[Technical Design Document 2](#_Toc3056)

[Overview 2](#_Toc10516)

[Technology Stack 2](#_Toc8300)

[Dotnet 2](#_Toc27404)

[EFCore 2](#_Toc25186)

[Clean Architecture 2](#_Toc31534)

[Repository Pattern 3](#_Toc26387)

[PostgreSQL Database 3](#_Toc10361)

[Application Architecture Diagram 3](#_Toc7482)

[Physical and Network Architecture 4](#_Toc31629)

[Tools and Technologies in use 5](#_Toc13438)

[Book Reservation API Info 5](#_Toc12504)

[1. CRUD API for the Book Entity 5](#_Toc7543)

[2. POST Endpoint to Reserve a Book 5](#_Toc27909)

[3. POST Endpoint to Remove Reserved Status 6](#_Toc20168)

[4. GET Endpoints for Reserved and Available Books 6](#_Toc17967)

[5. Swagger UI 6](#_Toc8218)

[6. Database and Docker 6](#_Toc20325)

[Status History (Additional Task) 6](#_Toc21002)

[Book Reservation System Project Structure 7](#_Toc380)

[/BookReservationSystem ├── BookReservationSystem.sln ├── BookReservationSystem.WebApi │ └── BookReservationSystem.WebApi.csproj ├── BookReservationSystem.Application │ └── BookReservationSystem.Application.csproj ├── BookReservationSystem.Infrastructure │ └── BookReservationSystem.Infrastructure.csproj └── BookReservationSystem.Domain └── BookReservationSystem.Domain.csproj 7](#_Toc21046)

[Deployment Steps in IIS 7](#_Toc29508)

[Appsettings Property Description 8](#_Toc1302)

[Entity Relationship Diagram(ER diagram) 8](#_Toc9373)

[Error Handling Middleware 8](#_Toc27463)

[Overview 8](#_Toc31486)

[Implementation 9](#_Toc26216)

[Unit test 9](#_Toc17261)

# 

# 

# **Technical Design Document**

## **Overview**

This document outlines the technical design and architecture of the application. It includes a detailed description of the technology stack, logical architecture, deployment steps, and other essential aspects to ensure the smooth development, deployment, and maintenance of the application.

## **Technology Stack**

### **Dotnet**

.NET is a developer platform made up of tools, programming languages, and libraries for building many different types of applications. The main component used in this application is ASP.NET Core, which is a cross-platform, high-performance framework for building modern, cloud-enabled, Internet-connected applications.

### 

### **EFCore**

Entity Framework Core (EF Core) is a modern object-database mapper for .NET. It supports LINQ queries, change tracking, updates, and schema migrations. It is used to interact with the database using strongly-typed .NET objects, which streamlines database access and reduces boilerplate code.

### **Clean Architecture**

Clean Architecture is a software design philosophy that emphasizes the separation of concerns. It ensures that the business logic is independent of UI and infrastructure concerns, making the system easier to test and maintain. The architecture typically includes layers such as:

* **Domain**: Core business logic.
* **Application**: Application-specific business rules.
* **Infrastructure**: Interaction with external systems.

### **Repository Pattern**

The repository pattern is used to encapsulate data access logic, providing a cleaner separation of concerns. It allows for more flexible data access strategies and easier unit testing. Each entity has its own repository(optional), which contains methods for CRUD operations. We have here implemented a generic repository pattern for ease of use.

### **PostgreSQL Database**

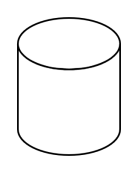
PostgreSQL is an advanced, open-source, multi-model relational database management system. It is widely used for running online transaction processing (OLTP), data warehousing (DW), and mixed (OLTP & DW) database workloads. PostgreSQL ensures data integrity, supports complex queries, and provides robust security features for storing and managing application data efficiently.

## **Application Architecture Diagram**

Browser

IIS

Book Reservation System



postgres

*Figure: Architectural representation of Book Reservation System*

## 

## 

## 

## 

## 

## 

## **Physical and Network Architecture**



Book Reservation System

Postgres



**Database Server**

**Application Server**

*Figure: Network Architecture for Book Reservation System*

# 

# 

# **Tools and Technologies in use**

|  |  |  |
| --- | --- | --- |
| **Tool** | **Type** | **Usage** |
| Postgres | Open source | Database |
| IIS | Closed source | Web Server |
| AspNetCore 8 | Open source | Platform(web development tool) |
| EntityFramework Core | Open Source | ORM framework |
| C# 12 | Open source | Programming language |

## 

## 

# **Book Reservation API Info**

### ****1. CRUD API for the Book Entity****

* **GetBooks**: Fetches all books (GET /api/books)
* **GetBook**: Fetches a specific book by ID (GET /api/books/{id})
* **AddBook**: Adds a new book (POST /api/books)
* **UpdateBook**: Updates an existing book (PUT /api/books/{id})
* **DeleteBook**: Deletes a book (DELETE /api/books/{id})

These endpoints handle the basic CRUD operations well.

