Asynchronous and Reactive Programming with async/await in C#

Adrian Vinojcic
Johannes Kepler University
Linz, Austria
adrian.vinojcic@gmail.com

Lukas Aichhorn
Johannes Kepler University
Linz, Austria
lukas.aichhorn07@gmail.com

***Abstract*—Writing responsive web services is a difficult problem, which many developers stumble upon. Many solutions for this scenario already exist, but this report specifies on the async/await feature in C#. We built a web application with different kinds of agents to explore concurrency in C#. To our satisfaction Microsoft provides an excellent feature to deal with asynchronous programming, which can be swiftly learned if one is already familiar with the language.**

***Index Terms*—async, await, c#, csharp, reactive, programming**

I. Introduction

This report deals with the topic of *asynchronous* and *reactive* programming in *C#*. First we have to resolve what that actually means. Asynchronous programming happens when the programmer writes the program in a non-blocking fashion. Reactive programming is a part of asynchronous programming and is achieved with data-streams where everything gets updated as a reaction to an event. C# allows to use this paradigm in the form of the *async/await* keywords in combination with Tasks. This report will explain how to use this feature of the language, show some methods which can be used in combination with *Tasks* and conclude with a case study which we made to try out the concept.

A. Excursion LINQ

But before getting into the Details we are going to take a short look at one of C#'s most important language features - LINQ. LINQ is an abbreviation for "Language Integrated Query" and as this name implies it is a built-in set of Query Instructions which can be used to get specific Data from a source.

```
IEnumerable<string> result =
  from s in list
  where s.Contains("async")
  select s;
```

Example 1. A LINQ

Example 1 shows the simplest structure of a LINQ. In this case the LINQ itself is reminiscent of a SQL-Query with its' usage of `from`, `where` and `select`. The functionality of the code snippet is the following: Take all the items stored in the List (this could be any Collection) and check them if they contain "async". If they do then store them in an `Enumerable`.

```
IEnumerable<string> result = list
  .Where(s => s.Contains("async"))
  .Select(s => s);
```

Example 2. A LINQ with a lambda

In Example 2 we can see the combination of LINQs and lambdas. This makes using LINQ a lot faster and easier to use especially when multiple different queries have to be performed at once on the dataset.

II. C# Asynchronous Programming Techniques

A. The basics of async/await

Asynchronous programming is all about concurrency. In order to achieve this form of execution C# introduced their `async` and `await` keywords as an easy way to implement these concepts.

The `async` keyword is a method modifier which marks the method which it is used on as an asynchronous method. This enables the programmer to use the `await` keyword which is the other essential part for making the code concurrent. Furthermore an `async` method must return either `void`, `Task` or `Task<T>`. It is however heavily discouraged to use void and suggested to use the `Task` return type instead.

The `await` keyword on the other hand is a signal for the program that at this point in the code there is an operation that is going to take some time to execute which can also be inferred from the expanded meaning of the keyword - asynchronous wait. Since before the first `await` keyword nothing has to be awaited, the method executes synchronously up to that point.

Figure 1 on the next page shows a typical execution of an `async` method. As already shortly mentioned above the code runs synchronously up until the await call. The call is now executed but the program does not wait until the call returns and is finished but instead immediately continues with the rest of the program with a `Task` which is similar to Futures or Promises in other languages. When the `await` call is then finished the `Task` receives the result of the call and the method can finish without having had to wait for the lengthy operation.

