seed
beyond the obvious

Chapter -

Invoking Lengthy Operations

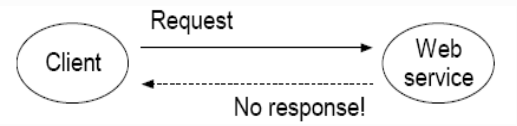
Objectives

- At the end of this chapter you will be able to understand
  - What if our web service involves manual processing that could take days to finish?
  - In this session we'll learn what the problems are and how to deal with them.

seed
beyond the obvious

Providing lengthy operations

- Suppose that we have a web service that processes business registration requests and that each request must be manually reviewed by a human being before it is approved. Then a business registration number is provided to the client.
- The problem is that this review process could take days and the web service client will be kept waiting for the HTTP response (assuming it is using SOAP over HTTP):



seed
beyond the obvious

Providing lengthy operations

- In that case, the HTTP client code in the client will think something may be wrong in the server. In order to avoid holding up the resources used by the connection, it will time out and terminate the connection.
- How to solve this problem??

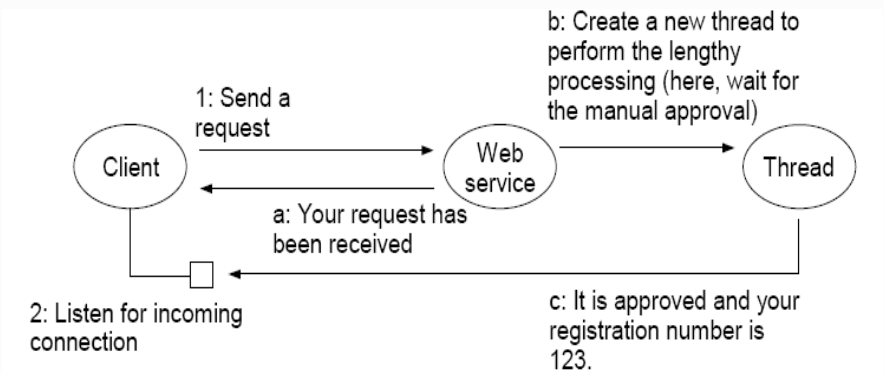
seed
beyond the obvious

Providing lengthy operations

- We can tell the client to send a request and then immediately listen on a port for incoming connection.
- On the server side, the web service will immediately return a short response saying that the request has been received for processing (not approved yet), then create a new thread to wait for the manual approval (so that the web service is free to serve other requests).
- When that thread gets the manual approval, it connects to the client and tells it that it has been approved and tells it the business registration number:

seed
beyond the obvious

Providing lengthy operations



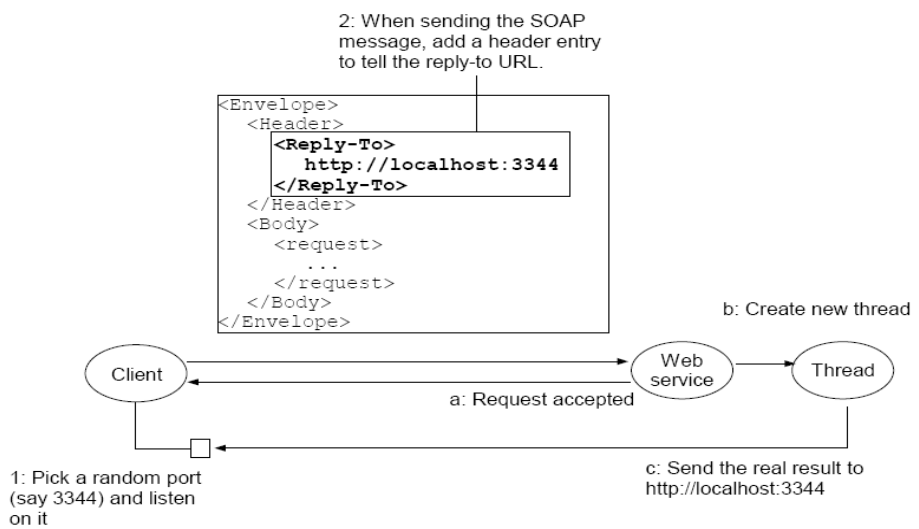
seed
beyond the obvious

Providing lengthy operations

- However, in step c above, how does it know the host name and port of the client??
- Therefore, when the client sends the request, it could pick a random port and then include its host name and the port number in the *reply-to* URL and include that URL in a SOAP header entry.
- This way, the background thread created by the web service can send the result to that URL. This is very much like having a *From address or Reply-To address* in an *email*. This is called "*WS-Addressing*":

