Actuarial Model Web Application Design

Technical Report

Prepared by:
Praneel Misthry

Date:
25th August 2024

Company:
Solidus Software Solutions

Project:
Actuarial Model Web Application

Objective:
To design and implement a scalable, maintainable, and user-friendly web application that interfaces with an actuarial model microservice written in Python.

The application is built using ReactJS for the frontend and ASP.NET Core for the backend, with Azure Storage queues managing asynchronous tasks processed by a Python microservice.

Contact Information:
Email: pmisthry@gmail.com
Phone: 082 683 6536

Table of Contents

1. Introduction

This document outlines the design of a web application that wraps an actuarial model microservice written in Python. The application leverages ReactJS for the frontend and ASP.NET Core for backend operations, with Azure Storage queues managing asynchronous tasks and SQL Server as the database.

The focus is on designing a UI layer that is both dynamic and scalable, while also ensuring a robust backend that can efficiently orchestrate the pipelines. The design is supplemented with relevant code snippets and architectural diagrams to demonstrate the key concepts.

---

2. High-Level Architecture

2.1 Overview

The architecture of the application is designed to maintain a clear separation of concerns, ensuring scalability and maintainability. It is divided into three main layers:

1. **Frontend (ReactJS):** Manages the user interface, handles user interactions, and dynamically renders content based on backend-provided schemas.
2. **Backend (ASP.NET Core):** Provides API endpoints for the frontend, manages data storage with SQL Server, and orchestrates tasks using Azure Storage queues.
3. **Pipeline Orchestration:** Uses Azure Storage queues to manage asynchronous tasks processed by a Python microservice.

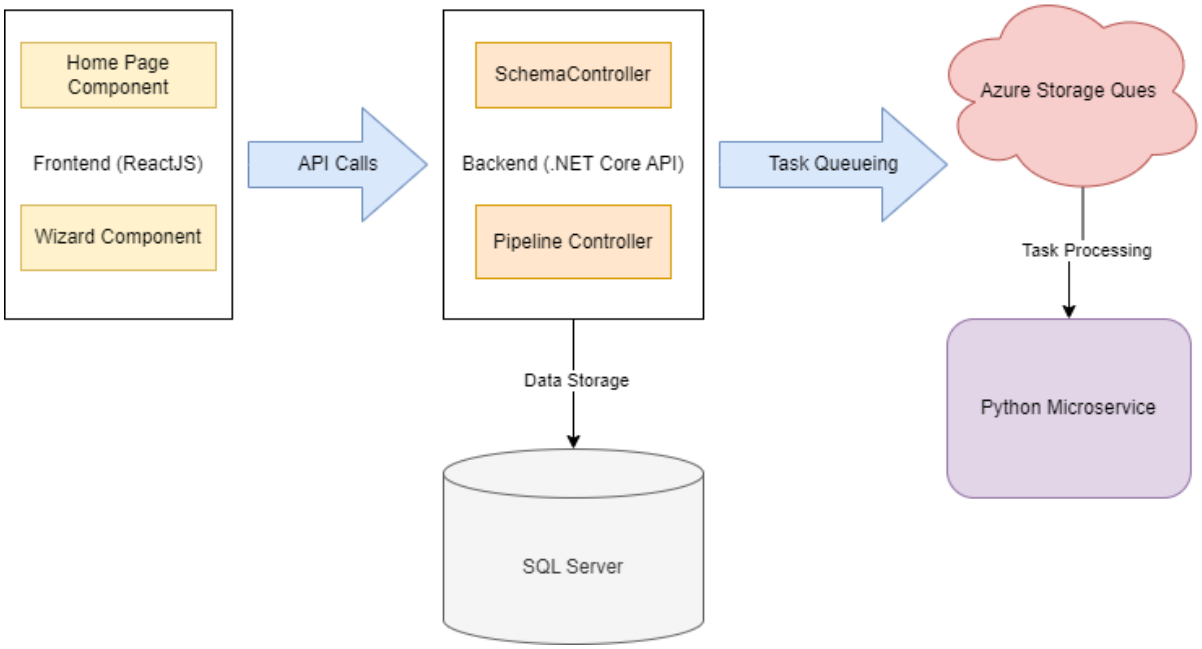

Figure 2.1a: High Level Architecture Diagram

2.2 Description

This diagram shown in Figure 2.1a illustrates the overall architecture of the application. It shows how the **Frontend (ReactJS)** communicates with the **Backend (.NET Core API)** via API calls, how data is stored in **SQL Server**, and how tasks are queued and processed using **Azure Storage Queues** and a **Python Microservice**. The arrows represent the flow of data and control between these components.

