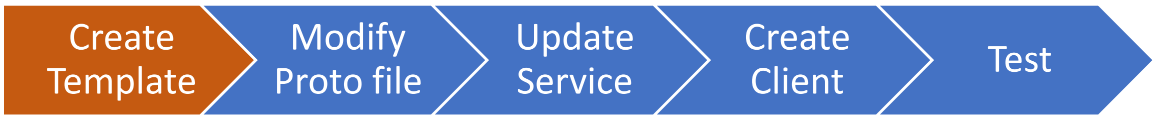
Dist. Sys. gRPC

In this lab you'll be looking at Remote Procedure Calling (RPC), specifically gRPC.

We'll be using C# and Visual Studio. gRPC functionality is provided through ASP.NET Core. You will need to ensure that you have fully updated Visual Studio and have the *ASP.NET and web development* workload installed.

Be aware that there are some differences when using **Mac devices in .NET 7 and earlier**. See troubleshooting here: https://docs.microsoft.com/en-us/aspnet/core/grpc/troubleshoot

The workflow we will be following to start this lab is:



Open Visual Studio and create a new project.

Search for 'gRPC Service' and choose this template. Click Next.

Name your project GrpcService and choose Next, use **.Net 8 LTS** then click Create.

The auto-generated project template is a 'Greeter Service'. Click the Protos folder in the Solution Explorer and open the *greet.proto* file.

Create Template
Modify Proto file
Update Service
Create Client
Test

The proto file is the contract which defines the available service(s), including:

- The service name - `service Greeter`
- Remote procedure call functions
  - Function name: `rpc SayHello`
  - Request Message type: `(HelloRequest)`
  - Response Message type: `returns (HelloReply);`
- Messages
  - Message name: `message HelloRequest`
  - Contents (objects/types sent as part of the message): `string name`
  - Order of contents: `= 1;`

We are going to make a remote Calculator service so we will need to make some changes to this proto file to modify the service and the messages.

First, rename *greet.proto* to *calc.proto* in the Solution Explorer.

Edit the package to `calc` and edit the service name to `Calculator`

Remove the existing `SayHello` function and add these new functions:

```
rpc Add (TwoIntsRequest) returns (IntValueReply);
rpc Multiply (TwoIntsRequest) returns (IntValueReply);
rpc Divide (TwoIntsRequest) returns (FloatValueReply);
```

Inspecting these three new calculator functions, you should be able to determine that we have Add, Multiply and Divide operations, with 3 message types:

1. A request containing two integers
2. A reply containing a single integer
3. A reply containing a single float

Remove the existing message contracts and add a ‘`TwoIntsRequest`’ message:

```
message TwoIntsRequest {
   int32 valueA = 1;
   int32 valueB = 2;
}
```

Now add the `IntValueReply` and `FloatValueReply` message contracts. Both should return a variable named `value` (one should be an integer and the other a float!)

The next thing to do is to have the framework generate the underlying abstract classes for us to use to build our service. This is done for us automatically at compilation time by the gRPC protocol buffer compiler, using the proto file we have just modified.

Right-click the solution in the Solution Explorer and build it.

You will see a number of build errors but, importantly, this action will have compiled the proto file and generated base classes which will perform marshalling and allow us access to the message contents as C# types.

Let’s take a look at the base class:
Find the Services folder in the Solution Explorer and right-click the *GreeterService.cs* class file. Choose Rename and change the class name to *CalculatorService.cs*. Visual Studio should ask you if you would like to perform a rename of all instances of this class name, choose Yes.

Open *CalculatorService.cs*. There are a number of errors. The first is that the base class is not recognised. This is because it refers to the, now removed, greeter service. Modify the class definition so it reads:

```
public class CalculatorService : Calculator.CalculatorBase
```

Now right-click on the word Calculator and choose ‘Go To Definition’.

A new file should open: *CalcGrpc.cs*. This file is the **automatically generated implementation of your contract as a C# base class**. The `Calculator` class is a partial class. If you inspect the first few lines of this class you will note the setup of grpc Marshaller fields for conversion of messages to and from C# objects.

In previous labs we have used very rudimentary means to convert data into a byte stream, primarily based on the fact that our data types are known to both client and server and therefore the encoding and decoding can be hard-coded. Clearly, this is not feasible for a protocol designed to be used more generally. When exploring Web-based messaging we have seen the use of XML and JSON – very human-readable, easily parse-able and well supported but not concise or compact.

Whilst it is not essential, it may be useful to examine how protobuf messages are marshalled to and from bytes. The core of this is ‘encoding’ and you can read more about it here:

https://developers.google.com/protocol-buffers/docs/encoding

In the abstract `CalculatorBase` class, further down the page, you should find virtual Add, Multiply and Divide methods, ready to be overridden by us in order to implement behaviours.

You will also find `BindService` methods, which allow these methods to be called from the gRPC service back end.

