RealEstateAPI

Balogh Szilárd, Vitályos Norbert, Opra-Bódi Botond

June 24, 2023

Part I
Introduction

The following documentation contains a comprehensive description of our Web API project for the "Dezvoltarea Aplicațiilor Web și Mobile cu Angular, .Net si Android" course within the Faculty of Mathematics & Informatics, University of Transilvania. It aims to provide an overview of the project's goals, architecture, implementation details, and instructions on how to use the project, while also discussing the difficulties that arose during the development process. In the following sections, we will explore each aspect of our project in detail, providing step-by-step instructions, code snippets, and any additional information necessary to understand and utilize the project effectively.

1 Motivation

RealEstateAPI is a University Project. That's why it is not intended for any real-world use, outside the scope of the course. The goal of this project isn't to develop something new, but to learn more in depth about backend and API development, CRUD applications and working in a team. That's why one of our biggest priorities was to utilize technologies that facillitate the development process and let the developer team work together effectively. We used widely accepted coding practices that improve code quality and readability, while also paying special attention to efficiency, ease of use and intuitive design.

2 Topic

2.1 What is RealEstateAPI?

RealEstateAPI is a Web API that uses technologies like ASP.NET Core for backend development, Entity Framework Core for seamless integration and querying of the database, and Microsoft SQL Server for storing information in a database (More in Technologies Used section). RealEstateAPI is an Announcement-based API, meaning that, the central elements are constituted by Announcements. These are containers of information, offering details about users, uploaded Real Estates and comments.

2.2 Brief Walkthrough of RealEstateAPI

RealEstateAPI can be accessed without authenticating, "Guests" (Anonymous users) being able to view the announcement feed, filter announcements, and view details about specific announcements. Other actions require authentication. Adding an announcement, for example, requires a user account to be created, so that the uploaders of the announcements can be identified. This can easily be done by our AccountController's Register and Login endpoints.

After logging in, the user receives a token, that is necessary for some operations, like adding or deleting announcements, sending comments, etc. (More in Security). It is important to note, that during the development of RealEstateAPI our first priority was security. That's why we used different technologies to make sure that all users can safely and securely enter their information, without ever worrying about it being stolen (More in Database Development and Technologies Used).

Users have a large variety of operations they can choose from. These include viewing the announcement feed, filtering announcements, adding and deleting announcements, sending comments (More in Endpoints). Moreover, we also included an Administrator Role, who is able to access, modify or delete announcements and comments, so that no inappropriate content is uploaded and persisted in the database (More in User Roles).

Part II
Features

This part aims to document all the implemented features as seen by the end user of the API.

3 Authentication & authorization

Authentication and authorization play crucial roles in securing access to resources and ensuring the integrity of the application. Authentication involves verifying the identity of users attempting to access the API, meanwhile, authorization determines the level of access granted to authenticated users.

3.1 Authentication

In our application, the authentication is realized via the Login and Register endpoints. For registration, we use a data transfer object called "RegisterDto", which is passed as a parameter to a function in a class called "UserService" (Services layer), which checks if the email is not already taken and if it is formatted correctly. If both conditions are met, a hash function is used to codify the password, a new user object is created based on the DTO, and the newly created user is inserted into the database.

When a user wants to log in, we first check if the email and password are correct. If the login is successful, a JWT token is sent back to the user. The hashing of the password and the token generation fall within the responsibility of a service called "AuthorizationService".

3.2 Authorization

Authorization is done via the "[Authorize]" attribute used in AnnouncementController. With its help, we can differentiate the operations that can be done by either a regular user or an administrator. For instance, any type of user has the right to view all the announcements, but only a regular user can add new announcements. Furthermore, a regular user is allowed to delete their own announcement, meanwhile, an admin has the right to delete anyone's announcement.