### ****2. POST Endpoint to Reserve a Book****

* **ReserveBook**: This method (POST /api/books/{id}/reserve) reserves a book, sets IsReserved, and logs a comment. If the book is already reserved, it returns an error with a 409 Conflict status, which matches the requirement.

### ****3. POST Endpoint to Remove Reserved Status****

* **UnreserveBook**: This method (POST /api/books/{id}/unreserve) removes the reserved status. If the book is not reserved, it returns an error with a 400 BadRequest. This also matches the requirements.

### ****4. GET Endpoints for Reserved and Available Books****

* **GetReservedBooks**: Retrieves the list of reserved books with comments (GET /api/books/reserved)
* **GetAvailableBooks**: Retrieves the list of available (not reserved) books (GET /api/books/available)

These two endpoints meet the requirements for fetching reserved and available books.

### ****5. Swagger UI****

* Swagger UI has been included to interact with the API.

### ****6. Database and Docker****

* In this project, repositories are used for data access, allowing flexibility to switch between different databases. For this implementation, a PostgreSQL database is used, and the database engine is containerized. A **dockerbuilder.sh** file is added to streamline the process of building the Docker image for the project. Additionally, a **docker-compose.yml** file has been created to containerize both the PostgreSQL database and the project itself locally. This setup enables easy deployment and management of both the application and the database within containers.

### ****Status History (Additional Task)****

* BooksStatusHistory entity that records changes in the reservation status of a book. The ReserveBookAsync and UnreserveBookAsync methods already log status changes, which is aligned with the optional additional task.

# **Book Reservation System Project Structure**

### /BookReservationSystem ├── BookReservationSystem.sln ├── BookReservationSystem.WebApi │ └── BookReservationSystem.WebApi.csproj ├── BookReservationSystem.Application │ └── BookReservationSystem.Application.csproj ├── BookReservationSystem.Infrastructure │ └── BookReservationSystem.Infrastructure.csproj └── BookReservationSystem.Domain └── BookReservationSystem.Domain.csproj

## **Deployment Steps in IIS**

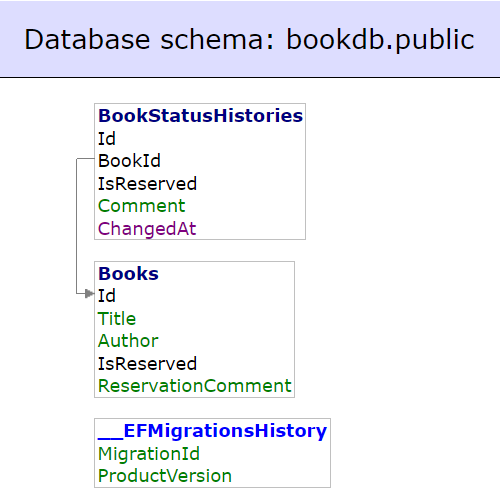
1. **Publish the Application:**
   * Open the project in Visual Studio.
   * Right-click the project and select Publish.
   * Choose Folder as the target and specify a location.
2. **Setup IIS:**
   * Open IIS Manager.
   * Create a new Application Pool.
   * Create a new Website and point it to the published folder.
   * Configure the Application Pool to use No Managed Code.
3. **Configure IIS Settings:**
   * Set the appropriate permissions for the folder.
   * Configure bindings (e.g., set up a hostname).
4. **Deploy the Application:**
   * Copy the published files to the IIS website folder.

### 

## **Appsettings Property Description**

* **Logging**: Configured logging levels for the application.
* **ConnectionStrings**: Contains the connection string for the PostgreSQL database.

## **Entity Relationship Diagram(ER diagram)**

****

## **Error Handling Middleware**

### **Overview**

Error handling middleware is a crucial component in an ASP.NET Core application. It intercepts exceptions thrown during the processing of HTTP requests and allows to handle them in a centralized manner. This ensures that unhandled exceptions are logged and users receive friendly error messages instead of seeing raw exception details.

### **Implementation**

In this application, the ErrorHandlingMiddleware class is responsible for handling exceptions. It catches any exceptions thrown during the request processing pipeline, logs the error details, and return response.

# **Unit test**

xUnit unit tests have been implemented for all API endpoints to ensure the reliability and correctness of the application. Unit testing helps catch bugs early, validates business logic, and ensures that API endpoints return the expected responses. This approach also improves code quality, simplifies refactoring, and ensures that the application functions as expected without introducing regressions during updates or changes