1. **Async streams** - <https://www.dotnetcurry.com/csharp/async-streams>

Async streams were one of the many new language features introduced in C# 8. If you want to refresh your memory, you can read about them in my DNC Magazine article [New C# 8 Features in Visual Studio 2019](https://www.dotnetcurry.com/csharp/1489/csharp-8-visual-studio-2019).

When compared to Async Streams, there are many other features in C# 8 that are more frequently used (like improved pattern matching, for example) or have a much higher impact on the language (nullable reference types).

This might be the reason why **async streams have attracted less attention**.

However, there are use cases in which **async streams can make your code simpler**and easier to follow by allowing you to do things that were not (easily) possible before.

For example, whenever you are dealing with a collection of items that can be slow to access or generate, you can use async streams to consume them asynchronously one-by-one as they become available.

## The benefits of async streams

In .NET, IEnumerable<T> is the base interface for representing collections of items.

An IEnumerable<T> could be accessed asynchronously even before C# v8 by changing the signature of the method from:

|  |
| --- |
| IEnumerable<string> GetItems(); |

to:

|  |
| --- |
| Task<IEnumerable<string>> GetItems(); |

Such a method would then be consumed with following code:

|  |
| --- |
| foreach (var item in await GetItems())  {      // use the item  } |

Although the code above is asynchronous, there are limitations to it.

For this to work, all of the items have to be *ready* before the iteration can start because only the GetItems() method result is retrieved *asynchronously*, while the iteration through it happens *synchronously*. This is still beneficial in scenarios where all the data can be retrieved in one batch, e.g., with a single API call.

But as soon as that API implements pagination to restrict the maximum number of items that can be retrieved in one call, the asynchronous method above will start to have a negative impact on the performance because the iteration will still start only *after* all the data is retrieved.

Even the data from the first API call will only be accessible after all the API calls are made.

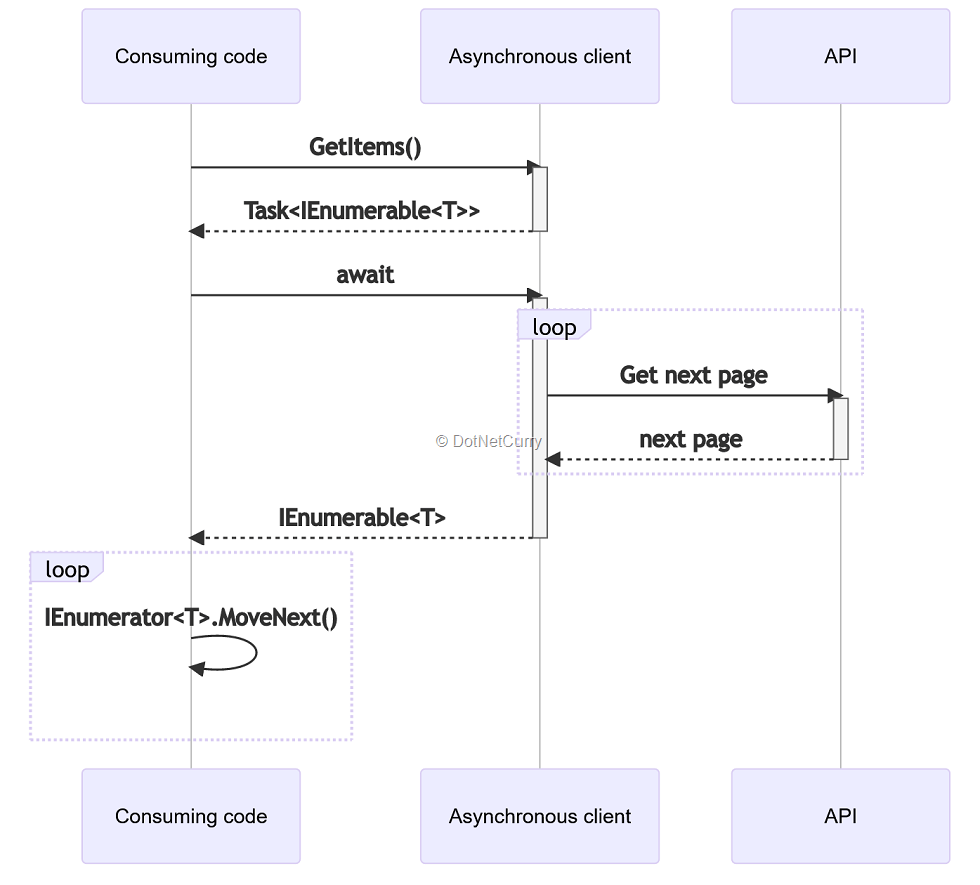


Figure 1: With Task<IEnumerable<T>>, all items are loaded before the iteration starts

Alternatively, one could access individual items asynchronous by using the following method signature:

|  |
| --- |
| IEnumerable<Task<string>> GetItems(); |

This would require the following code to consume:

|  |
| --- |
| foreach (var task in GetItems())  {      var item = await task;      // use the item  } |

Although it might not seem so at first glance, this makes it even more difficult to wrap paginated calls than the previous approach.

