**IAsyncEnumerable with yield in C# -** [**https://code-maze.com/csharp-async-enumerable-yield/**](https://code-maze.com/csharp-async-enumerable-yield/)

The support for **Async Streams** was one of the most exciting features that came out with .NET Core 3.0 and C# 8. This is possible with the use of IAsyncEnumerable with the yield operator. In this article, we are going to explore how to take advantage of this new feature to improve our code.

To download the source code for this article, you can visit our [**IAsyncEnumerable with yield in C#**](https://github.com/CodeMazeBlog/async-enumerable-yield-csharp) repository.

Well, a lot of ground to cover. So let’s get going.

## Introduction to IAsyncEnumerable<T>

Async Streams or **IAsyncEnumerable<T>** provides a way to iterate over an **IEnumerable**collection asynchronously while using the **yield**operator to return data as it comes in.

For instance, let’s consider a scenario of retrieving pages of data from a database or an API, or listening to data signals from a bunch of IoT sensors. In all these cases, we would want to receive each part of the data as soon as it arrives, rather than waiting for the complete data to be available. At the same time, we wouldn’t want to block the CPU while waiting for the chunks of data. This is where IAsyncEnumerable can help us.

## The Limitations of async IEnumerable

The best way to understand the advantages of IAsyncEnumerable<T> is to take a look at how we would implement a similar functionality before its introduction. Consider the scenario that we mentioned in the previous section where we have an async method that queries a data store or API for some data. Let’s assume that the method returns Task<IEnumerable<T>> and has this signature:

public async Task**<**IEnumerable**<**int**>>** GetAllItems**()**

Typically, in these types of methods, we perform some data access operations asynchronously. But to return the data, we will have to wait for all the data fetching operations to complete. This was a limitation of using enumerable types in async methods.

The problem with this approach will become more evident if we need to make multiple asynchronous calls for obtaining the data. For instance, our database or API could be returning data in pages, or each data point could be arriving asynchronously from various IoT sensors. In all these cases, we can return data only once we fetch the complete data.

### Async IEnumerable Example

Let’s try to simulate such a scenario where we need to wait for multiple data points to come through using a C# console application:

class Program

**{**

static async Task Main**(**string**[]** args**)**

**{**

Console.WriteLine**(**$"{DateTime.Now.ToLongTimeString()}: Start"**)**;

foreach **(**var item in await FetchItems**())**

**{**

Console.WriteLine**(**$"{DateTime.Now.ToLongTimeString()}: {item}"**)**;

**}**

Console.WriteLine**(**$"{DateTime.Now.ToLongTimeString()}: End"**)**;

**}**

static async Task**<**IEnumerable**<**int**>>** FetchItems**()**

**{**

List**<**int**>** Items = new List**<**int**>()**;

for **(**int i = 1; i **<**= 10; i++**)**

**{**

await Task.Delay**(**1000**)**;

Items.Add**(**i**)**;

**}**

return Items;

**}**

**}**

In this example, the FetchItems() method simulates a scenario where we fetch each data point separately, all of which could take some time to arrive. For implementing this, we have first created a list and then added each data point to it as it becomes available. Finally, once the complete data is available, we return the list.

In the Main() method, we await the FetchItems() method, loop through its result and print it. Additionally, we are capturing the timestamp to identify when each of the data points becomes available.

Here we see that once the application starts, it waits for 10 seconds without any data, till the complete data is available. Then, all of a sudden we can see the entire data points returned together.

### The Limitation

This approach is not going to be thread blocking, because it’s still async. But we don’t get the data as soon as it arrives. Instead, we have to wait till the complete data arrives. So even though we are using IEnumerable<T> type here, the behavior is more like a List<T>.

This type of implementation is quite inefficient, especially with larger datasets. Imagine each async call to fetch the data takes a long time and there are a large number of data points. This is going to take a long time to get the data and the caller needs to wait till complete data is available.

So how do we improve this? Let’s explore the various options.

