

Distributed Computing

Tutorial 1

What We're Doing Today

1. Setting up a Windows Environment
2. Writing a simple DLL
3. Writing a simple .NET Remoting server
4. Writing a really simple WPF Windows App

Introduction

Windows, and it's development environment Visual Studio, is a *lot* different to what you're used to in Linux land. We don't use Windows on a regular basis however, and unless you're a Microsoft fan or have done some Systems Admin units, you're not used to doing too much in Windows beyond playing games.

As modern distributed applications are developed by the progressively more pervasive "DevOps" role, we're going to teach you some Ops to go with the Dev. That's right, you're going to do little bits of systems administration during this unit! This will hopefully mean you don't have to run your applications in a Visual Studio debug mode when you want to distribute applications in the real world.

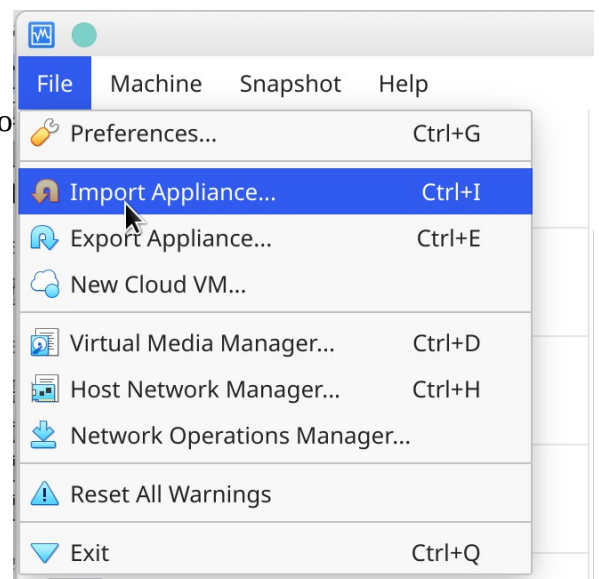
We're going to be building a very simple remoting application today, just to give you a taste of what distributing an application actually looks like. It's surprisingly simple.

Task 1: Setting up your Windows Environment

So first thing you'll want to do is find your Visual Studio OVA file. This is a compressed Virtual Machine running Windows Server 2019 Evaluation Edition and Visual Studio Community 2019 (if you're using a Windows machine at home, you just need to download VS Community 2019. Don't use Visual Studio Code! It won't have everything). Theoretically, this will be in /local/ on your machine.

That said, the Curtin labs are curious environments indeed, and it might not be there. If you can't find it, ask your friendly neighbourhood tutor.

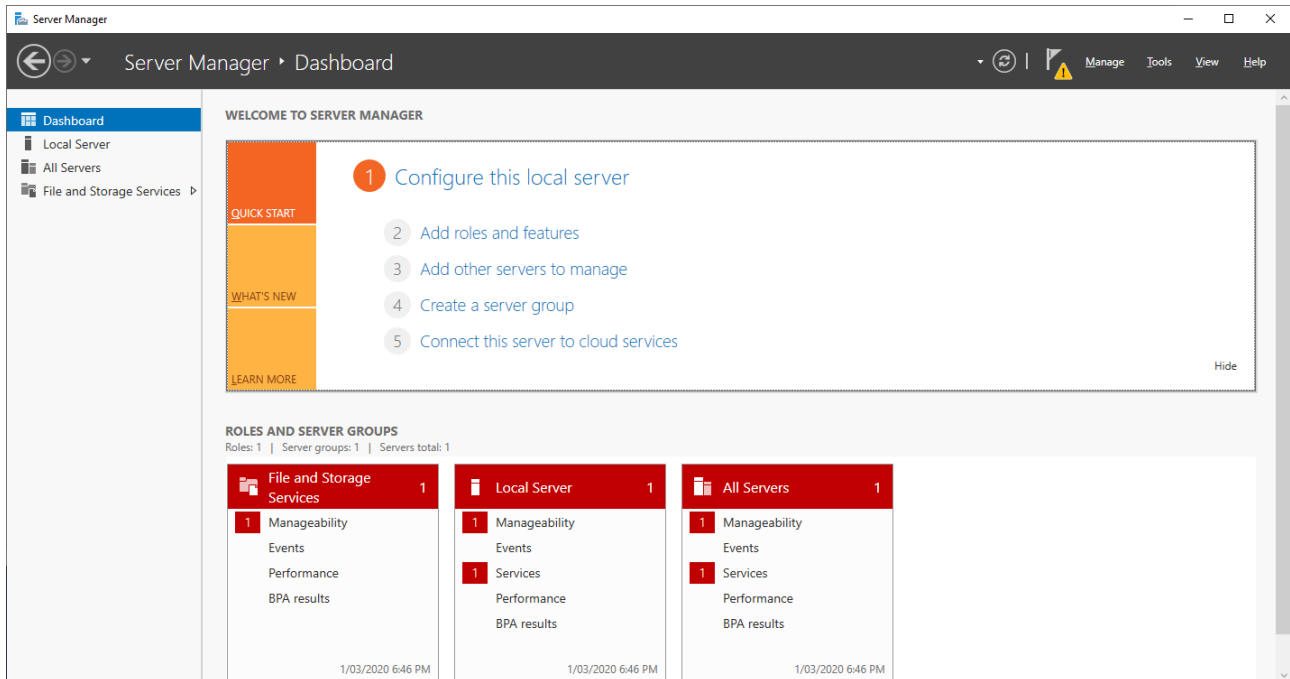
You'll need to import that into Virtualbox. This should be as easy as double clicking on the OVA file. If that doesn't work, open Virtualbox, click "File", and "Import



Appliance”, following the prompts until it’s installed. You’ll want to put this on a USB drive if at all possible, just for speed reasons.... But it *should* work on your home folder in a pinch.

