**.NET Project Assignment**

**Table of Contents**

[**Project Description** 2](#_Toc51322722)

[**Functional Requirements** 2](#_Toc51322723)

[**REST API** 2](#_Toc51322724)

[API Specification: 2](#_Toc51322725)

[**Public Part** 3](#_Toc51322726)

[Homepage 3](#_Toc51322727)

[Retrieve all beers 3](#_Toc51322728)

[Retrieve individual beer 3](#_Toc51322729)

[Search for beers 4](#_Toc51322730)

[Create beer 4](#_Toc51322731)

[Read beer reviews 4](#_Toc51322732)

[Rate and/or review a beer 4](#_Toc51322733)

[Register 5](#_Toc51322734)

[Login 5](#_Toc51322735)

[Logout 5](#_Toc51322736)

[Like reviews 5](#_Toc51322737)

[Flag reviews 6](#_Toc51322738)

[**Private Part** 6](#_Toc51322739)

[Create beer 6](#_Toc51322740)

[Rate and/or review a beer 6](#_Toc51322741)

[Like reviews 6](#_Toc51322742)

[Flag reviews 6](#_Toc51322743)

[**Administration Part** 6](#_Toc51322744)

[CRUD any beers/reviews 6](#_Toc51322745)

[Unlist beers 7](#_Toc51322746)

[Ban users 7](#_Toc51322747)

[Delete users 7](#_Toc51322748)

[Delete review 7](#_Toc51322749)

[**General Requirements** 8](#_Toc51322750)

[**Backend Requirements** 8](#_Toc51322751)

[**Frontend Requirements** 9](#_Toc51322752)

[**Teamwork Guidelines** 9](#_Toc51322753)

[**Projects Defenses** 9](#_Toc51322754)

[**Give Feedback about Your Teammate** 9](#_Toc51322755)

# **Project Description**

Welcome to **BeerOverflow**, a web application for managing users and the beers they drink.

**The mission** of BeerOverflow is to allow users to get an insight on beers from all around the world and choose the next best beer to drink.

Apart from the subjective criteria like being rated or reviewed by users, a beer **must** satisfy the minimal labeling requirements as per EU laws:

* Name.
* Type - [Porter, Ale, Lager etc.](https://i.pinimg.com/originals/52/fb/77/52fb77d63bb37409039820cf84be7b4b.jpg)
* [Brewery](http://boltbaitpackinstall49.zip/).
* [Alcohol By Volume %](https://www.thespruceeats.com/alcohol-by-volume-353204) .

***BeerOverflow*** is a community driven web app and every beer aficionado can **add**, **review** and **rate** beers.

# **Functional Requirements**

Each requirement is categorized in one of three categories:

* **must** - **highest priority**and must be addressed first.
* **should - medium priority**should be addressed afterthe**must** requirements have been satisfied.
* **could - lowest priority**and should be left for last.

The **final version** must have:

* **REST API** for 3rd party developers
* **Web pages for:**
  + **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.

## **REST API**

The REST API should work with JSON to transfer data between the request and the response.

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

### API Specification:

1. Countries
2. CRUD Operations
3. Breweries
4. CRUD Operations
5. Styles
6. CRUD Operations
7. Beers
8. CRUD Operations
9. Filter by different criteria (country, style)
10. Sort by name, ABV, rating
11. Rate a beer
12. Users
13. CRUD Operations
14. Add beer to wish list
15. Add beer to the list of beers already drank
16. Get wish list beers
17. Get drank list beers

Any additional API features are welcome, if you feel the need to implement them in BeerOverflow.

## **Public Part**

The **public part** of your web application should be **accessible** **without authentication**.

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

### Homepage

The public part of your application should be accessible without authentication. This includes the application start page, the user login and user registration forms, as well as list of all beers that have been added in the system. People that are not authenticated cannot see any user specific details, they can only browse the beers and see details of them.

* **Frontend**

The homepage is the face of your system. Ideally, It should have something engaging for the user (beer of the month, top-rated beers, 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.

### Retrieve all beers

A retrieve all beers functionality must exist that will allow the user to see all the beers in the database.

* **Backend**

You should get the beers 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.

* **Frontend**

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

### Retrieve individual beer

A view individual beer functionality must exist that will allow the user to see individual beers.

* **Backend**

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

* **Frontend**

The view individual beer functionality should make a call to the appropriate action and visualize the returned beer in some way.

### Search for beers

A search functionality must exist with. User should be able to select the criteria he would like to search for.

* **Backend**

You should get the beer/beers from the database if the criteria matches any beers in the db and return them as a response.

* **Frontend**

The search view should have a field for input as well as buttons to select the search criteria. The button sends a request to the appropriate action and if successful displays the beers in a table view. If no beers are found a notification should be displayed.

