Consuming APIs Using .Net Core and MVC

Adding External Data to Your Web App

Scott Tremaine
Software Developer and Educator

Breakpoint Coding Tutorials

Contents

Introduction

In the digital age, the ability to interact with external data sources and services through APIs has become a fundamental skill for developers. APIs are the backbone of modern web services, allowing your applications to remain scalable, modular, and efficient. This guide focuses on leveraging the .NET Core framework and MVC architecture to consume APIs effectively, ensuring that your applications can access a wide range of functionalities from external services.

Role of APIs in Modern Web Development

In modern web development, APIs (Application Programming Interfaces) serve as critical facilitators of connectivity and functionality across various digital platforms and systems. They are the building blocks that enable developers to enhance and extend the capabilities of their applications by interfacing with external services and resources. The influence of APIs spans across numerous aspects of technology, making them indispensable tools in the developer's toolkit.

Enabling Data Exchange and Integration

APIs provide a standardized way for applications to communicate with each other. They allow web applications to seamlessly send or receive data from external servers, facilitating real-time data exchanges that are pivotal in today's interconnected digital ecosystem. For example, when a user books a flight through an online travel service, APIs are what allow the website to interface with airline databases to retrieve flight options, make bookings, and even check seat availability—all in real time.

Integrating Third-Party Services

Beyond basic data exchange, APIs empower web applications to integrate functionality from third-party services. This can include embedding social media feeds, processing payments through gateways like PayPal or Stripe, or utilizing Google Maps for geolocation services. These integrations are made possible by APIs, which dictate how software components should interact without requiring developers to share all the code of their applications. This encapsulation of functionality allows developers to add complex features to an application without building them from scratch, significantly reducing development time and costs.

Supporting Scalability and Flexibility

APIs are designed to be independent of the underlying implementation details. This abstraction allows businesses to scale and evolve their applications without major overhauls. For example, if an application uses an API to fetch data, it doesn't need to care where the data comes from or how it is generated. This means that if the data source needs to change or scale up, the API can often handle this without any changes needed on the part of the client application.

Enhancing User Experience

By leveraging APIs, developers can create more responsive, intuitive, and feature-rich applications. APIs facilitate the creation of applications that can provide users with a seamless experience, regardless of where the data resides or what type of device they are using. This is particularly important in an era where users expect real-time interactions and services that are tailored to their needs and preferences.

Facilitating Innovation

APIs foster innovation by enabling developers to build upon existing platforms and services. By providing the "lego blocks" of functionality, APIs allow developers to experiment with new ideas and solutions quickly and cost-effectively. This ecosystem of APIs has led to the rapid expansion of digital services and products, propelling forward the modern web development landscape.

Setting Up Your Development Environment

Installing and Configuring .NET Core

.NET Core is a versatile, high-performance framework designed for building modern applications across any platform. The first step in leveraging this powerful framework is to install it.

Step-by-Step Installation Guide

1. **Download the .NET Core SDK**
    - Visit the official .NET download page.
    - Select the SDK according to your operating system (Windows, Linux, or macOS).
2. **Installation**
    - Follow the platform-specific instructions provided on the download page to install the .NET Core SDK.
3. **Verification**
    - Once installed, open your command line interface (CLI) and run the following command to verify the installation

```
dotnet --version
```

    - This command should return the version of the .NET Core SDK that you installed, confirming that it is ready for use.

Overview of the MVC Architecture

Model-View-Controller (MVC) is an architectural pattern widely used in web application development. MVC separates the application into three interconnected components, making it easier to manage complexity and development

- **Model:** The central component of the pattern. It directly manages the data, logic, and rules of the application.
- **View:** Any representation of information, such as a chart, diagram, or table. Multiple views of the same information are possible.
- **Controller:** Accepts input and converts it to commands for the model or view.

In the context of .NET Core, MVC provides a powerful way to build scalable and testable web applications.

Setting Up an MVC Project in Visual Studio

Visual Studio is a popular integrated development environment (IDE) from Microsoft. It offers comprehensive tools and services for developing .NET applications.

