Overview

We will be calling APIs with C# from our ASP.NET Core applications using a built-in feature called HttpClient. Here are the high-level steps to get started. More details on these steps follow. If you've called APIs with Angular, many of these steps will seem very familiar.

Summary of steps: *(details presented later in this document)*

Initial project setup:

1. Start with an ASP.NET Core project.
2. Create a service class for each API that your application will work with, e.g. DogApiService.
    a. Add your service as a scoped service in Program.cs.
    b. Use dependency injection to get an instance of HttpClient in your service.
    c. Set the BaseAddress and authentication/api-key headers if needed.
3. Use dependency injection in your controller(s) to access the service class(es).

Once the project is set up, follow these steps to use API endpoints:

1. Create model classes for the API's JSON structures.
2. Add methods to the service different API calls.
3. Call the service methods as needed (typically from the controllers).

Project Setup

Create a service class

A **service** is typically a class with a narrow, well-defined purpose. It should do something specific. In our case, we will create a service class to encapsulate the code for calling each API.

Create a folder called **Services**. Then create a C# class with a name ending in "Service". E.g. TodoApiService.

In order to allow the controllers and other classes to access our new service we need to add it to the list of services in Program.cs. This should be added below the comment "Add services to the container". For API services, we use AddHttpClient. This uses HttpClientFactory to manage instances of HttpClient and connection pools for this service. (See Make HTTP requests using IHttpClientFactory in ASP.NET Core | Microsoft Learn (Typed Clients section) for more details.)

```
builder.Services.AddHttpClient<TodoApiService>();
```

The API service needs access to an HttpClient instance, managed by the ASP.NET container. In order to receive an instance via dependency injection, add the following code to the service.. (See more about dependency injection below.)

```
private readonly HttpClient _httpClient;

// The constructor
public TodoApiService(HttpClient httpClient)
{
    _httpClient = httpClient;
    _httpClient.BaseAddress = new Uri("https://jsonplaceholder.typicode.com/");
    // ADD API key headers here if needed
    //_httpClient.DefaultRequestHeaders.Add("x-api-key", "4cbbc4a69fc0d85e9");
}
```

Inject the service into the controller

Using the same technique, set up dependency injection to provide the controller access to the service.

```
[Route("api/[controller]")]
[ApiController]
public class TodoApiController : ControllerBase
{
    private readonly TodoApiService _service;

    public TodoApiController(TodoApiService todoApiService)
    {
        _service = todoApiService;
    }
}
```

Dependency injection with ASP.NET Core

An ASP.NET Core project will have a number of classes that work together. Instances of things like controllers, services, and HttpClient are managed by the ASP.NET container. It automatically creates, configures, and tracks instances of these classes for use (and sometimes reuse) throughout the application.

When one class needs access to one of these managed objects, use dependency injection. Here "dependency" means the object you need and "injection" means the ASP.NET Core container will automatically "inject" it into your constructor. Just provide a constructor parameter, and the container will fill it in.

Using dependency injection involves two parts:

1. Register the service

For the class that will be the dependency, make sure to add it to Program.cs. Add a line below the "Add services to the container" comment. There are three ways to do this.

```
// Add services to the container.
builder.Services.AddHttpClient<TodoApiService>();
builder.Services.AddScoped<MyAppDbContext>();
builder.Services.AddSingleton<MyCacheService>();
```

1. **AddHttpClient**: Use this method when registering an API service that uses HttpClient. This instructs the container to manage appropriate HttpClients and connection pools for your API client.
2. **AddScoped**: This registers a service that will get a new instance created for each incoming HTTP request. This is the best choice for managing Entity Framework DBContext, which is designed to be created and then disposed for each batch of database work.
3. **AddSingleton**: This registers a service that gets a single instance for the entire life of your application (singleton pattern).

2. Use the service

For the class receiving the dependency, create a private variable to hold the instance and add a constructor parameter and set the private variable. The ASP.NET container will automatically fill in the constructor parameter with the matching service when the controller is created.

```
public class TodoApiController : ControllerBase
{
    private readonly TodoApiService _todoApiService;

    public TodoApiController(TodoApiService todoApiService)
    {
        _todoApiService= todoApiService;
    }
}
```

Anything from this service can now be used locally in the controller.

```
Todo[] todos = await _todoApiService.GetAllTodos();
```

Use API endpoints

Create model classes for JSON

We're going to automatically make a model based upon the JSON data. Copy the JSON or XML as a particular endpoint

When receiving data from an API, it is necessary to model that data with C# classes. Visual Studio has a way of creating these automatically using a sample of the JSON or XML data from the API.

1. Create a folder called Models
2. Add in a new class in your models folder, delete the class declaration, leaving just the namespace.
3. Go to Edit -> Paste Special -> Paste JSON as Classes (Or XML if you're using that)
4. This will automatically make a class or classes based upon the JSON.
5. Rename the Rootobject class to something that makes more sense. Rename any of the other classes if they're names could be improved.

Call the API from service methods

Add a method to your service for each of the API calls you want to make.

Consider this example:

```
public async Task<Todo[]> GetTodosByUser(int userId)
{
    Todo[]? todos = await _httpClient.GetFromJsonAsync<Todo[]>($"todos?userId={userId}"
    return todos ?? [];
}
```

There are several things to note:

1. We're using the _httpClient that was provided via dependency injection (see above).
2. The method is asynchronous. This means it has the "async" keyword and it must return a Task. We will call out APIs in an asynchronous manner, meaning that they happen in the background without having to pause our code. This is much easier to do if we're using an async method.
3. We use the "await" keyword when calling the API. This waits for the API call to complete before continuing to the next step.
4. The GetFromJsonAsync method uses the GET method and parses the response as JSON.
5. The URL path is passed as a parameter. Note that this is only the end of the URL. The beginning was provided when we specified _httpClient.BaseAddress in the constructor.

