OOP principles when coding
- Follow KISS, SOLID, DRY principles when coding
- Follow REST API design best practices when designing the REST API (see Appendix)
- Use tiered project structure (separate the application in layers)
- The service layer (i.e., "business" functionality) must have at least 80%-unit test code coverage
- You should implement proper exception handling and propagation
- Try to think ahead. When developing something, think – "How hard would it be to change/modify this later?"

Database

The data of the application must be stored in a relational database. You need to identify the core domain objects and model their relationships accordingly. Database structure should avoid data duplication and empty data (normalize your database).

Your repository must include two scripts – one to create the database and one to fill it with data.

Git

Commits in the GitLab repository should give a good overview of how the project was developed, which features were created first and the people who contributed. Contributions from all team members must be evident through the git commit history! The repository must contain the complete application source code and any scripts (database scripts, for example).

Provide a link to a GitLab repository with the following information in the README.md file:

- Project description
- Link to the Swagger documentation (must)
- Link to the hosted project (if hosted online)
- Instructions on how to setup and run the project locally
- Images of the database relations (must)

Optional Requirements

- Integrate your project with a Continuous Integration server (e.g., GitLab's own) and configure your unit tests to run on each commit to your master branch
- Host your application's backend in a public hosting provider of your choice (e.g., AWS, Azure, Heroku)
- Use branches while working with Git

Teamwork Guidelines

Please see the Teamwork Guidelines document.