Creating a New MVC Project

1. **Launch Visual Studio**
   - Start Visual Studio and select "Create a new project."
2. **Select the Project Type**
   - In the search box, type "MVC" and select "ASP.NET Core Web App (Model-View-Controller)" as the project template. Click Next.
3. **Configure Your Project**
   - Enter a project name and location.
   - Ensure that the Framework selected is the latest .NET Core version available.
4. **Project Settings**
   - You can configure additional settings such as authentication methods if required. For a basic setup, you can proceed with the default settings.
5. **Create Project**
   - Click on the "Create" button to generate your new MVC project.

Tools and Libraries for API Interaction in .NET Core

To enhance API interaction, several tools and libraries are available that simplify the processes of sending requests and processing responses.

- **Essential Tools and Libraries**
  - **HttpClient:** A library used for sending HTTP requests and receiving HTTP responses from a resource identified by a URI.
  - **Newtonsoft.Json:** Popularly known as Json.NET, this library helps to serialize and deserialize JSON data.
  - **Postman:** While not a .NET library, Postman is an indispensable tool for testing API requests and responses.

Implementing Asynchronous API Calls in .NET Core MVC

Asynchronous programming is a powerful feature in .NET Core that helps improve the performance and responsiveness of applications, especially those that perform I/O-bound operations like making API calls. The core elements of asynchronous programming in C# are the `async` and `await` keywords, along with the Task-based asynchronous pattern. Understanding these concepts is crucial for developing efficient web applications in .NET Core MVC.

The async Keyword

The `async` keyword is used to define a method as asynchronous. It modifies a method to indicate that the method contains operations that may need to wait for external resources or long-running tasks, without blocking the execution thread. An `async` method always returns a `Task` or `Task<T>` (for methods that return a value) or `void` (generally used for event handlers and is not recommended for most other cases due to the difficulty in error handling).

Key points about async methods

- An `async` method provides a way for the application to remain responsive and efficient by allowing other tasks to run while waiting for asynchronous operations to complete.
- The method itself doesn't perform the asynchronous work. Instead, it provides the framework for handling asynchronous operations using `await`, `Task`, and other related asynchronous constructs.

The await Keyword

The `await` keyword is used within an `async` method to suspend the method's execution until the awaited task completes. `await` can only be used in methods modified by `async`. When an `await` statement is reached, the current method is paused, and control returns

to the caller until the task completes. This feature is essential for managing concurrency in applications, as it allows the program to perform other operations instead of waiting and doing nothing (i.e., blocking).

Benefits of using await

- It simplifies the code by allowing asynchronous code to be written as though it were synchronous, particularly in handling sequential dependencies.
- It helps manage resources efficiently, as the thread handling the `await` can be used to execute other tasks when the method is waiting for the asynchronous operation to complete.

The Task Type

`Task` represents an asynchronous operation. It is a core component of the Task-based Asynchronous Pattern (TAP). A `Task` can tell you if the operation it represents has been completed and if it was completed successfully. `Task<T>` is a subclass of `Task` that can also provide a result value once the asynchronous operation completes.

Using Task in asynchronous methods

- Methods that perform asynchronous operations return a `Task` or `Task<T>`, which represents the ongoing work.
- The returned `Task` allows the caller to wait for the operation to complete and, in the case of `Task<T>`, to retrieve the result of the operation.

Example: Asynchronous Method Using Task

```
public async Task<IActionResult> GetTodoItems()
{
    try
    {
        string jsonResponse = await _httpClient.GetStringAsync("https
   ://api.example.com/todos");
        var todoItems = JsonSerializer.Deserialize<List<TodoItem>>(
   jsonResponse);
        return View(todoItems);
    }
    catch (Exception ex)
    {
        _logger.LogError("Error fetching todo items", ex);
        return View("Error");
    }
}
```

In this example, the `GetTodoItems` method is declared with the `async` keyword, indicating it performs asynchronous operations. The `await` keyword is used to pause the execution of the method until the `GetStringAsync` method of the `HttpClient` completes, without blocking the thread. The method returns a `Task<IActionResult>`, indicating the asynchronous operation will eventually produce an `IActionResult`.

By integrating `async`, `await`, and `Task`, developers can write more readable, maintainable, and performant applications in .NET Core MVC. These tools are essential for modern web development, where efficiency and responsiveness are key.

Handling Errors in Asynchronous Operations

Handling errors in asynchronous code involves catching exceptions that may be thrown during the execution of async tasks. Use try-catch blocks to manage exceptions and ensure that your application can gracefully handle errors and continue operation.

