Please complete **Tutorial 5** before starting this assignment.

The concept you intend or plan to utilize for building the web services or web app should be **distinct** from that presented in Tutorial 5 and should not be a duplicate of the sample concepts provided in this document.

➢ What is Full Stack Development?

Full stack development refers to the entire process of building of web applications, which includes the frontend (user interface) and backend (database and logic) components. In Tutorial 5, we utilized a full-stack method of development where the application incorporated both the frontend (which is situated in the frontend folder) and backend (which is situated in the backend folder) into a unified code base. Tutorial 5 specifically divided this method of full stack development into two distinct parts: Part A focused on the development of the code required for a web application to operate. This involved integrating with data management systems like MySQL and handling data through the Create, Read, Update, and Delete (CRUD) operations. The backend code consisted of several web services that formed a REST API developed as a Node.js project. On the other hand, Part B, or the frontend, utilized jQuery to facilitate user interaction with the backend code.

For an in-depth review of the distinction between frontend and backend in application development, AWS offers an informative article on this topic. Figure 1 presented below offers a concise overview of the communication between the several layers that are commonly found in a modern web-based application, including both the frontend and backend components. The API layer in this diagram utilizes the same approach previously presented in Tutorial 5, employing a RESTful architecture that incorporates web services to form an API.

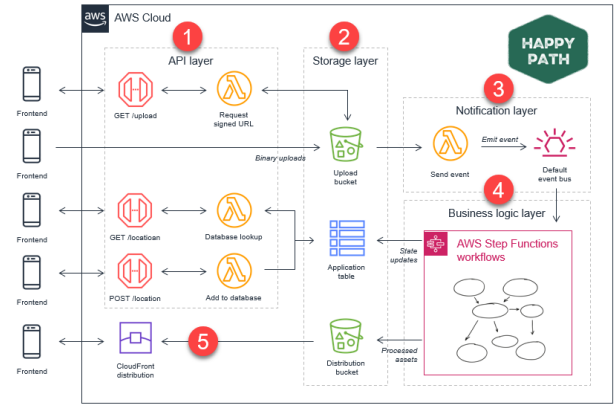


Fig. 1: Full Stack Development (Frontend and Backend) from AWS
image source: https://aws.amazon.com/compare/the-difference-between-frontend-and-backend/

➢ Building RESTful APIs

In tutorial 5, you have acquired knowledge and skills necessary to create a **Representational State Transfer (REST) Application Programming Interface** (API) composed of multiple web services that facilitates the **C**reate, **R**ead, **U**pdate, and **D**elete (CRUD) operations. You also learned how to develop a frontend interface that consumes or uses the API. As demonstrated in Tutorial 5 and Modules 3 & 4 lecture notes, it is evident that a resource identifier or URI has the capability to be coupled with several HTTP methods, including but not limited to the **GET**, **POST**, **PUT**, and **DELETE** methods, which are all relevant to the CRUD activities. In the context of this endeavor, our primary focus was geared towards two essential resource IDs.

- URI 1: **http://localhost:3000/**
  1. GET → retrieve ALL records for all cities
- URI 2: **http://localhost:3000/city**
  1. POST → **c**reate a new record for a city whose data is supplied by the client (request body)
  2. GET → **r**etrieve a single record for a city name (supplied by client)
  3. PUT → **u**pdate an existing record for a city whose data is supplied by the client
  4. DELETE → **d**elete an existing record for a city

Although it is feasible to utilize resource identifiers for performing CRUD operations by passing data in the URI's argument's section, it becomes evident that maintaining a consistent interface and managing resources through this approach becomes challenging from an application development standpoint, as exemplified by the following URI sample:

http://localhost/payroll?devkey=1&action=search&type=department&keyword=DepartmentID

scheme — host — service — delimiter — data via arguments (name: value pairs)

Although the above design has the capability of creating REST services, it departs from the core architectural principles of REST, rendering it **non-RESTful**. In the context of REST, which emphasizes on the fact that resources are identified by Uniform Resource Identifiers (URIs) and can typically be accessed via protocol like the Hypertext Transfer Protocol (HTTP). Nevertheless, the aforementioned design does not conform to this assertion.

