Async with C# 5

Andrew Clymer
andy@rocksolidknowledge.com

DEVELOPMENTOR

DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

Objectives

- Why use Continuations
- Simple examples of async/await
- Gotcha's
- Composition
- Under the hood
- Server side async/await

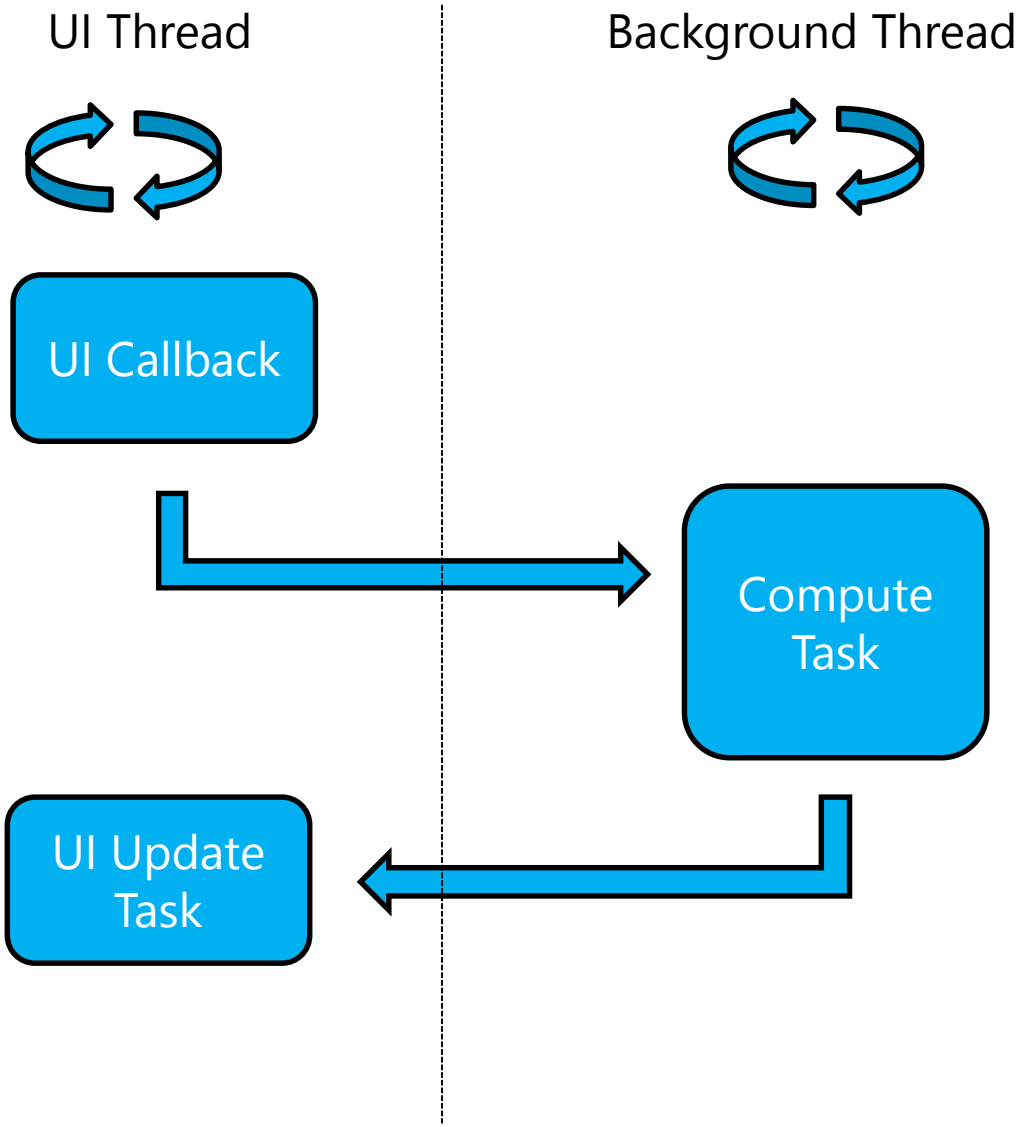
Historic Async

- Utilise framework API's
  - APM - BeginXXX/EndXXX
- Delegate Begin/End Invoke
- Thread Affinity issues
- XXXXAsync() / Completed Event
  - EAP - Event fires on the UI thread

