BracketBros

Group: Vinnergruppa

Members: Ole Kristian Håseth Rudjord, Jørgen Skontorp og Rasmus Skramstad

Table of Contents

1. Introduction

BracketBros is a programming discussion forum made for the ITPE3200 portfolio exam (project 1). It is inspired by classic BB Forums. The forum is made with .NET Core 6.0, C# and with a MVC framework and a MVVM layout. This is a MVP solution where the focus is the back-end. The front-end is made with basic HTML and CSS, JavaScript and jQuery for input validation.

1.1 Functionality

This forum website allows users to browse posts and comment. Both in a feed, by searching or by using categories and tags. The feed can be sorted after different criterias. The user can register an account to create posts, comment on other posts and like posts. When the user is logged in it gets a dashboard view with an overview over the users posts, comments and likes. We have admin users with extra functionality. They have their own admin dashboard where they can change categories and tags. There is a user dashboard where the user can change their information and upload a custom profile picture.

1.2 Project quick start

We have written a quick start in the README.md file to help you get started testing our application.

2. Architecture

BracketBros is built using the Model-View-Controller (MVC) architecture as required for this task. This is the architecture we would have chosen anyways, as it offers different advantages like seperation of the different components, easy to maintain and scale, reusable code and good support in .NET Core to mention some. As Figure 1 shows, we have a database connected with the Data Access Layer. Then we have the Model-Controller-View element. Views also consists of ViewModels, layout and partial views. Note that areas/identity is set up as a connection to ApplicationUser with its own MVC. Since this API auto generates a lot of files, it would be too massive to show in this diagram. Views are grouped in their respective folders to save space.