Once it’s all imported, you’ll need to start the virtual machine by double clicking on it. I’ve set the machine up in a post-install setup mode, so you will be greeted by a page asking about your keyboard input. Unless you’re working on something exotic at home, the defaults should do. You will also be asked to put in a password for the Administrator account.

Log in as Administrator. You should land at a page like this:

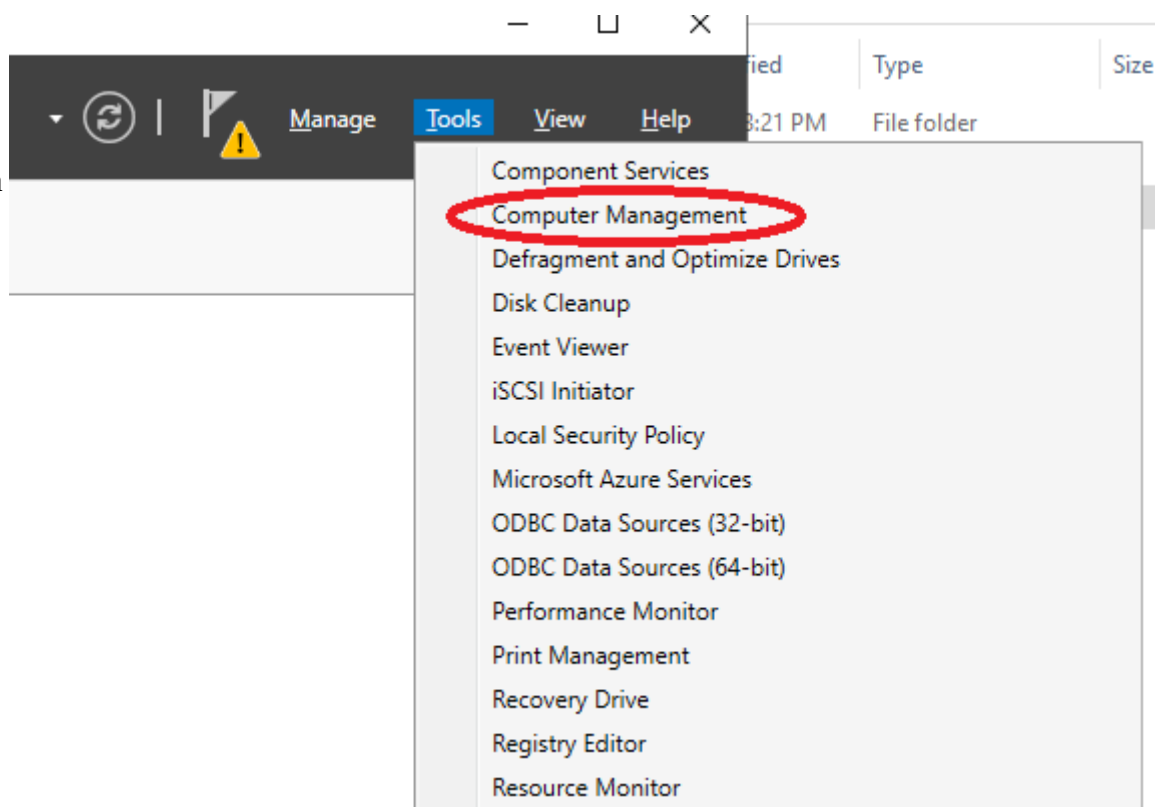


This is the Server Manager, the Admin’s command console for Windows Servers. Administrator is basically the same as root in Linux machines, it has all the power and you don’t want to run anything as it. So! You need to make a user account for development. Go hunt through the Windows configuration to find the users menu. There are a few ways to get there.

