http://www.developer.comjava/creating-soap-web-services-with-jax-ws.html

Back to article

Creating SOAP Web Services with JAX-WS

October 24, 2013

Creating SOAP Web Services with JAX-WS

SOAP web service depends upon a number of technologies (such as UDDI, WSDL, SOAP, HTTP) and protocol to transport and transform data between a service provider and the consumer. Though it may seem overwhelming at first, with so many complex technologies intermingling together in a perfect symphony, creating SOAP web service in Java is actually pretty simple. One can almost overlook the intricacies of creating SOAP Web Service and focus on the business logic while Java takes care of the most on behalf of the programmer. Without delving into the core of SOAP technology lets get our hands dirty in creating one.

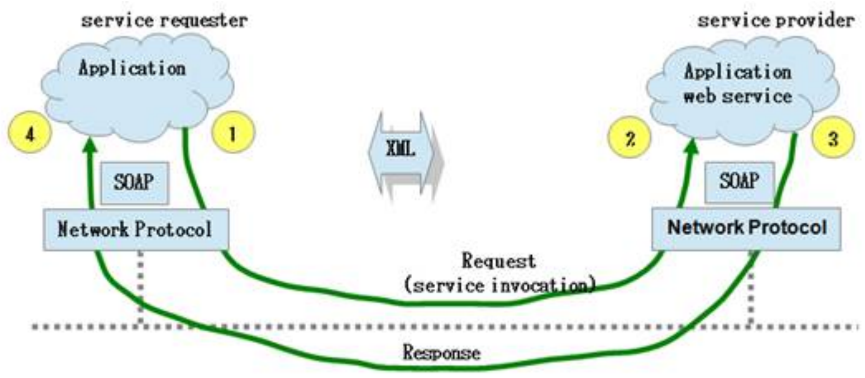


Fig 1: SOAP Request/Response

Writing SOAP Web Service from Scratch in Java EE7

Suppose, a web service producer/server provides a web method to display a list of products, and the client/consumer is a simple swing desktop application that wants to display the list of products in a table format. Thus our project structure can be illustrated as in Fig 2.

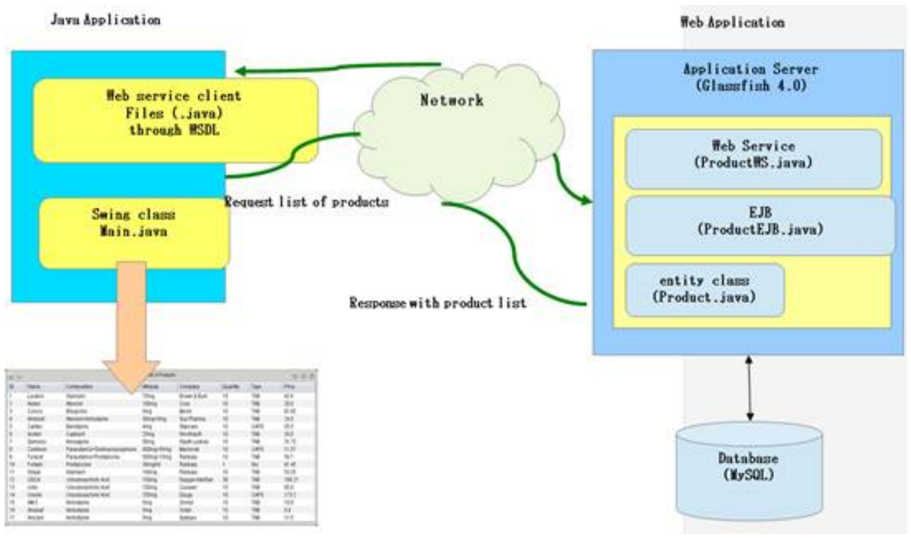
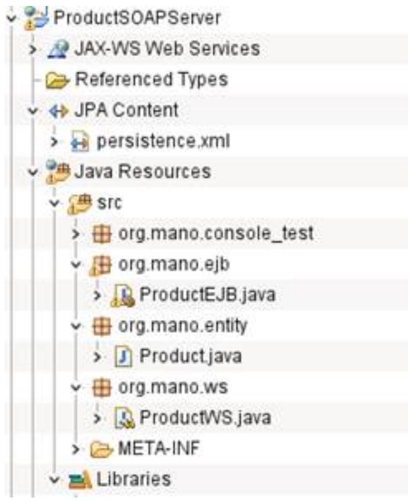


Fig 2: Hypothetical scenario demonstrating SOAP web service usage

Creating Web Service Producer

The Web Service producer in the example has a project structure and three classes as follows:



```
ProductSOAPServer
  JAX-WS Web Services
  Referenced Types
  JPA Content
    persistence.xml
  Java Resources
    src
      org.mano.console_test
      org.mano.ejb
        ProductEJB.java
      org.mano.entity
        Product.java
      org.mano.ws
        ProductWS.java
      META-INF
    Libraries
```

