Asynchronous Communication and Services

Duplex Channel – WCF messaging

Last week we discussed duplex channel and remote callbacks.

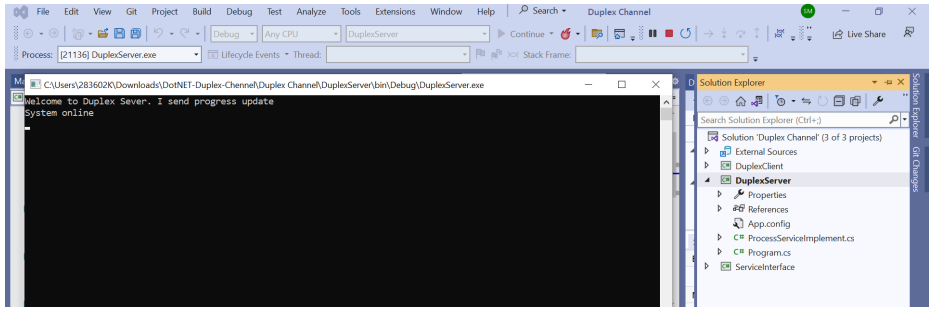
We also know about delegates and Async-Task-Await for asynchronous communication

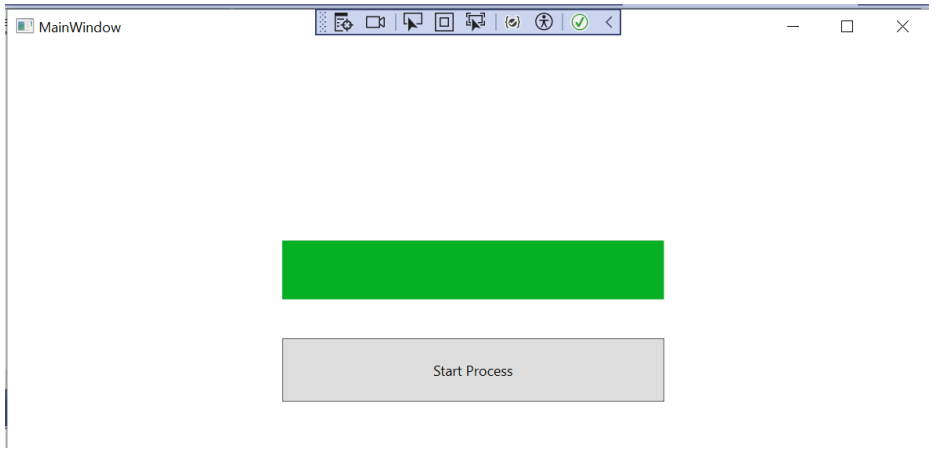
We also discussed Thread safety and Thread Synchronisation

Before we move to Service Oriented Architecture, let’s put all of them into a simple application.

Demo Objective

- A duplex server that process a long task and while processing it sends progress update to the clients.
- A WPF Client that sends the RPC request and also implement the remote callback, so it can display the progress in a progress bar.





Step 1: Create a DLL project for the service interface

- Make sure you refer to System.ServiceModel
- Create a process service interface – ProcessServiceInterface.cs

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

namespace ServiceInterface
{
    [ServiceContract(CallbackContract =typeof(ProcessServiceCallback))]
    5 references
    public interface ProcessServiceInterface
    {
        [OperationContract(IsOneWay = true) ]
        2 references
        void ProcessLongTask();
    }

    4 references
    public interface ProcessServiceCallback
    {
        [OperationContract(IsOneWay =true)]
        2 references
        void Progress(int percentageCompleted);
    }
}
```

Service interface details

- ProcessServiceInterface has only one method ProcessLongTask().
- **The operation is on-way, it means the caller does not need to wait. (Important!!)**
- ProcessServiceInterface callbacks to another interface names ProcessServiceCallback.
- ProcessServiceCallback has only one method Progress(int value).
- The callback will be implemented by the client. So, progress(int value) will be called through ProcessLongTask() in the server.
- [ServiceContract(CallbackContract=..)] specifies type of remote callback interface