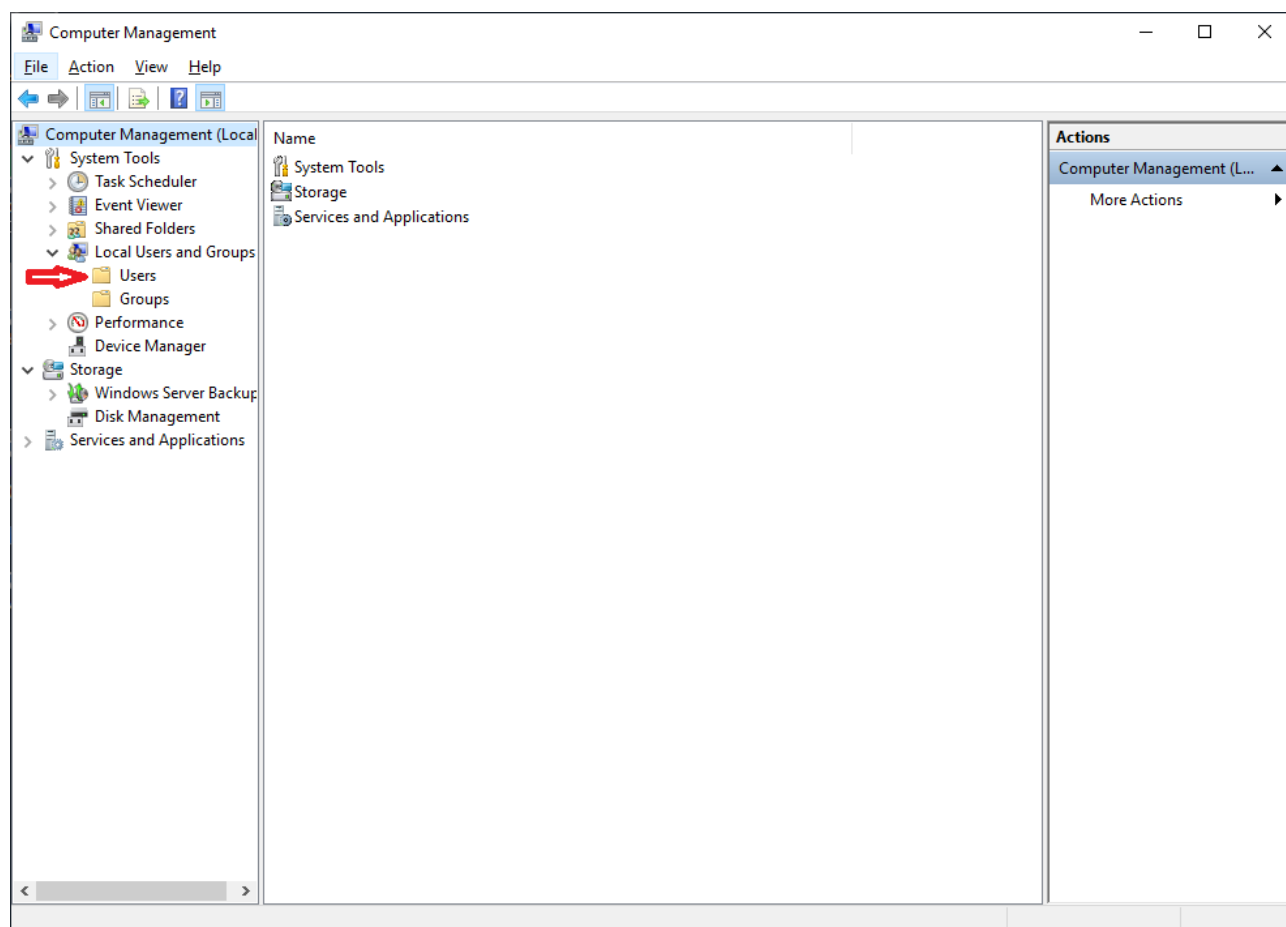
If you don’t want to jaunt through configuration and just want to get the job done, here’s one way:

Open the Tools menu in Server Manager, and click on Computer Management:

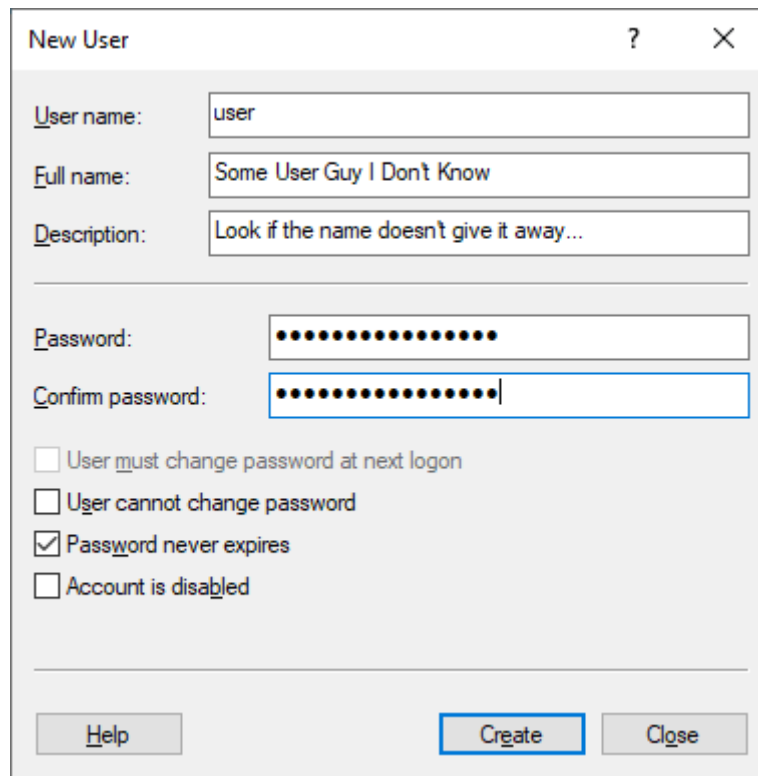
This will open the



Computer Manager console, which has the users folder hidden inside:



Right click on Users, and click New User. Add whatever you want in here, although it may be convenient to change the password features to reduce the amount of password changing you'll need to do:



The screenshot shows a 'New User' dialog box with the following fields and options:

- User name:** user
- Full name:** Some User Guy I Don't Know
- Description:** Look if the name doesn't give it away...
- Password:** [masked with dots]
- Confirm password:** [masked with dots]
- ☐ User must change password at next logon
- ☐ User cannot change password
- ☒ Password never expires
- ☐ Account is disabled
- Buttons:** Help, Create, Close

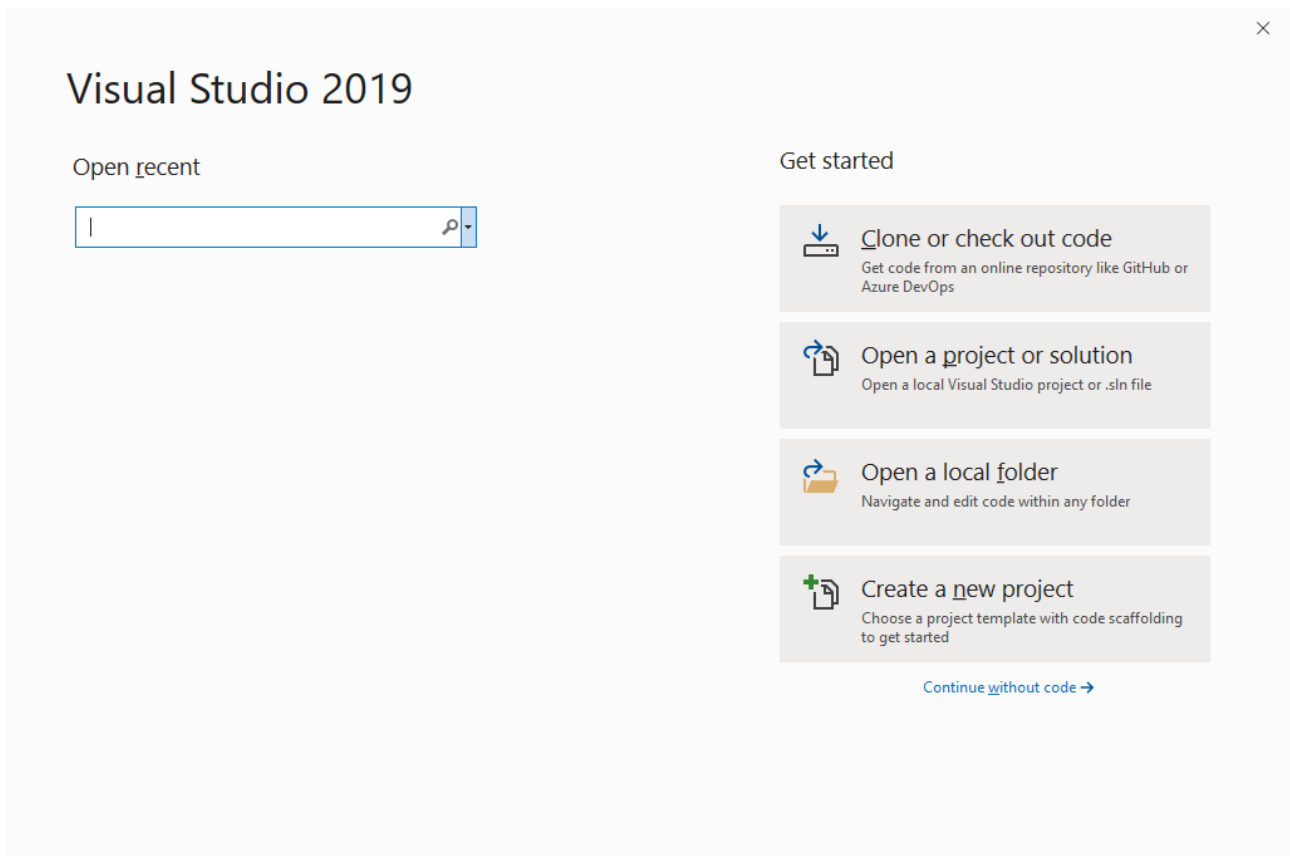
Now log out of Administrator, and log into your new user account. This is the account you'll be using in this unit.

Apart from being a terrible security hole waiting to happen, Visual Studio actually doesn't work properly if you run it as Administrator. **Make sure you have made a user account before proceeding!**