Example of Error Handling in an Asynchronous Method

```
public async Task<IActionResult> UpdateTodoItem(TodoItem item)
{
    try
    {
        var json = JsonSerializer.Serialize(item);
        var content = new StringContent(json, Encoding.UTF8, "
   application/json");
        HttpResponseMessage response = await _httpClient.PutAsync($"
   https://api.example.com/todos/{item.Id}", content);
        response.EnsureSuccessStatusCode();
        return RedirectToAction(nameof(Index));
    }
    catch (Exception e)
    {
        _logger.LogError("Failed to update todo item", e);
        return View("Error", new ErrorViewModel { RequestId = Activity.
   Current?.Id ?? HttpContext.TraceIdentifier, Message = "Failed to
   update item." });
    }
}
```

This example demonstrates the implementation of error handling in an asynchronous operation within a .NET Core MVC application. The `try-catch` block is effectively used to catch any exceptions that occur during the HTTP PUT request operation. If an error occurs, it is logged, and a user-friendly error page is displayed. This method ensures that the application handles errors robustly, maintaining a smooth user experience and minimizing disruptions in service.

Handling API Responses

Interpreting JSON Responses

When interacting with web APIs, it's common to encounter responses formatted in JSON (JavaScript Object Notation). JSON is a universally adopted data interchange format known for its readability and straightforward structure, making it both easy for humans to understand and simple for machines to parse. In the context of .NET Core applications, developers must be adept at interpreting these JSON responses to effectively manage the data returned by APIs.

Example of a JSON response

Here is a typical JSON response you might receive from an API

```
{
    "userId": 1,
    "id": 1,
    "title": "delectus aut autem",
    "completed": false
}
```

This JSON snippet represents an object with four properties: *userId*, *id*, *title*, and *completed*, where *userId* and *id* are integers, *title* is a string, and *completed* is a boolean.

Deserializing JSON in .NET Core

.NET Core provides robust tools for handling JSON. Two primary libraries used for JSON data manipulation are *System.Text.Json*—which is built into .NET Core 3.0 and later—and *Newtonsoft.Json*, a widely used third-party library that offers additional flexibility and features.

To work with JSON data, first, you'll need to map the JSON structure to a .NET class. Here's how you might define a class in C# that corresponds to the JSON data shown above

```
public class TodoItem
{
    public int UserId { get; set; }
    public int Id { get; set; }
    public string Title { get; set; }
    public bool Completed { get; set; }
}
```

In this class, *TodoItem*, each property corresponds to a key in the JSON object. The names and data types of the properties are made to match those of the JSON keys to ensure accurate and seamless data parsing.

Deserialization Process

Deserialization is the process of converting the JSON format back into a .NET object. This can be done using either *System.Text.Json* or *Newtonsoft.Json*.

Using System.Text.Json: This library is integrated into .NET Core and optimized for performance. It provides straightforward mechanisms for serializing and deserializing data. Here's how you might deserialize the JSON response into a *TodoItem* object using *System.Text.Json*

```
var todoItem = JsonSerializer.Deserialize<TodoItem>(jsonString);
```

In this line of code, *jsonString* represents the JSON data as a string. The *Deserialize* method is used to convert this string into an instance of *TodoItem*.

Using Newtonsoft.Json: Also known as Json.NET, this library offers extensive features for handling JSON, including settings to ignore missing members, handle null values, and customize date formats. Here's an example of deserializing using *Newtonsoft.Json*

```
var settings = new JsonSerializerSettings { MissingMemberHandling =
    MissingMemberHandling.Ignore };
var todoItem = JsonConvert.DeserializeObject<TodoItem>(jsonString,
    settings);
```

In this example, *settings* is configured to ignore any JSON properties that do not match properties in the *TodoItem* class. This is particularly useful for working with API responses that may include varying sets of data elements.

Error Handling Strategies

Proper error handling is crucial for building resilient applications, particularly when integrating external APIs. When consuming APIs, several challenges may surface, such as network issues, API downtime, or unexpected response formats. Implementing robust error handling practices can significantly enhance the reliability and user experience of your applications.

Try-Catch Blocks

To safeguard your application against runtime exceptions during API calls, implementing try-catch blocks is essential. This method captures exceptions that occur due to network failures, server errors, or malformed responses. In an MVC application, rather than logging errors directly to the console, it is more appropriate to handle these within the application flow, such as logging to a file or database and providing user-friendly error messages or notifications.