Task

- .NET 4 Introduces another way
  - Introduces common type for all asynchronous operations
    - Asynchronous I/O
    - Asynchronous Compute
  - Use continuations to handle results

Continuations



Control flow

- Sequential programming intent is pretty clear.
- Asynchronous programming screws with intent
- Do X Async, Then Do Y , Then Do Z Async
- How to handle errors
  - Where to place the try/catch

C# 5 async/await keywords

- Making async intent as clear as synchronous intent
  - Two new keywords for C# 5
  - Enables continuations whilst maintaining the readability of sequential code
    - Automatic marshalling back on to the UI thread
  - Built around Task, and Task<T>

Example :async and await

- async method must return void or a Task
- async method should include an await
- await <TASK>

```
private async void Button_Click(object sender, RoutedEventArgs e) {
   calcButton.IsEnabled = false;
   Task<double> piResult = CalcPiAsync(1000000000);

    // If piResult not ready returns, allowing UI to continue
    // When has completed, returns back to this thread
    // and coerces piResult.Result out and into pi variable
    double pi = await piResult;

    calcButton.IsEnabled = true;
    this.pi.Text = pi.ToString();
 }
```

- Can return Task<T>
  - Code returns T, compiler returns Task<T>

```
public static async Task<byte[]> DownloadDataAsync(Uri source)
{
   WebClient client =new WebClient();
   byte[] data = await client.DownloadDataTaskAsync(source);

   ProcessData(data);

   return data;
}
```

Favour continuations over waiting

- Threads aren’t free
- A thread waiting can’t be used for anything else.
- Using continuations can reduce the total number of required threads

async keyword does not make code run **asynchronously**

```
public static async Task DoItAsync()
{
    // Still on calling thread
      Thread.Sleep(5000);
      Console.WriteLine("done it..");
}
```

async/await, Gotcha #2

Avoid async methods returning void

```
// What no errors
private static async void UploadLogFilesAsync(string uri)
 {
          var client = new WebClient();
           string sourceDirectory = @"C:\temp\";

           foreach (FileInfo logFile in new DirectoryInfo(sourceDirectory).GetFiles("*.log"))
           {
               await client.UploadFileTaskAsync(uri, logFile.FullName);
           }
 }
```

- Prefer Task over void

  - Better to return Task than void
  - Allows caller to handle error
  - void is there for asynchronous event handlers

```
public static async Task DownloadData(Uri source)
{
   WebClient client =new WebClient();
   byte[] data = await client.DownloadDataTaskAsync(source);

   ProcessData(data);
}
```

async/await, Gotcha #3

THINK before using async lambda for Action delegate

```
List<Request> requests;
. . .
requests.ForEach( async request =>
        {
            var client = new WebClient();
            Console.WriteLine("Downloading {0}", request.Uri);
            request.Content = await client.DownloadDataTaskAsync(request.Uri);
        });

Console.WriteLine("All done..??");

requests.ForEach(r => Console.WriteLine(r.Content.Length));
```

async/await, Gotch #4

await exception handling only delivers first exception

Error handling

- Tasks can throw many exceptions via an AggregateException
  - Await re-throws only first exception from Aggregate
- Examine Task.Exception property for all errors

```
public static async Task LoadAndProcessAsync()
{
  Task<byte[]> loadDataTask = null;
   try {
    loadDataTask = LoadAsync();
    byte[] data = await loadDataTask;

    ProcessData(data);
   } catch(Exception firstError) {
       loadDataTask.Exception.Flatten().Handle( MyErrorHandler );
   }

}
```

ConfigureAwait

- Possibly the worst API ever conceived
- Not all await's need to make use of SynchronizationContext

```
public static async Task DownloadData(Uri source, string destination)
 {
   WebClient client =new WebClient();
   byte[] data = await client.DownloadDataTaskAsync(source);

// DON'T NEED TO BE ON UI THREAD HERE…
   ProcessData(data);

   using (Stream downloadStream = File.OpenWrite(destination))
   {
     await downloadStream.WriteAsync(data, 0, data.Length);
   }
   // Must be back on UI thread
   UpdateUI("Downloaded");
}
```