2.3 Design Principles

- **Client-Server Architecture:** Ensures that the frontend and backend are decoupled, allowing for independent scaling and development.
- **Microservices Architecture:** The backend interacts with a Python microservice for actuarial calculations, enabling scalability and modularity.
- **Design Patterns:** Utilizes the Repository and Unit of Work patterns in the backend for data access, and the Publisher-Subscriber and Chain of Responsibility patterns for pipeline orchestration.

2.4 Evaluation

- **Design Quality:** The architecture is clearly defined, modular, and ensures maintainability and scalability.
- **Architecture:** Proper separation of concerns with the use of design patterns that promote clean, scalable code.
- **Functionality:** The architecture supports the necessary interactions between the frontend, backend, and pipeline components.

---

3. Backend Design (.NET Core)

3.1 Overview

The backend is responsible for managing data flow, processing tasks, and interacting with the frontend via RESTful APIs. It also handles data persistence using SQL Server and Entity Framework Core, with design patterns ensuring clean code and maintainability.

3.2 Design Patterns

3.2.1 Repository Pattern

This pattern abstracts the data access layer, providing a clean interface for querying and persisting data. It decouples the business logic from data storage concerns. The csharp code snippet below illustrates this pattern.

```csharp
// csharp
public interface ISchemaRepository
{
    Task<Schema> GetUiSchemaAsync();
    Task<Schema> GetDataSchemaAsync();
}

public class SchemaRepository : ISchemaRepository
{
    private readonly ApplicationDbContext _context;
    public SchemaRepository(ApplicationDbContext context)
    {
        _context = context;
    }

    public async Task<Schema> GetUiSchemaAsync()
    {
        return await _context.Schemas.FirstOrDefaultAsync(s => s.Type == "UI");
    }

    public async Task<Schema> GetDataSchemaAsync()
    {
        return await _context.Schemas.FirstOrDefaultAsync(s => s.Type == "Data");
    }
}
```

3.2.2 Unit of Work Pattern

This pattern is used to manage database transactions. It ensures that all database operations within a business transaction are committed together, maintaining data integrity.

The csharp code snippet below illustrates this pattern.

```csharp
// csharp
public interface IUnitOfWork : IDisposable
{
    ISchemaRepository Schemas { get; }
    Task<int> CompleteAsync();
}

public class UnitOfWork : IUnitOfWork
{
    private readonly ApplicationDbContext _context;
    public ISchemaRepository Schemas { get; private set; }

    public UnitOfWork(ApplicationDbContext context)
    {
        _context = context;
        Schemas = new SchemaRepository(_context);
    }

    public async Task<int> CompleteAsync()
```

```
    {
        return await _context.SaveChangesAsync();
    }

    public void Dispose()
    {
        _context.Dispose();
    }
}
```

3.2.3 CQRS (Command Query Responsibility Segregation)

This pattern is used to separate the read (query) and write (command) operations, improving performance and scalability.

The csharp code snippet below separates the command (write) and query (read) operations for managing user data in an application.

```
// csharp
// Query Model: Separate classes for reading data
public class UserQueryModel
{
    public string UserId { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
}

// Command Model: Separate classes for writing data
public class CreateUserCommand
{
    public string Name { get; set; }
    public string Email { get; set; }
}

// Query Handler: Handles read operations
public interface IUserQueryHandler
{
    Task<UserQueryModel> GetUserByIdAsync(string userId);
}

// csharp
public class UserQueryHandler : IUserQueryHandler
{
    private readonly ApplicationDbContext _context;

    public UserQueryHandler(ApplicationDbContext context)
    {
        _context = context;
    }

    public async Task<UserQueryModel> GetUserByIdAsync(string userId)
    {
        return await _context.Users
            .Where(u => u.UserId == userId)
            .Select(u => new UserQueryModel
            {
                UserId = u.UserId,
                Name = u.Name,
                Email = u.Email
            })
            .FirstOrDefaultAsync();
    }
```

```
}
// Command Handler: Handles write operations
public interface IUserCommandHandler
{
    Task<string> HandleCreateUserAsync(CreateUserCommand command);
}

public class UserCommandHandler : IUserCommandHandler
{
    private readonly ApplicationDbContext _context;

    public UserCommandHandler(ApplicationDbContext context)
    {
        _context = context;
    }

    public async Task<string> HandleCreateUserAsync(CreateUserCommand command)
    {
        var user = new User
        {
            UserId = Guid.NewGuid().ToString(),
            Name = command.Name,
            Email = command.Email
        };

        _context.Users.Add(user);
        await _context.SaveChangesAsync();

        return user.UserId;
    }
}

// Controller: Demonstrates the use of CQRS pattern
[ApiController]
[Route("api/users")]
public class UserController : ControllerBase
{
    private readonly IUserQueryHandler _userQueryHandler;
    private readonly IUserCommandHandler _userCommandHandler;

    public UserController(IUserQueryHandler userQueryHandler, IUserCommandHandler
userCommandHandler)
    {
        _userQueryHandler = userQueryHandler;
        _userCommandHandler = userCommandHandler;
    }

    [HttpGet("{userId}")]
    public async Task<IActionResult> GetUserById(string userId)
    {
        var user = await _userQueryHandler.GetUserByIdAsync(userId);
        if (user == null)
        {
            return NotFound();
        }
        return Ok(user);
    }

    [HttpPost]
    public async Task<IActionResult> CreateUser([FromBody] CreateUserCommand command)
    {
        var userId = await _userCommandHandler.HandleCreateUserAsync(command);
        return CreatedAtAction(nameof(GetUserById), new { userId }, new { userId });
    }
}
```

