**CS673 Software Engineering**

**Team 4 - Project Portal**

**Software Design Document**

| Team Member | Role(s) | Signature | Date |
| --- | --- | --- | --- |
| Brenna Mahn | Team Lead | *BM* | 5/28/2024 |
| Natanim Eibrahim | Security Lead | *NE* | 5/28/2024 |
| Savien Love | Design & Implementation Lead | *SL* | 5/28/2024 |
| Pinwen Mu | Configuration Lead | *PM* | 5/28 |
| Fehmi Baltaci | QA Lead | *FB* | 5/28 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Revision history**

| **Version** | **Author** | **Date** | **Change** |
| --- | --- | --- | --- |
| **1.0** | **Brenna Mahn, Savien (Brooke) Love, Pinwen Mu, Natanim Eibrahim, Fehmi Baltaci** | **5/28/2024** | **First Draft** |
| **2.0** | **Brenna Mahn, Savien (Brooke) Love, Pinwen Mu, Natanim Eibrahim, Fehmi Baltaci** | **6/10/2024** | **Second Draft** |

[Introduction](#_87t9hln2vjz0)

[Software Architecture](#_3ipvmjgn6clp)

[Class Diagram](#_ky60nv8suxxm)

[UI Design (if applicable)](#_7ucksmkf6rzx)

[Security Design](#_x18fj36s1121)

[Business Logic and/or Key Algorithms](#_mtfbusfb0eq3)

[Design Patterns](#_9zvwkmc4luo5)

# Introduction

**Overview**

This document outlines the software design for the frontend UI of our project portal, built with React. Our team is responsible for creating an intuitive and user-friendly interface that integrates seamlessly with backend services provided by two other teams. This document details the architecture, design goals, class diagrams, UI design, database design, security considerations, key algorithms, and design patterns used in the development of our software system.

**Design Goals**

Our primary design goals are:

* User-Friendliness: Ensure the interface is intuitive and easy to navigate.
* Integration: Achieve integration with backend services to provide a cohesive user experience.
* Performance: Optimize the UI for fast loading times and responsiveness.
* Scalability: Design the system to accommodate future expansion and new features.
* Maintainability: Use clear, modular code that follows best practices.

# Software Architecture

Our software architecture centers around React as our primary framework, leveraging its component-based structure for better organization and scalability. We've identified key main pages such as the Dashboard, Landing, and Account Registration, each containing specific components tailored to their functionalities. For instance, the Dashboard uses components like the Search Bar, Cards, and Modal popup, facilitating smooth navigation and interaction with projects. The Search Bar allows users to quickly find relevant projects within the Dashboard, while the Cards provide a snapshot view of key project information. Additionally, the Modal popup enables users to view detailed project information or perform actions without leaving the Dashboard page. Within the Project page, components for managing individual projects, including Add/Edit functionality, project analytics, and task management, are seamlessly integrated. For example, users can add or edit project details, analyze project performance through analytics, and manage project tasks efficiently.

To maintain a clear codebase, we've adopted a folder structure within the /src directory, segregating components, pages, and assets into separate folders. This structured approach enhances code maintainability and readability. Shared components like the Footer and Navigation bar are strategically placed for consistent user experience across different pages, fostering familiarity and ease of navigation. We've also adopted a grouping method within the React folders to further streamline code organization, ensuring clarity and ease of development.

/src

├── components

│ ├── Header

│ │────Header.jsx

│ │────Assets

│ │—--- Header.css

│──── Images

├────────

│ ├── Footer.js

│ ├── NavigationBar.js

│ ├── SearchBar.js

│ └── ...

