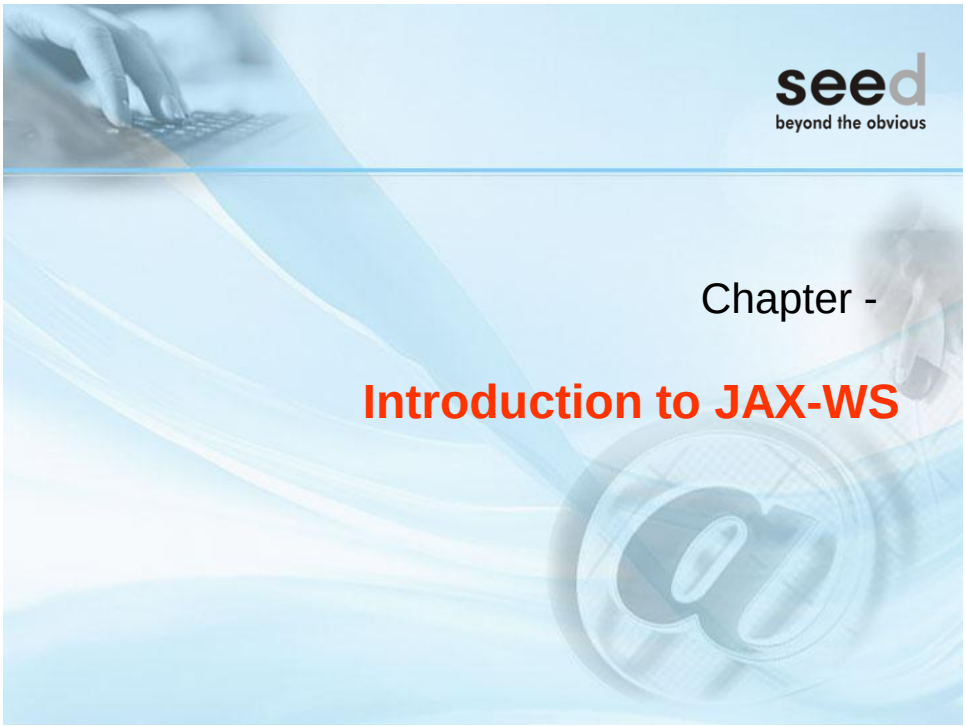

Objectives

- At the end of this chapter you will be able to understand
  - What JAX-WS 2.0 is
  - Enhancements in JAX-WS 2.0
  - Different approaches of Web service development (bottom-up, top-down)
  - Different end points
  - Different types of JAX-WS clients

Introduction

- JAX-WS 2.0 is a new programming model that simplifies application development through support of a standard, annotation-based model to develop Web Service applications and clients.
- The JAX-WS 2.0 specification strategically aligns itself with the current industry trend towards a more document-centric messaging model and replaces the remote procedure call programming model as defined by JAX-RPC.

Introduction

- JAX-WS is the strategic programming model for developing Web services and is a required part of the Java Platform, Enterprise Edition 5 (Java EE 5).
- The implementation of the JAX-WS programming standard provides the following enhancements for developing Web services and clients:

Enhancements in JAX-WS 2.0

- Better platform independence for Java applications.
  - Using JAX-WS APIs, development of Web services and clients is simplified with better platform independence for Java applications.
  - JAX-WS takes advantage of the dynamic proxy mechanism to provide a formal delegation model with a pluggable provider. This is an enhancement over JAX-RPC, which relies on the generation of vendor-specific stubs for invocation.

Enhancements in JAX-WS 2.0

- Annotations
  - JAX-WS introduces support for annotating Java classes with metadata to indicate that the Java class is a Web service.
  - JAX-WS supports the use of annotations based on the Metadata Facility for the Java Programming Language (JSR 175) specification, the Web Services Metadata for the Java Platform (JSR 181) specification and annotations defined by the JAX-WS 2.0 specification.

