Developing Web Services Using Java Technology

Volume I • Student Guide

DWS-4050-EE6 Rev A

D65185GC11
Edition 1.1
September 2010
D69077

ORACLE

Disclaimer

This document contains proprietary information, is provided under a license agreement containing restrictions on use and disclosure, and is protected by copyright and other intellectual property laws. You may copy and print this document solely for your own use in an Oracle training course. The document may not be modified or altered in any way. Except as expressly permitted in your license agreement or allowed by law, you may not use, share, download, upload, copy, print, display, perform, reproduce, publish, license, post, transmit, or distribute this document in whole or in part without the express authorization of Oracle.

The information contained in this document is subject to change without notice. If you find any problems in the document, please report them in writing to: Oracle University, 500 Oracle Parkway, Redwood Shores, California 94065 USA. This document is not warranted to be error-free.

Sun Microsystems, Inc. Disclaimer

This training manual may include references to materials, offerings, or products that were previously offered by Sun Microsystems, Inc. Certain materials, offerings, services, or products may no longer be offered or provided. Oracle and its affiliates cannot be held responsible for any such references should they appear in the text provided.

Restricted Rights Notice

If this documentation is delivered to the U.S. Government or anyone using the documentation on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS
The U.S. Government's rights to use, modify, reproduce, release, perform, display, or disclose these training materials are restricted by the terms of the applicable Oracle license agreement and/or the applicable U.S. Government contract.

Trademark Notice

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

This page intentionally left blank.

This page intentionally left blank.

Contents

List of Figures

List of Tables

List of Code Examples

About This Course

On completion of this course, you should be able to:

- Understand and explain how web services frameworks can be used to deploy and consume services.
- Understand the trade-offs associated with the use of SOAP-based or RESTful web services.
- Understand and use the JAX-WS technology to deploy and consume web services.
- Understand and use the JAX-RS technology to deploy and consume web services.
- Understand how to deploy web services that also leverage other Java™ Platform, Enterprise Edition (JavaEE)technologies, such as the web container infrastructure, Enterprise Java Beans, or the Java Persistence API.
- Understand, explain, and apply best practices and design patterns when designing and deploying web services using either JAX-WS or JAX-RS technologies.

