



Enabling Grids for E-scienceE

PPARC Summer School,
May 2005

Web Service Tools

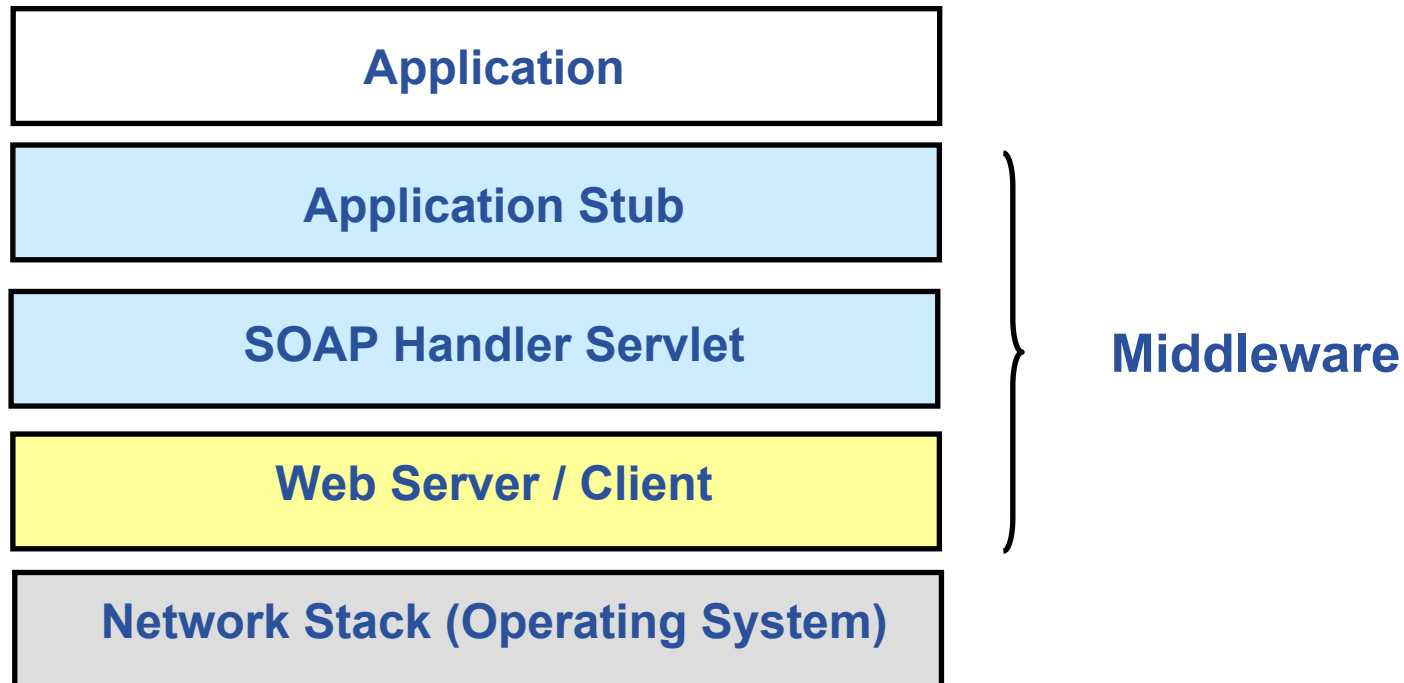
*Oliver Malham,
National e-Science Centre, Edinburgh*

*Based on material by David Fergusson,
Richard Hopkins*

www.eu-egee.org



- **Goals**
 - To understand the context and basic workings of current web services toolkits, focusing on Sun's Java Web Services Developers Pack (JWSDP)
- **Structure**
 - *Toolkits in General*
 - JWSDP (JAX-RPC)
 - Some Details



- **Purpose**

Provides the HTTP interface between the Web Services handler and the network infrastructure.

- **Web Service Compatible Server Platforms**

- Apache 1.3 & 2.0
(<http://www.apache.org>)
- Tomcat
(<http://jakarta.apache.org/tomcat/>)
- IBM WebSphere
(<http://www-306.ibm.com/software/webservers/appserv/express/>)
- Sun Java 2 Enterprise Edition (J2EE)
(<http://java.sun.com/j2ee/>)
- Microsoft Windows Server 2003
(<http://www.microsoft.com/windowsserver2003/>)
- Oracle Application Server 10g
(<http://www.oracle.com/appserver/>)

- **Purpose**

Provides the SOAP message handling, XML encoding and decoding, and method invocation functions.

