Enabling Grids for E-sciencE

ISSGC 05
Web Service Tools

NeSC Training Team

www.eu-egee.org

Overview

- **Goals**
  - To Understand the context and basic workings of the JAVA Web Services Development Pack
- **Structure**
  - General
  - JWSDP (JAX-RPC)
  - Some Details

PERL / C-based

PERL

- **SOAP::LITE - Collection of Perl modules which provides a simple and lightweight interface to SOAP on both client and server side.**

C-Based

- **gSOAP**
  - C and C++ toolkit which provides C/C++ - XML bindings for web services development
  - Comments from developers suggest that this implementation is fragile and can be buggy
- **.NET**
  - Microsoft web services implementation based on C# super-set of C.
  - Comments form developers – easy entry but lacks flexibility in more complex situations

XML Parsing

- **Xerces (originally Java, also C++ now)**
  - Used in JWSDP modules, Axis
- **DOM (Document Object Model)**
  - Creates representation of document structure in memory
- **SAX (Simple API for XML)**
  - Simpler but less powerful parsing model

JAVA

- Build Tool – **ANT**
- Containers
  - add functionality to web servers
  - **Tomcat** originally designed to add servlets to web servers – became used to support web services
  - **Axis** new development to specifically support web services
    - Axis also includes a web services development environment
- Development environments
  - Java 2 Enterprise Edition (J2EE)
  - Java Beans
  - Java Web Services Development Package (JWSDP)

JWSDP

- **Goals**
  - To Understand the context and basic workings of the JAVA Web Services Development Pack
- **Structure**
  - General
  - JWSDP (JAX-RPC)
  - Some Details

JWSDP packages

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

What does JAX-RPC do

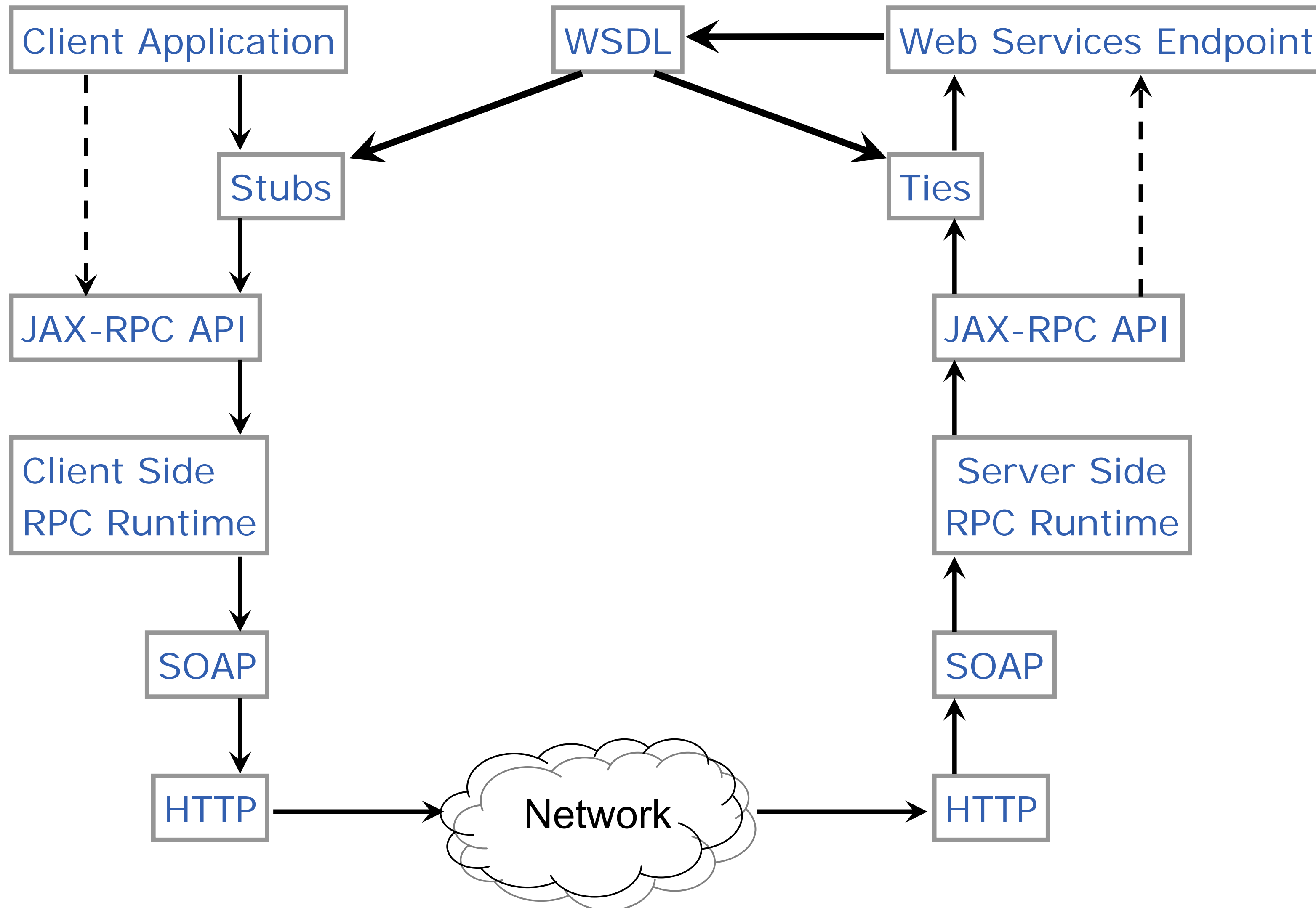
The jax-rpc provides packages which:

