

## Experiment No. 8

### Title :

Implementation of SOAP Web services in JAVA Application.

### Objective:

To learn SOAP Web services using Eclipse IDE.

### Tools used:

Internet , Java application eclipse IDE.

### Prerequisite:

Understanding of Java tools, Eclipse, different types of web services etc.

### Theory:

**SOAP** stands for **Simple Object Access Protocol** is a **network platform** used in a web service to exchange or communicate data between two different machines on a network. It uses the XML format of data to transfer messages over the HTTP protocol. In Web services, SOAP allows the user request to interact with other programming languages. In this way, it provides a way to communicate between applications running on different platforms (Operating system), with programming languages and technologies used in web service.

#### SOAP Message:

The SOAP message contains the following information in the XML format, as given below.

- It contains information about the message structure and instructions during processing on the network.
- The SOAP contains an envelope that represents the starting and end of the message in the XML format.
- In the message, the header is an optional element that contains application-specific information such as authentication, authorization, and payment etc.
- A fault element is an optional element that shows an error message during the processing of the information.

#### Characteristics of SOAP:

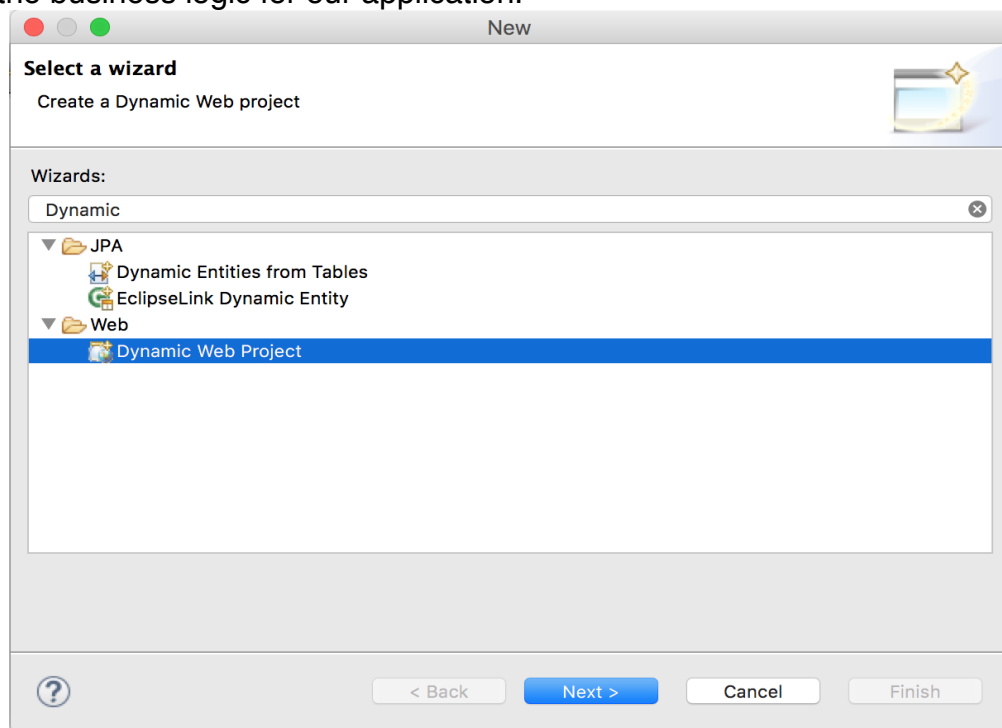
- It is an open standard protocol used in the web service to communicate via internet.
- It is used to broadcast a message over the network.
- It is used to call remote procedures and exchange documents.
- It can be used on any platform and can support multi-languages. So, it is a platform and language independent.

- It uses the XML format to send messages over the HTTP protocol.
- The structure of a SOAP message consists of an envelope, header, and body element.

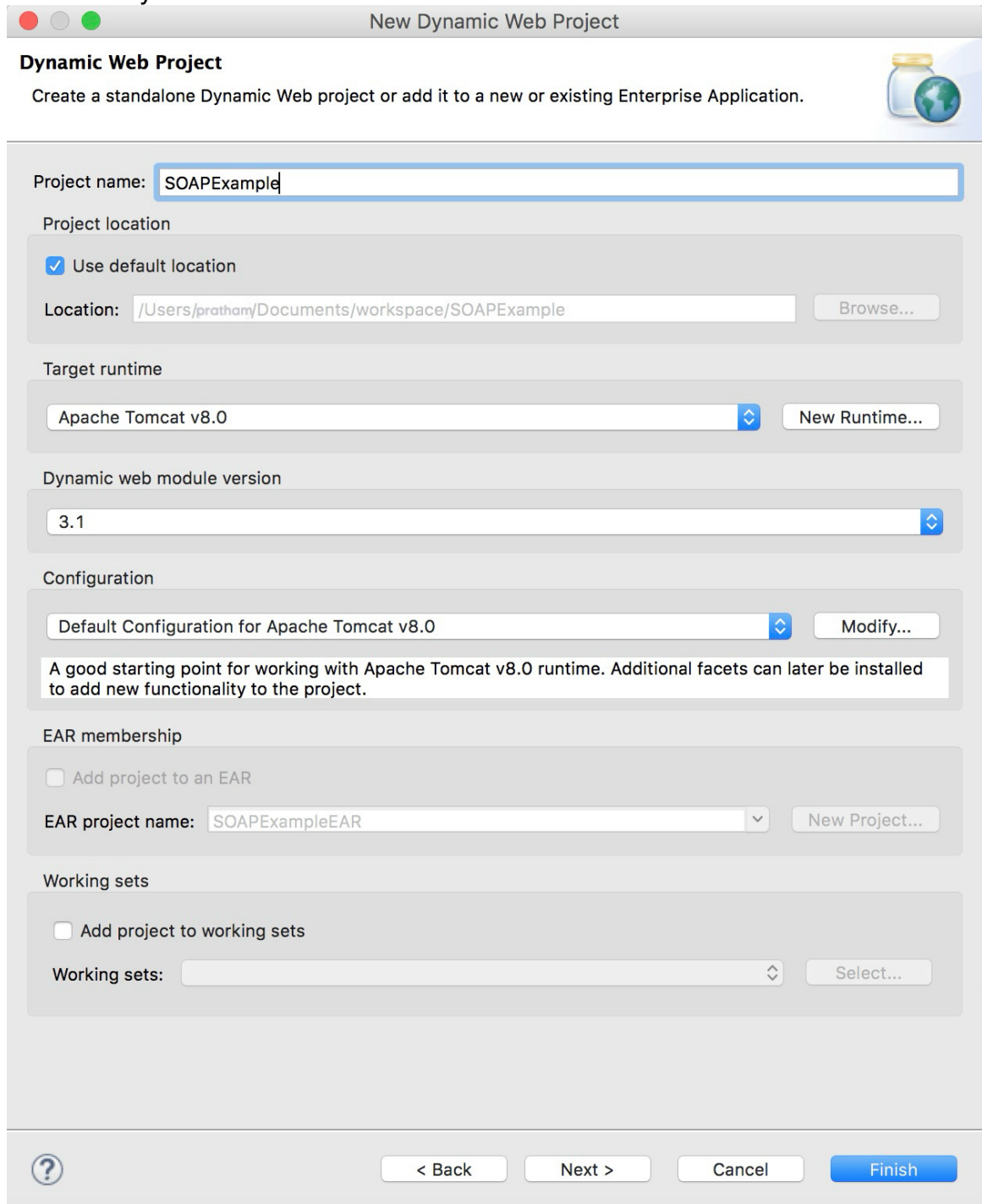
### Steps for performing soap web services in Java Application :