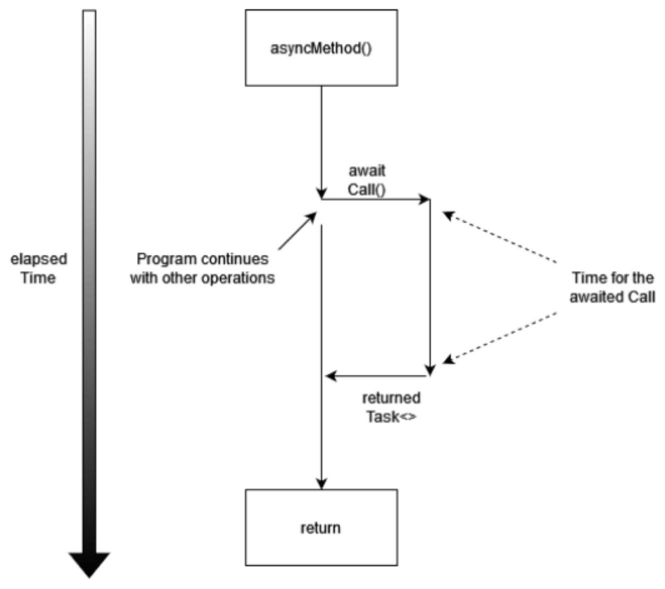

asyncMethod()
await Call()
elapsed Time
Program continues with other operations
Time for the awaited Call
returned Task<>
return

Fig. 1. Illustration of async/awaits' basic function

B. The Task<T> Type in C#

a) The basics of the Tasks type: As previously mentioned the `Task` type is very similar to Promises and Futures of other programming languages because their purpose is to be a sort of placeholder for a value until the program has access to the real result of a lengthy operation. This is what makes the process of awaiting possible.

```
async Task<int> Mul(int x, int y) {
  await Task.Delay(1);
  return a * b;
}

async Task PrintMulAsync() {
  Console.WriteLine(await Mul(1, 2));
}
```

Example 3. Multiplication Task

As shown in example 3 Tasks can also have a generic type parameter. This is used to define the return-type of the `Task`. Meaning that if a `Task` has the generic parameter `int` it will always produce an integer if no exceptions are involved. The generic can however also be omitted and in that case it is similar to a method with a return type `void`.

b) Exception Handling: `Exception` handling can be very tedious when trying to use concurrency in different frameworks. Not so in this case, because Microsoft has made this feature feel very natural when programming. In order to achieve this they store thrown exceptions in the `Task`. Example 4 shows a division, where no check is made for b to not be zero. This `ArithmeticException` is then stored in the returned `Task` if it occurs.

```
async Task<int> Div(int a, int b) {
  await Task.Delay(3);
  return a / b;
}
```

Example 4. A potential Exception

When a `Task` is awaited, its exception can be caught and handled just like usual exception would be as we can see in example 5. It is important to use the `await` keyword here, because otherwise the exception will not actually be thrown.

```
try {
  await Div(1, 0);
} catch (ArithmeticException e) {
  Console.WriteLine(e.Message);
}
```

Example 5. Catching the Exception

In order to produce an `Exception` in a `Task`, one has to use the `Task.FromException<>()` method. In the following example 6 a custom exception is thrown if b is equal to zero. As we see it is important to match the generic type of the `Task` to the one of the method.

```
async Task<int> Div(int a, int b) {
  await Task.Delay(3);
  if(b == 0) {
    return await Task.FromException<int>(new
        CustomException());
  }
  return a / b;
}
```

Example 6. A Task from an Exception

c) Task methods: In this section we are going to talk about the special methods which can be used in conjunction with the Tasks which can be used in variety of situations in order to make dealing with Tasks a lot more manageable.

But first we are going to introduce some small auxiliary methods which are going to be used to demonstrate the capabilities of the different methods.

```
private async Task<int> WaitAndReturn(int x) {
  await Task.Delay(x);
  return x;
}
```

Example 7. Auxiliary Function

As the name in ex. 7 suggests WaitAndReturn takes an integer which is the time the method waits and then returns. The reason for the wait is to simulate a lengthy operation to help visualize the benefits of using asynchronous programming.

```
private async Task<int> Add(int x, int y) {
  await Task.Delay(5);
  return x + y;
}
```

Example 8. Another auxiliary function

The Add method in 8 is just a simple method which adds two integers and has a delay of five seconds to again simulate heavy computation.

