ASYNCHRONOUS PROGRAMMING

SUCCINCTLY

BY **DIRK STRAUSS**

SUCCINCTLY E-BOOK SERIES

Syncfusion®

Asynchronous Programming Succinctly

By
Dirk Strauss

Foreword by Daniel Jebaraj

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

Technical Reviewer: James McCaffrey

Copy Editor: John Elderkin

Acquisitions Coordinator: Morgan Weston, social media marketing manager, Syncfusion, Inc.

Proofreader: Darren West, content producer, Syncfusion, Inc.

Table of Contents

About the Author

Dirk Strauss is a software developer and Microsoft .NET MVP from South Africa with over 13 years of programming experience.

Starting his career at a small software development house in Port Elizabeth, he moved on to EOH Applications in 2007. He spent the next 8 years writing software to integrate into SYSPRO. It was during this time that he started blogging frequently to avoid stagnating and to keep on learning.

In 2015 he joined the Evolution Software team where he lives out his creativity and works with incredibly inspirational individuals. During this time he published two books on C# as well as a few e-books. He continues learning and sharing whenever he can. You can find him @DirkStrauss on Twitter or on his blog ***dirkstrauss.com***.

Chapter 1 Getting Started

What is asynchronous programming?

Recently, a friend of mine was tasked with improving a Windows Forms application that had been written about four years ago. As I looked on, I saw that calls to the database locked up the UI for several seconds. The search functionality (which searched a table consisting of only 60,000 entries) took from 30 seconds to a minute to return results to the form. This meant that while the application was querying the database and doing what it needed to do, the UI was totally unresponsive. The user could not resize it, move it around, or interact with it in any way. Instead, users had to sit and wait for the application to respond. Reading and writing files was equally frustrating. The unresponsiveness of the application created negative feelings—that the application was malfunctioning, was old (which was technically true), and was no good.

In actual fact, the application was not malfunctioning. The search returned true results from the database. Saving information to the database worked every time, and the files the application created or read always worked. Nothing was "broken," but the fact that the UI was slow and unresponsive at times (especially as the customer's workload increased) made using the app extremely frustrating.

While this situation might have been due to a combination of poor code and inefficient SQL statements, in fact the form was not responsive because it was doing everything synchronously—any process that accesses the UI thread (UI-related tasks usually share a single thread) will be blocked in a synchronous application. Therefore, if one process is blocked, all processes are blocked.

Asynchrony is the saving grace for these kinds of applications. Using asynchronous methods will allow the application to remain responsive. The user can resize, minimize, maximize, move the form around, or even close the application should they wish to. Introducing asynchronous functionality into applications had long been a complicated endeavour for developers, but the introduction of async programming in Visual Studio 2012 introduced a simplified approach. Async programming leveraged the .NET Framework 4.5, which made it easier to code and moved all the heavy lifting and complicated code to the compiler. Don't get me wrong, though—while asynchronous programming is still a complex bit of technology, it has been made much easier to use.

Examples of potentially blocking activities

We now know one thing: async improves the responsiveness of your application. But just which activities are potentially blocking? For example, take accessing a web resource. In a synchronous scenario, the entire program will be blocked if the web resource being accessed is slow or overloaded. The following list defines several areas in which async programming will improve the responsiveness of your application.

Web Access

- HttpClient
- SyndicationClient

Files

- StorageFile
- StreamWriter
- StreamReader
- XmlReader

Images

- MediaCapture
- BitmapEncoder
- BitmapDecoder

WCF

- Asynchronous Service Operations

Note: Microsoft Visual Studio Enterprise 2015 (Update 3) will be used to create all the code samples in this e-book. The .NET Framework 4.6.1 is the framework that will be used.

Writing async methods

The **`async`** and **`await`** keywords are essential in async programming. Using these keywords, developers are able to create async methods. But what else characterizes async methods? As it turns out, there are a few things to consider. The signature of the method must include the **`async`** modifier. The method must have a return type of **`Task<TResult>`**, **`Task`**, or **`void`**. The method statements must include at least a single **`await`** expression—this tells the compiler that the method needs to be suspended while the awaited operation is busy. Lastly, the method name should end with the "async" suffix (even though this is more convention than required). Why would this convention be significant, you ask? Well, that's the convention used in the .NET Framework 4.5 and newer versions. Have a look at how easy it is to find the asynchronous methods of the **`HttpClient`** class. Typing ".async" after the instance of the **`HttpClient`** object will filter the available methods in the Intellisense list, as you can see in Figure 1.

Note: It is important to note that methods are not awaitable. Types are awaitable. That is why you can await a Task<TResult> or Task.

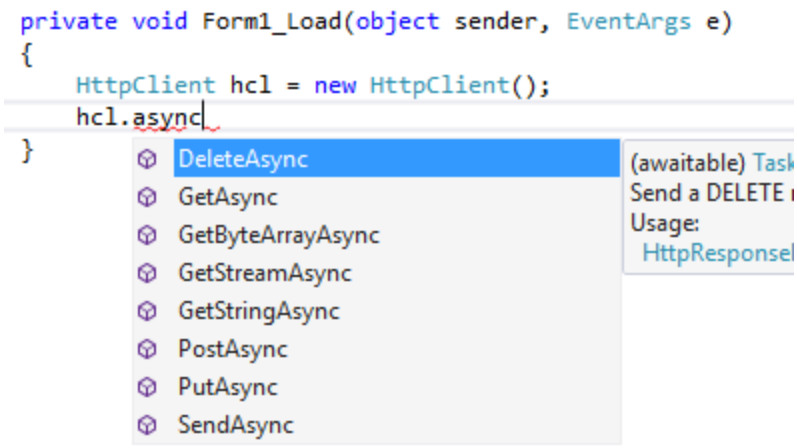


Figure 1: Async Methods of the HttpClient Class

Naming your asynchronous methods in a similar manner will allow other developers to easily spot the asynchronous methods you created in your classes.

Let's look at how easy it is to create a method that simulates reading text from some data store that takes six seconds to complete. We want our Windows Forms application to remain responsive throughout the process. To do this, we will put a timer on our form and click a button that delays for six seconds before returning text and setting the value of a label on the form. Throughout that process, the timer will continue to display the current time (including seconds) in the title bar of the Windows Form.

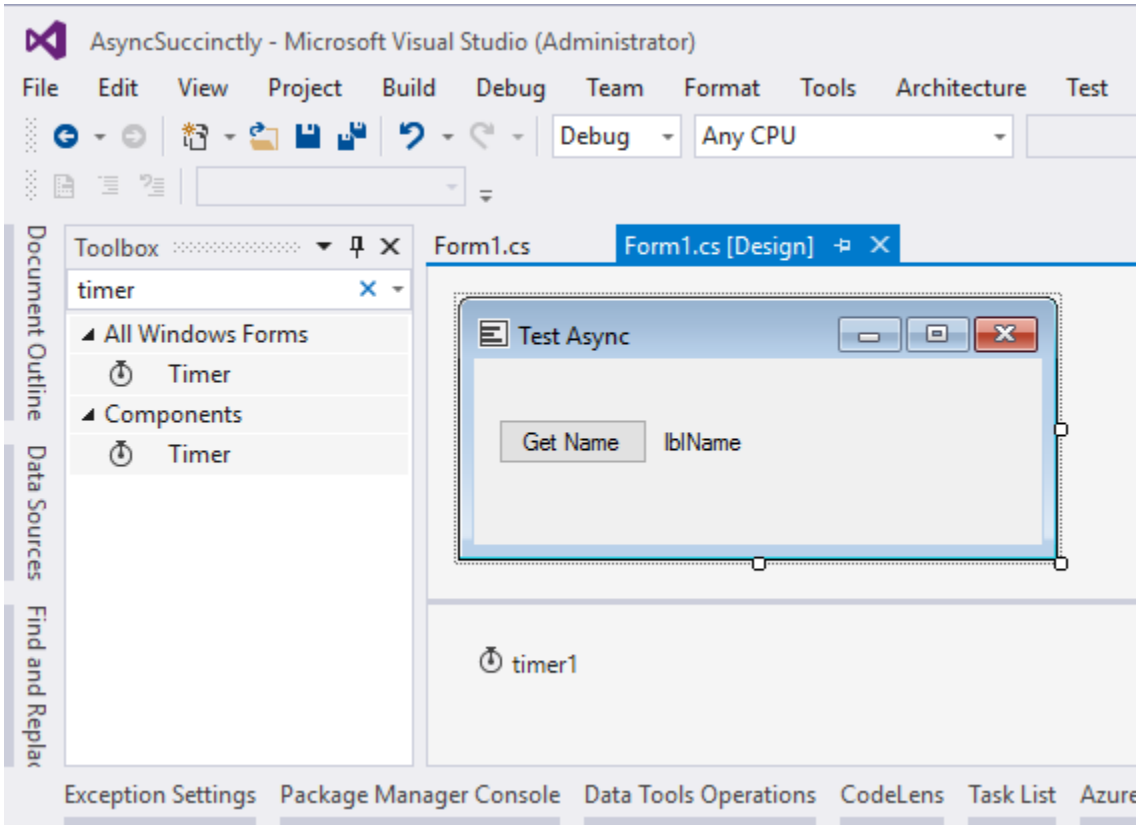


Figure 2: Create Windows Forms Application

You can design the form as you like, but there are a few components you'll need to add. These are:

- Button
- Label
- Windows Forms Timer (don't add the Components timer)

For the timer, create the tick event and add the following code:

Code Listing 1

```
private void timer1_Tick(object sender, EventArgs e)
{
    this.Text = DateTime.Now.ToLongTimeString();
}
```

In the Form Load, add the following code in order to start the timer. This is essential because otherwise the timer will not work.

Code Listing 2

```
private void Form1_Load(object sender, EventArgs e)
{
    timer1.Start();
}
```

Add a method called **`ReadTextAsync()`** and use the **`async`** keyword on the method signature. The **`Task<string>`** tells us that this method returns a string value. It will only return the string value after a six-second delay.

Code Listing 3

```
private async Task<string> ReadTextAsync()
{
    await Task.Delay(TimeSpan.FromMilliseconds(6000));
    return "async succinctly";
}
```

Next, create a click event for the button that you added to your Windows Form. Be sure to add the **`async`** keyword to this button click event. Call the **`ReadTextAsync()`** method using the **`await`** keyword. Lastly, set the label's text value to the text returned from the **`async`** method.

Code Listing 4

```
private async void button1_Click(object sender, EventArgs e)
{
    string text = await ReadTextAsync();
    lblName.Text = text;
}
```

Run the application and you will notice the time, with the seconds counting, in the title of the form.

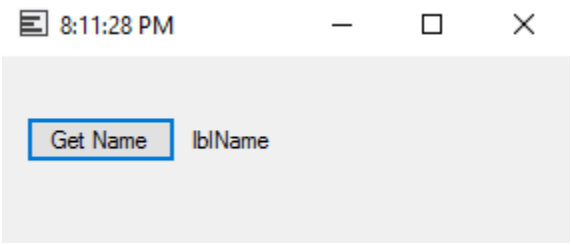


Figure 3: Async Windows Form

Click the **Get Name** button and wait for the label to be populated with the text returned from the **`ReadTextAsync()`** method. You will notice that the form remains responsive, allowing you to move it around and even resize it. You will also notice that the timer continues to tick.

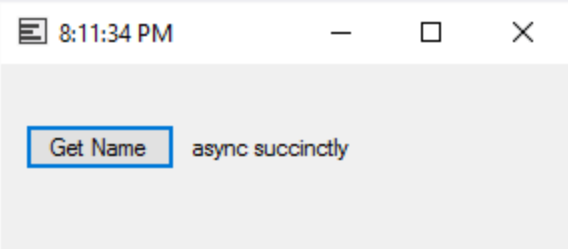


Figure 4: Return Text Asynchronously

Performing long-running or potentially blocking tasks asynchronously allows you to keep your application responsive, which improves usability.

> ***Tip: While it is possible to return a void from an async method, try not to use void unless you are creating an async event handler. As a general rule of thumb, all of your async methods should return Task if they return nothing or Task<T> if they return a value.***

Avoid deadlocks when using async

Many developers experience the problem of deadlocks when they begin exploring asynchronous programming. For example, you can easily cause a deadlock in your code when mixing asynchronous code with synchronous code. Let's first look at an example before we investigate why the application is deadlocked.

Start by adding a second button and label to your form. When you are done, your form should look like Figure 5.

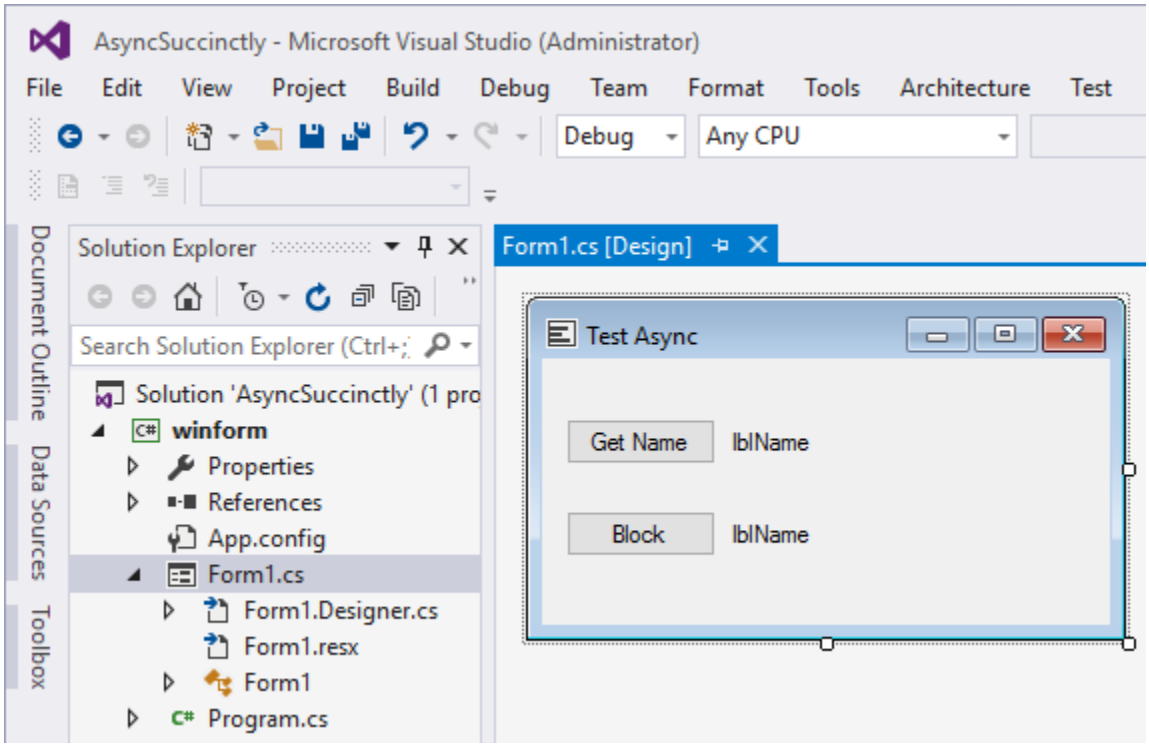


Figure 5: Modify Windows Form

I have simply added a second button and label to the form we created earlier. The code behind the **Block** button will call an async method that will cause a deadlock. The code in Code Listing 5 needs to be added to your code-behind.

Code Listing 5

```
private static async Task<string> ReadHelloWorldAsync(string value)
{
    await Task.Delay(TimeSpan.FromMilliseconds(6000));
    return $"hello {value}";
}
```

The **`ReadHelloWorldAsync(string value)`** is an async method that awaits a delay before returning a string value concatenated (by using string interpolation) with the string parameter value. In your button click event, the code in Code Listing 6 will cause a deadlock.

Code Listing 6

```
private void button2_Click(object sender, EventArgs e)
{
    var stringValue = ReadHelloWorldAsync("world");
    lblName2.Text = stringValue.Result;
}
```

Here, I am trying to illustrate a problem that affects UI applications, including ASP.NET apps. Of course, this does not cause an issue in a Console application. Run your application and click the **Block** button.

> ***Note: You should save all your important items before doing this. The code illustrated in this example will cause your application to deadlock.***

When you click **Block**, your application becomes unresponsive and will stay that way until you stop debugging it. In order to stop the application, hold down **Shift + F5** or click **Stop Debugging** in Visual Studio.

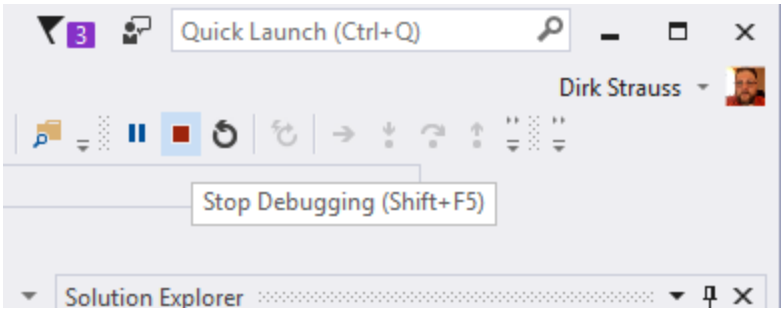


Figure 6: Stop Debugging

So, what happened here? Well, the answer has to do with contexts. What do we mean when we refer to a context? Without going into too much detail, here's an explanation:

- WinForm Applications uses an UI thread, and therefore the context is an UI context.
- When responding to ASP.NET requests, the context is an ASP.NET request context.
- If neither is the case, then the thread pool context is used.

The button click in our example blocks the context thread because it is waiting for the async method to finish. The async method, on the other hand, is waiting patiently for the context to get free so that it can finish. This happens because it continues on the same context that started it. If both wait for the other, we're deadlocked.

There are only two ways we can avoid a deadlocked application. The first technique is to use async in order to avoid blocking the task. Think of it this way—use async all the way down. The other technique is to use **`ConfigureAwait(false)`**. In order to illustrate this, we need to modify the code slightly in the async method. In the **`ReadHelloWorldAsync(string value)`** method, add **`ConfigureAwait(false)`** to the end of the **`Delay`**.

Code Listing 7

```
private static async Task<string> ReadHelloWorldAsync(string value)
{
    await
Task.Delay(TimeSpan.FromMilliseconds(6000)).ConfigureAwait(false);
    return $"hello {value}";
}
```

Doing this tells the async method not to resume on the context. It will then resume on a thread in the thread pool. Have a look at the definition of the property when calling **`ConfigureAwait`**.

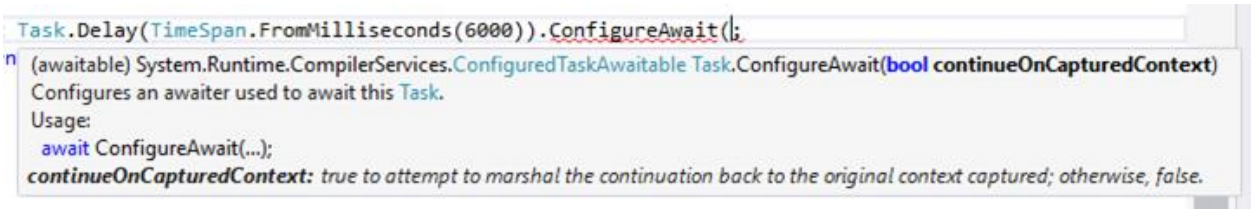


Figure 7: ConfigureAwait(false)

Running your application now will not cause a deadlock. The application becomes responsive again after the six-second delay is over.

I also mentioned avoiding deadlocks by using async all the way down. Do this by modifying the code as in the following code listings.

Code Listing 8

```
private static async Task<string> ReadHelloWorldAsync(string value)
{
    await Task.Delay(TimeSpan.FromMilliseconds(6000));
    return $"hello {value}";
}
```

The button click event now uses the async keyword and the code, then calls **`await`** on the async **`ReadHelloWorldAsync()`** method.

Code Listing 9

```
private async void button2_Click(object sender, EventArgs e)
{
    var stringValue = await ReadHelloWorldAsync("world");
    lblName2.Text = stringValue.ToString();
}
```

Run your application and notice that this time, not only does your application avoid a deadlock, but it also remains responsive throughout the delay. This is because all the waits are done asynchronously.

Chapter 2 How Do I Use Async

Exploring async further

With the basics covered, we now need to explore async a little further to see what it is capable of and how it can benefit developers who want to improve the responsiveness of their applications. In order to do this, let's look at the elements that constitute asynchronous code.