├── pages

│ ├── Dashboard

│ │────Dashboard.jsx

│ │────Assets

│ │—--- Dashboard.css

│──── Images

|---- utils

│ ├── LandingPage.js

│ ├── ProjectPage.js

│ └── AccountRegistrationPage.js

├── assets

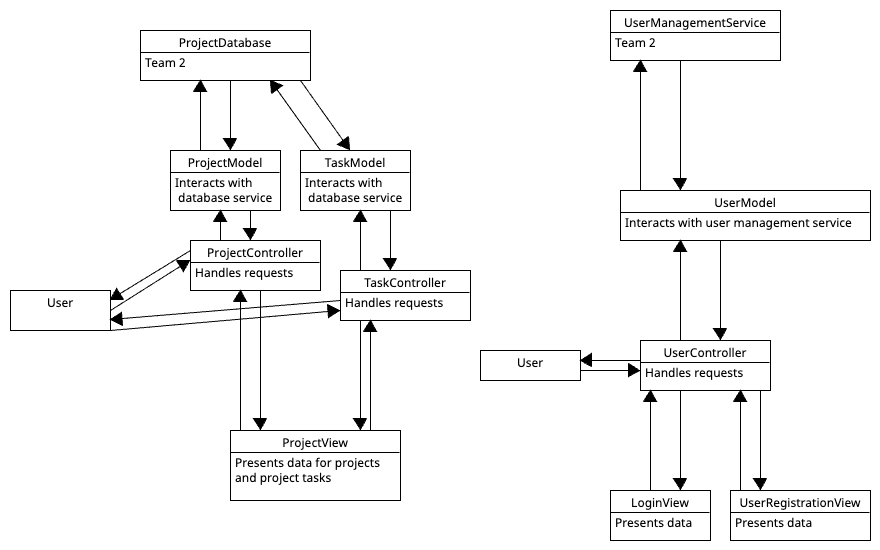
│ ├── images

│ └── styles

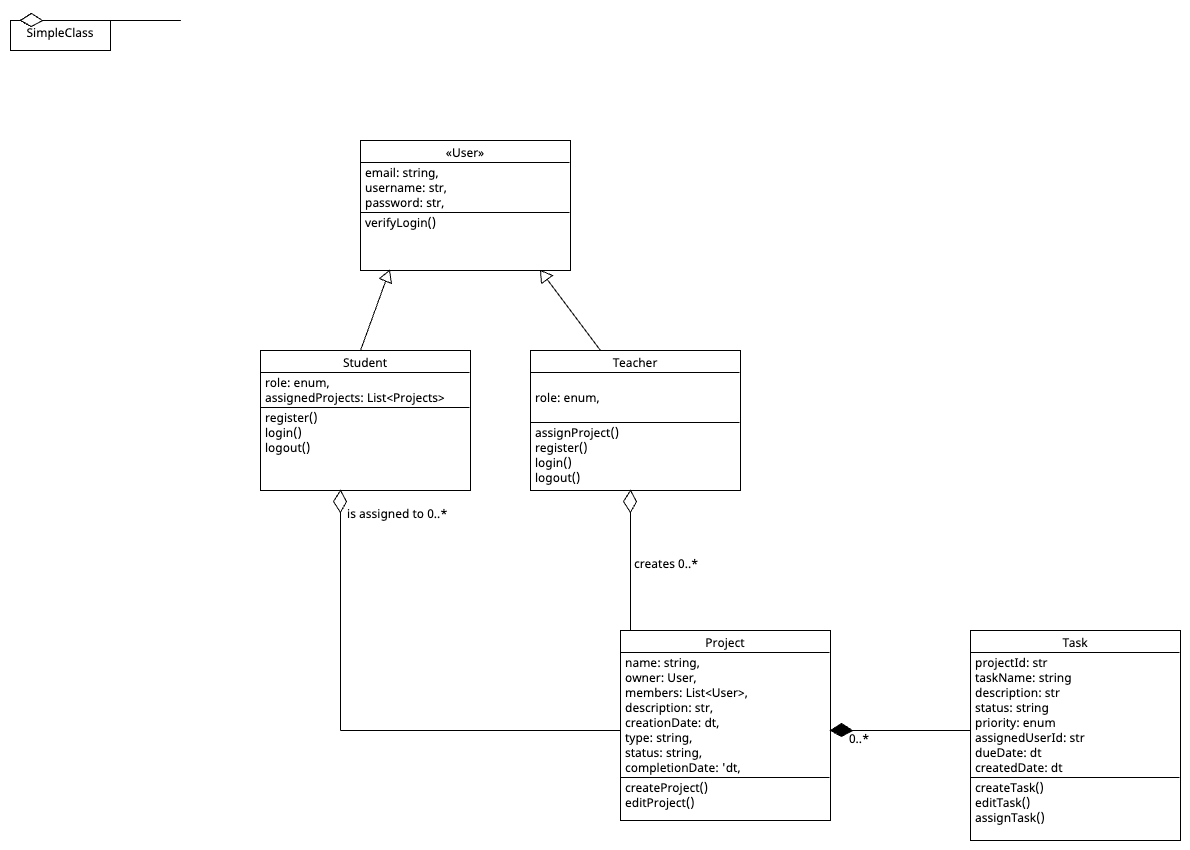
└── ...