Fig 3: Project structure of SOAP web service producer

1) **Product.java**: This class represents the Product entity. Except for the annotations used, this class is nothing more than a simple POJO. This class actually maps to a table in the database. For a class to be an entity, it has to be annotated @Entity. Persistence provider (here, EclipseLink is used as a persistence provider, which comes bundled with Glassfish) recognizes it and transforms it into an entity and not just a simple POJO. @Id defines the unique identifier or primary key of the relation. @GeneratedValue represents that the unique identifier is auto generated by the database. Unique id generation may be done in various ways defined through 'strategy'. @NamedQuery defines a static query, which in our case is used to retrieve products from the database.

Product.java

```
package org.mano.entity;
//...import statements
@Entity
@Table(name="product_tbl")
@NamedQuery(name="Product.getAllProducts", query="SELECT p FROM Product p")
public class Product implements Serializable {
      private static final long serialVersionUID = 1L;
      @Id @GeneratedValue(strategy=GenerationType.AUTO)
      private Long id;
```

```
       private String name;
       private String composition;
       private String type;
       private Integer quantity;
       private String company;
       private String attribute;
       private Float price;

       //...constructors, getters and setters
}
```

2) **ProductEJB.java**: This is an EJB class annotated @Stateless and contains the main business logic (e.g. getAllProducts()). Observe that there is no 'new' object created for EntityManager. Since there can be only one entity instance of EntityManager with the same persistent identity, we delegated the responsibility of creating the EntityManager object to the container, i.e. app server with annotation @PersistenceContext. The 'unitName' in @PersistenceContext is the same name that is used in Persistence.xml (<persistence-unit name="ProductSOAPServer" transaction-type="JTA">).

ProductEJB.java

```
package org.mano.ejb;
//...import statements
@Stateless
public class ProductEJB {
       @PersistenceContext(unitName="ProductSOAPServer")
       private EntityManager entityManager;

       public List<Product> getAllProducts(){
             TypedQuery<Product> query = entityManager.createNamedQuery("Product.getAllProducts",Product.class);
             return query.getResultList();
       }
}
```

The above two classes have nothing to do with Web Service per se, they are just necessary supporting classes for the project. The class described below is the interface to the world, real deal, our Web Service class

3) **ProductWS.java**: With the annotation @WebService any POJO becomes a Web Service. That's it. Since we will be fetching data from the database we can simply use the EJB class (ProductEJB) as a controller or service provider of the database. Also EJB is a container managed, Like EntityManager, we do not use 'new' to create the ProductEJB object, a simple @EJB annotation is good enough.

ProductWS.java

```
 package org.mano.ws;
//...import statements
@WebService
public class ProductWS {

       @EJB
       ProductEJB productEJB;

       public List<Product> getAllProductDetails(){
             return productEJB.getAllProducts();
       }

       public List<String> getAllProductNames(){
             List<Product> list=productEJB.getAllProducts();
             List<String> productNames = new ArrayList<>();
             for(Product p:list){
                    productNames.add(p.getName());
             }
             return productNames;
       }
}
```

Persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
       <persistence-unit name="ProductSOAPServer" transaction-type="JTA">
       <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
             <jta-data-source>jdbc/mysqldsn</jta-data-source>
             <class>org.mano.entity.Product</class>
       </persistence-unit>
</persistence>
```

Note: The content of Persistence.xml is mostly auto generated (at least in Eclipse or Netbeans). There is a subtle aspect; observe that we have used – transaction-type="JTA" and <jta-data-source> jdbc/mysqldsn </jta-data-source>. For this you have to configure Glassfish and create a MYSQL JDBC pool from the admin console of Glassfish. Please refer connector/j usage...glassfish-config on how to configure MYSQL JDBC Pool and use transaction type as JTA.

Testing Web Service without a Client

Once the Web Service producer is created we can test whether the Web Service is running properly or not without writing a client/consumer application. Observe that there is a function named *getAllProductNames* in *ProductWS.java*. We shall call this function and observe the output. The simplest way to do this is to:

- Run the project in the application server (in this case Glassfish).
- Open the admin console: http://localhost:4848
- Click on **Applications** from **Common Task**, then click on your project from Applications
- Click on **View Endpoints** from the **Modules and Components** list
- Click on Tester link (there are subsequent steps, which are self-explanatory) to test the Web Service
- You may also click the WSDL link to see the XML (this file is used by the wsimport to create Web Service client files, as we'll see down the line).

Observe that following XML request and response are actually created while we test – *getAllProductNames.* So that claim of 'data exchange' through XML in SOAP is true after all :)

SOAP Request