Async method return types

Let's have another look at the return types. An async method can have one of three possible return types:

- **`void`**
- **`Task`**
- **`Task<TResult>`**

Void return type

The **`void`** returning method is used with event handlers, and if you completed Chapter 1, you will reognize that you have already created an **`async void`** event handler.

Code Listing 10

```
private async void button1_Click(object sender, EventArgs e)
{
    string text = await ReadTextAsync();
    lblName.Text = text;
}
```

While it is possible to call a nonreturning asynchronous method using **`void`**, this is generally considered bad practice. A **`void`** returning method simply performs a task and does not return anything to the calling code. Instead of using **`void`** for these async methods, good practice means returning **`Task`** instead. Also, note that the **`Main`** method in a console application cannot be marked with the async modifier.

Task return type

If the async method you are calling does not return anything, call it using **`Task`** instead of **`void`**. This means that the following two methods in Code Listings 11 and 12 are essentially the same in synchronous and asynchronous code.

Code Listing 11

```
private void FireAndForget()
{
    // Code implementation
}
```

In Code Listing 12, the asynchronous method needs to be called with **Task** when nothing is returned.

Code Listing 12

```
private async Task FireAndForget()
{
    // Code implementation
}
```

Let's next look at Code Listing 13, which shows an example of using and calling an async method that returns nothing.

Code Listing 13

```
private async void button3_Click(object sender, EventArgs e)
{
    await ReadRemoteServerInfo();
}
```

The event handler calling the void returning async method, as in Code Listing 14, simply calls the method with the **await** keyword.

Code Listing 14

```
private async Task ReadRemoteServerInfo()
{
    await Task.Delay(TimeSpan.FromMilliseconds(6000));
    lblServerInfo.Text = "Server Information Set Here";
}
```

The **async** method sets the text of a label after a delay has been called. The delay here mimics the long-running process of reading information remotely via a web service or other data store.

Task<TResult> return type

The last return type is a **Task<TResult>**, which is used if your async method returns an object. In Code Listing 15, we are calculating the age of a person based on their birthday. It will return an integer value with the calculated age.

Code Listing 15

```
private async Task<int> GetAge(DateTime birthDay)
{
    await Task.Delay(TimeSpan.FromMilliseconds(6000));
    int age = DateTime.Today.Year - birthDay.Year;

    if (birthDay > DateTime.Today.AddYears(-age))
        age--;
    return age;
}
```

In the click event, we call the async method as we would any other method that returns a type. We must simply remember to **`await`** the call to the **`GetAge(birthday)`** method.

Code Listing 16

```
private async void button4_Click(object sender, EventArgs e)
{
    DateTime birthday = new DateTime(1975, 12, 21);
    int age = await GetAge(birthday);
    lblAge.Text = age.ToString();
}
```

Putting the call and awaiting in separate statements is another way of writing this.

Code Listing 17

```
private async void button4_Click(object sender, EventArgs e)
{
    DateTime birthday = new DateTime(1975, 12, 21);

    Task<int> ageTask = GetAge(birthday);
    lblAge.Text = "calculating...";
    int age = await ageTask;
    lblAge.Text = age.ToString();
}
```

We can also return objects we create in code. Let us assume that we have created a **`Person`** class that contains information calculated and passed to it from the calling code. Using an async method containing a delay to simulate a long-running task, we can add following code to create a **`Person`** class. The class takes the date of birth as a parameter passed to it in the constructor and uses that to calculate the age of the person.

Code Listing 18

```
public class Person
{
    public Person(DateTime birthDay)
    {
        int age = DateTime.Today.Year - birthDay.Year;

        if (birthDay > DateTime.Today.AddYears(-age))
            age--;

        Age = age;
    }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age { get; private set; }
}
```

We then create an async method that uses a factory pattern to return the created **Person** object.

Code Listing 19

```
private async Task<Person> GetPerson(string firstName, string lastName,
DateTime birthDay)
{
    await Task.Delay(TimeSpan.FromMilliseconds(6000));
    Person oPerson = new Person(birthDay);
    oPerson.FirstName = firstName;
    oPerson.LastName = lastName;

    return oPerson;
}
```

Code Listing 20 shows how calling the code can be written.

Code Listing 20

```
private async void button4_Click(object sender, EventArgs e)
{
    DateTime birthday = new DateTime(1975, 12, 21);
    Person person = await GetPerson("Dirk", "Strauss", birthday);
    // use the object
    lblAge.Text = person.Age.ToString();
}
```

We have seen that it is also possible to return an object from an async method. In fact, apart from using the **`async`** and **`await`** keywords along with the **`Task`** and **`Task<TResut>`** return types, we haven't written anything differently. We have been writing the plain old C# code that we all use. This brings me to another benefit of async programming—writing async methods are incredibly easy. We have not done anything special. All the heavy lifting and hard work is done by the compiler.

Cancel an async task

As developers, we can exhibit control over the cancellation of an async method. We might need to do this if, for example, the user wants to stop the process before it completes. The .NET Framework gives us total control over the cancellation of an async method.

Cancel using a cancel button

The code listings that follow illustrate the concept of canceling by using a cancel button, but first we need to modify our Windows Form. Add a label called **lblElapsedTime** and add two buttons called **Start Process** and **Cancel Process**.

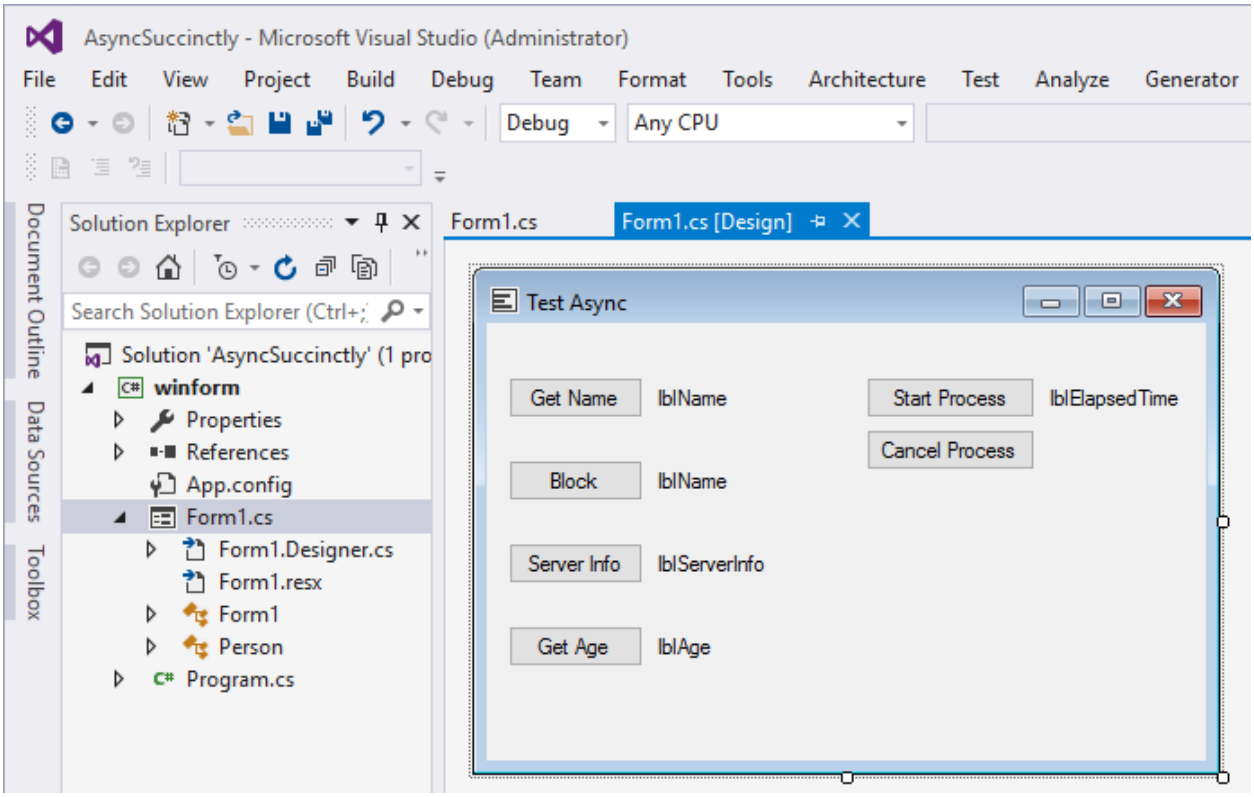


Figure 8: Cancel Async Method

In the code-behind, you will need to add the **`System.Threading`** namespace.

Code Listing 21

```
using System.Threading;
```

This allows us to add a **CancellationTokenSource** object to the code. Note that this is added as globally in scope.

Code Listing 22

```
public partial class Form1 : Form
{
    CancellationTokenSource cancelSource;
```

In the cancel button event, we need to call the **Cancel()** method of the **CancellationTokenSource** object if the object is not **null**.

Code Listing 23

```
private void btnCancelProcess_Click(object sender, EventArgs e)
{
    if (cancelSource != null)
        cancelSource.Cancel();
}
```

Our async method is simply delaying for six seconds in order to mimic the long-running task. It then returns the time taken for the process to complete. Note that we are passing a **CancellationToken** object to the async method as a parameter that is then used in the overloaded **Delay()** method. This is all we need in order to signal to the .NET Framework that the async process can be cancelled.

Code Listing 24

```
private async Task<DateTime> PerformTask(CancellationToken cancel)
{
    await Task.Delay(TimeSpan.FromMilliseconds(6000), cancel);
    return DateTime.Now;
}
```

In the start button event handler, we should begin by adding a try/catch to the handler. When the async method is cancelled, it will return an **OperationCancelledException**. We therefore need to cater to this by adding it as a specific catch before the more general **Exception** clause. If the async method is not cancelled, it will complete in the six seconds specified in the **Delay()** time span.

Code Listing 25

```
private async void btnStartProcess_Click(object sender, EventArgs e)
{
    try
    {

    }
    catch (OperationCanceledException)
    {

    }
    catch (Exception)
    {

    }
}
```

Next, further expand the start event handler by instantiating a new object of type **`CancellationTokenSource()`**. In the **`catch`** statements, display the elapsed time. Lastly, at the end of the start button event handler, set the **`CancellationTokenSource`** object **`cancelSource`** to **`null`**.

Code Listing 26

```
private async void btnStartProcess_Click(object sender, EventArgs e)
{
    cancelSource = new CancellationTokenSource();
    DateTime start = DateTime.Now;

    try
    {

    }
    catch (OperationCanceledException)
    {
        lblElapsedTime.Text = Convert.ToInt32((DateTime.Now -
start).TotalSeconds).ToString();
    }
    catch (Exception)
    {
        lblElapsedTime.Text = "Error";
    }

    cancelSource = null;
}
```

Lastly, in the **`try`** clause, add the code to await the **`PerformTask`** method. It is here that we will be passing the cancelSource token to the async method.

Code Listing 27

```
private async void btnStartProcess_Click(object sender, EventArgs e)
{
    cancelSource = new CancellationTokenSource();
    DateTime start = DateTime.Now;

    try
    {
        DateTime end = await PerformTask(cancelSource.Token);
        lblElapsedTime.Text = Convert.ToInt32((end -
start).TotalSeconds).ToString();
    }
    catch (OperationCanceledException)
    {
        lblElapsedTime.Text = Convert.ToInt32((DateTime.Now -
start).TotalSeconds).ToString();
    }
    catch (Exception)
    {
        lblElapsedTime.Text = "Error";
    }

    cancelSource = null;
}
```

Now run your application. When you click **Start Process** and leave it to complete, the delay of six seconds is displayed in the **lblElapsedTime** label.

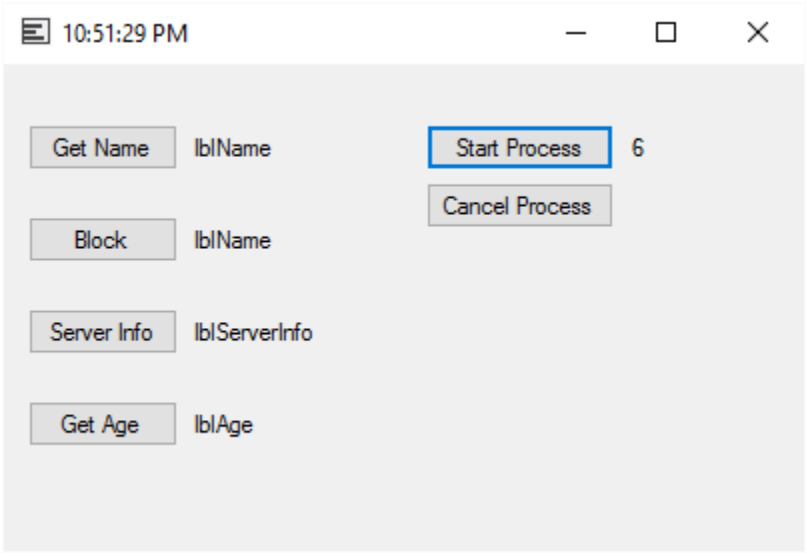


Figure 9: Start Process

If, however, we had started the process and then, before the process completed, clicked the **Cancel Process** button, the async method would have been cancelled and the elapsed time displayed in the **lblElapsedTime** label.

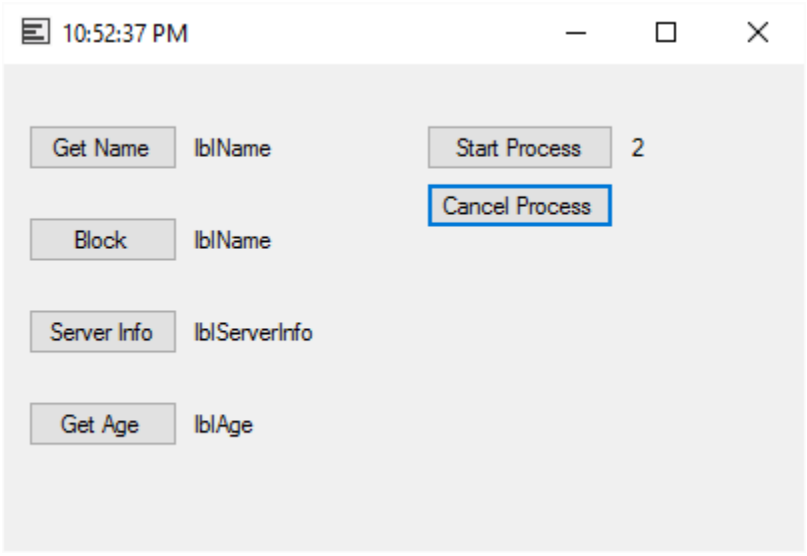


Figure 10: Cancel Process

You will notice that the time elapsed is less than the six-second delay, meaning that the delay never ran to completion.

Cancel after a specific time

Sometimes we need to specify a timeout associated with the async method. This might be because the process must be deemed a failure if the response isn't received within the elapsed timeout period. Or perhaps there isn't a cancel button that the user can click. For whatever reason, the .NET Framework can allow us to schedule a cancellation of the async method.

This is extremely easy. Before we await the async method, we must call the **`CancelAfter()`** method of the **`CancellationTokenSource()`** object by passing it the **`int`** milliseconds or **`TimeSpan`**. After this, the async method will be cancelled.

Code Listing 28

```
cancelSource.CancelAfter(3000);
```

Code Listing 29 shows how your start button event handler will now look.

Code Listing 29

```
private async void btnStartProcess_Click(object sender, EventArgs e)
{
    cancelSource = new CancellationTokenSource();
    DateTime start = DateTime.Now;

    try
    {
        cancelSource.CancelAfter(3000);
        DateTime end = await PerformTask(cancelSource.Token);
        lblElapsedTime.Text = Convert.ToInt32((end -
start).TotalSeconds).ToString();
    }
    catch (OperationCanceledException)
    {
        lblElapsedTime.Text = Convert.ToInt32((DateTime.Now -
start).TotalSeconds).ToString();
    }
    catch (Exception)
    {
        lblElapsedTime.Text = "Error";
    }

    cancelSource = null;
}
```

That's all there is to it. If you run your application and click the **Start Process** button, the async method will cancel after three seconds without the user clicking on the **Cancel Process** button.

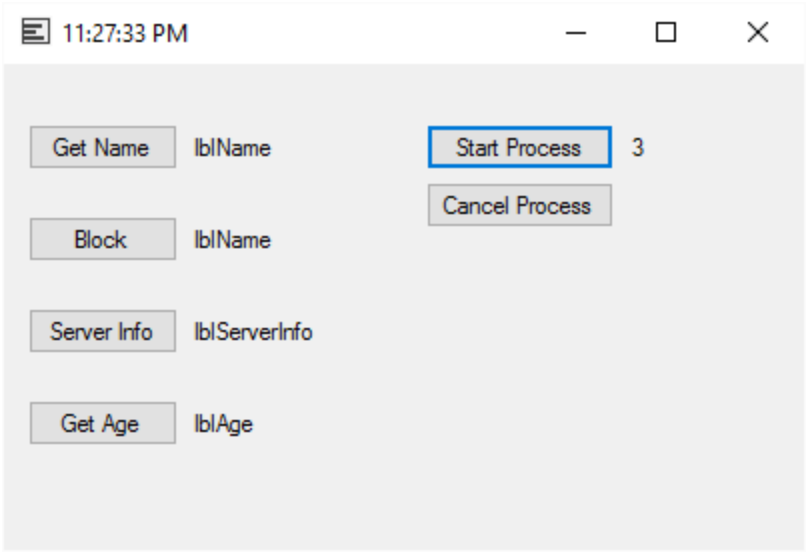


Figure 11: Cancel Async Method After a Timeout

Await in catch and finally

Some of you who are already using C# 6.0 might know that you can now await in the catch/finally when handling exceptions. This feature was not previously available to developers, but some very smart folks over in the C# design team figured it out, and now we can use async in catch/finally blocks.

Why would we want to do this? Let's assume that we need to access a web resource. If the resource fails or times out, we might want to get the source from a backup location. In order to demonstrate something like this, we will create a Windows Form that uses the URL to a web page to download the source as a string. Then we output that string to a rich text box control.

In case the original URL fails for whatever reason, we have a default URL. In our example, we will pass the async method a bad URL and cause it to throw an exception. In order to illustrate the improvements in C# 6.0, I will show you how a developer might have handled an exception before C# 6.0 introduced await in catch/finally blocks. I will then modify the code to illustrate how C# 6.0 has made it easier to await in a catch/finally block.

First, we'll start by creating a form similar to the one in Figure 12.

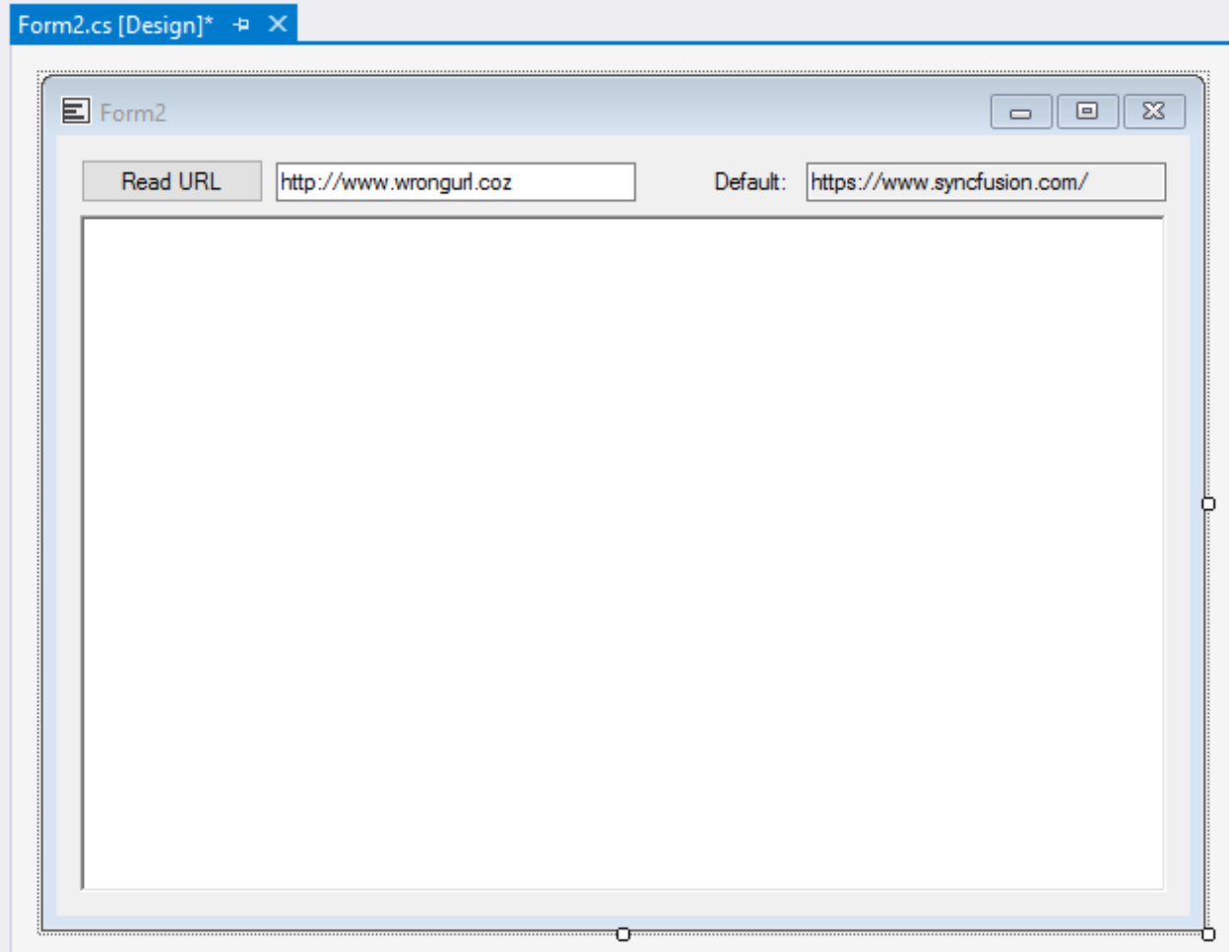


Figure 12: Exception Handling Demo

Add a reference to **`System.Net`**.