```
using System.ServiceModel;

namespace ServiceInterface
{
    [ServiceContract(CallbackContract =typeof(ProcessServiceCallback))]
    5 references
    public interface ProcessServiceInterface
    {
        [OperationContract(IsOneWay = true) ]
        2 references
        void ProcessLongTask();
    }

    4 references
    public interface ProcessServiceCallback
    {
        [OperationContract(IsOneWay =true)]
        2 references
        void Progress(int percentageCompleted);
    }
}
```

Step 2: Create a console (.NET FrameWork) as the Duplex server

It refers to the interface dll and the System.serviceModel

Implement two classes – ProcessServiceImplement.cs – that implements the processServiceInterface

Program.cs – that binds the service and run the server

ProcessServiceImplement.cs

- ProcessLongTask simulates the long process by sleeping in a thread in a for loop.
- After each iteration it send the update by getting the callbackchannel of the specified callback interface and the remote calling the callback method, i.e, progress(i).

```
using System.ServiceModel;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using ServiceInterface;

namespace DuplexServer
{
    [ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Multiple,
        UseSynchronizationContext =false)]
    internal class ProcessServiceImplement : ProcessServiceInterface
    {

        public void ProcessLongTask()
        {
            for(int i = 1; i <= 100; i++)
            {
                Thread.Sleep(50);
                OperationContext.Current.
                    GetCallbackChannel<ProcessServiceCallback>().Progress(i);
            }
        }
    }
}
```

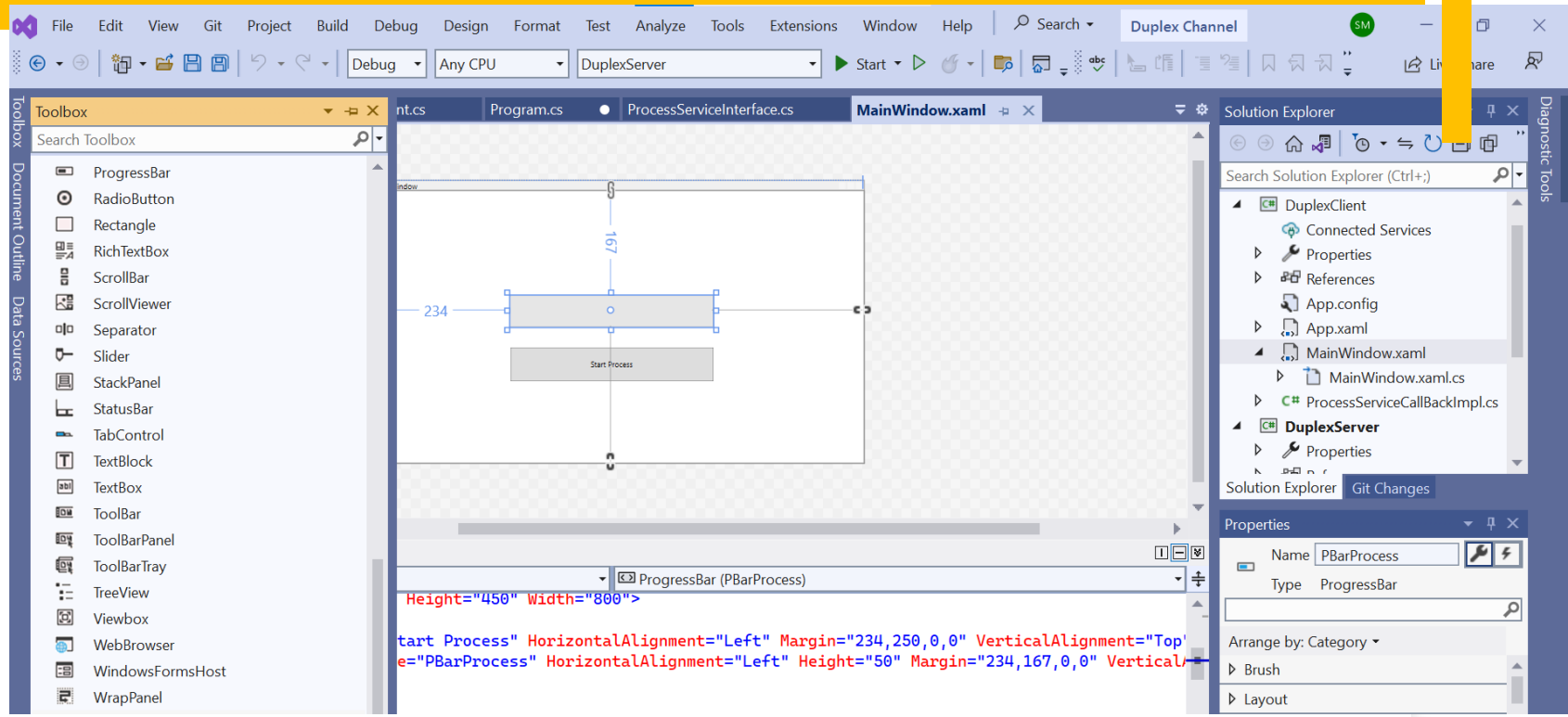
Program.cs – the console server

