Tutorial: Make HTTP requests in a .NET console app using C#

Article • 10/29/2022

This tutorial builds an app that issues HTTP requests to a REST service on GitHub. The app reads information in JSON format and converts the JSON into C# objects. Converting from JSON to C# objects is known as *deserialization*.

The tutorial shows how to:

- ✓ Send HTTP requests.
- ✓ Deserialize JSON responses.
- ✓ Configure deserialization with attributes.

If you prefer to follow along with the final sample for this tutorial, you can download it. For download instructions, see Samples and Tutorials.

Prerequisites

- The latest .NET SDK
- Visual Studio Code editor
- The C# DevKit

Create the client app

1. Open a command prompt and create a new directory for your app. Make that the current directory.

2. Enter the following command in a console window:

   .NET CLI

   ```
   dotnet new console --name WebAPIClient
   ```

   This command creates the starter files for a basic "Hello World" app. The project name is "WebAPIClient".

3. Navigate into the "WebAPIClient" directory, and run the app.

   .NET CLI

```
cd WebAPIClient
```

.NET CLI

```
dotnet run
```

dotnet run automatically runs dotnet restore to restore any dependencies that the app needs. It also runs dotnet build if needed. You should see the app output `"Hello, World!"`. In your terminal, press `Ctrl` + `C` to stop the app.

Make HTTP requests

This app calls the GitHub API to get information about the projects under the .NET Foundation umbrella. The endpoint is https://api.github.com/orgs/dotnet/repos. To retrieve information, it makes an HTTP GET request. Browsers also make HTTP GET requests, so you can paste that URL into your browser address bar to see what information you'll be receiving and processing.

Use the HttpClient class to make HTTP requests. HttpClient supports only async methods for its long-running APIs. So the following steps create an async method and call it from the Main method.

1. Open the `Program.cs` file in your project directory and replace its contents with the following:

   C#

   ```
   await ProcessRepositoriesAsync();

   static async Task ProcessRepositoriesAsync(HttpClient client)
   {
   }
   ```

   This code:

   - Replaces the `Console.WriteLine` statement with a call to `ProcessRepositoriesAsync` that uses the `await` keyword.
   - Defines an empty `ProcessRepositoriesAsync` method.

2. In the `Program` class, use an HttpClient to handle requests and responses, by replacing the content with the following C#.

   ```
   ```

```C#
using System.Net.Http.Headers;

using HttpClient client = new();
client.DefaultRequestHeaders.Accept.Clear();
client.DefaultRequestHeaders.Accept.Add(
    new
MediaTypeWithQualityHeaderValue("application/vnd.github.v3+json"));
client.DefaultRequestHeaders.Add("User-Agent", ".NET Foundation
Repository Reporter");

await ProcessRepositoriesAsync(client);

static async Task ProcessRepositoriesAsync(HttpClient client)
{
}
```

This code:

- Sets up HTTP headers for all requests:
  - An Accept header to accept JSON responses
  - A User-Agent header. These headers are checked by the GitHub server code and are necessary to retrieve information from GitHub.

3. In the `ProcessRepositoriesAsync` method, call the GitHub endpoint that returns a list of all repositories under the .NET foundation organization:

```C#
 static async Task ProcessRepositoriesAsync(HttpClient client)
 {
     var json = await client.GetStringAsync(
         "https://api.github.com/orgs/dotnet/repos");

     Console.Write(json);
 }
```

This code:

- Awaits the task returned from calling HttpClient.GetStringAsync(String) method. This method sends an HTTP GET request to the specified URI. The body of the response is returned as a String, which is available when the task completes.
- The response string `json` is printed to the console.

4. Build the app and run it.

.NET CLI

```
dotnet run
```

There is no build warning because the `ProcessRepositoriesAsync` now contains an `await` operator. The output is a long display of JSON text.

Deserialize the JSON Result

The following steps convert the JSON response into C# objects. You use the System.Text.Json.JsonSerializer class to deserialize JSON into objects.

1. Create a file named *Repository.cs* and add the following code:

   C#

   ```
   public record class Repository(string name);
   ```

   The preceding code defines a class to represent the JSON object returned from the GitHub API. You'll use this class to display a list of repository names.