4 Endpoints

Account

POST `/account/register`

Description Registers a new user
Authorization *none*
Request body *`RegisterDto`*
Parameters *none*
Responses

Code	*Description*	*Body*
200	successful registration	
409	EmailAlreadyRegistered: the email address is already registered	
400	InvalidEmailFormat: the format of the email address is invalid	

POST `/account/login`

Description Creates a JWT token for a login session
Authorization *none*
Request body *`LoginDto`*
Parameters *none*
Responses

Code	*Description*	*Body*
200	successful login	{ "token": <string> }
400	ResourceMissing: There is no user with the given email address	
403	Forbidden: Invalid login credentials	

User

GET `/user`

Description Get all users
Authorization *none*
Request body *none*
Parameters *none*
Responses

Code	*Description*	*Body*
200		`[ UserDto ... ]`

GET `/user/{id}`

Description Get all users
Authorization *none*
Request body *none*
Parameters

	Name	*Description*
Route	`id`	user ID

Responses

Code	*Description*	*Body*
200		`UserDto`
400	ResourceMissing: User with id {id} doesn't exist	

GET `/user/announcements`

Description Get all announcements of the currently authenticated user
Authorization Roles=User
Request body *none*
Parameters *none*
Responses

Code	*Description*	*Body*
200		`FeedAnnouncementDto`

GET `/user/comments`

Description Get all comments of the currently authenticated user
Authorization Roles=User
Request body *none*
Parameters *none*
Responses

Code	*Description*	*Body*
200		`CommentDto`

Announcement

GET `/announcement`

Description Get announcement feed with filtering, sorting, and pagination
Authorization *none*
Request body *none*
Parameters

	Name	*Description*
Query	`AnnouncementFilterDto`	filtering criteria
	`AnnouncementSortDto`	sorting criteria
	`page`	page index

Responses

Code	*Description*	*Body*
200		`[ FeedAnnouncementDto ... ]`

POST `/announcement`

Description Adds a new announcement
Authorization Roles=User
Request body `AddAnnouncementDto`
Parameters *none*
Responses

Code	*Description*	*Body*
201	successfully added announcement	

GET `/announcement/{id}`

Description Get announcement details
Authorization *none*
Request body *none*
Parameters

	Name	*Description*
Route	`id`	announcement ID

Responses

Code	*Description*	*Body*
200		`DetailedAnnouncementDto`
400	ResourceMissing: Announcement with id {id} not found	

PATCH `/announcement/{id}`
Description Updates an existing announcement
Authorization Roles=User
Request body *`UpdateAnnouncementDto`*
Parameters

	Name	*Description*
Route	`id`	announcement ID

Responses

Code	*Description*	*Body*
200	successfully updated announcement	
400	ResourceMissing: Announcement with id {id} not found	
401	Forbidden: Unauthorized to update announcement with id {id}	

DELETE `/announcement/{id}`
Description Deletes an existing announcement
Authorization Roles=User,Administrator (if User, authenticated user is the poster)
Request body *none*
Parameters

	Name	*Description*
Route	`id`	announcement ID

Responses

Code	*Description*	*Body*
200	successfully deleted announcement	
400	ResourceMissing: Announcement with id {id} not found	
401	Forbidden: Unauthorized to delete announcement with id {id}	

GET `/announcement/{id}/comments`
Description Get all comments on an announcement
Authorization *none*
Request body *none*
Parameters

	Name	*Description*
Route	`id`	announcement ID

Responses

Code	*Description*	*Body*
200		`[ CommentDto ... ]`
400	ResourceMissing: Announcement with id {id} not found	

POST `/announcement/{id}/comments`
Description Adds a new comment on an announcement
Authorization Roles=User
Request body *`AddCommentDto`*
Parameters

	Name	*Description*
Route	`id`	announcement ID

Responses

Code	*Description*	*Body*
201	successfully added comment	
400	ResourceMissing: Announcement with id {id} not found	

DELETE `/announcement/{id}/comments/{commentId}`

Description Deletes an existing comment on an announcement

Authorization Roles=User,Administrator (if User, authenticated user is the poster)

Request body *none*

Parameters

	Name	*Description*
Route	`id`	announcement ID
	`commentId`	commentId

Responses

Code	*Description*	*Body*
200	successfully deleted comment	
400	ResourceMissing: Announcement with id {id} not found	
400	ResourceMissing: Comment with id {commentId} not found	
401	Forbidden: Unauthorized to delete comment with id {commentId}	

5 Data types

Throughout our API, we used different kinds of data types, like int, string, DateTime. All of them are presented in the "Database schema" section. We also created some custom data types, called DTOs as endpoint parameters and return types. The main purpose of a DTO is to provide a standardized format for exchanging data and to decouple the data representation from the underlying domain model. For example, we created DTOs for logging in and registering, adding comments and announcements etc.

Part III
Implementation details

6 Technologies used

Our tech stack is almost entirely Microsoft:

- *Visual Studio* as IDE
- *.NET Core* as development platform
- *C#*
- *ASP.NET Core* as the web framework
- *Entity Framework Core* as the object-relational mapper for database interaction
- *SQL Server* as the database management system

We also use *Swagger* for interactive API testing, through the *Swashbuckle* NuGet packages.

