ApacheCon EU 2005

Open Standards / Open Source Java Web Services with Apache Geronimo

By
Tom McQueeney

What this tutorial covers: Part I lecture (1 hour)

- Web services background
- Introduce J2EE 1.4 web services
- Writing J2EE 1.4 web services ("endpoints")
- Simple example of a J2EE web service
- Deploying web services to a J2EE 1.4 container
- Techniques for writing web service client applications
- Tips for writing interoperable web services
- Future of J2EE web services (JSR 181, JSR 224)
- With J2EE 5 on the horizon, why learn J2EE 1.4?

What this tutorial covers: Part II lab (2 hours)

- Writing a more realistic Java web service
- WSDL
- Web service deployment descriptors
- JAX-RPC mapping files
- Using XDoclet to generate web service artifacts
- Using Spring when coding web services
- Sun's Java Web Services Developer Pack
- Where auto-generation falls short
- How Geronimo implements web services
- Advanced topics as time permits: stateful web services, handlers, holders, security

Web services background information (1 of 2)

- *Web services* started as a concept circa 1997 defining an exchange of XML service requests/responses (often procedure/method calls) between two systems over the web (http)
- Motivation and purpose for WS much like CORBA – allow different system written in different languages to call each other's services remotely over the network – except WS standards are being widely adopted
- Advantages of web services:
  - Language neutral
  - Platform neutral
  - Better firewall traversal (than CORBA, DCOM, RMI)
  - Flexibility: Exchanged messages described as XML

Web services background information (2 of 2)

- SOAP 1.1 and WSDL 1.1, which became W3C "standards" in 2001, help tighten the definition by defining message formats and communication protocols. *Web service* today often means an implementation of a service interface defined in a WSDL document
- WS-I "profiles" narrows definition even further to promote interoperability
- OASIS adding higher-level specifications for WS, like security (SAML), business processes (ebXML)

Introduction to J2EE 1.4 web services (1 of 2)

- Adding web services to J2EE delayed 1.4 spec for several months
- Goals of J2EE 1.4 web services:
  - Interoperability: Non-Java clients can call J2EE web services and J2EE components can call non-Java web services
  - Portable code: Can develop a web-service and deploy it to any J2EE server
  - Simplify: Hide details of SOAP, WSDL, HTTP, etc., from the developer
- Specs add a new J2EE component type (JSE) and augment EJB 2.1 specification to expose SLSBs as a web service endpoint
- Defines client programming model to interact with web services. Web service clients can live inside or outside the J2EE container
- Defined using SOAP 1.1 and WSDL 1.1 (and UDDI)

Introduction to J2EE 1.4 web services (2 of 2)

- Interoperability: Complies with WS-I Basic Profile 1.0
- Requires support for JAX-RPC 1.1, which defines:
  - How WSDL maps to Java interfaces, and vice versa
  - How incoming SOAP (XML) messages are converted (unmarshalled) to Java data types, and vice versa (does not use JAXB)
  - How WSDL "out" and "inout" parameters are handled

Writing J2EE 1.4 web services ("endpoints")

- You – or deployment tool – must supply a WSDL document describing your web service. WSDL must be deployed as part of component
- Two J2EE components can implement a web service:
  - JSE (*JAX-RPC Service Endpoint*): *Can* look like a servlet. Acts like a servlet. Runs in web container like a servlet. Isn't a servlet (but is called by one)
  - SLSB: Exposed as web service via deployment descriptors
- Both are coded as remote objects, and implement the `java.rmi.Remote` interface
- Which to use?
  - JSE is easier because it's (mostly) just a POJO

Writing J2EE 1.4 web services ("endpoints")

- Might prefer using a stateless session EJB if:
  - You already have a SLSB you want to expose as a web service
  - You need the services of EJB, like declarative transaction management or role-based security, in your implementation
  - But: The SOAP *client* calling the EJB web service cannot participate in the transaction (at least the spec does not define it)
  - Your EJB method transaction attributes probably will be Requires or RequiresNew. Mandatory is illegal in EJB endpoints. Can you figure out why?

    ➔Because SOAP client can't propagate its transaction
  - If you use Supports, Never or NotSupported, ask yourself, "Why am I using EJB?"

JSE: JAX-RPC Service Endpoints (1 of 3)

- JSEs share similarities to servlets:
  - Instances run in the web container
  - Container invokes JSE methods in response to incoming requests
  - Instances follow servlet-like lifecycle
  - Must have a public, no-arg constructor
  - Uses web.xml as base deployment descriptor (hack for simplicity)
  - Can be deployed with servlets (and other JSEs) in a WAR file
  - May be called from multiple threads at one time. Practice safe threading – Don't save client state in instance vars
  - Can use servlet filters in web.xml to pre- and post-process raw HTTP request/response
- If you want, you can even code your JSE similar to a servlet
  - implement **`javax.xml.rpc.server.ServiceLifecycle`** interface:
  - Provides init() and destroy() lifecycle methods
  - init() is passed a JSE context object, which provides access to servlet container services (ServletContext, HTTPSession, security Principal)

JSE: JAX-RPC Service Endpoints (2 of 3)

- Un-servlet-like features and deployment requirements:
  - JSE can have **SOAP** "handlers" to process the SOAP request/response. Handlers are like servlet filters, except they see SOAP messages
  - Adds required webservices.xml deployment descriptor in WEB-INF to describe web service
  - Adds required "jaxrpc-mapping.xml" deployment descriptor under WEB-INF
  - Requires a WSDL document under WEB-INF that describes the web service
- JSE must be coded as a Java remote object. i.e. YOU MUST...
  - Write/supply a remote interface the JSE implements, called a *service endpoint interface*. (SEI can be auto-generated via XDoclet or J2EE vendor tools)
  - SEI must extend marker interface `java.rmi.Remote`
  - SEI methods must be declared to throw `java.rmi.RemoteException`