Code Listing 30

```
using System.Net;
```

Now, we need to create an async method that returns a string. It will read the source from a URL as a string and return it to the calling code. Next, it will try to get the URL from the URL supplied (which, in our demo, is an incorrect URL). If the URL is incorrect, it will read the default URL.

As you can see, in the past C# 6.0 developers had to set a flag in the catch block. Depending on the value, we load the default URL.

Code Listing 31

```
public async Task<string> ReadURL()
{
    WebClient wc = new WebClient();
    string result = "";
    bool downloaded;
    try
    {
        result = await wc.DownloadStringTaskAsync(new Uri(txtURL.Text));
        downloaded = true;
    }
    catch
    {
        downloaded = false;
    }

    if (!downloaded)
        result = await wc.DownloadStringTaskAsync(new
Uri(txtDefault.Text));

    return result;
}
```

Lastly, we add the button click event to call the async method and return the result to the rich text box.

Code Listing 32

```
private async void btnReadURL_Click(object sender, EventArgs e)
{
    string strResult = await ReadURL();
    rtbHTML.Text = strResult;
}
```

Run the application and click **Read URL**.

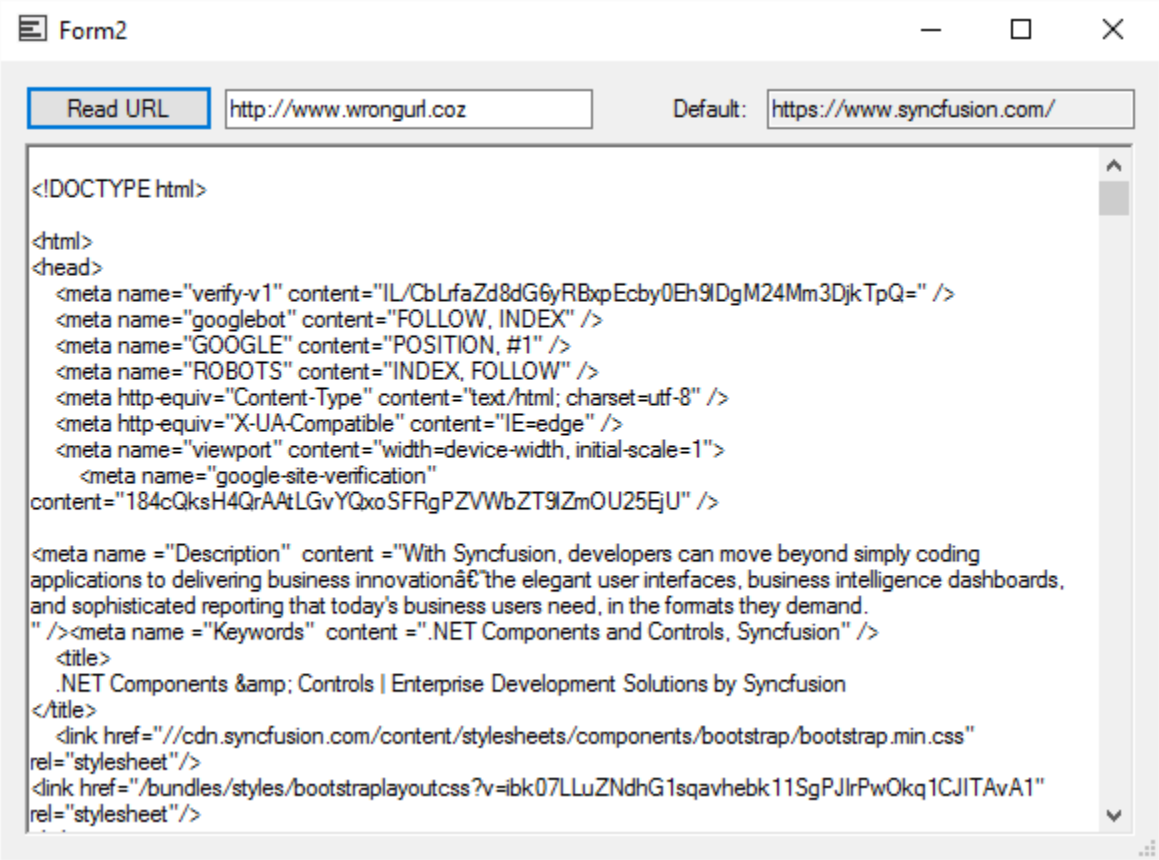


Figure 13: Default URL Source Read

As you can see, the code works. Personally, in the past I disliked having to write code this way. Luckily, now we can simply modify our code in the **`ReadURL()`** async method.

Code Listing 33

```
public async Task<string> ReadURL()
{
    WebClient wc = new WebClient();
    string result = "";
    try
    {
        result = await wc.DownloadStringTaskAsync(new Uri(txtURL.Text));
    }
    catch
    {
        result = await wc.DownloadStringTaskAsync(new
Uri(txtDefault.Text));
    }

    return result;
}
```

Instead of using the **`bool`** value to check if an exception has been thrown, we can simply tell the compiler to check the default URL if the URL in the **`try`** throws an exception. This code is cleaner and easier to read. Note that you can also **`await`** in the **`finally`** block.

Object Oriented Programming and async

Abstract classes

Let us have a look at abstract classes. The idea behind abstract classes is to ensure that inherited classes contain common functionality. This is great if you want your inherited class to always do a specific thing. But keep in mind that abstract classes can't ever be instantiated—they merely describe what needs to be implemented in the derived classes. Your implementation detail therefore resides in the derived class.

We should also remember that async is an implementation detail. This means that you can't define a method in an abstract class as an async method. So, what do we do? Well, remember that types are awaitable. Methods are not, which means you must then define your abstract method as returning a **`Task`** or **`Task<TResult>`**. With that, you should be good to go. Let's look at an example.

Add two new classes to your Visual Studio solution. Call one **Automation.cs** and the other **ProjectAutomation.cs**.

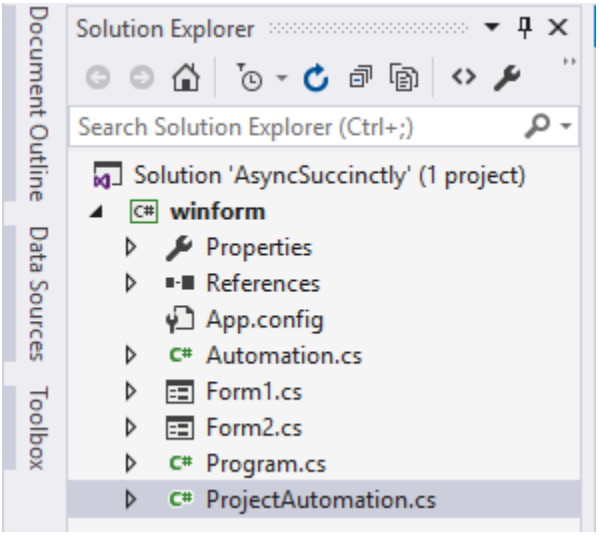


Figure 14: Classes in Solution

We are going to assume that a company creates various types of automated objects in the database. Sometimes they create new projects, other times they create jobs, and so on. All these objects are automations because once the entry is created in the database, another system picks up the creation of this object (e.g., **`ProjectAutomation`** or **`JobAutomation`** or **`TaskAutomation`**) and performs additional work in an external ERP system.

When these automations are completed, the database object is flagged as completed by the external system. Our application then purges the automation object. Add the **`System.Threading.Tasks`** namespace to both classes (**Automation.cs** and **ProjectAutomation.cs**).

Code Listing 34

```
using System.Threading.Tasks;
```

Now add the code for the abstract **`Automation`** class. Notice that because we want the **`AutomationCompleted()`** and **`PurgeAutomation()`** methods to be async methods in the derived class, we specify the return types as **`Task<bool>`** and **`Task`** for the void returning async method.

Code Listing 35

```
public abstract class Automation
{
    public abstract void StartUp(int ownerID, int automationType);
    public abstract Task<bool> AutomationCompleted();
    public abstract Task PurgeAutomation(int ownerID, DateTime
purgeDate);
}
```

When we create our derived class called **`ProjectAutomation`**, we specify that it has to inherit from the abstract class **`Automation`**. I have added a bit more meat to this derived class than necessary, but I want to focus your attention on the **`AutomationCompleted()`** and **`PurgeAutomation()`** methods. Because they are returning **`Task<TResult>`** and **`Task`**, we can use the **`async`** and **`await`** keywords here to mark them as async methods even though they are not defined as async in the abstract class.

Code Listing 36

```
public class ProjectAutomation : Automation
{
    public int Owner { get; private set; }
    public int AutomationType { get; private set; }

    public override void StartUp(int ownerID, int automationType)
    {
        Owner = ownerID;
        AutomationType = automationType;
        // Create the automation in the database.
    }

    public override async Task<bool> AutomationCompleted()
    {
        await Task.Delay(TimeSpan.FromMilliseconds(3000));
        // Read the completion flag in the database.
        return true;
    }

    public override async Task PurgeAutomation(int ownerID, DateTime
purgeDate)
    {
        await Task.Delay(TimeSpan.FromMilliseconds(3000));
    }
}
```

Interfaces

This e-book wouldn't be complete without us taking a look at interfaces. If you guessed that you can't define interface methods as asynchronous, you guessed correctly. The same applies to interfaces as for abstract classes. The method of creating async methods in the derived class is identical to the method we followed for abstract classes.

Let's create a new class file called **IHydratable.cs**. This interface will mark the automation object as able to rehydrate itself from the database.

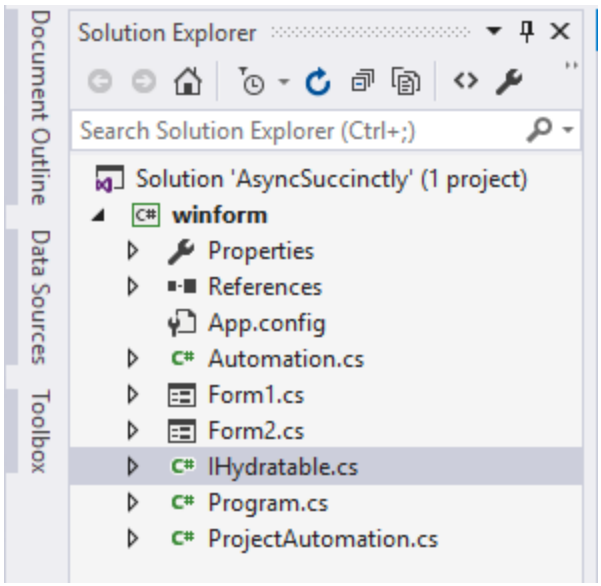


Figure 15: Add Interface to Solution

You will need to add the **`System.Threading.Tasks`** namespace to your interface **`IHydratable`**.

Code Listing 37

```
using System.Threading.Tasks;
```

As you can see, Code Listing 38 will define your interface simply by specifying that the **`RehydrateAutomation()`** method returns a type of **`Task<int>`**.

Code Listing 38

```
namespace AsyncSuccinctly
{
    interface IHydratable
    {
        Task<int> RehydrateAutomation(int ownerID, int automationType);
    }
}
```

In order to implement the interface in our existing **`ProjectAutomation`** class, we add it after we have declared that the **`ProjectAutomation`** class inherits from the abstract class **`Automation`**.

Code Listing 39

```
public class ProjectAutomation : Automation, IHydratable
```

The **`RehydrateAutomation()`** method is added to the end of the **`ProjectAutomation`** class, and we can then define it as an async method by adding the **`async`** and **`await`** keywords.

Code Listing 40

```
public class ProjectAutomation : Automation, IHydratable
{
    public int Owner { get; private set; }
    public int AutomationType { get; private set; }

    public override void StartUp(int ownerID, int automationType)
    {
        Owner = ownerID;
        AutomationType = automationType;
        // Create the automation in the database.
    }

    public override async Task<bool> AutomationCompleted()
    {
        await Task.Delay(TimeSpan.FromMilliseconds(3000));
        // Read the completion flag in the database.
        return true;
    }

    public override async Task PurgeAutomation(int ownerID, DateTime
purgeDate)
    {
        await Task.Delay(TimeSpan.FromMilliseconds(3000));
    }

    public async Task<int> RehydrateAutomation(int ownerID, int
automationType)
    {
        Owner = ownerID;
        AutomationType = automationType;
        await Task.Delay(TimeSpan.FromMilliseconds(3000));
        // Read the purged automation in the database and return the
record ID.
        return 0;
    }
}
```

Note: The `RehydrateAutomation()` method does not do anything but return an integer of zero. All the methods in this class await on Task.Delay to mimic the process of a long-running task.

Chapter 3 Some Real World Examples

Displaying the progress of an async method

All developers should consider developing asynchronous code in order to make their applications more responsive. Rather than adding async code as an afterthought, consider developing your code with async methods baked in from the start. This makes it easier to pick up issues and design problems.

One feature worth considering is having the ability to report the progress of an async method to the user. To the user, the async method is not obvious because they can't see (or understand) the code. In all fairness, they should not need to see it. But one thing the user does see or feel is an app that doesn't seem to be doing anything.

Whenever your application performs a host of long-running tasks that you have made asynchronous, you should probably consider notifying the user of the progress of those tasks. The standard Windows Progress bar control has been around for ages, and its implementation with async methods is easy. In order to allow the application to report its progress, you must use the **`IProgress`** interface.

The code listings here will illustrate how to accomplish this. I reckon that there are better ways to code this, but I want to demo a concept here. You can roll your own if you prefer.

First, you will need to create a new Windows Form. Add to this Windows Form a button control, a ProgressBar control, and a label control. This form will create a number of **`Student`** objects with a delay between each object before adding each to a **`List<Student>`** collection.

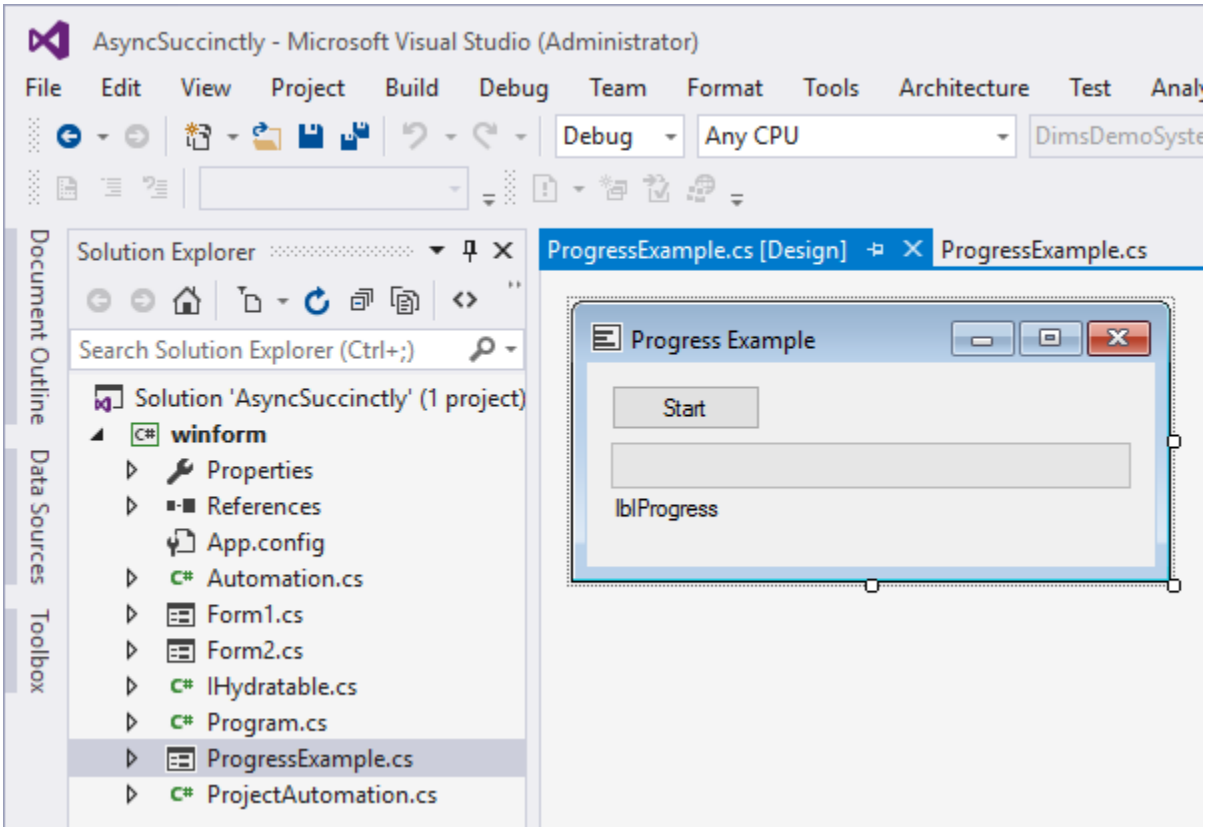


Figure 16: Progress Example Form Design

In the code-behind for the file **ProgressExample.cs**, import the following namespace **`System.Threading.Tasks`** if not already imported.

Code Listing 41

```
using System.Threading.Tasks;
```

Now we need to create a new class called Student. You can create a new class file called Student.cs or simply add it to the existing code at the bottom of your existing code file. Add the various properties needed (as in Code Listing 42 or add your own).

Code Listing 42

```
public class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string StudentNumber { get; set; }
    public int Age { get; set; }
    public string FullName() => FirstName + " " + LastName;
}
```

Code Listing 43 creates a Helper class with an extension method called **`ToPercent`**. This helper method will act on an integer value and calculate the percentage of the max value passed as a parameter that that integer occupies.

You will notice that I have taken a few liberties in the helper method, but I want to illustrate the progress bar update from an async method, not how to write an extension method. The comments in the helper method explain its purpose. For background reading on Extension Methods, see the following MSDN article: https://msdn.microsoft.com/en-us//library/bb383977.aspx.

Suppose that an integer value is the number 50. If I want to know what percentage of 250 the integer value of 50 is, I would call this extension method.

Code Listing 43

```
public static class Helper
{
    public static int ToPercent(this int value, int maxValue)
    {
        int retVal = 0;
        // value = 50
        // maxValue = 250
        // 250 / 50 = 5
        // 100% / 5 = 20%
        double factor = 0.0;
        if (value != maxValue)
        {
            factor = (maxValue / (value + 1));
            retVal = Convert.ToInt32(100 / factor);
        }
        else
            retVal = 100;

        return retVal;
    }
}
```

Next, let's write the async method. You will notice that the async method returns a **`List<Student>`** object. One of the parameters to this async method is the progress of type **`IProgress<T>`** where **`T`** is the type of progress on which you want to report. In our case, we will be returning an integer count for the progress.

Note that we're simply creating a number of **`Student`** objects. The number created is passed to the async method as a parameter. Assume that we only want to return the first 250 **`Student`** objects. As each **`Student`** object is created (with dummy data), we perform a small delay.