```
using System.ServiceModel;
using ServiceInterface;

namespace DuplexServer
{
    0 references
    internal class Program
    {
        0 references
        static void Main(string[] args)
        {
            Console.WriteLine("Welcome to Duplex Sever. I send progress update");

            NetTcpBinding netTcpBinding = new NetTcpBinding();
            ServiceHost serviceHost = new ServiceHost(typeof(ProcessServiceImplement)
            serviceHost.AddServiceEndpoint(typeof(ProcessServiceInterface),
                netTcpBinding, "net.tcp://0.0.0.0:8100/ProcessService");
            serviceHost.Open();
            Console.WriteLine("System online");
            Console.ReadLine();
            serviceHost.Close();
        }
    }
}
```

- This is the same server, we are building in previous lectures and tutorials.

Step 3: Create a WPF (.NET Console) project

- The client has only a button and progress bar.
- When the button is pressed, it calls the ProcessLongTask() (RPC) in the server.
- The progress bar gets updated when the server performs the remote callback on the client.
- You can find the progress bar in the all WPF tools. We name it ‘PBarProces’.
- Create reference to service dll and System.ServiceModel



A new thing: The client implements the callback interface

- Add a class in the client project – ProcessServiceCallBackImpl.cs
- Its constructor refers to the client window.
- It implements the progress method of the callback interface, which calls a progress functional in the client window, i.e., passing the received value from the service to the client window.

```
using System.Text;
using System.Threading.Tasks;
using ServiceInterface;

namespace DuplexClient
{
    [CallbackBehavior(ConcurrencyMode = ConcurrencyMode.Multiple, UseSynchronizationContext = false)]
    2 references
    internal class ProcessServiceCallBackImpl : ProcessServiceCallback
    {
        private MainWindow window;

        1 reference
        public ProcessServiceCallBackImpl(MainWindow window)
        {
            this.window = window;
        }
        2 references
        public void Progress(int percentageMainCompleted)
        {
            window.Progress(percentageMainCompleted);
        }
    }
}
```

WPF Client – MainWindow.xaml.cs

- Define a delegate for ProcessLongTask
- Connect to server using DuplexChannelFactory

```
namespace DuplexClient
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public delegate void ProcessLongTask();

    4 references
    public partial class MainWindow : Window
    {
        private ProcessServiceInterface foob;
        private ProcessServiceCallback foobCallback;
        private ProcessLongTask longTask;
        IAsyncResult asyncResult;
        0 references
        public MainWindow()
        {
            InitializeComponent();
            DuplexChannelFactory<ProcessServiceInterface> foobFactory;
            NetTcpBinding netTcpBinding = new NetTcpBinding();
            string URL = "net.tcp://localhost:8100/ProcessService";
            foobCallback = new ProcessServiceCallBackImpl(this);
            foobFactory = new DuplexChannelFactory<ProcessServiceInterface>
                (foobCallback, netTcpBinding, URL);
            foob = foobFactory.CreateChannel();