3.3 Component Interaction

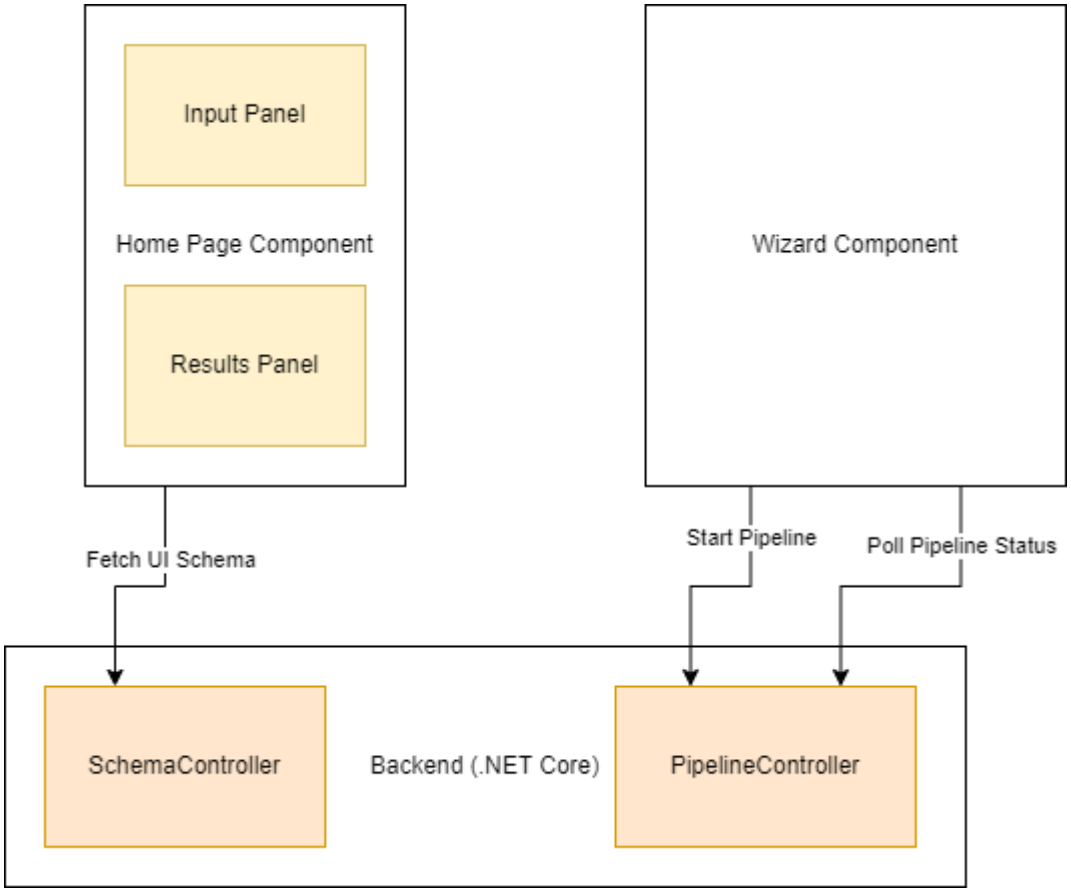

Figure 3.3a: Component Interaction Diagram

3.3.1 Description

The diagram in Figure 3.3a illustrates the interactions between the Home Page Component and Wizard Component within the frontend and the SchemaController and PipelineController within the backend. It highlights how the frontend components interact with the backend to fetch the UI schema and manage pipeline tasks.