```
<?xml version="1.0" encoding="UTF-8"?><S:Envelope
xmlns:S="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header/>
    <S:Body>
        <ns2:getAllProductNames xmlns:ns2="http://ws.mano.org/"/>
    </S:Body>
</S:Envelope>
```

SOAP Response

```
<?xml version="1.0" encoding="UTF-8"?><S:Envelope
xmlns:S="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header/>
    <S:Body>
        <ns2:getAllProductNamesResponse xmlns:ns2="http://ws.mano.org/">
            <return>Levalon</return>
            <return>Aloten</return>
             ...
            <return>Amanat</return>
        </ns2:getAllProductNamesResponse>
    </S:Body>
</S:Envelope>
```

Creating Web Service Consumer

We have tested the Web Service without writing a client app. Now create a simple Java application, which we shall use as a Web Service client or consumer. Thus, the Web Service consumer project has the following structure and files.

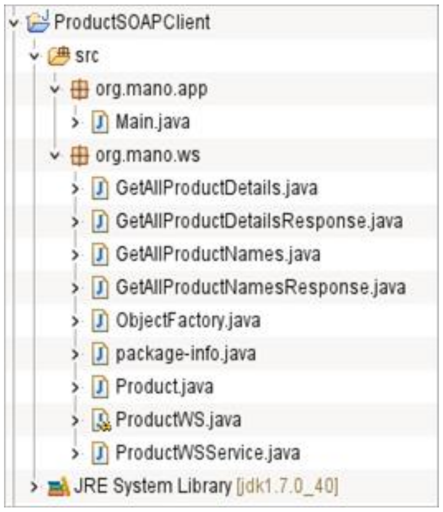


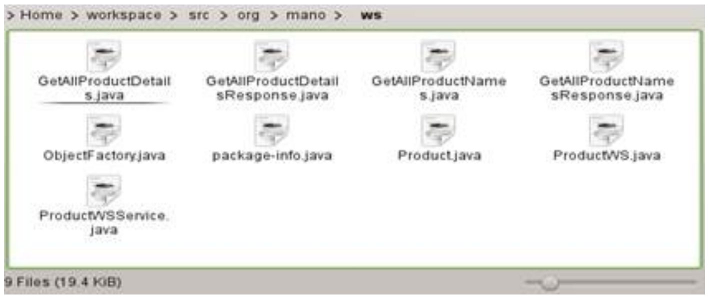
Fig 4: Project structure of SOAP web service Consumer

1) **Web Service client files:** Web Service client files can be auto generated with the help of wsimport as follows.

- Create a directory named **src**. We shall import the Web Service client files into this directory.
    - Wsimport can be found in jdk.../bin directory. Use wsimport to import the necessary wWeb Service client files. wsimport generates the necessary java classes from the WSDL link. Where to find the WSDL link? Check out the section above - **Testing Web Service without client.** Type the following in the command line interface with the WSDL. In my case:

wsimport -keep -s src http://localhost:8080/.../ProductWSService?wsdl

- The following java files will be created in **src/org/mano/ws** directory.


> Home > workspace > src > org > mano > ws
GetAllProductDetails.java
GetAllProductDetailsResponse.java
GetAllProductNames.java
GetAllProductNamesResponse.java
ObjectFactory.java
package-info.java
Product.java
ProductWS.java
ProductWSService.java
9 Files (19.4 KiB)
Fig 5: Files generated by wsimport

- Observe the directory hierarchy (in my case org/mano/ws). Create a package (according to the same, org.mano.ws in the client project and copy-paste the java files. Now you are ready to use Web Services in your client project.

2) **Main.java:** This is the client interface where products are displayed in a table format. The class uses javax.swing.JTable to display the list. The product list is fetched with the help of following code.

```
ProductWSService webService = new ProductWSService();
ProductWS productWS = webService.getProductWSPort();
```

```
products = productWS.getAllProductDetails();
```

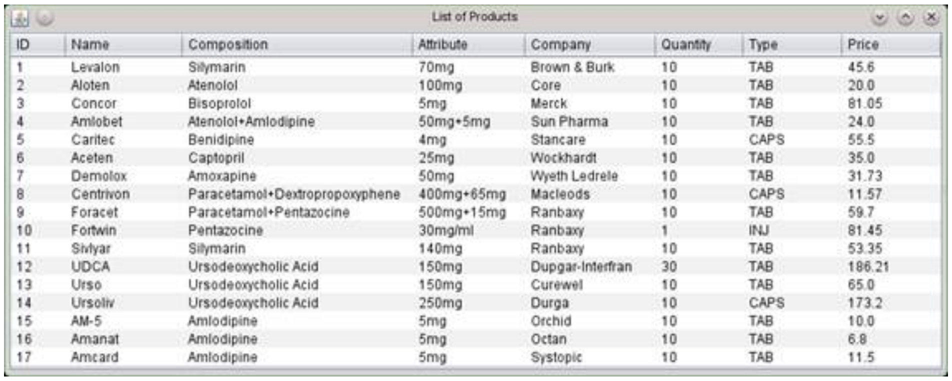
Main.java

