# Project Documentation

## Project Requirements

### Overview:

## The project is a small application designed with a specific functionality in mind. It's intended to be lightweight and efficient, focusing only on essential features to avoid overcomplicating the architecture.

### Frontend:

## Framework: React.js

## UI Library: Material-UI (MUI) for grid layouts and other UI components.

## Design: A grid layout where boxes can be colored either grey, red, or green.

### Backend:

## Framework: ASP.NET 6

## Architecture: Onion Architecture with the following layers:

## Domain Layer: Contains the core business logic and entities.

## Presentation Layer: Houses the API endpoints and request/response models.

## Infrastructure Layer: Deals with data access, third-party services, etc.

## API: RESTful API utilizing the Command and Query Responsibility Segregation (CQRS) pattern.

## Database: PostgreSQL.

## Direct interactions with the database instead of using Entity Framework to keep it lightweight. Potential tools: Dapper or ADO.NET.

## Validation: Considering the project's scale, simple validation mechanisms such as data annotations might be used. Depending on complexity, FluentValidation can be considered.

### Deployment and Environment:

## Docker: The application will be containerized using Docker, ensuring consistency across environments.

## Database: PostgreSQL will be used as the primary database, and it will be containerized using Docker.

### Key Decisions:

## Omission of certain tools like AutoMapper and MediatR to keep the architecture simple and direct.

## Given the limited scale and scope of the project, decisions have been made to prioritize simplicity and directness over scalability. However, the architecture is designed in a way that can allow for expansion if needed in the future.

## The project aims to be a straightforward and efficient solution without overcomplicating the architecture. The chosen tools and practices reflect the goal of simplicity while maintaining a clear separation of concerns and ensuring code maintainability.

## 

## 

## 

## Project Overview

# **The aim of the project is to create a small application with two tables. Given the project's scale, we are aiming for simplicity and maintainability without using overkill solutions. We'll be using the Onion Architecture to ensure that our core business logic remains unaffected by external changes and make it easy for other people to jump into this project for both senior developers and junior developers, separate each layer in a away so other people can work together without breaking the whole application when the developer in team in case a developer do something wrong in the application, i will not ruin the whole application. This is considering a big benefit of choosing this pattern here. Additionally, deploying the project using Docker with PostgreSQL as the database makes it easy, if in the feature would like to deploy the application on a cloud base server using azure or aws.**

## Architecture Overview

### 1. Onion Architecture

The Onion Architecture focuses on the separation of concerns by dividing the application into concentric layers:

* Domain Layer: Contains the core business logic and entities. It's the innermost layer and is unaware of the outside world.
* Presentation Layer: This is where the UI components reside. We'll be using React with MUI to create the grid UI.
* Infrastructure Layer: Manages external concerns like databases, file systems, and APIs. We'll be using FastEndpoint and ASP.NET 6 here.

### 2. CQRS Pattern (Command Query Responsibility Segregation)

To keep things simple and maintainable:

* Commands: Deal with modifying state.
* Queries: Deal with reading state.

This clear distinction makes it easier to maintain and understand the project's flow.

## Backend

* Framework: ASP.NET 6
* API: FastEndpoint for building the REST API.
* Database: PostgreSQL, containerized using Docker for ease of deployment and scalability.
* Dapper: Dapper is a micro ORM Object Relational Mapper, which is lightweight, fast, and allows for more direct SQL interactions than EF. It is quite popular for scenarios where performance is crucial. For this small project it will be overkill with entity framework (EF) because you get more than what you need with ef for this project so Dapper will be better here.
* Data Annotations: ASP.NET provides built-in validation using data annotations. These allow us to specify validation rules directly on our model properties using attributes like [Required], [StringLength], etc. They are simple and integrate well with ASP.NET. For validation, these requirements are basic, data annotations will be sufficient here.