3.3.2 Necessary Endpoints

1. **Return the UI Schema and Data Schema**

- **Endpoint:** `GET /api/schema/ui and GET /api/schema/data`
- **Purpose:** Provides the frontend with the necessary schemas to render the form dynamically and validate data.
- **Implementation:**

```
// csharp
```

```
[ApiController]
[Route("api/schema")]
public class SchemaController : ControllerBase
{
    private readonly IUnitOfWork _unitOfWork;

    public SchemaController(IUnitOfWork unitOfWork)
    {
        _unitOfWork = unitOfWork;
    }

    [HttpGet("ui")]
    public async Task<IActionResult> GetUiSchema()
    {
        var schema = await _unitOfWork.Schemas.GetUiSchemaAsync();
        return Ok(schema);
    }

    [HttpGet("data")]
    public async Task<IActionResult> GetDataSchema()
    {
        var schema = await _unitOfWork.Schemas.GetDataSchemaAsync();
        return Ok(schema);
    }
}
```

2. **Manage the UI State**
    - **Endpoint:** `POST /api/ui/state`
    - **Purpose:** Stores or updates the UI state in the database, ensuring the user's progress is saved.
    - **Implementation:**

```
// csharp

[ApiController]
[Route("api/ui")]
public class UiStateController : ControllerBase
{
    private readonly IUnitOfWork _unitOfWork;

    public UiStateController(IUnitOfWork unitOfWork)
    {
        _unitOfWork = unitOfWork;
    }

    [HttpPost("state")]
    public async Task<IActionResult> SaveUiState([FromBody] UiStateModel uiState)
    {
        _unitOfWork.UiStates.Add(uiState); // Assume a UiStateRepository is part of
UnitOfWork
        await _unitOfWork.CompleteAsync();
        return Ok();
    }
}
```

3. Start a Pipeline with User Data

- **Endpoint:** `POST /api/pipeline/start`
- **Purpose:** Initiates a processing pipeline with user-submitted data, which is enqueued for processing by the Python microservice.
- **Implementation:**

```csharp
// csharp

[ApiController]
[Route("api/pipeline")]
public class PipelineController : ControllerBase
{
    private readonly IPipelineService _pipelineService;

    public PipelineController(IPipelineService pipelineService)
    {
        _pipelineService = pipelineService;
    }

    [HttpPost("start")]
    public async Task<IActionResult> StartPipeline([FromBody] UserDataModel userData)
    {
        var taskId = await _pipelineService.EnqueueTaskAsync(userData);
        return Ok(new { TaskId = taskId, Status = "Pipeline started successfully" });
    }
}
```

4. Return the Status of the Pipeline

- **Endpoint:** `GET /api/pipeline/status/{taskId}`
- **Purpose:** Provides the status of a processing pipeline to the frontend.
- **Implementation:**

```csharp
// csharp

[ApiController]
[Route("api/pipeline")]
public class PipelineController : ControllerBase
{
    private readonly IPipelineService _pipelineService;

    public PipelineController(IPipelineService pipelineService)
    {
        _pipelineService = pipelineService;
    }

    [HttpGet("status/{taskId}")]
    public async Task<IActionResult> GetPipelineStatus(string taskId)
    {
        var status = await _pipelineService.GetTaskStatusAsync(taskId);
        return Ok(status);
    }
}
```

3.4 Evaluation

- **Design Quality:** The backend design ensures a clean separation of concerns, scalability, and maintainability using appropriate design patterns.

- **Functionality:** The design meets all functional requirements, with robust endpoints for handling UI state management, schema retrieval, and pipeline orchestration.

- **Scalability:** The backend is designed to handle many requests, with asynchronous processing for pipeline tasks.

---

4. Frontend Component Design (ReactJS)

The frontend is designed using ReactJS, focusing on creating a dynamic and responsive user interface. The UI components are designed to handle user input, communicate with the backend, and display results in an intuitive manner.

4.1 Component Interaction

Refer to the Component Interaction diagram shown previously in Figure 3.3a as a reference.

This diagram shows how the **Home Page Component** and **Wizard Component** within the frontend interact with the backend's **SchemaController** and **PipelineController**. The Home Page Component fetches the UI schema from the backend, while the Wizard Component manages the pipeline initiation and status polling.

4.2 Components

1. **Home Page:**

- **Design:** The Home Page component displays input forms on the left panel and results on the right panel. It dynamically generates input fields based on the UI schema provided by the backend.

- **Interaction with Backend:** The component fetches the UI schema from the backend upon mounting and renders the input fields accordingly.

```
import React, { useEffect, useState } from 'react';

const HomePage = () => {
    const [schema, setSchema] = useState(null);

    useEffect(() => {
        // Fetch UI schema from backend
        fetch('/api/schema/ui')
            .then(response => response.json())
            .then(data => setSchema(data));
```

```
    }, []);

    return (
        <div className="home-page">
            <div className="input-panel">
                {schema && schema.fields.map(field => (
                    <input key={field.name} type={field.type}
placeholder={field.placeholder} />
                ))}
            </div>
            <div className="results-panel">
                {/* Placeholder for results */}
            </div>
        </div>
    );
};

export default HomePage;
```