6. To use a path or query param in the endpoint URL, use string interpolation (the dollar sign-quote string literal).
7. The type parameter <Todo[]> of GetFromJsonAsync specifies the model class that we want to use when parsing the JSON response. In this case, we expect an array of Todo objects.
8. GetFromJsonAsync always returns a nullable, so we added ? to the data type of todos.
9. This is optional, but we used the null-coalescing operator to provide a default empty list if todos is for some reason null (`?? []`).

Different HTTP methods

Here are examples for GET, POST, PUT, and DELETE when calling an API. Note that the request body for POST and PUT is provided as a second parameter to the HttpClient method.

In most of these examples, we are using the null-forgiving operator (!) when returning the result. This is because, while the HttpClient methods return nullable types, in practice this won't be null unless something goes really wrong. So for our purposes, we will assume they are not null.

All of these methods throw an exception if the API call does not succeed. The one exception is GetTodo2, which handles the 404 response specifically by returning null.

```
// GET
public async Task<Todo> GetTodo(int id)
{
    Todo? todo = await _httpClient.GetFromJsonAsync<Todo>($"todos/{id}");
    return todo!;
}

// GET (but handle 404 gracefully)
public async Task<Todo?> GetTodo2(int id)
{
    HttpResponseMessage response = await _httpClient
            .GetAsync($"todos/{id}");

    if (response.StatusCode == HttpStatusCode.NotFound)
    {
        return null;
    }

    response.EnsureSuccessStatusCode();
    Todo? todo = await response.Content.ReadFromJsonAsync<Todo>();
    return todo;
}
```

```
// POST
public async Task<Todo> CreateTodo(Todo todo)
{
    HttpResponseMessage response = await _httpClient
            .PostAsJsonAsync("todos", todo);
    response.EnsureSuccessStatusCode();

    Todo? newTodo = await response.Content.ReadFromJsonAsync<Todo>();
    return newTodo!;
}

// PUT
public async Task<Todo> UpdateTodo(Todo todo)
{
    HttpResponseMessage response = await _httpClient
            .PutAsJsonAsync($"todos/{todo.id}", todo);
    response.EnsureSuccessStatusCode();

    Todo? newTodo = await response.Content.ReadFromJsonAsync<Todo>();
    return newTodo!;
}

// DELETE
public async Task DeleteTodo(int id)
{
    HttpResponseMessage response = await _httpClient
            .DeleteAsync($"todos/{id}");
    response.EnsureSuccessStatusCode();
}
```

Slashes?

Make sure the BaseAddress's Uri ends with a slash.

When making the API call, do NOT start with a slash.

```
private readonly HttpClient _httpClient;

// The constructor
public TodoApiService(HttpClient httpClient)
{
```

```
    _httpClient = httpClient;
    _httpClient.BaseAddress = new Uri("https://api.example.com/");
}

public async Task<Todo> GetTodo(int id)
{
    Todo? todo = await _httpClient.GetFromJsonAsync<Todo>($"todos/{id}");
    return todo!;
}
```

Call service methods from controllers

We can call the methods of the API service with the variable we used for dependency injection (`_service` in this example).

Because our service methods are async, we do need to use await and make our controller methods async as well.

1. Add the "async" keyword to the method signature.
2. Make sure the return type includes Task.
3. Add the "await" keyword when calling the service method.

```
[HttpGet]
public async Task<IActionResult> GetTodos()
{
    Todo[] todos = await _service.GetAllTodos();
    return Ok(todos);
}
```

More on async

The API call (GetAsync) is what's known as *asynchronous*. Remember to put the word **await** before the call like so:

```
List<Breed> breeds = await
connection.Content.ReadAsAsync<List<Breed>>();
```

In order to use await, the calling method needs to also be asynchronous. To do so, we put the word async before the return type, and we wrap the return type with Task:

```
public static async Task<List<Breed>> GetBreeds(int count)
```

However, when we do the actual return statement, we don't add a Task around it. Notice our return type is List and that's all we return:

```
List<Breed> breeds = await
connection.Content.ReadAsAsync<List<Breed>>();
return breeds;
```

This might seem like a mismatch but it is indeed correct.

If the async method return type is void, then you just put Task for the return type.

Quick console app

You can also use HttpClient with a simple console app, or any other app.

You need to add the Microsoft.AspNet.WebApi.Client NuGet package. Open the manager window from the menu: Tools > NuGet Package Manager > Manage NuGet Packages for Solution... Choose the Browse tab. Select and install the **Microsoft.AspNet.WebApi.Client** package in NuGet. (You can search for webapi.client).

Also notice also that we added the async keyword before the return type in Main. And notice that the return type is normally void with the Main, so instead we just return Task.

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

```
namespace APIConsole
{
    class Program
    {
        static async Task Main(string[] args)
        {
            HttpClient client = new HttpClient();
            client.BaseAddress = new Uri("https://zoo-animal-
api.herokuapp.com/");
            string response = await
client.GetStringAsync("animals/rand/3");
            Console.WriteLine(response);
        }
    }
}
```

Additional Resources

- Make HTTP requests with the HttpClient - .NET | Microsoft Learn
- Call a Web API From a .NET Client (C#) - ASP.NET 4.x | Microsoft Learn
- Make HTTP requests using IHttpClientFactory in ASP.NET Core | Microsoft Learn
- HttpClient guidelines for .NET - .NET | Microsoft Learn - best practices for HttpClient lifecycle management
- DbContext Lifetime, Configuration, and Initialization - EF Core | Microsoft Learn - This includes how to best use dependency injection with EF Core DbContext.