Soap Web services in java can be developed in different ways. We can create SOAP web service and its client program using Eclipse. Here we will not use JAX-WS, we will be using Apache Axis that is integrated in the Eclipse and provide quick and easy way to transform a application into Java Web Service and create client with test JSP page for testing purpose.

**Step 1 :-** First of all we will create simple Dynamic Web Project in Eclipse that will contain the business logic for our application.



**Step 2 :-** Click on Next button above and you will get next page to provide your web project your web project name and Target Runtime. We are using Apache Tomcat 8, you can use any other standard servlet container too.



**New Dynamic Web Project**

Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.

Project name:

Project location

☒ Use default location

Location:

Target runtime

Dynamic web module version

Configuration

A good starting point for working with Apache Tomcat v8.0 runtime. Additional facets can later be installed to add new functionality to the project.

EAR membership

☐ Add project to an EAR

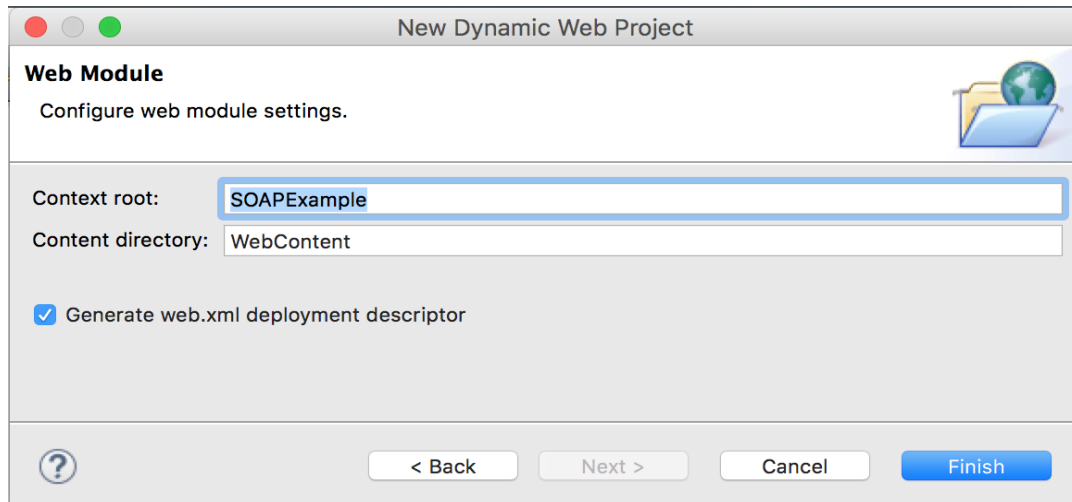
EAR project name:

Working sets

☐ Add project to working sets

Working sets:

**Step 3 :-** Click on Next and you will be asked to provide “Context Root” and Content Directory location. You can leave them as default.



**Step 4 :-** Click on Finish and Eclipse will create the project skeleton for you. Let's get started with our business logic. So for our example, we would like to publish a web service that can be used to add/delete/get an object. So first step is to create a model bean.

```
package com.journaldev.jaxws.beans;
import java.io.Serializable;
public class Person implements Serializable{
    private static final long serialVersionUID = -5577579081118070434L;
    private String name;
    private int age;
    private int id;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public int getId() {
        return id;
    }
}
```

```
public void setId(int id) {  
    this.id = id;  
}  
@Override  
public String toString(){  
    return id+"::"+name+"::"+age;  
}  
}
```

**Step 5 :-** Notice that above is a simple java bean, we are implementing Serializable interface because we will be transporting it over the network. We have also provided toString method implementation that will be used when we will print this object at client side. Next step is to create service classes, so we will have an interface as PersonService and it's simple implementation class PersonServiceImpl.

```
package com.journaldev.jaxws.service;  
import com.journaldev.jaxws.beans.Person;  
public interface PersonService {  
  
    public boolean addPerson(Person p);  
    public boolean deletePerson(int id);  
    public Person getPerson(int id);  
    public Person[] getAllPersons();  
}
```