seed
beyond the obvious

Providing lengthy operations



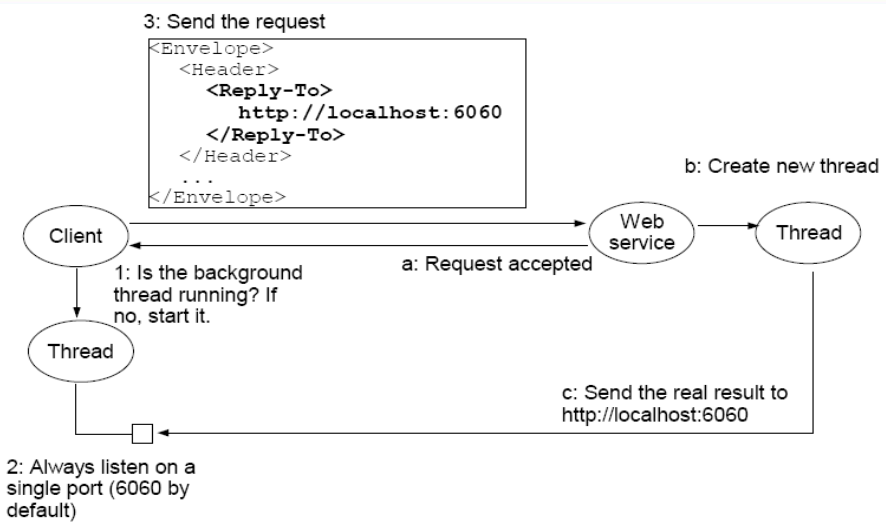
seed
beyond the obvious

Providing lengthy operations

- However, there is still a problem. If the client sends multiple requests to the web service or to different web services, if it opens a new port for each request, then it will use a lot of ports and will waste a lot of resources.
- Therefore, it will open a single port only and let a single background thread listening on it:

seed
beyond the obvious

Providing lengthy operations



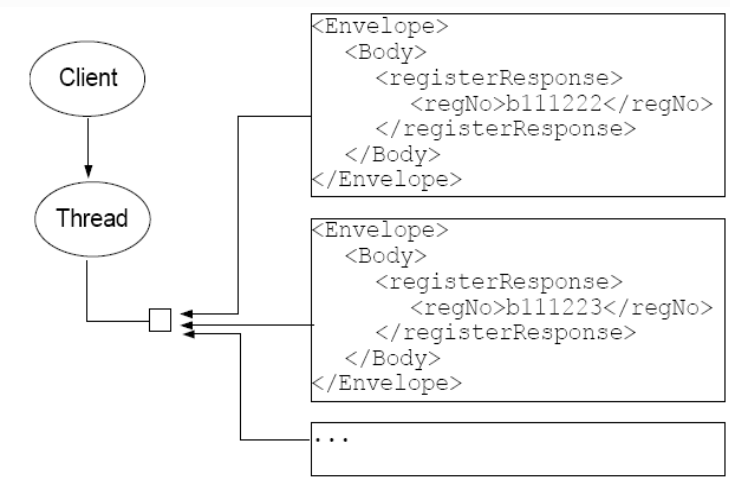
seed
beyond the obvious

Providing lengthy operations

- However, if multiple requests were sent, then multiple responses will arrive. Then in step c above, how can the background thread tell the response is for which request??