Finally, let's take a look at *Program.cs*. This file contains calls to configure the ASP.NET runtime. This will be different depending on what type of application you are developing. In our case you can see:

```
services.AddGrpc();
```

This adds the gRPC back end service to ensure gRPC requests are correctly managed and for use by the automatically generated files.

```
endpoints.MapGrpcService<CalculatorService>();
```

This maps incoming gRPC calls to the concrete service implementation.

Create Template
Modify Proto file
Update Service
Create Client
Test

Let's now update that service implementation so it actually implements the methods described in our proto file.

Open *CalculatorService.cs.* Remove the existing `SayHello` method then, in its place type:

```
public override
```

Once you insert a space after the override keyword Visual Studio will automatically advise the virtual base methods that can be overridden. Select the Add method and a basic method implementation will be generated for you. Currently this just calls the same method in the base class which, in turn, throws a not-implemented exception.

Remove the line `return base.Add(request, context);`

Replace with:

```
return Task.FromResult(new IntValueReply
{
    Value = request.ValueA + request.ValueB
});
```

Note:

If you haven't used the above syntax before, you may find it useful to know that it is functionally comparable to:

```
IntValueReply response = new IntValueReply();
response.Value = request.ValueA + request.ValueB;
return Task.FromResult(response);
```

Now implement the Multiply and Divide methods in the same way.

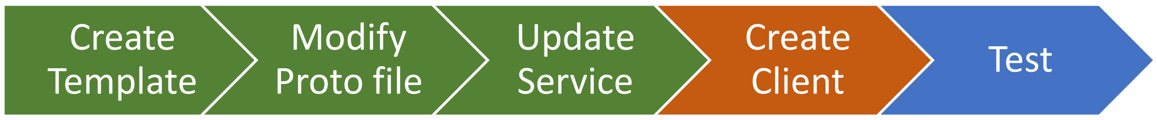
Remember that Divide returns a FloatValueReply.

Also remember that ValueA and ValueB are integers so you will need to ensure you perform appropriate casting when dividing and creating a float Value to return.

Once done your server is set up and ready! Easy!

You should no longer have compiler errors and your solution should build without a problem.

The biggest issue now is that there is no way to test your server's gRPC responses in isolation from a client. There are tools which allow this: https://docs.microsoft.com/en-us/aspnet/core/grpc/test-tools but they need to be set up to find or be given the proto file(s) in order for them to understand the contract. Given the simplicity of our service, we'll implement a client for testing and evaluation purposes.



For our client we will be implementing a WPF application.

Before we do that, we'll create a library that allows us to make gRPC calls.

We will do this for two reasons:

1. There is a 'known issue' in WPF that prevents the automatic generation of the gRPC backend at compile time (see: https://github.com/dotnet/wpf/issues/810) by using proto files.
2. By moving the RPC calls to a library we improve our abstraction which gives us looser coupling. This means our client could be changed to use a different RPC (or any other) protocol in the future by just swapping out the library. No change necessary to the client code itself.

Add a new Class Library (.NET Core) project to the solution. Give it the name *GrpcClientLibrary.*

Now right-click the project in the Solution Explorer and click 'Manage NuGet Packages'. Browse for and install the latest stable releases of these packages:

- Grpc.Net.Client
- Grpc.Tools
- Google.Protobuf

We have now got all of the libraries we need to make sense of proto files and for the client-side gRPC backend; importantly the generation of proxies and marshalling.

Before that can happen, though, we need to actually add the proto file. Remember, this is the contract and both sides **must** know what is in the contract before they can make sense of the data sent (very different to if we were using self-descriptive JSON/XML).

Add a new folder to your GrpcClientLibrary project called *Protos*

Copy the calc.proto file from your GrpcService project into this new folder.

Open the client-side proto file and modify the namespace to GrpcClientLibrary:

```
option csharp_namespace = "GrpcClientLibrary";
```

This is a language-specific option that ensures the automatically generated files are placed in the correct namespace for use when compiled.

Now comes an interesting, and ever-more required, part of the process…

Often, when you add NuGet packages or modify configuration options on Visual Studio projects, the project file is automatically changed so as to ensure that Visual Studio and the compiler behave as required. In our case, because we have added a folder and file manually, and the gRPC tooling isn't (yet?) clever enough automatically determine if we are creating a client or server, we need to modify the project file ourselves to identify where this file is and what it is.

Double-click the GrpcClientLibrary project (or right-click and choose 'Edit Project File').

Replace the XML below (if it exists):

```
<Protobuf Include="Protos\calc.proto" GrpcServices="Server" />
```

With:

```
<Protobuf Include="Protos\calc.proto" GrpcServices="Client" />
```

If the XML does not exist, you can add it manually immediately above the `</Project>` tag, remembering to make the change above (this may be because you haven't built yet).

The Protobuf element must be inside `<ItemGroup> </ItemGroup>` tags

Check the folder and file name are correct, then press Save.

Build the project.

The build process compiles the proto file and automatically generates the proxies we can use to make the back end gRPC calls. Let's use them!

Return to the GrpcSercvice project and open the launchSettings.json file inside the Properties folder. This file contains server configuration.

