Architecture Pattern Selection and Justification for ShebaBondhu

November 4, 2025

1 Architecture Pattern Selection and Justification

1.1 Introduction

ShebaBondhu is designed as a **web application** that connects homeowners with service providers. Although the system is not a native mobile application, the frontend is developed using **React with Next.js**, which encourages a component-driven architecture and client-side interactivity. To ensure scalability, maintainability, and clean separation of concerns, an appropriate architectural pattern must be selected.

1.2 Candidate Architecture Patterns

Based on the Software Requirements Specification (SRS), multiple architectural patterns were evaluated:

- Model-View-Controller (MVC)
- Model-View-Template (MVT)
- Model-View-ViewModel (MVVM)
- Model-View-Intent (MVI)

Each pattern supports UI, business logic, and data handling, but with different structural workflows.

1.3 Why MVVM Fits ShebaBondhu Best

1.3.1 SRS-Based Requirement Mapping

The ShebaBondhu SRS contains features that require:

- Real-time updates (service acceptance, tracking status)
- Different dashboards for roles (homeowner, provider)
- Interactive UI with heavy client-side state handling

- Future possibility of converting into a mobile application
- Clean separation of UI logic from data state

These requirements align strongly with MVVM characteristics:

- View handles UI rendering only
- ViewModel manages UI state, validation, observers, API sync
- Model represents database + backend logic (Node.js + MongoDB)

Thus, MVVM provides better abstraction, easier testing, and scalable component reusability.

1.4 Why MVC Was Not Selected

MVC is mainly suitable for:

- Traditional server-rendered websites
- Light client-side interactions

However, ShebaBondhu includes:

- Heavy client-side state
- Interactive event-driven UI

Thus, controllers would become complex and tightly coupled with UI changes, making future scaling difficult.

1.5 Why MVT Was Not Selected

MVT is used primarily in Django-based server-rendered apps. ShebaBondhu:

- Uses React + Next.js, not Django
- Requires dynamic component-based rendering, not template-driven output

Therefore, using MVT would not align with the chosen technology stack.

1.6 Why MVI Was Not Selected

MVI is powerful for real-time synchronized applications but:

- Has higher development complexity
- Requires state immutability and a unidirectional data flow
- Often used in Redux-heavy environments or Android Jetpack Compose

Our system does not currently require strict global state enforcement.

Pattern	Best For	UI State Handling	Complexity	Examples / Frameworks
MVC	Traditional server-rendered web apps	Manual and coupled with controller	Low	Laravel, Spring MVC
MVT	Template-based backend web apps	Framework-managed (SSR)	Low	Django
MVVM	Highly interactive UI with local state	Data binding + hooks	Medium	React, Angular
MVI	Real-time dynamic apps requiring full predictability	Strict unidirectional flow	High	Redux, Jetpack Compose

Table 1: High-level Comparison of Architecture Patterns

1.7 Pattern Comparison Summary

1.8 Final Decision

After evaluating the pattern suitability based on SRS requirements and chosen technologies, **MVVM is selected as the architecture pattern for ShebaBondhu** because:

- It aligns with component-based development in React + Next.js
- It allows cleaner UI logic separation for future scalability
- It supports interactive dashboards and real-time user experience
- It enables potential reuse of ViewModel for a future mobile version

Thus, MVVM ensures ShebaBondhu remains modular, maintainable, and adaptable for upcoming enhancements.

2 MVVM Implementation Example in Next.js + React

2.1 Problem Example

To demonstrate MVVM principles in practice, we present an example from ShebaBondhu where we need to display a **list of booked services** for a homeowner. The user can **mark a service as completed**, and the UI updates automatically.

This scenario is perfect for MVVM because:

- **Model:** Represents the service data (from database/API)
- **ViewModel:** Holds state, handles logic (fetching, updating, formatting)
- **View:** React component renders the UI and observes the ViewModel

2.2 Folder Structure (MVVM)

The following folder structure demonstrates how MVVM components are organized:

```
/components
    ServiceList.js      <-- View
/viewmodels
    ServiceViewModel.js <-- ViewModel
/models
    ServiceModel.js     <-- Model (API functions)
/pages
    dashboard.js        <-- Next.js page importing ServiceList
```