seed
beyond the obvious

Providing lengthy operations



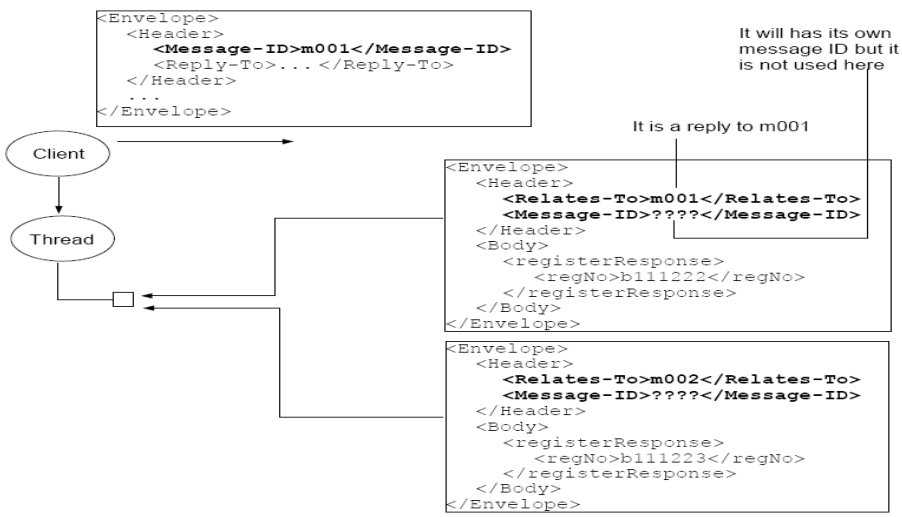
seed
beyond the obvious

Providing lengthy operations

- To solve this problem, when sending the request, the client will generate a unique message ID (e.g., m001) and include it in a header block.
- When the web service generates the response message, it will copy the message ID m001 into the *<Relates-To>* header block. This way, when the background thread receives the response, it knows that it is the response for request m001:

seed
beyond the obvious

Providing lengthy operations



seed
beyond the obvious

Providing lengthy operations

- All these *<Reply-To>*, *<Message-ID>*, *<Relates-To>* header blocks are part of the *WS-Addressing* standard

seed
beyond the obvious

Creating the WSDL for business registrations

Use this urn as the target namespace

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="urn:fake.gov:biz/reg"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="ManualService"
   targetNamespace="urn:fake.gov:biz/reg">
   <wsdl:types>
      <xsd:schema
         targetNamespace="urn:fake.gov:biz/reg"
         xmlns:xsd="http://www.w3.org/2001/XMLSchema">
         <xsd:element name="register">
            <xsd:complexType>
               <xsd:sequence>
                  <xsd:element name="bizName" type="xsd:string" />
                  <xsd:element name="ownerId" type="xsd:string" />
               </xsd:sequence>
            </xsd:complexType>
         </xsd:element>
         <xsd:element name="registerResponse">
            <xsd:complexType>
               <xsd:choice>
                  <xsd:element ref="tns:approved"></xsd:element>
                  <xsd:element ref="tns:rejected"></xsd:element>
               </xsd:choice>
            </xsd:complexType>
         </xsd:element>
         <xsd:element name="approved" type="xsd:string"></xsd:element>
         <xsd:element name="rejected" type="xsd:string"></xsd:element>
      </xsd:schema>
   </wsdl:types>
   ...
</wsdl:definitions>
```

This is the request. It contains the business name and the id of the business owner.

This is the response. It contains either an <approved> or a <rejected> element.

<choice> says that one and only one element below will be there

Refers to this element

```
<registerResponse>
   <approved>123</approved>
</registerResponse>
```

```
<registerResponse>
   <rejected>business name in use</rejected>
</registerResponse>
```


seed
beyond the obvious

Creating the WSDL for business registrations

- The rest of the WSDL file is as usual:

seed
beyond the obvious

Creating the WSDL for business registrations

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="urn:fake.gov:biz/reg"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="ManualService"
  targetNamespace="urn:fake.gov:biz/reg">
  <wsdl:types>
    <xsd:schema
      targetNamespace="urn:fake.gov:biz/reg"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <xsd:element name="register">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="bizName" type="xsd:string" />
            <xsd:element name="ownerId"   type="xsd:string" />
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="registerResponse">
        <xsd:complexType>
          <xsd:choice>
            <xsd:element ref="tns:approved"></xsd:element>
            <xsd:element ref="tns:rejected"></xsd:element>
          </xsd:choice>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="approved" type="xsd:string"></xsd:element>
      <xsd:element name="rejected" type="xsd:string"></xsd:element>
    </xsd:schema>
  </wsdl:types>
```