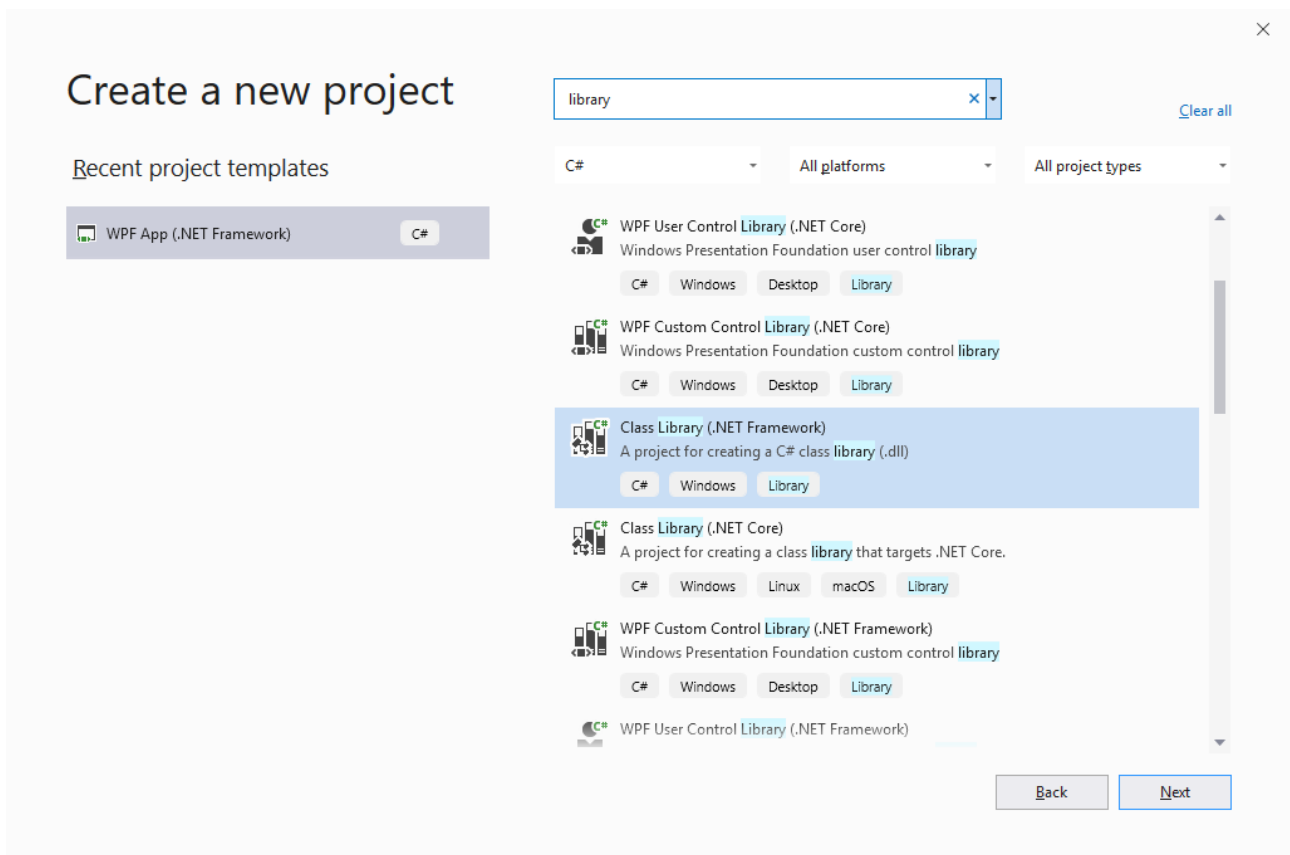
Task 2: Writing a simple DLL

Now that your Windows installation is ready to Visual Studio, start up Visual Studio! It may ask you to log in. This is entirely up to you, if you don't want to, don't log in.

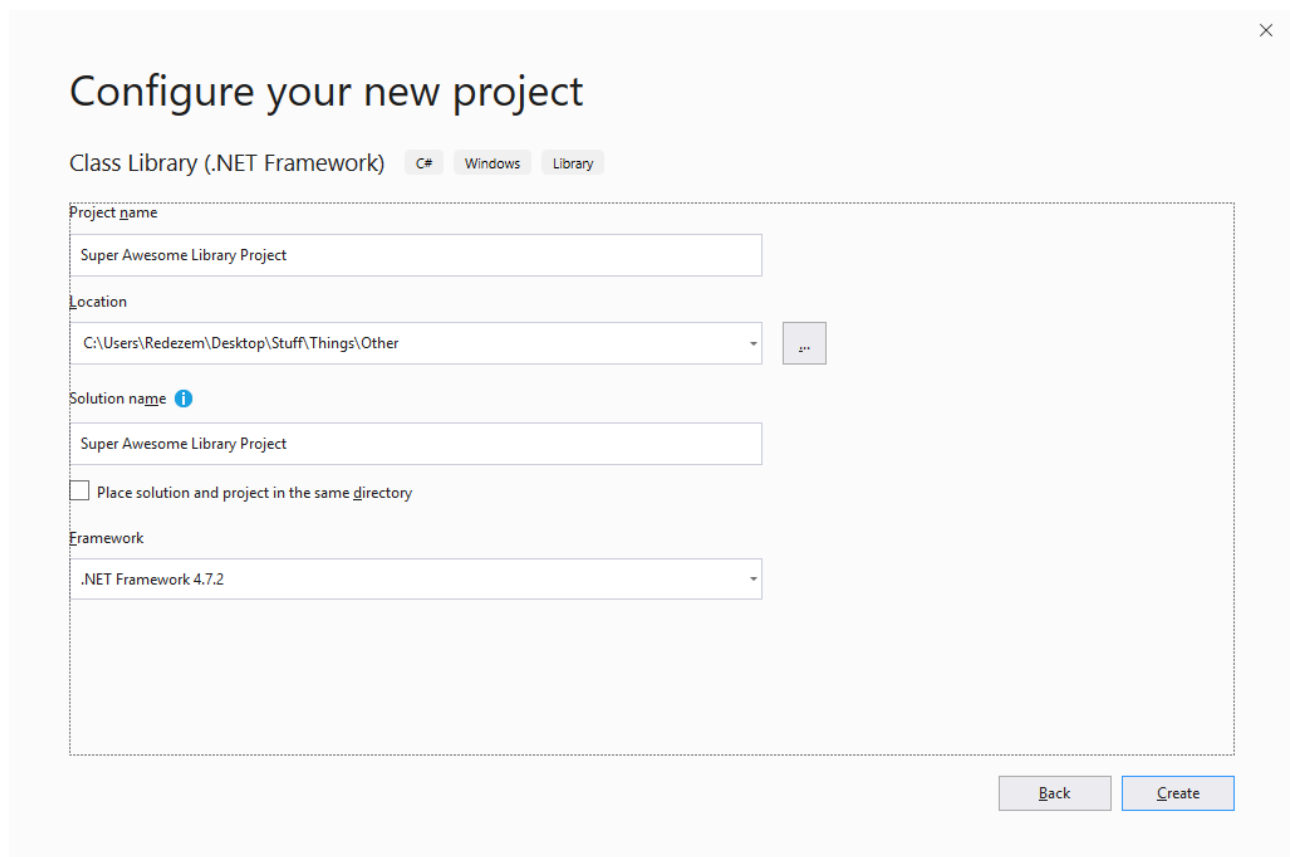
You should find yourself at this screen:



You'll want to create a new project. Find the .NET Framework Class Library project



Give it a name and a location (keep a track of this otherwise you'll get lost):



What we're going to build are 3 classes pretending to be a "database". Two of them will be private to the library. One of them will be a publicly accessible class that will act as a pseudo-interface to the "database". I'll let you figure out how to add classes to a project in VS (hint, Google is your friend!).

Class 1: Database Generator

Our first class is a pseudo-random generator of database entries (I did say *pretend* to be a database). You'll need to declare it as an `internal` class.

You will need to implement the following private functions. Remember, they need to *generate* their outputs randomly! How you achieve that is up to you!:

```
private string GetFirstname()  
private string GetLastname()  
private uint GetPIN()  
private uint GetAcctNo()  
private int GetBalance()
```

You'll also need a public function that lets someone request a "record". This should ideally use the private functions.

```
public void GetNextAccount(out uint pin, out uint acctNo, out string firstName, out string  
lastName, out int balance)
```

You'll notice that this returns void, but has a whole bunch of "out" variables. These are "out mode" parameters, and are basically C#'s way of doing multiple returns. It's kind of like passing pointers through in C, except you *can't* pass things in to the function using out, values only go one way. If you really want to pass things through as if they were a pointer reference, you can do that with the similar "ref" keyword. We'll look at this in a later week.

Class 2: Database storage class

Once you're done here, you'll need to make the second internal class. This will store individual records in the "database". It will need a field for each variable that the generator can create. Create something similar to:

```
internal class DataStruct
{
    public uint acctNo;
    public uint pin;
    public int balance;
    public string firstName;
    public string lastName;
    public DataStruct()
    {
        acctNo = 0;
        pin = 0;
        balance = 0;
        firstName = "";
        lastName = "";
    }
}
```

Note the constructor. In general, it's always a good idea to initialise variables, even in a data only object.

Class 3: Database Class

For our last class we'll be building a publicly accessible object that will define the "database". Create a public class, and ensure it has a class field that holds a list of database storage classes. Something like the following is perfect:

```
public class DatabaseClass
{
    List<DataStruct> dataStruct;
}
```

Create a constructor that instantiates the List:

```
public DatabaseClass()
{
    dataStruct = new List<DatabaseLib.DataStruct>();
}
```

The constructor will also need to load the List up with a large number of entries. That's up to you to implement! In addition, you need to implement the following functions:

```
public uint GetAcctNoByIndex(int index)
public uint GetPINByIndex(int index)
public string GetFirstNameByIndex(int index)
public string GetLastNameByIndex(int index)
```

```
public int GetBalanceByIndex(int index)
public int GetNumRecords()
```

Task 3: Building a simple .NET Remoting server.

Right click on the solution in the Solution Explorer and add a new project. You want to make a [Console application for the .NET Framework](#).

You will need to create a new interface, as well as a class. The interface is going to define the .NET Remoting network interface, while the class is going to be the internal implementation of the interface.

The Interface:

You'll need to make a public interface for the .NET server. This is done by classifying an interface as a [ServiceContract]. The functions that will be used by the .NET clients are also contractual interfaces, so they each need to be labelled as [OperationContract]. This should look something like the following:

```
//Make this a service contract as it is a service interface
[ServiceContract]
public interface DataServerInterface
{
    //Each of these are service function contracts. They need to be tagged as
    OperationContracts.
    [OperationContract]
    int GetNumEntries();
    [OperationContract]
    void GetValuesForEntry(int index, out uint acctNo, out uint pin, out int bal, out
string fName, out string lName);
}
```

The Implementation

To actually do anything, we need an implementation of the interface. We don't actually want clients to use the implementation (as we want it to use the remote interface) so this will be classified as an [internal](#) class.

