Tutorial: Generate and consume async streams using C# and .NET

Article • 03/25/2023 • 9 minutes to read

Async streams model a streaming source of data. Data streams often retrieve or generate elements asynchronously. They provide a natural programming model for asynchronous streaming data sources.

In this tutorial, you'll learn how to:

- ✓ Create a data source that generates a sequence of data elements asynchronously.
- ✓ Consume that data source asynchronously.
- ✓ Support cancellation and captured contexts for asynchronous streams.
- ✓ Recognize when the new interface and data source are preferred to earlier synchronous data sequences.

Prerequisites

You'll need to set up your machine to run .NET, including the C# compiler. The C# compiler is available with Visual Studio 2022 or the .NET SDK .

You'll need to create a GitHub access token so that you can access the GitHub GraphQL endpoint. Select the following permissions for your GitHub Access Token:

- repo:status
- public_repo

Save the access token in a safe place so you can use it to gain access to the GitHub API endpoint.

> ⚠ **Warning**
>
> Keep your personal access token secure. Any software with your personal access token could make GitHub API calls using your access rights.

This tutorial assumes you're familiar with C# and .NET, including either Visual Studio or the .NET CLI.

Run the starter application

You can get the code for the starter application used in this tutorial from the dotnet/docs repository in the asynchronous-programming/snippets folder.

The starter application is a console application that uses the GitHub GraphQL interface to retrieve recent issues written in the dotnet/docs repository. Start by looking at the following code for the starter app `Main` method:

```C#
static async Task Main(string[] args)
{
    //Follow these steps to create a GitHub Access Token
    // https://help.github.com/articles/creating-a-personal-access-token-for-the-command-line/#creating-a-token
    //Select the following permissions for your GitHub Access Token:
    // - repo:status
    // - public_repo
    // Replace the 3rd parameter to the following code with your GitHub access token.
    var key = GetEnvVariable("GitHubKey",
    "You must store your GitHub key in the 'GitHubKey' environment variable",
    "");

    var client = new GitHubClient(new Octokit.ProductHeaderValue("IssueQueryDemo"))
    {
        Credentials = new Octokit.Credentials(key)
    };

    var progressReporter = new progressStatus((num) =>
    {
        Console.WriteLine($"Received {num} issues in total");
    });
    CancellationTokenSource cancellationSource = new CancellationTokenSource();

    try
    {
        var results = await RunPagedQueryAsync(client, PagedIssueQuery, "docs",
            cancellationSource.Token, progressReporter);
        foreach(var issue in results)
            Console.WriteLine(issue);
    }
    catch (OperationCanceledException)
    {
        Console.WriteLine("Work has been cancelled");
    }
}
```

You can either set a `GitHubKey` environment variable to your personal access token, or you can replace the last argument in the call to `GetEnvVariable` with your personal access token. Don't put your access code in source code if you'll be sharing the source with others. Never upload access codes to a shared source repository.

After creating the GitHub client, the code in `Main` creates a progress reporting object and a cancellation token. Once those objects are created, `Main` calls `RunPagedQueryAsync` to retrieve the most recent 250 created issues. After that task has finished, the results are displayed.

When you run the starter application, you can make some important observations about how this application runs. You'll see progress reported for each page returned from GitHub. You can observe a noticeable pause before GitHub returns each new page of issues. Finally, the issues are displayed only after all 10 pages have been retrieved from GitHub.

Examine the implementation

The implementation reveals why you observed the behavior discussed in the previous section. Examine the code for `RunPagedQueryAsync`:

```C#
private static async Task<JArray> RunPagedQueryAsync(GitHubClient client,
string queryText, string repoName, CancellationToken cancel, IProgress<int>
progress)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    JArray finalResults = new JArray();
    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new
Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results =
JObject.Parse(response.HttpResponse.Body.ToString()!);

        int totalCount = (int)issues(results)["totalCount"]!;
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"]!;
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)
["startCursor"]!.ToString();
        issuesReturned += issues(results)["nodes"]!.Count();
        finalResults.Merge(issues(results)["nodes"]!);
        progress?.Report(issuesReturned);
        cancel.ThrowIfCancellationRequested();
    }
    return finalResults;

    JObject issues(JObject result) => (JObject)result["data"]!["repository"]!
["issues"]!;
    JObject pageInfo(JObject result) => (JObject)issues(result)["pageInfo"]!;
}
```