ConfigureAwait

- First attempt, but wrong

```
public static async Task DownloadData(Uri source, string destination)
{
   WebClient client =new WebClient();
   byte[] data = await client
                          .DownloadDataTaskAsync(source)
                          .ConfigureAwait(false);

   // Will continue  not on UI thread
   using (Stream downloadStream = File.OpenWrite(destination)) {
     await downloadStream.WriteAsync(data, 0, data.Length);
   }

   // Hmmm…Need to be back on UI thread here

   UpdateUI("All downloaded");
}
```

ConfigureAwait

- Effective use of ConfigureAwait with composition
- Get compiler to create Task per context

```
public static async Task DownloadData(Uri source, string destination){
  await DownloadAsync(source, destination);
  // on UI thread
  UpdateUI("All downloaded");
}

private static async Task DownloadAsync(Uri source, string destination) {
   WebClient client = new WebClient();
   byte[] data = await client
                        .DownloadDataTaskAsync(source)
                        .ConfigureAwait(continueOnCapturedContext:false);

   using (Stream downloadStream = File.OpenWrite(destination)) {
      await downloadStream.WriteAsync(data, 0, data.Length);
   }
}
```

ConfigureAwait

- GOTCHA

```
private static async Task DownloadAsync(Uri source, string destination) {
   WebClient client = new WebClient();
   byte[] data = await client
                        .DownloadDataTaskAsync(source)
                        .ConfigureAwait(continueOnCapturedContext:false);

   // await's that complete immediately will continue on same thread
   // So what thread are we running on ?

   using (Stream downloadStream = File.OpenWrite(destination)) {
      await downloadStream.WriteAsync(data, 0, data.Length);
   }
}
```

ConfigureAwait

- Need to re-assert ConfigureAwait

```
private static async Task DownloadAsync(Uri source, string destination) {
   WebClient client = new WebClient();
   byte[] data = await client
                        .DownloadDataTaskAsync(source)
                        .ConfigureAwait(continueOnCapturedContext:false);

   // await's that complete immediately will continue on same thread
   // So what thread are we running on ?

   using (Stream downloadStream = File.OpenWrite(destination)) {
      await downloadStream
            .WriteAsync(data, 0, data.Length)
            .ConfigureAwait(continueOnCapturedContext:false);

   }
}
```

Everything a task

- TaskCompletionSource
  - Used to model the lifecycle of a Task
  - TaskCompletionSource.Task provides a task
  - New Task state signalled via
    - TaskCompletionSource.SetResult
  - Can be used for
    - building adapters for legacy apis
    - Stubbing asynchronous methods for testing

Orchestrating many tasks

- Continue when all tasks complete
  - Fork/join
  - WhenAll
- Continue when any one task completes
  - First result in wins
  - WhenAny