2. Wizard Component:

- **Design:** The Wizard component guides users through multiple steps, including data input, pipeline initiation, and result display. It ensures that users can navigate through different stages of the process smoothly.
- **Interaction with Backend:** The Wizard component sends user data to the backend to start the pipeline and regularly checks for the status of the pipeline to update the UI.

```
import React, { useState } from 'react';

const Wizard = ({ steps }) => {
    const [currentStep, setCurrentStep] = useState(0);

    const nextStep = () => setCurrentStep(prev => prev + 1);
    const prevStep = () => setCurrentStep(prev => prev - 1);

    return (
        <div className="wizard">
            {steps[currentStep]}
            <div className="navigation">
                <button onClick={prevStep} disabled={currentStep === 0}>Previous</button>
                <button onClick={nextStep} disabled={currentStep === steps.length -
1}>Next</button>
            </div>
        </div>
    );
};

export default Wizard;
```

4.3 Evaluation

- **Design Quality:** The components are designed to be reusable and maintainable, with clear separation of responsibilities.
- **Functionality:** The design meets the functional requirements, ensuring that the UI is dynamic, responsive, and interacts effectively with the backend.

- **UI/UX:** The interface is user-friendly, guiding users through the necessary steps in a clear and intuitive manner.

- **Scalability:** The frontend is built to scale, with components that can handle increasing user load and interaction complexity.

---

5. State Management Plan

5.2 Overview

The state management strategy in the application ensures that the UI remains responsive and consistent, with the state synchronized between the frontend and backend. The plan covers how the UI schema and data are fetched, stored, and updated, as well as how the UI reflects the status of the pipeline.

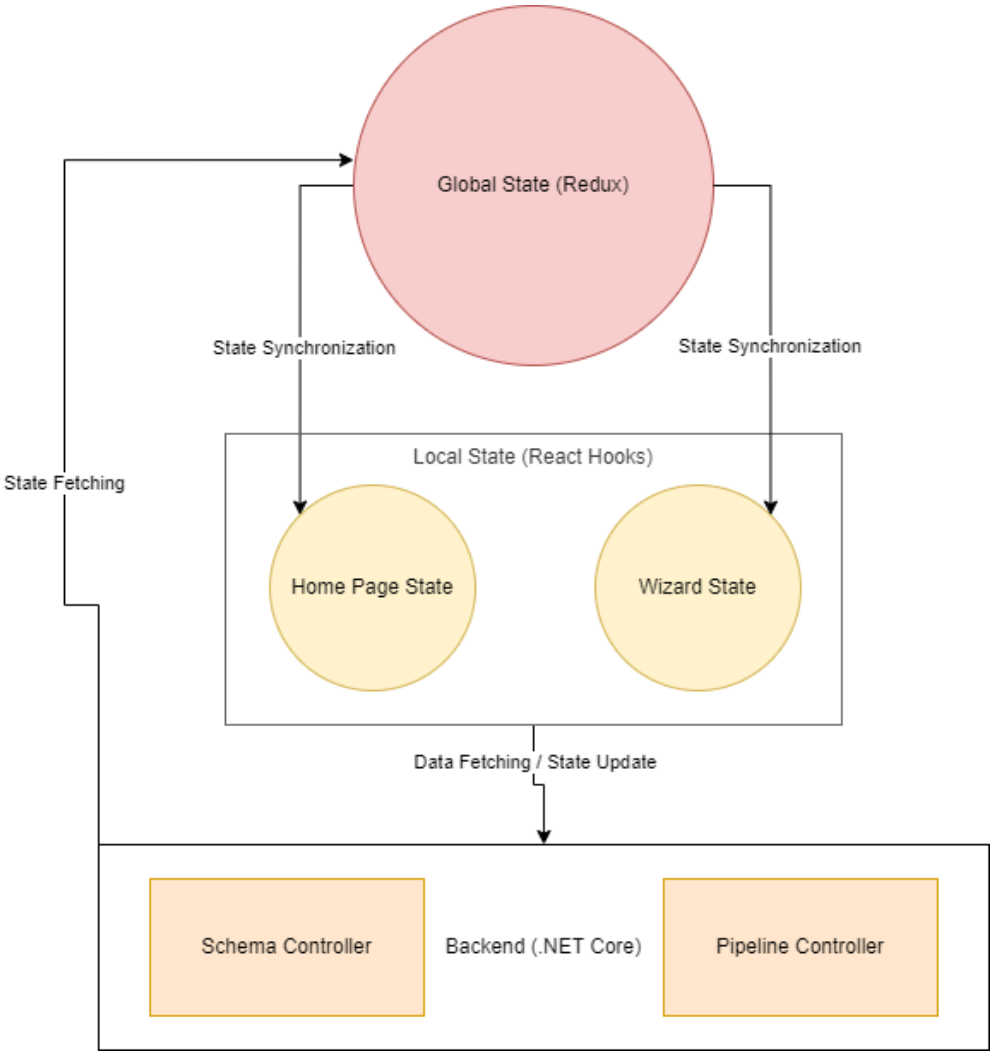

Figure 5.2a: State Management Flow Diagram

5.3 Description

This diagram above in Figure 5.2a illustrates the state management flow within the application, showing how the **Global State (Redux)** and **Local State (React Hooks)** interact with the backend's **SchemaController** and **PipelineController**. It demonstrates how state synchronization, data fetching, and state updates are handled to ensure a consistent user experience.