**Step 6 :-** Below is the implementation service class, we are using Map to store Person objects as data source. In real world programming, we would like to save these into database tables.

```
package com.journaldev.jaxws.service;  
  
import java.util.HashMap;  
  
import java.util.Map;  
  
import java.util.Set;  
  
import com.journaldev.jaxws.beans.Person;  
  
public class PersonServiceImpl implements PersonService {  
  
    private static Map<Integer,Person> persons = new HashMap<Integer,Person>();  
  
    @Override  
    public boolean addPerson(Person p) {  
        if(persons.get(p.getId()) != null) return false;  
        persons.put(p.getId(), p);  
    }  
}
```

```
        return true;
    }

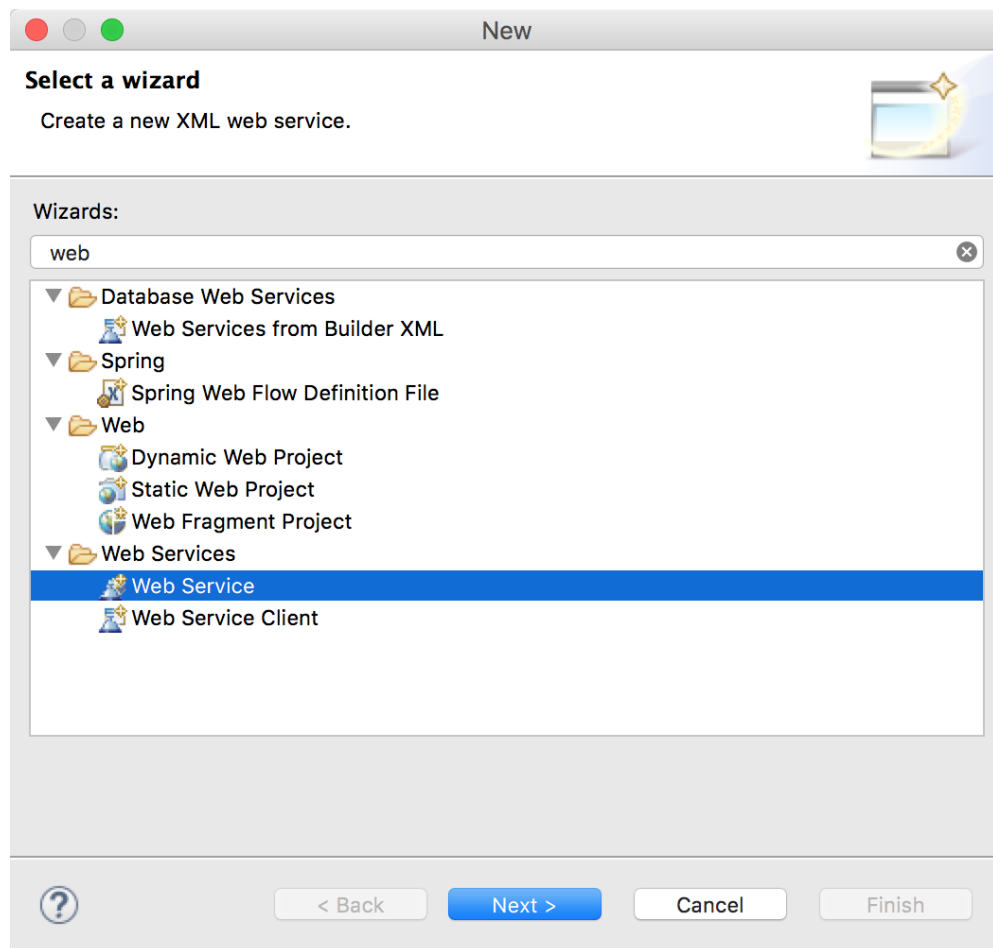
    @Override
    public boolean deletePerson(int id) {
        if(persons.get(id) == null) return false;
        persons.remove(id);
        return true;
    }

    @Override
    public Person getPerson(int id) {
        return persons.get(id);
    }

    @Override
    public Person[] getAllPersons() {
        Set<Integer> ids = persons.keySet();
        Person[] p = new Person[ids.size()];
        int i=0;
        for(Integer id : ids){
            p[i] = persons.get(id);
            i++;
        }
        return p;
    }
}
```

Since we will use these in a web service, there is no point of creating web pages here. Notice that we have no reference to any kind of web services classes in above code.

**Step 7 :-** Next step is to use Eclipse to create a web service application from this. Create a new project and select Web Service wizard.



**Step 8 :-** Click Next button and you will get a page where web service and it's client details have to be provided. This is the most important page in creating web service. Make sure you select "Web Service type" as "Bottom up Java bean Web Service" because we are implementing with bottom up approach. There are two ways to create

**web service:**

Web Service


**Web Services**

Select a service implementation or definition and move the sliders to set the level of service and client generation.

Web service type: Bottom up Java bean Web Service

Service implementation: com.journaldev.jaxws.service.PersonServiceImpl Browse...


Test service



Configuration:  
[Server runtime: Tomcat v8.0 Server](#)  
[Web service runtime: Apache Axis](#)  
[Service project: SOAPExample](#)

Client type: Java Proxy

Test client



Configuration:  
[Server runtime: Tomcat v8.0 Server](#)  
[Web service runtime: Apache Axis](#)  
[Client project: SOAPExampleClient](#)

☐ Publish the Web service

☐ Monitor the Web service

?

