JAX-WS

Agenda

- Quick overview of JAX-WS
  - Differences from JAX-RPC
- JAX-WS programming Model
  - Layered programming model
  - Server side
  - Client side

Quick Overview of JAX-WS 2.0

- Simpler way to develop/deploy Web services
  - Plain Old Java Object (POJO) can be easily exposed as a Web service
  - No deployment descriptor is needed - use Annotation instead
  - Layered programming model
- Part of Java SE 6 and Java EE 5 platforms
- Integrated data binding via JAXB 2.0
- Protocol and transport independence

Layered Programming Model

Programming Model at the Server Side

Two ways to create a Web Service

- Starting from a WSDL file (top-down approach)
  - Generate classes using ***wsimport***
    - WS interface
    - WS implementation skeleton class
    - Add business logic to the WS implementation class
    - Build, deploy, and test
- Starting from a POJO (bottom-up approach)
  - Annotate POJO
  - Build and deploy
  - WSDL file generated automatically

Server-Side Programming Model: (Starting from POJO)

1. Write a POJO implementing the service
2. Add @WebService annotation to it
3. Optionally, inject a WebServiceContext
4. Deploy the application
5. Point your clients at the WSDL
   - e.g. http://myserver/myapp/MyService?WSDL

Example 1: Servlet-Based Endpoint

```
@WebService
public class Calculator {
    public int add(int a, int b) {
                        return a+b;
                }
    }
```

- @WebService annotation
  - All public methods become web service operations
- WSDL/Schema generated automatically
  - Default values are used

Example 2: EJB-Based Endpoint

```
@WebService
@Stateless
        public class Calculator {
@Resource
WebServiceContext context;
        public int add(int a, int b) {
                return a+b;
        }
}
```

- It's a regular EJB 3.0 component, so it can use any EJB features
  - Transactions, security, interceptors...

Customizing through Annotations

```
@WebService(name=”CreditRatingService”,
                        targetNamespace=”http://example.org”)
public class CreditRating {

@WebMethod(operationName=”getCreditScore”)
public Score getCredit(
@WebParam(name=”customer”) Customer c) {
        // ... implementation code ...
        }
}
```

Demo

- Build a “Hello World” Web service using @WebService annotation
- Test the Web service
- Display the generated WSDL document

Client Side programming: Java SE & Java EE

Java SE Client-Side Programming

1. Point a tool (wsimport) at the WSDL for the service
   wsimport http://example.org/calculator.wsdl
1. Generate annotated classes and interfaces
2. Call new on the service class
3. Get a proxy using a get<ServiceName>Port method
4. Invoke any remote operations

Example: Java SE-Based Client

```
CalculatorService svc = new CalculatorService();

Calculator proxy = svc.getCalculatorPort();

int answer = proxy.add(35, 7);
```

- No need to use factories
- The code is fully portable
- XML is completely hidden from programmer

Demo

- Build and run a Web service client of “Hello World”
- Web service using the WSDL document

Java EE Client-Side Programming

1. Point a tool (wsimport) at the WSDL for the service
   wsimport http://example.org/calculator.wsdl

1. Generate annotated classes and interfaces

1. Inject a @WebServiceReference of the appropriate type
   1. No JNDI needed

1. Invoke any remote operations

Example: Java EE-Based Client

```
@Stateless
public class MyBean {
      // Resource injection
      @WebServiceRef(CalculatorService.class)
      Calculator proxy;
      public int mymethod() {
            return proxy.add(35, 7);
      }
```

Annotations

Annotations Used in JAX-WS

- JSR 181: Web Services Metadata for the Java Platform
- JSR 222: Java Architecture for XML Binding (JAXB)
- JSR 224: Java API for XML Web Services (JAXWS)
- JSR 250: Common Annotations for the Java Platform

JSR 181 (Web Services Metadata) Annotations

1. javax.jws.WebService
2. javax.jws.WebMethod
3. javax.jws.OneWay
4. javax.jws.WebParam
5. javax.jws.WebResult
6. javax.jws.HandlerChain
7. javax.jws.soap.SOAPBinding