seed
beyond the obvious

Creating the WSDL for business registrations

```
<wsdl:message name="registerRequest">
  <wsdl:part name="parameters" element="tns:register" />
</wsdl:message>
<wsdl:message name="registerResponse">
  <wsdl:part name="parameters" element="tns:registerResponse"></wsdl:part>
</wsdl:message>
<wsdl:portType name="ManualService">
  <wsdl:operation name="register">
    <wsdl:input message="tns:registerRequest" />
    <wsdl:output message="tns:registerResponse" />
  </wsdl:operation>
</wsdl:portType>
<wsdl:binding name="ManualServiceSOAP" type="tns:ManualService">
  <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http" />
  <wsdl:operation name="register">
```


seed
beyond the obvious

Creating the WSDL for business registrations

```
      <soap:operation soapAction="urn:fake.gov:biz/reg/register" />
      <wsdl:input>
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="ManualService">
    <wsdl:port binding="tns:ManualServiceSOAP"
      name="ManualServiceSOAP">
      <soap:address
        location="http://localhost:8080/axis2/services/ManualService" />
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```


seed
beyond the obvious

The build.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project basedir="." default="jar.server">
  ...
  <property name="name" value="ManualService" />
  ...
  <target name="generate-service">
    <wsdl2code
      wsdlfilename="${name}.wsdl"
      serverside="true"
      generateservicexml="true"
      skipbuildxml="true"
      serversideinterface="true"
      namespacetopackages="urn:fake.gov:biz/reg=gov.fake.bizreg"
      targetsourcefolderlocation="src"
      targetresourcesfolderlocation="src/META-INF"
      overwrite="true"
      unwrap="true" />
    <replaceregexp
      file="src/META-INF/services.xml"
      match="${name}Skeleton"
      replace="${name}Impl" />
  </target>
  <target name="generate-client">
    <wsdl2code
      wsdlfilename="${name}.wsdl"
      skipbuildxml="true"
      namespacetopackages="urn:fake.gov:biz/reg=gov.fake.bizreg.client"
      targetsourcefolderlocation="src"
      overwrite="true"
      unwrap="true" />
  </target>
</project>
```


seed
beyond the obvious

Server and Client stub generation

- Then, generate the service stub and client stub. All these are pretty standard stuff.

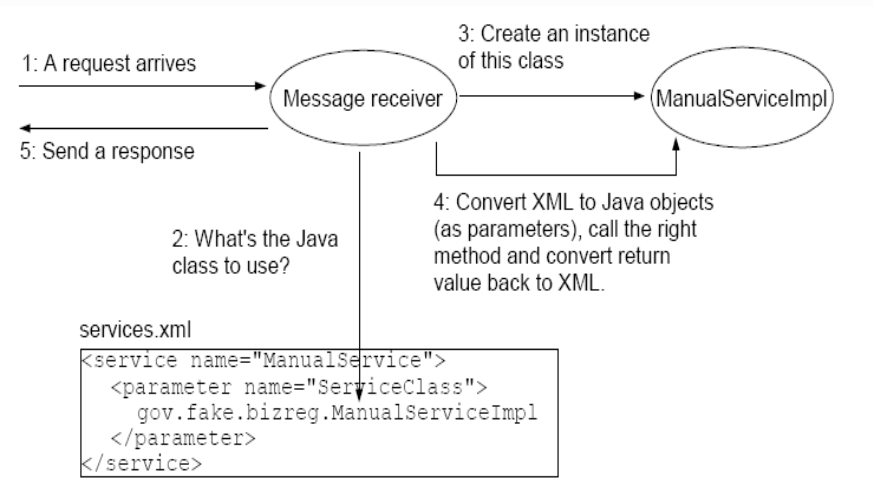
seed
beyond the obvious

Creating a new thread for lengthy processing

- In order to let the web service create a new thread to do the lengthy processing, we need to understand the concept of message receiver in Axis.
- There is a message receiver for each web service. When a request for our web service arrives, the message receiver will be handed the message.
- Message receiver functioning is as shown:

seed
beyond the obvious

Creating a new thread for lengthy processing – Message Receiver functioning



seed
beyond the obvious

Creating a new thread for lengthy processing – Message Receiver functioning

- All these are happening in the same thread by default.
- Now, we will tell our message receiver to create a new thread to call our implementation class, while returning an "accepted" response at the same time.
- To do that, we can modify our message receiver, which is the *ManualServiceMessageReceiverInOut* class generated by the <*wsdl2code*> Ant task:

