BBB[1]: RESTful API

Board-games library and management system

Student: Paraskevi-Dimitra Simaresi

Team: 20

1 Introduction

This report documents the results and reflections of the web application titled “BBB – Boring Boardgames Bureau”, developed as part of the group project and individual extension for the Web Technologies course.

The system is a publicly accessible catalogue and management platform for board games, reflecting a real-world use case. It was developed using the ASP.NET Core 9 framework following the MVC architecture and allows the “SDU Boardgames” club to manage its board game collection[2] and handle borrowing operations.

The focus of the individual extension is the introduction of RESTful APIs. This extension was motivated by limitations encountered during development, as all interactions were handled through MVC views, reducing scalability and integration potential. In addition, the lack of API documentation, such as Swagger, made debugging and further development more challenging.

The main contributions of this work include the refactoring of existing controllers to follow RESTful API conventions, the integration of Swagger for API documentation and testing, and the implementation of role-based API endpoints for administrative operations.

2 Reflections

Front-end

Front-end development meets the initial requirements, offering a simple, responsive UI with easy navigation. The team chose not to use a front-end framework, building directly with HTML, JavaScript, and CSS. This directly led to challenges like manual DOM management and less structured component organization. However, code is modular with reusable components (footer, sidebar, navbar), semantic HTML tags, and CSS/JS organized by feature.

The feature I am most proud of is the modular and component-based HTML structure, as I was directly involved and is the foundation for a scalable application. Additionally, the application would benefit from the introduction of a Framework like React paired with a build tool (Vite) and a utility CSS framework(tailwind).

Resource-Management

The strength of this topic’s implementation is the use of dependency injection for DbContext and services it follows ASP.NET patterns and there are no memory leaks. The main struggle was asynchronous programming and implementing Dependency Injection, where it was a key challenge for me and I relied on guidance from more experienced team members.

Implementation of Dependency Injection allows for loose coupling improving maintenance. Important is the use of *using* statement for disposable resources to ensure proper disposal. Area for improvement is on error handling for database failures and on consistent naming.

Authentication & Authorization

Authentication and authorization are implemented and functional. The main strengths are the use of industry-standard PBKDF2 with salt for password hashing. Challenging was keeping validation and error handling consistent. Code is readable following C# conventions.

The use of PBKDF2 is a valuable takeaway, as it reflects real-world practices. But currently Controllers handle both business logic and authentication. To improve controllers should focus on request handling and authorization should separate into custom filters for role-based validations.

3 Individual Extension

Motivation

The group project application was built as a typical ASP.NET Core MVC application with client-server interactions were handled through views and form posts. While it resulted in a functional system it had limitations on flexibility and responsiveness and limited integration possibilities with future mobile apps. By implementing RESTful API, the system becomes scalable and future-proof.

Technical description

Controllers were mostly impacted as the business logic refactored into API controllers (*AccountController, AdminController, GamesController, AuthController*) using *[ApiController]* and *[Route("api/[controller]")]* conventions. While Razor pages are still served by view controllers. As a result, all core operations, such as authentication, admin actions and game borrowing are performed as JSON endpoints adhering to

[1] BBB is abbreviation of *Boring Boardgames Bureau*

[2] The Boardgames are not fully and officially registered, though sources from within the club estimate the number of Boardgames to be around 80.

RESTful principles for CRUD, GET for retrieval, POST for creation, PUT/PATCH for updates, DELETE for removal and route naming conventions (e.g., /api/auth/, /api/admin/games/). Session-based validation was preserved and with validation performed for each API call.

Part of the front-end was refactored to align. A centralized JavaScript API client(*apiClient.js*) was created to handle HTTP requests and errors allowing for abstraction. An additional service layer(*services.js*) provide reusable functions for abstract API calls. Logic adjusted to use AJAX/fetch for data interactions, no form post or full-page reloads are performed. Razor Views remained for initial page load and layout, and dynamic data is loaded through API.

Model validation on backend remained. HTTP status codes and JSON error responses are forwarded from the Controllers to apiClient and through the UI and displayed to the user. Admin-only endpoints are protected by querying user's role (*RequireRoleAttribute*) and check in Controllers by applying the pattern *[RequireRole(UserRole.Admin)]*. Swagger integration enabled API documentation and testing use cases

Reflections

Refactoring benefited by clear separating concerns. The API can be re-used for other clients. Consistent JSON responses and use of swagger serve better error handling and debugging. Especially challenging and time consuming was separating View Controllers logic from API Controller logic and migration of front-end logic to use APIs, specifically Form posts. Prioritized future expansions could be implementation of JWT (to also expand my knowledge), API versioning and automated endpoint testing.

4 Security Reflections

Passwords are hashed using PBKDF2 hasher (*Services/PBKDF2Hasher.cs*) with unique salt per user whereas Auth entity stores hash and salt and plain text password is never stored. Password hashing logic was maintained for the RESTful APIs for the individual extension. Auth endpoints do not expose hashes or salts.

Database uses Entity Framework Core with SQLite and operations are performed via the *AppDbContext.cs* with relationships defined in *Configurations/.* To prevent accidental data loss Cascade, delete was avoided. For the individual extension API controllers validate users and sessions before a database operation.

To mitigate the possibility of SQL Injection Entity Framework, parametrizes queries. Session Hijacking is avoided by storing session to HTTPOnly cookies.

The extension by exposing RESTful endpoints increased the attack surface. But authentication and role checks (for admin operations) are enforced. Also input validation is performed server-side (use of Models) and client-side and errors do not leak sensitive data. Moreover, secure HTTP would be mandatory in production.

Additionally, swagger is available only for development.

5 Performance & Scalability Reflections

During development I noticed extensive database querying on entities loading (fetching games), SQL logs helped identifying the issue. Also some endpoints returned unnecessary data, increasing response time and data exposure which was identified by inspecting web browser's dev tool. For counting resources direct SQL count queries avoided loading all entities in memory. Conscious decisions include the limited delay added as API calls check session's validity and skipping caching for simplicity and data consistency.

Based on Performance tool on browser's dev tool local LCP(largest Contentful Paint) is 1.32s (good) and CLS (Cumulative Layout Shift) is 0 with 4x CPU slowdown and fast 4G network throttling.

RESTful API design improves scalability by separating server and client responsibilities. Its structure supports future migration to stateless authorization. Also, connection pooling in relation with a non-file and more scalable DB (e.g., PostgreSQL) will serve multiple users concurrently. Further optimizing querying will reduce loading of unnecessary data. Also caching could store frequently used data.

The impact from the individual extension was improving separation of server and client-side concerns serving efficient updates and less page reloads. The newly added JS apiClient centrally handles errors reducing redundant calls. Also, Admin routes perform bulk operations in a single API call.

6 Conclusions

This report documented the refactoring of the *BBB Board-games System* through the implementation of RESTful APIs as individual extension. Refactoring system's codebase included the Controllers for clear separation of View and JSON formatted data. Moreover, introducing a central JS apiClient and enforce role-based routing access.

Key takeaways include the improved separation of concerns server and client side, enhanced scalability across different clients and maintainability by introducing swagger for API documentation. This resulted in a system which meets original requirements. Laid also the foundations for future integration, like mobile clients and third-part applications

Security was prioritized by using PBKDF2 password hashing and session validation. While performance was optimized by reducing querying and transferring unnecessary data.

In the future the system could benefit from JWT authentication, API versioning, automated API call testing and migration to a more scalable database.

Use of AI

Generative AI was utilized within the guidelines set by SDU and as assisting tool.