- 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 • javax.xml.rpc.handler.soap	processing XML messages
• javax.xml.rpc.holders	support the use of holder lasses
• javax.xml.rpc.server	minimal API for web service implementation
• Javax.xml.rpc.soap	specific SOAP binding

JAX-RPC Architecture



Client operation modes

- **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

Interface method definitions

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

Types That can be in the interface

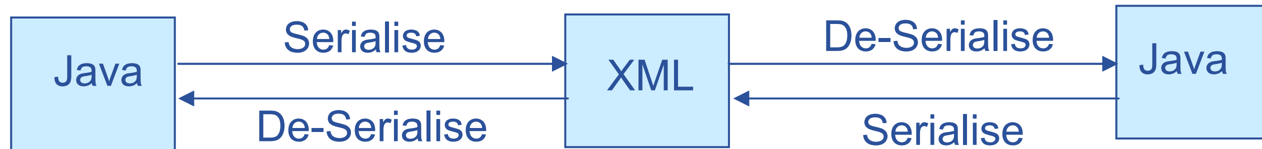
- **Java primitives (eg. `bool, int, float`, etc)**
- **Primitive wrappers (`Boolean, Integer, Float`, etc)**
- **Standard java classes**

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

- **Holder classes**
- **"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)**

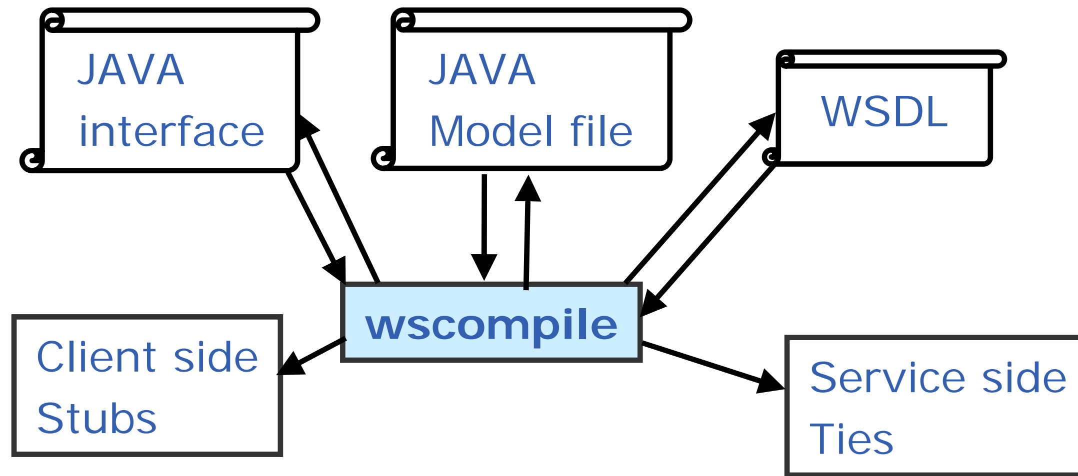
Object by reference is not supported

(de-) serialisation



- 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.