Just after we add the **`Student`** object to the **`List<Student>`** collection, we report the progress back to the progress bar. Therefore, the value of **`i`** being reported to the progress bar is the iterator count.

Code Listing 44

```
public async Task<List<Student>> GetStudents(IProgress<int> progress, int
iRecordCount)
{
    // Read students from database.
    // Return only the first iRecordCount entries (e.g. 250).

    List<Student> oStudents = new List<Student>();
    for (int i = 0; i < iRecordCount; ++i)
    {
        await Task.Delay(100);
        Student oStudent = new Student()
        {
            FirstName = "Name" + i,
            LastName = "LastName" + i,
            Age = 20,
            StudentNumber = "S203237" + i
        };

        oStudents.Add(oStudent);
        if (progress != null)
            progress.Report(i);
    }
    return oStudents;
}
```

Next, we add the following code in the click event handler of the button on the form. As we've seen earlier, we are assuming a maximum count of 250 entries. This is where things get interesting. The progress bar contains a maximum of 100 percent. We will therefore need to calculate the percentage of the maximum of 250 that reflects the iterator count. This is why we need the extension method.

Code Listing 45

```
private async void btnStartProgress_Click(object sender, EventArgs e)
{
    int iMaxRecordToRead = 250;
    progressBar.Maximum = 100;

    var progress = new Progress<int>(percentComplete =>
    {
        int perc = percentComplete.ToPercent(250);
        progressBar.Value = perc;
        lblProgress.Text = $"Student records {perc}% processed";
    });

    List<Student> oStudents = await GetStudents(progress,
iMaxRecordToRead);

    lblProgress.Text = "Done!";
}
```

As the async method processes, the iterator count is reported to the progress bar, but not after it is calculated into the percentage part of 100%. Run your application and click **Start**.

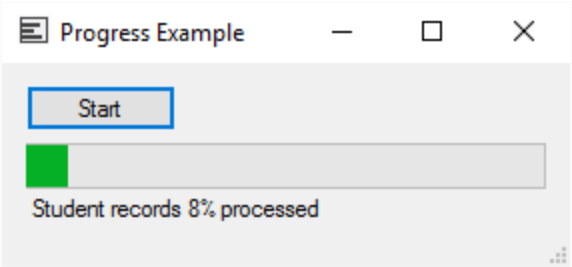


Figure 17: Progress Complete

Pausing the progress of an async method

Figure 18 shows the **`Task.Delay`** we have been using throughout this e-book in order to mimic a long-running task. Here, however, we are using it for its intended purpose—to pause a task.

We might need to pause a task because we need to retry a process until the user cancels or until the retry count has matured. Imagine for a minute that the process invoked by the **Start** button is a file download. This file, however, is versioned, and the application must always download a newer version than it previously downloaded.

If the file has not been updated, the application will retry until the user cancels or the retry count lapses. The application in the following code listings illustrates this logic. It contains a delay to pause the application between retries. It also contains a cancel button to cancel the process prematurely, and it updates the current progress to a label on the form.

Start by creating a Windows Form with a button and label control placed on it.

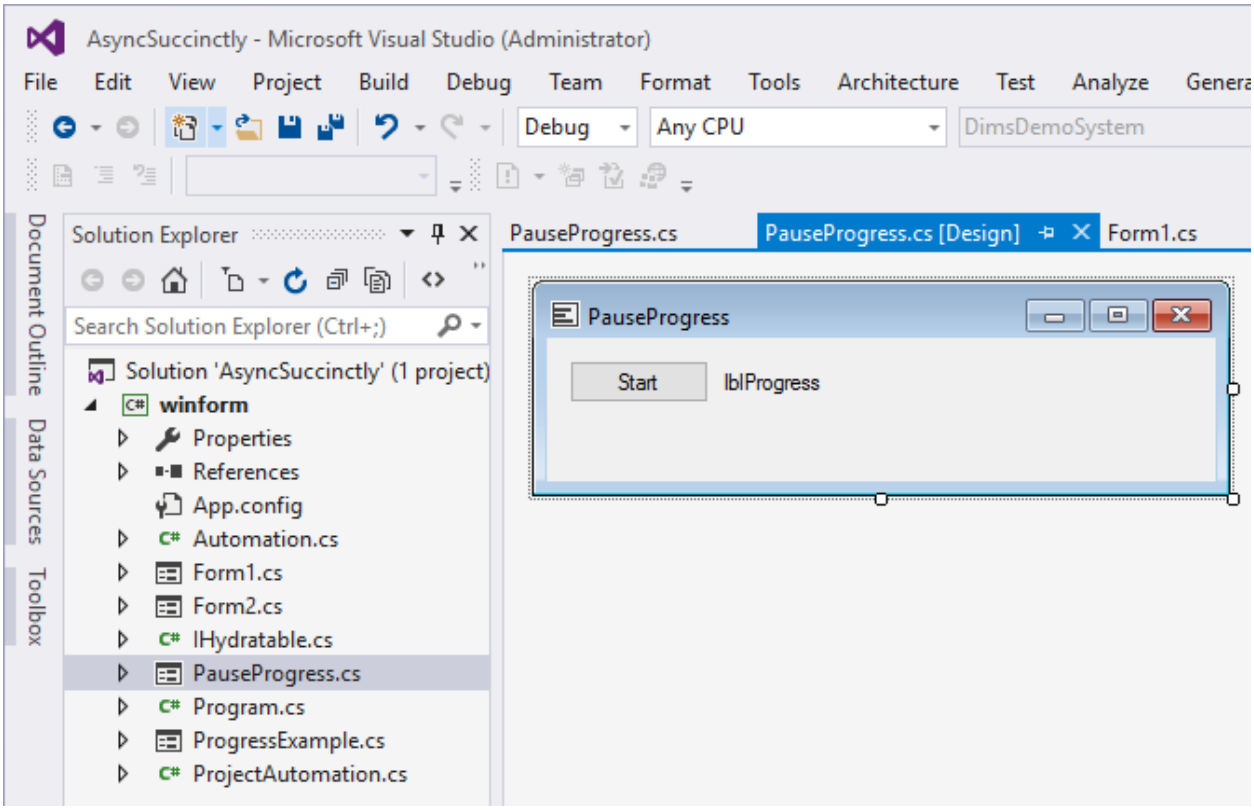


Figure 18: Pause Progress Form Design

Import the **`System.Treading.Tasks`** namespace.

Code Listing 46

```
using System.Threading.Tasks;
```

At the top of the Windows Form, add a **`CancellationTokenSource`** object so that it is in scope to the whole form.

Code Listing 47

```
public partial class PauseProgress : Form
{
    CancellationTokenSource cancelSource;
```

Next, add a method named **PerformTask** that tries to process the file. Pass arguments of type **`IProgress`** and **`CancellationToken`** to the async method. The **`cancel`** object will allow us to cancel the async process prematurely when the user clicks **Cancel**. The **`progress`** object is used to report the progress of the current async method. You will notice that the **`IProgress<T>`** in this instance takes a **`string`** as the type of **`T`**.

Code Listing 48

```
private async Task PerformTask(IProgress<string> progress,
CancellationToken cancel)
{
    UpdateProgress(progress, "Started Processing...");
    bool blnTaskCompleted = false;
    int iDelaySeconds = 0;
    while (!blnTaskCompleted)
    {
        iDelaySeconds += 2;
        await Task.Delay((iDelaySeconds * 1000), cancel);
        // Retry long-running task.
        if (iDelaySeconds >= 10)
        {
            blnTaskCompleted = true;
            UpdateProgress(progress, "Process completed");
        }
        else
            UpdateProgress(progress, $"Process failed. Retrying in
{iDelaySeconds} seconds...");
    }
}
```

Let's next add a method to update the progress in the label. Instead of adding the code all over the place, we can simply call the **`UpdateProgress`** method.

Code Listing 49

```
private void UpdateProgress(IProgress<string> progress, string message)
{
    if (progress != null)
        progress.Report(message);
}
```

Now, I must admit that I cheated a little with the **Start** button. When the **Start** button is clicked, it will change the text to **Cancel**. If the **Cancel** button is clicked, it will change the text to **Start**. Next, I check the button text, then I either start or cancel the async method.

You will notice that the **`CancellationTokenSource`** is instantiated when the Start button is clicked. I also create the process object and specify that it needs to return a string when updating the progress.

When the async method is cancelled, an **`OperationCancelledException`** is thrown, and we need to handle that in the try/catch.

Code Listing 50

```
private async void btnStart_Click(object sender, EventArgs e)
{
    if (btnStart.Text.Equals("Start"))
    {
        btnStart.Text = "Cancel";
        cancelSource = new CancellationTokenSource();

        try
        {
            var progress = new Progress<string>(progressReport =>
            {
                lblProgress.Text = progressReport;
            });

            await PerformTask(progress, cancelSource.Token);
        }
        catch (OperationCanceledException)
        {
            lblProgress.Text = "Processing Cancelled";
        }
    }
    if (btnStart.Text.Equals("Cancel"))
    {
        btnStart.Text = "Start";
        if (cancelSource != null)
            cancelSource.Cancel();
    }
}
```

Now, run the application. Click **Start** and notice how the text changes to **Cancel**. The async method starts delaying.

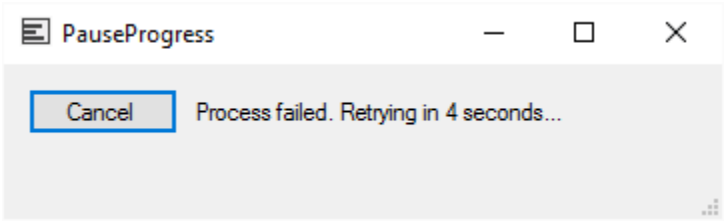


Figure 19: Delay before Retry

After a while, we assume that the file processing has been completed and that the application completed the async task.

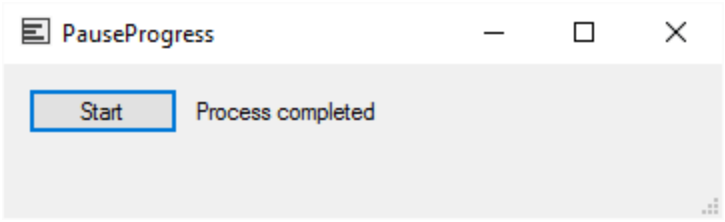


Figure 20: Process Completed

Run the application again. This time, however, click **Cancel** before the process can complete. Note that the **`Task.Delay`** is immediately cancelled, and the application completes processing, updating the status label to notify the user of the cancellation.

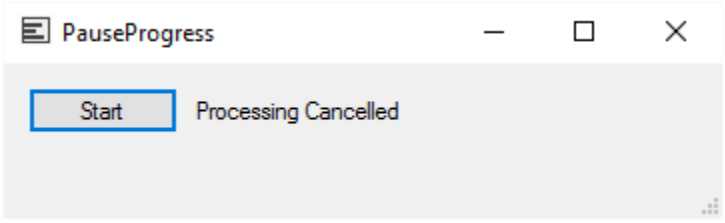


Figure 21: Process Cancelled

Using Task.WhenAll() to wait for all tasks to complete

When you need to run several async methods, remember that you must wait for all those methods to complete before you carry on. The **`Task.WhenAll()`** provides a perfect construct for developers that allows them to do just that.

The example that follows illustrates how to wait for three async methods that return nothing and three async methods that return integers. First, create a new Windows form and add a **TextBox** control with its **Multiline** property set to true.

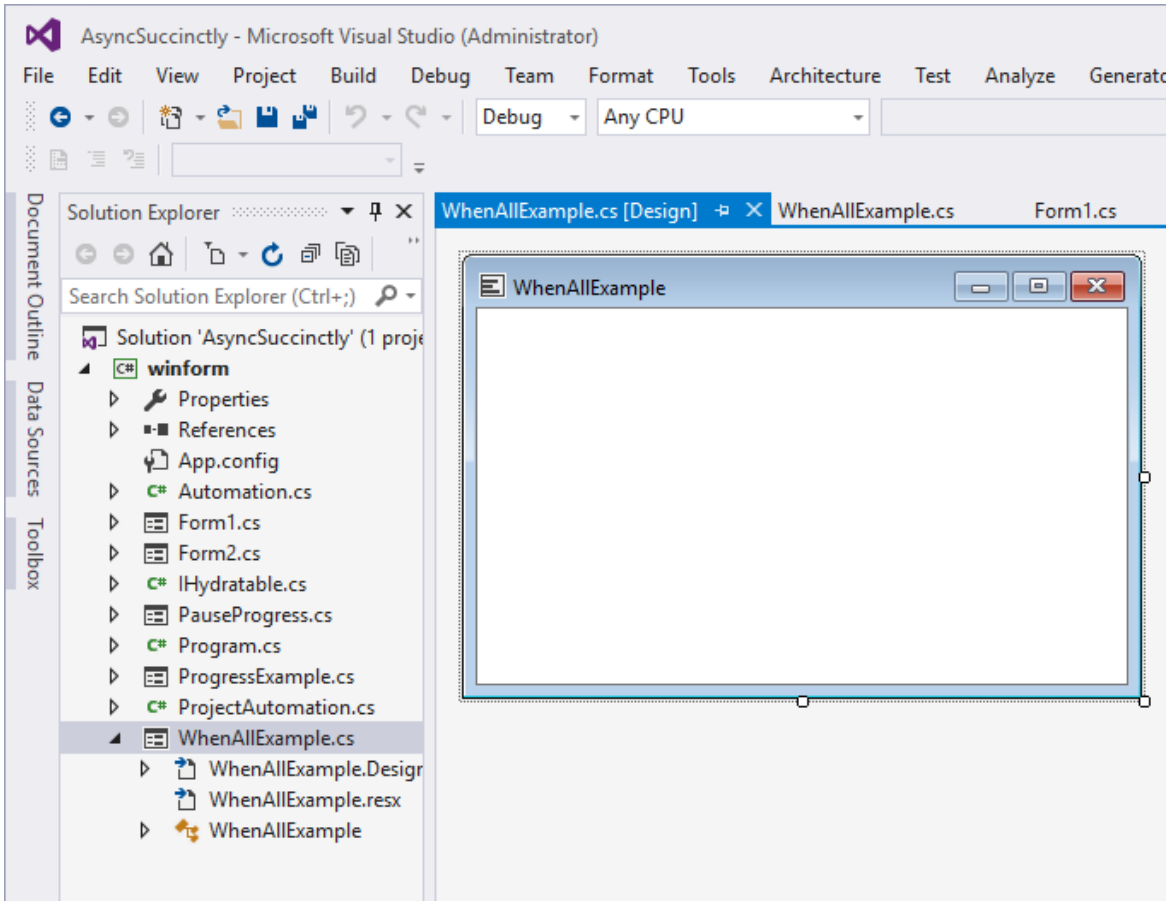


Figure 22: Form Designer for WhenAll

After you have set the **Multiline** property to **True**, set the **Dock** property to **Fill**.

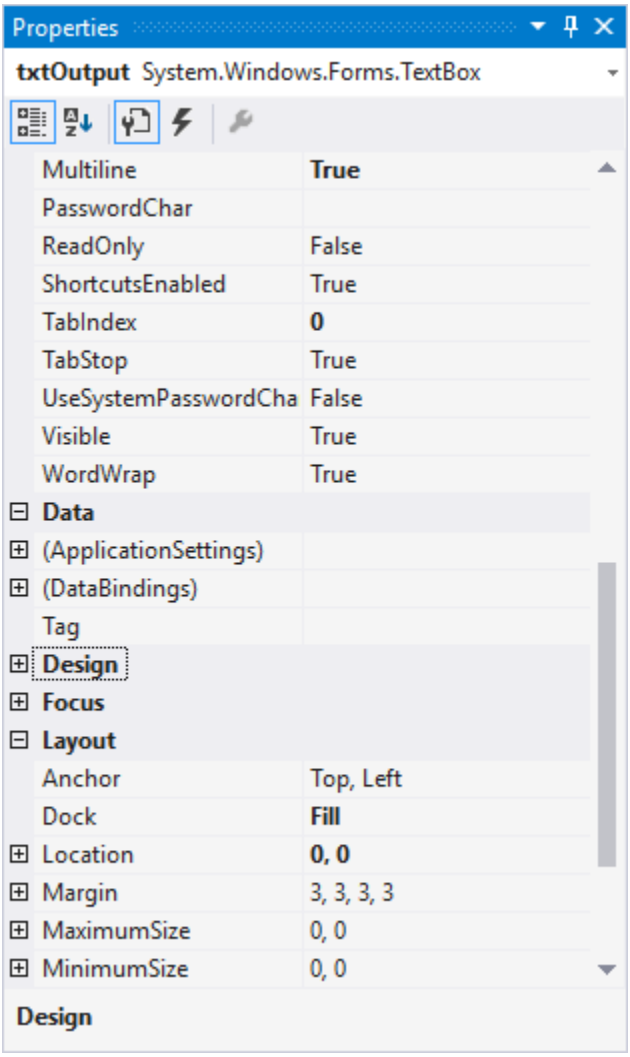


Figure 23: Form Properties

In the code view, ensure that you add the **`System.Threading.Tasks`** namespace to your form.

Code Listing 51

```
using System.Threading.Tasks;
```

Next, add the following method that simply appends the string argument passed.

Code Listing 52

```
private void Output(string val)
{
    txtOutput.AppendText("\r\n" + val);
}
```

Now, we will add three void async methods. Because these async methods return nothing, we will call the **`Output()`** method and pass the **`nameof(<methodName>)`** to be written to the textbox.

In each async method, we will delay each by one, two, and three seconds, respectively.

Code Listing 53

```
private async Task Delay1kms()
{
    await Task.Delay(1000);
    Output($"{nameof(Delay1kms)} completed");
}

private async Task Delay2kms()
{
    await Task.Delay(2000);
    Output($"{nameof(Delay2kms)} completed");
}

private async Task Delay3kms()
{
    await Task.Delay(3000);
    Output($"{nameof(Delay3kms)} completed");
}
```

Next, add three async methods that each return an integer value. As before, each method will delay for one, two, and three seconds, respectively.

Code Listing 54

```
private async Task<int> DoWorkA()
{
    await Task.Delay(1000);
    return 1;
}

private async Task<int> DoWorkB()
{
    await Task.Delay(2000);
    return 2;
}

private async Task<int> DoWorkC()
{
    await Task.Delay(3000);
    return 3;
}
```

Lastly, you must add the code in Code Listing 55 to the **`Form1_Load`** event handler method. Then you need to **`await Task.WhenAll()`** and pass it the three void async methods that delay and output the text to the textbox.

You will see each async method name output to the textbox before seeing the “DelayTasks Completed” text. This means that all three methods finished before continuing.

The second **`Task.WhenAll()`** will call the async methods that return the integer values. These are returned to an integer array and are output to the textbox.

Code Listing 55

```
private async void WhenAllExample_Load(object sender, EventArgs e)
{
    await Task.WhenAll(Delay1kms(), Delay2kms(), Delay3kms());
    Output("DelayTasks Completed");

    int[] iArr = await Task.WhenAll(DoWorkA(), DoWorkB(), DoWorkC());

    for (int i = 0; i <= iArr.GetUpperBound(0); i++)
    {
        Output(iArr[i].ToString());
    }
}
```

Next, run the application.

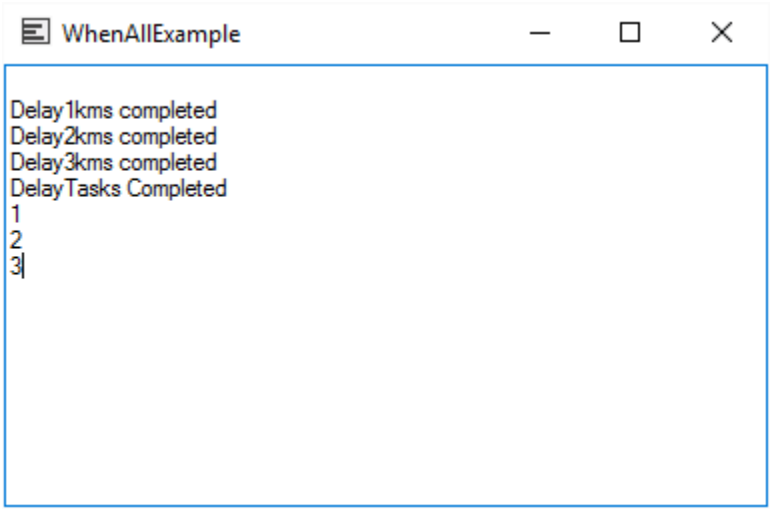


Figure 24: Run WhenAllExample

You will notice that as each void async method completes, the method name is output to the textbox. When all async methods have completed, the last delay message is output to the textbox. Lastly, the **`int`** returning async methods run and only output the array values after the methods have completed.