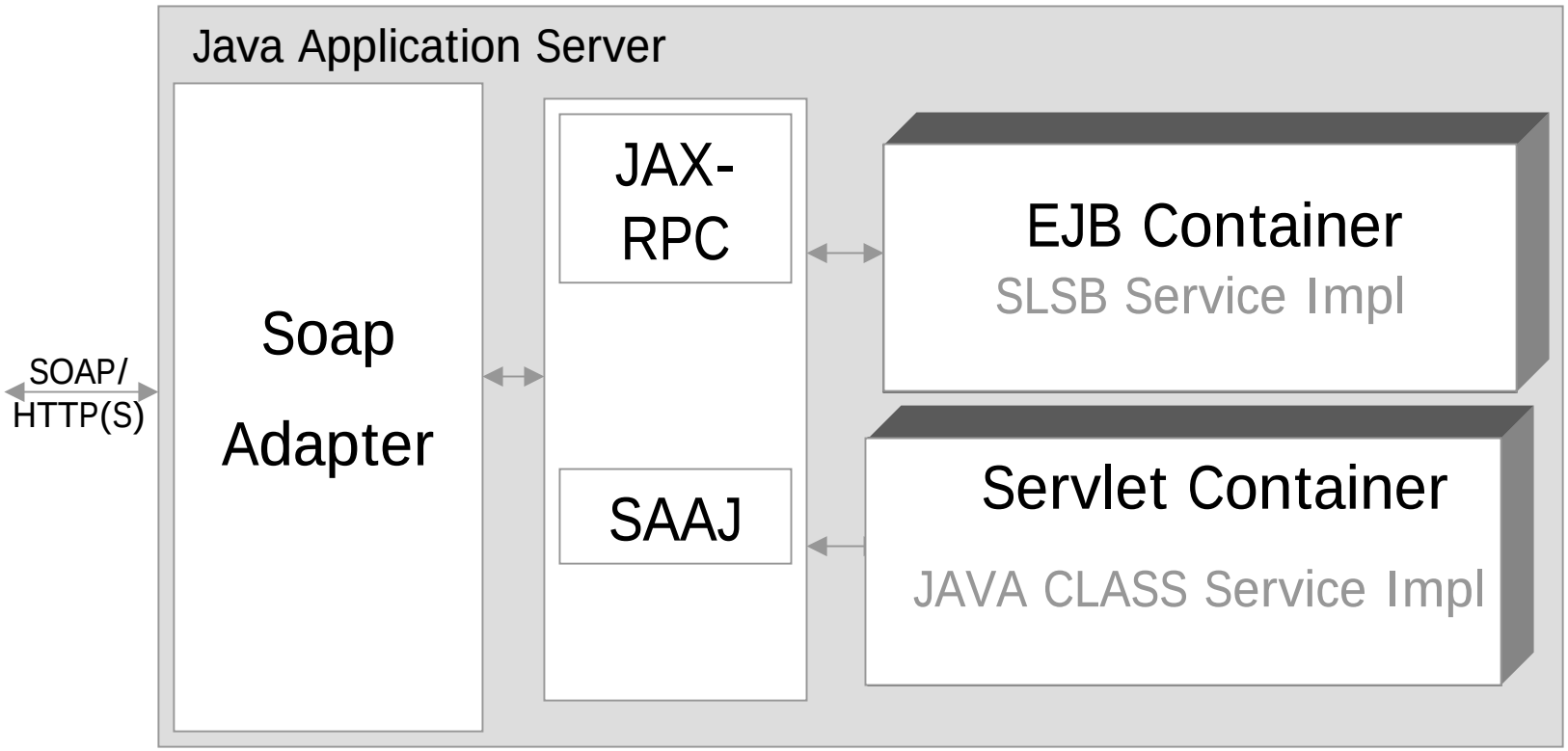
JSE: JAX-RPC Service Endpoints (3 of 3)

- BUT! JSE itself can be coded to look like any other Java class
- It must implement or extend **no** J2EE component
- Web container treats your JSE as a remote object, but *you* don't have to
- I.e. your JSE methods don't *have* to throw `RemoteException`

EJB Service Endpoints

- J2EE 1.4 spec changes `ejb-jar.xml` deployment descriptor to allow SLSB to be defined as a web service
- Like JSE, adds requirements to SLSB deployment:
  - Requires `webservices.xml` deployment descriptor to describe web service endpoint
  - Requires "`jaxrpc-mapping.xml`" deployment descriptor
  - Requires WSDL file to be deployed with EJB
  - All above goes in/under the META-INF directory of your EJB jar file