Course Map

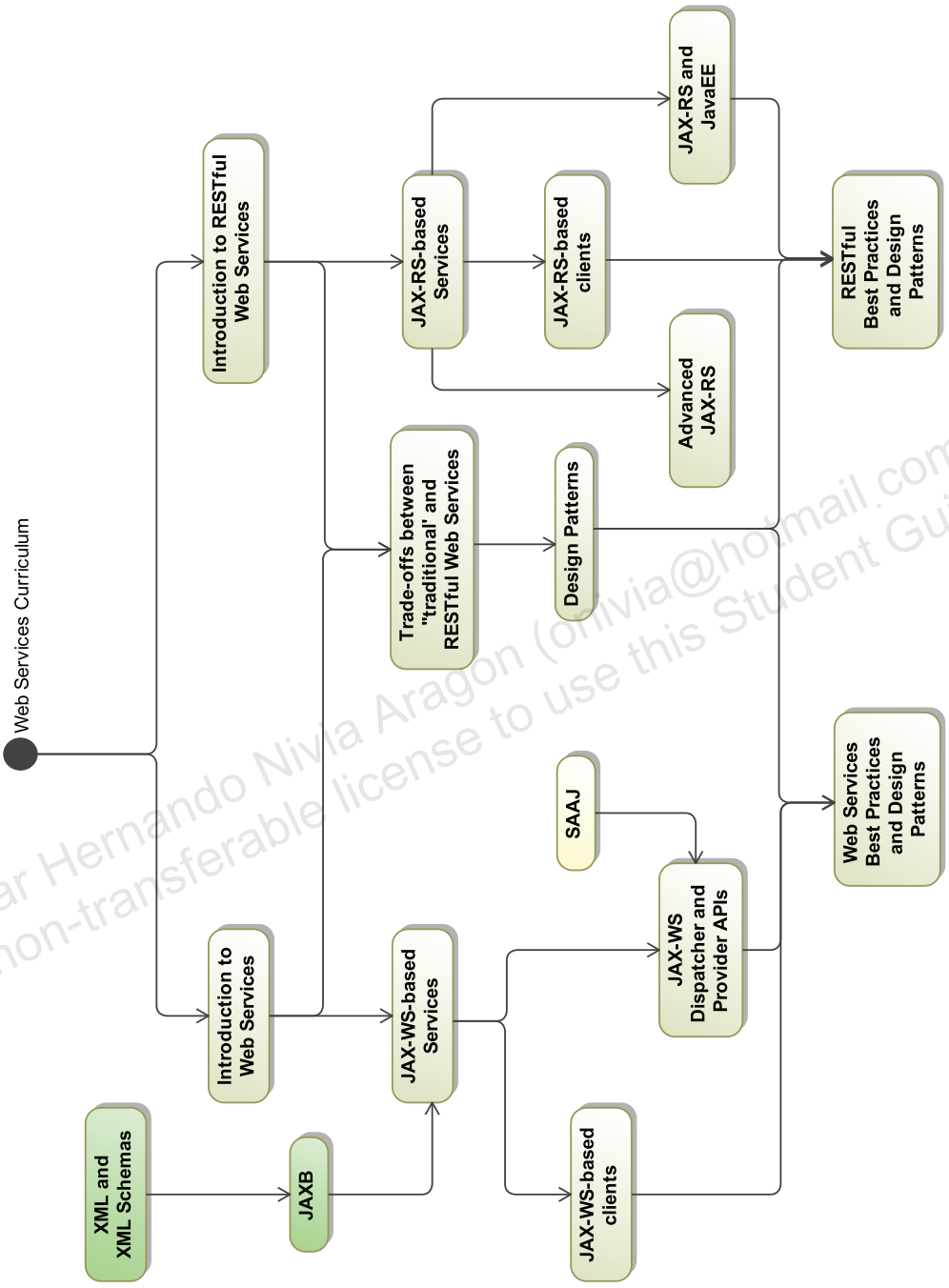


Figure 1: Course Map

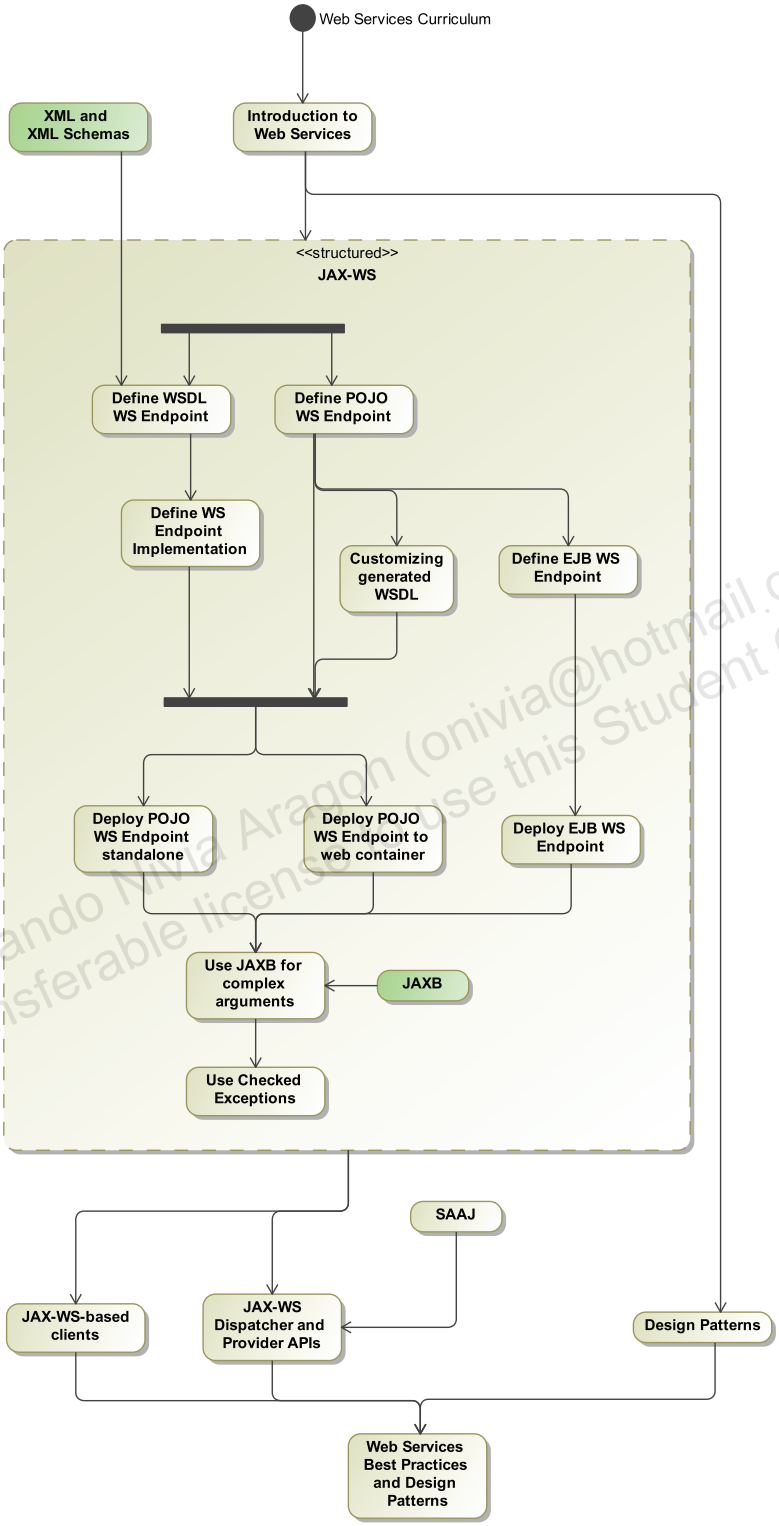
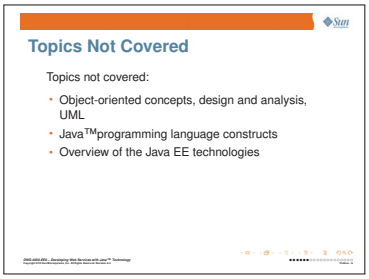