With access to every item being asynchronous, the consuming code could decide to access the items out of order or even skip some of the items. This works great when each item is individually time-consuming to access, i.e., when it requires separate I/O operation or processing. But when the items are retrieved in batches, such an approach will get even more difficult to implement (either by retrieving the full page whenever an item in it is requested or by caching items in the page that haven’t been requested yet).

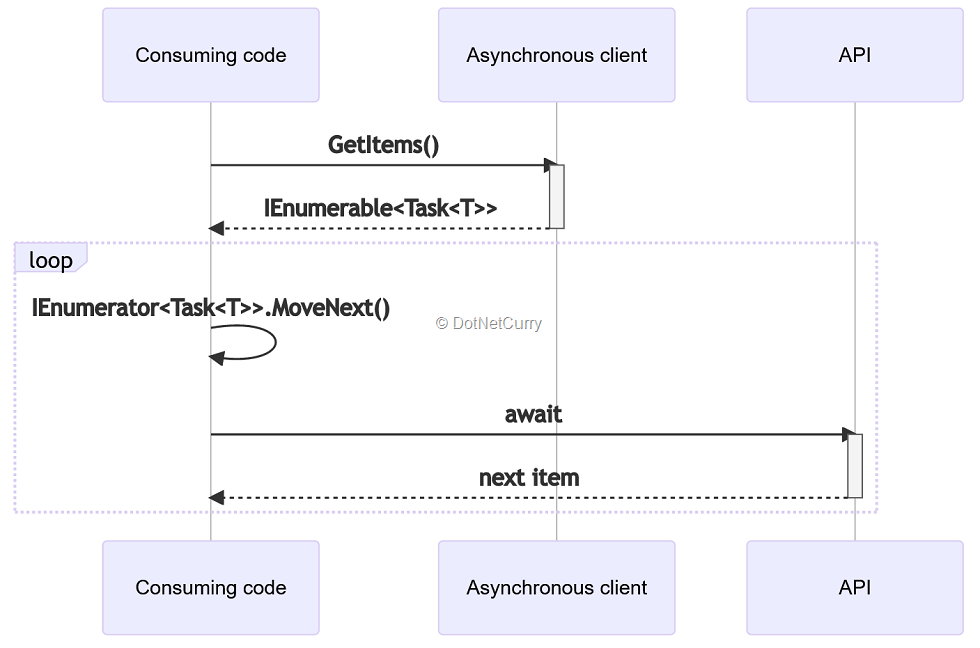


Figure 2: With IEnumerable<Task<T>>, each item is loaded separately

The async streams feature in C# 8 adds a third approach to iterating items in a sequence asynchronously:

|  |
| --- |
| IAsyncEnumerable<string> GetItems(); |

To consume such a collection, an asynchronous version of the foreach statement was added to C# 8:

|  |
| --- |
| await foreach (var value in GetItems())  {      // use the item  } |

In this case, each individual item is still accessed asynchronously so there is no need for the providing method to retrieve all the data in advance. The consumer is still forced to perform an asynchronous access to the items in order so that the items can be efficiently retrieved in batches. For example, the API request for the next batch (i.e., page) will be made when all items from the previous batch have been enumerated and the first item from the next batch is requested.