Alternatively, the following architectural style is RESTful:

- URI 1: **http://localhost:3000/campus**
- URI 2: **http://localhost:3000/building**
- URI 3: **http://localhost:3000/building/floor**
- URI 4: **http://localhost:3000/building/floor/room**

Thermostat at the 4th floor of a smart building can be managed via a RESTful API such that:

- HTTP GET → retrieve a status
- HTTP POST → turns on a thermostat
- HTTP PUT → controls or manages a thermostat
- HTTP DELETE → turns off a thermostat

As an illustration, URI 2 has the capability to employ HTTP methods for the purpose of establishing a mapping that enables remote control over all sensors located within a designated building. The management and monitoring of these sensors on a large scale might involve individuals who have adequate authorization, such as security personnel, law enforcement officials, or building administrators. The utilization of the HTTP DELETE method has the capability to deactivate all smart sensors throughout the entirety of the building. The design presented in this example demonstrates that the organization of resources may be effectively represented by **resource identifiers**. Additionally, the **interfaces** are designed in a consistent manner, regardless of their specific functionalities. Moreover, the level of granularity increases as the hierarchical folder structure grows, while the functionality attributed to the innermost folder becomes more coherent and highly specific, transitioning from **generalization** to **specialization**. The provided example exemplifies a RESTful architecture design for building APIs.

[**An Operating Scenario**] In the example above, assuming that the UWT campus has smart buildings equipped with sensors, the retrieval of sensor status within a specific office can be accomplished by issuing an HTTP GET request to the following URI: http://localhost:3000/UWT/CP/227. Similarly, to activate the office light in CP 227 at the UWT campus, a POST HTTP request can be sent to the URI http://localhost:3000/UWT/CP/227/light. Did you figure out by now the URI to deactivate or turn off the office light in CP 227 yet?

Task for Assignment 3

[**Objective & Goals**] The objective of this assignment is to develop and implement a full stack web application (web app) utilizing **jQuery** for the client user interface (frontend), **Node.js** for the server-side functionality (backend), and a **database** for data storage. To this extent, identify an idea or concept which you can use to build this web app. Then, identify the features or functionalities that this app will offer through the implementation of an API. Then, develop a frontend that showcases the usefulness of the web app and/or features implemented in the API. By completing this assignment, students will acquire the skills to develop a RESTful API, execute **CRUD operations**, design a user-friendly interface, and technically be able to describe the API.

[**Web App Folder Organization**] All code elements of this assignment must be stored into a single root folder (e.g., '\assign3\backend' and '\assign3\frontend\'). The backend code should be written in a directory called "backend". The frontend code should be written in a directory called "frontend". Instructions on how to run the assignment, including both the frontend and backend using terminal command should be clearly provided within the root of the assignment folder in a readme.txt file.

[Part A: Backend] The **backend** of this web application is a **RESTful API comprising two web services** that collectively work together to deliver a larger programming functionality. The level of granularity exhibited by a web service is contingent upon its specific functionality. Therefore, it is imperative that the design and implementation of each web service for building the RESTful API have sufficient depth and complexity comparable to the examples provided in this activity. Additionally, it is required that every web service offers a **second-degree service capability** in the Uniform Resource Identifier (URI), in conjunction with the utilization of Create, Read, Update, and Delete (CRUD) operations using HTTP methods (sample Appendix A). The various features offered within the index.js file that was developed throughout this activity can be considered as a **single web service** which aims to **deliver city-specific features in terms of population-related information**. It strongly recommended that any web service should make use of HTTP methods to the greatest extent possible. However, not all web services must have features that utilize all HTTP methods. The web services you develop should not rely on existing web services from existing service providers (i.e., you need to implement your own web services).

[RESTful API Design & Modularity]

- Design a well-structured RESTful API that adheres to recommended REST practices and conventions (refer to the sample in Appendix A). Ensure the URL endpoints are clear and indicative of the resources and actions they

represent. You do not need to implement the same number of features listed in Appendix A; the number of features for each URL endpoint should align with your design and idea. Appendix A provides only an example.
  - **HTTP headers** play a vital role in transmitting additional information between the client and server.
  - HTTP headers can be either in the request or response messages, or both.
  - Implement commonly used HTTP headers such as 'Content-Type', 'Cache-Control', or custom headers to transmit custom metadata.
- Structure your API's folder hierarchy by implementing each web service as a separate JavaScript file or controller. Arranging components within your API fosters modularity (e.g., use a components folder).