Example of a refined try-catch block in an API request in MVC

```
public async Task<ActionResult> GetTodoItem()
{
    try
    {
        var response = await httpClient.GetAsync("https://api.example.
    com/todos/1");
        // Throws an exception if the HTTP response status is an error
    code.
        response.EnsureSuccessStatusCode();
        var json = await response.Content.ReadAsStringAsync();
        var todoItem = JsonSerializer.Deserialize<TodoItem>(json);
        return View(todoItem);
    }
    catch (HttpRequestException e)
    {
        // Log error and provide feedback to the user
        _logger.LogError($"An error occurred when calling the API: {e.
    Message}");
        return View("Error", new ErrorViewModel { RequestId = Activity.
    Current?.Id ?? HttpContext.TraceIdentifier });
    }
}
```

HTTP Status Codes

Handling HTTP status codes effectively is another critical aspect of robust API consumption. These codes are standardized indicators that describe the outcome of an API request. Checking these codes allows your application to appropriately respond to different scenarios, such as a successful request, an unauthorized access attempt, or a server-side error.

Key HTTP Status Codes to Handle:

- **200 OK:** The request has been successfully processed by the server.
- **401 Unauthorized:** The request lacks valid authentication credentials.
- **404 Not Found:** The requested resource does not exist on the server.
- **500 Internal Server Error:** A generic error occurred on the server which prevents it from fulfilling the request.

Handling these status codes in MVC typically involves conditional checks and tailored responses based on the outcome, as illustrated below

```
var response = await httpClient.GetAsync("https://api.example.com/todos
    /1");
if (response.IsSuccessStatusCode)
{
    var json = await response.Content.ReadAsStringAsync();
    var todoItem = JsonSerializer.Deserialize<TodoItem>(json);
    return View(todoItem);
}
else if (response.StatusCode == System.Net.HttpStatusCode.NotFound)
{
    return View("NotFound");
}
else if (response.StatusCode == System.Net.HttpStatusCode.Unauthorized)
{
    return RedirectToAction("Login", "Account");
}
else
{
    return View("Error");
}
```

Connecting to a Public API

In this section, we'll explore how to connect to the GitHub API. The GitHub API is accessible without authentication for basic requests, such as fetching public user data. This makes it an excellent starting point for learning how to interact with APIs.

Example: Fetching User Data from GitHub

Setting Up Your MVC Project

1. Create a New MVC Project in Visual Studio

- Open Visual Studio and select *File - New - Project*.
- Choose *ASP.NET Core Web App (Model-View-Controller* and click *Next*.
- Name your project and choose a suitable location for it.
- Select *.NET Core* and *ASP.NET Core 8* (or later) from the dropdown, and click *Create*.

2. Install Newtonsoft.Json for JSON Parsing

- Right-click on your project in the Solution Explorer and select *Manage NuGet Packages*.
- Search for *Newtonsoft.Json* and install the latest stable version. This package is essential for deserializing the JSON data returned from the API into .NET objects.

Creating the Artifacts

To display the user data in your `UserProfile.cshtml` view using Razor syntax, you will need to prepare the view to accept a model of type `User` (assuming `User` is a class that reflects the data structure returned by the GitHub API). Here's how you can set up the view and use Razor syntax to render the properties of the `User` object.

Defining the User Model

First, ensure that your `User` model is properly defined, something like this

```
public class User
{
    public string Login { get; set; } // GitHub username
    public string Name { get; set; } // User's full name
    [JsonProperty("avatar_url")]
    public string AvatarUrl { get; set; } // URL of the user's avatar
    public int PublicRepos { get; set; } // Number of public
   repositories
    public string Bio { get; set; } // Biography
}
```

Create the UserProfile View

To properly set up a view for displaying user data in an ASP.NET Core MVC application, follow these steps to create and configure the view file using Razor syntax.

Navigate to the `Views` folder, then find the folder named Home

Add a New View

1. Right-click on the `Home` folder.
2. Select *Add ¿ New Item....*
3. In the dialog that appears, choose *Razor View - Empty*.