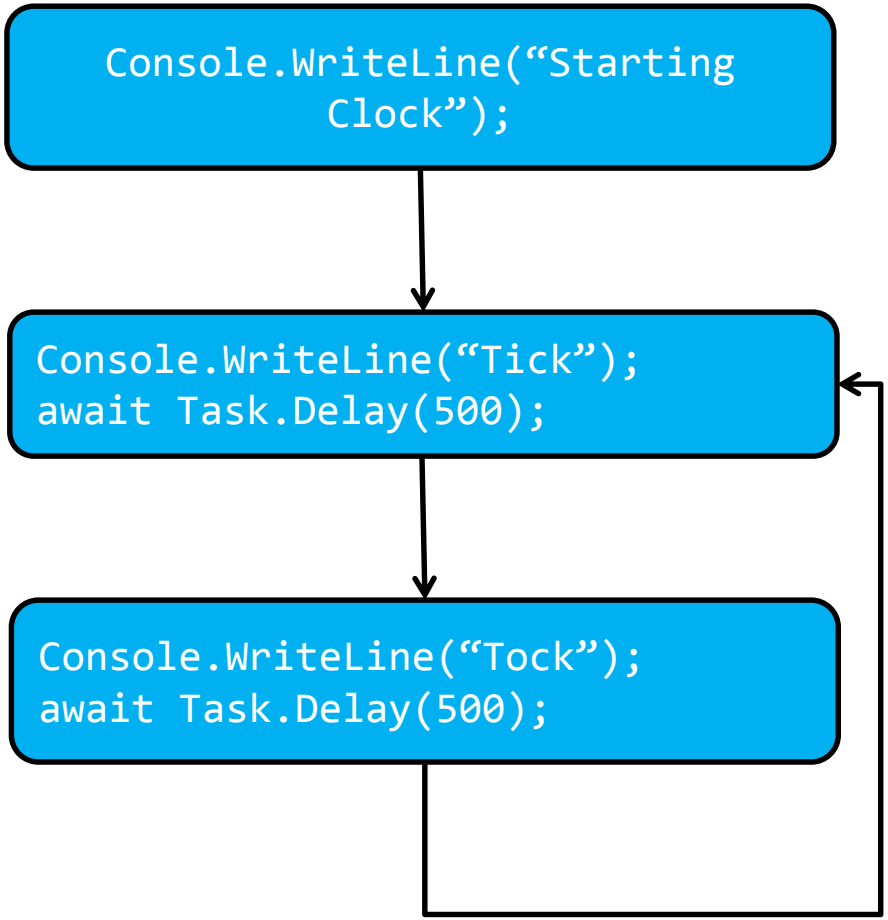
Async on the server

- Not just client side technology
  - ASP.NET Web Forms
    - Page marked as async
    - Page load method execute async/await
  - WCF 4.5
  - WebAPI

async/await under the hood

- Compiler builds state machine

```
private static async void TickTockAsync()
{
 Console.WriteLine("Starting Clock");

while (true)
 {
  Console.WriteLine("Tick");
  await Task.Delay(500);

  Console.WriteLine("Tock");
  await Task.Delay(500);
 }
}
```

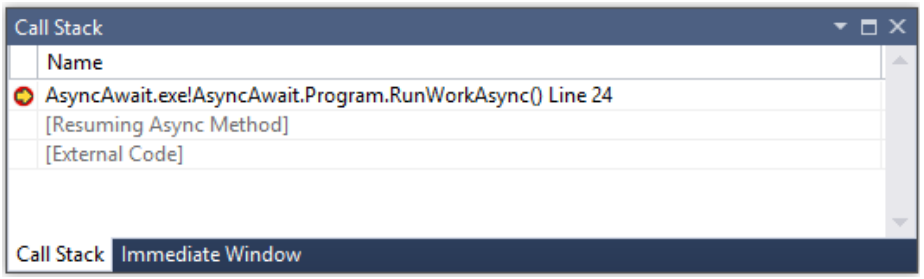


Improved debugging support

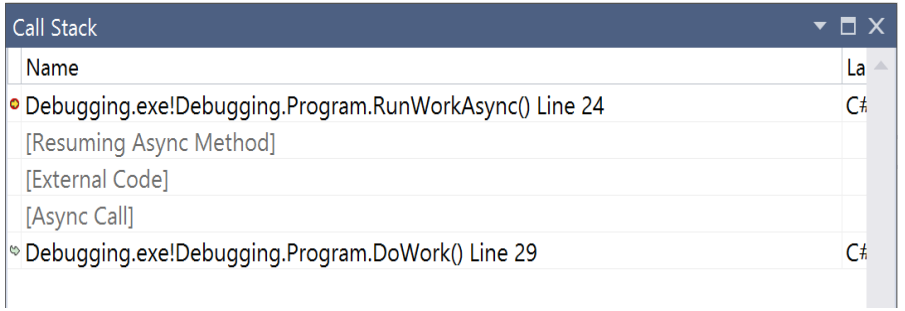
- VS2013 + Windows 8.1 or Server 2012 R2

```
static async Task RunWorkAsync() {
  Console.WriteLine("Starting work");

  await Task.Delay(2000);

  Console.WriteLine("background work complete");
}

static async Task DoWork() {
  await RunWorkAsync();
  Console.WriteLine("DoWork");
}
  }
```

Visual Studio 2012



Visual Studio 2013



Summary

- Utilise async/await to
  - Simplify continuations
  - Reduce number of threads
- Use ConfigureAwait to reduce work on UI thread
- Utilise TaskSource to build async/await friendly abstractions