As the class is defining the behaviours of a service, we will need to define it as a [ServiceBehavior]. It needs some specialised inputs to it (as you can see below). Basically:

1. `ConcurrencyMode = ConcurrencyMode.Multiple` makes the service multi-threaded. This is important, as otherwise Visual Studio will try to play it safe for you and make it single-threaded. This creates problems when scaling to many hundreds to thousands of clients.
2. `UseSynchronizationContext = false` tells Visual Studio to allow us to manage our own thread synchronisation. You have *probably* done some threading before now, so you will be able to do a lot better job than VS.

You need to create an object that will implement the interface, so something similar to:

```
//Time to describe our service behavior
```



```

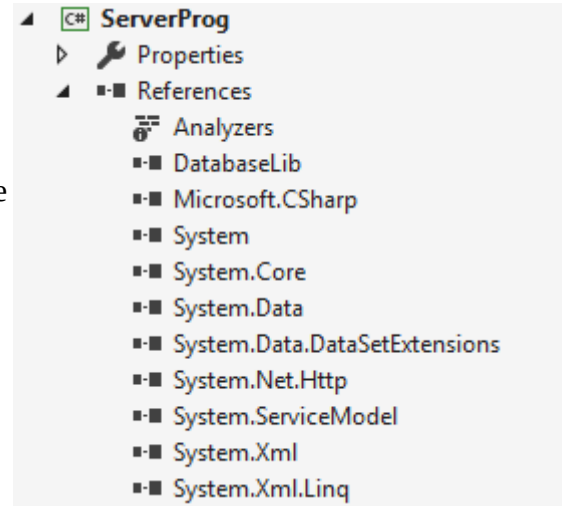
[ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Multiple , UseSynchronizationContext
= false)]
internal class DataServer : DataServerInterface
{
    public DataServer();
    public int GetNumEntries();
    public void GetValuesForEntry(int index, out uint acctNo, out uint pin, out int bal,
out string fName, out string lName);
}

```

The server will need to encapsulate the Database Class that we made in the DLL. To do this, first add a reference to the library project you made earlier (to start, right click on the References submenu in the Solution Explorer for your console app). Once that is done, add a reference to it in code like the following:

```
using DatabaseLib;
```

I'll let you figure out how to best use the Database Class from that point.



The Server

The code here is a good start for building the actual server part:

```

static void Main(string[] args)
{
    //This should *definitely* be more descriptive.
    Console.WriteLine("hey so like welcome to my server");
    //This is the actual host service system
    ServiceHost host;
    //This represents a tcp/ip binding in the Windows network stack
    NetTcpBinding tcp = new NetTcpBinding();

    //Bind server to the implementation of DataServer
    host = new ServiceHost(typeof(DataServer));
    //Present the publicly accessible interface to the client. 0.0.0.0 tells .net to
    accept on any interface. :8100 means this will use port 8100. DataService is a name for the
    actual service, this can be any string.
    host.AddServiceEndpoint(typeof(DataServerInterface), tcp,
"net.tcp://0.0.0.0:8100/DataService");
    //And open the host for business!
    host.Open();
    Console.WriteLine("System Online");
    Console.ReadLine();
    //Don't forget to close the host after you're done!
    host.Close();
}

```

Alright! Sanity check time!

- Build the entire solution (Ctrl+Shift+B)
- Right click on your server
- Go to Debug