- **Commonly Available Toolkits**

- C++

- Axis C++ (<http://ws.apache.org/axis/cpp/>)
- Microsoft .Net (<http://www.microsoft.com/net/>) (also handles C#, ASP.Net, VB.Net, etc)

- Perl

- SOAP::Lite (<http://www.perl.com/pub/a/2001/01/soap.html>)

- Java

- Axis (<http://ws.apache.org/axis/java/>)
- Sun Java Web Services Developer Pack (JWS DP) (<http://java.sun.com/webservices/jwsdp/index.jsp>)

- **Purpose**

A number of complete, all-in-one web services containers, or application servers, are available, which combine all elements of the software stack.

- **Examples**

- gSOAP (C++)

(<http://www.cs.fsu.edu/~engelen/soap.html>)

- IBM WebSphere (Java)

(<http://www-306.ibm.com/software/webservers/appserv/express/>)

- Sun Java 2 Enterprise Edition (J2EE) (Java)

(<http://java.sun.com/j2ee/>)

- **Goals**
 - To understand the context and basic workings of current web services toolkits, focusing on Sun's Java Web Services Developers Pack (JWSDP)
- **Structure**
 - Toolkits in General
 - ***JWSDP (JAX-RPC)***
 - Some Details

JWSDP Packages

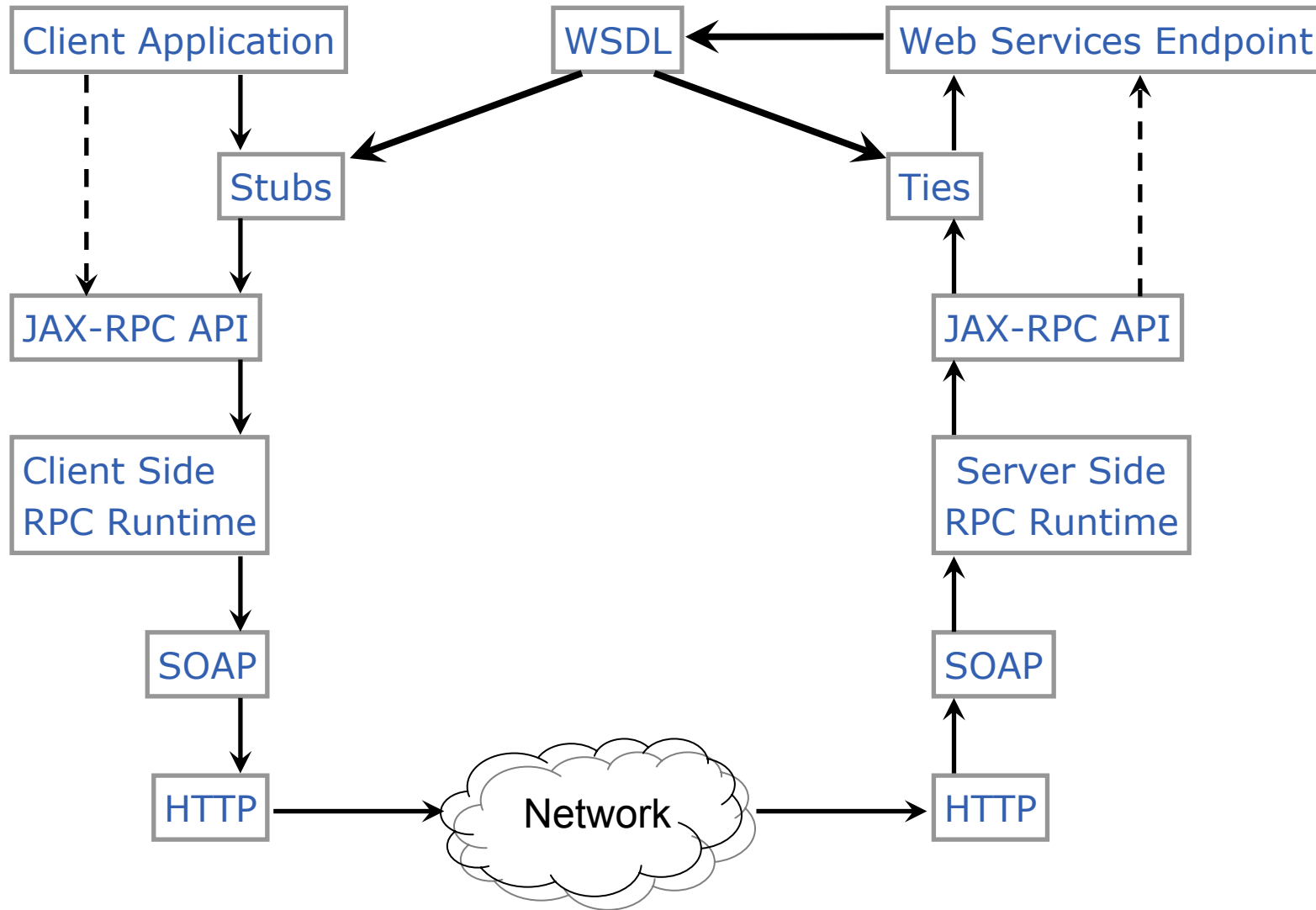
- **saaj**
 - soap with attachments API for java
- **jaxp**
 - jax parsing (XML)
- **jaxb**
 - XML → Java “bindings” = de-serialisation
- **jaxr**
 - Jax for registries
- **jax-rpc**
 - Jax remote procedure call