For our architecture we plan to use a modified MVC pattern. React doesn't strictly adhere to the traditional MVC pattern, so we will adopt a similar approach where:

* Model: Connect with backend services to handle data management and business logic.
* View: React components are responsible for rendering the UI.
* Controller: React components also handle user input and interactions, coordinating between the view and model.



# Class Diagram



# UI Design

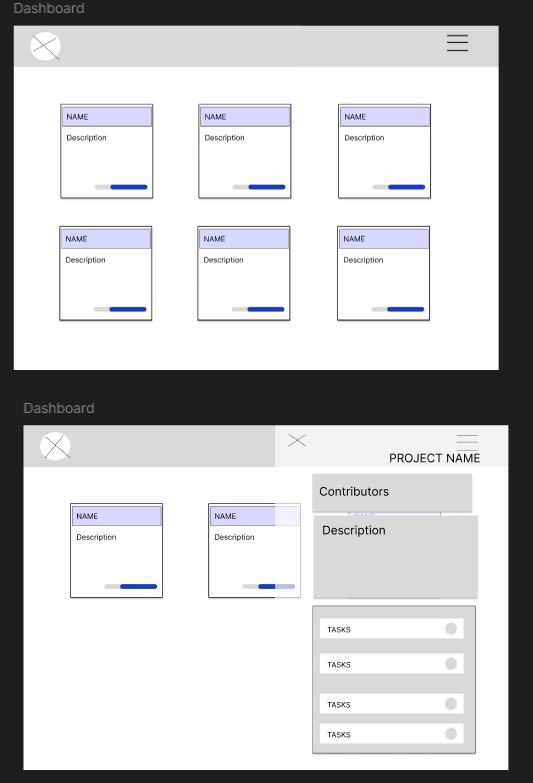
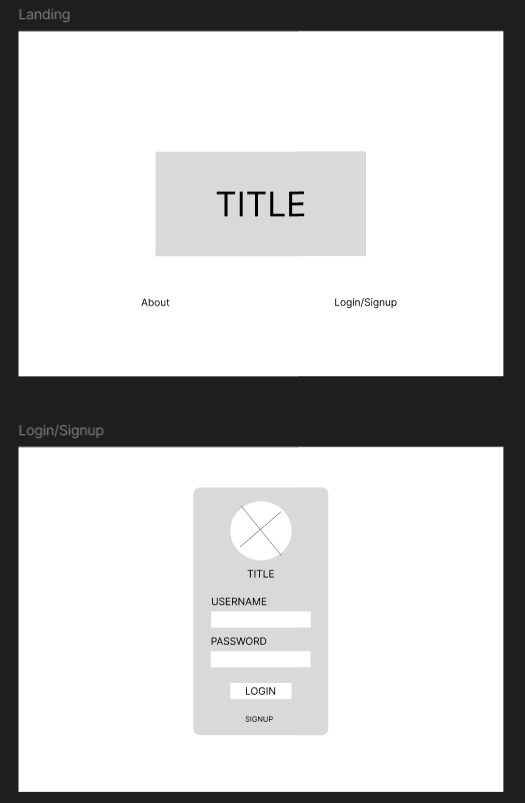
Our design concept achieves a blend of minimalism, user-friendliness, and aesthetic elegance, drawing inspiration from Apple's renowned design ethos. We aspire to create an interface that uses clean lines, subtle animations, and a calming color palette. By incorporating elements such as card popups and gentle animations, we aim to elevate the user experience with engaging visual cues while maintaining a sense of simplicity and ease of use.