JSR 224 (JAX-WS) Annotations

1. javax.xml.ws.BindingType
2. javax.xml.ws.RequestWrapper
3. javax.xml.ws.ResponseWrapper
4. javax.xml.ws.ServiceMode
5. javax.xml.ws.WebEndpoint
6. javax.xml.ws.WebFault
7. javax.xml.ws.WebServiceClient
8. javax.xml.ws.WebServiceProvider
9. javax.xml.ws.WebServiceRef
10. javax.xml.ws.Action
11. javax.xml.ws.FaultAction

JSR 222 (JAXB) Annotations

1. javax.xml.bind.annotation.XmlRootElement
2. javax.xml.bind.annotation.XmlAccessorType
3. javax.xml.bind.annotation.XmlType
4. javax.xml.bind.annotation.XmlElement
5. javax.xml.bind.annotation.XmlSeeAlso

JSR 250 (Common Annotations) Annotations

- javax.annotation.Resource
- javax.annotation.PostConstruct
- javax.annotation.PreDestroy

Protocol and Transport Independence

Protocol and Transport Independence

- Typical application code is protocol-agnostic
- Default binding in use is SOAP 1.1/HTTP
- Server can specify a different binding, e.g. @BindingType(SOAPBinding.SOAP12HTTP_BINDING)
- Client must use binding specified in WSDL
- Bindings are extensible, expect to see more of them
  - e.g. SOAP/Java Message Service(JMS) or XML/SMTP

Example

```
@WebService
@BindingType
(value=SOAPBinding.SOAP12HTTP_BINDING)
public class AddNumbersImpl {
// More code
```

Handler

Handler Types

- JAX-WS 2.0 defines a *Handler* interface, with subinterfaces *LogicalHandler* and *SOAPHandler*.
- The Handler interface contains
  - handleMessage(C context)
  - handleFault(C context)
  - C extends *MessageContext*
  - A property in the *MessageContext* object is used to determine if the message is inbound or outbound
- *SOAPHandler* objects have access to the full soap message including headers
- Logical handlers are independent of protocol and have access to the payload of the message

Logical Handler

```
public class MyLogicalHandler implements
    LogicalHandler<LogicalMessageContext> {
public boolean handleMessage(LogicalMessageContext
    messageContext) {
    LogicalMessage msg = messageContext.getMessage();
    return true;
        }
        // other methods
}
```

SOAP Handler

```
public class MySOAPHandler implements
    SOAPHandler<SOAPMessageContext> {
public boolean handleMessage(SOAPMessageContext messageContext) {
SOAPMessage msg = messageContext.getMessage();
return true;
}
// other methods
}
```

Example

```
@WebService
@BindingType(value=SOAPBinding.SOAP12HTTP_BINDING)
public class AddNumbersImpl {
// More code
```


JAX-WS:
Advanced Features

Agenda

- Dispatch
- Messaging layer

Dispatch

Dispatch<T> Interface

- Dynamic, low level API
- Methods
  - T invoke(T msg)
  - Response<T> invokeAsync(T msg)
  - Future<?> invokeAsync(T msg, AsyncHandler<T> h)
  - void invokeOneWay(T msg)
- Supported types for T
  - javax.xml.transform.Source
  - javax.activation.DataSource
  - javax.xml.soap.SOAPMessage
  - Object—when using JAXB

Messaging Layer

Messaging in JAX-WS 2.0

- Lower layer in JAX-WS
- Mostly out of view until you need it
- Many more control knobs → more complexity
- Motivated by advanced applications:
  - Dynamic clients (e.g. a management console)
  - Dynamic servers (e.g. a gateway)
  - Protocols without an established description language

What Is a WebServiceContext?

- Used on the server via Dependency Injection

  @Resource WebServiceContext Context;
- *WebServiceContext* gives access to
  - Security information (e.g. getUserPrincipal method)
  - The message context
  - In the future, other information on the client/service
- *MessageContext* is a bag of properties
  - Data which is not part of the XML payload for a message ends up in the context
  - E.g. HTTP query string

  http://myserver/myapp/MyService?format=image/jpeg

Client-Side RequestContext

- Bag of properties for use by the application
- Any data is copied to the message context before each invocation
- Useful to configure a message on-the-fly
  - Endpoint address
  - Username/password
  - Attachments
  - HTTP query string
  - SOAP action...

```
BindingProvider.getRequestContext()
```