[Database Interaction, CRUD Operations, & Middleware Functions]

- Utilize a database system such as MongoDB, PostgreSQL, or MySQL to facilitate the storing and retrieval of data (see to Part B for further details).
- Develop and execute CRUD (Create, Read, Update, Delete) operations to interact with data resources.
  - Object-Relational Mappers (ORM) like Mongoose for MongoDB or Sequelize for PostgreSQL/MySQL can be utilized to design database models, execute CRUD operations, and handle database connections.
- Establish endpoints for every CRUD operation and ensure they handle requests in an appropriate manner.
- Incorporate middleware functions such as '**body-parser**' to parse JSON request bodies and '**cors**' to handle cross-origin requests.

[Error Handling]

- API should effectively manage errors by delivering useful error messages and HTTP status codes in response messages.

[API Documentation]

- Document detailed API endpoints, including the request methods, request/response formats, and arguments.
- Provide specific and thorough documentation to instruct frontend developers on the utilization of backend API.
- Your documentation must be in an HTML file that includes full details regarding API documentation.
- For a demonstration of the structure and components of endpoint documentation, please go here.
- Appendix C contains examples for the API documentation.

[**Illustrative Example**] WashingtonAirQuality Inc. has developed a RESTful API that enables its clients to utilize air quality data manually gathered through two data service providers: (a) the web service offered by the US Department of Energy and (b) the web service provided by the City of Tacoma. The collected data by WashingtonAirQuality Inc. is ultimately or manually stored in a MySQL database table named "airqualitydb". WashingtonAirQuality Inc. has built a RESTful API that offers a wide range of features to its customers, allowing for seamless integration of these services into other web applications. The majority of these features primarily utilize the Create, Read, Update, and Delete (CRUD) operations over the HTTP. However, not every feature must make use of all CRUD operations. This depends on the design and the functionality offered the web service resource identifiers. Appendix A provides a complete RESTful API reference for two web services that share the database for illustration purposes.

[Part B: Database] The web application must incorporate a data system or a **database** in its underlying structure in order to facilitate the exchange, interaction, and manipulation of data within the backend. It is permissible to utilize pre-existing datasets, either in whole or in part, for the purpose of constructing the backend database. You may also simulate or create your own data for the database. However, the magnitude of the data should be adequate to the level of data used in this activity (e.g., 10-15 records or so). You may wish to create multiple tables to organize the data as needed. However, this is not required as database design is not strictly part of the overall evaluation. You may choose any suitable database system (e.g., MongoDB, PostgreSQL, MySQL).

[Part C: Frontend] Create a user-friendly and aesthetically appealing frontend interface that is consistent with the theme of the web application. Appendix D provides a sample UI example. Appendix E provides code examples related to the below elements.

- [**User Interface Design**] The frontend should integrate modern UI design concepts, such as responsive layout, consistent style, and user-friendly navigation. For example, provide a navigation bar that includes links to different parts of the application, a sidebar for filtering data, and interactive components for user interaction.

- [**Component Implementation & State Management**] Develop modular UI elements in jQuery to encapsulate both the user interface and their accompanying functionality. Create reusable jQuery functions for common UI elements, such as a 'Card' component to display data items with a consistent style and functionality across the application. Manage state using JavaScript variables and manipulate the DOM to reflect state changes.
- [**Routing**] Utilize a jQuery-based routing solution or implement custom routing logic to enable client-side navigation between different views or pages. Define the routes for each page or component within the application, updating the DOM to display the appropriate content based on the current URL.
- [**Data** Fetching **and Binding**] Retrieve data from the backend API by sending asynchronous requests using jQuery's AJAX methods. Display the fetched data in real-time within the user interface elements. Ensure to handle loading issues and errors that may occur during request/response exchanges (see Module 3 Example).
- [**Form Handling**] Develop user input forms with validation and error handling functionality. Ensure that you are able to manage form submissions and data alterations. For instance, design a form that allows users to add or update items. Ensure that the form includes input validation for needed field elements, input format, and inline display of error warnings.
- [**Styling and Design**] Implement uniform style and adaptable layout for user interface components. You have the option to either develop your own unique styling, utilize pre-existing styling, or employ CSS frameworks like Bootstrap (or Bootswatch).