Wscompile



"Model" –

Partially compiled interface

Usage Modes –

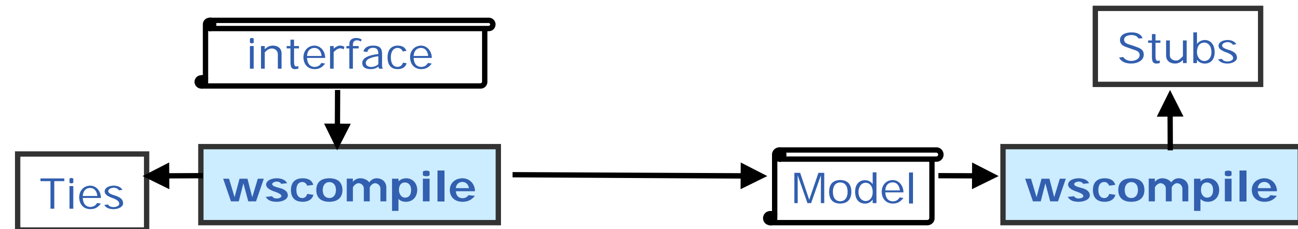
Interface → Model, WSDL

WSDL → Model, Interface

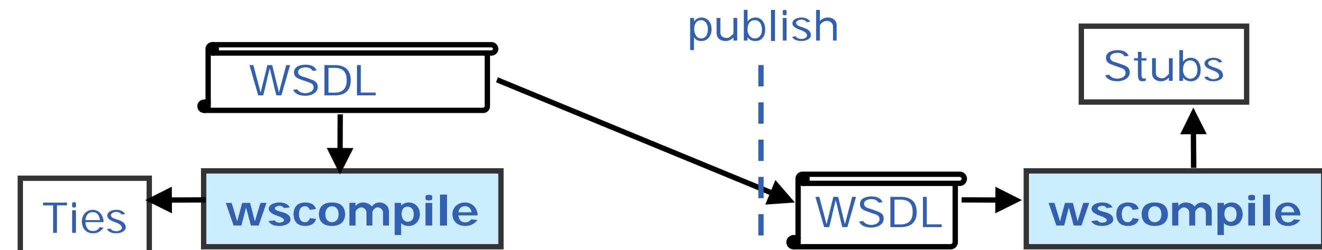
Model → Interface, Interface

wscompile – usage patterns

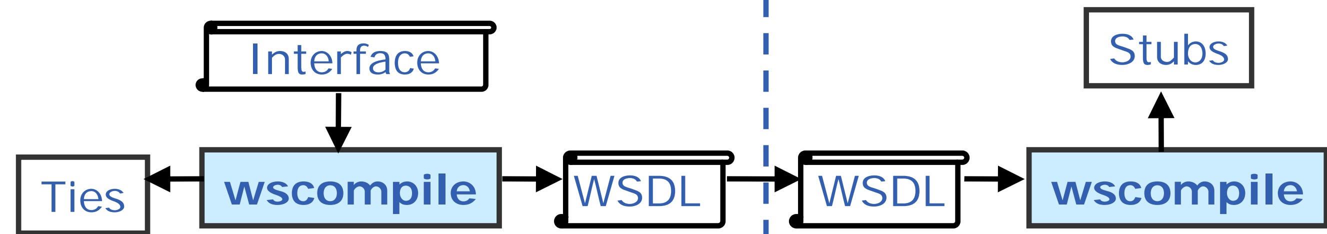
Local
Client and Server
same organisation



Remote
Client and Server
different organisation



Remote
Starting from Java
Rather than WSDL



Some Details

- **Goals**
  - To Understand the context and basic workings of the JAVA Web Services Development Pack
- **Structure**
  - General
  - JWSDP (JAX-RPC)
  - Some Details