        }
```

WPF Client – MainWindow.xaml.cs

- When the button is clicked, the RPC is performed. Note we are not doing any EndInvoke, since it is a one-way call.
- The progress method uses Dispatcher to update the PBarProcess.Value

```
1 reference
private void Button_Click(object sender, RoutedEventArgs e)
{
    longTask = foob.ProcessLongTask;
    asyncResult = longTask.BeginInvoke(null, null);
    //foob.ProcessLongTask();
}

1 reference
public void Progress(int percentageCompleted)
{
    PBarProcess.Dispatcher.Invoke(new Action(() => {PBarProcess.Value = percentageCompleted;}));
    if (percentageCompleted == 100)
    {
        asyncResult.AsyncWaitHandle.Close();
    }
}
```

Distributed Computing

The Internet and Web Services

Services

- Services are collections of functionalities
  - A bunch of functions that do related stuff
  - A grouping of components
- Have special rules to reduce network reliance
  - Internal objects cannot be passed outside of the service boundary
  - Services are stand alone and aren’t incorporated as part of other services
    - Instead other services might use your service

Problem with Services

- Generally rely on an RPC infrastructure to provide access
  - RPC infrastructure allows for functionality usage
  - Also allows for *security*
- Cross platform/framework services are hard to implement.

Web Services

- Services, but the RPC infrastructure is the World Wide Web.
  - This just means HTTP/S.

Why do this?

- Basically, you want a connection that is:
  - Shared between all frameworks
  - Easy to use
    - Otherwise why not use raw sockets?
  - Able to operate on zero trust
    - This is useful when dealing with clients you don't control or know.
- This has evolved over time

Raw Sockets

- Most simple solution.
  - Just jam bytecode over a TCP or UDP connection
- Pros:
  - You have total control!
  - Works everywhere!
- Cons:
  - *Tons* of work
  - You have to do *everything at least once* on any platform.
  - No inbuilt security or permissions system

Build your own protocol

- Build your own system for passing stuff around a network
- Pros:
  - Can include any kind of extension you need
- Cons:
  - You gotta build it first
  - On every system you want to use it on.
    - And you have to make sure your extensions work.

Just borrow someone else's

- Just steal another protocol and slap your application on top
- Pros:
  - You don't have to do any work
- Cons:
  - You're limited to where it's implemented
  - You're limited by the implementation
    - If it doesn't do what you want, too bad!

Best protocol to steal

- Should be everywhere
  - All devices should already have a good means of accessing it
- Should be as freeform as possible
  - You don’t want it interfering with your code
- Should support a lot of extensions already
  - But preferably those are optional
- Should be easy to use!

But where could we find such a protocol?

A Brief History Of HTTP

- Developed in 1989 at CERN by Tim Berners-Lee.
  - He wanted to share research documents.
  - Now we share memes.

HTTP

- Became core of the World Wide Web in the 90s.
  - This meant it was a *very* popular network protocol
- Used by random flash games, but *also* by large organisations for documents
  - Has very good integration with modern development tools and distributed environments
- Is a *very* simple protocol at heart.
  - Just sends markup text over TCP.

The Problem with Protocols

- So we're all using HTML
- But…. What format do we use for returned data?
  - HTML?
    - But what if I'm not a human?
  - Just raw text?
    - Now I've got to parse this too!
  - Something Else?
    - Protocols *on* protocols?!

HTML

- Structured document made of markup tags
  - <head></head>
  -
  - <h1>This is a header</h1>
- Can be used to transfer information
  -
- Tedious, still requires some way of serializing objects on either side.

What else can be sent over HTTP?

- HTML is a document type suitable for Browsers.
- There's nothing stopping you from sending other document types over HTTP
  - Example, pictures, files, youtube…
- So, why not just come up with a document type that makes more sense for data?

XML

- A 1998 rethink of the HTML standard
  - Now you can define any document
  - <quizzes>, <thingObject>, <html> too!
- Allows you to use the DOM (Document Object Model) to traverse the document much in the same way as an object (or map/dictionary)
  - document[“quiz”][“questions”][0][“answer”]
    - Returns “Damascus” or something.

XML

- XML revolutionised web services
  - Was a far easier way of transferring information
  - First class data type in web languages
  - Microsoft loves XML, so it is used all throughout C#
    - Loves it so much that you can turn any object into XML by pushing it through the XmlSerializer object in System.
- Is the foundation of most web service systems

XML example