You can also write this code by adding the async methods to a **List<Task>** and **List<Task<int>>** collections. Then you pass these collections to the **Task.WhenAll()** method.

Code Listing 56

```
List<Task> oVoidTasks = new List<Task>();

oVoidTasks.Add(Delay1kms());

oVoidTasks.Add(Delay2kms());

oVoidTasks.Add(Delay3kms());

await Task.WhenAll(oVoidTasks);

Output("DelayTasks Completed");


List<Task<int>> oIntTasks = new List<Task<int>>();

oIntTasks.Add(DoWorkA());

oIntTasks.Add(DoWorkB());

oIntTasks.Add(DoWorkC());
int[] iArr = await Task.WhenAll(oIntTasks);

for (int i = 0; i <= iArr.GetUpperBound(0); i++)
{
    Output(iArr[i].ToString());
}
```

Running the application again, you'll see that the output remains the same as in the previous example. Using **Task.WhenAll()** is an efficient way to ensure that all the required async methods have completed before you continue your code.

Using Task.WhenAny() to wait for any tasks to complete

Sometimes you might need to call several async methods, but keep in mind that you only need the result from any one async method. This means that the first async method to complete will be used while the rest can be cancelled.

Think of it like accessing a couple of web services that return the same result from different sources. Any one of these results can be used in your code, but in order to optimize your application you need to use the fastest code possible. The calls to the web services don't consistently complete in the same amount of time, so there is no way for you to choose the fastest web service.

`Task.WhenAny()` offers you a solution—you can call them all asynchronously but only use the first method to return a result, then cancel the rest. Let's look at this following example that first calls **`Task`** async methods and second calls **`Task<T>`** async methods.

You will need to create a new Windows Form and add a textbox.

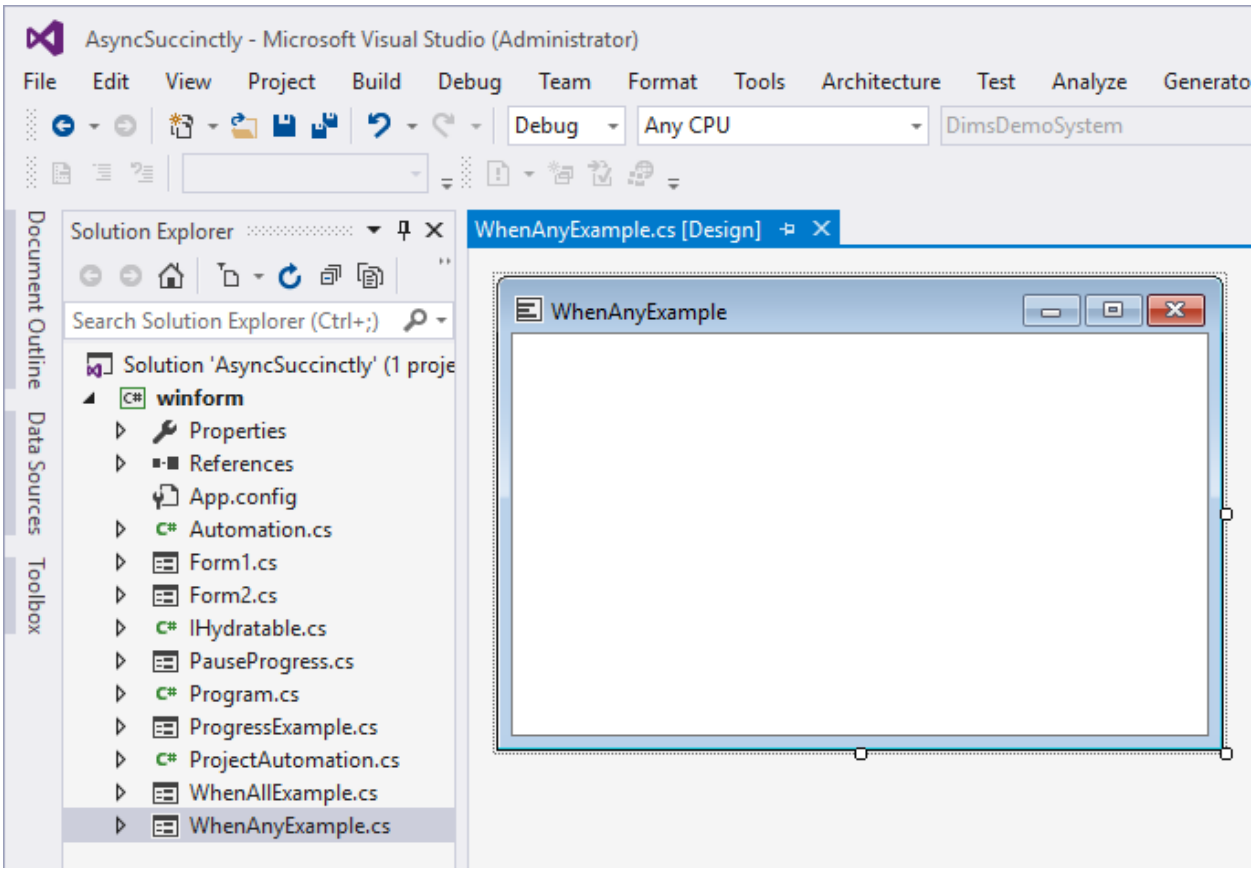


Figure 25: WhenAnyExample Form Designer

Next, select the textbox, then set the **Multiline** property to **True** and the **Dock** property to **Fill**.

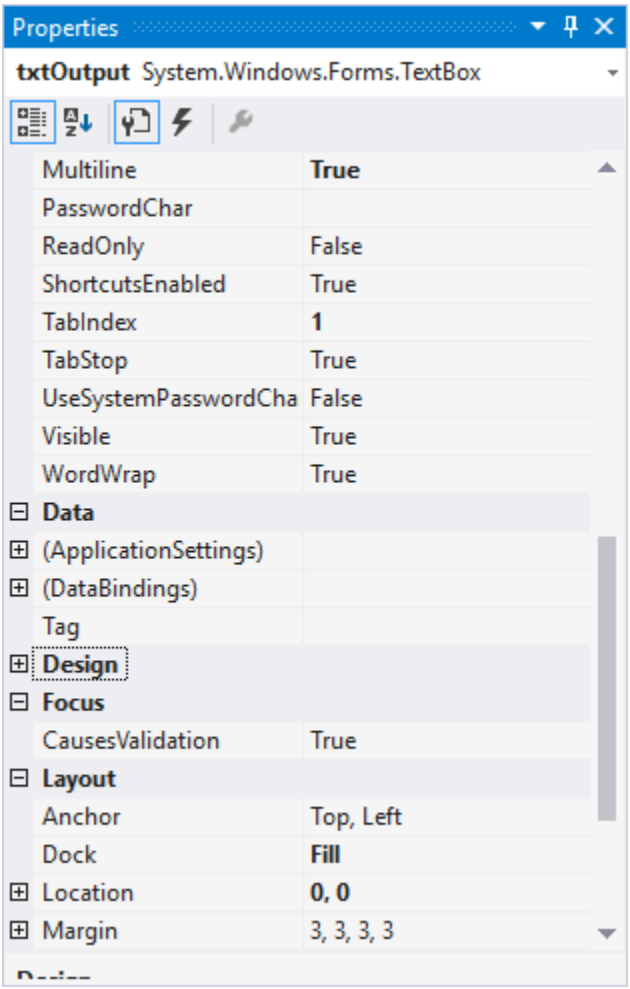


Figure 26: WhenAny Textbox Properties

In the code-behind, add the **`System.Threading`** and **`System.Threading.Tasks`** namespace to your form.

Code Listing 57

```
using System.Threading;
using System.Threading.Tasks;
```

Add a **`CancellationTokenSource`** object to your form with the scope visible to the entire form.

Code Listing 58

```
public partial class WhenAnyExample : Form
{
    CancellationTokenSource cancelSource;
```

Add three **async Task** methods and add **Task.Delay** to each async method with different delays. As parameter, pass the **CancellationToken**, then pass that to the **Task.Delay** method. Lastly, output the name of the method that first completes the task.

Code Listing 59

```
private async Task Delay1kms(CancellationToken cancel)
{
    await Task.Delay(1000, cancel);
    Output($"{nameof(Delay1kms)} completed first");
}

private async Task Delay2kms(CancellationToken cancel)
{
    await Task.Delay(2000, cancel);
    Output($"{nameof(Delay2kms)} completed first");
}

private async Task Delay3kms(CancellationToken cancel)
{
    await Task.Delay(3000, cancel);
    Output($"{nameof(Delay3kms)} completed first");
}
```

Create the **Output()** method and set the **txtOutput** textbox equal to the **val** parameter.

Code Listing 60

```
private void Output(string val)
{
    txtOutput.AppendText("\r\n" + val);
}
```

Next, change the Form Load event handler method to an async method, then instantiate the **CancellationTokenSource** object. You will now call the **Task.WhenAny()** method and pass it each **Task** async method. Notice that each method is passed the **cancelSource.Token** object. The first async method that completes will allow the **cancelSource** object to be cancelled, which effectively stops the remaining async methods—you don't want to have code still running when you aren't going to use the result. Wrap this code in a **try catch** block because the cancel will throw an **OperationCanceledException**.

Code Listing 61

```
private async void WhenAnyExample_Load(object sender, EventArgs e)
{
    cancelSource = new CancellationTokenSource();

    try
    {
        await Task.WhenAny(Delay1kms(cancelSource.Token),
Delay2kms(cancelSource.Token), Delay3kms(cancelSource.Token));
        if (cancelSource != null)
            cancelSource.Cancel();
    }
    catch (OperationCanceledException)
    {

    }
    catch (Exception)
    {

    }

    cancelSource = null;
}
```

Run your application and you will see that the fastest async method completes first and calls the **`Output()`** method. The remaining async methods are cancelled, so they output nothing.

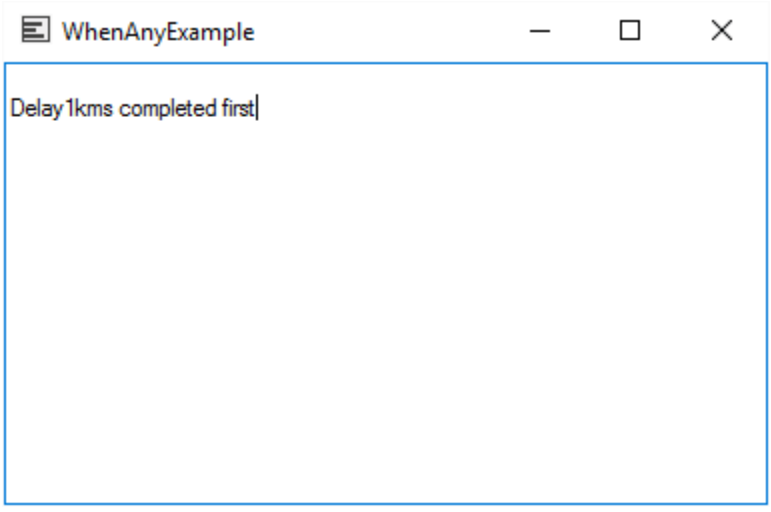


Figure 27: WhenAny Example Task Async Method

Let's now add async methods that return **`Task<TResult>`**. These also take the **`CancellationToken`** as parameter that is passed to the **`Task.Delay()`**. Each async method delays for a different duration before calling the **`Output()`** method. Each then returns the **`int`** result.

Code Listing 62

```
private async Task<int> DoWorkA(CancellationToken cancel)
{
    await Task.Delay(3500, cancel);
    Output($"{nameof(DoWorkA)} completed first");
    return 1;
}

private async Task<int> DoWorkB(CancellationToken cancel)
{
    await Task.Delay(2800, cancel);
    Output($"{nameof(DoWorkB)} completed first");
    return 2;
}

private async Task<int> DoWorkC(CancellationToken cancel)
{
    await Task.Delay(1900, cancel);
    Output($"{nameof(DoWorkC)} completed first");
    return 3;
}
```

Modify your load method to call the **`Task.WhenAny()`** method passing the **`DoWorkA()`**, **`DoWorkB()`**, and **`DoWorkC()`** methods. Each of these takes the **`cancelSource.Token`** object as parameter. The **`Task<int>`** result is returned and the rest of the async methods are cancelled. After that, we simply pass the returned value to the **`Output()`** method.

Code Listing 63

```
private async void WhenAnyExample_Load(object sender, EventArgs e)
{
    cancelSource = new CancellationTokenSource();

    try
    {
        Task<int> firstTask = await
Task.WhenAny(DoWorkA(cancelSource.Token), DoWorkB(cancelSource.Token),
DoWorkC(cancelSource.Token));
        if (cancelSource != null)
            cancelSource.Cancel();

        Output(firstTask.Result.ToString());
    }
    catch (OperationCanceledException)
    {

    }
    catch (Exception)
    {

    }

    cancelSource = null;
}
```

Run your application again and you will see that the **`DoWorkC()`** async method completed first, returning a value of **`3`**. The remaining async methods are cancelled immediately.

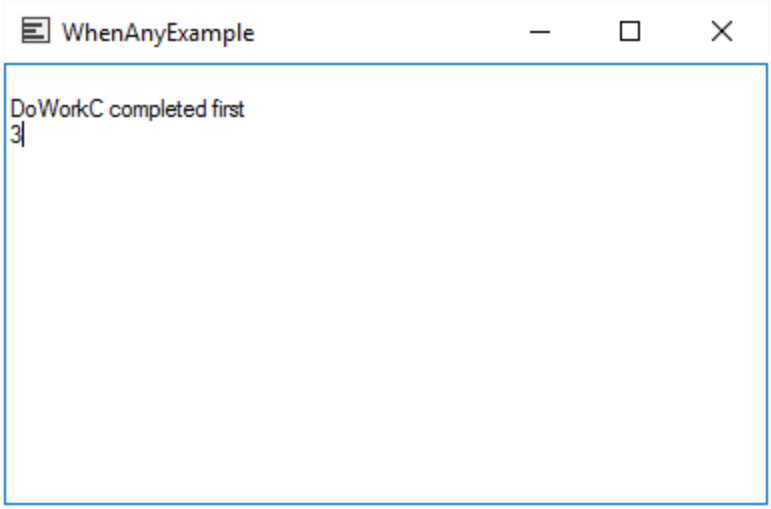


Figure 28: WhenAny Example Task<T> Async Method

This way of using what you need and cancelling the rest of the async methods is very useful when you want to take a ‘first come, first served’ approach.

Process tasks as they complete

The example in the following code listings illustrates how to process tasks as they complete. Sometimes you might be in the position of needing to process all tasks while also needing to continue some other process while they complete.

In our example, we are reading the size of the HTML returned from three websites. As each one completes, the URL is loaded into a WebBrowser control. Therefore, we can logically expect that the smallest site will be loaded first, then the second largest, and finally the largest. This is one of those examples that is quite tough to explain in words but is pretty easy to understand when you create the code and run the application. So, if the explanations below don’t fully make sense, just try reading the code and running the application.

First, create a new Windows Form that resembles the design in Figure 29.

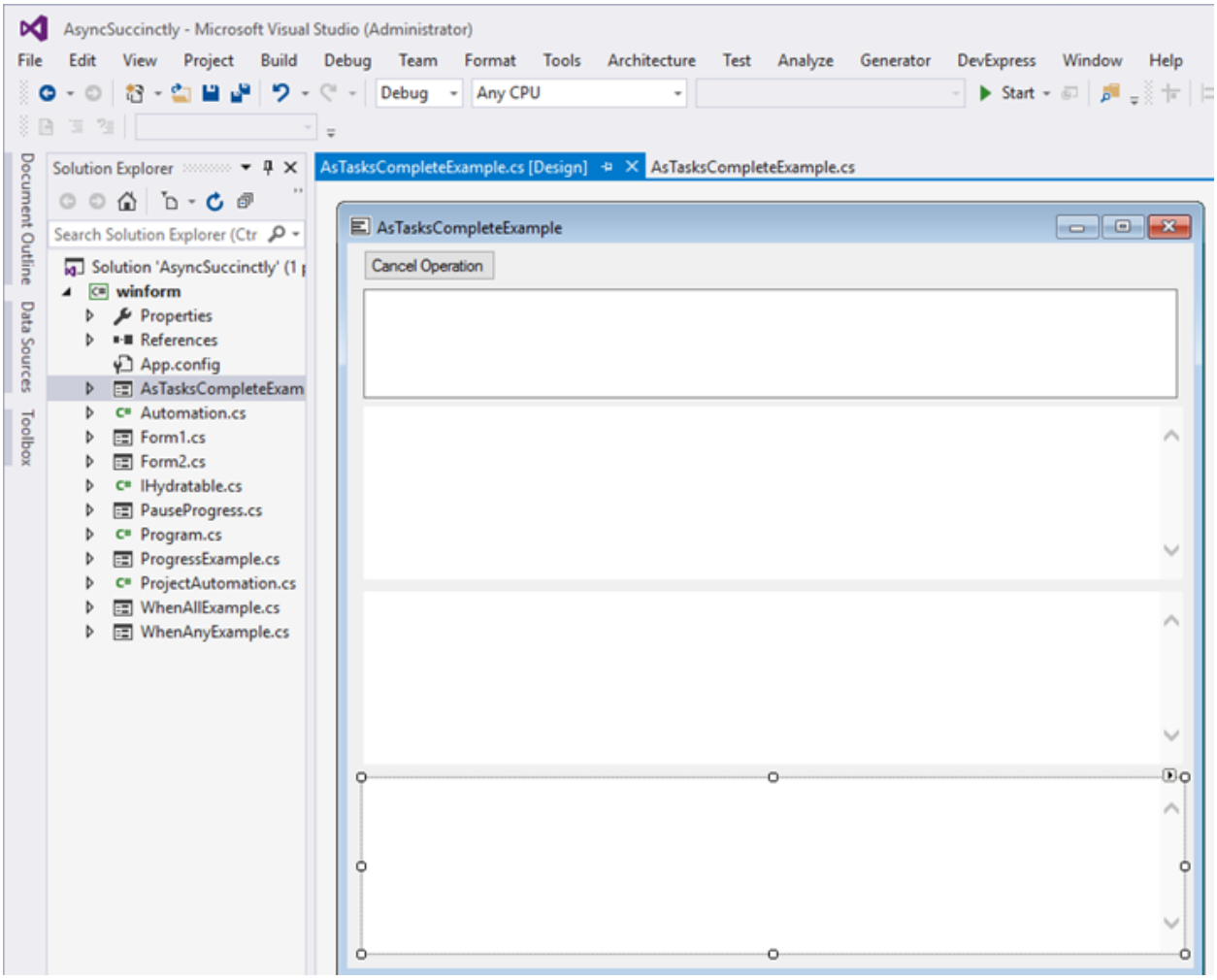


Figure 29: AsTasksCompleteExample Form Designer

Add a button to the Windows Form and call it **Cancel Operation**. Then add a textbox and set the **Multiline** property to **True**. Lastly, add three WebBrowser controls to the form and call them **site1**, **site2** and **site3**.

Be sure to add the **`System.Net.Http`**, **`System.Threading`** and **`System.Threading.Tasks`** namespaces to the code-behind.

Code Listing 64

```
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;
```

Right below the class declaration, add a **`CancellationTokenSource`** object and a **`List<string>`** object to the form so that they are visible to the entire form (globally scoped). We will be using the **`CancellationTokenSource`** object to cancel the async processing if that becomes necessary. The **`CancellationTokenSource`** object will provide a cancellation token via the **`Token`** property and send a cancellation message to the async method.

The **`List<string>`** will contain the three website URLs that we want to load into the WebBrowser controls **site1**, **site2**, and **site3**.

Code Listing 65

```
public partial class AsTasksCompleteExample : Form
{
    CancellationTokenSource cancelSource;
    List<string> urlList;
```

Create an event handler for the cancel button that will set the **`CancellationTokenSource`** object to a cancelled state. Because it is scoped globally, all the async methods that use the **`CancellationTokenSource`** object will receive the cancellation token.

Code Listing 66

```
private void btnCancel_Click(object sender, EventArgs e)
{
    if (cancelSource != null)
        cancelSource.Cancel();
}
```

Next, we need to add the code to load the list of URLs when the form loads. Note that the URLs are in reverse alphabetical order. The last URL (alphabetically) will be processed first, and so on. Also make sure that you add the async keyword to the form load event handler as **`private async void AsTasksCompleteExample_Load`**.