Obtaining the WSDL

- **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

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

Client-side use

artefact=stubs

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

artefact=ties

server-side use

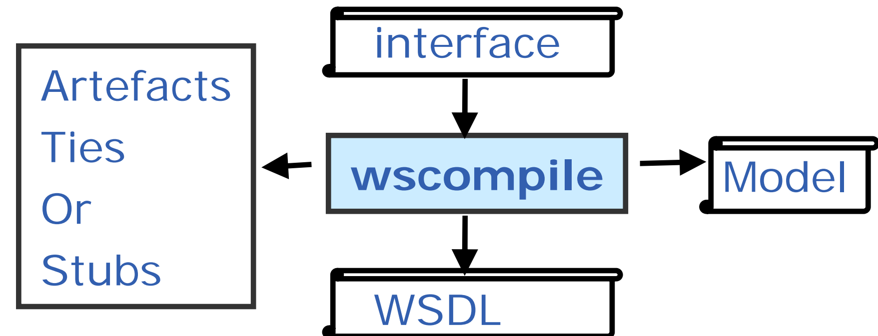
```
wscompile
      –gen:server –d outputdir –classpath dir1 –keep –s dir2
      - model mfile.z                                  config.xml
```

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

Configuration File – from interface

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>
```

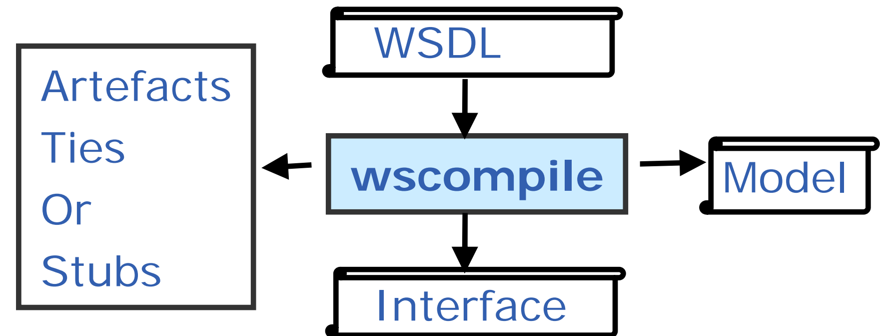


service name = name of service for WSDL definition
targetNamespace = namespace of WSDL for names associated with the service e.g. port type
typeNamespace = namespace of WSDL for data types
packageName = name of java package
interface name = name of the java interface
servantName = the name of the class that implements the interface

Configuration File – from WSDL / Model

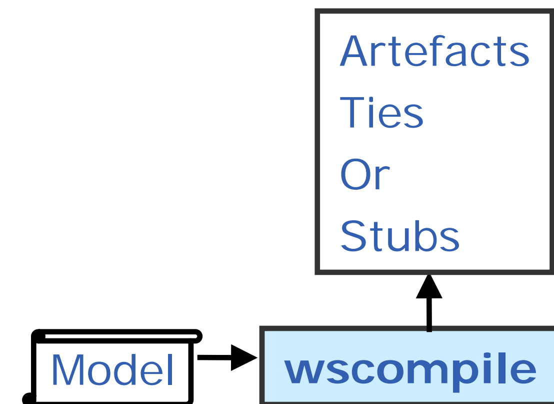
config.xml

```
<?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

Generated files

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

Accessing the Service

- 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.aml.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
  – (Unfortunately this name is only recommended)

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

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

Deploying to a web container

- **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.**

Additional WAR files required for JWSDP

`WEB-INF/web.xml`	**Web application deployment descriptor**
`WEB-INF/jaxrpc-ri.xml`	**JWSDP-specific deployment information**
`WEB-INF/model`	**Model file generated by** `wscompile`