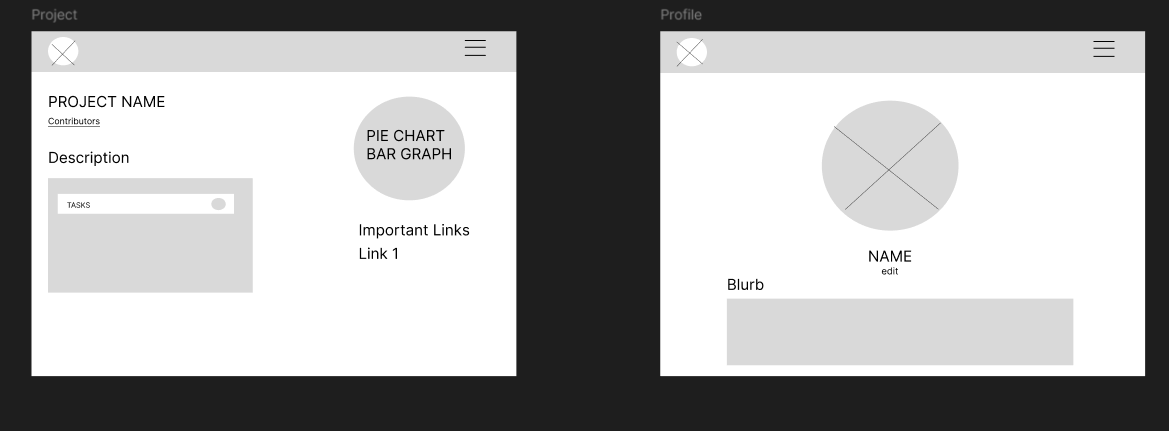
The layout, crafted on Figma, orchestrates a seamless journey for users, guiding them from the inviting Landing page to the streamlined Login process and onward to the intuitive Dashboard. Within this cohesive framework, users can effortlessly access their profiles, manage projects, and explore additional functionalities with clarity and efficiency. Our design prioritizes clarity, ensuring that users can navigate the interface with ease and find the information they need without unnecessary complexity.