```
package org.mano.app;

//... import statements

public class Main extends JFrame {

       private static List<Product> products = new ArrayList<>();
       private JTable table;

       public Main() {
             super("List of Products");
             setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
             setSize(400, 400);
             setLayout(new BorderLayout(5, 5));
             ProductWSService webService = new ProductWSService();
             ProductWS productWS = webService.getProductWSPort();
             products = productWS.getAllProductDetails();
             table = new JTable();
             populateTable();
             add(new JScrollPane(table), BorderLayout.CENTER);
             setVisible(true);
       }

       private void populateTable() {
             String columnNames[] = { "ID", "Name", "Composition", "Attribute",
                          "Company", "Quantity", "Type", "Price" };
             DefaultTableModel dataModel = new DefaultTableModel(columnNames,
                          products.size());
             int row = 0;
             for (Product p : products) {
                    dataModel.setValueAt(p.getId(), row, 0);
                    dataModel.setValueAt(p.getName(), row, 1);
                    dataModel.setValueAt(p.getComposition(), row, 2);
                    dataModel.setValueAt(p.getAttribute(), row, 3);
                    dataModel.setValueAt(p.getCompany(), row, 4);
                    dataModel.setValueAt(p.getQuantity(), row, 5);
                    dataModel.setValueAt(p.getType(), row, 6);
                    dataModel.setValueAt(p.getPrice(), row, 7);
                    row++;
             }
             table.setModel(dataModel);
       }

       public static void main(String[] args) throws Exception{
             new Main();
       }

}
```

Note: Make sure that Glassfish with the Web Service producer is still running; otherwise the client/web service consumer application will not work. Also note that I have already inserted some dummy data in to the database. You may use SQL to insert some dummy product details as follows:

INSERT INTO product_tbl (ID, NAME, COMPOSITION, TYPE, QUANTITY, COMPANY, ATTRIBUTE, PRICE) VALUES (1, 'Levalon', 'Silymarin', 'TAB', 10, 'Brown & Burk', '70mg', 45.60);

Now, run the client application from the Eclipse/ Netbeans/ command line; you'll get the following output.



List of Products

ID	Name	Composition	Attribute	Company	Quantity	Type	Price
1	Levalon	Silymarin	70mg	Brown & Burk	10	TAB	45.6
2	Aloten	Atenolol	100mg	Core	10	TAB	20.0
3	Concor	Bisoprolol	5mg	Merck	10	TAB	81.05
4	Amlobet	Atenolol+Amlodipine	50mg+5mg	Sun Pharma	10	TAB	24.0
5	Caritec	Benidipine	4mg	Stancare	10	CAPS	55.5
6	Aceten	Captopril	25mg	Wockhardt	10	TAB	35.0
7	Demolox	Amoxapine	50mg	Wyeth Ledrele	10	TAB	31.73
8	Centrivon	Paracetamol+Dextropropoxyphene	400mg+65mg	Macleods	10	CAPS	11.57
9	Foracet	Paracetamol+Pentazocine	500mg+15mg	Ranbaxy	10	TAB	59.7
10	Fortwin	Pentazocine	30mg/ml	Ranbaxy	1	INJ	81.45
11	Sivlyar	Silymarin	140mg	Ranbaxy	10	TAB	53.35
12	UDCA	Ursodeoxycholic Acid	150mg	Dupgar-Interfran	30	TAB	186.21
13	Urso	Ursodeoxycholic Acid	150mg	Curewel	10	TAB	65.0
14	Ursoliv	Ursodeoxycholic Acid	250mg	Durga	10	CAPS	173.2
15	AM-5	Amlodipine	5mg	Orchid	10	TAB	10.0
16	Amanat	Amlodipine	5mg	Octan	10	TAB	6.8
17	Amcard	Amlodipine	5mg	Systopic	10	TAB	11.5

Fig 6: Client application output: Product details displayed in JTable.

Conclusion

Creating a SOAP web Service producer/consumer is pretty simple. However, looks can be deceiving, there are lots of things going on behind the scene obviously not as simple as they seem. Thanks to Java and its tools, programmers can ignore the intricacies most of the time and enjoy creating SOAP Web Services.