Find the `applicationUrl` string and make a note of the HTTP URL and port number.

Return to the GrpcClientLibrary project

Right-click Class1.cs and rename to *RemoteCalculator.cs* (Yes on the popup)

To the top of the class add:

```
private readonly GrpcChannel channel = GrpcChannel.ForAddress("<URL:PORT>");
private readonly Calculator.CalculatorClient client;
```

Replace `<URL:PORT>` with the url and port you made a note of earlier.

Create a parameterless constructor for the RemoteCalculator class. Inside it we will create an instance of CalculatorClient (using the channel we just created) and assign it to the `client` field:

```
client = new Calculator.CalculatorClient(channel);
```

If you haven't already, you'll need to add this using statement to the top of the file:

```
using Grpc.Net.Client;
```

Add a method:

```
public async Task<int> Add(int valueA, int valueB)
{    }
```

You'll notice that we are using the async keyword to enable this method to operate asynchronously and we are returning a Task<int> (an asynchronous operation returning an integer - see previous labs).

You may need to add this using statement to the top of the file:

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

Inside the method we will call into the `AddAsync` proxy method. Add the lines:

```
TwoIntsRequest request = new TwoIntsRequest {
    ValueA = valueA,
    ValueB = valueB
};
IntValueReply response = await client.AddAsync(request);
return response.Value;
```

This code creates a request, using the `TwoIntsRequest` message format, it then makes a call to the automatically generated `CalculatorClient.AddAsync` proxy method, passing the request. When a response is returned it is marshalled and sent back as an `IntValueReply` message. The Value property is the returned integer.

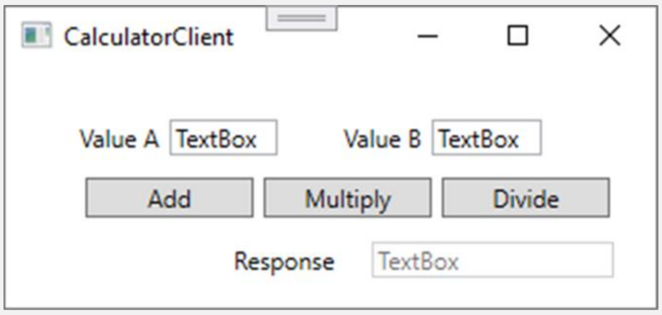
Create similar Multiply and Divide methods inside RemoteCalculator.

Now BUILD GrpcClientLibrary.

Once that's done, let's make the client UI.

Add a new WPF App to the solution. Give it the name *GrpcClient.*

Add labels, textboxes and buttons so your UI looks something like this:



Give the TextBoxes the names:

- ValueATb
- ValueBTb
- ResponseValueTb

Right-click the GrpcClient Dependencies in the Solution Explorer and choose 'Add Project Reference…'.

Ensure Projects -> Solution is selected on the left and select the project GrpcClientLibrary. Click OK.

Double-click the Add button in MainWindow.xaml to generate an event handler in the code behind (MainWindow.xaml.cs).

Modify the method signature by adding `async`

Then add the using statement to make use of our library:

```
using GrpcClientLibrary;
```

and create an instance of the RemoteCalculator inside the MainWindow class (I've called my instance `remoteCalc`)

Call the asynchronous Add method we just made, sending it the parsed values from our text boxes and awaiting the integer returned by the remote service.

```
int responseValue = await remoteCalc.Add(
    int.Parse(ValueATb.Text),
    int.Parse(ValueBTb.Text)
);
```

And, finally, set the response value text box to this value:

```
ResponseValueTb.Text = responseValue.ToString();
```

Repeat this process for the Multiply and Divide buttons.

And that should be it! Build your solution and everything should be ready to go.

You might like to set up multiple startup projects, so as to start the server and then client automatically when debugging.

Create Template
Modify Proto file
Update Service
Create Client
Test

Start your server and client.

Insert test values in the *Value A* and *Value B* text boxes and press the Add / Multiply / Divide buttons.

Change the values and click the buttons to see it update. Notice how the UI is not blocked whilst waiting for the remote service to return a value (thank you async!).

Next Steps:

- Add other calculator functions to your service, e.g.
    - Add a subtract remote function. Remember to update the .proto file on both server and client!
    - Create Add, Multiply, Divide and Subtract functions that accept floats. A float is a larger data type so requires more data to be sent across the network. Your client library should determine if the user has inserted floats or ints before calling the best remote procedure.
    - Experiment with different functions and messages. Add new functionality to your service and client.
- Make the RemoteCalculator class disposable inside the GrpcClientLibrary by implementing IDisposable.
    - Asynchronously shut down the channel and dispose of it using the provided methods
    - This will ensure that, should a client want to close a channel and then reopen, it can do so without the resources being tied up in inaccessible memory for an unknown period.
- Consider distributing the solution over the LAN network if you have multiple machines or are using the University computers. Connect a few clients to a single server. You will need to update the Kestrel settings and client IP address for this. Remember that you will to use port 5000 in the campus labs.