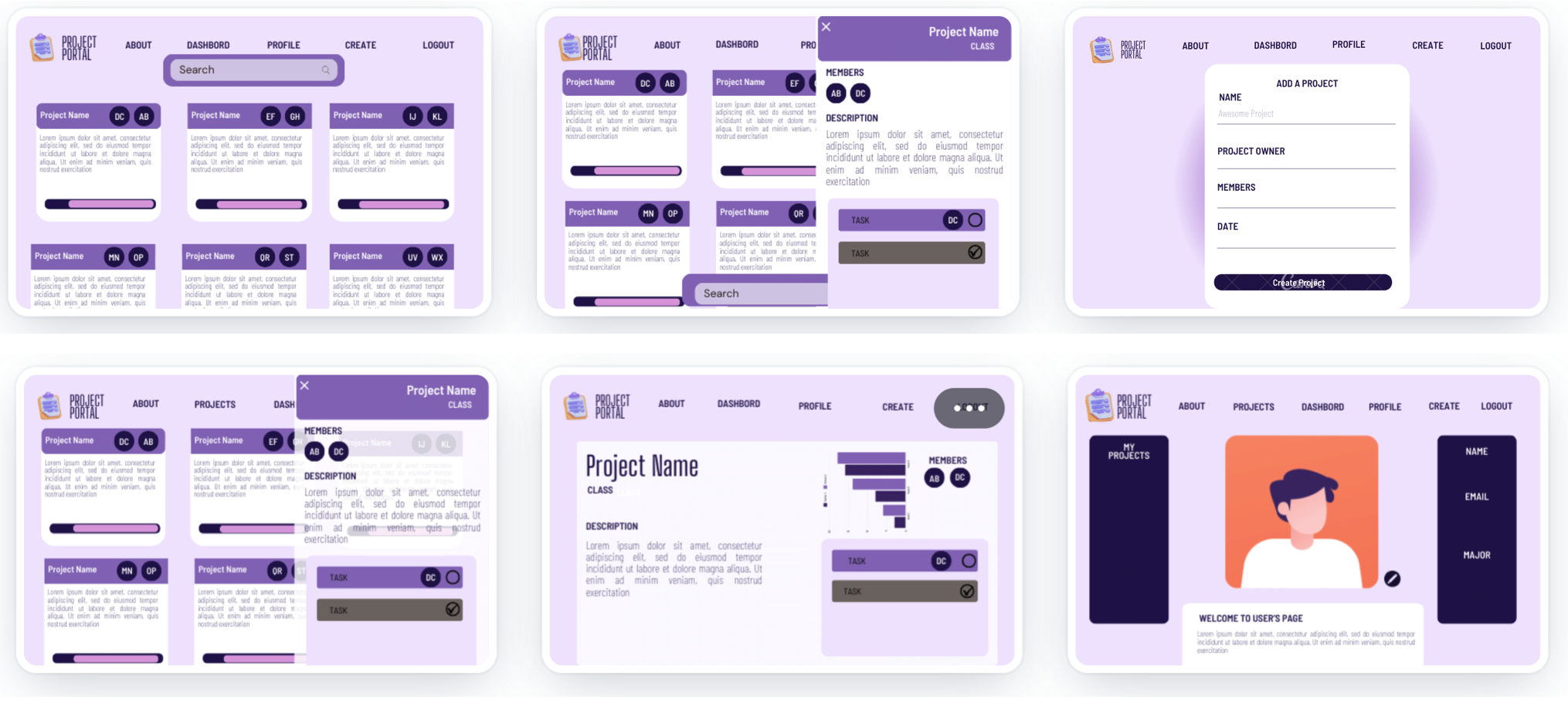
In the implementation phase, our focus remains on users from the moment they interact with the application. Through iterative design refinements and user testing, we seek to refine the user experience further, ensuring that every interaction is intuitive, delightful, and aligned with our overarching goal of creating a unified and serene digital environment.

**Wireframes:**

[Figma Design Link](https://www.figma.com/design/EYwwM9DDkhWEu2cDELePtI/Untitled?node-id=0:1&t=QG0BAaEahLoD4FTK-1)







# Security Design

The project portal platform will be implemented with the necessary security measures listed as below on the frontend side of the platform before sending requests to the backend API.

**Authentication and authorization:** Users of the platform will be required to use authentication forms to access secured routes within the application. The process includes:

1. User Registration and Login:

* Users will have access to dedicated registration and login pages, which include necessary input fields with validation and error messaging.
* Upon successful submission of valid credentials during registration or login, the frontend will send an API request to the backend with the provided credentials.

1. Token-Based Authentication:

* The backend will implement necessary security measures and, upon successful authentication, will generate authentication tokens.
* These tokens will be stored in HTTP-only cookies on the user's browser for enhanced security against XSS attacks.

1. Secure API Requests:

* To access secure routes on the backend, the frontend will include the authentication token in the headers of subsequent API requests.
* This token will be used to authenticate and authorize users, ensuring that only authenticated users can access protected resources.

**Input Validation and Sanitization:** We prioritize security by ensuring that all user inputs are properly validated and sanitized. To achieve this, we utilize the Yup library, a JavaScript schema builder for value parsing and validation and integration with Formik , also a react library.