The jax-rpc provides packages which:

- Given WSDL or Java Interface definitions generate 'stub' classes for web service providers or consumers.
- Handle Java ↔ XML serialisations / de-serialisation
- Handle the generation of SOAP messages

API Packages

- `javax.xml.rpc` Core classes for the client side programming mode
- `javax.xml.rpc.encoding` Java objects <-> XML SOAP messages
- `javax.xml.rpc.handler` processing XML messages
- `javax.xml.rpc.handler.soap`
- `javax.xml.rpc.holders` support the use of holder classes
- `javax.xml.rpc.server` minimal API for web service implementation
- `javax.xml.rpc.soap` specific SOAP binding



- **JAX-RPC allows two modes of operation**
- **Synchronous – two-way RPC**
 - This involves blocking the client until it receives a response
 - Is similar to a traditional java method call
 - Even if no actual return value – Public void request (...)
 - Have wait for a success/exception response
- **One-way RPC - Asynchronous**
 - No client blocking
 - Service performs a operation without replying.
 - Not analogous to traditional method calls
 - Cannot throw an exception

A java web service end point interface must obey the following rules:

- The interface must extend `java.rmi.remote`
- Service endpoint interfaces may be extensions of other interfaces
- Interface methods must declare that it throws `java.rmi.RemoteException`
- Service dependent exceptions can be thrown if they are checked exceptions derived from `java.lang.Exception`

Types That can be in the interface

- Java primitives (eg. `bool`, `int`, `float`, etc)
- Primitive wrappers (`Boolean`, `Integer`, `Float`, etc)
- Standard java classes

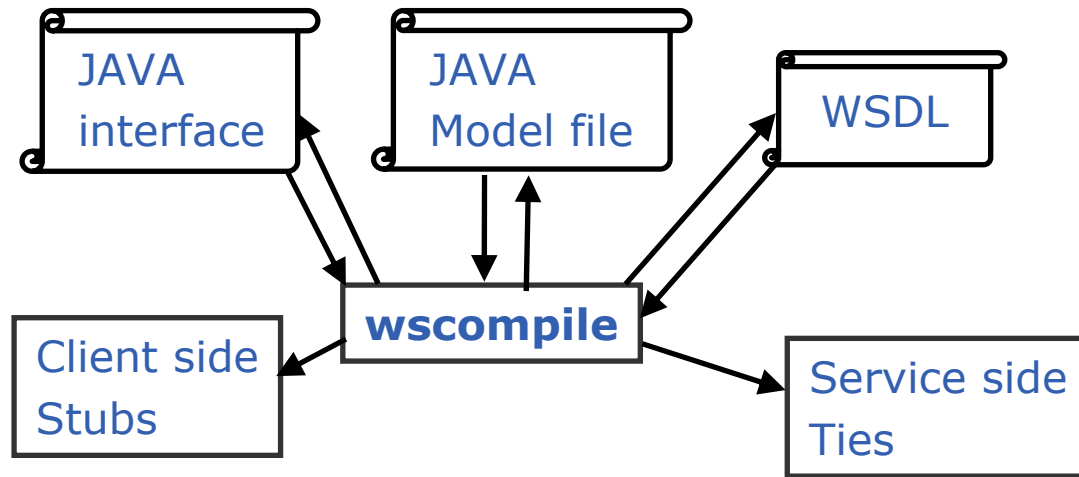
```
java.lang.String,    java.util.Calendar,  
java.util.Date,     java.math.BigDecimal,  
java.math.BigInteger
```