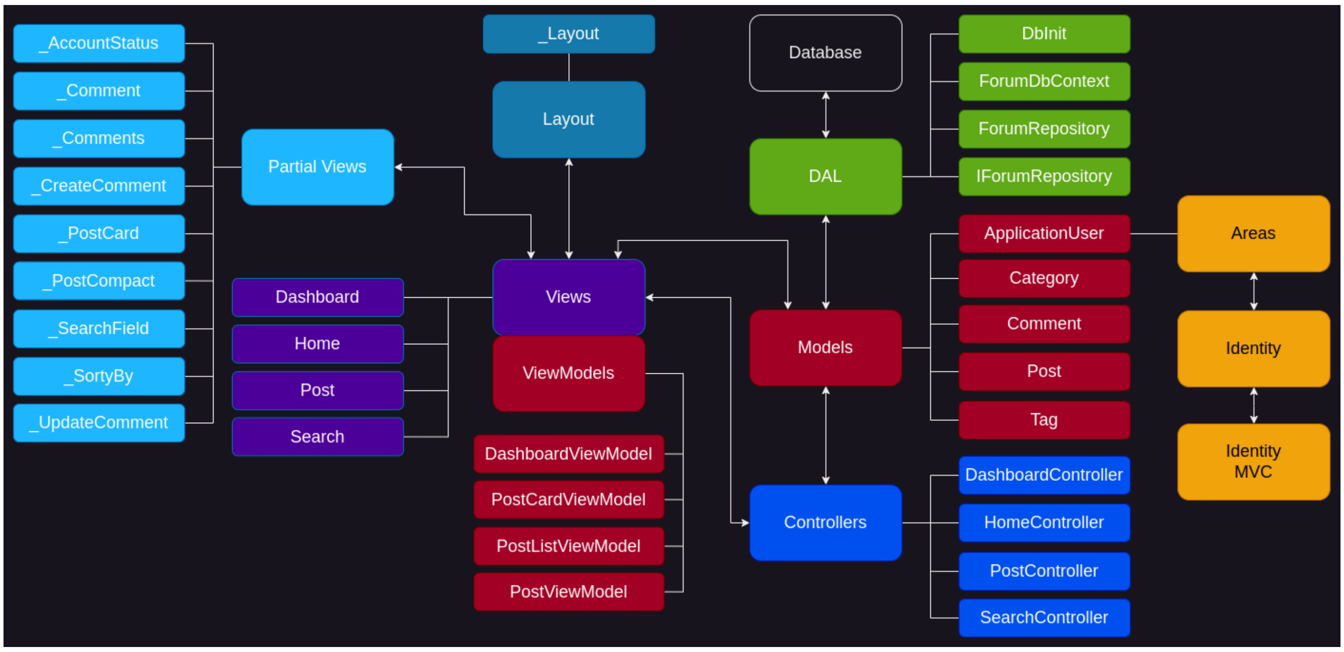

Figure 1: Diagram of the project architecture

The following sections break down the architecture into its different layers and component and providing detail for how they contribute to the application as a whole.

2.1 Models

This layer contains the models that represent the data in the application. Our models includes ApplicationUser, Category, Comment, Post and Tag. These models contain the properties and attributes that defines their structures in the database. We have defined which attributes each model should have, what types of data, navigation properties for handling relationships, setting optional and required feels, set temp data, and descriptive metadata. They also contain getters and setters to be able to fetch and update data. The fields where a user can input data contains RegEx and limitations for input validation. ApplicationUser inherits IdentityUser, which is generated by the Identity package. We have set navigation properties to connect user to the content, added profile picture and added DateTime for account creation.

2.2 Views

This layer contains the Razor Views that are used to render the user interface. When a controller needs to render a page, it passes the appropriate model to the view. The view then uses the model to generate the HTML output for the page. Our views are grouped by dashboard, home, post, search and shared.

2.2.1 Layout

The layout is the primary template for the user interface. This ensures consistent layout across all our pages. We have a header with our logo, navigation links (and hamburger menu for mobile) and a account status which either shows login or the username and picture. The body is a dynamic area that changes based on the different pages. The body is wrapped in a main-inner-container which can be styled consistently accross all pages. @RenderBody() injects the content from the other views. The footer section contains static information with copyright, social media icons and an address. We use external CSS and JS files to style and add interactivity. We have a Favicon with our logo and icons library from FontAwesome among other things.

2.2.2 Other views

As mentioned before, we have many different other views. These are injected into the body based on what page the user is on. Dashboard contains the Views for the user and admin dashboard. The user dashboard contains a list of posts and comments created by the user as well as liked posts and comments. Admin dashboard contains Views for CRUD functionality for categories and tags. Home has the index page with an introduction and cards with the different forum categories. Post includes views for card-view or compact-view as well as creation, deletion and updating of posts. And of course the view for posts themselves. Search contains the search field and search result view.

2.2.3 Partial Views

We have many partial views in our project. These are portions of views that can be rendered within other views. They are designed to render a portion of a view, making them reusable across multiple views. We use this for modularizing some content that we reuse many places. For instance, partial views handle the comment input box (with its validation), account status display, search fields, like updates on posts, and dropdown menus.

2.2.4 ViewModels

ViewModels are designed for presentation in the view. They contain a combination of models tailored for a particular UI requirement. This allows for separating the business logic model from the presentation. We have ViewModels for dashboard, posts and card view and list view for posts. The dashboard and post ViewModel is mainly for enabling the user to select category and tags for posts. The other two are for letting the user choose between card and list view of the posts.

2.3 Controllers

This layer contains the controllers that handle HTTP requests. When a user requests a page, the corresponding controller is responsible for returning the appropriate response. We have four controllers:

DashboardController
Manages user-specific functionalities for both regular users and admins. The controller fetches the appropriate views based on the user role. In adition it also controls CRUD functionality via GET and POST requests for categories and tags.

HomeController
Handles the homepage of the forum. It is responsible for fetching the categories and tags to display on the front page.

PostController
Takes care of all the post-related functionalities. This includes fetching individual posts, all posts, compact/card view of the posts, and CRUD functionalities regarding posts.

SearchController
Manages the search functionality. It handles redirection to posts, finding posts by id and searching for posts based on search criterias amongst others.