Validation with Yup: Yup is a powerful library that allows us to define schemas to validate the shape and content of objects, strings, numbers, arrays, and other data types. By using Yup, we ensure that user inputs meet the required criteria before being processed or stored.

Input Sanitization: Sanitization involves cleaning the data to remove any malicious code or unwanted characters, further protecting against attacks like XSS. We will sanitize user inputs to prevent injection attacks and user addition libraries to implement user input sanitization such as DOMPurify to sanitize HTML.

**Secure Coding Practice:** Implementing a secure coding practice and taking security measures to prevent security attacks. During the implementation we will use secure coding practice to prevent potential threats and injection attacks such as:

* Avoiding Inline JavaScript
* Using Content Security Policies (CSP)
* Secure use of Third-party Libraries
* Cross-Site Request Forgery (CSRF) protection
* Token Expiration
* Secure Communication Using HTTPS

# Business Logic

Our business logic primarily involves:

* Form Validation: Checking that all user input is valid before sending data to the backend.
* State Management: Using React's built in state and context APIs to manage application state.
* Notification Handling: Logic to display and manage user notifications.

# Design Patterns

We plan to use React’s component-based architecture for modular and reusable UI components. Therefore, we plan to use some component design patterns as follows:

* Higher-Order Components (HOCs)
  + Use Case: Authentication
  + Explanation: Create an HOC that checks if a user is authenticated before allowing access to a specific route or component. This HOC can be used to wrap any component that requires authentication, ensuring that the logic is not duplicated across multiple components.In authentication, an HOC can be leveraged to enforce access control by wrapping components that require authentication. This encapsulation ensures that authentication logic remains centralized and can be easily applied across various parts of the application, promoting code reusability and maintainability. By using HOCs for authentication, we can streamline the implementation of security measures and reduce the risk of inconsistencies or errors in authentication logic.
* Render Props
  + Use Case: Data Fetching
  + Explanation: Use a component with a render prop to fetch project data and pass a function to render the data. This allows different components to use the same data-fetching logic but render the data differently based on their specific requirements. When dealing with data fetching, a component with a render prop can encapsulate the logic for fetching data from an external source, such as an API. By passing a function as a prop to this component, other components can consume the fetched data and render it according to their specific requirements. This approach promotes code modularization and reusability, as the data-fetching logic is decoupled from the presentation logic. Additionally, Render Props allow for greater flexibility in how data is consumed and rendered, enabling developers to adapt to changing requirements or UI designs more easily.
* Container and Presentational Components
  + Use Case: Project Management
  + Explanation: Use Container Components to handle the logic of fetching and updating project data, and Presentational Components to display the project details and UI elements. This separation helps in keeping the code organized and easier to manage. Container Components are responsible for managing state and data fetching, often containing business logic related to data manipulation and communication with external APIs. On the other hand, Presentational Components focus solely on rendering UI elements based on the props they receive. By separating concerns in this manner, developers can achieve a clearer separation of concerns and improve the maintainability of their codebase. This pattern also facilitates code reuse, as Presentational Components can be easily reused in different parts of the application without being tightly coupled to specific data or logic.
* Controlled and Uncontrolled Components
  + Use Case: Forms
  + Explanation: Use Controlled Components for forms that require validation and state management, such as the login or registration forms. For simpler inputs, like a search bar, use Uncontrolled Components to reduce the complexity of state management. Controlled Components maintain their state within React's component state, with their values controlled by React. This makes them suitable for forms that require validation or need to respond to user input dynamically. On the other hand, Uncontrolled Components rely on DOM references to manage their state, making them simpler to use for basic form inputs where complex state management is unnecessary. By choosing the appropriate approach based on the complexity of the form and its requirements, developers can strike a balance between simplicity and functionality in their form implementations

These patterns will guide our development process, ensuring a well-structured and maintainable codebase.

# 

# 