## Introducing yield

What we ideally want is to be able to use the **yield** keyword to return data as we receive it so that the caller can process it immediately. The **yield** keyword in C# performs iteration over a collection and returns each element, one at a time.

Before .NET Core 3.0, we were able to use yield only with synchronous code. With synchronous code, a method that returns IEnumerable<T> can use the yield return statement to return each piece of data to the caller as it is returned from the data source.

To see this in action, let’s modify the above example to use yield return:

class Program

**{**

static void Main**(**string**[]** args**)**

**{**

Console.WriteLine**(**$"{DateTime.Now.ToLongTimeString()}: Start"**)**;

foreach **(**var item in FetchItems**())**

**{**

Console.WriteLine**(**$"{DateTime.Now.ToLongTimeString()}: {item}"**)**;

**}**

Console.WriteLine**(**$"{DateTime.Now.ToLongTimeString()}: End"**)**;

**}**

static IEnumerable**<**int**>** FetchItems**()**

**{**

for **(**int i = 1; i **<**= 10; i++**)**

**{**

Thread.Sleep**(**1000**)**;

yield return i;

**}**

**}**

**}**

Great!

Notice that the FetchItems() method is now synchronous and cannot be awaited.

This time we can see that using yield, we can get each data point as soon as it arrives without having to wait for the complete data. This is good!

However, doing so means the FetchItems() method is now synchronous and even if the data fetch operation is async, it will behave as a blocking call and we don’t get the advantages of fetching data asynchronously. So this is not a good approach as it is not going to scale. Let’s see how we can solve this.

## Solving the Problem Using IAsyncEnumerable with yield

So, the solution is to use yield return with asynchronous methods. But that wasn’t possible until .NET Core 3.0 and C# 8 introducing the IAsyncEnumerable<T>.

IAsyncEnumerable<T> exposes an enumerator that has a MoveNextAsync() method that can be awaited. This means a method that produces this result can make asynchronous calls in between yielding results.

That said, let’s modify the FetchItems() method to return IAsyncEnumerable<int> instead of IEnumerable<int> and yield return to emit data:

static async IAsyncEnumerable**<**int**>** FetchItems**()**

**{**

for **(**int i = 1; i **<**= 10; i++**)**

**{**

await Task.Delay**(**1000**)**;

yield return i;

**}**

**}**

Cool! This method can now yield data asynchronously.

Remember that, to consume this method, we need to make some modifications to the calling code as well so that it can support this implementation.

## Consuming IAsyncEnumerable with await foreach

To consume the IAsyncEnumerable<T> results, we need to use the new await foreach() syntax, which is also a new feature available in C# 8. Let’s see how to do that:

static async Task Main**(**string**[]** args**)**

**{**

Console.WriteLine**(**$"{DateTime.Now.ToLongTimeString()}: Start"**)**;

await foreach **(**var item in FetchItems**())**

**{**

Console.WriteLine**(**$"{DateTime.Now.ToLongTimeString()}: {item}"**)**;

**}**

Console.WriteLine**(**$"{DateTime.Now.ToLongTimeString()}: End"**)**;

**}**

This will await the foreach loop itself rather than awaiting the FetchItems() method call within the foreach loop.

This time again we can see that instead of waiting for 10 seconds to get full data and then returning all data at once, we get each data point as it arrives. On top of this, we get the advantage that this call is not a blocking one and has the advantages of asynchronous execution.

Cool!

So far we have learned how to use IAsyncEnumerable<T> with yield in a C# console application. Next, we’ll see how we can use this in an ASP.NET Core application.

## IAsyncEnumerable<T> in ASP.NET Core

Now let’s explore how we can use async streams in ASP.NET Core applications. ASP.Net Core has added the support for IAsyncEnumerable<T> starting from version 3.0. For instance, we can use it to return data from an API controller’s action. When we do that, we can return a streaming result set of a method directly from the controller. This is great because using this we can now effectively stream data from the database to the HTTP response by implementing something like this:

**[**HttpGet**]**

public IAsyncEnumerable**<**Product**>** Get**()** =**>** productsService.GetAllProducts**()**;

But ASP.NET Core 3.0 and later used to buffer the result of a controller action before providing it to the serializer. So while invoking the API, we will not notice any difference as we would still get the complete buffered results at once.