2.4 Data Access Layer (DAL)

This layer contains the code that interacts with the database. This code is responsible for performing CRUD operations on the data in the database, based on the models. Our DAL consists of four components:

DbInit
This is the initializer for our database. This is quite large, since we have a lot of dummy data which needs to be initialized when the application is set up. DbInit seeds the database with initial data and ensures that when the application runs for the first time it has the necessary records. Our DbInit contains the categories, tags, roles, users, posts, comments and all their related data.

ForumDbContext
This is the Object-Relational Mapper (ORM) for our forum. This bridges the gap between our object-oriented code and the relational database. That means it sits as a "translator" between the application and the database, by translating object-oriented code into SQL commands and vice versa. It defines data structures and relationships and ensures smooth data operations. It contains getters and setters for the entities in the database, configuration for schemas and entities, as well as creating junction tables for many-to-many relationships. And it enables lazy loading when it is needed.

ForumRepository and IForumRepository
The repository provides a more organized way to access data. The interface defines the methods and properties for the repository, while the repository is the implementation of this. Our repository consists of functions for CRUD operations on entities, posts and tags. These methods are made generic so they can be applied to a variety of entities. This makes our data access more reusable.

2.5 Areas/Identity

We use 'Microsoft.AspNetCore.Identity' for authentication and authorization. This is an ASP.NET Core API that supports user interface for login functionality. It manages users, passwords, profile data and more. Identity creates a Razor Class Library with the 'Identity' area endpoint, e.g. '/Identity/Account/Login'. As Figure 1 shows, Identity generates its own MVC, connected to the different parts of our project. It is using ApplicationUser (child of IdentityUser) as a shared model.

The user is not authorized to do anything other than to browse the forum without an user account. The user must register with a username, email and password. The password must consist of at least 8 characters, one alphanumeric character, one lowercase and one uppercase letter. When a user is authenticated it can create/edit posts, comment and like posts. The user has access to a dashboard with overview of their post, comments and likes. In the upper right corner they have your profile panel. Here they can change username, upload a profile picture, change mail/password, enable 2FA and download all their user data. They can delete their user and data under "personal data". Then all the users posts and comments will be marked as posted by 'Anonymous'.

Identity offers different roles. We have an admin role which has access to an admin dashboard. Here the user can edit, delete or add new categories and tags. They can change both the color, name and picture. An admin can also delete all posts and comments, while a regular user can only delete its own content.

3. Database

Our database consists of 14 tables, as seen in Figure 2. Four of these tables are junction tables to help with the relationships and seven are generated in conjunction to the ASP.NET user identity API. These seven tables are named with the AspNet prefix. We used DBeaver to generate an ER-diagram. For some reason it did not have the right relationship notation for every relationship, as it lacks the notation for some of the many-relationships. The primary key is at the top in bold, and the data type is both showed with an iconand text in caps to the right. Our database can be split into two categories: Content related tables and user related tables.