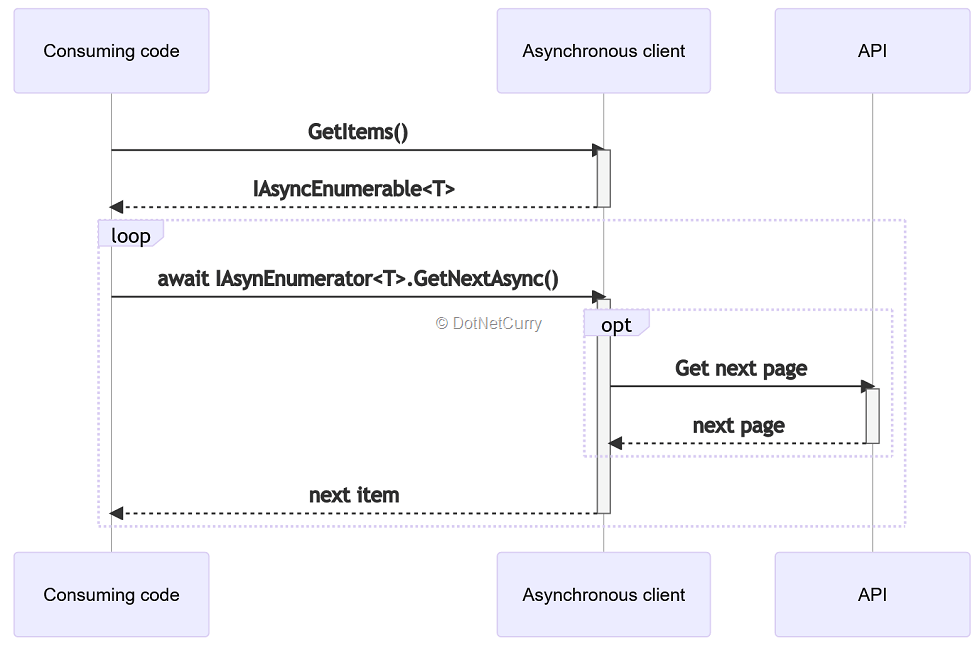


Figure 3: With IAsyncEnumerable<T>, items can be loaded in batches as needed

As a bonus, installing the [System.Linq.Async NuGet package](https://www.nuget.org/packages/System.Linq.Async/) will give you access to LINQ extension methods for the new IASyncEnumerable<T> interface, allowing you to write code like the following:

|  |
| --- |
| var singleItem = await GetItems().Skip(5).FirstAsync(); |

The extension methods match the ones we are used to from the IEnumerable<T> and IQueryable<T> interfaces. The only big difference is that some methods have the Async suffix (like the FirstAsync method in the snippet above). These need to be awaited because they return types other than IAsyncEnumerable<T>, i.e., they return actual data instead of transforming the enumeration.

### Examples of async streams in .NET

Of course, the new interface only makes sense if there are libraries taking advantage of it.

#### **Entity Framework Core**

The most obvious library to benefit from async streams is **Entity Framework Core**. Records being read from a database are a great use case for a collection of items to access asynchronously.

With the AsAsyncEnumerable() extension method, any instance of IQueryable<T> can be converted to IAsyncEnumerable<T> and then accessed using the asynchronous foreach statement:

|  |
| --- |
| var persons = context.Persons      .Where(person => person.LastName == "Doe")      .AsAsyncEnumerable();    await foreach (var person in persons)  {      // use the person  } |

Make sure, though, that you only use this extension method at the end of your queries because any LINQ queries following the AsAsyncEnumerable() method call will not be included in the generated SQL query, making your code inefficient.

The following code snippet is something you do **not** want to do:

|  |
| --- |
| var persons = context.Persons.AsAsyncEnumerable()      .Where(person => person.LastName == "Doe"); |

In this case, the generated SQL query would not include the WHERE clause, causing all the records from the Persons table to be read from the database only to be then filtered in the application code:

|  |
| --- |
| SELECT [p].[Id], [p].[FirstName], [p].[LastName]  FROM [Persons] AS [p] |

The **correct** code would place the invocation of Where() before the invocation of

|  |
| --- |
| AsAsyncEnumerable():  var persons = context.Persons      .Where(person => person.LastName == "Doe")      .AsAsyncEnumerable(); |

This will generate the SQL query with the WHERE clause as you would expect:

|  |
| --- |
| SELECT [p].[Id], [p].[FirstName], [p].[LastName]  FROM [Persons] AS [p]  WHERE [p].[LastName] = N'Doe' |

There is one more thing worth pointing out related to using async streams with Entity Framework Core. If you install the System.Linq.Async NuGet package in your project, the following code will not compile anymore:

|  |
| --- |
| var persons = context.Persons      .Where(person => person.LastName == "Doe")      .AsAsyncEnumerable(); |

The following error will be emitted instead:

The call is ambiguous between the following methods or properties: ‘System.Linq.Queryable.Where<TSource>(System.Linq.IQueryable<TSource>, System.Linq.Expressions.Expression<System.Func<TSource, bool>>)’ and ‘System.Linq.AsyncEnumerable.Where<TSource>(System.Collections.Generic.IAsyncEnumerable<TSource>, System.Func<TSource, bool>)’

The ambiguity is caused by the fact that DbSet<T> implements both IQueryable<T> and IAsyncEnumerable<T>. Since the LINQ extensions for both interfaces are in the System.Linq namespace, the compiler cannot determine which one we want to use. To resolve the ambiguity, we must explicitly specify the interface we want to use:

|  |
| --- |
| var persons = context.Persons.AsQueryable()      .Where(person => person.LastName == "Doe")      .AsAsyncEnumerable(); |

This not only makes the code unambiguous to the compiler but also more clearly states our intentions to human readers.