### Create beer

A create beer functionality must exist with the required fields (ex. name, brand, style, abv etc.).

* **Backend**

You should get the beer from the request body, validate the properties and add the beer in the database.

* **Frontend**

The create beer functionality can be done in the ‘view all beers’ view with a button somewhere. The feature should enable the user to create a beer with a form that contains all the required params for it 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 appropriate action with the beer’s data. On response you can redirect or show a notification to the user.

### Read beer reviews

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

* **Backend**

Each individual beer 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 /reviews)!**

* **Frontend**

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

### Rate and/or review a beer

A rate and/or review a beer functionality must exist that will allow the user to rate and/or review a beer.

* **Backend**

You should get the beer (or only the changed properties) from the request body, validate the properties, find the same beer in the database by something unique (like id) and add the rating/review to it.

* **Frontend**

The rate and/or review a beer functionality should make a call to the appropriate action and visualize the returned beer in some way.

### 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.

* **Frontend**

The register functionality 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 action inside the controller 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.

* **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 appropriate action with the user’s input. If the server returns a successful response 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.

* **Frontend**

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

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

### Like reviews

The reviews should 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 **like** field (bool).

* **Backend**

Add a new vote to the review and update it.

* **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 should have **flagged** as an **additional property** and the user should be able to flag reviews that he sees as inappropriate.

* **Backend**

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

## **Private Part**

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

### Create beer

Create beer is now user- specific and available only after authentication.

### Rate and/or review a beer

Rate and/or review a beer is now user-specific and available only after authentication.

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

### Like reviews

Like reviews is now user- specific and available only after authentication.

### Flag reviews

Flag reviews is now user-specific and available only after authentication. Additionally, you can send a notification to the admin users or to the user that the reviews is belonging to.

## **Administration Part**

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

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

### CRUD any beers/reviews

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

* **Backend**

You should check if the authenticated user has admin rights

* **Frontend**

The admin should be able to see an appropriate view which displays the beers/users as well as buttons for each operation.

### Unlist beers

An admin user should be able to unlist beers. If the beer is unlisted, it should not be retrievable by users who don’t have admin rights.

* **Backend**

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

* **Frontend**

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

### Ban users

An admin user should be able to ban users. The users should have **banstatus** as an **additional property** and the admin should be able to ban users from all private activities. A **banstatus** model could exist with at least an **isBanned** field and a **description** field. Additionally, it could feature a ban expiration date.(Consider the included ASP.NET Identity Lockout properties instead).

* **Backend**

You should find the user in the database and update his ban status. A banned user should be restricted from every operation in **BeerOverflow** (creating beers, writing reviews etc.) except the publicly available views.

* **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 **BeerOverflow**.

* **Backend**

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

* **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 system.

* **Backend**

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

* **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 beer 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 **Git Kanban** or **Trello** for project management.
* You **must** follow the **OOP** **principles**.
* You **must** use correct naming and write clean, **self-documenting code**.
* Use should take advantage of the **branches** for writing your features.
* You should write **documentation** of the project and project architecture (as .md file, including screenshots, diagrams, etc.)
* You **could** Integrate your app with a **Continuous Integration server** (Jenkins, AppVeyor or other) - configure your unit tests to run on each commit to your master branch.

# **Backend Requirements**

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

* + You **must** use **ASP.NET Core 3.1 and Visual Studio 2019.**
  + You **must** follow the **REST** architectural principles.
* You **must** use **MS SQL Server** as database back-end**.**
* You **must** use **Entity Framework Core** to access your database.
* You **must** use the standard **ASP.NET Identity System** for managing users and roles.
  + registered users should have at least one of the two roles: **user** and **administrator**
  + You **must** use **services and/or 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** use Dependency Inversion principle and Dependency Injection technique following the best practices.
* You **must** write **unit tests** for your "business" functionality following the best practices for writing unit tests (**at least 80% code coverage**).
  + Unit test framework: MSTest, XUnit or NUnit.
  + Mocking framework: Moq.
  + DB Provider: EF InMemory.

# **Frontend Requirements**

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

* + You **must** create **usable and responsive UI** (use **Bootstrap/Material**).
* You **must** create **tables with data** with **server-side paging and sorting** for a model entity.
  + You can use Kendo UI grid, jqGrid, DataTables or any other library or generate your own HTML tables.
* You **should** use AJAX for making asynchronous requests to the server where you find it appropriate.

# **Teamwork Guidelines**

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

# **Projects Defenses**

Each team member will have around **30 minutes** to:

* Present the project overall
* Explain how they have contributed to the project
* Explain their teammates’ source code
* Answer some theoretical questions related to this course and all the previous ones

# **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.**