Figure 2: Course Map – JAXWS

Topics Not Covered

In order to concentrate on the topics directly relevant to this course, we make certain assumptions about other topics that students ought to be familiar with, before attending this course.

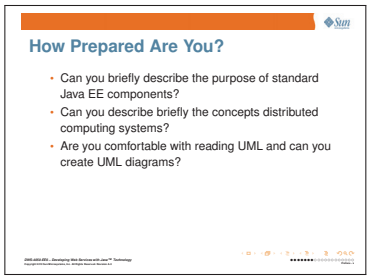


The list of these other topics includes:

- Object-oriented concepts, design and analysis, UML
- Java™programming language constructs
- Overview of the Java EE technologies

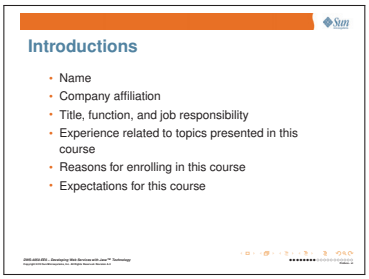
How Prepared Are You?

- Can you briefly describe the purpose of standard Java EE components?
- Can you describe briefly the concepts distributed computing systems?
- Are you comfortable with reading UML and can you create UML diagrams?



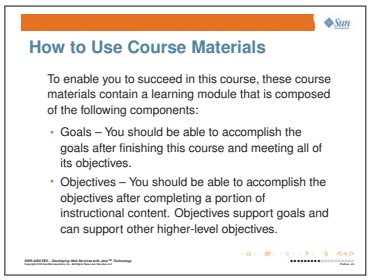
Introductions

- Name
- Company affiliation
- Title, function, and job responsibility
- Experience related to topics presented in this course
- Reasons for enrolling in this course
- Expectations for this course



How to Use Course Materials

To enable you to succeed in this course, these course materials contain a number of learning modules, which are designed to be equals parts lecture and discussion, on the on the one hand, and practical exercises to apply the concepts just learned. Each learning module is composed of the following components:



- Goals – You should be able to accomplish the goals after finishing this course and meeting all of its objectives.
- Objectives – You should be able to accomplish the objectives after completing a portion of instructional content. Objectives support goals and can support other higher-level objectives.
- Lecture – The instructor presents information specific to the objective of the module. This information helps you learn the knowledge and skills necessary to succeed with the activities.
- Activities – The activities take on various forms, such as an exercise, self-check, discussion, and demonstration. Activities help you facilitate the mastery of an objective.
- Visual aids – The instructor might use several visual aids to convey a concept, such as a process, in a visual form. Visual aids commonly contain graphics, animation, and video.

Conventions

The following conventions are used in this course to represent various training elements and alternative learning resources.

Self-check – Identifies self-check activities, such as matching and multiple-choice.

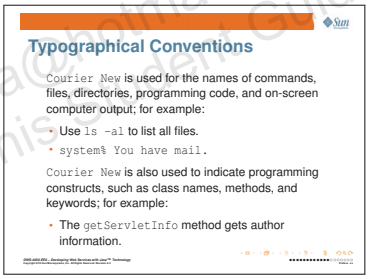
Additional resources - Indicates other references that provide additional information on the topics described in the module.