Let's concentrate on the paging algorithm and async structure of the preceding code. (You can consult the GitHub GraphQL documentation for details on the GitHub GraphQL API.) The `RunPagedQueryAsync` method enumerates the issues from most recent to oldest. It requests 25 issues per page and examines the `pageInfo` structure of the response to continue with the previous page. That follows GraphQL's standard paging support for multi-page responses. The response includes a `pageInfo` object that includes a `hasPreviousPages` value and a `startCursor` value used to request the previous page. The issues are in the `nodes` array. The `RunPagedQueryAsync` method appends these nodes to an array that contains all the results from all pages.

After retrieving and restoring a page of results, `RunPagedQueryAsync` reports progress and checks for cancellation. If cancellation has been requested, `RunPagedQueryAsync` throws an OperationCanceledException.

There are several elements in this code that can be improved. Most importantly, `RunPagedQueryAsync` must allocate storage for all the issues returned. This sample stops at 250 issues because retrieving all open issues would require much more memory to store all the retrieved issues. The protocols for supporting progress reports and cancellation make the algorithm harder to understand on its first reading. More types and APIs are involved.

You must trace the communications through the CancellationTokenSource and its associated CancellationToken to understand where cancellation is requested and where it's granted.

Async streams provide a better way

Async streams and the associated language support address all those concerns. The code that generates the sequence can now use `yield return` to return elements in a method that was declared with the `async` modifier. You can consume an async stream using an `await foreach` loop just as you consume any sequence using a `foreach` loop.

These new language features depend on three new interfaces added to .NET Standard 2.1 and implemented in .NET Core 3.0:

- System.Collections.Generic.IAsyncEnumerable<T>
- System.Collections.Generic.IAsyncEnumerator<T>
- System.IAsyncDisposable

These three interfaces should be familiar to most C# developers. They behave in a manner similar to their synchronous counterparts:

- System.Collections.Generic.IEnumerable<T>
- System.Collections.Generic.IEnumerator<T>
- System.IDisposable

One type that may be unfamiliar is System.Threading.Tasks.ValueTask. The `ValueTask` struct provides a similar API to the System.Threading.Tasks.Task class. `ValueTask` is used in these interfaces for performance reasons.

Convert to async streams

Next, convert the `RunPagedQueryAsync` method to generate an async stream. First, change the signature of `RunPagedQueryAsync` to return an `IAsyncEnumerable<JToken>`, and remove the cancellation token and progress objects from the parameter list as shown in the following code:

C#

```
private static async IAsyncEnumerable<JToken> RunPagedQueryAsync(GitHubClient
client,
    string queryText, string repoName)
```

The starter code processes each page as the page is retrieved, as shown in the following code:

```C#
finalResults.Merge(issues(results)["nodes"]!);
progress?.Report(issuesReturned);
cancel.ThrowIfCancellationRequested();
```

Replace those three lines with the following code:

```C#
foreach (JObject issue in issues(results)["nodes"]!)
    yield return issue;
```

You can also remove the declaration of `finalResults` earlier in this method and the `return` statement that follows the loop you modified.

You've finished the changes to generate an async stream. The finished method should resemble the following code:

```C#
private static async IAsyncEnumerable<JToken> RunPagedQueryAsync(GitHubClient
client,
    string queryText, string repoName)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new
Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results =
JObject.Parse(response.HttpResponse.Body.ToString()!);

        int totalCount = (int)issues(results)["totalCount"]!;
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"]!;
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)
["startCursor"]!.ToString();
        issuesReturned += issues(results)["nodes"]!.Count();

        foreach (JObject issue in issues(results)["nodes"]!)
            yield return issue;
    }

    JObject issues(JObject result) => (JObject)result["data"]!["repository"]!
["issues"]!;
    JObject pageInfo(JObject result) => (JObject)issues(result)["pageInfo"]!;
}
```

Next, you change the code that consumes the collection to consume the async stream. Find the following code in `Main` that processes the collection of issues:

```C#
var progressReporter = new progressStatus((num) =>
{
    Console.WriteLine($"Received {num} issues in total");
});
CancellationTokenSource cancellationSource = new CancellationTokenSource();

try
{
    var results = await RunPagedQueryAsync(client, PagedIssueQuery, "docs",
        cancellationSource.Token, progressReporter);
    foreach(var issue in results)
        Console.WriteLine(issue);
}
catch (OperationCanceledException)
{
    Console.WriteLine("Work has been cancelled");
}
```