How J2EE servers implement endpoints

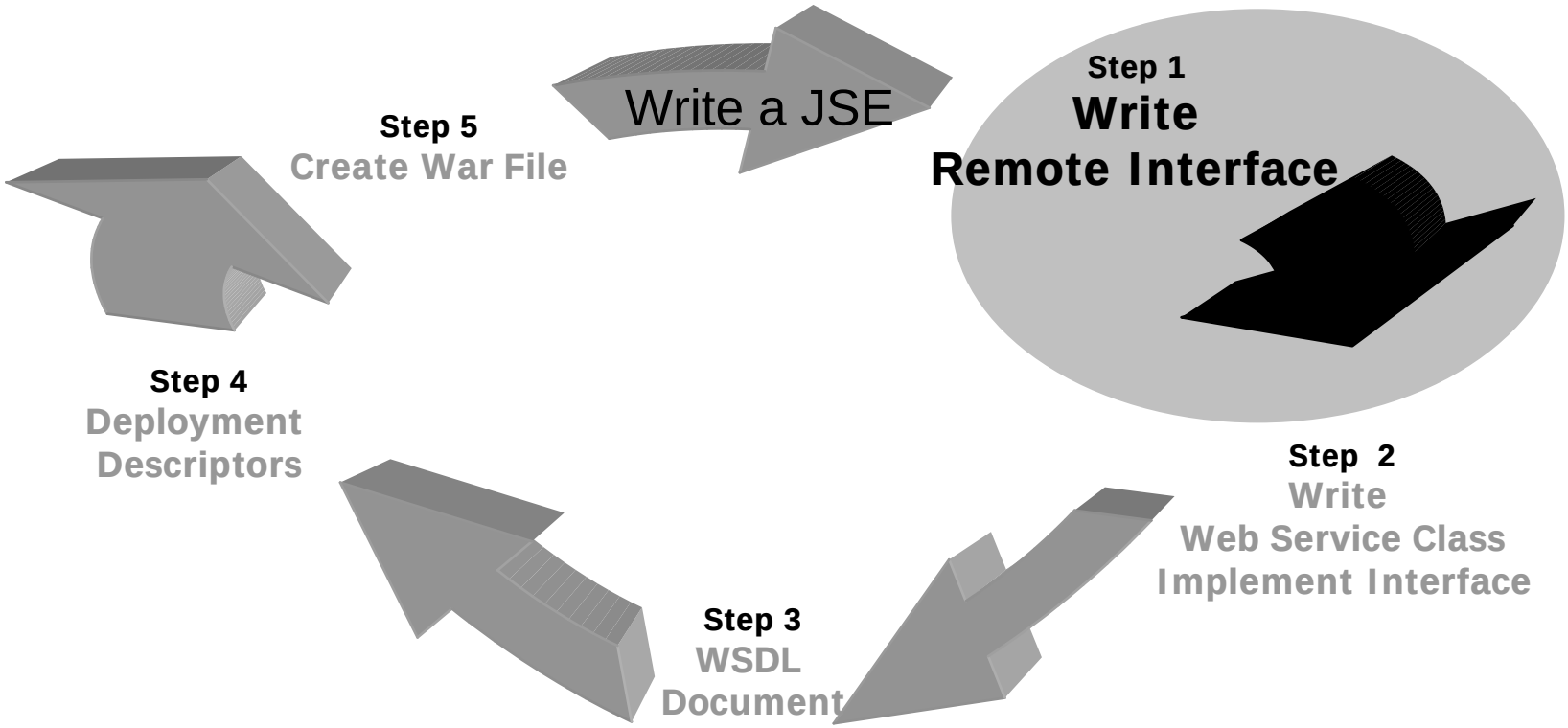


Code example: Steps to writing a JSE

1. Write remote interface (the service endpoint interface)
2. Write class that implements the interface
3. Write (generate) WSDL document that describes the web service (vendor tool)
4. Write (or generate) deployment descriptors:
    1. web.xml defines the services and the URL mappings to them
    2. webservices.xml ties J2EE components to "port" definitions in the WSDL
    3. jaxrpc-mapping.xml maps WSDL portType, message and operations to Java classes, methods and parameter types
    4. Application-server specific deployment descriptors, if needed
5. Create WAR file

Demo: Write and deploy JSE and EJB web services

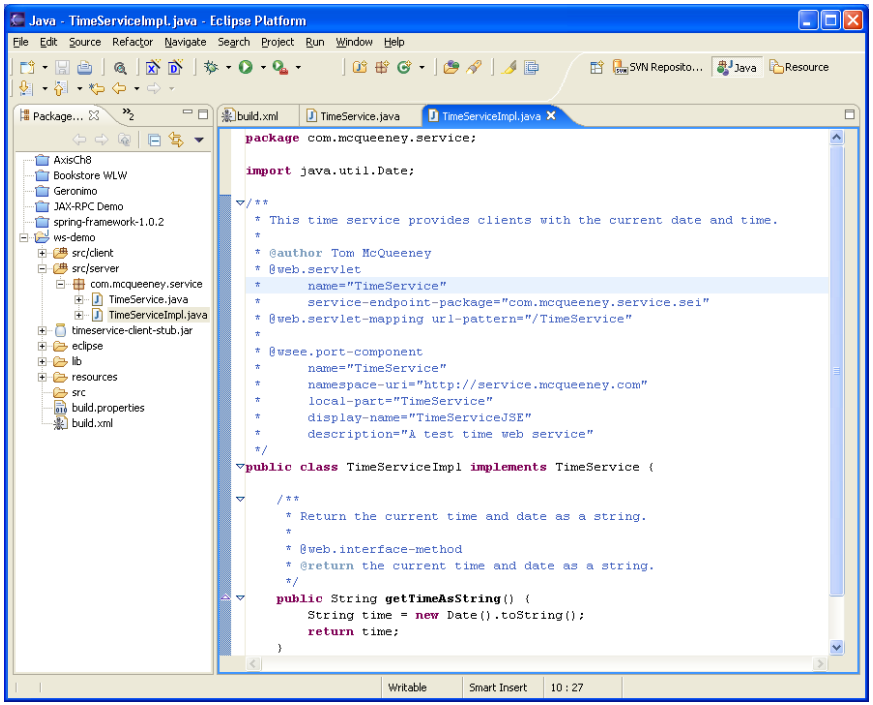
- Demo shows how to write web services as a JSE (JAX-RPC Service Endpoint) and as an EJB
- We will use Apache Geronimo as our J2EE container