What to Submit

Submit a **userid-a3zip** file containing the following elements (where userid is your UW userid):

a) **userid-a3-source.zip**: This compressed file should contain **all** of the Node.js files and folders **except** node_modules (i.e., root folder files, src, and public are only needed). Do not include the node_modules folder as it can be recreated using npm install. Ensure to include the package.json and the README.md file.

b) **readme.txt**: Create a readme.txt file containing accompanying documentation which:
- briefly describes the web application, highlighting its purpose, key features, and technologies used,
- instructions on how to run the (i) backend, (ii) frontend, (iii) and the database, and
- additional notes on the data source(s) that has been utilized in this API to compile the data within the database.
- Appendix B provides a sample example of a readme.txt file.

c) **a3.html:** The HTML file serves as a documentation for the RESTful service API, offering instructions on how to effectively utilize the service. It encompasses essential information such as adequate documentation, guidelines on utilizing the service (including URL address design), sample examples, and input/output specifications. It is assumed that the HTML documentation shall serve as the means by which clients can make use of the web services being offered. The clarity and organization of how the content of this documentation file should be prioritized. Below are sample examples that you can follow:
- https://aqicn.org/json-api/doc/
- https://openweathermap.org/api/air-pollution

d) **assign3.mp4** (or webm or equivalent): Use screen recording software to capture a video presentation of the web application. The video demonstration should be 10 minutes long.
- Ensure that the demonstration covers all the key features and functionalities of the web application.
- The presentation should demonstrate CRUD operations within the API, and practical application of JSON in the context of data sharing and handling.
  - While the frontend should be used to demonstrate all key features of the web application, you may also use tools like ThunderClient in VSCode, and/or browser developer tools to demonstrate specific operations in the backend API during the demonstration.

- Demonstration Strategy: It is recommended to develop a test strategy prior to recording the presentation, outlining the procedures that will be presented, and thereafter incorporating them into the recording. It would be useful to create a PowerPoint presentation to organize the presentation. Below is a recommended presentation format (slide per each).

Introduction (briefly introduce the purpose and goals of the web application)	1 min
Data source(s) used and database employed for the web application	½ min
Overview: Provide guided tour of the web application's user interface (frontend) and demonstrate the main features and functionalities of the application. Highlight any key aspects of the application.	2 mins
Backend/API Demonstration: Demonstrate on a working environment how the frontend interacts with the backend API, and demonstrate the CRUD operations using the application. Further, demonstrate aspects such as error handling, HTTP status codes in response messages, HTTP headers, among others.	4 mins
API Documentation: Walkthrough all of the API endpoints, request methods, request/response formats and parameters.	2 mins
Possible Enhancements: Discuss any possible extensions of the API or future work.	½ min

- Clarity and Demonstration Environment
  - Ensure clear audio and video quality.
  - Ensure the video file size is manageable for uploading and sharing. Please consider compressing the video size (use video compression online platforms like veed.io, clideo.com, etc.).
  - You may consider enhancing the video with subtitles (e.g., https://www.veed.io/tools).
  - Ensure that the web application is fully functional and accessible during the presentation recording.

Rubric and Grading Scheme (35 marks)

14 marks	Backend API & Database • [4 marks] API design and implementation (clear and well-structured RESTful API design • [4 marks] proper implementation of CRUD operations • [2 marks] proper use of database queries and having access to sufficient data records • [2 marks] effective use of middleware for request handling • [1 mark] use of appropriate HTTP headers • [1 mark] error handling and proper use of HTTP status code
12 marks	Frontend • [4 marks] intuitive and visually appealing UI, and use of consistent styling and layout • [4 marks] use of reusable and modular components, effective state management • [2 marks] proper handling of user interactions and data display • [2 marks] form handling and validation
4 marks	API Documentation • [2 marks] informative and descriptive API documentation for all endpoints and HTTP methods • [2 marks] easy to navigate and follow
5 marks	Demonstration • [2 marks] comprehensive coverage of all key features and functionalities • [2 marks] demonstrate backend/frontend interaction, CRUD operation and other features • [1 mark] effective communication of key points during demonstration