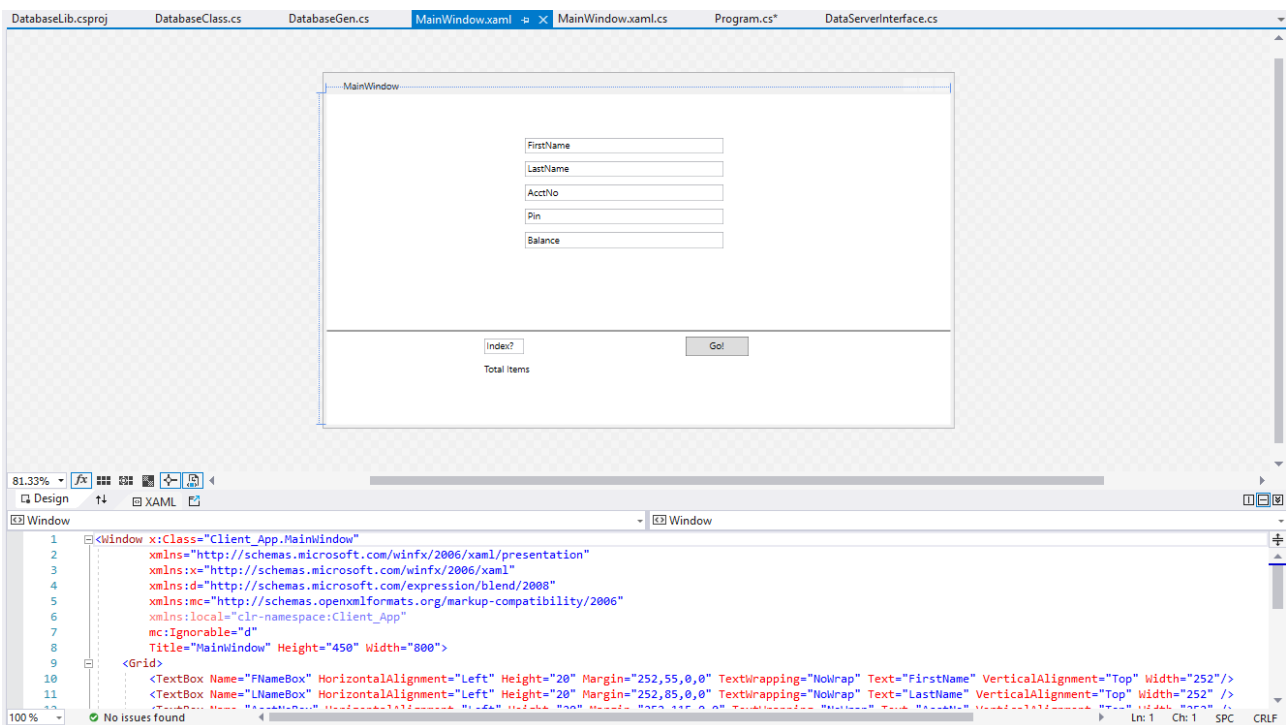
- Click Start New Instance

If nothing breaks, Congratulations! You've successfully made a .NET Remoting server!

Task 4: Building the WPF Client

Now that you have a server, you need something to use it. Add a project to the solution of type WPF App for the .NET Framework. If you click on MainWindow.xaml, you will see a blank window ready for you to add stuff to.

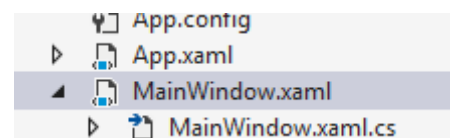
You need to create a basic front end layout. You can use the toolbox and VS's WYSIWYG editor to build it by dragging and dropping tools into the page, *or* you can just edit the XAML in the editor window to the bottom.



What you'll want to make sure is that you have a field for each value in the database, a field for the index, and a go button. I've included a field for total items in the database for ease of use too.

Importantly, these each need a unique Name field. This defines their name for the C# code in the backend.

If you look into the MainWindow.xaml submenu on the Solution Explorer, you'll find a MainWindow.xaml.cs file. Double clicking on this will open the C# file responsible for the backend code for your window. The constructor is already pre-filled with a function call that will start up the window renderer.



You'll want to add to the constructor to start up and hold onto a connection to your server. Something similar to the following is a good start:

```
public MainWindow()
{
    //Set up the window
```

```

        InitializeComponent();

        //This is a factory that generates remote connections to our remote class. This
is what hides the RPC stuff!
        ChannelFactory<ServerProg.DataServerInterface> foobFactory;
        NetTcpBinding tcp = new NetTcpBinding();

        //Set the URL and create the connection!
        string URL = "net.tcp://localhost:8100/DataService";
        foobFactory = new ChannelFactory<DataServerInterface>(tcp, URL);
        foob = foobFactory.CreateChannel();
        //Also, tell me how many entries are in the DB.
        TotalNum.Text = foob.GetNumEntries().ToString();
    }

```

You'll notice you can't refer to anything in your server program without creating a reference to it. Windows is quite good at solving this problem, any compiled C# code can be imported just like a DLL, so you can quite easily create a reference to your server executable, and it will simply import public objects!

Now, jump back to the XAML WYSIWYG editor, and double click on the button you added. This will edit the XAML and add an on-click action. It will then take you to the C# file automatically and create a blank function for the on-click action function. Fill it out with something similar to the following.

```

private void GoButton_Click(object sender, RoutedEventArgs e)
{
    int index = 0;
    string fName = "", lName = "";
    int bal = 0;
    uint acct = 0, pin = 0;

    //On click, Get the index....
    index = Int32.Parse(IndexNum.Text);
    //Then, run our RPC function, using the out mode parameters...
    foob.GetValuesForEntry(index, out acct, out pin, out bal, out fName, out lName);
    //And now, set the values in the GUI!
    FNameBox.Text = fName;
    LNameBox.Text = lName;
    BalanceBox.Text = bal.ToString("C");
    AcctNoBox.Text = acct.ToString();
    PinBox.Text = pin.ToString("D4");
}

```

You should now be able to build the whole solution (Ctrl+Shift+B). You should also be able to start new debug instances for both your server and your GUI client.

If all works, you should be able to access elements of your database from your GUI!
Congratulations!

Task 5: Time for you to do some stuff

If you've made it here, you now have some idea on how to build a distributed application using .NET Remoting. However, there are still some things you need to do, and things you can learn.

1. Currently there is no exception handling. You might want to do some error checking on the client side.... However what happens if someone tries to send bad input to the server? You might want to implement some exception handling on the server too.
 - There's another problem too. Exceptions can't cross the network boundary.... How do you fix this?
2. Currently, in order to use the Data Server Interface, your GUI has to import the server executable. This is less than ideal as the server will have to be bundled with deployments of your frontend. For our purposes this is okay, but it's not great for real world setups.

Try moving the Interface to a DLL, and including it in both applications that need to see it.

3. Obviously this application is very, *very* simple. Try adding profile pictures to the application. You will need:
 - A way of reading images into the Database. Hint: Look up the Bitmap class.
 - A way of sending the images to the client. This *should* be pretty straight forward.
 - A way of presenting the images in the GUI. I'd suggest google for this one.

Good luck! And remember, if you get stuck, ask your tutor :)