7 Database schema

The database was created using EF Core migrations. Migrations are a feature that allows to manage and evolve a database schema over time. We created an initial migration in which we created the Users, Comments and Announcements tables. Later, we used the update command on the time of the creation of a comment. The relationships between the tables are shown in the figure below.

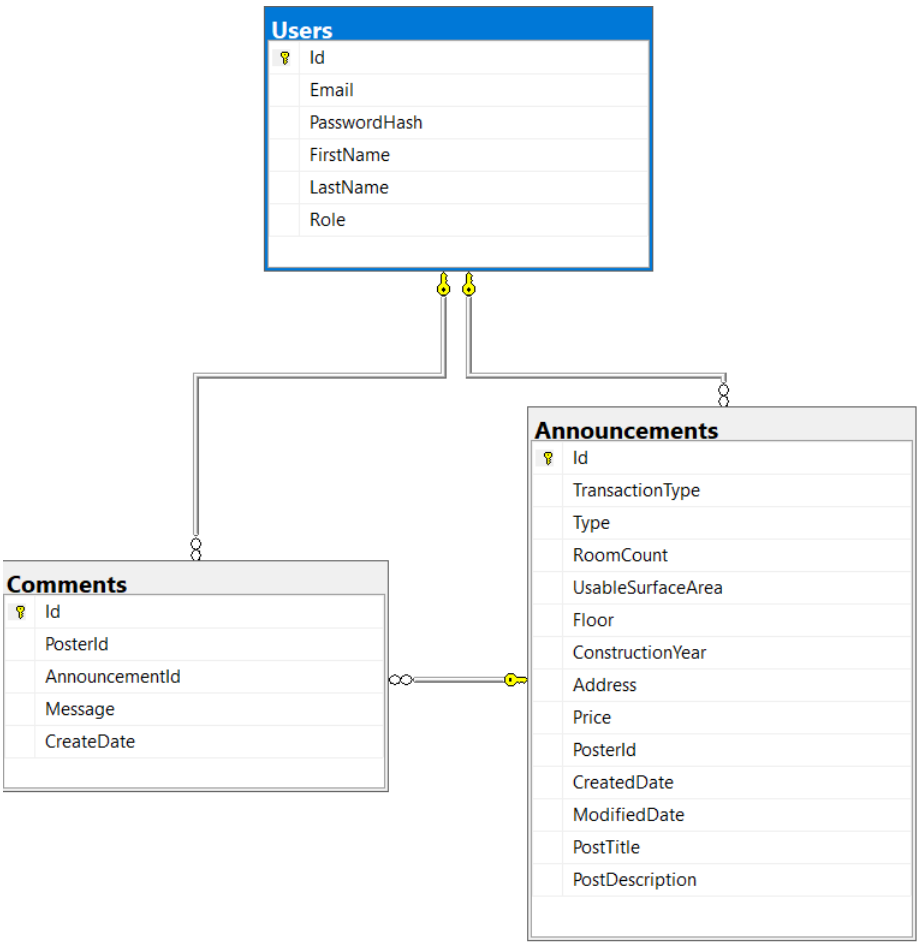

Figure 1: Database schema diagram

7.1 Data types used in the Database

7.1.1 Data types used in the Users table

Field	Data type in the application	Data type in the database
Id	Guid	uniqueidentifier
Email	string	nvarchar(MAX)
PasswordHash	string	nvarchar(MAX)
FirstName	string	nvarchar(MAX)
LastName	string	nvarchar(MAX)
Role	enum	real

7.1.2 Data types used in the Announcements table

Field	Data type in the application	Data type in the database
Id	Guid	uniqueidentifier
TransactionType	enum	nvarchar(MAX)
Type	enum	nvarchar(MAX)
RoomCount	int	int
UsableSurfaceArea	float	real
Floor	int	int
ConstructionYear	int	int
Address	string	nvarchar(MAX)
Price	decimal	decimal
PosterId	Guid	uniqueidentifier
CreatedDate	DateTime	datetime2
ModifiedDate	DateTime	datetime2
PostTitle	string	nvarchar(MAX)