Appendix A: Air Quality in Washington State (AirQuWash) API Example

This RESTful API is composed of two web services: (a) Web Service 1 (serviceA.js) and (b) Web Service 2 (service.js). The goal of this RESTful API, which we call AirQuWash API is to provide air quality data across major cities in the Washington State. Assume that the air quality data is collected from a data source (e.g., air quality data source provider) and stored within a database associated with the AirQuWash API. The AirQuWash API has two service endpoints: '**/airquality**' and '**/city**'. Both endpoints would retrieve and provide real-time hourly air quality data, which is stored within the database. However, '**/airquality**' is intended to cover larger geographic setting (e.g., state-wide) whereas '**/city**' is intended to serve a smaller region (e.g., city-specific). Let's introduce the two web services in the AirQuWash API.

Web Service 1: http://localhost:3000/airquality

- The proposed service endpoint or URI serves at developing a service endpoint that offers a comprehensive **summary** of the air quality in at least 10 major cities in Washington State. This endpoint would retrieve and provide real-time hourly air quality data, which is stored within the database. The primary emphasis of this web service revolves around the utilization of the GET HTTP method for the purpose of retrieving data through multiple offered features within that service.
- The table below provides a summary of all the features or functionalities offered by this web service. Note that this service provides a **2nd degree granularity** (http://localhost:3000/**airquality**/{**degree1**}/{**degree2**}).

Service endpoint Root: http://localhost:3000/**airquality**	**HTTP Method**	**Overall Functionality or Expected Output**
http://localhost:3000/airquality	GET	Provides an overall assessment of air quality metrics for the entire state of Washington (limited to city data collected).
http://localhost:3000/airquality/pm2.5	GET	Provides a collection of the four top cities in Washington State that exhibit the most extreme levels of PM2.5 concentration.
http://localhost:3000/airquality/pm10	GET	Provides a compilation of the top four cities in Washington State that exhibit the most severe levels of PM10 concentration.
http://localhost:3000/airquality/aqlevels	GET	Provides an overview of the scale employed to measure the levels of air quality index (AQI) for various pollutant concentrations.
http://localhost:3000/airquality/high	GET	Provides a list of cities that exhibit high levels of air quality contaminants across all pollutant concentrations.
http://localhost:3000/airquality/high/PM25	GET	Provides a collection of cities that exhibit high levels of PM2.5.
http://localhost:3000/airquality/high/CO	GET	Provides a collection of cities that exhibit high levels of carbon monoxide concentration.
http://localhost:3000/airquality/low	GET	Provides a collection of cities that have the most minimal levels of all air quality pollutants.
http://localhost:3000/airquality/low/PM25	GET	Provides a collection of cities that have the most minimal concentrations of PM 2.5.
http://localhost:3000/airquality/low/CO	GET	Provides a collection of cities that have the most minimal concentrations of carbon monoxide.
http://localhost:3000/airquality/average	GET	Provides a collection of the averages of all air quality pollutants across all cities.
http://localhost:3000/airquality/average/CO	GET	Provides a single average value of carbon monoxide across all cities.

Web Service 2: http://localhost:3000/city

- The service endpoint offers city-specific air quality information about a particular city in Washington State, and incorporates all CRUD operations into its service design. The table below provides a summary of all the features or functionalities offered by this web service. Note that this service provides a **2nd degree granularity** (http://localhost:3000/**city**/{**degree1**}/{**degree2**}).