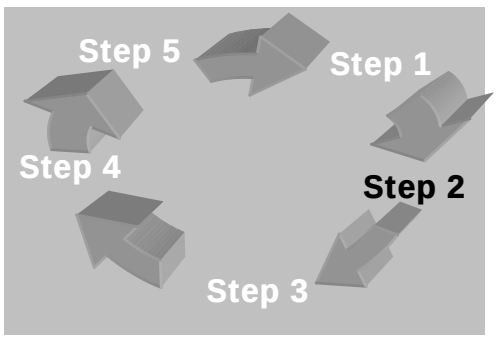


Step 1: Write remote interface (SEI)

```
package com.geronimolive.service;
import java.rmi.Remote;
import java.rmi.RemoteException;

/**
 * This is the Service Endpoint Interface for the TimeService web service.
 * It serves as a remote interface in the JAX-RPC world.
*/
public interface TimeService extends Remote {

    /**
     * Returns the current time and date as a string.
     * @return the current time and date as a string.
     * @throws RemoteException if any problem occurs.
     */
    public String getTimeAsString() throws RemoteException;
}
```

Step 2: Write web service implementation class



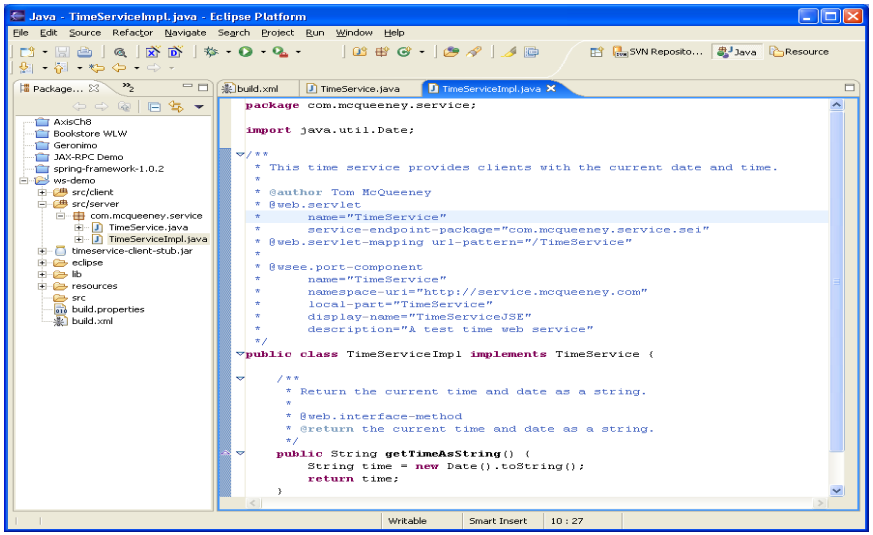



```
package com.geronimolive.service;
import java.util.Date;

/**
 * This time service provides clients with
   the current date and time.
*/
public class TimeServiceImpl implements
    TimeService {
   /**
    * Returns the current time and date
    as a string.
    */
   public String getTimeAsString() {
      String time = new Date().toString();
      return time;
   }
}
```

Step 2: Write web service implementation class

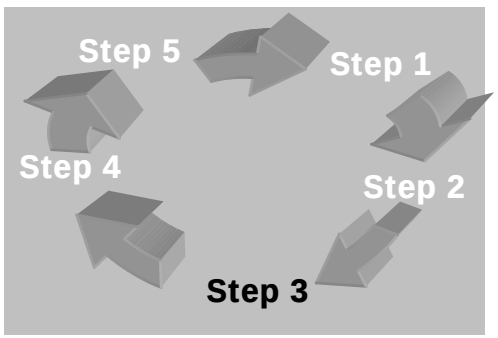
```
package com.geronimolive.service;
import java.util.Date;

/**
 * This time service provides clients with the current date and time.
*/
public class TimeServiceImpl implements TimeService {
    /**
     * Return the current time and date as a string.
     */
    public String getTimeAsString() {
        String time = new Date().toString();
        return time;
    }
}
```

Step 3: Generate WSDL




```
<axis-java2wsdl
  output="${generated.wsdl}/TimeService.wsdl"
  classname="com.geronimolive.service.TimeService"
  namespace="http://service.geronimolive.com"
  style="rpc"
  use="literal“
  location="http://${axis.server}:${axis.port}
>
  <classpath refid="axis.class.path" />
</axis-java2wsdl>
```

We'll use an open source tool, Apache Axis, to generate the WSDL from the interface class.