< Back

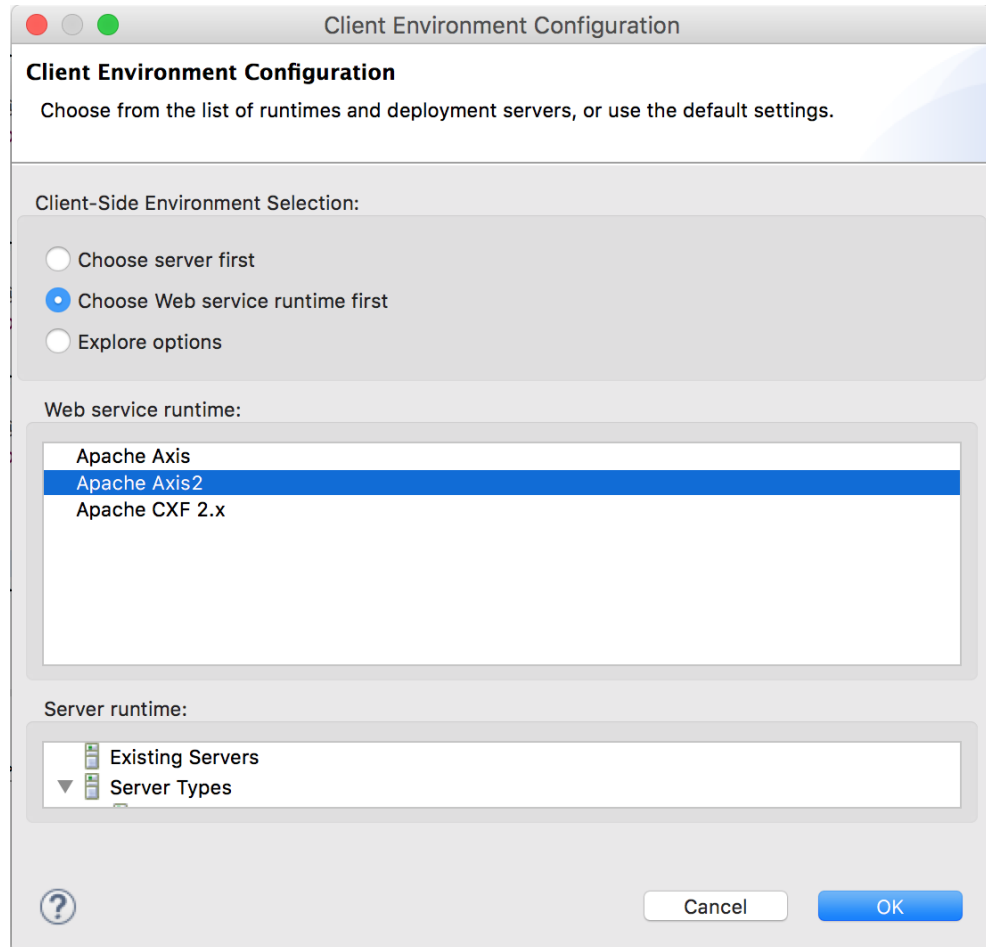
Next >

Cancel

Finish

**Step 9 :-** In the service implementation, provide the implementation class `PersonServiceImpl` fully classified path. Make sure you move the slider in service and client type to left side so that it can generate client program and also UI to test our web service. Check for the configurations in web service implementation, you should provide correct details for Server runtime, Web service runtime and service project. Usually they are auto populated and you don't need to make any changes here. For client configurations, you can provide the client project name as you like. I have left it to default as `SOAPExampleClient`. If you will click on the link for web service runtime, you will get different options as shown in below image. However I have left it as the default one.





The image shows a 'Client Environment Configuration' dialog box. It has a title bar with standard window controls (red, yellow, green buttons) and the title 'Client Environment Configuration'. Below the title bar, the text 'Client Environment Configuration' is displayed, followed by the instruction 'Choose from the list of runtimes and deployment servers, or use the default settings.'.

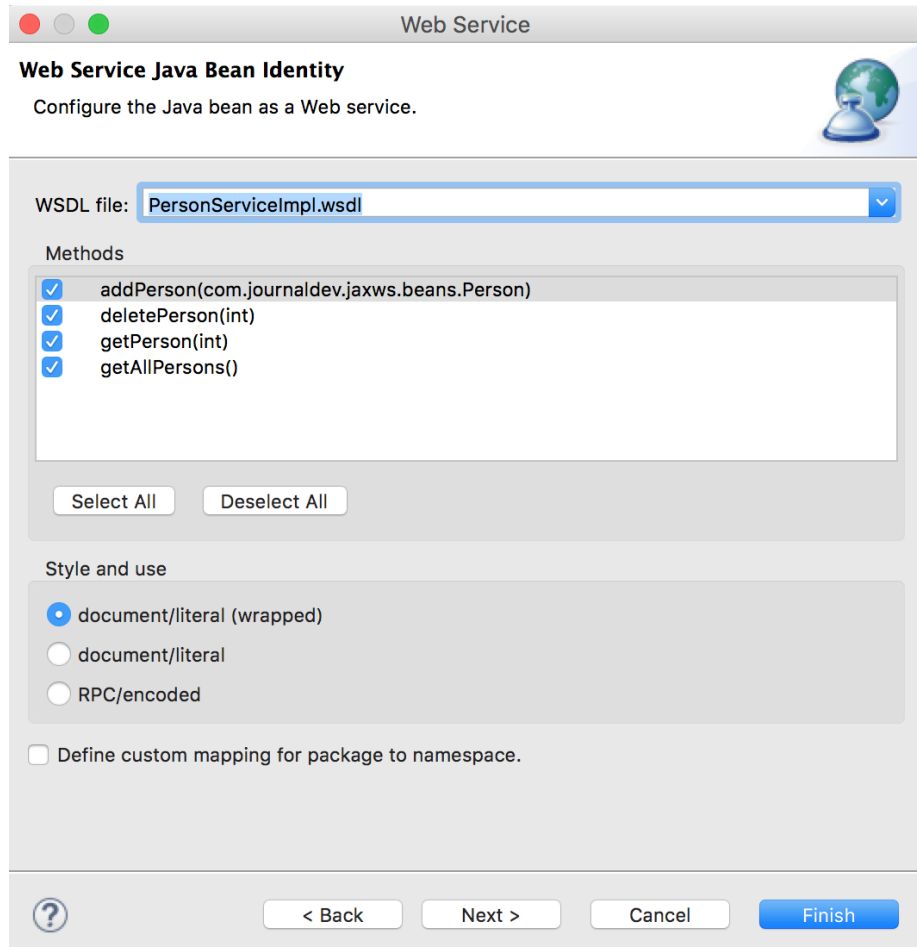
The main content area is divided into two sections:

- Client-Side Environment Selection:** This section contains three radio buttons:
  - ☐ Choose server first
  - ☒ Choose Web service runtime first
  - ☐ Explore options
- Web service runtime:** This section contains a list box with three items: 'Apache Axis', 'Apache Axis2' (which is highlighted with a blue background), and 'Apache CXF 2.x'.