#### **ASP.NET Core web API**

Support for the IAsyncEnumerable<T> interface in ASP.NET Core web API brings an even larger potential benefit. What I mean is the ability to return objects implementing the IAsyncEnumerable<T> interface from action methods:

|  |
| --- |
| public IAsyncEnumerable<Person> GetAllPersons()  {      return context.Persons.AsAsyncEnumerable();  } |

This will allow the items in the sequence to be read asynchronously without blocking the thread before they are passed to the JSON serializer preparing the actual HTTP response body.

### Implementing an async stream in C#

As we have seen, iterating through an IAsyncEnumerable<T> is very similar to iterating through its synchronous counterpart IEnumerable<T>. We just use the asynchronous version of the foreach statement instead of the synchronous one.

This high level of similarity is also present when creating an async stream instead of its synchronous counterpart. If you need a refresher on how to create an IEnumerable<T>, you can read my DNC Magazine article [Implement a method returning an IEnumerable (Iterators in C#)](https://www.dotnetcurry.com/csharp/1516/return-ienumerable-using-iterators).

In both cases (for IAsyncEnumerable<T> and IEnumerable<T>) you can use the yield return statement to make your job easier. The difference is that for IAsyncEnumerable<T> your method will return IAsyncEnumerable<T> instead of IEnumerable<T> and that your method will be asynchronous instead of synchronous so that you will be able to call other asynchronous methods inside of it using the await keyword.

The following example calls the GitHub REST API to retrieve issues for a repository:

|  |
| --- |
| public async IAsyncEnumerable<GitHubIssue> GetIssues(      string owner,      string repo,      [EnumeratorCancellation] CancellationToken cancellationToken = default)  {      var url = $"https://api.github.com/repos/{owner}/{repo}/issues";      var page = 1;        while (page <= MaxPages && url != null)      {          var response = await httpClient.GetAsync(url, cancellationToken);          var json = await response.Content.ReadAsStringAsync();          var issues = JsonSerializer.Deserialize<List<GitHubIssue>>(              json,              jsonSerializerOptions);            foreach (var issue in issues)          {              yield return issue;          }            url = GetNextPageUrl(response);          page++;          cancellationToken.ThrowIfCancellationRequested();      }  } |

The GitHub API implements pagination and by default only returns 30 issues per request. To hide that from the consumer, the method calls the API to retrieve the next page of results when it runs out of issues on the current page. If the consuming code stops the iteration before that happens, the method will never send that next request.

You have probably noticed the CancellationToken method parameter preceded by the EnumeratorCancellationAttribute. Thanks to this attribute, the cancellation token does not need to be passed directly as a method parameter. The standard WithCancellation() extension method can be used instead:

|  |
| --- |
| var cancellationTokenSource = new CancellationTokenSource();  await foreach (var issue in client.GetIssues("dotnet", "runtime")      .WithCancellation(cancellationTokenSource.Token))  {      // process the issue      // call cancellationTokenSource.Cancel() to cancel the iteration  } |

Passing a CancellationToken when iterating through an IAsyncEnumerable<T> gives you even more flexibility in prematurely stopping the iteration than by simply exiting the foreach loop using a break statement. For example, in the GetIssues() method I pass the CancellationToken to the HttpClient call. This means that if cancellation is requested during that call, it can be interrupted before it completes.

### C# Async Streams vs Observables

If you have ever used [Reactive Extensions for .NET](https://github.com/dotnet/reactive), you might think that async streams are similar to observables.

This is true to some extent, but there are also important differences between the IAsyncEnumerable<T> interface and the IObservable<T> interface:

* The IAsyncEnumerable<T> interface is **pull-based**, i.e., the next value will be retrieved or created when it is requested by the consumer either implicitly by going into the next iteration of the asynchronous foreach loop or explicitly by calling the MoveNextAsync() method on the [IAsyncEnumerator<T> instance](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.iasyncenumerator-1?WT.mc_id=DOP-MVP-4020672).
* The IObservable<T> interface is **push-based**. Once the consumer calls its Subscribe() method, the provided handler will be called with new values without any explicit action by the consumer.