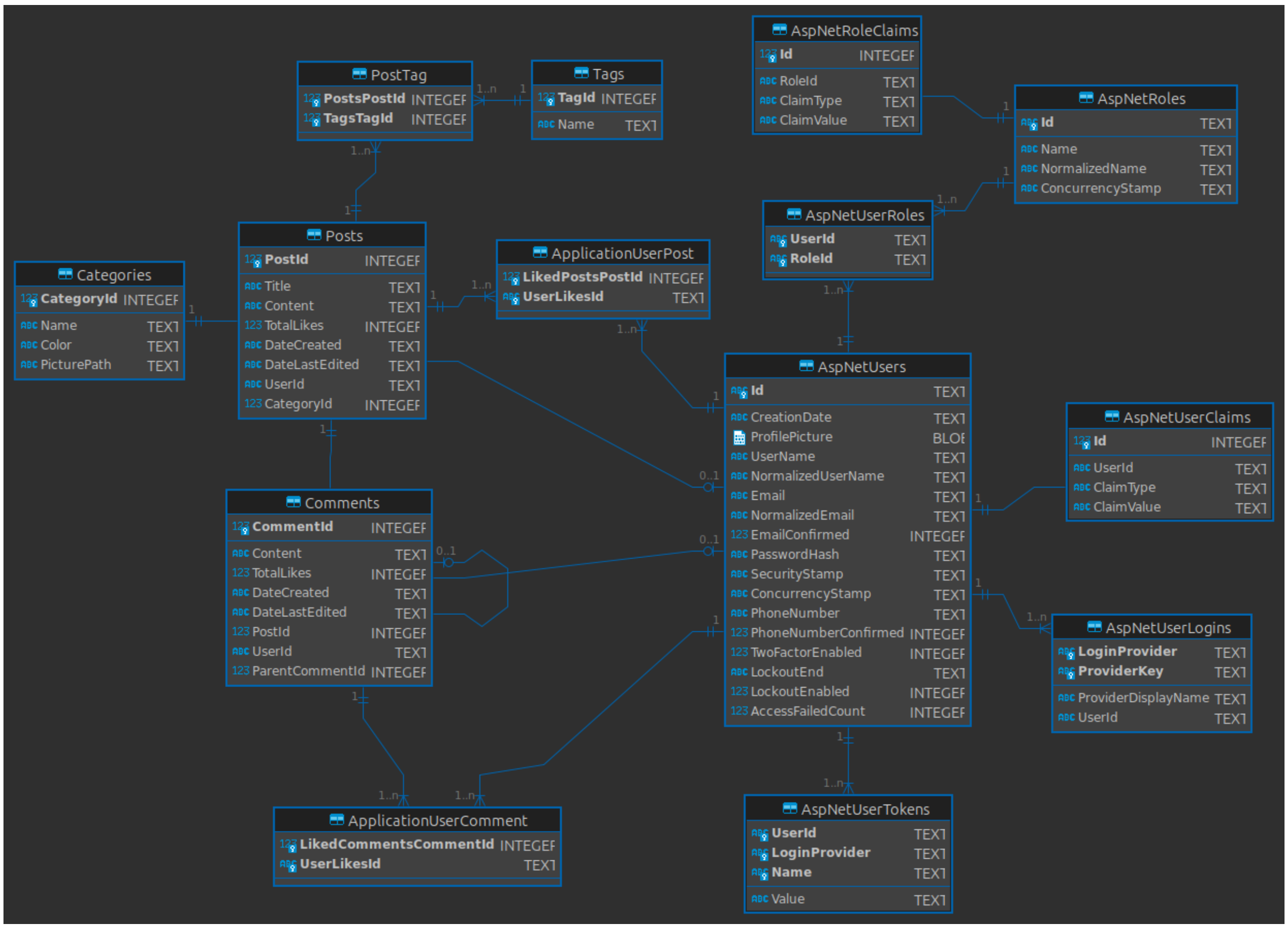

Figure 2: Entity Relations diagram of the forum database

Content
For the content we have a table for Posts, Comments, Categories and Tags, as well as junction tables. Each post belongs to exactly one category, and a category can have many posts. Tags have a many-to-many relationship with Posts, with a junction table between them. Each comment belongs to exactly one post, but each post can have many comments. Posts and Comments have a many-to-

many relationship with Users, with a junction table between them. Comments also has a connection to itself for sub-comments.

User
The AspNetUser part is generated from the Identity package. Each user can have one or many tokens, while a token is connected to only one user. AspNetUserLogins manage the third-party login and AspNetUserClaims the claims each user has. Users has a many-to-many connection to AspNetRoles with a junction table between them. Roles has a Role claim to keep track of the claims.

4. Coding details

We have tried to follow the C# coding convention for syntax and naming conventions to make sure the code is readable and managable. However, we haven't focused much on this, so the code might sometimes break conventions. The code is documented with comments throughout the project. The code should either be self explanatory or explained with comments. We have also provided comments with links where we have found inspiration or code from the internet.