Below the 'Web service runtime' section is the **Server runtime:** section, which contains a list box with two items: 'Existing Servers' and 'Server Types' (which is expanded, showing a dropdown arrow).

At the bottom of the dialog box, there is a question mark icon on the left, and 'Cancel' and 'OK' buttons on the right.

**Step 10 :-** Click on Next button and then you will be able to choose the methods that you want to expose as web service. You will also be able to choose the web service style as either document or literal. You can change the WSDL document name but it's good to have it with implementation class name to avoid confusion later on.



Web Service

**Web Service Java Bean Identity**

Configure the Java bean as a Web service.

WSDL file:

Methods

- ☒ addPerson(com.journaldev.jaxws.beans.Person)
- ☒ deletePerson(int)
- ☒ getPerson(int)
- ☒ getAllPersons()

Select All   Deselect All

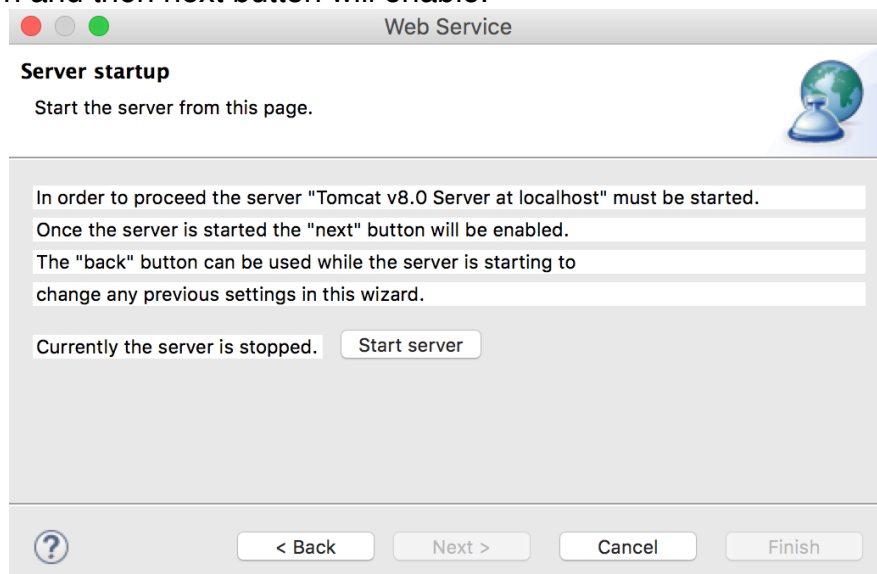
Style and use

- ☒ document/literal (wrapped)
- ☐ document/literal
- ☐ RPC/encoded

☐ Define custom mapping for package to namespace.

? < Back   Next >   Cancel   Finish

**Step 11 :-** Click on Next button and you will get server startup page, click on the “Start server” button and then next button will enable.



Web Service

**Server startup**

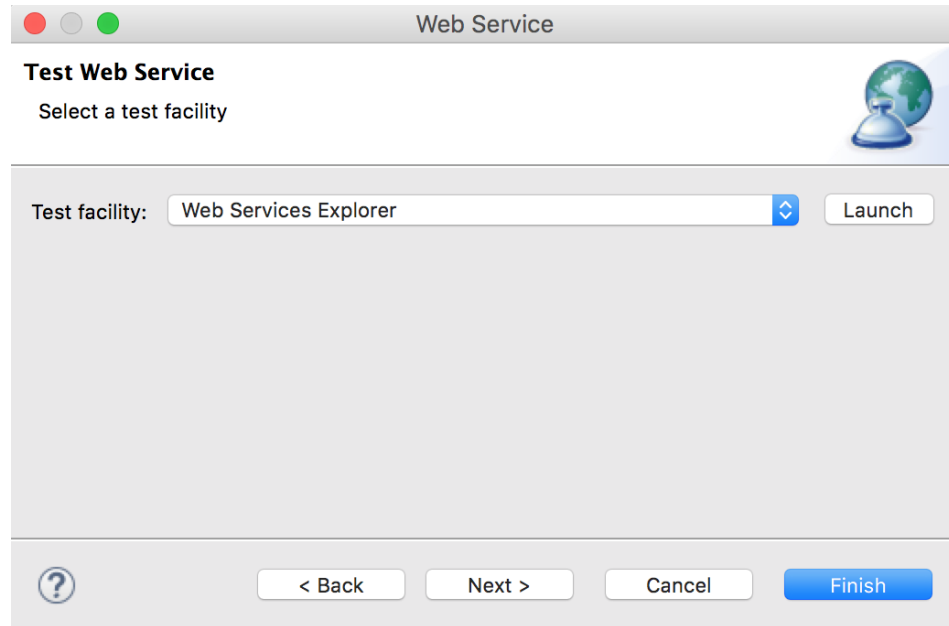
Start the server from this page.

In order to proceed the server "Tomcat v8.0 Server at localhost" must be started.  
Once the server is started the "next" button will be enabled.  
The "back" button can be used while the server is starting to  
change any previous settings in this wizard.