   The JSON for a repository object contains dozens of properties, but only the `name` property will be deserialized. The serializer automatically ignores JSON properties for which there is no match in the target class. This feature makes it easier to create types that work with only a subset of fields in a large JSON packet.

   The C# convention is to capitalize the first letter of property names, but the `name` property here starts with a lowercase letter because that matches exactly what's in the JSON. Later you'll see how to use C# property names that don't match the JSON property names.

2. Use the serializer to convert JSON into C# objects. Replace the call to GetStringAsync(String) in the `ProcessRepositoriesAsync` method with the following lines:

   C#

   ```
   await using Stream stream =
       await
   client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
   var repositories =
       await JsonSerializer.DeserializeAsync<List<Repository>>(stream);
   ```

The updated code replaces GetStringAsync(String) with GetStreamAsync(String). This serializer method uses a stream instead of a string as its source.

The first argument to JsonSerializer.DeserializeAsync<TValue>(Stream, JsonSerializerOptions, CancellationToken) is an `await` expression. `await` expressions can appear almost anywhere in your code, even though up to now, you've only seen them as part of an assignment statement. The other two parameters, `JsonSerializerOptions` and `CancellationToken`, are optional and are omitted in the code snippet.

The `DeserializeAsync` method is *generic*, which means you supply type arguments for what kind of objects should be created from the JSON text. In this example, you're deserializing to a `List<Repository>`, which is another generic object, a System.Collections.Generic.List<T>. The `List<T>` class stores a collection of objects. The type argument declares the type of objects stored in the `List<T>`. The type argument is your `Repository` record, because the JSON text represents a collection of repository objects.

3. Add code to display the name of each repository. Replace the lines that read:

```
C#

Console.Write(json);
```

with the following code:

```
C#

foreach (var repo in repositories ?? Enumerable.Empty<Repository>())
    Console.Write(repo.name);
```

4. The following `using` directives should be present at the top of the file:

```
C#

using System.Net.Http.Headers;
using System.Text.Json;
```

5. Run the app.

```
.NET CLI

dotnet run
```

The output is a list of the names of the repositories that are part of the .NET Foundation.

Configure deserialization

1. In *Repository.cs*, replace the file contents with the following C#.

```
C#

using System.Text.Json.Serialization;

public record class Repository(
    [property: JsonPropertyName("name")] string Name);
```

   This code:

   - Changes the name of the `name` property to `Name`.
   - Adds the JsonPropertyNameAttribute to specify how this property appears in the JSON.

2. In *Program.cs*, update the code to use the new capitalization of the `Name` property:

```
C#

foreach (var repo in repositories)
   Console.Write(repo.Name);
```

3. Run the app.

   The output is the same.

Refactor the code

The `ProcessRepositoriesAsync` method can do the async work and return a collection of the repositories. Change that method to return `Task<List<Repository>>`, and move the code that writes to the console near its caller.

1. Change the signature of `ProcessRepositoriesAsync` to return a task whose result is a list of `Repository` objects:

```
C#

static async Task<List<Repository>> ProcessRepositoriesAsync(HttpClient
```

```
client)
```

2. Return the repositories after processing the JSON response:

```C#
await using Stream stream =
    await
client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
var repositories =
    await JsonSerializer.DeserializeAsync<List<Repository>>(stream);
return repositories ?? new();
```

   The compiler generates the `Task<T>` object for the return value because you've marked this method as `async`.

3. Modify the *Program.cs* file, replacing the call to `ProcessRepositoriesAsync` with the following to capture the results and write each repository name to the console.

```C#
var repositories = await ProcessRepositoriesAsync(client);

foreach (var repo in repositories)
    Console.Write(repo.Name);
```

4. Run the app.

   The output is the same.

Deserialize more properties

The following steps add code to process more of the properties in the received JSON packet. You probably won't want to process every property, but adding a few more demonstrates other features of C#.