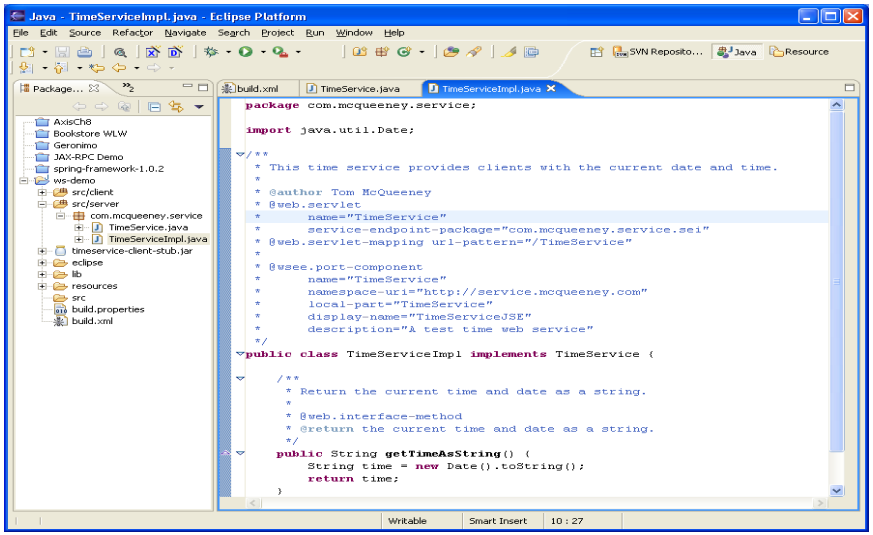


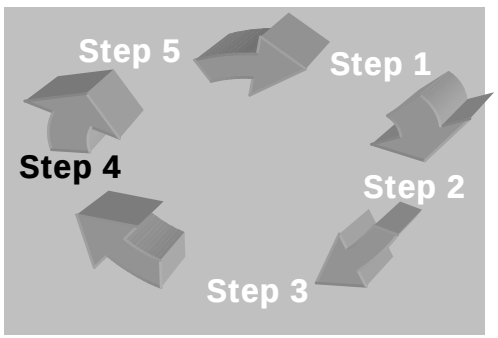
Step 3: Generate WSDL

- We'll use our vendor tool, Apache Axis, to generate the WSDL from the interface class. Ant task (abridged):

```
<axis-java2wsdl
   output="${generated.wsdl}/TimeService.wsdl"
   classname="com.geronimolive.service.TimeService"
   namespace="http://service.geronimolive.com"
   style="rpc"
   use="literal”
  location="http://${axis.server}:${axis.port}/${axis.servletpath}/
webservices/TimeService"
>
   <classpath refid="axis.class.path" />
</axis-java2wsdl>
```

Step 4: Write/generate deployment descriptors






```
<web-app xmlns="…" xmlns:xsi="…"
    xsi:schemaLocation="…"
    version="2.4"
>
    <servlet>
        <servlet-name>
        TimeService
        </servlet-name>
        <servlet-class>

    com.geronimolive.service.TimeServiceImpl
        </servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>
        TimeService
        </servlet-name>
        <url-pattern>/TimeService</url-
    pattern>
    </servlet-
</web-app>
```

web.xml (we'll generate it)

Step 4: Write/generate deployment descriptors (1 of 2)

- web.xml (generated)

```
<web-app xmlns="…" xmlns:xsi="…" xsi:schemaLocation="…"
    version="2.4"
>
    <servlet>
        <servlet-name>TimeService</servlet-name>
        <servlet-class>
        com.geronimolive.service.TimeServiceImpl
        </servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>TimeService</servlet-name>
        <url-pattern>/TimeService</url-pattern>
    </servlet-mapping>
</web-app>
```

Step 4: Write/generate deployment descriptors (2 of 2)

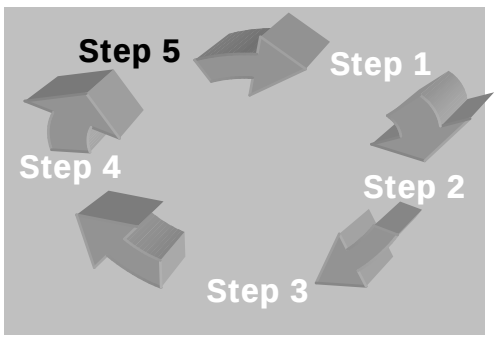
- geronimo-jetty.xml

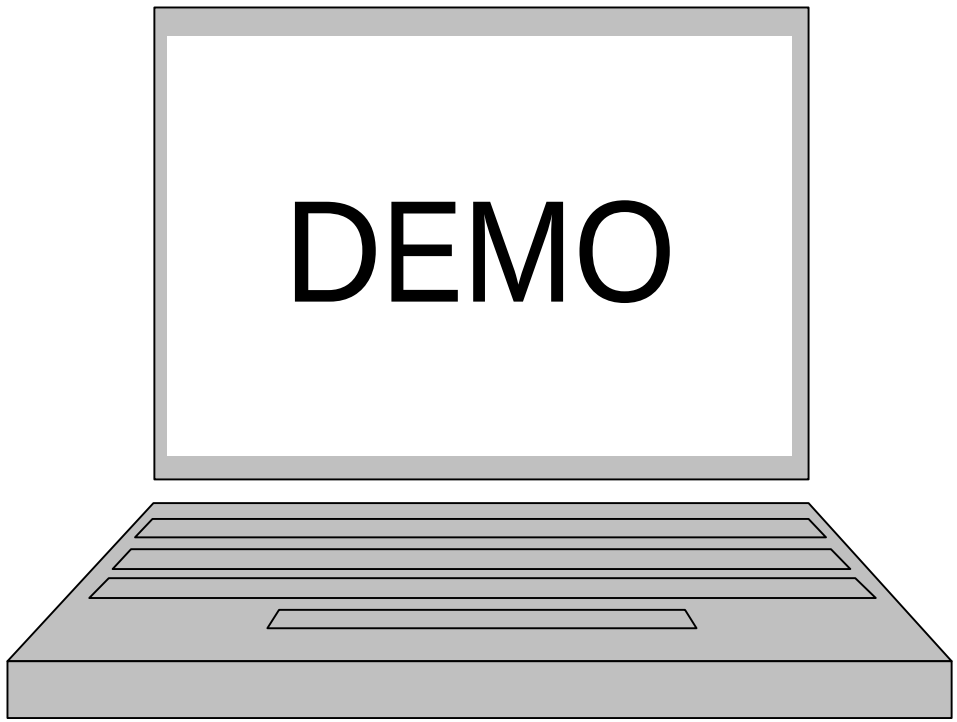
```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
    xmlns="http://geronimo.apache.org/xml/ns/web/jetty"
    configId="com/geronimolive/gcal"
    parentId="org/apache/geronimo/Server"
>
    <context-root>/TimeService</context-root>
    <context-priority-classloader>false</context-priority-classloader>
</web-app>
```