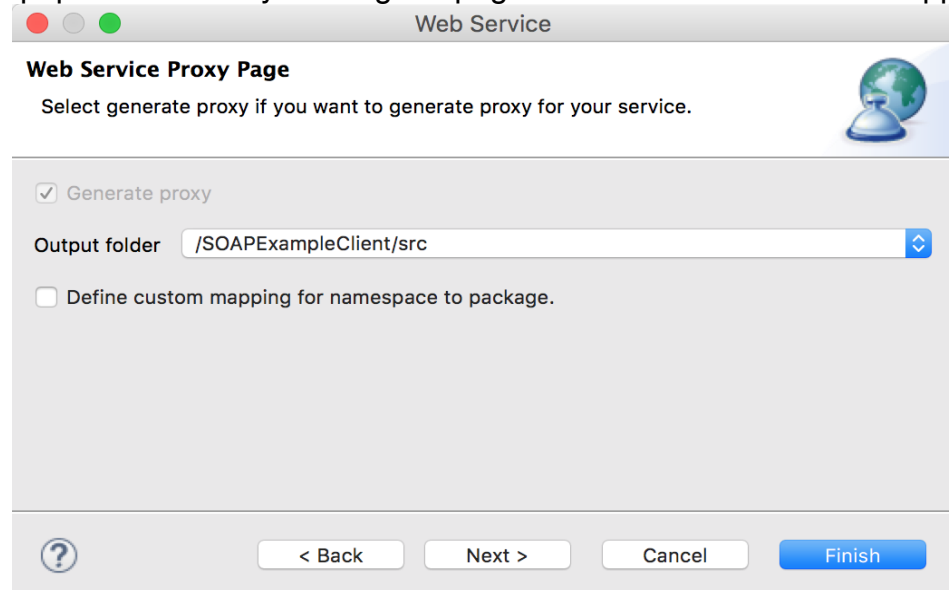
Currently the server is stopped.

? < Back   Next >   Cancel   Finish

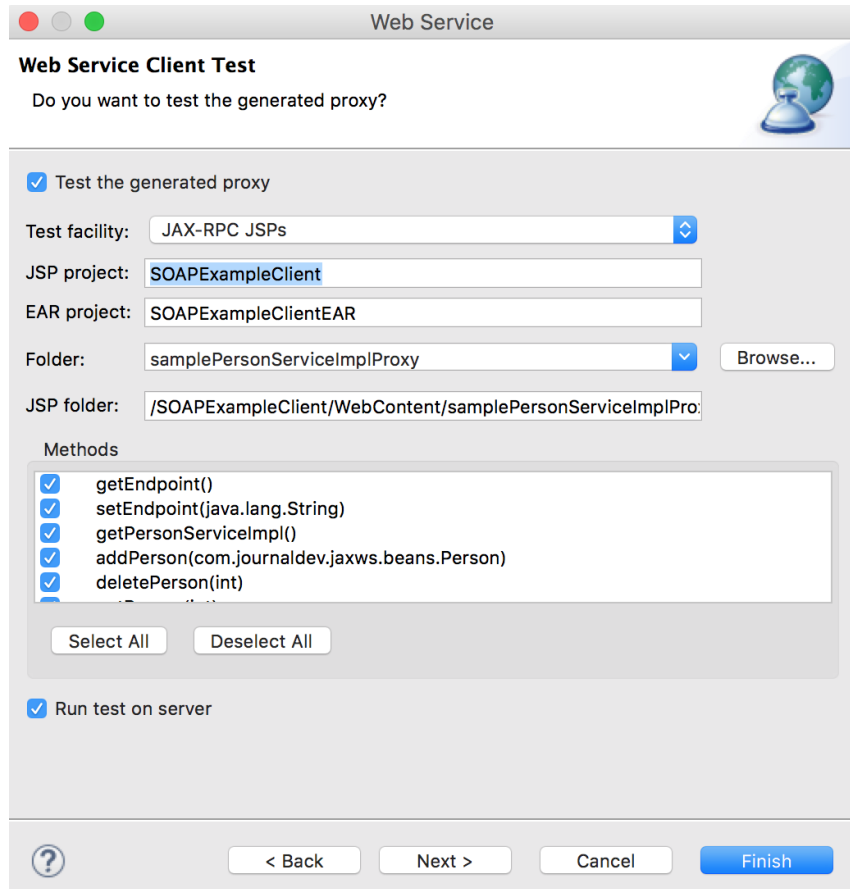
**Step 12 :-** Click on Next button and you will get a page to launch the “Web Services Explorer”.



**Step 13 :-** We can do some sanity testing here, but for our simple application I am ready to go ahead with client application creation. Click on the Next button in the Eclipse web services popup window and you will get a page for source folder for client application.



**Step 14 :-** Click on Next button and you will get different options to choose as test facility. I am going ahead with **JAX-RPC JSPs** so that client application will generate a JSP page that we can use.



Web Service Client Test

Do you want to test the generated proxy?

☒ Test the generated proxy

Test facility: JAX-RPC JSPs

JSP project: SOAExampleClient

EAR project: SOAExampleClientEAR

Folder: samplePersonServiceImplProxy Browse...

JSP folder: /SOAExampleClient/WebContent/samplePersonServiceImplPro

Methods

- ☒ getEndpoint()
- ☒ setEndpoint(java.lang.String)
- ☒ getPersonServiceImpl()
- ☒ addPerson(com.journaldev.jaxws.beans.Person)
- ☒ deletePerson(int)