Listing 1: MVVM Folder Structure

2.3 Model Layer Implementation

The Model handles API calls and raw data operations, with no knowledge of the UI.

```
// ServiceModel.js
// Handles API calls / raw data

export async function fetchServices(userId) {
  const res = await fetch(`/api/services?userId=${userId}`);
  const data = await res.json();
  return data;
}

export async function markServiceCompleted(serviceId) {
  const res = await fetch(`/api/services/${serviceId}/complete`, {
    method: 'POST',
  });
  return res.json();
}
```

Listing 2: ServiceModel.js - Model Layer

Key Points:

- Represents the data layer (Model)
- Handles fetching and updating data
- No UI dependencies

2.4 ViewModel Layer Implementation

The ViewModel manages state and business logic, acting as a bridge between Model and View.

```
// ServiceViewModel.js
import { useState, useEffect } from 'react';
import * as ServiceModel from '../models/ServiceModel';

export function useServiceViewModel(userId) {
  const [services, setServices] = useState([]);
  const [loading, setLoading] = useState(true);

```

```
  // Fetch services from the Model
  const loadServices = async () => {
    setLoading(true);
    const data = await ServiceModel.fetchServices(userId);
    setServices(data);
    setLoading(false);
  };

  // Mark a service as completed
  const completeService = async (serviceId) => {
    await ServiceModel.markServiceCompleted(serviceId);
    // Update the UI automatically
    setServices((prev) =>
      prev.map((s) =>
        s.id === serviceId ? { ...s, status: 'Completed' } : s
      )
    );
  };

  // Load services initially
  useEffect(() => {
    loadServices();
  }, [userId]);

  return { services, loading, completeService };
}
```

Listing 3: ServiceViewModel.js - ViewModel Layer

Key Points:

- Manages state (`services`, `loading`) and logic (`completeService`)
- Observes the Model but does not know about UI components
- Can be reused in multiple views
- Implements reactive state updates

2.5 View Layer Implementation

The View is a pure React component that renders UI based on ViewModel state.

```
// ServiceList.js
import React from 'react';

export default function ServiceList({ viewModel }) {
  const { services, loading, completeService } = viewModel;

  if (loading) return <p>Loading services...</p>;

  return (
    <ul>
      {services.map((service) => (
        <li key={service.id}>
          <span>
            {service.name} - {service.status}
          </span>
```

```
          {service.status !== 'Completed' && (
            <button onClick={() => completeService(service.id)}>
              Mark Completed
            </button>
          )}
        </li>
      ))}
    </ul>
  );
}
```

Listing 4: ServiceList.js - View Layer

Key Points:

- View only renders UI elements
- Observes the ViewModel for data and actions
- Does not handle business logic or API calls
- Purely presentational component

2.6 Next.js Page Integration

The Next.js page connects the ViewModel and View together.

```
import React from 'react';
import ServiceList from '../components/ServiceList';
import { useServiceViewModel } from '../viewmodels/ServiceViewModel';

export default function Dashboard() {
  const userId = '123'; // Example logged-in user
  const serviceVM = useServiceViewModel(userId);

  return (
    <div>
      <h1>My Services</h1>
      <ServiceList viewModel={serviceVM} />
    </div>
  );
}
```

Listing 5: dashboard.js - Next.js Page

Key Points:

- Page imports ViewModel and passes it to the View
- UI automatically reacts to state changes managed in ViewModel
- Clean separation of concerns maintained

2.7 MVVM Layer Responsibilities

2.8 Benefits of This MVVM Implementation

The implementation demonstrates several key advantages:

Layer	Responsibility
Model	Fetch and update raw data (API calls)
ViewModel	Holds state, handles logic, updates View
View	Renders UI based on ViewModel, observes state
Page	Connects View + ViewModel

Table 2: MVVM Layer Responsibilities in ShebaBondhu

1. **Clean Separation of Concerns:** Each layer has a distinct responsibility
2. **Reusable Logic:** ViewModel can be used across multiple pages/components
3. **Automatic UI Updates:** UI reactively updates when ViewModel state changes
4. **Testability:** Business logic (ViewModel) can be tested without rendering UI
5. **Maintainability:** Changes to one layer don't affect others
6. **Scalability:** Easy to extend with new features and components

3 Conclusion

The MVVM architecture pattern provides an optimal solution for ShebaBondhu's requirements. Through clear separation of Model, ViewModel, and View layers, the system achieves:

- Scalable and maintainable codebase
- Reactive and responsive user interface
- Testable business logic independent of UI
- Flexibility for future enhancements
- Alignment with modern React and Next.js best practices

This architectural foundation ensures ShebaBondhu can evolve to meet growing user demands while maintaining code quality and development efficiency.