Service endpoint Root: http://localhost:3000/**city**	**HTTP Method**	**Overall Functionality or Expected Output**
http://localhost:3000/city	GET	Obtains air quality data for the designated city
	PUT	Updates air quality data for the specified city
http://localhost:3000/city/station	GET	Provide a collection of air quality station(s) located for the specified city.
	POST	Allows to add or associate a new station for the specified city.
	PUT	Updates a station that is associated with the specified city.
	DELETE	Removes a station that is associated with the specified city.
http://localhost:3000/city/forecast	GET	Provides AQI forecast information for the specified city.
http://localhost:3000/city/pm25	GET	Retrieves PM2.5 AQI level for the specified city.
	PUT	Updates the PM2.5 AQI level for the specified city.
http://localhost:3000/city/pm10	GET	Retrieves the PM10 AQI level for the specified city.
	PUT	Updates the PM10 AQI level for the specified city.
http://localhost:3000/city/pm10/high	GET	Retrieves minimum concentrations of PM 10 for the specified city in the last week.
http://localhost:3000/city/pm10/low	GET	Retrieves maximum concentrations of PM 10 for the specified city in the last week.
http://localhost:3000/city/pm25/high	GET	Retrieves minimum concentrations of PM 2.5 for the specified city in the last week.
http://localhost:3000/city/pm25/low	GET	Retrieves maximum concentrations of PM 2.5 for the specified city in the last week.
http://localhost:3000/city/pm10/average	GET	Computes the average concentrations of PM 10 for the specified city during the last month.
http://localhost:3000/city/pm25/average	GET	Computes the average concentrations of PM 2.5 for the specified city during the last month.

Appendix B: Sample readme.txt

```
# Recipes Web Service API

## Overview

This web service API consists of two main services:
1.Recipe Management
2.Find Easy Recipes


## Installation/Running Services
**Installation
   - In VS code open my project in a new workspace, and open the terminal
   - In the terminal navigate to the `backend` directory:
     cd backend

   - Run the following commands to initialize the project and install the required Node modules:
     npm init -y
     npm i express nodemon mysql cors

**To run Recipe Management (Runned on index.js)**

   - Ensure that your package.json has the following script for starting the Recipe Management Web Service:

     "scripts": {
           "start": "nodemon index.js"
      }

   - Then in the terminal Run the following command to start the Recipe Management Web Service:
     npm start

   - This service manages recipes and provides CRUD operations.

**To run Find Easy Recipes (Runned on index2.js)**

   - After running the CRUD operations for the Recipe Management service
     In the terminal stop connecting to the database by entering:
     Ctrl+c  click enter
     y    click enter
     In your package.json change the index.js to index2.js:

     "scripts": {
           "start": "nodemon index2.js"
      }

   - To start the service, run the following command in the terminal:
     npm start

   - This service focuses on finding easy recipes based on various criteria.

## Data Source
The data utilized in this web service API is sourced from the "assign4.sql" file,
which contains a simulated collection of 17 recipes created by me.
```

Appendix C: Sample API Documentation

Recipes Web Service Documentation

Web Service 1: Recipe Management

Runned on index.js
{in package.json make sure that ~ "start": "nodemon index.js"
Then on terminal: npm start}

Get All Recipes

Method:
GET

Endpoint:
/recipes

Example:

```
http://localhost:3000/recipes

Response:
[
{
  "Recipe_id": 1,
  "Title": "Spaghetti Bolognese",
  "Description": "Classic Italian dish with a rich meat sauce.",
  "Instruction": "Cook pasta, make sauce, combine.",
  "Prep_time_minutes": 15,
  "Cooking_time_minutes": 30,
  "Cuisine": "Italian",
  "Difficulty": "Easy",
  "Ingredients": [
    {
```