Select All Deselect All

☒ Run test on server

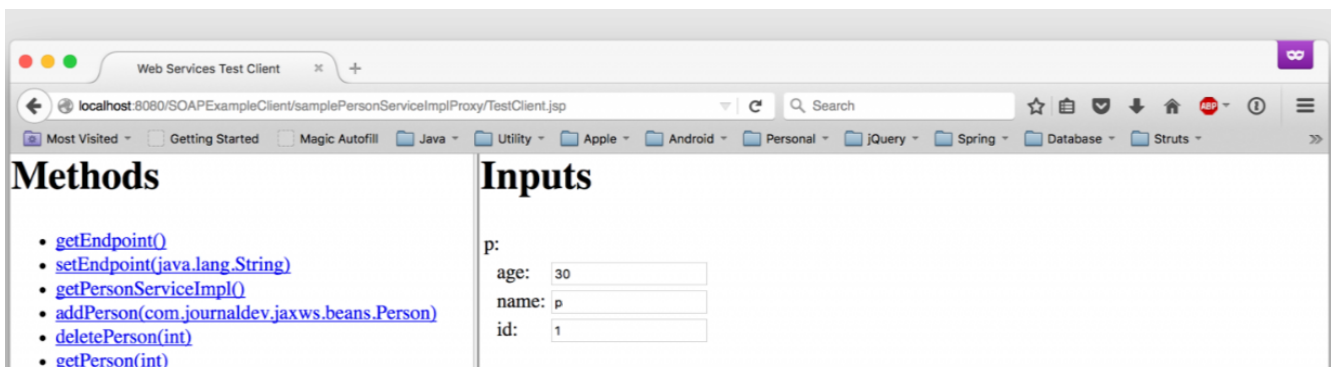
< Back Next > Cancel Finish

Notice the methods `getEndpoint()` and `setEndpoint(String)` added that we can use to get the web service endpoint URL and we can set it to some other URL in case we move our server to some other URL endpoint. Click on Finish button and Eclipse will create the client project in your workspace, it will also launch client test JSP page as shown below.

You can copy the URL and open in any browser you would like. Let's test some of the services that we have exposed and see the output.

### Eclipse SOAP Web Service Test

- addPerson



Web Services Test Client

localhost:8080/SOAExampleClient/samplePersonServiceImplProxy/TestClient.jsp

Methods

- [getEndpoint\(\)](#)
- [setEndpoint\(java.lang.String\)](#)
- [getPersonServiceImpl\(\)](#)
- [addPerson\(com.journaldev.jaxws.beans.Person\)](#)
- [deletePerson\(int\)](#)
- [getPerson\(int\)](#)

Inputs

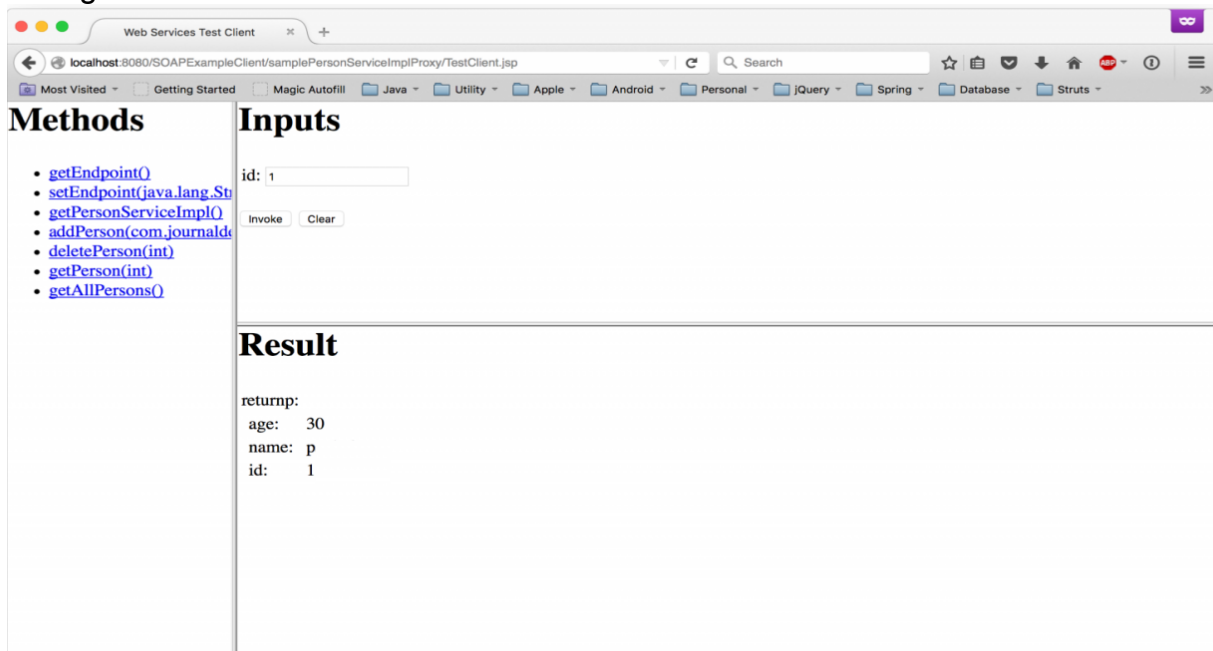
p:

age: 30

name: p

id: 1

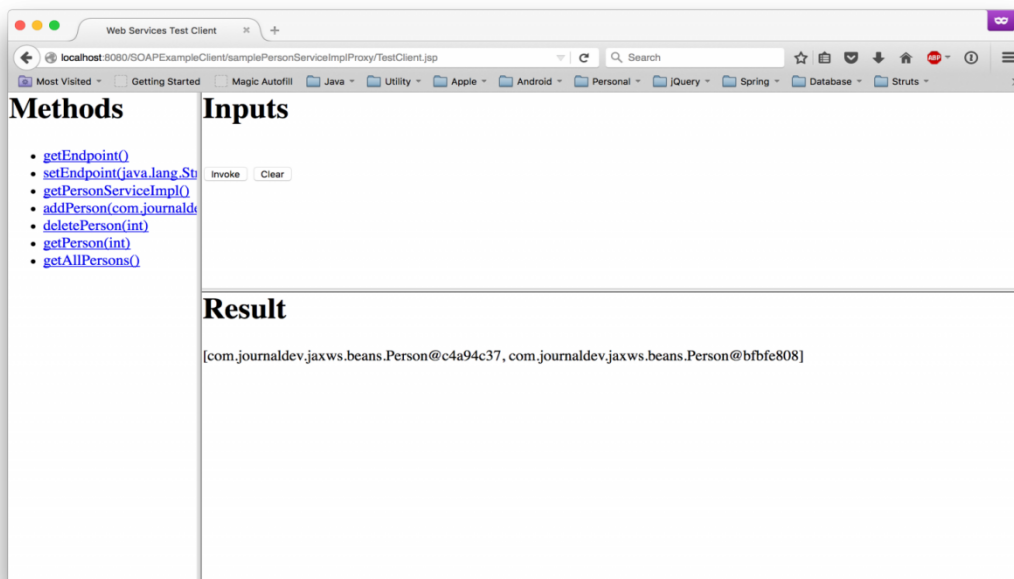
- **getPerson**



The screenshot shows the Web Services Test Client interface. The URL bar displays `localhost:8080/SOAPExampleClient/samplePersonServiceImplProxy/TestClient.jsp`. The **Methods** list on the left includes `getEndpoint()`, `setEndpoint(java.lang.String)`, `getPersonServiceImpl()`, `addPerson(com.journaldev.Person)`, `deletePerson(int)`, `getPerson(int)`, and `getAllPersons()`. The **Inputs** section shows a text field with the value `1` and buttons for `Invoke` and `Clear`. The **Result** section displays the following output:

```
returnp:
age: 30
name: p
id: 1
```

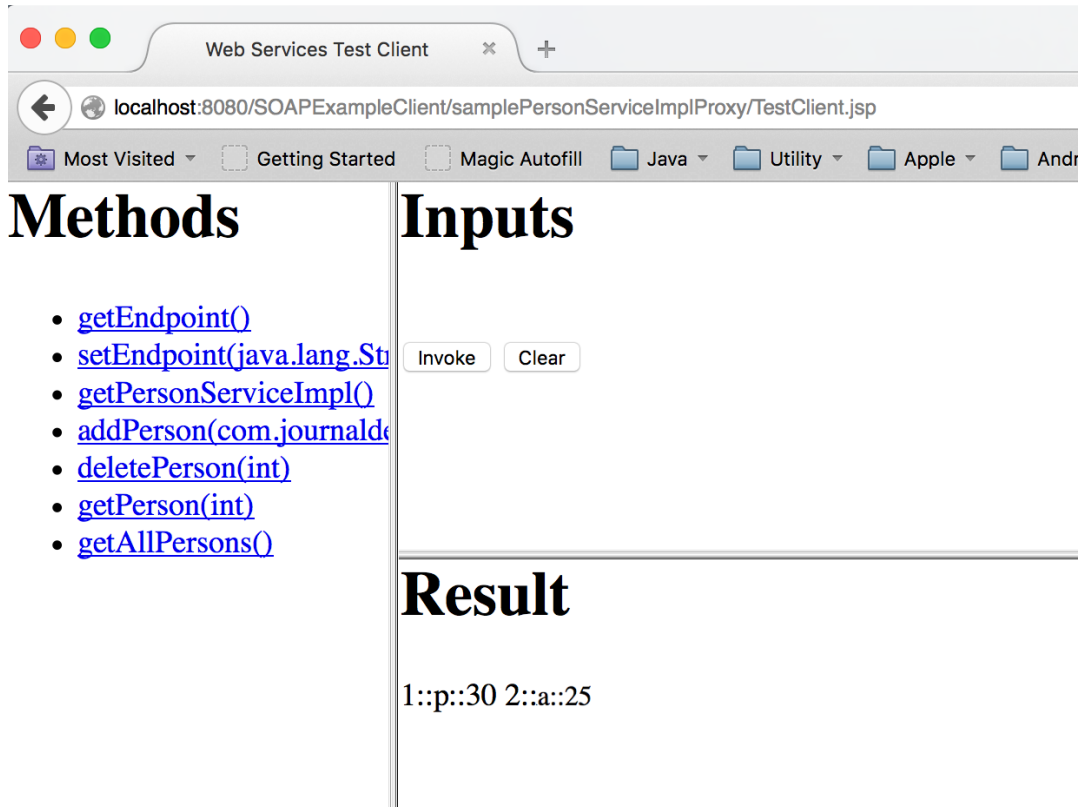
- **getAllPersons**



The screenshot shows the Web Services Test Client interface. The URL bar displays `localhost:8080/SOAPExampleClient/samplePersonServiceImplProxy/TestClient.jsp`. The **Methods** list on the left includes `getEndpoint()`, `setEndpoint(java.lang.String)`, `getPersonServiceImpl()`, `addPerson(com.journaldev.Person)`, `deletePerson(int)`, `getPerson(int)`, and `getAllPersons()`. The **Inputs** section shows buttons for `Invoke` and `Clear`. The **Result** section displays the following output:

```
[com.journaldev.jaxws.beans.Person@c4a94c37, com.journaldev.jaxws.beans.Person@bfbfe808]
```

After that we get below output, note that Eclipse is doing hot deployment here, so I didn't had to redeploy my application.



The screenshot shows a web browser window titled "Web Services Test Client". The address bar displays "localhost:8080/SOAPExampleClient/samplePersonServiceImplProxy/TestClient.jsp". Below the address bar is a navigation bar with links: "Most Visited", "Getting Started", "Magic Autofill", "Java", "Utility", "Apple", and "Android". The main content area is divided into two columns. The left column, titled "Methods", lists several SOAP methods: [getEndpoint\(\)](#), [setEndpoint\(java.lang.St](#), [getPersonServiceImpl\(\)](#), [addPerson\(com.journalde](#), [deletePerson\(int\)](#), [getPerson\(int\)](#), and [getAllPersons\(\)](#). The right column, titled "Inputs", contains two buttons: "Invoke" and "Clear". Below the "Inputs" section is a section titled "Result" which displays the output: "1::p::30 2::a::25".

So it looks like our web service and client applications are working fine, make sure to spend some time in looking at the client side stubs generated by Eclipse

### Conclusion:

Successfully Implemented of SOAP Web services in JAVA Application.