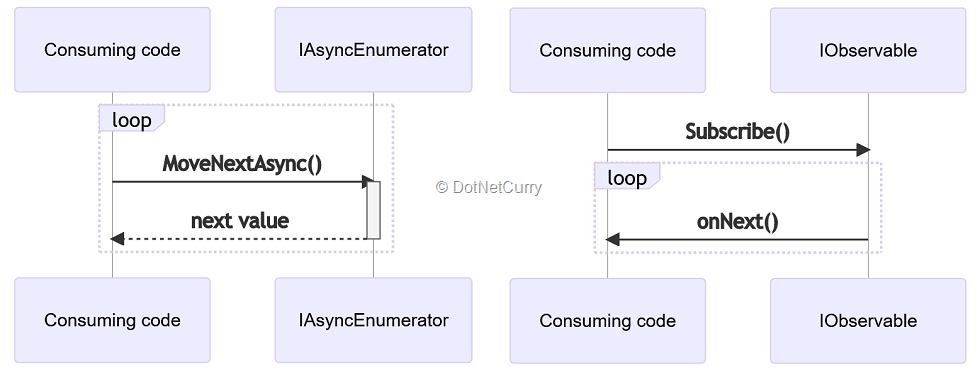


Figure 4: Differences between IAsyncEnumerable<T> and IObservable<T>

This makes observables more like events, than collections. A typical use case for observables would be to create a wrapper around an event:

|  |
| --- |
| Observable.FromEventPattern<TextChangedEventHandler, TextChangedEventArgs>(      handler => this.input.TextChanged += handler,      handler => this.input.TextChanged -= handler)      .Throttle(TimeSpan.FromMilliseconds(500))      .Subscribe(\_ => this.PerformSearch()); |

The sample code above throttles events triggered by an input field as the user is typing into it. Instead of performing a new search with the current value of the input field whenever it changes, it waits until there is no new event for half a second and performs a search then. This reduces the total number of searches but still performs a search when the user stops typing for a while.

The important part is that the events are generated without any action from the consuming code and that the PerformSearch() method is being called based only on the events being emitted as the user types into the input field.

Still, nothing is preventing us from implementing a wrapper for the GitHub REST API that returns an IObservable<T> instead of an IAsyncEnumerable<T>:

|  |
| --- |
| public IObservable<GitHubIssue> GetIssues(string owner, string repo)  {      return Observable.Create<GitHubIssue>(async (observer, cancellationToken) =>      {          var url = $"https://api.github.com/repos/{owner}/{repo}/issues";          var page = 1;            while (page <= MaxPages && url != null)          {              var response = await httpClient.GetAsync(url, cancellationToken);              var json = await response.Content.ReadAsStringAsync();              var issues = JsonSerializer.Deserialize<List<GitHubIssue>>(                  json,                  jsonSerializerOptions);                foreach (var issue in issues)              {                  observer.OnNext(issue);              }                url = GetNextPageUrl(response);              page++;              cancellationToken.ThrowIfCancellationRequested();          }      });  } |

If you compare this code to the one returning an IAsyncEnumerable<T> in the previous section, you will find a lot of similarities. The differences do not affect how the method is implemented:

* This method is not asynchronous. It synchronously calls Observable.Create and passes an asynchronous lambda to it instead.
* There are no yield return statements. Instead, observer.OnNext is called to return the next value.

The more important differences are in how this method is consumed:

|  |
| --- |
| var subscription = client.GetIssues("dotnet", "runtime").Subscribe(      value =>      {          // process the issue      }); |

Once the consumer is subscribed, it will receive new values at the pace of the producer, i.e., the GetIssues() method.

There is also no apparent way to stop receiving the values. However, this can still be done. The Subscribe() method returns an IDisposable which I stored in the subscription variable in the code above. After calling Dispose() on that variable, the handler passed to the Subscribe() method will not be called anymore:

|  |
| --- |
| subscription.Dispose(); |

The previously mentioned System.Linq.Async NuGet package includes extension methods for converting between the two interfaces: IAsyncEnumerable<T> and IObservable<T>:

|  |
| --- |
| public static IAsyncEnumerable<TSource> ToAsyncEnumerable<TSource>(      this IObservable<TSource> source);  public static IObservable<TSource> ToObservable<TSource>(      this IAsyncEnumerable<TSource> source); |

The two methods allow you to consume either interface the way it is more convenient to you by converting it beforehand if necessary. But although these methods are available, you should keep in mind the core design difference between the two interfaces:

* IAsyncEnumerable<T> is pull-based, and
* IObservable<T> is push-based.

When deciding to return one or the other in your code, this should be the basis for your final decision.