Create a New Recipe

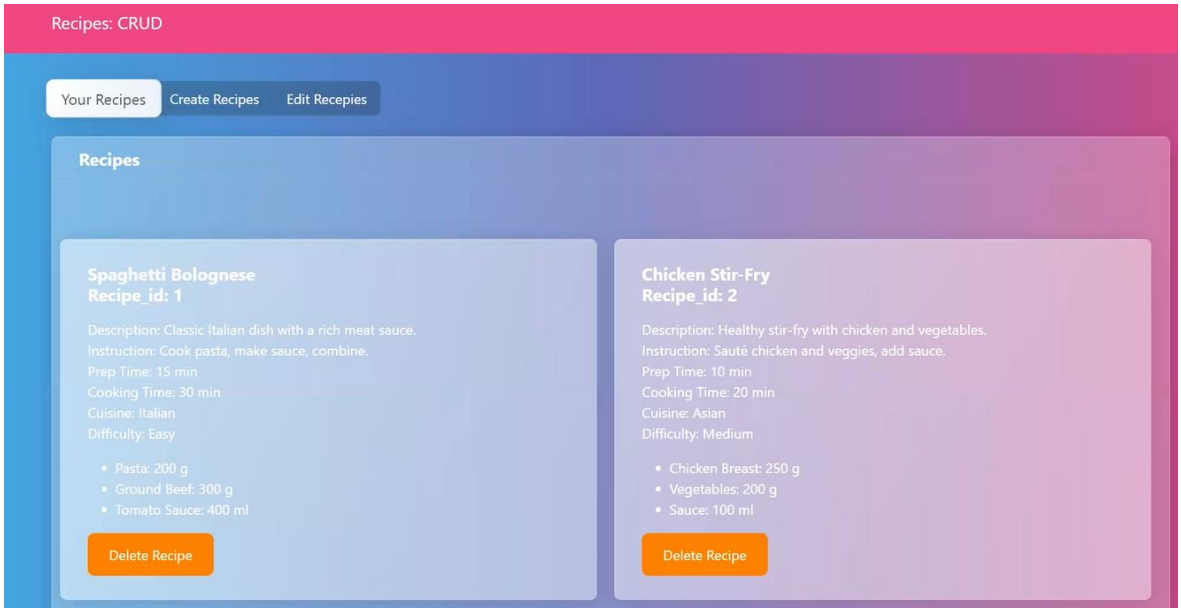
Method:
POST

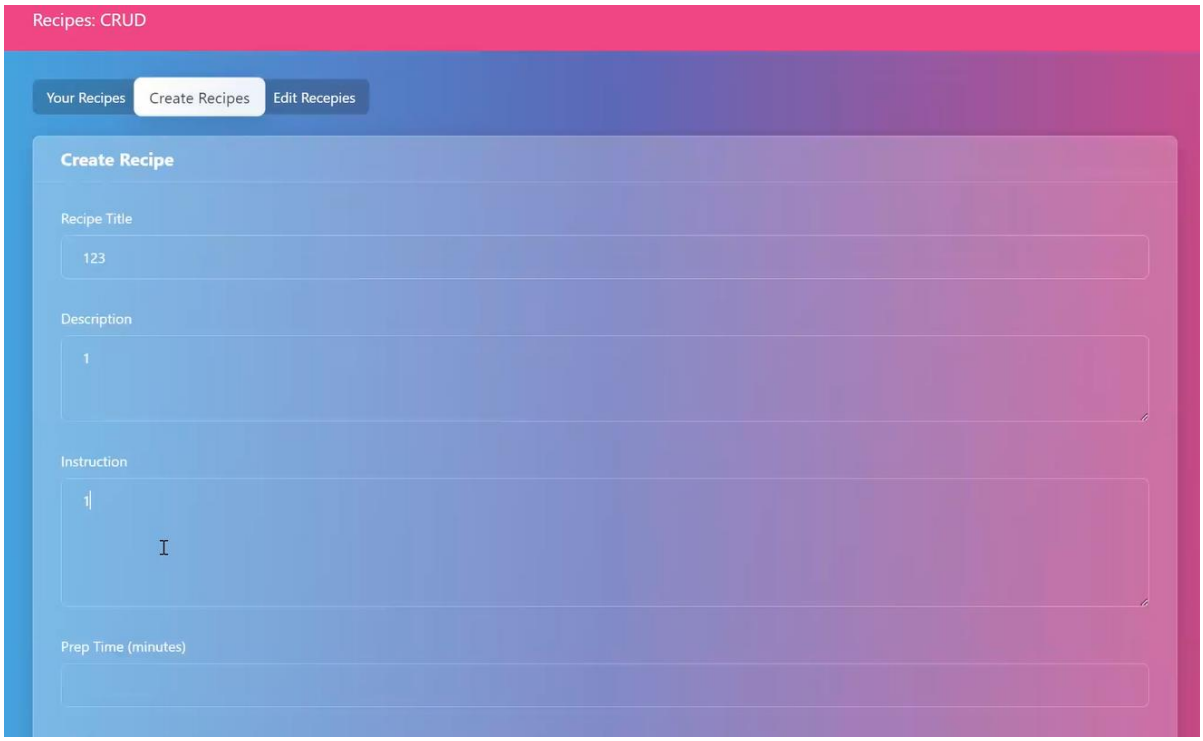
Endpoint:
/recipes

Content-Type: application/json
Request Body:

```
{
    "Title": "Recipe Title",
    "Description": "Recipe Description",
    "Instruction": "Recipe Instruction",
    "Prep_time_minutes": 30,
    "Cooking_time_minutes": 60,
    "Cuisine": "Italian",
    "Difficulty": "Medium",
    "Ingredients": [
        {"name": "Ingredient 1", "quantity": 100, "unit": "g"},
        {"name": "Ingredient 2", "quantity": 200, "unit": "ml"}
    ]
}
Success-Response:
Status: 200 OK
{
    "Success": "Successful: Recipe was added!"
}
Error-Response:
Status: 400 Bad Request
```

Appendix D: Sample Frontend UI in jQuery





Appendix E: Code Samples for Implementing UI in jQuery

jQuery Reusable Card Component

```
// Creates a reusable card component with the given data.
function createCard(data) {
    // Create the card container
    var card = $('<div class="card"></div>');

    // Add the title to the card
    card.append('<h3>' + data.title + '</h3>');

    // Add the description to the card
    card.append('<p>' + data.description + '</p>');

    // Return the jQuery object representing the card
    return card;
}

// Example usage of the createCard function.
$(document).ready(function() {
    // Example data array
    var dataItems = [
        { title: 'Card 1', description: 'This is the first card.' },
        { title: 'Card 2', description: 'This is the second card.' },
        { title: 'Card 3', description: 'This is the third card.' }
    ];

    // Iterate over the data items and create cards
    dataItems.forEach(function(item) {
        // Create a card for each data item
        var card = createCard(item);

        // Append the card to the container
        $('#card-container').append(card);
    });
});
```

jQuery State Management

```
// Define the initial state object
var state = {
    items: []
};

// Renders the items in the state by creating and appending card components.
function render() {
    // Clear the container before rendering
    $('#container').empty();