- **webservices.xml (hand-code)**
- **jaxrpc-mapping.xml (hand-code)**

Step 5: Build WAR file

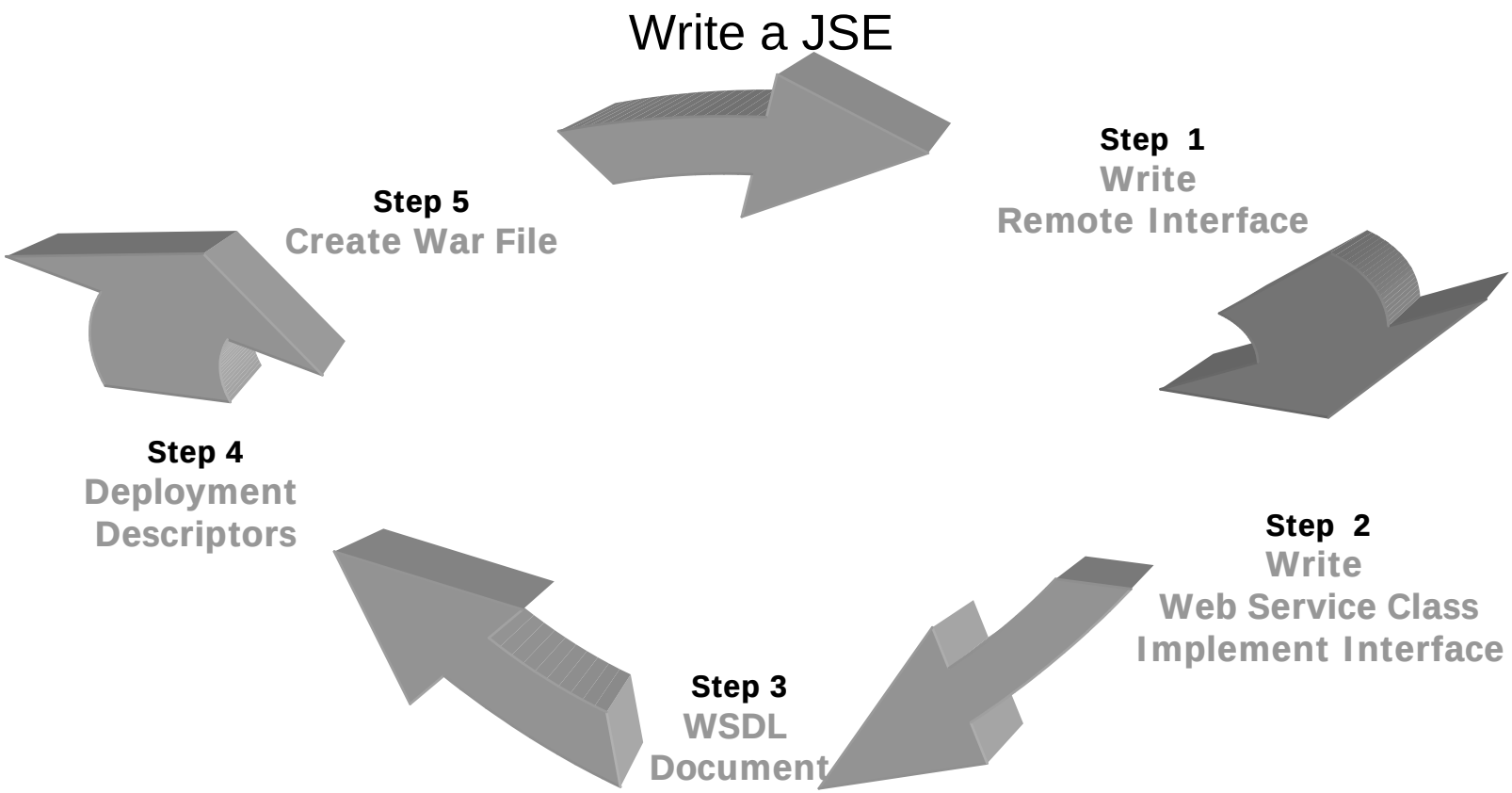
- Build the WAR file containing the JSE
- Deploy it to Geronimo