1. Replace the contents of `Repository` class, with the following `record` definition:

```C#
using System.Text.Json.Serialization;

public record class Repository(
    [property: JsonPropertyName("name")] string Name,
    [property: JsonPropertyName("description")] string Description,
    [property: JsonPropertyName("html_url")] Uri GitHubHomeUrl,
```

```
    [property: JsonPropertyName("homepage")] Uri Homepage,
    [property: JsonPropertyName("watchers")] int Watchers);
```

The Uri and `int` types have built-in functionality to convert to and from string representation. No extra code is needed to deserialize from JSON string format to those target types. If the JSON packet contains data that doesn't convert to a target type, the serialization action throws an exception.

2. Update the `foreach` loop in the *Program.cs* file to display the property values:

```C#
foreach (var repo in repositories)
{
    Console.WriteLine($"Name: {repo.Name}");
    Console.WriteLine($"Homepage: {repo.Homepage}");
    Console.WriteLine($"GitHub: {repo.GitHubHomeUrl}");
    Console.WriteLine($"Description: {repo.Description}");
    Console.WriteLine($"Watchers: {repo.Watchers:#,0}");
    Console.WriteLine();
}
```

3. Run the app.

   The list now includes the additional properties.

Add a date property

The date of the last push operation is formatted in this fashion in the JSON response:

```JSON
2016-02-08T21:27:00Z
```

This format is for Coordinated Universal Time (UTC), so the result of deserialization is a DateTime value whose Kind property is Utc.

To get a date and time represented in your time zone, you have to write a custom conversion method.

1. In *Repository.cs*, add a property for the UTC representation of the date and time and a readonly `LastPush` property that returns the date converted to local time, the file should look like the following:

```C#
```

```
using System.Text.Json.Serialization;

public record class Repository(
    [property: JsonPropertyName("name")] string Name,
    [property: JsonPropertyName("description")] string Description,
    [property: JsonPropertyName("html_url")] Uri GitHubHomeUrl,
    [property: JsonPropertyName("homepage")] Uri Homepage,
    [property: JsonPropertyName("watchers")] int Watchers,
    [property: JsonPropertyName("pushed_at")] DateTime LastPushUtc)
{
    public DateTime LastPush => LastPushUtc.ToLocalTime();
}
```

The `LastPush` property is defined using an *expression-bodied member* for the `get` accessor. There's no `set` accessor. Omitting the `set` accessor is one way to define a *read-only* property in C#. (Yes, you can create *write-only* properties in C#, but their value is limited.)

2. Add another output statement in *Program.cs*: again:

```
C#

Console.WriteLine($"Last push: {repo.LastPush}");
```

3. The complete app should resemble the following *Program.cs* file:

```
C#

using System.Net.Http.Headers;
using System.Text.Json;

using HttpClient client = new();
client.DefaultRequestHeaders.Accept.Clear();
client.DefaultRequestHeaders.Accept.Add(
    new
MediaTypeWithQualityHeaderValue("application/vnd.github.v3+json"));
client.DefaultRequestHeaders.Add("User-Agent", ".NET Foundation
Repository Reporter");

var repositories = await ProcessRepositoriesAsync(client);

foreach (var repo in repositories)
{
    Console.WriteLine($"Name: {repo.Name}");
    Console.WriteLine($"Homepage: {repo.Homepage}");
    Console.WriteLine($"GitHub: {repo.GitHubHomeUrl}");
    Console.WriteLine($"Description: {repo.Description}");
    Console.WriteLine($"Watchers: {repo.Watchers:#,0}");
    Console.WriteLine($"{repo.LastPush}");
```

```
    Console.WriteLine();
}

static async Task<List<Repository>> ProcessRepositoriesAsync(HttpClient
client)
{
    await using Stream stream =
        await
client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
    var repositories =
        await JsonSerializer.DeserializeAsync<List<Repository>>
(stream);
    return repositories ?? new();
}
```

4. Run the app.

   The output includes the date and time of the last push to each repository.

Next steps

In this tutorial, you created an app that makes web requests and parses the results. Your version of the app should now match the finished sample.

Learn more about how to configure JSON serialization in How to serialize and deserialize (marshal and unmarshal) JSON in .NET.