Discussion - Indicates a small-group or class discussion on the current topic is recommended at this time.

Note - Indicates additional information that can help students but is not crucial to their understanding of the concept being described. Students should be able to understand the concept or complete the task without this information. Examples of notational information include keyword shortcuts and minor system adjustments.

Code Listing – indicates that the complete source code listing associated with a code example is available by following this link.

`Courier New` is used for the names of commands, files, directories, programming code, and on-screen computer output; for example:

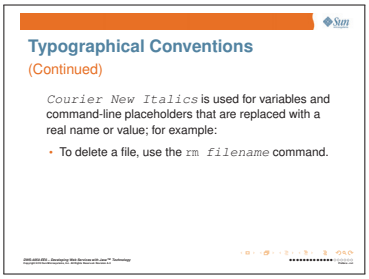


- Use `ls -al` to list all files.
- `system% You have mail.`

`Courier New` is also used to indicate programming constructs, such as class names, methods, and keywords; for example:

- The `getServletInfo` method gets author information.

`Courier New Italics` is used for variables and command-line placeholders that are replaced with a real name or value; for example:

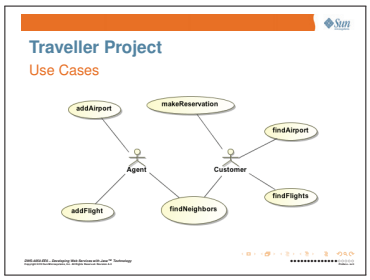


- To delete a file, use the *`rm filename`* command.

Course Projects

The Traveller Project

The Traveller application is the project from which the course draws all of its examples. The Traveller application supports a set of use cases that would necessary to provide an online airline reservation web site, as illustrated by Figure 3.



addAirport Adds a new airport to the system.

findAirport Finds an airport known to the system, by airport code, or by airport name.

findNeighbors Find all airports that can be reached from a given airport in one flight.

findFlights Find all (direct) flights between two airports.

makeReservation Make a reservation for a customer on a flight. In this system, a round-trip flight requires two reservations, one for each one-way trip.

addFlight Add a flight between the two airports specified to the system.

The uses cases listed above are just a sample of the uses cases that such an application should support – but they are enough to describe all the examples used throughout the course.

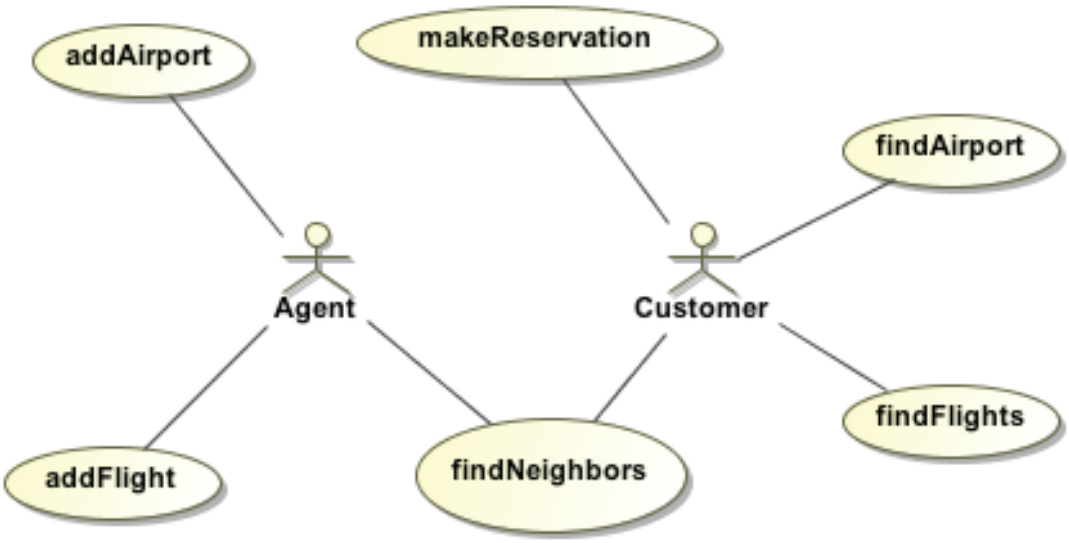
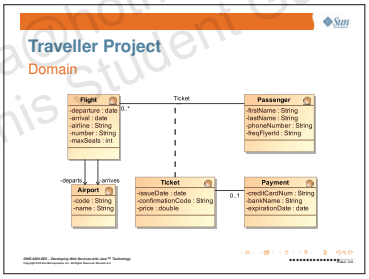


Figure 3: Use Cases for Traveller Project

Figure 4 is a sketch of the domain types used within the Traveller project to capture the information necessary to implement the uses cases described in Figure 3. In the design used by the course, all of these types are persistent – and all of these are implements as JPA entity classes, to minimize the effort required to manage their persistence.