    // Iterate over each item in the state and create a card component
    state.items.forEach(function(item) {
        var card = createCard(item);
        $('#container').append(card);
    });
}
// Adds a new item to the state and re-renders the UI.
function addItem(item) {
    // Add the new item to the state
    state.items.push(item);

    // Re-render the UI to reflect the new state
    render();
}

// Example usage of the state management and rendering logic.
$(document).ready(function() {
    // Example items to be added
    var newItem1 = { title: 'New Card 1', description: 'Description for new card 1' };
    var newItem2 = { title: 'New Card 2', description: 'Description for new card 2' };

    // Add the new items to the state
    addItem(newItem1);
    addItem(newItem2);
});
```

AJAX Requests for Data Fetching and Posting

```
// Fetches items from the backend API and updates the state.
function fetchData() {
    $.get('/api/items', function(data) {
        state.items = data;
        render();
    });
}

// Posts a new item to the backend API and adds it to the state.
function postData(item) {
    $.post('/api/items', item, function(response) {
        addItem(response);
    });
}
```

Fetching Data with AJAX

```
// Fetches data from the backend API and updates the UI.
function fetchData() {
    $.ajax({
        url: '/api/data',
        method: 'GET',
        success: updateUI,
        error: function() {
            alert('Error fetching data');
        }
    });
}

// Updates the UI with the fetched data.
function updateUI(data) {
    $('#data-container').empty();
    data.forEach(function(item) {
        $('#data-container').append('<div>' + item.name + '</div>');
    });
}

// Fetch data when the document is ready.
$(document).ready(fetchData);
```

Custom Routing Implementation

```
// Handles hash change events to load the corresponding page.
$(window).on('hashchange', function() {
    loadPage(window.location.hash.substring(1));
});

// Loads and displays the specified page, hiding others.
function loadPage(page) {
    $('.page').hide();
    $('#' + page).show();
}
```