We next call the async method **AccessWebAsync()**, passing it the **CancellationTokenSource** object that was initialized on the first line of code in the form load event. Note that the code is wrapped in a **try catch** event handler. This is done because the cancellation of an asynchronous method will throw an **OperationCancelledException**.

Code Listing 67

```
private async void AsTasksCompleteExample_Load(object sender, EventArgs
e)
{
    cancelSource = new CancellationTokenSource();
    txtResults.Text = "Loading Sites";
    urlList = new List<string>
    {
        "http://www.wikipedia.com",
        "http://www.google.com",
        "http://www.apple.com"
    };

    try
    {
        await AccessWebAsync(cancelSource.Token);
        txtResults.Text += "\r\nProcessing done.";
    }
    catch (OperationCanceledException)
    {
        txtResults.Text += "\r\nProcessing canceled.";
    }
    catch (Exception)
    {
        txtResults.Text += "\r\nError processing.";
    }

    cancelSource = null;
}
```

The **AccessWebAsync()** method does several important things. It processes the URL list, returning a collection of **Task<string>** objects into the **taskCollection** variable. It then creates a **List<Task<string>>** object that contains the process URLs. Each **Task<string>** object will contain a pipe delimited string as **[siteLength] | [url]** that will be split later on in the **AccessWebAsync()** method.

We also need to create a **List<string>** object to contain the names of the WebBrowser controls on the form. Here, you can change the code and make it more generic. You could, for example, create a method that loops through all the controls on the Windows Form and finds only WebBrowser controls, ordering them by the integer value added to the end of the control name. Then you could add any number of WebBrowser controls to the form without hard coding the **List<string> siteControls** object. However, for the purposes of this example, I have simply hard coded the three WebBrowser controls to the list using the **nameof** keyword to return the string representation of the control name into the list.

We next need to loop through the **downloadTasks** list and **await** the processing. As soon as one of those completes, we continue by calling **Task.WhenAny()**. Then we remove that task from the list so that we don't process it again, and we split the returned value into the array **arrVal**. We use this same logic for the **siteControls** list. It gets the first WebBrowser control name in the **List<string> siteControls** object that holds these control names in alphabetical order as **site1**, **site2**, and **site3**. As soon as we get the first WebBrowser control name, we remove it from the list. We do this so that we don't overwrite previously loaded WebBrowser controls with a different URL returned from the **downloadTasks** list.

The very last lines of code write the output to our multiline textbox and load the processed sites URL into the appropriate WebBrowser control. It is here that the first task that has finished processing will be loaded into the first WebBrowser control, then the second, and finally the last, which will be loaded into the third WebBrowser control.

Code Listing 68

```
private async Task AccessWebAsync(CancellationToken cancel)
{
    HttpClient httpClnt = new HttpClient();

    // Get a collection of tasks.
    IEnumerable<Task<string>> taskCollection =
        from url in urlList select ProcessURLList(url, httpClnt, cancel);

    // Get a list of Tasks.
    List<Task<string>> downloadTasks = taskCollection.ToList();
    List<string> siteControls = new List<string>() { nameof(site1),
nameof(site2), nameof(site3) };
    // Process each task for each site until none are left.
    while (downloadTasks.Count > 0)
    {
        string strSite = "";
        // Identify the first task that completes.
        Task<string> firstFinishedTask = await
Task.WhenAny(downloadTasks);
        strSite = siteControls.First();
        // Remove so that you only process once.
        downloadTasks.Remove(firstFinishedTask);
        siteControls.Remove(strSite);
        // Await the completed task.
        string strValue = await firstFinishedTask;
        string[] arrVal = strValue.Split('|');

        txtResults.Text += $"\r\nDownload length of {arrVal[1]} is
{arrVal[0]}";
        await LoadBrowserControl(strSite, arrVal[1]);
    }
}
```

The **`ProcessURLList()`** method processes the URL to return the HTTP content of the website as a string and then get the length of the string. That is concatenated to the URL to return the pipe delimited **`Task<string>`** object for processing by the calling code.

Code Listing 69

```
private async Task<string> ProcessURLList(string url, HttpClient cl,
CancellationToken cancel)
{
    // Get the Task<HttpResponseMessage> object asynchronously.
    HttpResponseMessage resp = await cl.GetAsync(url, cancel);

    // Serialize the HTTP content to a string asynchronously.
    string strSite = await resp.Content.ReadAsStringAsync();
    // Return the string.Length|www.url.com
    return strSite.Length + "|" + url;
}
```

Lastly, we create the method that loads the URL into the appropriate WebBrowser control on the form. Because we added the WebBrowser control names into the **`siteControls`** list by using the **`nameof`** operator (new to C# 6.0), we have a string representation of the control name. That means we can pass that string variable to the **`LoadBrowserControl()`** method and switch on it. This will set the correct WebBrowser control based on which **`Task<string>`** object from the **`downloadTasks`** list completes first, second, and third.

Code Listing 70

```
private async Task LoadBrowserControl(string controlName, string strUrl)
{
    Uri url = new Uri(strUrl);
    switch (controlName)
    {
        case nameof(site1):
            site1.Url = url;
            break;

        case nameof(site2):
            site2.Url = url;
            break;

        case nameof(site3):
            site3.Url = url;
            break;

        default:
            break;
    }

    await Task.CompletedTask;
}
```

When you have added all the code, build and run your application.

Note: We are calling Task.CompletedTask because the method is called asynchronously but doesn't actually return anything or run any asynchronous methods.

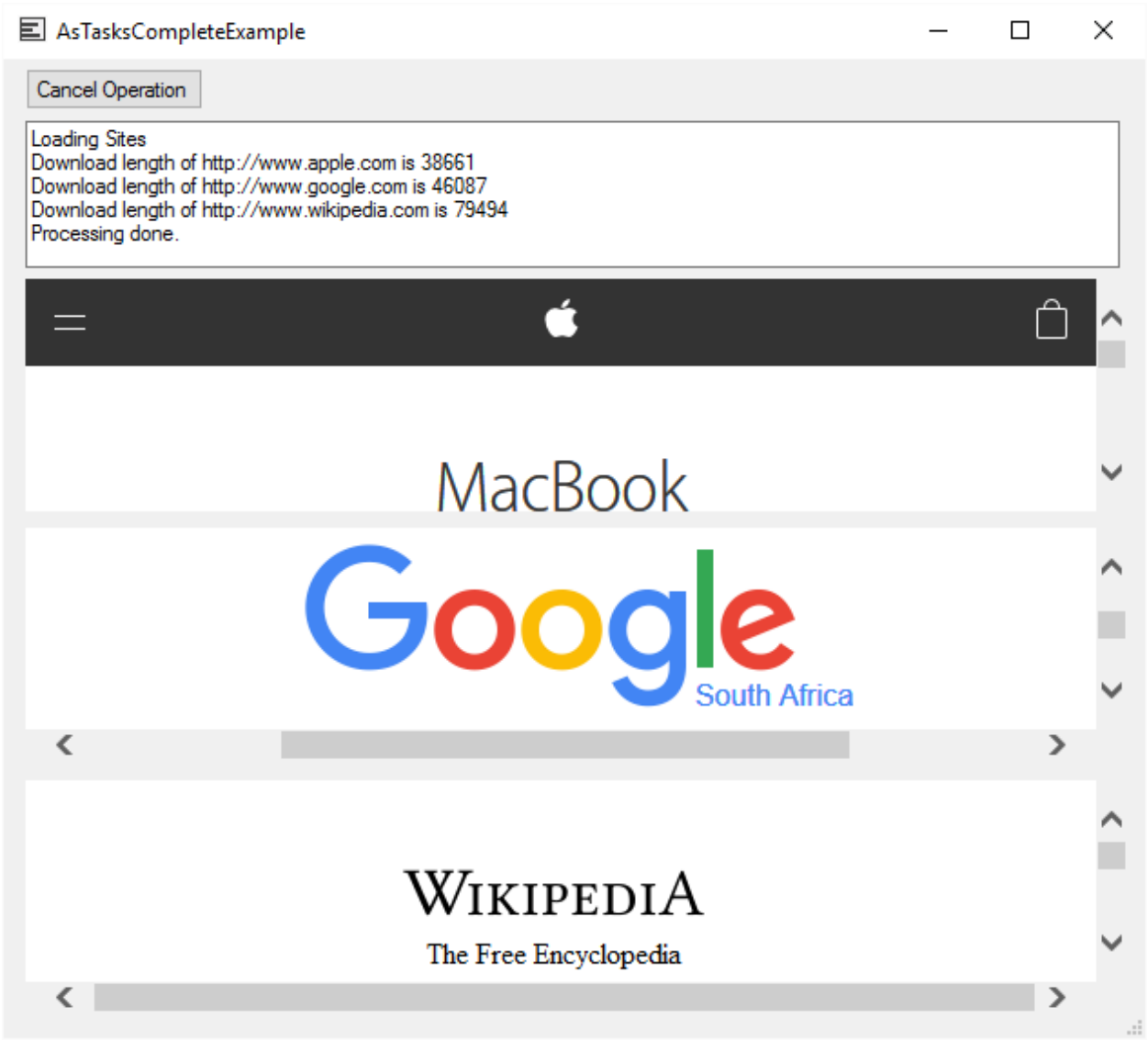
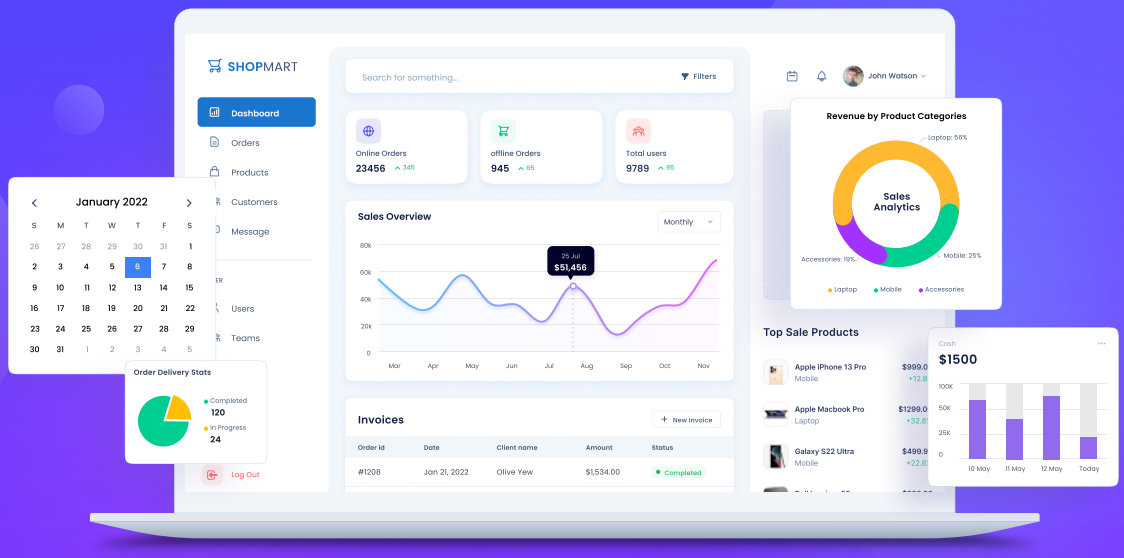


Figure 30: Websites Loaded

When the form loads, you will see that as the processed URLs are added to the multiline textbox control, the WebBrowser controls are loaded with the processed URLs. This is all done in the order that each completes processing (as opposed to the order in which each started).

THE WORLD'S BEST UI COMPONENT SUITE FOR BUILDING POWERFUL APPS

4.6 out of 5 stars



GET YOUR FREE .NET AND JAVASCRIPT UI COMPONENTS

syncfusion.com/communitylicense

- 1,700+ components for mobile, web, and desktop platforms
- Support within 24 hours on all business days
- Uncompromising quality
- Hassle-free licensing
- 28000+ customers
- 20+ years in business

Trusted by the world's leading companies

Syncfusion

Chapter 4 Use SemaphoreSlim to Access Shared Data

Sometimes you need to access shared data safely because other areas in your application (or other applications) might read that data asynchronously.

We need to ensure that no other area of code is reading or writing to the specific section of code we are working on. Here we use the **`SemaphoreSlim`** object that has been extended in the .NET Framework 4.5 in order to include asynchronous methods.

The following example illustrates how existing code that does not implement **`SemaphoreSlim`** reads a shared data source and returns incorrect results. We will then add **`SemaphoreSlim`** to the code to ensure that no other code accesses the data source while we are busy with it.

Let's create a new Windows Form containing a multiline textbox to display output.

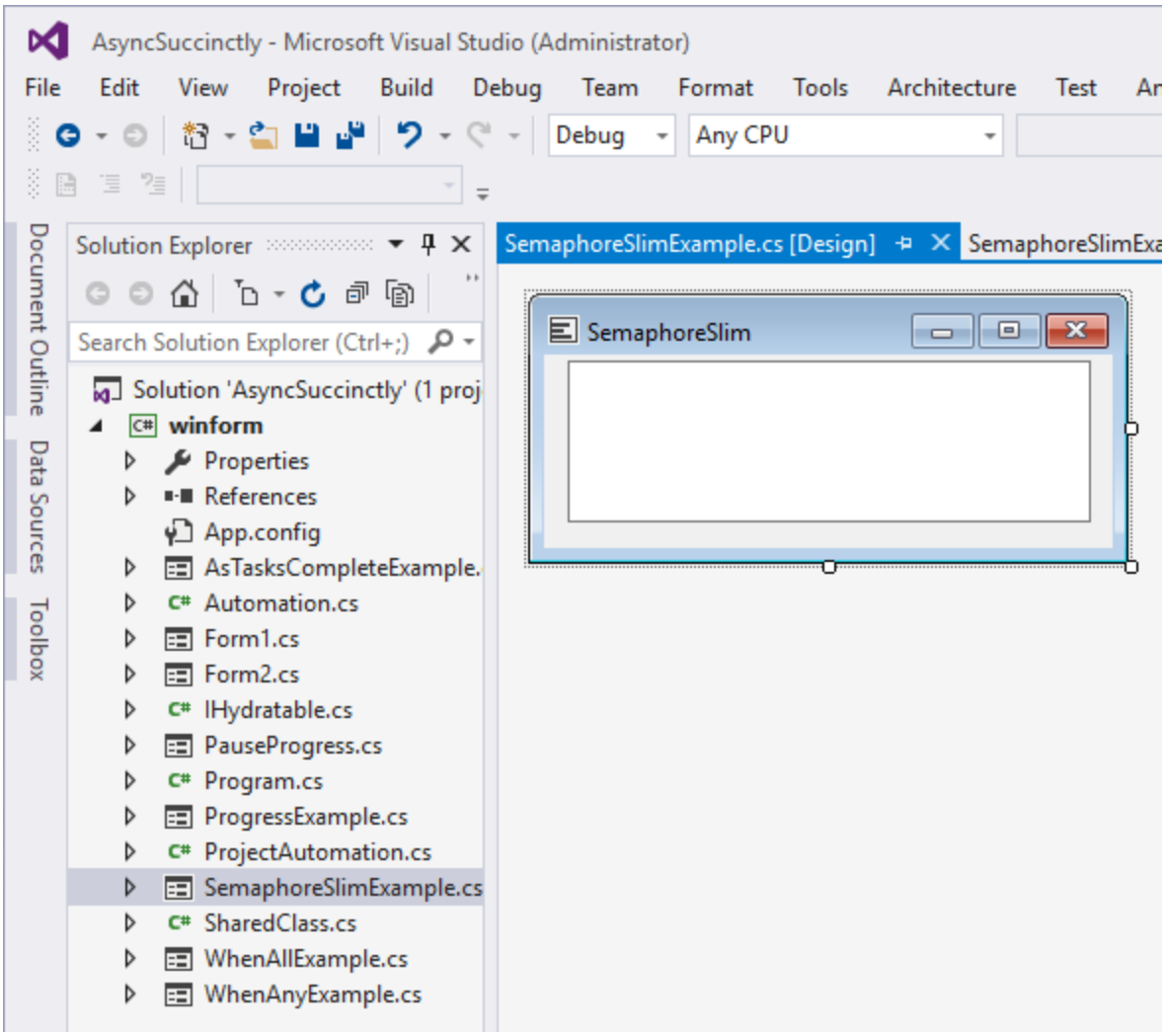


Figure 31: SemaphoreSlim Example Form Designer

Add a global variable to the code-behind that initializes the records to process.

Code Listing 71

```
public partial class SemaphoreSlimExample : Form
{
    int recordsToProcess = 0;
```

This form's constructor will need to be modified in order to accept two arguments. The first argument is **`formNumber`**, which displays the sequence of this form's creation. The second argument is an integer of the records to read. We are going to open two instances of this Windows Form. Next, each instance will call the same shared data source that is nothing more than a static class.

Code Listing 72

```
public SemaphoreSlimExample(int formNumber, int readRecords)
{
    InitializeComponent();
    recordsToProcess = readRecords;
    this.Text += $" {formNumber}";
}
```

In the form load, add the code to write the output to the textbox on the form and to call the static class called **SharedClass**. We pass the record count we want to read to the static method **AccessSharedResource()**.

Code Listing 73

```
private async void SemaphoreSlimExample_Load(object sender, EventArgs e)
{
    txtOutput.AppendText($"Records already processed =
{SharedClass.RecordsProcessed}");
    txtOutput.AppendText($"\r\nUpdate database processed field =
{SharedClass.RecordsProcessed}");
    txtOutput.AppendText($"\r\nRead the next {recordsToProcess}
records");
    await SharedClass.AccessSharedResource(recordsToProcess);
    txtOutput.AppendText($"\r\nRecords processed =
{SharedClass.RecordsProcessed}");
}
```

Now, we need to create the shared resource that is the static class. Ensure that the **System.Data** and **System.Threading.Tasks** namespaces have been imported.

Code Listing 74

```
using System.Data;
using System.Threading.Tasks;
```

At the very top of the class, add an auto-implemented property called **RecordsProcessed**. Set the default value to **0**.

Code Listing 75

```
public static class SharedClass
{
    public static int RecordsProcessed { get; private set; } = 0;
```

We now need to create the **public static async Task AccessSharedResource()** method. This reads a data table and sets the **RecordsProcessed** property to the records returned.

Code Listing 76

```
public static async Task AccessSharedResource(int readRecords)
{
    DataTable dtResults = await ReadData(readRecords);
    RecordsProcessed += dtResults.Rows.Count;
}
```

The last method is simply setup code that creates a data table with the number of rows returned and a delay. You can simply return an integer value here if you don't want to go through all the effort of creating the data table. My purpose here is illustrate that some data work is being performed and awaited on.

Code Listing 77

```
private static async Task<DataTable> ReadData(int readRecords)
{
    DataTable dtResults = new DataTable();
    dtResults.Columns.Add("ID");

    try
    {
        for (int row = 0; row <= readRecords - 1; row++)
        {
            DataRow dr = dtResults.NewRow();
            dtResults.Rows.Add(dr);
        }
    }
    catch (Exception ex)
    {
        throw;
    }

    await Task.Delay(3000);
    return dtResults;
}
```

Lastly, we need to add two calls to the **SemaphoreSlimExample** form. We pass it the number from which the form was created and the number of records to be read from the database.

Code Listing 78

```
private void btnSemiphoreSlim_LinkClicked(object sender,
LinkLabelLinkClickedEventArgs e)
{
    SemaphoreSlimExample sslm = new SemaphoreSlimExample(1, 100);
    sslm.Show();

    SemaphoreSlimExample sslm2 = new SemaphoreSlimExample(2, 25);
    sslm2.Show();
}
```

Build your application and run it. Two instances of the same form are displayed and the form load methods are called. You will notice that both forms begin writing text to the output immediately. The first form reports that there are **0** records read. We can assume that it next updates a flag in the database with that value. Then the form reads the next **100** records and writes the records read to the output.

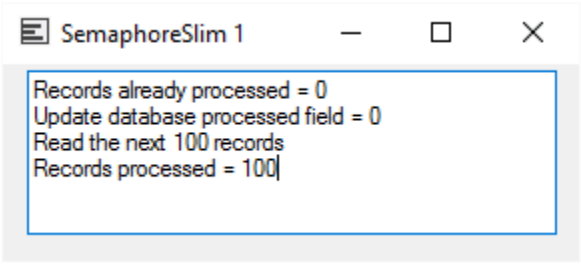


Figure 32: Form 1 Accessing Shared Resource

The second instance of the form does the same thing, but this time it reports that there are still **0** records already processed. This is incorrect, and this incorrect value is written back to the database. The form then reads the next 25 records. By the time the process completes, the first form has updated the static class's **RecordsProcessed** property.