seed
beyond the obvious

Creating a new thread for lengthy processing- Message Receiver

When a request (message) arrives, this method will be called. You're now overriding it.

```
import org.apache.axis2.AxisFault;
import org.apache.axis2.context.MessageContext;

public class ManualServiceMessageReceiverInOut extends
   AbstractInOutSyncMessageReceiver {
   public void receive(MessageContext messageCtx) throws AxisFault {
      messageCtx.setProperty(DO_ASYNC, "true");
      super.receive(messageCtx);
   }
   public void invokeBusinessLogic(
      ...
   }
}
```

Tell the parent class that the message should be handled asynchronously.

This method will perform data decoding and encoding and call your implementation class. Now it will be executed in a new thread.

seed
beyond the obvious

Creating a service implementation class
ManualServiceImpl.java

Return a hard-coded registration number for now

```
public class ManualServiceImpl implements ManualServiceSkeletonInterface {
  public RegisterResponse register(Register register) {
    System.out.println("Got request");
    String regNo = "123";
    try {
      Thread.sleep(5000);
    } catch (InterruptedException e) {
    }
    RegisterResponse response = new RegisterResponse();
    response.setApproved(regNo);
    return response;
  }
}
```

Sleep for five seconds to simulate human review

seed
beyond the obvious

Creating a service implementation class-
ManualServiceImpl.java

- Now the message receiver will call our register() method in a new thread.

seed
beyond the obvious

Creating an asynchronous client- BizRegClient.java

To encode the reply-to URL and message ID using the WS-Addressing standard, Axis provides a "module" to do that. This module is named "addressing". You can simply enable ("engage") it.

Internally the stub uses this object to call the web service

```
public class BizRegClient {
   public static void main(String[] args) throws RemoteException {
      ManualServiceStub stub = new ManualServiceStub();
      ServiceClient serviceClient = stub._getServiceClient();
      serviceClient.engageModule("addressing");
      Options options = serviceClient.getOptions();
      options.setUseSeparateListener(true);
      Register request = new Register();
      request.setBizName("Foo Ltd.");
      request.setOwnerId("Kent");
      ManualServiceCallbackHandler callback =
         new ManualServiceCallbackHandler() {
         public void receiveResultregister(RegisterResponse result) {
            System.out.println("Got result: " + result.getApproved());
         }
      };
      stub.startregister(request, callback);
      System.out.println("Request sent");
   }
}
```

The background thread will extract the response and pass it to your callback

Send the request and return immediately

This is the critical step. It causes the client to kick start the background thread to listen on port 6060 for the response. Conceptually, the background thread maintains an internal table like this:

Message ID	Callback
m001	
m002	
...	...

Callback1

Callback2

When it receives a response and finds that it is related to m001, it will call the callback for m001.

seed beyond the obvious

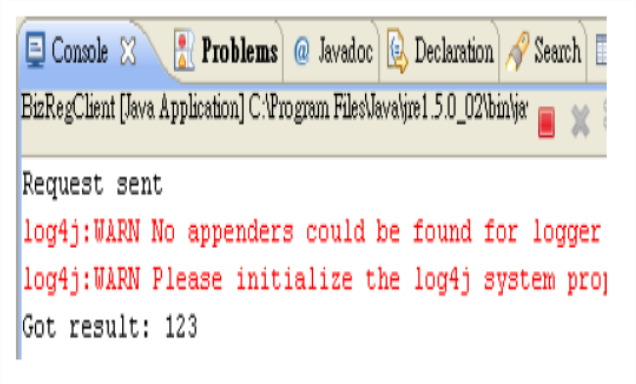
Enabling addressing on the server side

- For the web service to decode the *message ID* and *reply-to URL* from the SOAP message, we need to engage the addressing module in the web service. This is the case by default.
- We can verify that in global configuration file for Axis, *<axis_home>\conf\axis2.xml*:

```
<axisconfig name="AxisJava2.0">
  ...
  <module ref="addressing"/>
  ...
</axisconfig>
```


seed beyond the obvious

Result after deploying the service and running the client



seed
beyond the obvious

Inspecting the WS-Addressing header blocks

This is the target URL. Why is it needed? This allows routing the request message through intermediate hops because the target URL is maintained in the message.