4. Name the file `UserProfile.cshtml`.
5. Click *Add* to create the view file.

Update the View

1. Define the Model at the Top of the View

```
@model YourNamespace.Models.User
```

2. Create HTML to Display User Data

```
<div class="user-profile">
    <h1>User Profile: @Model.Name</h1>
    <img src="@Model.AvatarUrl" alt="User Avatar" style="width:150
   px; height:150px;">
    <p><strong>Username:</strong> @Model.Login</p>
    <p><strong>Biography:</strong> @Model.Bio</p>
    <p><strong>Public Repositories:</strong> @Model.PublicRepos</p>
</div>
```

Creating the API Client

- Create a new class called *GitHubApiClient*
  - In the Solution Explorer, right-click on your project, select *Add*, and then *Class...*.
  - Name the class *GitHubApiClient* and click *Add*.
  - Configure HttpClient within the *GitHubApiClient* class

```
using System;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Threading.Tasks;

public class GitHubApiClient
{
    private readonly HttpClient _client;

    public GitHubApiClient()
    {
        _client = new HttpClient {
            BaseAddress = new Uri("https://api.github.com/"),
            DefaultRequestHeaders =
            {
            UserAgent = { ProductInfoHeaderValue.Parse("YourApp")}
            }
        };
    }

    public async Task<string> GetUserProfile(string username)
    {
```

```
        var response = await _client.GetAsync($"users/{username}");
        response.EnsureSuccessStatusCode();
        return await response.Content.ReadAsStringAsync();
    }
}
```

Explanation: The HttpClient is configured with the base URL of the GitHub API. The UserAgent header is added because GitHub requires a user agent header to identify the client making the request.

Updating the Controller

- Update the *HomeController*
- Inject the *GitHubApiClient* and fetch user data
- Create the *UserProfile* action
- Create a POST method for the *Index* action to get the provided username and send it to the *UserProfile* action

```
using Microsoft.AspNetCore.Mvc;
using Newtonsoft.Json;

public class HomeController : Controller
{
    private readonly GitHubApiClient _apiClient;

    public HomeController(GitHubApiClient apiClient)
    {
        _apiClient = apiClient;
    }

    public IActionResult Index()
    {
        return View();
    }

    [HttpPost]
    public IActionResult Index(string username)
    {
        return RedirectToAction("UserProfile", new { username =
   username });
    }

    public async Task<IActionResult> UserProfile(string username)
    {
        var data = await _apiClient.GetUserProfile(username);
        var user = JsonConvert.DeserializeObject<User>(data);
        return View(user);
    }
}
```

Explanation: The *UserProfile* action method calls the *GetUserProfile* method of *GitHubApiClient* to fetch user data from GitHub and deserializes it into a *User* object. The data is then passed to the view.

Update the Index View

The Index view now needs to be updated to provide the user with the ability to enter a user name form the API to find.

```
<h2>Enter GitHub Username</h2>
<form method="post" action="@Url.Action("Index", "Home")">
    <label for="username">GitHub Username:</label>
    <input type="text" id="username" name="username" required />
    <button type="submit">Get Profile</button>
</form>
```

Updating Program.cs for Dependency Injection

In a .NET Core MVC application, particularly in .NET 6 and later where `Program.cs` serves as the central configuration point, you handle dependencies and service configurations in this file. We need to add a few lines to `Program.cs` as part of setting up the application's dependency injection (DI) container and configuring HTTP client services. Here's how each line of code functions and why it's important

Add HttpClient Service

`AddHttpClient()` registers the `IHttpClientFactory` which allows your application to create instances of `HttpClient` with pre-configured settings. The factory is useful for managing the lifetimes of handlers and allows you to centralize the configuration and policies for HTTP requests in your app.

```
builder.Services.AddHttpClient();
```

Register GitHubApiClient

By registering `GitHubApiClient` with `AddTransient`, you're telling the dependency injection (DI) container to create a new instance of `GitHubApiClient` each time it is requested. Transient services are very suitable for lightweight, stateless services.

```
builder.Services.AddTransient<GitHubApiClient>();
```

Test Your Application

Testing your application ensures that all components are correctly integrated and functioning as expected. Here's how you can run your application and verify that it interacts properly with the GitHub API.

Run Your Application

1. Start your application and navigate to the Home index URL (typically `/` or `/Home/Index`).
2. You should see a form asking for a GitHub username.