This means that the correct total value of records processed is written to the database. However, for the second instance of the form, the values returned are totally out of sync. Imagine looking at this data in a database table and trying to figure out where the error is (because **0 + 25** is not **125**).

While we can identify the issue, keep in mind that we are dealing with only two forms here. Imagine several forms accessing the same shared resource.

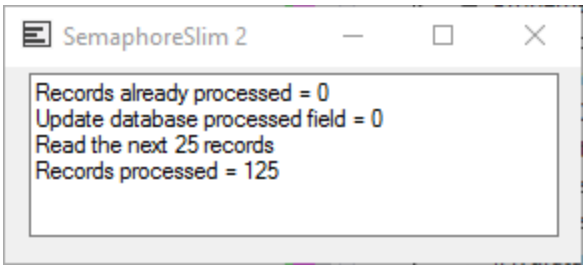


Figure 33: Form 2 Accessing Shared Resource

In fact, it might look something like Figure 34.

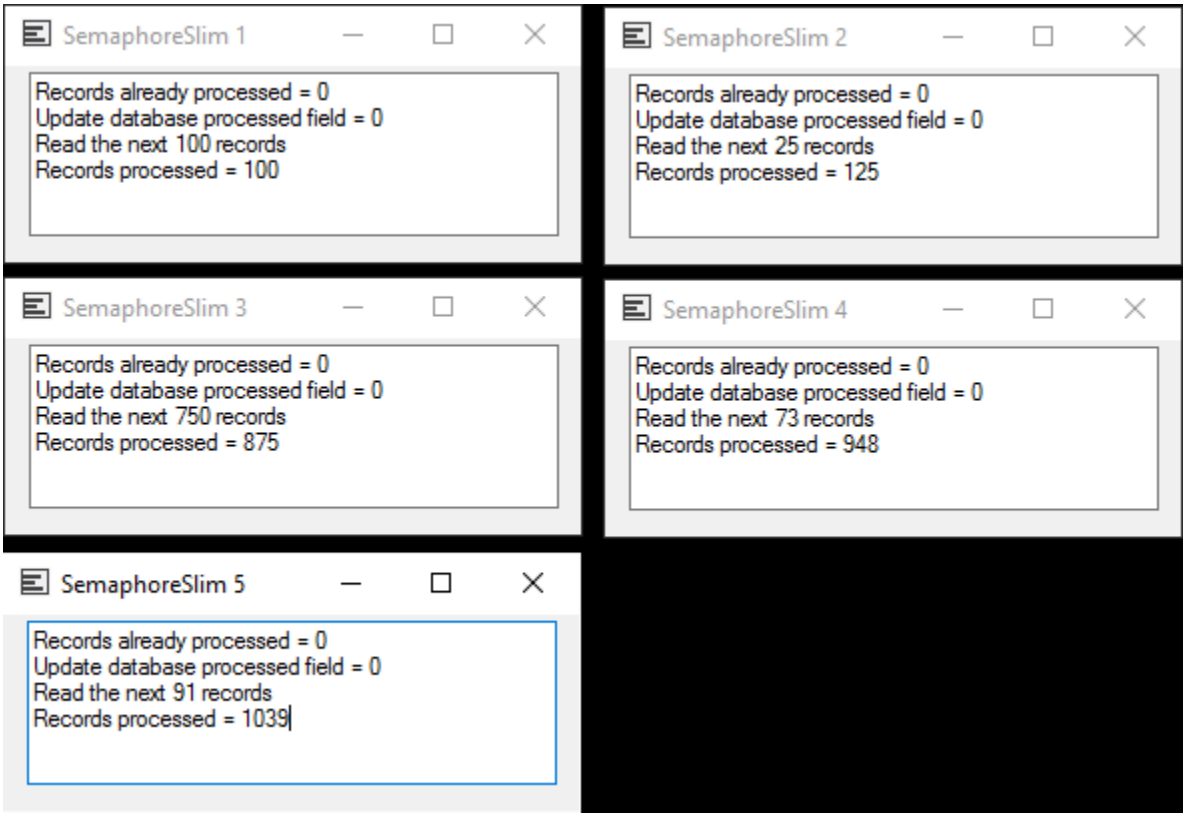


Figure 34: Several Forms Accessing the Same Data Source

This can get quite confusing because none of the data (other than on the first instance of the form) makes any sense. So let us implement the **`SemaphoreSlim`** object.

Note: `SemaphoreSlim` is a "lighter" version of the Semaphore class and is intended for use in a single app because it can be used only as a local Semaphore.

Start by adding the **`System.Threading`** namespace to your Windows Form.

Code Listing 79

```
using System.Threading;
```

Now, add the **SemaphoreSlim** object to the form with global scope within the form as a **private readonly** object.

Code Listing 80

```
public partial class SemaphoreSlimExample : Form
{
    private static readonly SemaphoreSlim sem = new SemaphoreSlim(1);
    int recordsToProcess = 0;
```

Next, modify the form load event and add a **try catch** to the code. This is very important because the next line of code we add is **await sem.WaitAsync()**, which locks the code that follows.

If an exception occurs somewhere during the processing, the **sem.Release()** line will never be reached and the code will not be released. We therefore add the **sem.Release()** method to the finally of the **try catch** block in order to ensure that the lock is released.

Code Listing 81

```
private async void SemaphoreSlimExample_Load(object sender, EventArgs e)
{
    await sem.WaitAsync();
    try
    {
        txtOutput.AppendText($"Records processed =
{SharedClass.RecordsProcessed}");
        txtOutput.AppendText($"\r\nRead the next {recordsToProcess}
records");
        await SharedClass.AccessSharedResource(recordsToProcess);
        txtOutput.AppendText($"\r\nRecords processed =
{SharedClass.RecordsProcessed}");
    }
    catch (Exception ex)
    {
        throw;
    }
    finally
    {
        sem.Release();
    }
}
```

Build your application and run it a second time. As usual, the first instance of the form performs as expected.

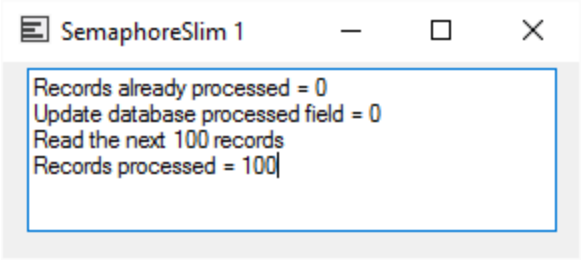


Figure 35: Form Implements SemaphoreSlim

The second instance of the form now waits for the first form to finish, then it carries on after the lock is released. As you can see, the output is correct and the values are correctly output to the database.

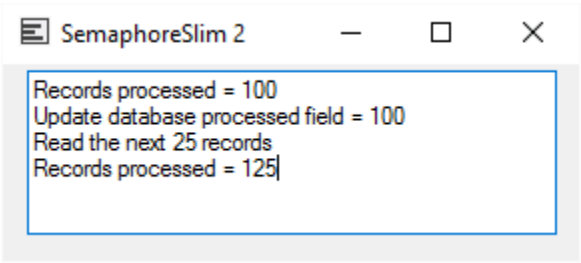


Figure 36: Form 2 Accessing Shared Resource

Chapter 5 Unit Tests and async and await

An overview of unit testing async methods

Unit testing async methods can be a challenge if you don't approach the unit tests with an "async" mindset. In other words, you can't approach unit testing async methods as you would synchronous methods.

There are mountains of resources on the Internet regarding async and await, and there are some excellent, concise articles and blog posts written by qualified individuals on unit testing async methods. At the end of this e-book, I'll expand more on some useful resources for learning async and await.

For now, however, getting unit testing set up and configured for async methods might vary depending upon which unit-testing framework you're using. For this demonstration, I will use the built-in MS Test framework.

Create a unit test in Visual Studio

Let's start by creating a new class to our Windows Forms project called **`TimeMachine`**. Notice that the namespace is **`AsyncSuccinctly`**. Add the **`System.Threading.Tasks`** namespace to the class. Also, take note that throughout this class we will be setting the **`Task.Delay`** values to two seconds.

Code Listing 82

```
using System.Threading.Tasks;
```

Add two properties called **`Year`** and **`TimeMachineEnabled`**.

Code Listing 83

```
namespace AsyncSuccinctly
{
    public class TimeMachine
    {
        public int Year { get; private set; }
        public bool TimeMachineEnabled { get; private set; }
```

In the constructor, simply initialize the **`Year`** property to the current year. If you're using C# 6.0, you can use an auto-implemented property by writing the property as follows: **`public int Year { get; private set; } = DateTime.Now.Year;`**.

Code Listing 84

```
public TimeMachine()
{
    Year = DateTime.Now.Year;
}
```

Create an async method called **`GetDaysInMonth()`** that accepts two integer parameters called **`year`** and **`month`**.

Code Listing 85

```
public async Task<int> GetDaysInMonth(int year, int month)
{
    await Task.Delay(2000);
    return DateTime.DaysInMonth(year, month);
}
```

Create a second async method called **`GoBackInTime()`** that takes an integer parameter called **`yearsBack`**. This async method also sets the **`TimeMachineEnabled`** property to **`true`**.

Code Listing 86

```
public async Task GoBackInTime(int yearsBack)
{
    await Task.Delay(2000);
    Year = DateTime.Now.Year - yearsBack;
    TimeMachineEnabled = true;
}
```

Lastly, create an async method called **`ResetTimeMachine()`** that sets the **`Year`** property back to the current year.

Code Listing 87

```
public async Task ResetTimeMachine()
{
    await Task.Delay(2000);
    Year = DateTime.Now.Year;
}
```

Next, we need to add a Unit Test project to the solution. We do this by right-clicking on the solution and selecting **Add**, **New Project…** from the context menu. Select the **Unit Test Project** from the project templates screen.

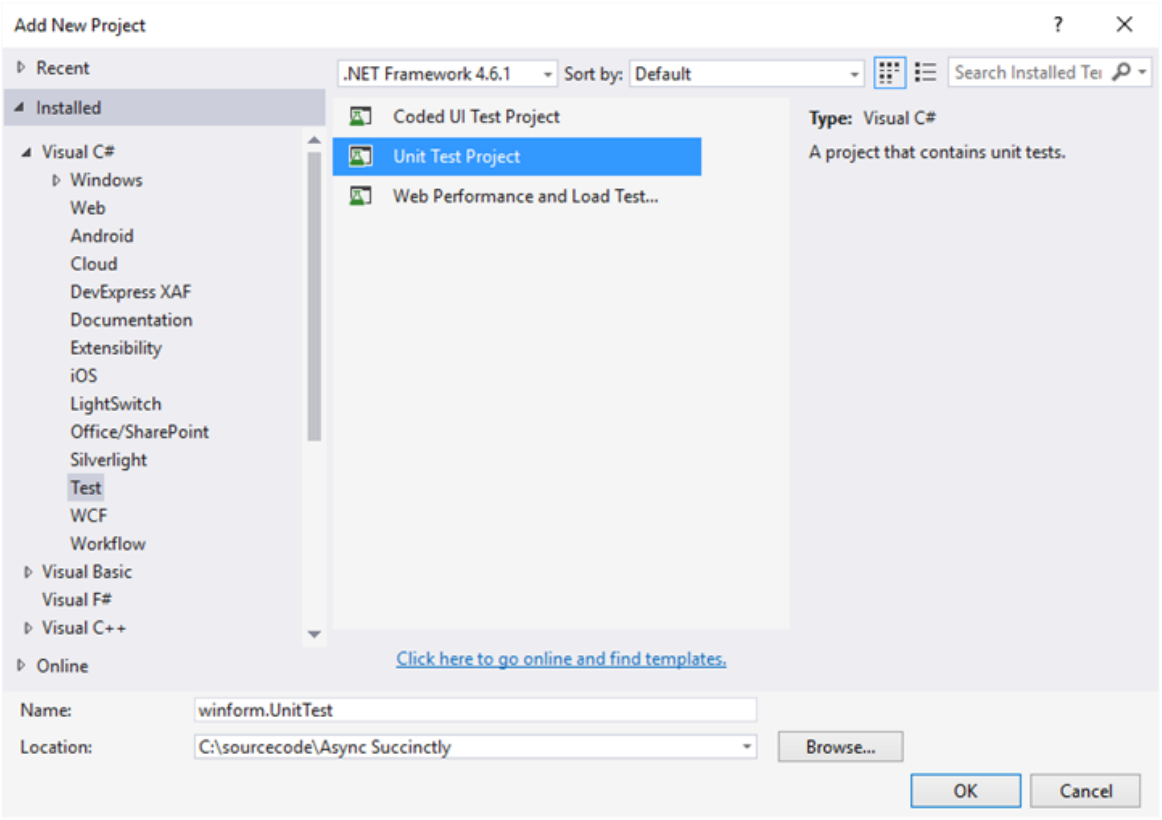


Figure 37: Create Unit Test Project

After the unit test project has been added, you will see it in your Solution Explorer with a default class added called **UnitTest1.cs**.

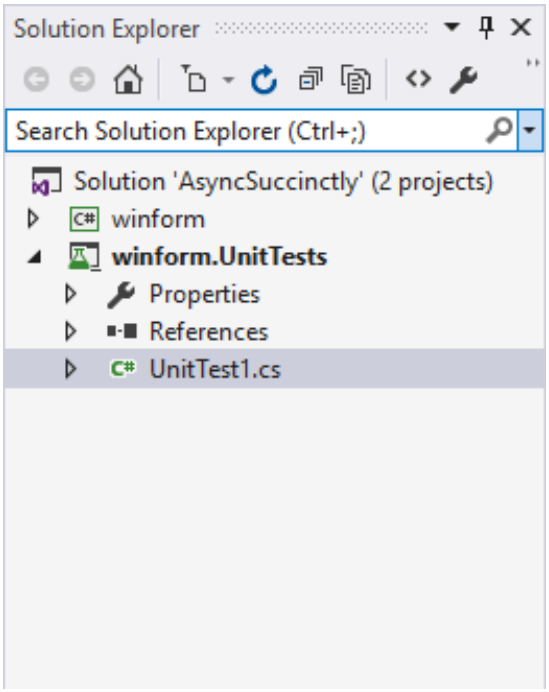


Figure 38: Unit Test Project in Solution Explorer

When you view the generated code for the **`UnitTest1`** class, you will notice that the first test method has already been added. For those of you not yet familiar with unit tests, I want to point out a few things here.

The structure of a unit test consists of:

- The **`Microsoft.VisualStudio.TestTools.UnitTesting`** namespace being imported.
- The attribute **`[TestClass]`** denoting that this class may contain unit test methods. If this attribute isn't added, the test methods are ignored.
- The unit test methods having the **`[TestMethod]`** attribute applied. This is essential for the unit test to be run.

Tip: Have a look at the following article, which is a good primer to unit testing. https://msdn.microsoft.com/en-us/library/ms182517(v=vs.100).aspx

Code Listing 88

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace winform.UnitTests
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

In the unit test project, right-click the **References** section in the solution explorer and click **Add Reference…** from the context menu.

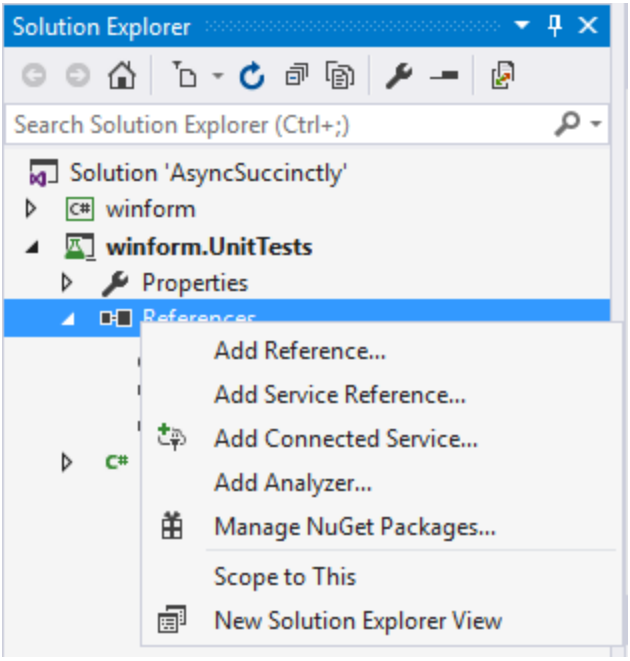


Figure 39: Solution Explorer—Add Reference

The **Reference Manager** is displayed, which means you must add the Windows Forms project as a project reference to your unit test project.

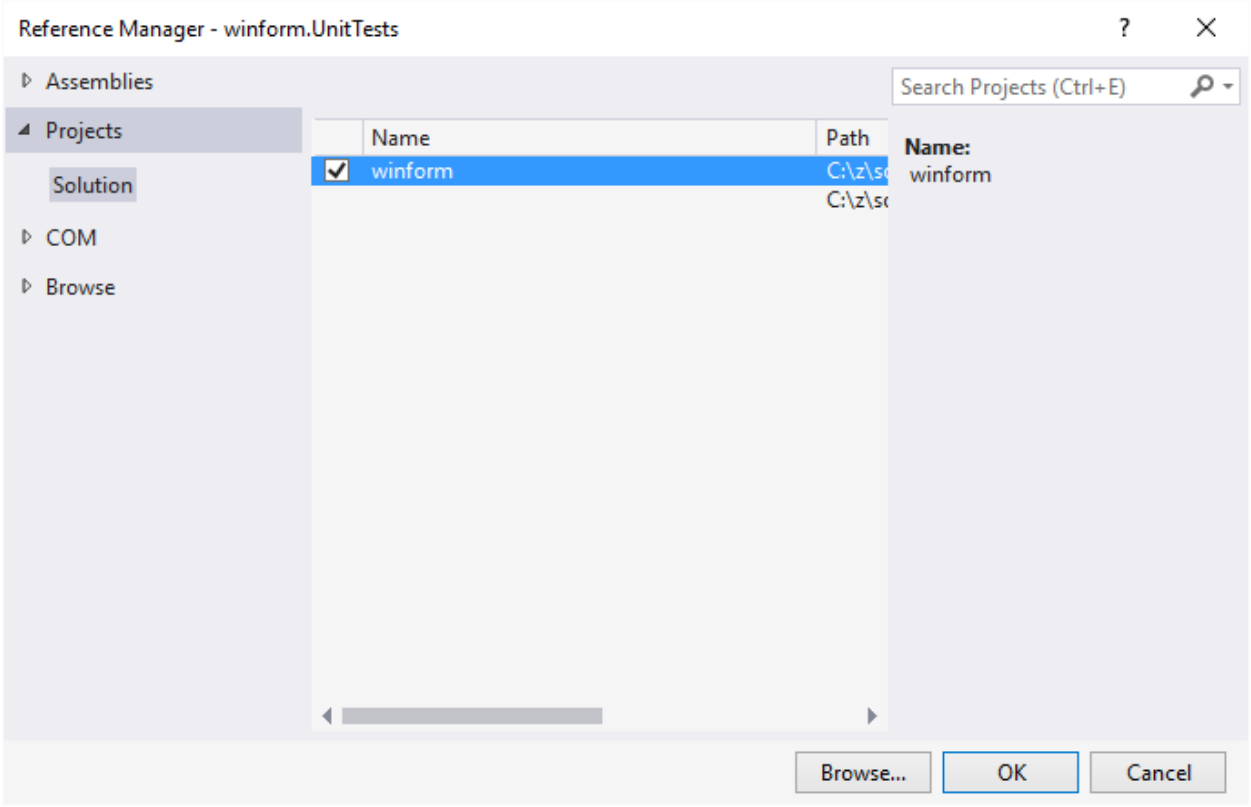


Figure 40: Select winform as Reference

After you have added the reference, import the namespace **`AsyncSuccinctly`** to your unit test class **`UnitTest1`**.

Code Listing 89

```
using System;
using AsyncSuccinctly;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Threading.Tasks;
```

Next, delete the default unit test that was added to the UnitTest1 class. After doing all that, your unit test class **`UnitTest1`** should look like Code Listing 90.

Code Listing 90

```
using System;
using AsyncSuccinctly;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Threading.Tasks;

namespace winform.UnitTests
{
    [TestClass]
```

```
    public class UnitTest1
    {

    }
}
```

We now need to add the unit test methods for our async methods in the **TimeMachine** class. Be sure to note that test methods must have the **async** keyword applied to them. And never make your test methods **void**—they must always return the type **Task** or **Task<T>**.