- “Value types”
 - Class has a public no-argument constructor
 - May be extended from any other class, may have static and instance methods, may implement any interface (except `java.rmi.Remote` and any derived)
 - May have static fields, instance fields that are public, protected, package private or private but these must be supported types.
- Arrays (where all elements are supported types)

Object by reference is not supported



- Java web services (also C based ones) allow a developer to treat service classes as if they are local - i.e. stubs are created
- All web services messages are XML (SOAP)
- This means that objects sent across web services must be translated to XML and back – (de-)serialisation
- What is serialised is the “accessible state”; either
 - directly accessible fields
 - Fields with mutator/accessor methods
- The values returned by service methods are in fact local classes created by JAX-RPC from the XML serialisation
 - Classes seen by either side may not be identical
 - So avoid comparisons using `==` ; `equals()` should be used instead
- If you want to pass an un-supported java class you have to create your own serialiser / de-serialiser to translate to and from XML.
- This not a trivial task as there is no JAX-RPC framework.



“Model” –

Partially compiled interface

Usage Modes –

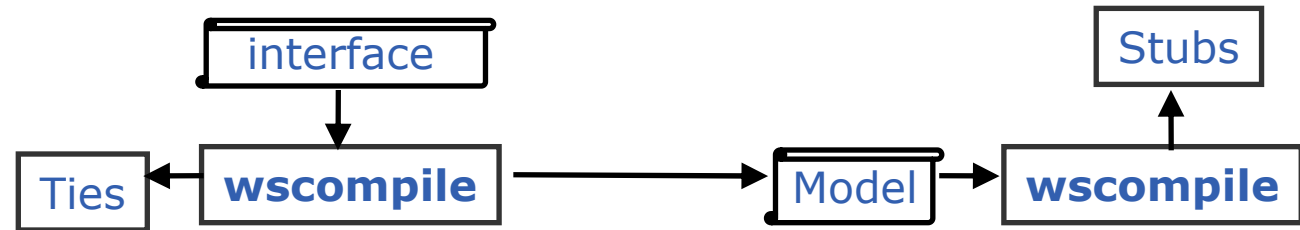
Interface → Model, WSDL

WSDL → Model, Interface

Model → Interface, Interface

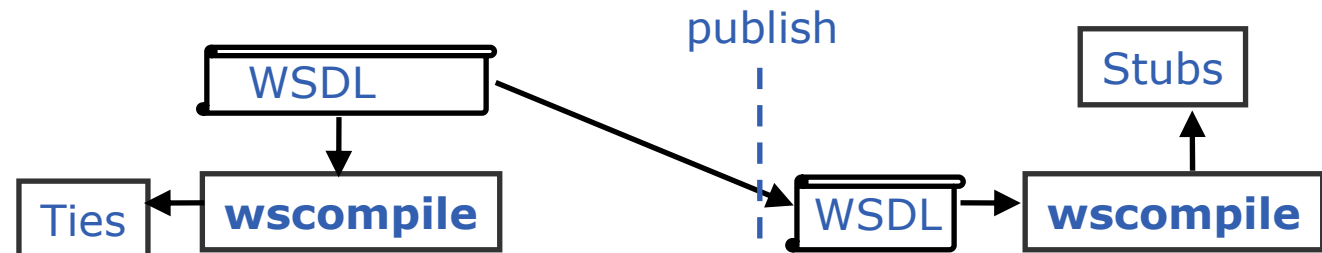
Local

Client and Server
same organisation



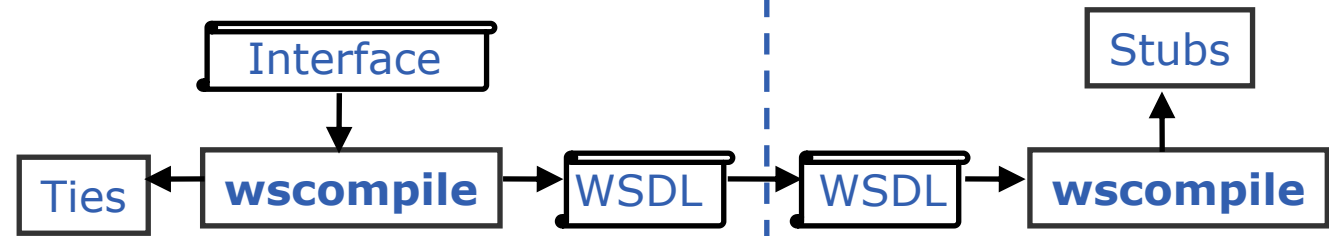
Remote

Client and Server
different organisation



Remote

Starting from Java
Rather than WSDL



- **Goals**

- To understand the context and basic workings of current web services toolkits, focusing on Sun's Java Web Services Developers Pack (JWSDP)

- **Structure**

- Toolkits in General
- JWSDP (JAX-RPC)
- ***Some Details***

- **WSDL can be downloaded from a UDDI registry**
- **If the service uses JAXRPCServlet you can attach ?WSDL (or ?model) to the URL request to get the WSDL (or model file).**
 - E.g. `http://localhost:8080/Service/ServiceName?WSDL`

wscompile

`-gen:client -d outputdir -classpath dir1 -keep -s dir2 config.xml`

Client-side use

artefact=stubs

artefact=ties
server-side use

Where to put
generated
artefacts

To override
standard
classpath

To retain
Java source
For
generated
Output; and
where to
put it

Definition
of the
Service –
Model
or WSDL
or
Interface

wscompile

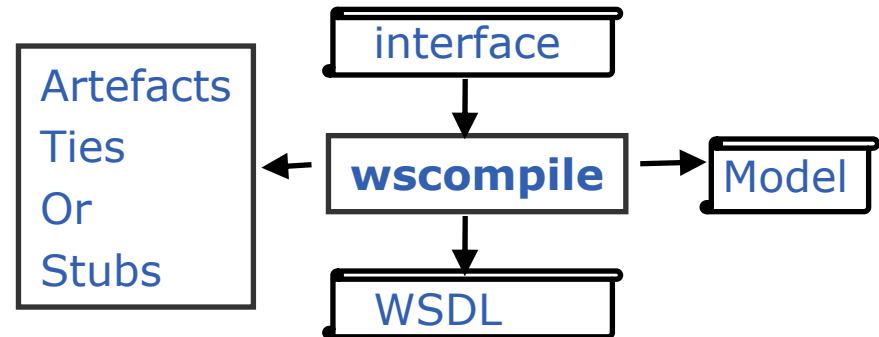
`-gen:server -d outputdir -classpath dir1 -keep -s dir2
- model mfile.z`