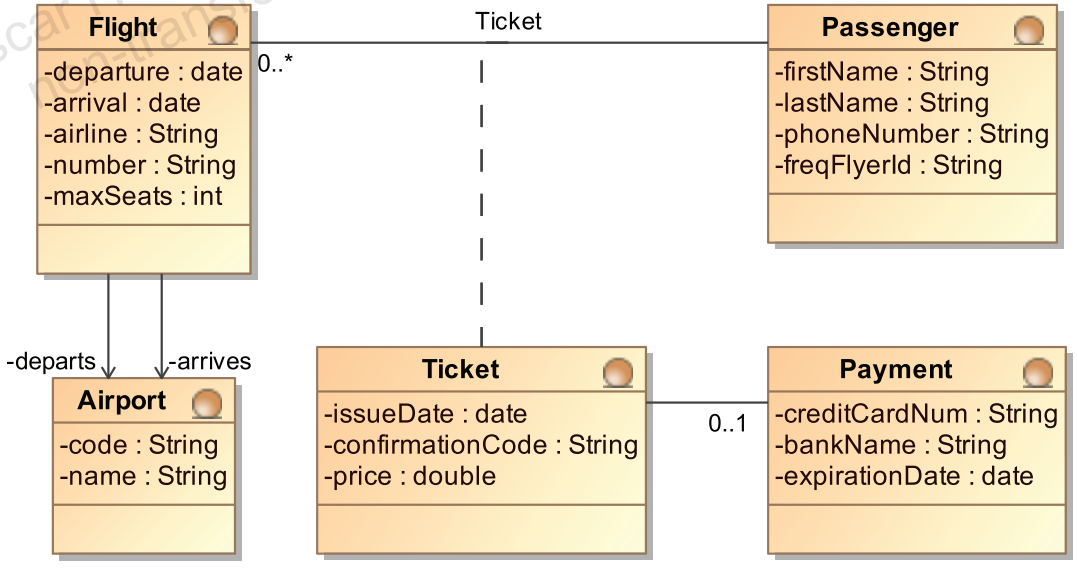


Figure 4: Domain Types for Traveller Project

Figure 5 presents a UML diagram of a number of candidate service types as *façades* to the business logic required by the application. (An SOA architecture for this application would probably further abstract some of the details exposed by the API presented in this diagram).

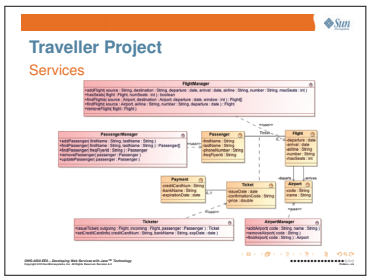
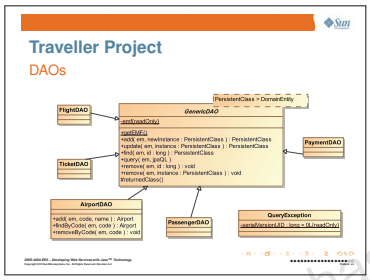


Figure 6 presents a UML diagram of a number of *Data Access Object* classes, which try to hide the JPA machinery that is used to manage the persistent instances used by the application. In the implementation offered for the examples, these DAO classes will be implemented two ways: as simple Java classes, and as Enterprise Java beans. The first implementation is less successful than the second one at hiding the JPA machinery: absent EJBs, the application logic must manage persistence contexts and transactions explicitly – and this may need to be exposed all the way to business methods...