- Example 9, `Task.ContinueWith(...)`:
  Used when Tasks are dependent on one another.

```
Task<int> threeTask = WaitAndReturn(3);

Task<int> continued = await threeTask
   .ContinueWith(async t => await
       Add((await t), 5));
```

Example 9. ContinueWith

- Example 10, `Task.WhenAny(...)`:
  Is used on collection of Tasks to get the Result of the Tasks that finished first which is especially interesting when waiting for the same result from multiple sources.

```
var tasks = new HashSet<Task<int>>();
for(var i = 0; i < 5; i++) {
   tasks.Add(WaitAndReturn(i));
}
int result = await await
     Task.WhenAny(tasks);
```

Example 10. WhenAny

- Example 11, `Task.WhenAll(..)`:
  Is also used on a collection of Tasks but instead of using the first `Task` it completes when all the Tasks are completed, which is very useful when you have a lot of independent Tasks which are not dependant on each other.

```
var tasks = new HashSet<Task<int>();
for(var i = 0; i < 5; i++) {
   tasks.Add(WaitAndReturn(i));
}
int[] result = await Task.WhenAll(tasks);
```

Example 11. WhenAll

III. Case Study

Goal of the case study was to build an application which accesses different banks and shops. Focus of the work was to fetch data simultaneously from these agents and combine the different prices. Figure 2 shows the architecture of the System. In our case Agents are self-built Shops and Banks which can return only one price/course per request. A customer can use a web interface in any browser as a client. The Hub is an ASP.Net application which on request fetches all needed data and returns it to the Customer.

To better understand our case study, we will explain the auxiliary classes from Example 12 first.

- `enum Currency`: A few listed currencies
- `enum Item`: A list of grocery items
- `class Product`: A product consisting of an `Item` with a price in a `Currency`

```
enum Currency { EUR, USD, ... }
enum Item { Apple, Banana, ... }
class Product {
    double Price { get; }
    Currency Currency { get; }
    Item Type { get; }
}
```

Example 12. Auxiliary classes

A. Agents

The following code in example 13 in essence shows how the Agents are built. The important thing is, that every `BankAgent` has a method to convert any amount of one currency into another currency. Different agents may use different conversion functions internally. Furthermore every `ShopAgent` has its own dictionary of products which can be queried with a method.

```
abstract class Agent {
   public string Name;
   public string Description;
}
class BankAgent : Agent {
   Func<double, Currency, Currency,
       Task<double>> convert;

   public async Task<double> Convert(double p,
       Currency f, Currency t) {
     return await convert(p, f, t);
   }
}
public class ShopAgent : Agent {
   Dictionary<Item, (Product, int)>
       availableProducts = ...

   public async Task<(Product p, int q)>
       GetItemWithQuantity(Item i) {
      // return item with quantity if available
   }
}
```

Example 13. Agent Code

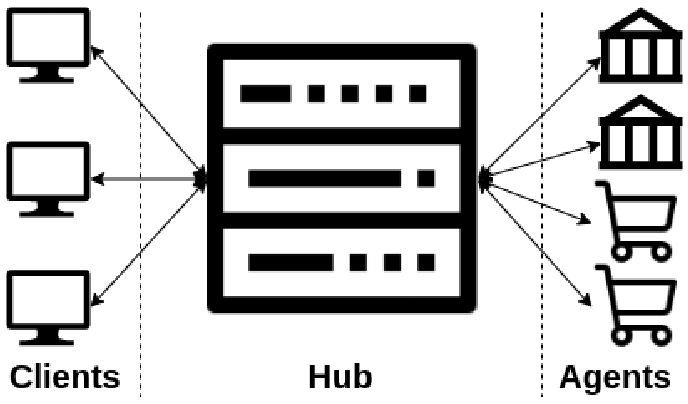
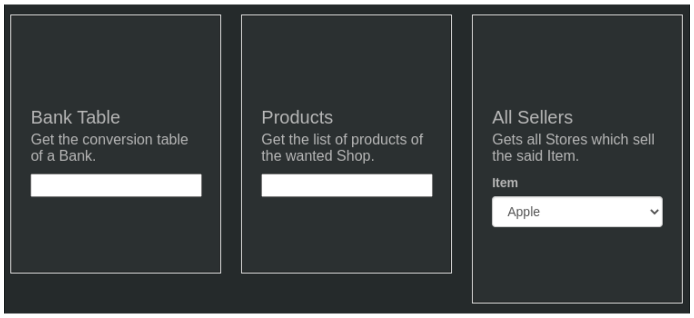