If you cannot avoid using a **void** test method, there are workarounds. It is, however, best practice to never have a **void** async method.

I try to name my unit tests exactly the same as the methods under test while adding the word **Test** to the end of the method name. In Code Listing 91, I am creating a unit test for the **ResetTimeMachine()** async method in the **TimeMachine** class. I instantiate the class and, because the unit test **ResetTimeMachineTest()** is an **async** method, it requires me to specify an **await** in my method. I therefore **await** the **ResetTimeMachine()** method. The method is a **Task** return type method.

Code Listing 91

```
[TestMethod]
public async Task ResetTimeMachineTest()
{
    TimeMachine tm = new TimeMachine();
    await tm.ResetTimeMachine();
}
```

Note that the **GetDaysInMonthTest()**returns a **Task<int>**, and we need to follow the same logic as before. The unit test method must be **async**, and it must specify a return type of **Task**.

Code Listing 92

```
[TestMethod]
public async Task GetDaysInMonthTest()
{
    int year = 2017;
    int month = 1;
    TimeMachine tm = new TimeMachine();
    int days = await tm.GetDaysInMonth(year, month);
}
```

The third unit test we will create is for the **GoBackInTime()** async method. Follow the same rules as with the first **Task** returning async method.

Code Listing 93

```
[TestMethod]
public async Task GoBackInTimeTest()
{
    int yearsBack = 150;
    TimeMachine tm = new TimeMachine();
    await tm.GoBackInTime(yearsBack);
}
```

Save all the code you just wrote and perform a rebuild of both the Windows Forms project and the unit test project. Then open up the **Test Explorer** window. From the main menu in Visual Studio, select **Test**, **Windows**, **Test Explorer**.

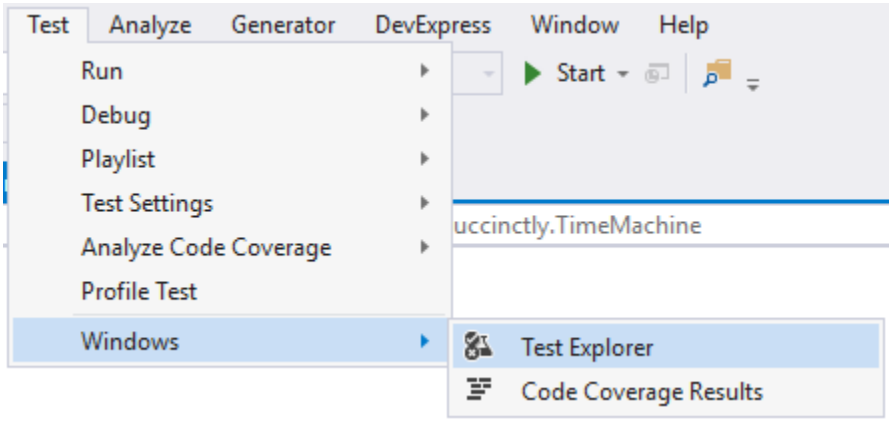


Figure 41: Open Test Explorer

The **Test Explorer** window will be displayed. You will see that it lists the unit test methods you created earlier.

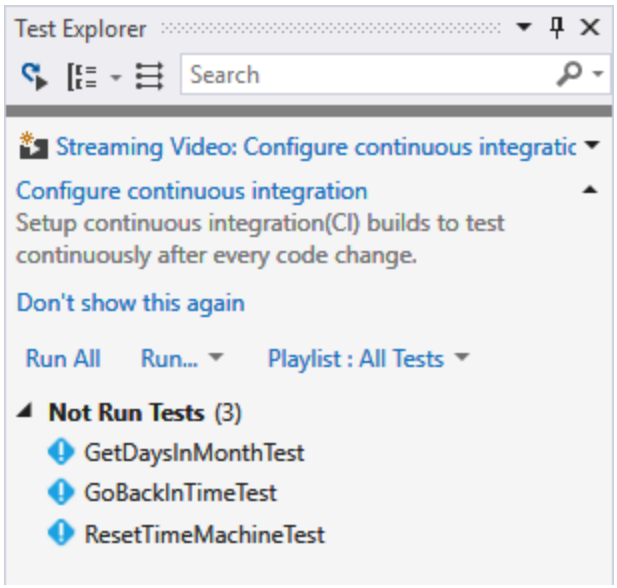


Figure 42: Test Explorer

Tip: It is very important that you always perform a rebuild of your unit test project or project under test when you make changes. I have found that sometimes unit tests don't "see" the changes I have made to a unit test or a method under test unless I perform a rebuild.

In your unit test class **`UnitTest1`**, right-click the editor and select **Run Tests** from the context menu. You can also click the **Run All** link in the **Test Explorer** window to run the unit tests.

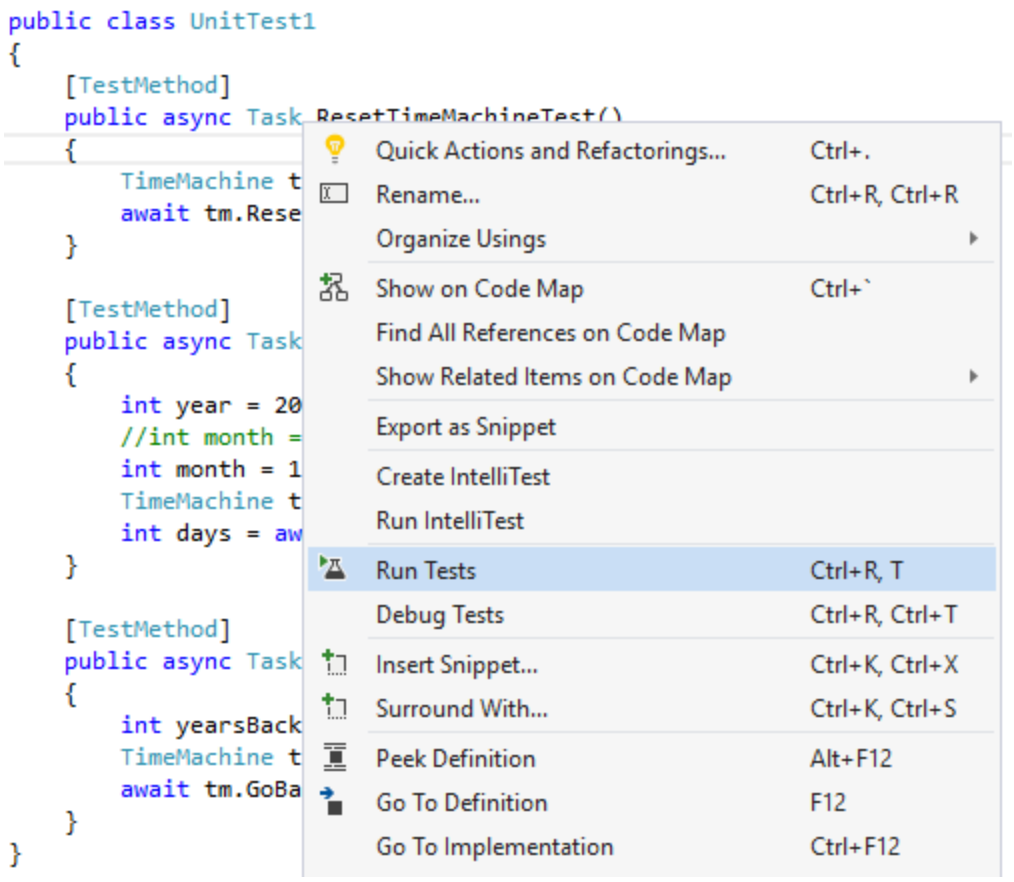


Figure 43: Run Tests

The unit test will be run and the results will be displayed in the **Test Explorer** window. Notice how each method under test has **2 sec** displayed next to the results in the **Test Explorer** window. This happens because we have set all the async methods in the **`TimeMachine`** class to delay by setting the **`Task.Delay`** to two seconds.

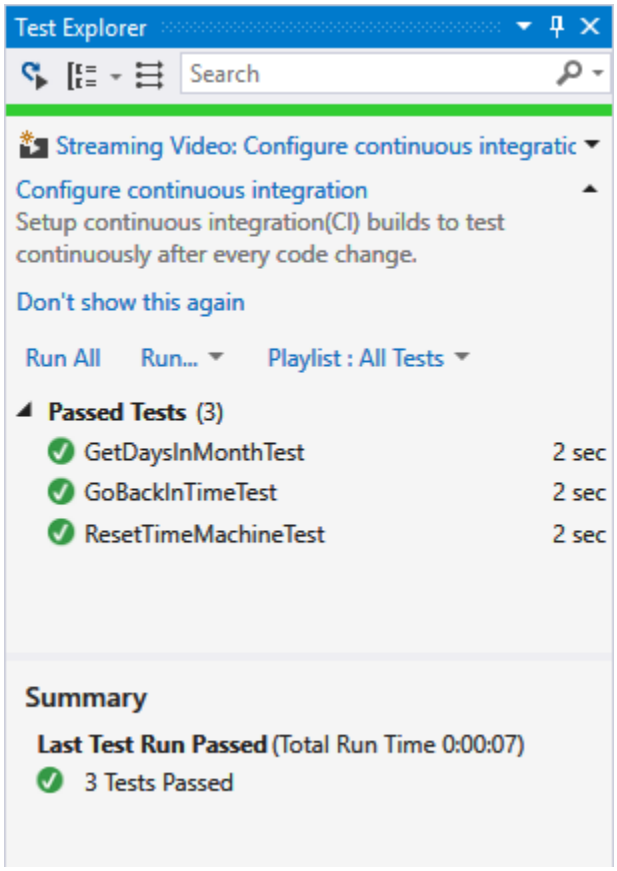


Figure 44: Unit Tests Passed

This is all well and good, but if you implement your unit tests incorrectly (by not adding the **`Task`** and **`async`** keywords to the unit test method), you might have the unit test report a test passed when in actual fact it has thrown an exception. This is why it is essential that you add **`async Task`** or **`async Task<T>`** to your unit test methods so that you can properly test your async methods.

Let's have a look at what happens when some of these methods fail. In the unit test method **`GetDaysInMonthTest()`**, change the **`month`** value to **`0`**.

Code Listing 94

```
[TestMethod]
public async Task GetDaysInMonthTest()
{
    int year = 2017;
    int month = 0;
    TimeMachine tm = new TimeMachine();
    int days = await tm.GetDaysInMonth(year, month);
}
```

In the **TimeMachine** class, modify the **GetDaysInMonth()** method and add code to cause a **DivideByZeroException**. Simply add **int iCauseZeroException = year / month;** directly after the await **Task.Delay(2000);**.

Code Listing 95

```
public async Task<int> GetDaysInMonth(int year, int month)
{
    await Task.Delay(2000);
    int iCauseZeroException = year / month;
    return DateTime.DaysInMonth(year, month);
}
```

Next, for the **ResetTimeMachine()** method, force a new **Exception** to be thrown. You can give the exception a custom message.

Code Listing 96

```
public async Task ResetTimeMachine()
{
    await Task.Delay(2000);
    Year = DateTime.Now.Year;
    throw new Exception("Async Method threw an Exception");
}
```

Remember to rebuild your solution after saving these changes and run your unit tests again by clicking on the **Run All** link in the **Test Explorer** window. This time you will see that the two methods we modified in the **TimeMachine** class have failed. This is good—we now know that our async unit tests are working correctly.

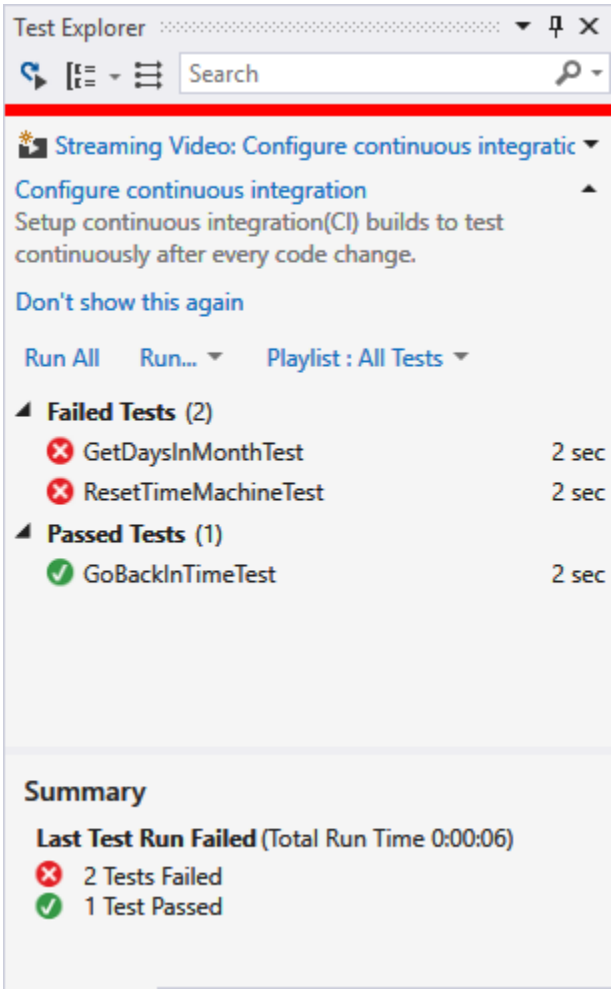


Figure 45: Test Failures

In order to see the specific error returned by the unit test, click the test in the **Failed Tests** node in the **Test Explorer** window.

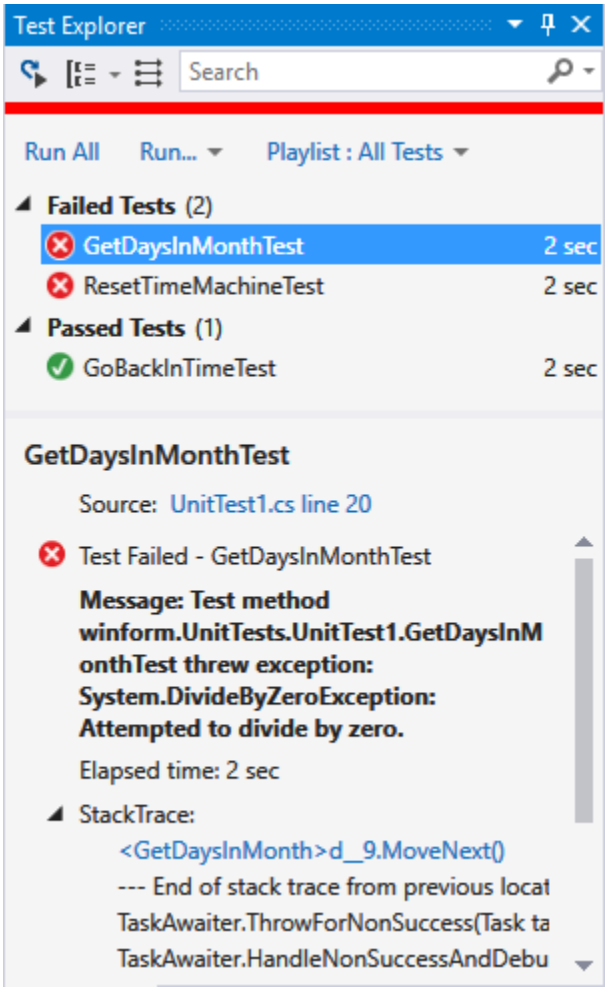


Figure 46: View Failed Tests

You can see that the unit test **`GetDaysInMonthTest()`** reported a **`DivideByZeroException`** as expected. If you click the unit test **`ResetTimeMachineTest()`**, you will see the custom exception message we added earlier.

> ***Note: While I have shown the very basics of unit testing async methods, you will probably want to add more functionality by using Assert to test return values from a Task<T> async method.***

Avoiding deadlocks in unit tests

Deadlocks can arise while we do unit testing on async methods. A nice way to avoid this is to specify a timeout on your individual unit tests by using the **`[Timeout]`** attribute. In order to do this, modify your **`[TestMethod]`** attribute to include the **`[Timeout]`** attribute as follows **`[TestMethod, Timeout(3000)]`**.

Code Listing 97

```
[TestMethod, Timeout(3000)]
public async Task ResetTimeMachineTest()
{
    TimeMachine tm = new TimeMachine();
    await tm.ResetTimeMachine();
}

[TestMethod, Timeout(3000)]
public async Task GetDaysInMonthTest()
{
    int year = 2017;
    int month = 1;
    TimeMachine tm = new TimeMachine();
    int days = await tm.GetDaysInMonth(year, month);
}

[TestMethod, Timeout(3000)]
public async Task GoBackInTimeTest()
{
    int yearsBack = 150;

    TimeMachine tm = new TimeMachine();
    await tm.GoBackInTime(yearsBack);
}
```

Next, set all the delays in your **`TimeMachine`** class to **`Task.Delay(4000)`**.

Code Listing 98

```
public async Task<int> GetDaysInMonth(int year, int month)
{
    await Task.Delay(4000);
    return DateTime.DaysInMonth(year, month);
}

public async Task GoBackInTime(int yearsBack)
{
    await Task.Delay(4000);
    Year = DateTime.Now.Year - yearsBack;
    TimeMachineEnabled = true;
}

public async Task ResetTimeMachine()
{
    await Task.Delay(4000);
    Year = DateTime.Now.Year;
}
```

Rebuild and run your unit tests and see that all the unit tests fail because the async methods under test exceeded the timeout defined in the unit test **`[Timeout]`** attribute.

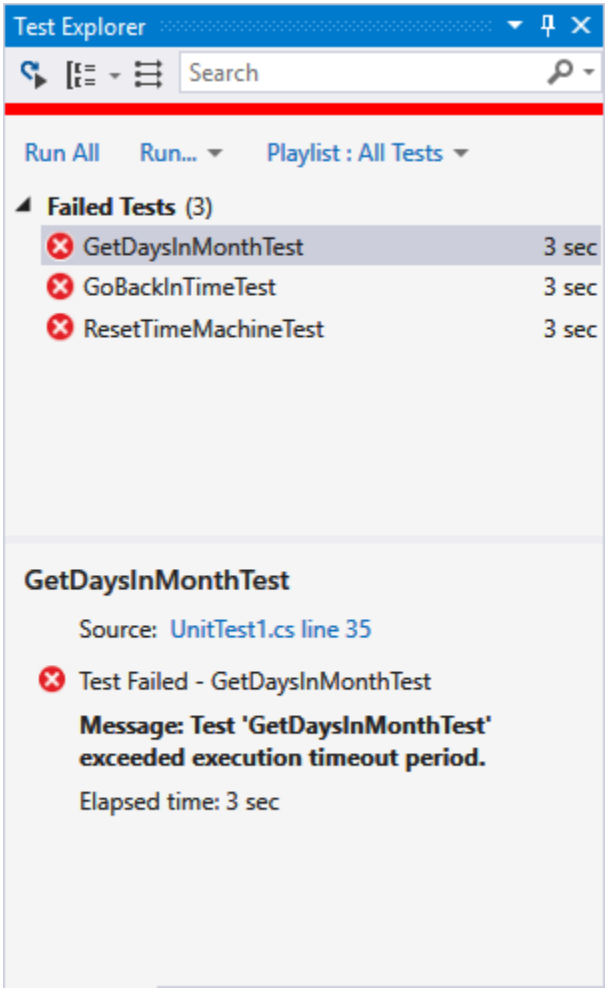


Figure 47: Unit Test Exceeded Timeout

Final thoughts

I have come across instances where the added unit tests do not show up in the **Test Explorer**. I'm not sure why this happens, but it seems to be a common issue—there are numerous threads on the Internet discussing this very issue.

Here are some things you can try if you encounter this:

- Ensure that you have rebuilt your solution, including the test project.
- Make sure that the **Default Processor Architecture** matches the solution. If your solution is 64-bit, select from the main Visual Studio menu **Test**, **Test Settings**, **Default Processor Architecture**, and set it to **x86** or **x64**.

- If all else fails, I'm sorry to say that you will probably have to restart Visual Studio.

Additional resources—where to learn more

Two names that stand out here: Stephen Cleary and Stephen Toub. If you want to learn more about async and await, start with the following resources.

Stephen Cleary

http://blog.stephencleary.com/

https://stackoverflow.com/search?q=user:263693+[async-await]

https://github.com/StephenCleary

Stephen Toub

https://channel9.msdn.com/events/speakers/Stephen-Toub

https://blogs.msdn.microsoft.com/pfxteam/