Enhancements in JAX-WS 2.0

- Annotations (continued)
  - Using annotations within the Java source and within the Java class simplifies development of Web services by defining some of the additional information that is typically obtained from *deployment descriptor* files, *WSDL* files, or mapping metadata from XML and WSDL files into the source artifacts.
  - For example, we can embed a simple *@WebService* tag in the Java source to expose the bean as a Web service.

```
@WebService
public class QuoteBean implements StockQuote {
public float getQuote(String sym) { ... }

}
```


Enhancements in JAX-WS 2.0

- Annotations (continued)
  - The @WebService annotation tells the server runtime to expose all public methods on that bean as a Web service.
  - Additional levels of granularity can be controlled by adding additional annotations on individual methods or parameters.
  - Using annotations makes it much easier to expose Java artifacts as Web services.
  - In addition, as artifacts are created from using some of the top-down mapping tools starting from a WSDL file, annotations are included within the source and Java classes as a way of capturing the metadata along with the source files.

Enhancements in JAX-WS 2.0

- Annotations (continued)
  - Using annotations also improves the development of Web services within a team structure because we do not need to define every Web service in a single or common deployment descriptor as required with JAX-RPC Web services.
  - Taking advantage of annotations with JAX-WS Web services allows *parallel development* of the service and the required metadata.

Enhancements in JAX-WS 2.0

- Invoking Web services asynchronously
  - With JAX-WS, Web services are called both synchronously and asynchronously. JAX-WS adds support for both a *polling* and *callback* mechanism when calling Web services asynchronously.
  - Using a polling model, a client can issue a request, get a response object back, which is polled to determine if the server has responded. When the server responds, the actual response is retrieved.
  - Using the callback model, the client provides a callback handler to accept and process the inbound response object.
  - Both the polling and callback models enable the client to focus on continuing to process work without waiting for a response to return, while providing for a more dynamic and efficient model to invoke Web services

Enhancements in JAX-WS 2.0

- Using resource injection
  - JAX-WS supports resource injection to further simplify development of Web services.
  - JAX-WS uses this key feature of Java EE 5 to shift the burden of creating and initializing common resources in a Java runtime environment from our Web service application to the application container environment itself.
  - JAX-WS provides support for a subset of annotations that are defined in JSR-250 for resource injection and application lifecycle in its runtime.

Enhancements in JAX-WS 2.0

- Using resource injection (continued)
  - The following example illustrates using the *@Resource* annotation for resource injection:

```
@WebService public
class MyService {
    @Resource private WebServiceContext ctx;
    public String echo (String input) { //some code
    }
    }
```

Enhancements in JAX-WS 2.0

- Data binding with Java Architecture for XML Binding (JAXB) 2.0
  - JAX-WS leverages the JAXB 2.0 API and tools as the binding technology for mappings between Java objects and XML documents.
  - JAX-WS tooling relies on JAXB tooling for default data binding for two-way mappings between Java objects and XML documents.

Enhancements in JAX-WS 2.0

- Dynamic and static clients
  - The dynamic client API for JAX-WS is called the dispatch client (javax.xml.ws.Dispatch). The dispatch client is an XML messaging oriented client.
  - The static client programming model for JAX-WS is called the proxy client. The proxy client invokes a Web service based on a Service Endpoint interface (SEI) which must be provided.

Enhancements in JAX-WS 2.0

- Support for Message Transmission Optimized Mechanism (MTOM)
  - Using JAX-WS, we can send binary attachments such as images or files along with Web services requests.
  - JAX-WS adds support for optimized transmission of binary data as specified by Message Transmission Optimization Mechanism (MTOM).

Enhancements in JAX-WS 2.0

- Multiple data binding technologies
  - JAX-WS exposes the following binding technologies to the end user:
    - XML Source
    - SOAP Attachments API for Java (SAAJ) 1.3, and
    - Java Architecture for XML Binding (JAXB) 2.0.

Enhancements in JAX-WS 2.0

- Support for SOAP 1.2
  - Support for SOAP 1.2 has been added to JAX-WS 2.0.
  - JAX-WS supports both SOAP 1.1 and SOAP 1.2 so that we can send binary attachments such as images or files along with Web services requests.
  - JAX-WS adds support for optimized transmission of binary data as specified by MTOM.