web.xml file

```
<?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>
```

Creating a deployable WAR file

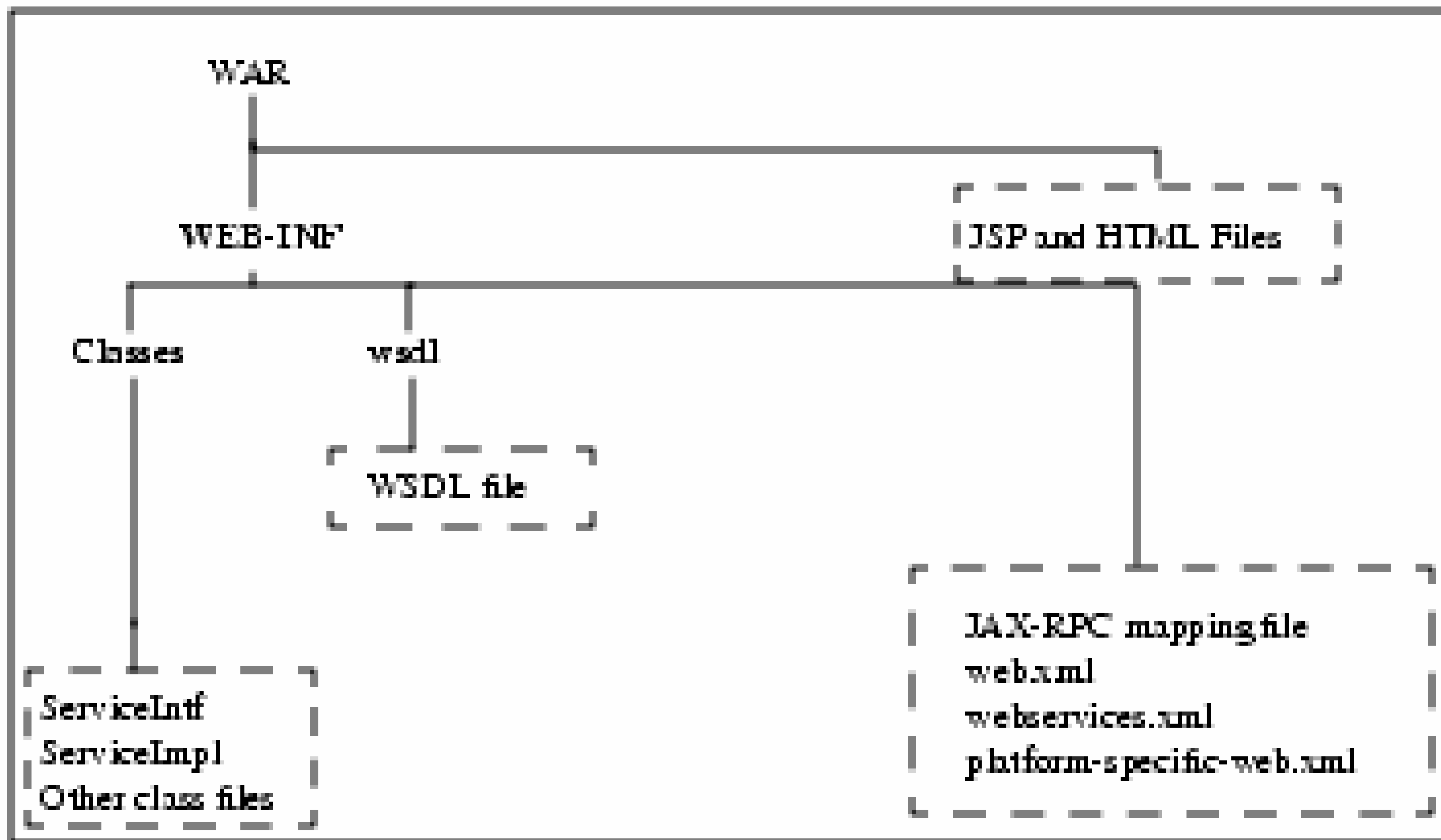
```
wsdeploy -o targetFileName portableWarFileName
```

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`)

Package Structure for JAX-RPC Service Endpoint



Files required in the JAR

File type	Filename
Service end point interface	*Classpath.service.name*
	Classpath.service.Info
	Classpath.service.Exception
Service interface	*Classpath.service.Service*
Application implementation	*Classpath.client.ServiceAppClient*
WSDL file	*Service.wsdl*
Deployment descriptors	*META-INF/application-client.xml*
	META-INF/mapping.xml or META-INF/model
	META-INF/webservicesclient.xml
Manifest file	*META-INF/MANIFEST.MF*