4.2 Design

The main design aspect have been user friendliness. There are consistent use of colors, spacing, font sizes, transition effects, etc. We follow the WCAG guidelines for color contrast, and use colors and hover effects to highlight interactivity. The website uses a dark theme designed to follow certain guidelines from Google's Material Design system but with additions and changes that make it look and feel unique. The website design is responsive, making the user experience good no matter device size. We have used FontAwesome icons for providing a visually appealing interface, and made sure they have a consistent style. The category images are generated by DALL-E 3 to make sure we have the same style for all our pictures. The site also features interactive elements like buttons, links and forms to create a clear feedback to the user.

4.3 File upload

Each category has its own image which is rendered on the home page. We wanted the admin user to be able to also change the image for the category or upload a picture for a new category. We also have file upload for the users profile picture.

For the updating of a category, we first save the old picture path so that we can delete that in the database later. Then we use Request.Form.Files to set the file as the first file uploaded in the http request form. Then we do a file upload, with logging and error handling along the way in case something goes wrong. If it is a success we update the category in the database. We have also put a fail safe where we just enter the old path if the user does not set one.

For the profile picture we have a FileUpload function that takes an IformFile as argument. We have set a max size of 8MB. We check for file size and do logging along the way. We then set fileName to be Guid.NewGuid and Path.GetExtension. We create the full path with the right directories. If the directory does not exist, we create one. Lastly we use the File.Create method. We used the Microsoft documentation on File methods for this.

4.4 CRUD operations

We have CRUD operations for almost all of our tables in the database. There are some limitations for a regular user, like deleting tags and categories. With an admin user, we can do CRUD operations on posts, comments, categories and tags as well as users (except for deletion). Roles are hard coded and do not support CRUD.

The CRUD operations takes place in ForumRepository. As previously mentioned, we made our methods generic, so instead of having a create, update and delete function each, we made them so that they require TEntity or id as an argument. We did this by starting the class with this line «public class ForumRepository<TEntity> : IForumRepository<TEntity> where TEntity : class». We defined a generic class which can work with any type of entity, TEntity and a constraint on the generic type so that TEntity can only be a class and not a value type.

4.5 Validation and error handling

Ensuring that our application is robust and reliable is very important. We have tried to incorporate a lot of validation mechanisms and error handling.

In the front-end we have validation with jQuery and RegEx. This way we ensure that the data that from the user input adheres to the formats we expect. This helps with reducing malicious data or data with errors being sent to the server. All input fields have a RegEx pattern which is quite restrictive. We have HTML sanitization with HtmlSanitizer, to protect against XSS attacks. This is especially important when we support Markdown, and since we are a programming forum.

On the server-side we have made sure that each CRUD operation is encased in try/catch blocks and if-cases. This is setup so that we can gracefully handle unforseen errors or exceptions. Our if statements are used to verify conditions, check for null values, etc. We raises exceptions with meaningful messages to provide information if an error do occur. We also log all errors and exceptions, as well as some informational logging to have a traceable record and help with debugging.

With the Identity package, Microsoft also provides a lot of validation when it comes to the user management. So this is pretty much handled for us.

4.6 Dependencies

We use **Jdenticon.AspNetCore** for generating random profile pictures if the user has not uploaded one. The image is generated by converting the username into a hash, which is used as a unique id for generation. The user can later upload a custom profile picture in their profile settings.

Markdig is a Markdown processor for .NET. This parses Markdown to HTML in the back-end, so we are able to use Markdown in the posts. We use this in conjunction with Simple MDE which renders Markdown in the front-end.

HtmlSanitizer is a .NET library for cleaning HTML. This is used as a validator tool to clean the HTML to prevent XSS attacks, and avoid invalid HTML. We use this so users are able to post tags and scripts in posts, without the Markdown processor rendering the code.

5. Conclusion

This project has been a really good kick start when it comes to our bachelor assignment. Since we are also going to write our bachelor together and the assignment is a web application in C# and ASP.NET Core, this has been invaluable to us. We have both learned what to expect from each other, how to work smoothly together as a team and learned a lot about very relevant technologies. We have also been able to see some room for improvements in communication and structure, to work better as a team next time.