Benefits Over Network APIs

- Higher-level (no sockets)
- JAXB support built in
- No need to parse MIME multipart packages
- Can plug in message handlers
- Bindings get tested for interoperability
- Extensible to new protocols/transports

Client-side Messaging API: Dispatch

```
// T is the type of the message
public interface Dispatch<T> {

    // synchronous request-response
    T invoke(T msg);

    // async request-response
    Response<T> invokeAsync(T msg);
    Future<?> invokeAsync(T msg, AsyncHandler<T> h);

    // one-way
    void invokeOneWay(T msg);
}
```

Choosing a Message Type

1. Do you want to see the whole protocol message?

   If yes, use MESSAGE mode and the appropriate message type (e.g. SOAPMessage for SOAP 1.1/1.2)

1. If not, use PAYLOAD mode and answer the next question
2. Do you want to use JAXB?
3. If yes, use java.lang.Object
4. Otherwise use javax.xml.transform.Source
5. Pass message type and mode to:

   Service.createDispatch(mode, port, type)

Examples

Dispatch<T> → T invoke(T msg) "T in, T out"

- Payload mode with JAXB:

  `Dispatch<Object>`

- SOAP Message mode:

  `Dispatch<SOAPMessage>`

- HTTP binding payload mode without JAXB:

  `Dispatch<Source>`

- HTTP binding message mode:

  `Dispatch<DataSource>`

Server-side Messaging API: Provider

```
// T is the type of the message
public interface Provider<T> {

    T invoke(T msg, Map<String,Object> context);

}
```

- The same considerations for mode and message type apply here
- Use @ServiceMode to select a mode

Example: Payload Mode, No JAXB

```
@ServiceMode(Service.Mode.PAYLOAD)
public class MyProvider
        implements Provider<Source> {
  public Source invoke(Source request,
                       Map<String,Object> context) {
    // process the request using XML APIs, e.g. DOM
    Source response = ...

    // return the response message payload
    return response;
  }
}
```

Example: Message Mode, SOAP 1.1/1.2

```
@ServiceMode(Service.Mode.MESSAGE)
public class SOAPProvider
        implements Provider<SOAPMessage> {
  public SOAPMessage invoke(SOAPMessage request,
                            Map<String,Object> context) {
    // process the request using SAAJ
    SOAPMessage response = ...

    // return the response message payload
    return response;
  }
}
```

Example:Polling

```
@WebService
public interface CreditRatingService{
 // sync operation
      Score getCreditScore(Customer customer);
      // async operation w/ polling
      Response<Score>
          getCreditScoreAsync(Customer customer);
      // async operation w/callback
      Future<?>
          getQuoteAsync(Customer customer,
                AsyncHandler<Score> handler);
}
```

Example:Polling Client

```
CreditRatingService svc = ...;
Response<Score> response =
   svc.getCreditScoreAsync(customerFred);
// client app does other things...
// ready to deal with the response

// no cast needed, thanks to generics
Score score = response.get();

// or use
// Score score = response.get(10L, TimeUnit.SECONDS);
// to wait 10 seconds
```

Example: Callback

```
@WebService
public interface CreditRatingService{
// sync operation
    Score getCreditScore(Customer customer);
    // async operation w/ polling
    Response<Score>
        getCreditScoreAsync(Customer customer);
    // async operation w/callback
    Future<?>
        getQuoteAsync(Customer customer,
            AsyncHandler<Score> handler);
}
```

Example: Callback Client

```
CreditRatingService svc = ...;
Future<?> invocation =
   svc.getCreditScoreAsync(customerFred,
new AsyncHandler<Score>() {
        public void handleResponse
            (Response<Score> response) {
           Score score = response.get();
           // do work here...
        }
      });
// to cancel the request, use
// invocation.cancel(true);
```

- com.sun.xml.ws.transport.http.servlet.WSServlet

Day 2 Agenda

- Brief previous day coverage -- 10:00-10:30
- WSDL and soap in detail – 10:30-11:30
- Create a Web client – 12:00 – 12:30
- JAXB 12:30-1:00
- Creating a DII client – 2:00- 3:30
- Creating a asynchronous client 3:30 – 4:00
- Implementing Provider and creating its client 4:00 – 5:00
- Creating and deploying handlers
- Analysis of The WSDL File