5.3 Global and Local State Management

1. **Global State (Redux):**

- **Usage:** Redux is used to manage the global state, which includes UI schemas, user data, and pipeline status. This ensures that the state is accessible across different components and that updates are propagated consistently.

```
import { createStore } from 'redux';

const initialState = {
    schema: null,
    userData: {},
    pipelineStatus: null
};

const rootReducer = (state = initialState, action) => {
    switch (action.type) {
        case 'SET_SCHEMA':
            return { ...state, schema: action.payload };
        case 'SET_USER_DATA':
            return { ...state, userData: action.payload };
        case 'SET_PIPELINE_STATUS':
            return { ...state, pipelineStatus: action.payload };
        default:
            return state;
    }
};

const store = createStore(rootReducer);
```

2. **Local State (React Hooks):**

- **Usage:** React Hooks are used within components to manage local state, such as form inputs and step navigation in the Wizard component. This allows for fine-grained control over individual component behaviour.

```
const [formData, setFormData] = useState({});

const handleInputChange = (e) => {
    setFormData({ ...formData, [e.target.name]: e.target.value });
};
```

5.4 Data Flow and Synchronization

1. **Fetching UI and Data Schemas:**

- **Flow:** On component mount, the UI schema is fetched from the backend and stored in the global state. This schema is then used to dynamically render input fields on the Home Page.

```
useEffect(() => {
    fetch('/api/schema/ui')
        .then(response => response.json())
        .then(data => dispatch({ type: 'SET_SCHEMA', payload: data }));
}, []);
```

2. **Storing and Updating UI State:**

- **Flow:** User inputs are captured and stored in the global state using Redux. When the user initiates a pipeline, this data is sent to the backend, and the state is updated to reflect the pipeline status.

```
const handleSubmit = () => {
    fetch('/api/pipeline/start', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(formData)
    })
    .then(response => response.json())
    .then(data => dispatch({ type: 'SET_PIPELINE_STATUS', payload: data.status }));
};
```

3. **Updating UI Based on Pipeline Status:**

- **Flow:** The frontend regularly polls the backend for pipeline status updates. These updates are stored in the global state, triggering UI changes to display the current status or final results.

```
useEffect(() => {
    const interval = setInterval(() => {
        fetch(`/api/pipeline/status/${taskId}`)
            .then(response => response.json())
            .then(data => dispatch({ type: 'SET_PIPELINE_STATUS', payload: data.status }));
    }, 5000);

    return () => clearInterval(interval);
}, [taskId]);
```

5.5 Evaluation

- **Design Quality:** The state management plan is comprehensive, ensuring that the application's state is consistent and synchronized across all components.

- **Functionality:** The plan effectively meets the requirements, handling complex state dependencies and ensuring a responsive UI.

- **Scalability:** The use of Redux for global state management ensures that the application can scale while maintaining state consistency.

---

6. Pipeline Orchestration Design

6.1 Overview

The pipeline orchestration is a critical component of the application, responsible for managing and processing user data submitted through the UI. This is done using Azure Storage queues, which handle the asynchronous processing of tasks by a Python microservice. The design focuses on ensuring that data is passed efficiently from the frontend to the backend, processed in a scalable manner, and that the UI is kept updated in real-time based on the status and results of the pipeline.