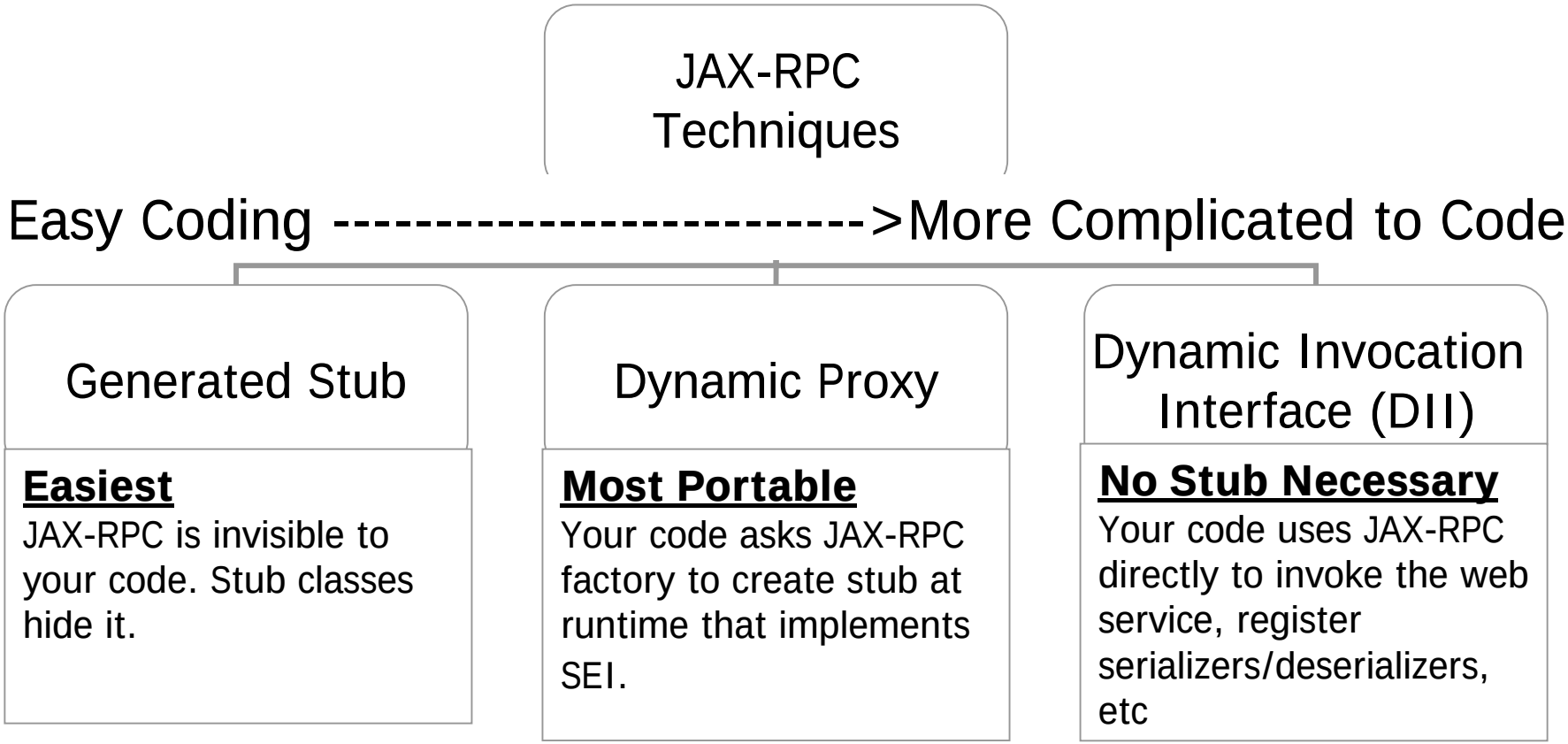
Five Steps Completed

- That's the basic 5-step process for writing a JSE



Writing web service clients

- Client model is defined by JAX-RPC 1.1 spec.
- You can write web service clients using three JAX-RPC techniques:

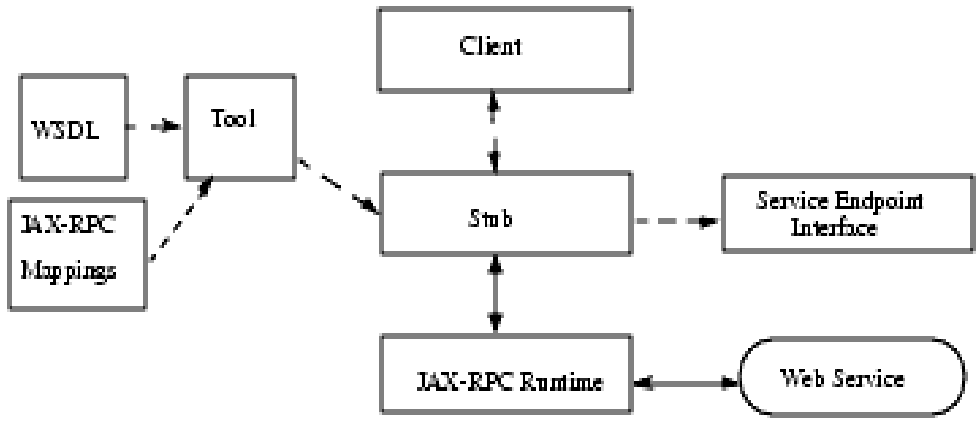


Writing web service clients

Generated Stub

Generated Stub

- Use this approach if have WSDL at coding time and can use a vendor tool to create stub classes.



- Considerations

Vendor tools produce vendor-specific implementations of the JAX-RPC classes

Runtime will require that vendor's JAR file

Your code is directly coupled to SEI and a service-locator helper class (service locator then uses/instantiates other stub classes)

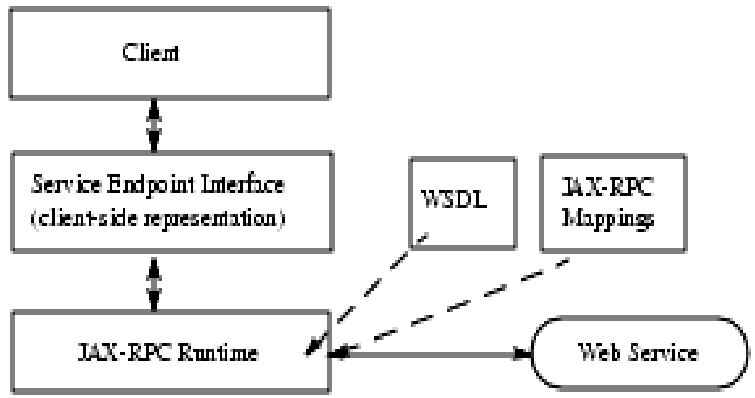
Usually this isn't a problem

Writing web service clients

Dynamic Proxy

Dynamic Proxy

- When it is important your code is not coupled to any one vendor's JAX-RPC implementation, use this approach.



- Considerations
  - Slightly more complicated to code. Your client speaks a little JAX-RPC
  - But your code is directly coupled only to SEI
  - Your code uses standard JAX-RPC classes to create proxy at runtime that implements the SEI

Additional runtime overhead

Writing web service clients

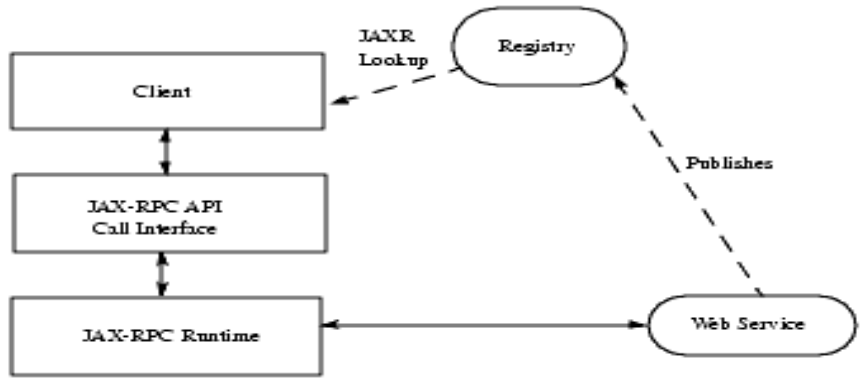
Dynamic Invocation Interface (DII)

Dynamic invocation interface (DII)

- Use this approach if your client doesn't know anything about the web service at development time (e.g. no WSDL)

- Considerations

  Your client uses JAX-RPC Call interface to invoke a web service.