Enhancements in JAX-WS 2.0

- New development tools
  - JAX-WS provides the **wsgen** and **wsimport** command-line tools for generating portable artifacts for JAX-WS Web services.
  - When creating JAX-WS Web services, we can start with either a WSDL file or an implementation bean class.
  - If we start with an implementation bean class, use the ***wsgen*** command-line tool to generate all the Web services server artifacts, including a WSDL file if requested.
  - If we start with a WSDL file, use the ***wsimport*** command-line tool to generate all the Web services artifacts for either the server or the client.
  - The wsimport command line tool processes the WSDL file with schema definitions to generate the portable artifacts, which include the service class, the service endpoint interface class, and the JAXB 2.0 classes for the corresponding XML schema.

Developing JAX-WS Web services (bottom-up development)

- When developing a JAX-WS Web service starting from JavaBeans, we can use a bean that already exists and then enable the implementation for JAX-WS Web services.
- The use of annotations simplifies the enabling of a bean for Web services. Adding the @WebService annotation to the bean defines the application as a Web service and how a client can access the Web service.
- JavaBeans can have a service endpoint interface, but it is not required.

Developing JAX-WS Web services (bottom-up development)

- Enabling JavaBeans for Web services includes annotating the bean and the optional service endpoint interface, assembling all artifacts required for the Web service, and deploying the application.
- We are not required to develop a WSDL file because the use of annotations can provide all of the WSDL information necessary to configure the service endpoint or the client. It is, however, a best practice to develop a WSDL file.

Developing JAX-WS Web services (top-down development)

- We can use a top-down development approach to create a JAX-WS Web service with an existing WSDL file using JavaBeans.
- We can use the JAX-WS tool, **wsimport**, to process a WSDL file and generate portable Java artifacts that are used to create a Web service. The portable Java artifacts created using the **wsimport** tool are:
  - Service endpoint interface (SEI)
  - Service class
  - Exception class that is mapped from the wsdl:fault class (if any)
  - Java Architecture for XML Binding (JAXB) generated type values which are Java classes mapped from XML schema types

Service Implementation Bean (SIB)

- WS-Metadata 2.0 defines the requirements for deploying a Java class as a Web service. Classes that meet these requirements are called service implementation beans (***SIB***s).
- While JAX-RPC required services to be implemented using service endpoint interfaces (SEIs) that had to extend *java.rmi.Remote*, that requirement no longer applies with JAX-WS and WS-Metadata.

Service Implementation Bean (SIB)

- In addition, both POJOs and EJBs that conform to the service implementation bean requirements can be deployed as Web services. This is a big improvement over J2EE 1.4, which has two different bean specifications—
  - one for EJB deployment (which requires a Web service to be implemented with the stateless session bean interfaces)
  - and one for Web container (servlet) deployment

seed
beyond the obvious

Different Endpoints

- WSEE 1.2 specifies that a POJO, as long as it meets the requirement spelled out in WS-Metadata for a SIB, can be used to implement a Web service deployed to the Web container. This is commonly referred to as a ***servlet endpoint***
- WSEE 1.2 specifies that a stateless session bean can be used to implement a Web service to be deployed in the EJB container. This is commonly referred to as an ***EJB endpoint***.
- The requirements for creating a SIB from a stateless session bean are spelled out in WSEE
- Demo:- .\eclipse-workspace-servletendpoint,

.\eclipse-workspace-ejbendpoint

- Demo:- .\eclipse-workspace-servletendpoint,

.\eclipse-workspace-servletendpoint

seed
beyond the obvious

Developing JAX-WS clients

- The JAX-WS client programming model supports both the Dispatch client API and the Dynamic Proxy client API.
- The Dispatch client API is a dynamic client programming model, whereas the static client programming model for JAX-WS is the Dynamic Proxy client.
- The Dispatch and Dynamic Proxy clients enable both synchronous and asynchronous invocation of JAX-WS Web services.

Developing JAX-WS clients

- ***Dispatch client***: Use this client when we want to work at the XML message level or when we want to work without any generated artifacts at the JAX-WS level.
- ***Dynamic Proxy client***: Use this client when we want to invoke a Web service based on a service endpoint interface.

Quick Recap . . .

- In this session we have seen :
  - Introduction to enhancements in JAX-WS 2.0
  - Introduction to Developing JAX-WS Web services
    - Top down approach
    - Bottom up approach
  - Introduction to JAX-WS clients
    - Dynamic Proxy client
    - Dispatch client