Enter a Username and Submit

1. Type a valid GitHub username into the form and submit it.
2. The form should redirect to the `UserProfile` action in the `HomeController`, where the user profile data for the entered username will be displayed.

If the application is not functioning as expected, use breakpoints and your debugging skills to find and resolve the issue.

Building a Weather Application Using Live Data

Next we will develop a small web application using .NET Core and MVC that fetches live weather data from the OpenWeatherMap API. This project will demonstrate how to connect to an external API, interpret JSON responses, and display this data in a user-friendly format.

Registering with OpenWeatherMap API

Before beginning this project, ensure you have an API key

- Sign up for a free API key at OpenWeatherMap.org
- Once registered, you will receive an API key. This key is essential for making requests to their API.

Setting Up the Project

Create a New MVC Project

1. Open Visual Studio and select *"Create a new project."*
2. Choose *"ASP.NET Core Web App (Model-View-Controller)."* Name your project *"WeatherApp."*
3. Ensure your project targets .NET Core 8.0 or later. You can set this in the project properties under the *"Target framework"* dropdown.

Add Required Packages

- Open the Package Manager Console (Tools - NuGet Package Manager - Package Manager Console).
- Install Newtonsoft.Json by entering **Install-Package Newtonsoft.Json**.

This initial setup prepares the foundation for developing a web application that can interact with external APIs to fetch and display data, using .NET Core and MVC patterns. The next steps will guide you through integrating the OpenWeatherMap API into your application.

Integrating the OpenWeatherMap API

To integrate the OpenWeatherMap API into your .NET Core MVC application, you will need to securely store and access your API key. Follow these steps to add your API key to the project

Add the API Key

Store your OpenWeatherMap API key in a secure location. For this project, you can place it in the `appsettings.json` file to keep it outside your source code and easily configurable. Here is how you can add it

```
{
  "ApiSettings": {
    "OpenWeatherApiKey": "your_api_key_here"
  }
}
```

Configure the API Key

- Create a new file named ApiSettings.cs.
- Add the following code to manage your API settings securely

```
public class ApiSettings
{
    public string ApiKey { get; set; }
}
```

Register HttpClient and ApiSettings in Dependency Injection Container

- Open the Program.cs file.
- Modify the builder setup to include your ApiSettings configuration

```
// Add services to the container.
builder.Services.AddControllersWithViews();

// Register Http Client
builder.Services.AddHttpClient();

// Register ApiSettings
builder.Services.Configure<ApiSettings>(builder.Configuration.
   GetSection("ApiSettings"));
```

This configuration allows you to maintain the security of your API key while making it accessible within your application code via configuration management tools provided by .NET Core. In the next steps, you will see how to use this key to make API requests to OpenWeatherMap.

Creating the Weather Service

To fetch weather data, you'll create a service class that handles API requests and response parsing. This class will directly interact with OpenWeatherMap to pull data based on user input or predefined criteria.

Create a Service Class and Inject the HTTP Client and Configuration Options

1. In your MVC project, create a new folder named `Services`.
2. Inside this folder, create a class file named `WeatherService.cs`.

```
public class WeatherService
{
    private readonly HttpClient _httpClient;
    private readonly string _apiKey;

    public WeatherService(HttpClient httpClient, IOptions<ApiSettings>
   options)
    {
        _httpClient = httpClient;
        _apiKey = options.Value.OpenWeatherApiKey;
    }

}
```

Implement a Method to Fetch Weather Data Using HttpClient

The `WeatherService` class will use `HttpClient` to make HTTP requests to the OpenWeatherMap API.

```
public async Task<string> GetWeatherAsync(string cityName)
{
    // Construct the URL to access the OpenWeatherMap API
    string url = $"https://api.openweathermap.org/data/2.5/weather?q={
   cityName}&appid={_apiKey}&units=metric";

    try
    {
        // Send a GET request to the specified URL
        HttpResponseMessage response = await _httpClient.GetAsync(url);

        // Ensure the request was successful
        response.EnsureSuccessStatusCode();

        // Read the response as a string
        return await response.Content.ReadAsStringAsync();

    }
    catch (HttpRequestException e)
    {
        // Handle potential network errors
        Console.Error.WriteLine($"Request exception: {e.Message}");
        return null;
    }
    catch (Exception e)
    {
        // Handle other potential errors
        Console.Error.WriteLine($"General exception: {e.Message}");
        return null;
    }
}
```

Register the WeatherService

Register the WeatherService in your Program.cs for dependency injection.

```
// Add services to the container.
builder.Services.AddControllersWithViews();

// Register ApiSettings
builder.Services.Configure<ApiSettings>(builder.Configuration.
    GetSection("ApiSettings"));

// Register WeatherService
builder.Services.AddTransient<WeatherService>();
```

Create the MVC Components

To efficiently present weather data fetched from the OpenWeatherMap API, we need to establish several key MVC components: the model, controller, and view.