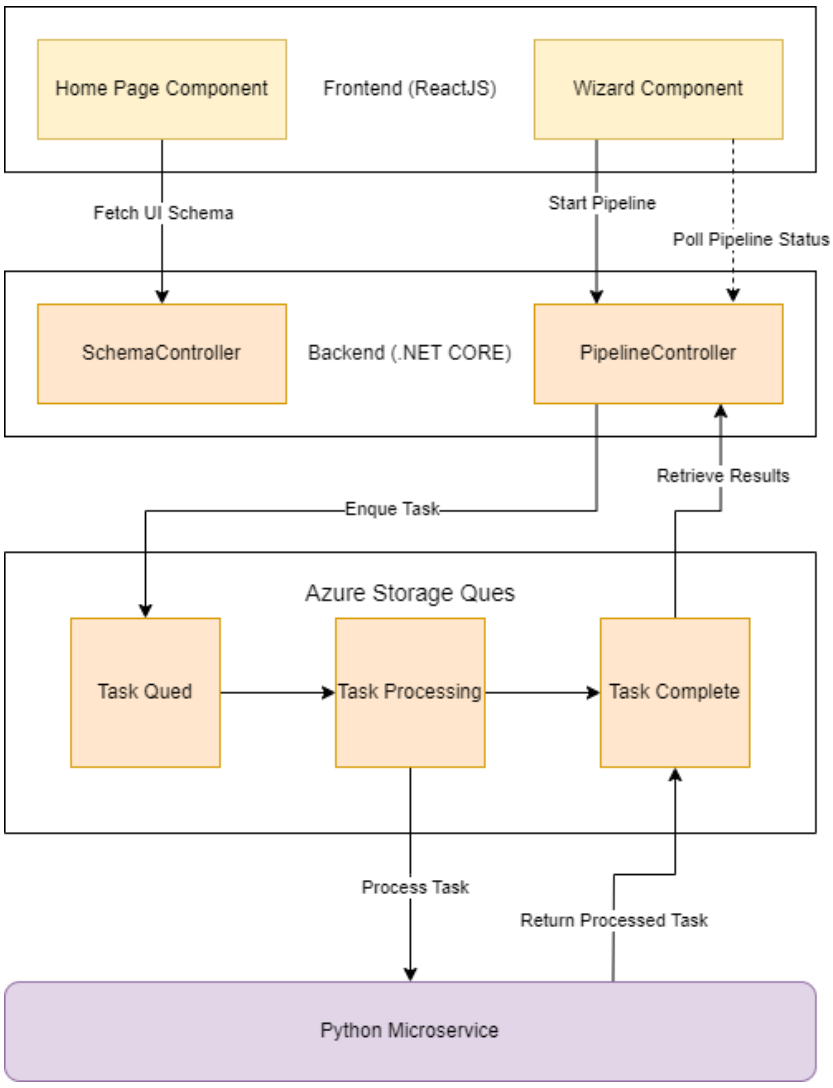

Figure 6.1a: Pipeline Flow Diagram

6.2 Description

This diagram shown in Figure 6.1a illustrates the flow of tasks through the pipeline, from initial enqueueing by the **PipelineController** to processing by the **Python Microservice**, and the return of results to the **PipelineController**. It shows how tasks are managed and processed asynchronously using **Azure Storage Queues**.

6.3 Design Patterns

- **Publisher-Subscriber Pattern:** The backend publishes tasks (user data) to Azure Storage queues, while the Python microservice subscribes to these tasks, processing them and returning the results. This pattern ensures loose coupling between the task producers (frontend/backend) and consumers (microservice).

- **Chain of Responsibility Pattern:** Applied in the pipeline, this pattern allows each stage of the pipeline to process tasks sequentially, passing the task to the next stage if applicable. This design supports complex workflows where different processing stages are needed.

6.4 Detailed Pipeline Orchestration Design

1. **Design the Use of Azure Storage Queues to Manage the Pipeline Tasks**

- **Task Creation and Enqueueing:**

    - When a user submits data through the frontend, the backend receives this data via a RESTful API. The data is validated and transformed into a task object, which is then enqueued into an Azure Storage queue.

    - Each task contains metadata, such as a unique task ID, user-specific data, and the status of the task (e.g., "Queued", "Processing", "Completed").

    - Azure Storage queues are chosen for their reliability, scalability, and ability to handle large volumes of tasks asynchronously.

```
public async Task<string> EnqueueTaskAsync(UserDataModel userData)
{
    var task = new PipelineTask
    {
        TaskId = Guid.NewGuid().ToString(),
        UserData = userData,
        Status = "Queued",
        CreatedAt = DateTime.UtcNow
    };

    // Convert the task to JSON and enqueue it
    var message = new CloudQueueMessage(JsonConvert.SerializeObject(task));
    await _queue.AddMessageAsync(message);

    return task.TaskId;
}
```

- **Task Processing by Python Microservice:**
    - The Python microservice continuously monitors the Azure Storage queue for new tasks. Upon dequeuing a task, the microservice processes the data, performs the necessary calculations (e.g., actuarial models), and updates the task status.
    - After processing, the results are re-enqueued in another queue (e.g., a results queue) or sent directly back to the backend.

```python
# python
def process_task():
    while True:
        task = queue.get_message()
        if task:
            # Process the task (e.g., run actuarial models)
            results = run_actuarial_model(task.user_data)

            # Update task status and return results
            task.status = "Completed"
            task.results = results

            # Enqueue the processed task in the results queue
            results_queue.put(task)
```