```
<?xml version="1.0" standalone="yes"?>
<site>
<regions>
<africa>
<item id="item0">
<location>United States</location>
<quantity>1</quantity>
<name>duteous nine eighteen </name>
<payment>Creditcard</payment>
<description>
<parlist>
<listitem>
<text>
page rous lady idle authority capt professes stabs monster petition heave humbly removes rescue runs shady peace most piteous worser oak assembly holes patience but malice whoreson mirrors master tenants smocks yielded <keyword> officer embrace such fears distinction attires </keyword>
</text>
</listitem>
<listitem>
<text>
shepherd noble supposed dotage humble servilius theirs venus dismal wounds gum merely raise red breaks earth god folds closet captain dying reek
</text>
</listitem>
</parlist>
</description>
<shipping>Will ship internationally, See description for charges</shipping>
<incategory category="category540"/>
<incategory category="category418"/>
<incategory category="category985"/>
<incategory category="category787"/>
<incategory category="category12"/>
```

XML Bad?

- XML is verbose
  - Way too verbose
  - So many tags!
    - And those tags have attributes…
- XML obfuscates structure
  - Whitespace doesn't delineate structure, tags do.
  - Tags don't make visual sense to… anyone
- Parsers hard to write
  - The format doesn't lend itself to easy parsing
  - This makes it hard to extend other languages with XML support if they don't already have it.

JSON

- The no-frills 2000's alternative to XML.
- "Just write objects as if they are Javascript code"
- This has a number of benefits
  - Visually easy to understand (as it looks like JS code)
  - Explicitly structured
  - No verbosity whatsoever
- JSON is basically the gold standard of service data delivery in the modern age.

JSON example

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    },
    {
      "type": "mobile",
      "number": "123 456-7890"
    }
  ],
  "children": [],
  "spouse": null
}
```

JSON Good?

- It's shorter than XML
  - And theoretically compresses better
- It's native to web applications
  - JSON is just objects in Javascript
- It's *really* easy to write an interpreter for
  - Because it's already just Javascript
- It's very easy to read
  - At least for developer types

JSON Bad?

- JSON is hard to understand
  - Structure is obvious, but semantics are lost
- JSON has no attributes
  - This can result in strange ways of representing data
- JSON doesn't use DOM
  - This means that you have to have your own way of navigating the data structure
    - You can't just ask for all correct answers, you have to go find them yourself.

XML or JSON?

- Choose the best one for the job!
  - Do humans need to read it? Might use XML
  - SOAP? XML.
  - Are you using software that has inbuilt support for JSON? Might use that.
- Not major differences in data size
  - Usually a matter of bytes
  - Both compress surprisingly well

WSDL

- Web Services have a unique problem
  - Namely, they are hosted somewhere and the code is generally speaking hidden
  - But yet, they expect you to use them
- This is solved by a Web Service Description Language
  - WSDL is an actual specification used for SOAP based Web Services.
  - Can more generally represent a Web Service definition

WSDL Example

```
Content-Type: text/xml; charset=utf-8                    ← Response MIME Header
Content-Length: <length>                                 ← Usually calculated for you by browser/Javascript

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/soap/soap-envelope">
  <soap:Body>
    <AddResponse xmlns="http://www.curtin.edu.au/">      ← Called function
      <AddResult>3</AddResult>                           ← Return value
                                                         ← out parameters would go here
    </AddResponse>                                       ← End return
  </soap:Body>
</soap:Envelope>

<s:element name="AddResponse">                           ← Return value for Add()
  <s:complexType>
    <s:sequence>
      <s:element minOccurs="1" maxOccurs="1" name="AddResult" type="s:int" />
    </s:sequence>
  </s:complexType>
</s:element>
</s:schema>
</wsdl:types>
```