Fig. 2. High level architecture

B. Client

For the client no special code is necessary, everything is handled by the API-gateway. The clients only send REST requests to the hub in order to receive the information they need. These requests can easily be done from a website provided by the gateway. The following figure shows some available requests for the gateway.


Bank Table
Get the conversion table of a Bank.

Products
Get the list of products of the wanted Shop.

All Sellers
Gets all Stores which sell the said Item.
Item
Apple

Fig. 3. Screenshot of some available actions

C. Gateway

The gateway is the heart of this case study. It is built in the ASP.Net framework and naively provides an asynchronous way to deal with REST requests in C#. In this framework it is sufficient to write a method in the Controller in order to provide functionality via REST. All our methods in the controller look similar like example 14. The basic scheme is to request some data with an `async` method and provide it in the form of (mostly) a list as soon as it is available to the view. The view is a cshtml-file, which is sent to the requester.

```
public async Task<ActionResult> ShopAgents()
    =>
  await Task
    .Run(async () => ViewData["shops"] =
        (await
       _caseStudy.AllShops()).ToList())
    .ContinueWith(View);
```

Example 14. A controller method

As we can see in the above example there is a caseStudy object wich does some asynchronous calculations. This is the most interesting part of the case study because there we combined our entire knowledge about asynchronous programming in C#. Let us now take a deeper look into some of these methods:

The following code searches for the bank, which returns the most of the second `Currency` for `price` of the first `Currency`. First it asynchronously gets all banks and requests a conversion from them. When all banks are finished the method continues with selecting the best conversion rate by sorting and taking the last element.

```
public async Task<BankAgent> BestBank(double
    price, Currency from, Currency to) {
  var conversionTasks = (await AllBanks())
    .Select(async bank => (bank, conversion:
        await bank.Convert(price, from, to)));

  return await await
      Task.WhenAll(conversionTasks)
    .ContinueWith(async task =>(await task)
      .OrderBy(c => c.conversion)
      .Last().bank);
}
```

Example 15. Query for the best bank

Example 16 shows, how the different features from section II-B0c can be used to get different results from the Banks. `CheapestConvert` waits for the `BestBank`-Task to finish and then continues with the conversion. `FastestConvert` gets all banks, starts a conversion with each bank and selects the first result.

```
public async Task<double> CheapestConvert(
  double p, Currency f, Currency t) =>
  await await BestBank(100, f, t)
    .ContinueWith(async a => p /
      await (await a).Convert(1, f, t));

public async Task<double> FastestConvert(
  double p, Currency f, Currency t) =>
  await await Task.WhenAny((await AllBanks())
    .Select(async bank => p /
      await bank.Convert(1, f, t)));
```

Example 16. Fastest and cheapest conversions

IV. Conclusion

Overall our opinion towards `async/await` in C# is positive. The features are well integrated within the language. The feature methods fit well within the LINQ-system. Unfortunately we had a few lost hours trying to make the where-clause asynchronous until we understood that that is not possible, because this usually requires comparisons with other elements. To our disappointment the version we chose for the project, did not support Tasks as return values on Linux yet. After a bit of struggle of finding out what is happening, we both agreed to work on Windows, which is the native platform for C# anyway.

Material

[1] Stephen Cleary. *Concurrency in C# Cookbook. Asynchronous, Parallel, and Multithreaded Programming*. 2nd ed. O'Reilly Media, Inc., 2019.

[2] Microsoft. *Task Class*. 2023. URL: https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.task?view=net-7.0 (visited on 02/24/2023).

[3] Bill Wagner. *Asynchronous programming with async and await*. 2023. URL: https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/ (visited on 02/24/2023).