Useful for tools, i.e. dynamically retrieve WSDL then show user on a GUI what services are available and allow user to select what should be called

Your client needs to set the web service's **URL, operation to invoke**, its required parameters and return values, XML serializers/deserializers needed, etc.

Writing web service clients

- WS clients written to run inside a J2EE container vs. outside differ only in how the web service interface is looked up:
  - Inside-container clients use JNDI to look up a Service object
  - Outside-container clients use a `ServiceFactory` class to obtain a Service object.
- Client API allows you to set standard client properties to: set a user name and password for HTTP Basic authentication, participate in a stateful session if the remote service supports it

Code example: Writing an external WS client

- Generate stub (and helper classes) and write client

Code example: Steps to writing an EJB endpoint

1. Write remote interface (the service endpoint interface)
2. Write SLSB that implements the interface's methods and optionally implements the interface
3. Write (generate) WSDL document that describes the web service (vendor tool)
4. Write (or generate) deployment descriptors:
    1. ejb-jar.xml
    2. webservices.xml and jaxrpc-mapping.xml
    3. Application-server (Geronimo) specific deployment descriptor
5. Create EJB-JAR file

Tips for writing interoperable web services (1 of 2)

- Be aware that different web service platforms work differently
- If you expect to have wide range of client applications calling your web service, WS-I is your friend
- WS-I, the Web Services Interoperability Organization, defines standards for web services. Follow its web service specification "profiles"
- Basic Profile 1.0 (Aug. 2003 but in draft since 2002) restricts how XML Schema, SOAP, WSDL and UDDI may be used if you want your web service to be callable from the largest set of platforms/languages
- Avoid using rpc/encoded web services: WS-I BP 1.0 does not support
- You can send complex data types (mapped to XML in jaxrpc-mapping.xml), but for utmost interoperability, send only the data types defined by XML Schema language:
  - string, int, integer, long, short, float, double, boolean, byte, date, dateTime, base64Binary, ...
  - Those all have equivalent Java types you can use as parameters or return values: String, int, BigInteger, long, short, float, double, boolean, byte, Calendar, byte[]

Tips for writing interoperable web services (2 of 2)

- Use WS-I testing tools (available in Java and C#) to test your web service for interoperability. Tests aren't fool-proof, but a good step to find interop issues before customers do.
- J2EE 1.4 complies with WS-I BP 1.0, but adds features not allowed by BP 1.0:
  - RPC/Encoded SOAP messages
  - SOAP Messages with Attachments (W3C spec for MIME-encoded attachments like images and audio) allowed by J2EE 1.4 but not allowed by BP 1.0.
    - Note: SwA allowed by BP 1.1 (4-month old spec)

Future of J2EE web services

- J2EE 5: JAX-RPC 2.0 (JSR-221) and Java annotations (JRS-181)
- Goals of JAX-RPC 2.0:
  - Support WS-I BP 1.1 and Attachment Profile 1.0
  - Support SOAP 1.2 and WSDL 2.0
  - Make writing WS simpler by using metadata to annotate classes and methods (JSR 181). No more SEI, webservices.xml, jaxrpc-mapping.xml, and WSDL document. These can be generated from metadata in the web services implementation
  - Remove requirement to define a remote object: No more requirement to extend java.rmi.Remote and to throw RemoteException
  - Compliant tools ***may*** produce J2EE 1.4 JAX-RPC artifacts so you can use new features in a J2EE 1.4 app server running J2SE 5.0
  - Better support for document-centric web services
  - Integrate with JAXB to define Java-to-XML mappings instead of defining own mapping mechanism
  - Web service versioning

Resources for more information (1 of 2)

- Major J2EE 1.4 web services specs:
  - WSEE 1.1 / JSR 109 (Web Services for J2EE): `jcp.org/en/jsr/detail?id=921`
  - JAX-RPC 1.1 / JSR 101: `jcp.org/en/jsr/detail?id=101`
- W3C (World Wide Web Consortium) web services standards
  - SOAP 1.1: `http://www.w3.org/TR/2000/NOTE-SOAP-20000508/`
  - WSDL 1.1: `http://www.w3.org/TR/wsdl`
- WS-I (Web Services Interoperability Organization)
  - Basic Profile 1.0, `www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html`
  - Interop Java testing tool: `www.ws-i.org/Testing/Tools/2004/01/WSI_Test_Java_01.00.01_bin.zip`
- J2ME clients calling web services
  - JSR 172, `jcp.org/en/jsr/detail?id=172`

Resources for more information (2 of 2)

- Articles:
  - "Build interoperable Web services with JSR-109"
    www-128.ibm.com/developerworks/webservices/library/ws-jsrart/
  - "Developer's introduction to JAX-RPC, Part 1: Learn the ins and outs of the JAX-RPC type-mapping system"
    www-106.ibm.com/developerworks/webservices/library/ws-jaxrpc1/
  - "… Part 2: Mine the JAX-RPC specification to improve Web service interoperability"
    www-106.ibm.com/developerworks/webservices/library/ws-jaxrpc2/
- Books:
  - "J2EE Web Services" by Richard Monson-Haefel (Addison-Wesley)
  - "Designing Web Services with the J2EE Platform" by Inderjeet Singh et al. (Addison-Wesley):
    java.sun.com/blueprints/guidelines/designing_webservices/html/index.html

Questions

Feel free to find me during the conference or email me:

tom@mcqueeney.com