However, with ASP.NET Core 6.0 things have completely changed. In ASP.NET Core 6, while formatting the endpoint result using the System.Text.Json, it no longer buffers the IAsyncEnumerable instances. Instead, it relies on the support that System.Text.Json has been added for the async stream types.

PS: ASP.NET Core 6 is still in preview as of the date of writing this article and Microsoft is planning to release it by November 2021. But It is possible to test the new behavior using a preview release.

### IAsyncEnumerable in ASP.NET Core Example

Let’s implement an API controller that returns IAsyncEnumerable:

**[**Route**(**"api/[controller]"**)]**

**[**ApiController**]**

public class ValuesController : ControllerBase

**{**

**[**HttpGet**]**

public IAsyncEnumerable**<**int**>** Get**()**

**{**

return FetchItems**()**;

**}**

static async IAsyncEnumerable**<**int**>** FetchItems**()**

**{**

for **(**int i = 1; i **<**= 10; i++**)**

**{**

await Task.Delay**(**1000**)**;

yield return i;

**}**

**}**

**}**

In this example, we are using the same FetchItems() method that we implemented earlier which returns an async stream of data. From the API endpoint, we just return the async stream as such. Also, notice that the Get action method returns IAsyncEnumerable type.

## IAsyncEnumerable<T> with Databases

At this point, we might be wondering if we can make use of the IAsyncEnumerable while fetching data from a database.

### EF Core Example

Let’s look at an example of fetching items from a SQL Server database using EF Core:

**[**Route**(**"api/[controller]"**)]**

**[**ApiController**]**

public class ItemsController : ControllerBase

**{**

private readonly ItemContext \_context;

public ItemsController**(**ItemContext context**)**

**{**

\_context = context ?? throw new ArgumentNullException**(**nameof**(**context**))**;

**}**

**[**HttpGet**]**

public async Task**<**IEnumerable**<**Item**>>** GetItemsAsync**()**

**{**

return await FetchItems**()**;

**}**

async Task**<**IEnumerable**<**Item**>>** FetchItems**()**

**{**

return await \_context.Items.ToListAsync**()**;

**}**

**}**

Can we modify this code to use IAsyncEnumerable?

Well, for making use of IAsyncEnumerable, we must be using a yield return statement to fetch each part of data and return it. However, while fetching data from a database using EF Core, we usually get the entire list at once and not in parts. So the best way here is to return this as a plain old Task<IEnumerable<T>> type.

### CosmosDB Example

That said, IAsyncEnumerable can be a good choice while dealing with a database that supports returning data in parts and if we want to stream each part of the data as we receive it. For example, [**Azure Cosmos DB**](https://code-maze.com/azure-cosmos-db-with-asp-net-core-web-api/) supports this behavior while we are fetching multiple records from the database using the QueryIterator and ReadNextAsync() method.

We have explained how to work with Azure Cosmos DB in the [**Building APIs that Talks with Azure Cosmos DB**](https://code-maze.com/azure-cosmos-db-with-asp-net-core-web-api/#building) section of the [**Azure Cosmos DB with ASP.NET Core Web API**](https://code-maze.com/azure-cosmos-db-with-asp-net-core-web-api/#building) article. In the GetMultipleAsync() method inside the CosmosDbService class of this example, we are getting data in parts using the ReadNextAsync() method of the QueryIterator and we add each part of data to a List and finally return it. Replacing the List with IAsyncEnumerable<T> can be a good choice in these types of scenarios as it provides us an option to return part of data as we receive it rather than waiting for the complete data to arrive.

But other than that, with usual data fetching operations, especially with EF Core, it is best to use a Task object and not implement the IAsyncEnumerable.