Replace that code with the following `await foreach` loop:

C#

```
int num = 0;
await foreach (var issue in RunPagedQueryAsync(client, PagedIssueQuery,
"docs"))
{
    Console.WriteLine(issue);
    Console.WriteLine($"Received {++num} issues in total");
}
```

The new interface IAsyncEnumerator<T> derives from IAsyncDisposable. That means the preceding loop will asynchronously dispose the stream when the loop finishes. You can imagine the loop looks like the following code:

C#

```
int num = 0;
var enumerator = RunPagedQueryAsync(client, PagedIssueQuery,
"docs").GetEnumeratorAsync();
try
{
    while (await enumerator.MoveNextAsync())
    {
        var issue = enumerator.Current;
        Console.WriteLine(issue);
        Console.WriteLine($"Received {++num} issues in total");
    }
} finally
{
    if (enumerator != null)
        await enumerator.DisposeAsync();
}
```

By default, stream elements are processed in the captured context. If you want to disable capturing of the context, use the TaskAsyncEnumerableExtensions.ConfigureAwait extension method. For more information about synchronization contexts and capturing the current context, see the article on consuming the Task-based asynchronous pattern.

Async streams support cancellation using the same protocol as other `async` methods. You would modify the signature for the async iterator method as follows to support cancellation:

C#

```
private static async IAsyncEnumerable<JToken> RunPagedQueryAsync(GitHubClient
client,
    string queryText, string repoName, [EnumeratorCancellation]
CancellationToken cancellationToken = default)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new
Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results =
JObject.Parse(response.HttpResponse.Body.ToString()!);

        int totalCount = (int)issues(results)["totalCount"]!;
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"]!;
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)
["startCursor"]!.ToString();
        issuesReturned += issues(results)["nodes"]!.Count();

        foreach (JObject issue in issues(results)["nodes"]!)
            yield return issue;
    }

    JObject issues(JObject result) => (JObject)result["data"]!["repository"]!
["issues"]!;
    JObject pageInfo(JObject result) => (JObject)issues(result)["pageInfo"]!;
}
```

The System.Runtime.CompilerServices.EnumeratorCancellationAttribute attribute causes the compiler to generate code for the IAsyncEnumerator<T> that makes the token passed to `GetAsyncEnumerator` visible to the body of the async iterator as that argument. Inside `runQueryAsync`, you could examine the state of the token and cancel further work if requested.

You use another extension method, WithCancellation, to pass the cancellation token to the async stream. You would modify the loop enumerating the issues as follows:

C#

```csharp
private static async Task EnumerateWithCancellation(GitHubClient client)
{
    int num = 0;
    var cancellation = new CancellationTokenSource();
    await foreach (var issue in RunPagedQueryAsync(client, PagedIssueQuery,
"docs")
        .WithCancellation(cancellation.Token))
    {
        Console.WriteLine(issue);
        Console.WriteLine($"Received {++num} issues in total");
    }
}
```

You can get the code for the finished tutorial from the dotnet/docs repository in the asynchronous-programming/snippets folder.

Run the finished application

Run the application again. Contrast its behavior with the behavior of the starter application. The first page of results is enumerated as soon as it's available. There's an observable pause as each new page is requested and retrieved, then the next page's results are quickly enumerated. The `try` / `catch` block isn't needed to handle cancellation: the caller can stop enumerating the collection. Progress is clearly reported because the async stream generates results as each page is downloaded. The status for each issue returned is seamlessly included in the `await foreach` loop. You don't need a callback object to track progress.

You can see improvements in memory use by examining the code. You no longer need to allocate a collection to store all the results before they're enumerated. The caller can determine how to consume the results and if a storage collection is needed.

Run both the starter and finished applications and you can observe the differences between the implementations for yourself. You can delete the GitHub access token you created when you started this tutorial after you've finished. If an attacker gained access to that token, they could access GitHub APIs using your credentials.

In this tutorial, you used async streams to read a individual items from a network API that returns pages of data. Async streams can also read from "never ending streams" like a stock ticker, or sensor device. The call to `MoveNextAsync` returns the next item as soon as it's available.