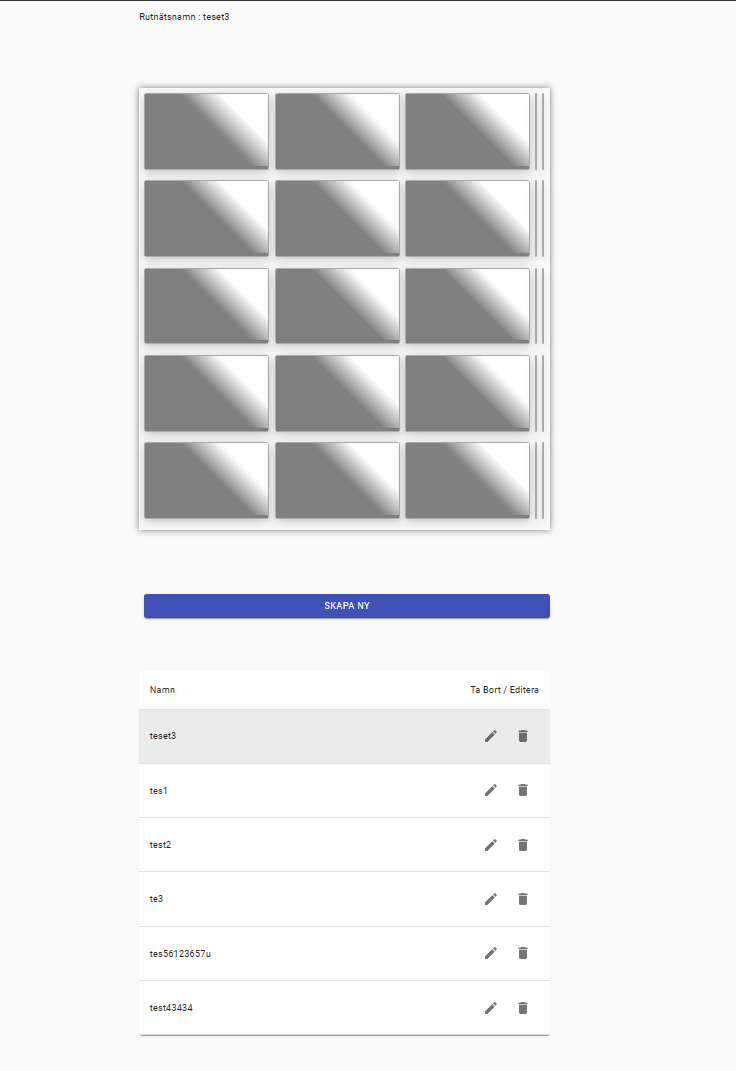
## Frontend

* Framework: React
* UI Components: MUI for grid layouts and other UI components.
* Context API at frontend for handling shared api calls small projects like this so skipping redux here is a good choice
* npm install @testing-library/react@^11.0.0

npm install @mui/material@^5.0.0

npm install @emotion/react @emotion/styled

**GUI for the application**



## Considerations

* Why skipping AutoMapper: Given the project's scale, introducing AutoMapper might be an overkill. Manual mapping will suffice for our limited entities.
* Not using MediatR: MediatR is a great tool for decoupling application layers and adding in process messaging capabilities. However, for a project of this size, direct handling within the FastEndpoint framework should suffice, removing the overhead that MediatR might introduce.
* Docker: Using Docker allows us to ensure that the application's environment remains consistent across various stages of development and deployment.
* Database: PostgreSQL offers robustness and scalability. As a relational database, it's a suitable choice for our table-based application.

## Conclusion

For this project, focusing on simplicity and maintainability, making architectural and technological choices that align with the project's scale. With clear separation of concerns and avoiding unnecessary complexities, we aim to ensure that the application remains easy to understand, develop, and maintain. The combination of Onion Architecture with the CQRS pattern provides a structured approach, while our choice of tools and technologies, such as React with MUI and ASP.NET 6 with FastEndpoint, offers a modern and efficient development experience. Our decision to omit tools like AutoMapper and MediatR stems from our commitment to keep the codebase lean for this specific project scope. By containerizing the application with Docker and choosing PostgreSQL, we are also ensuring a smooth deployment and scalability process. In conclusion, this documentation provides a roadmap that balances sophistication with simplicity, tailored to meet the project's requirements while ensuring future maintainability.

Project Estimation Plan for SPA Web Application

Business Requirement:

Develop a SPA web application allowing users to interact with a grid layout, change cell statuses, save the grid, view saved grids, and work offline.

Key Drivers:

1. User-friendly interaction with the grid to change cell status.

2. Offline functionality (PWA) and data synchronization when online.

3. Backend developed using .NET with a focus on Onion Architecture.

4. Frontend leveraging React.js and Material-UI.

5. Scalable and secure database storage with PostgreSQL.

Functional Requirements:

Epic 1: Initial Project Setup

Feature 1.1: Backend Environment Setup

- Task 1.1.1: Initialize an ASP.NET 6 project with Onion Architecture.

- Task 1.1.2: Set up PostgreSQL database.

- Task 1.1.3: Configure connection strings.

- Task 1.1.4: Define initial API routes FastEndpoint REST API.

Feature 1.2: Frontend Environment Setup

- Task 1.2.1: Setup a React project.

- Task 1.2.2: Integrate Material-UI for grid layouts and UI components.

- Task 1.2.3: Set up state management using Context API.

Epic 2: Core Functionality Development

\*Feature 2.1: Grid Manipulation UI

- Task 2.1.1: Design and implement grid layout using Material-UI.

- Task 2.1.2: Implement cell status cycling with visual feedback.

- Task 2.1.3: Create functionality to initiate a new grid.

- Task 2.1.4: Implement grid naming and saving mechanism to the server.

- Task 2.1.5: Create an component and implementation at frontend handling shared api calls (CRUD) with Context API

Feature 2.2: Backend CRUD Operations

- Task 2.2.1: Set up database models for grids and their statuses.

- Task 2.2.2: Implement CRUD operations using Dapper.

- Task 2.2.3: Establish API endpoints for saving, retrieving, and deleting grids.

- Task 2.2.4: Implement name validation using data annotations.

Feature 2.3: Frontend Interaction with Backend

- Task 2.3.1: Set up API calls to save and retrieve grids.

- Task 2.3.2: Implement grid listing, loading, and deletion functionality.

- Task 2.3.3: Add offline capabilities using Service Workers.

- Task 2.3.4: Implement data synchronization logic for when the app is back online.

Epic 3: Finalization and Optimization

Feature 3.1: Testing

- Task 3.1.1: Implement unit tests for backend logic.

- Task 3.1.2: Add tests for frontend components and user interactions.

- Task 3.1.3: Implement in backend and frontend error handling across the application

Feature 3.2: Performance and Security Enhancements

- Task 3.2.1: Optimize API responses (consider caching).

- Task 3.2.2: Implement security best practices (data validation, securing endpoints).

Feature 3.3: Deployment and Documentation

- Task 3.3.1: Document the codebase and API functionalities, emphasizing the Onion Architecture and CQRS pattern.

- Task 3.3.2: Write a comprehensive README, including setup instructions and architectural decisions.

- Task 3.3.3: Containerize the application using Docker for deployment, ensuring compatibility for potential future deployments on cloud platforms like Azure or AWS.