The WS-Addressing namespace

As described before

```
<soapenv:Envelope
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:wsa="http://www.w3.org/2005/08/addressing">
   <soapenv:Header>
      <wsa:To>
         http://localhost:1234/axis2/services/ManualService
      </wsa:To>
      <wsa:ReplyTo>
         <wsa:Address>
            http://192.168.0.146:6060/axis2/services/ManualService25107363
         </wsa:Address>
      </wsa:ReplyTo>
      <wsa:MessageID>
         urn:uuid:E8A307C115655F0CFC1197866807896
      </wsa:MessageID>
      <wsa:Action>urn:fake.gov:biz/reg/register</wsa:Action>
   </soapenv:Header>
   <soapenv:Body>
      <ns1:register xmlns:ns1="urn:fake.gov:biz/reg">
         <bizName>Foo Ltd.</bizName>
         <ownerId>Kent</ownerId>
      </ns1:register>
   </soapenv:Body>
</soapenv:Envelope>
```

It allows the client to uniquely specify the operation it wants to call. This is also specified by the WS-Addressing standard.

```
<wsdl:definitions ...>
   ...
   <wsdl:binding name="ManualServiceSOAP" type="tns:ManualService">
      <soap:binding style="document"
         transport="http://schemas.xmlsoap.org/soap/http" />
      <wsdl:operation name="register">
         <soap:operation soapAction="urn:fake.gov:biz/reg/register" />
         <wsdl:input>
            <soap:body use="literal" />
         ...
         </wsdl:input>
         <wsdl:output>
            <soap:body use="literal" />
         </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>
   ...
</wsdl:definitions>
```


seed
beyond the obvious

Avoiding modifications to the message receiver

- Currently we're modifying *ManualServiceMessageReceiverInOut.java* which is generated by *<wsdl2code>*. This is no good as it will be overwritten if we run <wsdl2code> again.
- Therefore, a better way is to extend it. For example, create *ManualServiceReceiver.java* and move the *receive()* method into there:

seed
beyond the obvious

Avoiding modifications to the message receiver

```
public class ManualServiceReceiver extends ManualServiceMessageReceiverInOut {
  public void receive(MessageContext messageCtx) throws AxisFault {
    messageCtx.setProperty(DO_ASYNC, "true");

      super.receive(messageCtx);
    }
  }
```


seed
beyond the obvious

Modified build.xml

```
<target name="generate-service">
  <wsdl2code
    wsdlfilename="${name}.wsdl"
    serverside="true"
    generateservicexml="true"
    skipbuildxml="true"
    serversideinterface="true"
    namespacetopackages="urn:fake.gov:biz/reg=gov.fake.bizreg"
    targetsourcefolderlocation="src"
    targetresourcesfolderlocation="src/META-INF"
    overwrite="true"/>
  <replaceregexp
    file="src/META-INF/services.xml"
    match="${name}Skeleton"
    replace="${name}Impl" />
  <replaceregexp
    file="src/META-INF/services.xml"
    match="${name}MessageReceiverInOut"
    replace="${name}Receiver" />
</target>
```


seed
beyond the obvious

Quick Recap . . .

- To support a lengthy operation in a web service, its message receiver needs to enable the *DO_ASYNC* flag so that it creates a new thread to call our business logic and return the response in that thread. For this to work, the client needs to kick start a background thread to listen on a certain port for the response and include a reply-to URL in a header block in the request SOAP message.
- To distinguish which response is for which request, the client also needs to include a unique message ID into the message and the web service needs to copy that into a relates-to header block. *WS-Addressing* supports the encoding and decoding of the message ID, relates-to and reply-to URL.

seed
beyond the obvious

Quick Recap . . .

- WS-Addressing is implemented by a *module* called *"addressing"* in Axis. A module is just some functionality that can be enabled or disabled. When it is enabled, it is said to be *"engaged"*.
- The client API can be synchronous or asynchronous, independent of whether the transport is synchronous or not.
- If our code should proceed without waiting for the result, use the asynchronous API. If it must wait for the result, use the synchronous API.

seed
beyond the obvious