config.xml

To generate a model file and where to put it – for use by wsdeploy

config.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<configuration
  xmlns="http://java.sun.com/.../config">
  <service name="....."
    targetNamespace=" ...// .../.../wsdl"
    typeNamespace=" ...// .../.../types"
    packageName="...">
    <interface name="..."
      servantName="..."/></>
  </configuration>
```



service name = name of service for WSDL definition

targetNamespace = namespace of WSDL for names associated with the service e.g. port type

typeNamespace = namespace of WSDL for data types

packageName = name of java package

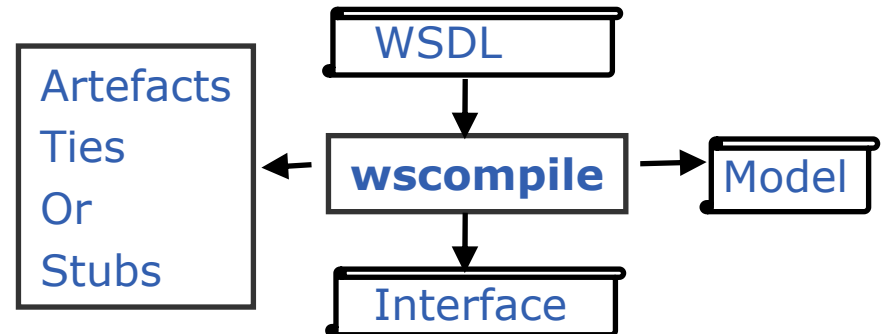
interface name = name of the java interface

servantName = the name of the class that implements the interface

config.xml

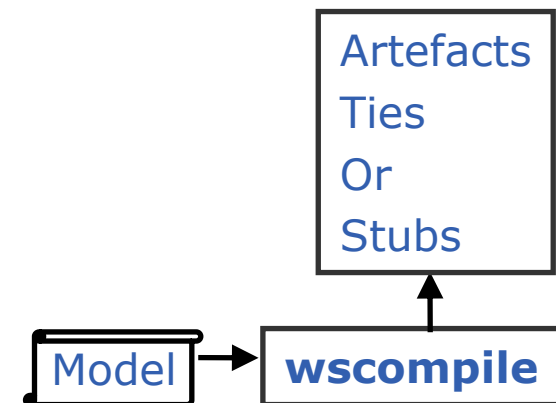
```
<?xml version="1.0" encoding="UTF-8" ?>
<configuration
  xmlns="http://java.sun.com/.../config">
  <wSDL
    location-"..//.../serviceDef .wsdl"
    packageName=" ...."/>
</configuration>
```

Location = URL for the WSDL
 packageName = name of java package to be generated



```
<?xml version="1.0" encoding="UTF-8" ?>
<configuration
  xmlns="http://java.sun.com/.../config">
  <model location-"myModel.z"/>
</configuration>
```

Location = file name of previously generated model



Some of the client side generated files:

Service	Service.java
	Service_Impl.java
	Service_SerializerRegistry.java
Exception	ServiceException_SOAPSerializer.java
	ServiceException_SOAPBuilder.java
Value type	Info_SOAPSerializer.java
	Info_SOAPBuilder.java
Interface	Interface_Stub.java
	method.java

- The `Service.java` file corresponds to the definition of the interface for the web service,

```
package servicePackage;
import javax.xml.rpc.*;
Public interface Service extends javax.xml.rpc.Service
{ public servicePackage getServicePort(); }
```

- An object implementing the interface is like a “service factory” –
- `getServicePort` returns an instance of (the stub for) the actual service
- The required service factory is `Service_Impl`

```
Service_Impl service = new Service_Impl ();
value* name = (value)service.getServicePort ();
```

With this reference you can call the methods of the service.

- **Create a WAR file**
 - Java class file for service endpoint interface
 - Java class files for service implementation and resources
 - `web.xml` file containing deployment information
 - Class files for JAX-RPC tie classes

- **JAX-RPC tie classes are implementation specific.**

<code>WEB-INF/web.xml</code>	Web application deployment descriptor
<code>WEB-INF/jaxrpc-ri.xml</code>	JWSDP-specific deployment information
<code>WEB-INF/model</code>	Model file generated by <code>wscouple</code>

```
<?xml version="1.0" encoding="UTF-8" ?>
```

```
<!DOCTYPE web-app
```

```
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application  
  2.3//EN"
```

```
  "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
```

```
<web-app>
```

```
  <display-name>Service Name</display-name>
```

```
  <description>A web service application</description>
```

```
</web-app>
```

```
wsdeploy -o targetFileName portableWarFileName
```

The process is informed by the content of the `jaxrpc-ri.xml` file.

The archive contains:

- class files and resources

- compiled class files for the ties

- compiled class files for serializers

- WSDL (in WEB-INF directory)

- model file for the service (in WEB-INF)

- modified `web.xml` file

- `jaxrpc-ri-runtime.xml` (based on `jaxrpc-ri.xml`)