WSDL Example Some More

```
<wsdl:message name="AddSoapIn">                                  ← Possible messages for calling methods
  <wsdl:part name="parameters" element="tns:Add" />               ← Add() function parameters (refs prev page)
</wsdl:message>
<wsdl:message name="AddSoapOut">
  <wsdl:part name="parameters" element="tns:AddResponse" /> ← Add() return value (refs prev page)
</wsdl:message>

<wsdl:portType name="CalculatorSoap">                            ← Web Service Calculator: list of methods
                                                                   and what messages are involved
  <wsdl:operation name="Add">
    <wsdl:input message="tns:AddSoapIn" />                       ← Add() function call message
    <wsdl:output message="tns:AddSoapOut" />                     ← Add() function return value message
  </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="CalculatorSoap" type="tns:CalculatorSoap"> ← SOAP 1.1 message format definition
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http" />
  <wsdl:operation name="Add">
    <soap:operation soapAction="http://tempuri.org/Add" style="document" />
    <wsdl:input>
      <soap:body use="literal" />
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal" />
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
```

WSDL Example Even More

```
<wsdl:binding name="CalculatorSoap12" type="tns:CalculatorSoap">     ← SOAP 1.2 message format definition
  <soap12:binding transport="http://schemas.xmlsoap.org/soap/http" />
  <wsdl:operation name="Add">
    <soap12:operation soapAction="http://tempuri.org/Add" style="document" />
    <wsdl:input>
      <soap12:body use="literal" />
    </wsdl:input>
    <wsdl:output>
      <soap12:body use="literal" />
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>

<wsdl:service name="Calculator">                                     ← Put all the above together to define a Web Service
  <wsdl:port name="CalculatorSoap" binding="tns:CalculatorSoap">
    <soap:address location="http://localhost:1031/WebServices/Service.asmx" />
  </wsdl:port>
  <wsdl:port name="CalculatorSoap12" binding="tns:CalculatorSoap12">
    <soap12:address location="http://localhost:1031/WebServices/Service.asmx" />
  </wsdl:port>
</wsdl:service>

</wsdl:definitions>                                                  ← End WSDL definition
```

WSDL vs IDL

- We’ve already discussed an Interface Definition Language
- WSDL is like an IDL but for the web service
- Where it is not an actual definition, comes in the form of API documentation.
  - This happens a lot
    - A *lot*

SOA vs ROA

- The last major advance in Web Services is the distinction between
  - Service Oriented Architectures (SOA) and
  - Resource Oriented Architectures (ROA).
- This is essentially the fault line between
  - SOAP and
  - REST

SOA

- SOA focuses on the Service itself
  - Essentially it views the Web as a point to point protocol through which RPC/Data travels.
  - Generally a single URI for each Service.
    - And by Service, we mean the full collection of objects and functions you can use
  - Generally speaking unfriendly to user interfaces (browsers), and needs to be interpreted via another application.
  - Uses SOAP in general.

SOAP

- Simple Object Access Protocol
- Is an XML dialect that defines a sort of RPC tunnel through HTTP.
  - Each XML message is delivered via POST
  - Returned messages are in the returned SOAP/XML document.
- Downsides, not at all browser understandable
  - Post isn't something you do often as a browser
  - XML isn't particularly HTML friendly
    - Unless it *is* HTML

SOAP example

```
Content-Type: text/xml; charset=utf-8                    ← Request MIME Header
Content-Length: <length>                                 ← Usually calculated for you by browser/Javascript
SOAPAction: "http://www.curtin.edu.au/Add"               ← Function to call. NOTE: Part of the request header

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/soap/soap-envelope">
  <soap:Body>
    <Add xmlns="http://www.curtin.edu.au/">              ← Function to call (a bit of duplication with the header here!)
      <operand1>1</operand1>                             ← Parameter: int operand1
      <operand2>2</operand2>                             ← Parameter: int operand1
    </Add>                                               ← End function call
  </soap:Body>
</soap:Envelope>
```