We think that our forum showcases our abilities and goes beyond just being a simple MVP and proof-of-concept. We look forward to implement a framework for the front-end and see how we can improve upon our work.

5.1 Limitations

Even though this is supposed to be a MVP solution, we have decided to mention some of the limitations of our application.

We don't have the ability for password reset or authentication of email. This is because the Identity API requires us to set up an email server to do this, and we see that as out of scope for this assignment. We did not add the posibility to delete users either, as we saw no need for it, however, for a real forum that would be essential.

The setup for JavaScript and CSS could be better. Almost all scripts and stylesheets needs to load. To reduce bandwitdh and the amount of network requests, it would have been better for them to load only when they are being used. Since the point of this assignment was working on back-end, we didn't prioritize fixing that.

The error messages from post controller is not optimal. The message turns up on a new site with only the error text. There are no layout or even a back button, so the user has to press the back button on the browser. We did not have enough time to fix this unfortunately.

Since we are dependent on a bunch of seed data, there are a lot of SQL statements being ran when the application starts. This creates some warnings. However, if this was a real forum we would not have needed to seed that much data.

6. Contributions

Rasmus has taken the lead on all of the back-end, with rubber ducking and quality assurance from Jørgen. Rasmus has done the main work when it comes to the database, controllers, DAL and identity. Ole Kristian has taken the lead on all of the front-end. He has done the main work on views, ViewModels and design. Jørgen did rubber ducking and quality assurance on the back-end. He has checked, formated and commented code. He has done some error handling and logging, and he has written all the documentation for this project.

7. References

We have provided comments with links where we have found inspiration or code from the internet. These links are provided at the relevant code. Here is a list of sources we have used:

User management and authentication:
https://codewithmukesh.com/blog/user-management-in-aspnet-core-mvc/

https://learn.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-7.0&tabs=visual-studio

https://stackoverflow.com/questions/29485285/can-not-find-user-identity-getuserid-method

https://itecnote.com/tecnote/r-unable-to-resolve-service-for-type-iemailsender-while-attempting-to-activate-registermodel/

https://learn.microsoft.com/en-us/aspnet/core/security/authentication/identity-configuration?view=aspnetcore-7.0

https://learn.microsoft.com/en-us/aspnet/core/security/authentication/cookie?view=aspnetcore-7.0&viewFallbackFrom=aspnetcore-3.0

File management:
https://learn.microsoft.com/en-us/dotnet/api/system.io.file.create?view=net-6.0

https://learn.microsoft.com/en-us/dotnet/api/system.io.file.delete?view=net-6.0

Error handling and input validation:
https://weblog.west-wind.com/posts/2018/Aug/31/Markdown-and-Cross-Site-Scripting

https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes/pattern

Database:
https://stackoverflow.com/questions/48202403/instance-of-entity-type-cannot-be-tracked-because-another-instance-with-same-key

https://learn.microsoft.com/en-us/ef/core/modeling/relationships/many-to-many

https://learn.microsoft.com/en-us/ef/core/querying/sql-queries

User comments and posts
https://www.cardenhall.com/wp-content/uploads/2016/10/POSITIVE-COMMENTS-List.pdf

ChatGPT is used for generating user posts

https://github.com/sparksuite/simplemde-markdown-editor

Model View Controller:
https://dotnettutorials.net/lesson/generic-repository-pattern-csharp-mvc/

https://learn.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application#implement-a-generic-repository-and-a-unit-of-work-class

Styling:
https://www.paulirish.com/2009/random-hex-color-code-snippets/

https://stackoverflow.com/questions/62408646/select-an-object-property-within-an-attribute-of-the-dom

https://www.w3schools.com/jsref/prop_node_parentelement.asp

https://jdenticon.com/#quick-start-asp-net-core

FontAwesome for icons

Google Fonts for fonts

Category images are generated by DALL-E 3