2. **Plan How User Data from the UI Will Be Passed to the Pipeline Tasks**

- **Data Flow from UI to Backend:**
    - When a user submits data, the frontend packages this data according to the UI schema provided by the backend. The data is then sent via a POST request to the backend's API.
    - The backend receives this data, validates it against the data schema, and prepares it for processing. The prepared data is then included in a pipeline task object and enqueued into the Azure Storage queue.

```javascript
// javascript
const handleSubmit = (inputData) => {
    fetch('/api/pipeline/start', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(inputData)
    })
    .then(response => response.json())
    .then(data => {
        console.log('Pipeline started:', data);
        // Update UI to reflect that processing has started
    });
};
```

- **Task Object Structure:**
    - The task object includes user data, task ID, status, and timestamps. This object is serialized into JSON before being enqueued, ensuring that it can be easily processed by the microservice.

```csharp
// csharp
public class PipelineTask
{
    public string TaskId { get; set; }
    public UserDataModel UserData { get; set; }
    public string Status { get; set; }
    public DateTime CreatedAt { get; set; }
    public DateTime? CompletedAt { get; set; }
    public string Results { get; set; }
}
```

3. Describe How the UI Will Be Updated Based on the Pipeline Status and Results

- **Real-Time UI Updates:**
    - The frontend regularly polls the backend for the status of the pipeline. This polling ensures that the user is kept informed about the progress of their task.
    - Once the backend receives a completed task from the results queue, it updates the task status in the database and returns the results to the frontend.

```javascript
// javascript
useEffect(() => {
    const interval = setInterval(() => {
        fetch(`/api/pipeline/status/${taskId}`)
            .then(response => response.json())
            .then(data => dispatch({ type: 'SET_PIPELINE_STATUS', payload: data.status }));
  }, 5000);

  return () => clearInterval(interval);
}, [taskId]);
```

- **Final UI Updates:** When the task status changes to "Completed", the frontend updates the UI to display the results. This might involve updating a results panel, generating charts, or displaying success/failure messages.

6.5 Evaluation

- **Design Quality:** The use of Azure Storage queues ensures that the pipeline is scalable and reliable, with the ability to handle high volumes of tasks asynchronously.
- **Functionality:** The design effectively meets the requirements for task management, real-time updates, and UI interaction, providing a seamless user experience.
- **Scalability:** The architecture is designed to handle varying loads by leveraging cloud-native services and design patterns that promote decoupling and parallel processing.

---

7. Sample Code

7.1 Backend API Endpoints (ASP.NET Core)

1. **Get UI Schema**

```csharp
// csharp
[HttpGet("ui")]
public async Task<IActionResult> GetUiSchema()
{
    var schema = await _unitOfWork.Schemas.GetUiSchemaAsync();
    return Ok(schema);
}
```

2. **Save UI State**

```csharp
// csharp
[HttpPost("state")]
public async Task<IActionResult> SaveUiState([FromBody] UiStateModel uiState)
{
    _unitOfWork.UiStates.Add(uiState);
    await _unitOfWork.CompleteAsync();
    return Ok();
}
```

3. **Start Pipeline**

```csharp
// csharp
[HttpPost("start")]
public async Task<IActionResult> StartPipeline([FromBody] UserDataModel userData)
{
    var taskId = await _pipelineService.EnqueueTaskAsync(userData);
    return Ok(new { TaskId = taskId, Status = "Pipeline started successfully" });
}
```

7.2 Frontend Code Snippets (ReactJS)

1. **Dynamic Form Rendering**

```javascript
// javascript
useEffect(() => {
    fetch('/api/schema/ui')
        .then(response => response.json())
        .then(data => setSchema(data));
}, []);
```

2. **Wizard Component Navigation**

```javascript
// javascript
const Wizard = ({ steps }) => {
    const [currentStep, setCurrentStep] = useState(0);
```

```
    const nextStep = () => setCurrentStep(prev => prev + 1);
    const prevStep = () => setCurrentStep(prev => prev - 1);

    return (
        <div className="wizard">
            {steps[currentStep]}
            <div className="navigation">
                <button onClick={prevStep} disabled={currentStep === 0}>Previous</button>
                <button onClick={nextStep} disabled={currentStep === steps.length -
1}>Next</button>
            </div>
        </div>
    );
};
```

7.3 Evaluation

- **Functionality:** The provided code snippets demonstrate the backend's capability to handle user requests, manage pipeline tasks, and synchronize with the frontend for a cohesive application experience.

---

8. Conclusion

This document provides a comprehensive blueprint for a scalable, maintainable, and user-friendly web application that interfaces with an actuarial model microservice. The design leverages modern design patterns, best practices, and cloud-native technologies to meet all specified requirements.