SOAP Response Example

```
Content-Type: text/xml; charset=utf-8                    ← Response MIME Header
Content-Length: <length>                                 ← Usually calculated for you by browser/Javascript

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/soap/soap-envelope">
  <soap:Body>
    <AddResponse xmlns="http://www.curtin.edu.au/">      ← Called function
      <AddResult>3</AddResult>                           ← Return value
                                                         ← out parameters would go here
    </AddResponse>                                       ← End return
  </soap:Body>
</soap:Envelope>
```

ROA

- ROA are newer than SOA
  - More of a Web 2.0 thing, and we'll find out why
- Idea is not to be *a* service, but a *resource*
  - Difference is almost entirely superficial
  - Services are designed for high interoperability
    - Use them in many places, not just where intended!
  - Also designed in general to reflect the Web
  - This means they can be websites themselves!

API

- ROA systems generally forgo a strict WSDL in favour of a more generic API approach
  - In WSDL, you carefully define interface XML for service interface
  - In API, you give a document explaining the format of inputs, URI locations of services, and expected outputs.
  - WSDL is designed for programs to read, API's are designed for humans (ergo, programmers) to understand.

Why API?

- APIs fit the more interoperable ROA approach
  - ROA's are supposed to be used as a *resource*, so they are expected to be used by someone else for purposes other than a specific service.
  - Humans are going to be writing programs using ROAs
  - So, definitions need to be "human compatible"

REST

- Representational State Transfer
- Object based service access methodology
  - Less a distinct language and approach to SOA, more of a “general vibe”, or a strategy
  - Attempts to better represent the Web.
  - Attempts to, like the Web, be easily accessible and useable.

REST URIs

- REST doesn't believe in a single service URI.
  - Instead it has many URI's representing many parts of the resource.
  - Each URI represents an individual function, essentially
    - Ie http://thing.org/Calculator/Add and http://thing.org/Calculator/Divide for example.
  - Functions tend to follow HTTP methods…
- Let's take a look at those real quick

HTTP Methods

- HTTP is not just one operation!
  - Hopefully by now you know at least of GET (URI embedding of variables) and POST (embeds variables in request body)
- There are in fact 5 main methods:
  - GET (variable in URL ie: .../something?var=2&i=10)
  - POST (variables in body)
  - PUT
  - PATCH
  - DELETE

REST methods

- REST methods try to follow the general idea of HTTP methods, but with regards to data.
- This is very useful for say, a RESTful Data Tier!
- This is *entirely optional*, as is everything in REST.
- Let’s look at some examples...

REST Get

- RESTful GET should “get” whatever resource the URI indicates
  - GET http://thing.org/Calculator/Settings should get the current settings value.
  - This isn’t a rule per se, but it is a *good idea* not to let your GET methods change things

REST Post

- POST Creates a new element in the resource the URI indicates
  - This can override the current entry if one so wishes.
    - But not always!
  - POSTing to http://thing.org/Calculator/Settings should let me replace the settings wholesale. Or maybe it sets up the settings for a new calculator.

REST Put

- PUT generally lets one update a single entry
  - Or, alternatively it creates one if it doesn't already exist.
  - Basically assume the same as POST under most circumstances

REST Patch

- PATCH lets you update a specific item in the resource
  - Does not create it if it does not exist, that's an error.
  - If you're using a Replace to Update sort of approach, may not be different to POST

REST Delete

- DELETE… deletes stuff
  - It’s pretty simple honestly

The Minimum REST

- GET to get stuff
- POST to create/update/replace stuff
- DELETE to delete stuff?
  - You *could* POST to a delete function, and it would delete the matching item
  - So you don't actually need DELETE

REST as webpages

- GET and POST are basic functions used by most web pages
  - And are the core functions of a RESTful interface
- This means that REST systems can be hybrid Webpage/Web Service systems
  - A web page for GET, uses POST via web forms to change state of the system
  - Web page as a basic interface!
  - What if the service returns bits of web pages...?