Creating the Weather Model

First, define a model to represent the weather data that the API returns. This model will encapsulate the weather data elements that you wish to display.

```
public class WeatherModel
{
    public string City { get; set; }
    public string Temperature { get; set; }
    public string Description { get; set; }
    public string Humidity { get; set; }
    public string WindSpeed { get; set; }
}
```

This model includes basic weather parameters like temperature, weather description, humidity, and wind speed. You can expand it based on the data you plan to use from the API.

Creating the Weather Controller

The controller will use the WeatherService to fetch the weather data and pass it to the view. It acts as the intermediary between the model and the view.

Make sure you installed the Newtonsoft.Json nuget package

```
using Microsoft.AspNetCore.Mvc;
using WeatherApp.Services;
using WeatherApp.Models;
using Newtonsoft.Json;

public class WeatherController : Controller
{
    private readonly WeatherService _weatherService;

    public WeatherController(WeatherService weatherService)
    {
        _weatherService = weatherService;
    }

    public async Task<IActionResult> Index(string city = "New York")
    {
        string json = await _weatherService.GetWeatherAsync(city);
        var weatherData = JsonConvert.DeserializeObject<dynamic>(json)
   ;
        var model = new WeatherModel
        {
            City = city,
            Temperature = $"{weatherData.main.temp}  C ",
            Description = weatherData.weather[0].description,
            Humidity = $"{weatherData.main.humidity} %",
            WindSpeed = $"{weatherData.wind.speed} m/s"
        };

        return View(model);
    } return View(model);
    }
}
```

This controller includes a single action `Index` that takes a city name as an input parameter (defaulting to "New York") and uses the WeatherService to fetch the weather data. It constructs a WeatherModel from this data and passes it to the view.

Understanding the dynamic Keyword

The `dynamic` keyword in C# introduces a way to bypass static type checking. When you declare an object as dynamic, you are essentially telling the compiler to defer type checking to runtime. This allows for writing code that is more flexible and can interact with data structures whose format is not known at compile time.

Usage in JSON Deserialization

When deserializing JSON data, the dynamic type can be particularly useful.

```
var weatherData = JsonConvert.DeserializeObject<dynamic>(json);
```

Here, `JsonConvert.DeserializeObject<dynamic>(json)` uses the dynamic type for the `weatherData` variable. This means that the structure of the JSON does not need to be known beforehand, nor do you need to define a specific class to map the JSON to. The properties of the JSON can be accessed directly on `weatherData` as if they were normal properties, even though they are not statically typed.

```
var temperature = weatherData.main.temp;
var description = weatherData.weather[0].description;
```

Benefits

- **Flexibility:** The primary advantage is flexibility. You can write code that handles data structures that may vary in their format without needing to define numerous different classes.
- **Ease of Use:** It simplifies accessing variable properties directly from JSON without creating a lot of boilerplate classes or using dictionaries. It's particularly useful for quick prototypes or when working with complex or frequently changing data schemas.

Using `dynamic` enables developers to adapt their code more readily to changing data specifications, making it a powerful tool for applications that require high levels of flexibility.

Creating the Weather View

Finally, create a view to display the weather information. This view will render the properties of the WeatherModel in a user-friendly format.

```
@model WeatherApp.Models.WeatherModel

<h2>Weather for @Model.City</h2>
<div>
    <p><strong>Temperature:</strong> @Model.Temperature</p>
    <p><strong>Description:</strong> @Model.Description</p>
    <p><strong>Humidity:</strong> @Model.Humidity</p>
    <p><strong>Wind Speed:</strong> @Model.WindSpeed</p>
</div>
```

This simple view displays the weather data in a straightforward format. Enhancements can be made with CSS for better styling and additional interactive elements if needed.

Test The Application

Run your application and navigate to the Weather controller. You should see weather information from New York.