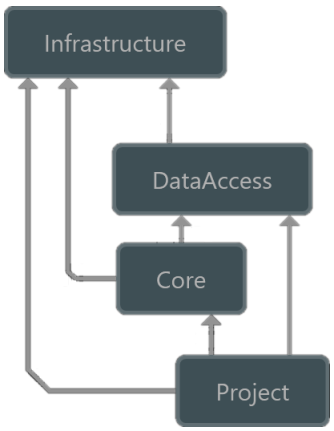

Figure 2: Dependencies between the projects

7.1.3 Data types used in the Comments table

Field	Data type in the application	Data type in the database
Id	Guid	uniqueidentifier
PosterId	Guid	uniqueidentifier
AnnouncementId	Guid	uniqueidentifier
Message	string	nvarchar(MAX)
CreateDate	DateTime	datetime2

7.2 Delete constraints

The delete constraints ensure referential integrity between the tables. When a referenced row in the parent table is deleted, the corresponding rows in the child tables are also deleted. In our database, the delete constraints are given by the foreign key constraints.

7.2.1 Foreign key constraint in the "Announcements" table

The foreign key constraint is defined by the PosterId. It references the "Users" table and the "Id" column as the primary key. When a user is deleted, all the related announcements with the matching "PosterId" will be deleted.

7.2.2 Foreign key constraint in the "Comments" table

The foreign key constraint is defined by the AnnouncementId. It references the "Announcements" table and the "Id" column as the primary key. When an announcement is deleted, all the related comments with the matching "AnnouncementId" are also deleted.

8 Solution structure

We implemented a 3-layer architecture as is common for web apps of this size. Our Visual Studio solution is structured in 4 projects: `Infrastructure`, `DataAccess`, `Core`, `Project`. With this, we tried to ensure a clean separation between the different concerns of the application. The dependencies between the projects is shown on Figure 2.

8.1 DataAccess

The data access layer is the one responsible for containing the entities, enums, migrations and repositories. Regarding the entities, the most significant difference compared to the database tables is that we also have a real estate entity. It, however, isn't converted into a separate table in the database, as the real estate is contained by the announcement.

The repositories are responsible for data acces and abstracting away the database-specific details. They provide methods for querying, inserting, updating, and deleting data from the database. In our application, we have repositories for all of our tables, all of which inherit from the "RepositoryBase", which is a generic class and includes all of the CRUD operations. The "RepositoryBase" relies on a database context, which represents

the connection to the database. The context is also responsible for managing the database connection, tracking changes, and executing queries.

We also implemented a "UnitOfWork" class, which is responsible for managing the transactions and coordinating the persistence of changes made to the entities within the application. It exposes properties for accessing all of the repositories within a logical unit of work. The "SaveChanges()" and "SaveChangesAsync()" methods are responsible for persisting the changes to the database. The "UnitOfWork" class holds a reference to the "AppDbContext" object, which is injected into the "UnitOfWork" constructor and shared among the repositories. This ensures that all the repositories within the "UnitOfWork" work with the same database context, enabling the coordination of changes and transaction management.

8.2 Core

The role of the "Core" project is to encapsulate the core business logic, domain models, and services. As such, our "Core" project contains all the necessary DTOs and Enums for the services. The DTOs include "AddAnnouncementDto", "AddCommentDto", "LoginDto", "RegisterDto", "UserDto". The enums are used for sorting purposes. We created some extension methods, the main purpose of these is to facilitate the conversion of a regular object to a DTO or vice versa.

The "AnnouncementService" and "UserService" use a "UnitOfWork" (described in the above section) through dependency injection. They contain all of the business logic of the application, like filtering the announcements, checking if an email address is valid etc. They are also responsible for retrieving, inserting and modifying data with the help of "UnitOfWork". The "UserService" also uses an "AuthorizationService" for logging in and registering. The authorization is described in the corresponding section.

8.3 Project

This project houses the ASP.NET Core related functionality and is the entry point of the application. It contains controllers which implement endpoints, sets up dependency injection and configures the web API server. The configured services include authentication and authorization, custom middleware, Swagger for testing purposes etc.

8.4 Infrastructure

In the infrastructure project, there are two folders: Exceptions and Middlewares. These two are closely related as the only middleware we implemented is the one responsible for handling all the exceptions. For the exception-handling middleware to work, we had to configure the Program.cs file accordingly. The exceptions we created include EmailAlreadyRegisteredException, InvalidEmailFormatException, ForbiddenException, and ResourceMissingException. Thanks to the middleware, we can handle all the exceptions that occur in one centralized file.