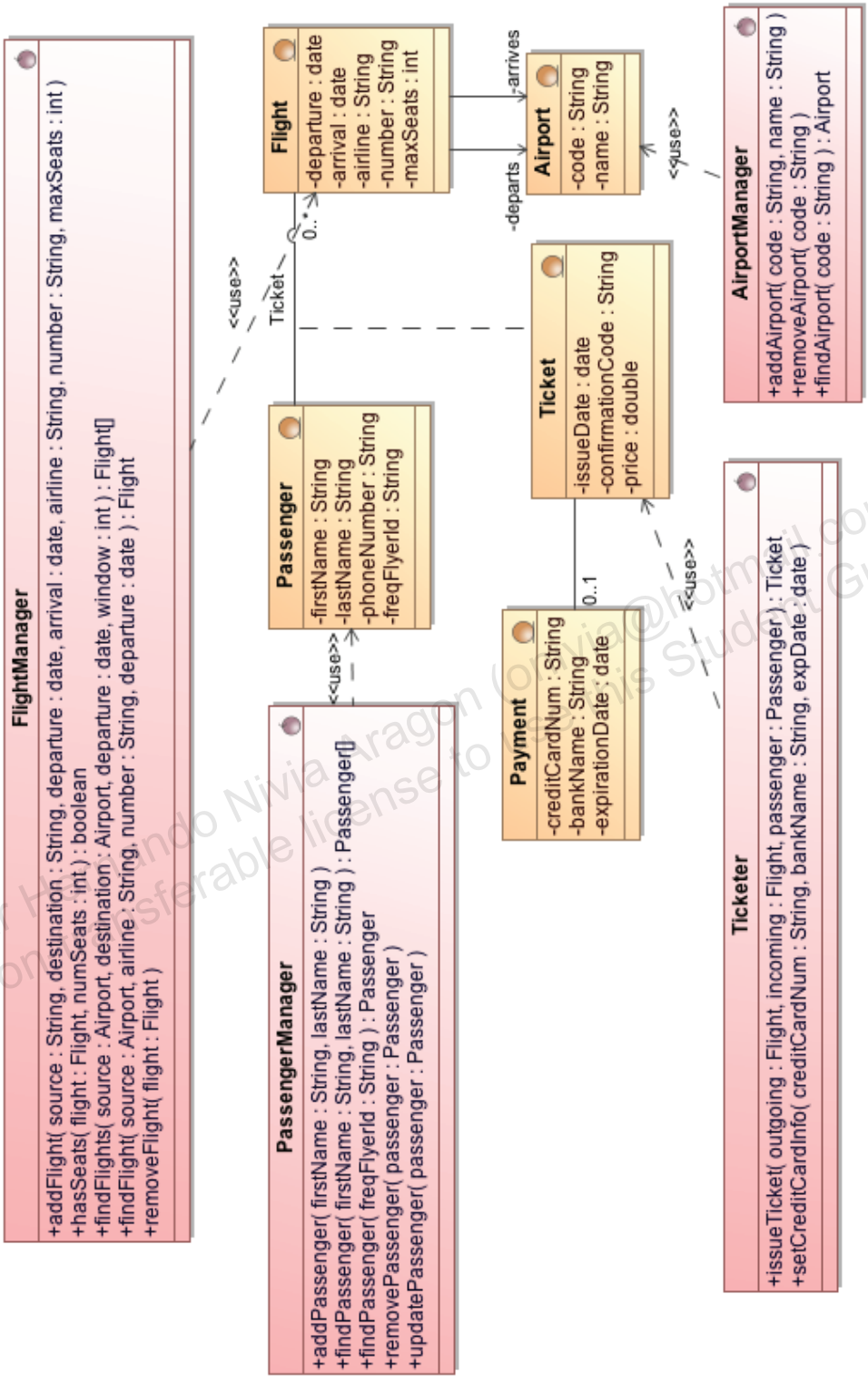


Figure 5: Application Services for the Traveller Project

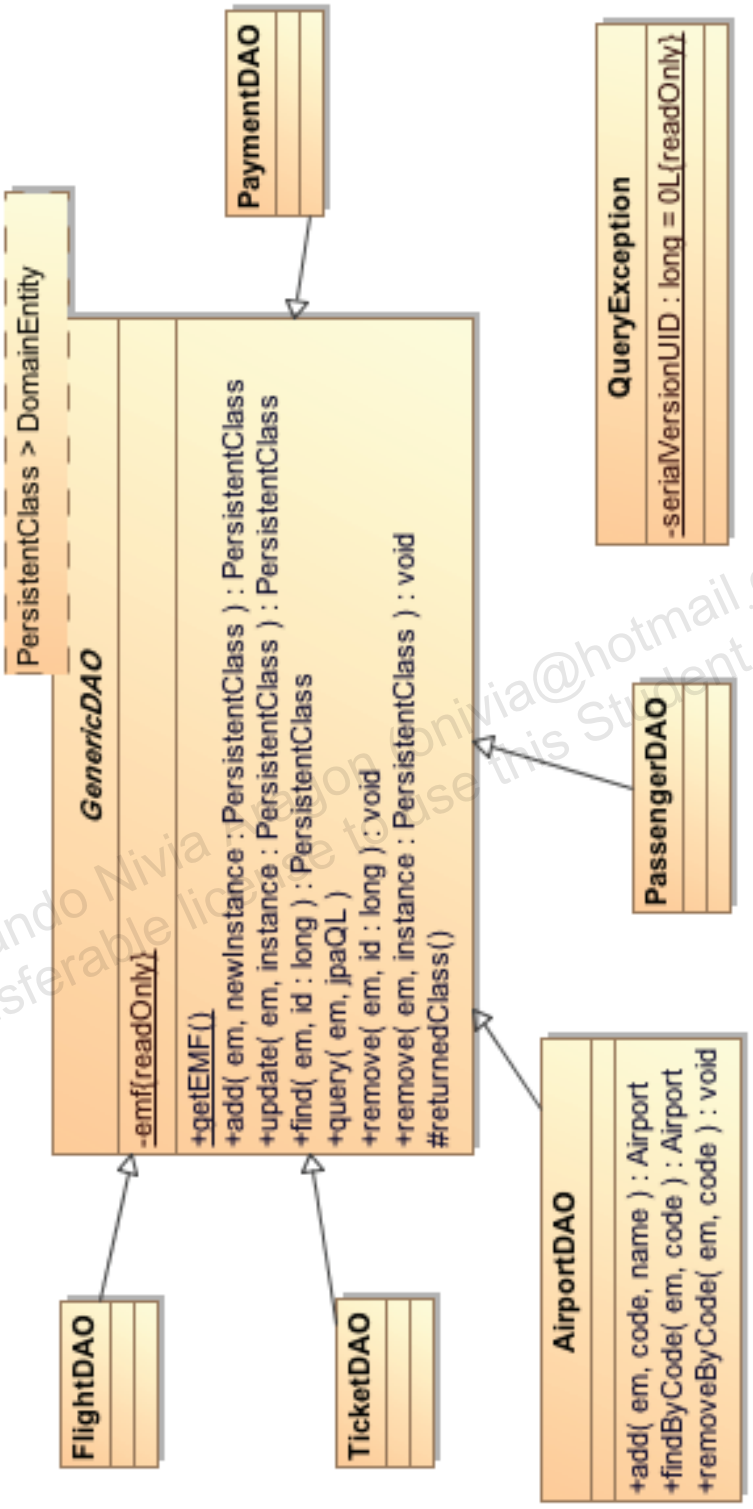
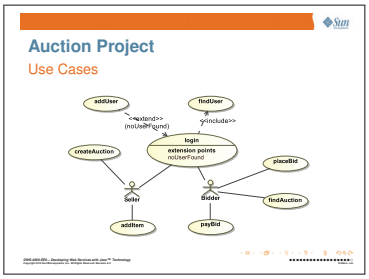


Figure 6: DAO Classes for the Traveller Project

The Auction Project



The Auction application supports the set of use cases necessary to provide an online auction web site, as illustrated by Figure 7:

`login` All users of the Auction system must first log on to the system, before they can invoke other operations. To log on, all users must authenticate themselves (perhaps by providing a userid and a password). This use case includes either of the `addUser` or `findUser` use cases.

It might be important to distinguish between sellers and bidders, since there are tasks in the Auction system that only one kind of user may request.

`addUser` Adds a new user to the system database. This use case is invoked by the `login` use case, when a new user accesses the Auction system for the first time. In addition to the user ID and password already supplied, the system asks the user for additional personal information: name, address (both location and email). Optionally, the user will be able to provide credit card information to be used as a default method of payment.

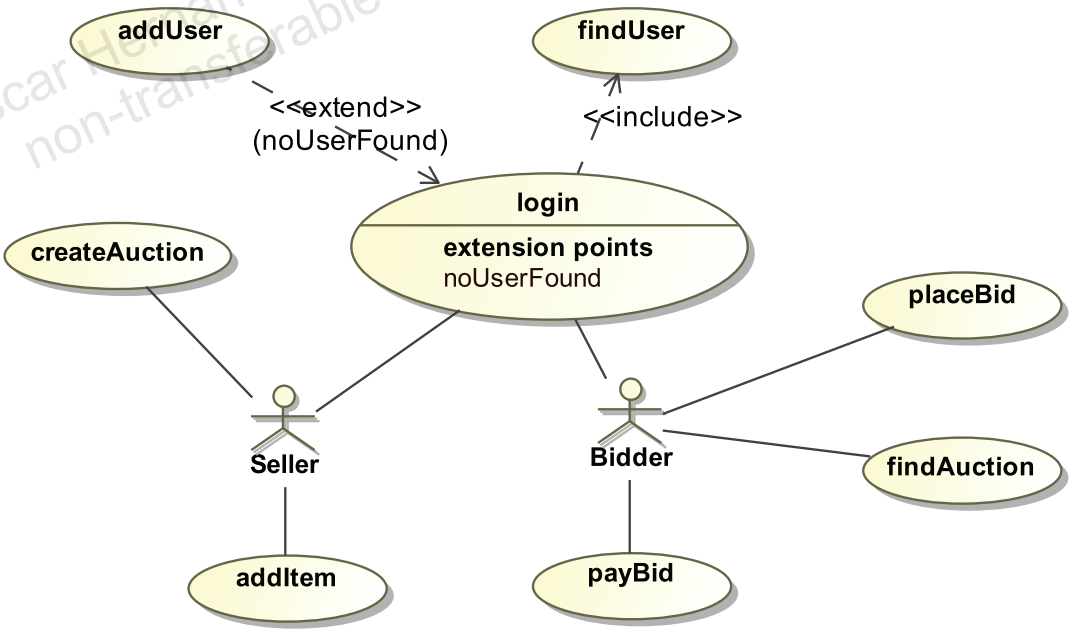


Figure 7: Use Cases for Auction System

`findUser` Finds an existing user in the system database. This use case is invoked by the `login` use case, when an existing user returns to the Auction system.

`addItem` Sellers are allowed to create items to be auctioned off. The item to be auctioned and the auction for the item are considered to be independent from each other. This allows a seller who has multiples of a given item to hold separate auctions for each copy, while reusing the same item for each auction. Items hold information, such as the name for the item, a description, and a picture.

`createAuction` Sellers are allowed to create auctions, to auction items off. Each auction holds information, such as the item being auctioned, the initial bid price, the date the auction starts and the date it will end (or how long the auction will last), and a list of all bids placed on it.

The Auction system should also allow the seller to create multiple auctions for a number of copies of the same item conveniently. This can be achieved by allowing the seller to create a first auction, and then allowing the seller to create "another just like it", or by allowing the seller to specify the number of copies of an auction to create, as part of the process of creating each auction.

`placeBid` Bidders are allowed to bid on auctions. Given an auction, the bidder is allowed to enter a bid on that auction. The Auction system should not allow a bidder to place a bid on an auction that is lower than the current bid price for that auction. The Auction system should also allow the user to find out what the current bid price for the auction is, or even notify the user automatically when the current bid price for the auction changes, or when the bidder is no longer the current high bidder for the auction.

The bidder should be able to select a collection of auctions they are interested in bidding on, and to allow the bidder to place bids on any of the auctions in that collection. In this case, the Auction system should present the bidder with the current bid price for each of the auctions in the collection, plus a running total of the amount of money the bidder is currently bidding on all auctions in the list, taken together. Also, bidders should be able to find out what the current bid prices for all auctions in the list are, or be notified automatically when the current prices change, or when the bidder is no longer the high bidder for any of the auctions in the collection.

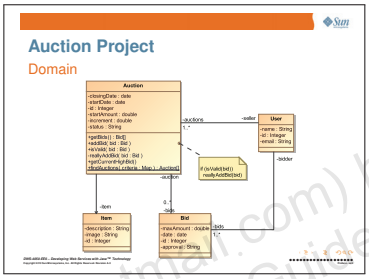
`findAuction` Users must be able to provide criteria to be used to retrieve a collection of bids in which the user might be interested. The user must also be able to narrow down the results of a previous query.

These query operations must be supported while other users are using

these same auctions. That is, users must be allowed to find auctions of interest while other users do the same, or bid on some of the same auctions.

`payBid` Bidders must be able to pay for their winning bid on an auction, once the action is over. The Auction system offers to use the credit card information the user entered when the user first registered with the system, although it also allows the user to provide an alternative means of payment. The system should also notify the seller of that auction, when the winning bidder actually pays for that auction.

The business model for the Auction system represents all the information necessary to support the use cases listed previously. Figure 8 illustrates the following abstractions, as a starting point to build the business tier for this application:



`Auction` Each instance represents an auction. It is responsible for the dates the auction starts and ends, the initial and winning bid price, and a list of all bids placed on this auction.

`Item` Each instance represents an item that can be auctioned off. It is responsible for information about each item, such as its name, description, or a picture of the item.

`Bid` Each instance represents a bid that a user has placed on an auction. It is responsible for information about the bid, such as the maximum amount the bidder was willing to bid on this bid, the time the bid was placed, and the bidder who placed the bid.

`User` Each instance represents a user of the Auction system. It is responsible for information about each user, such as the user's name, address, email, or credit card information.

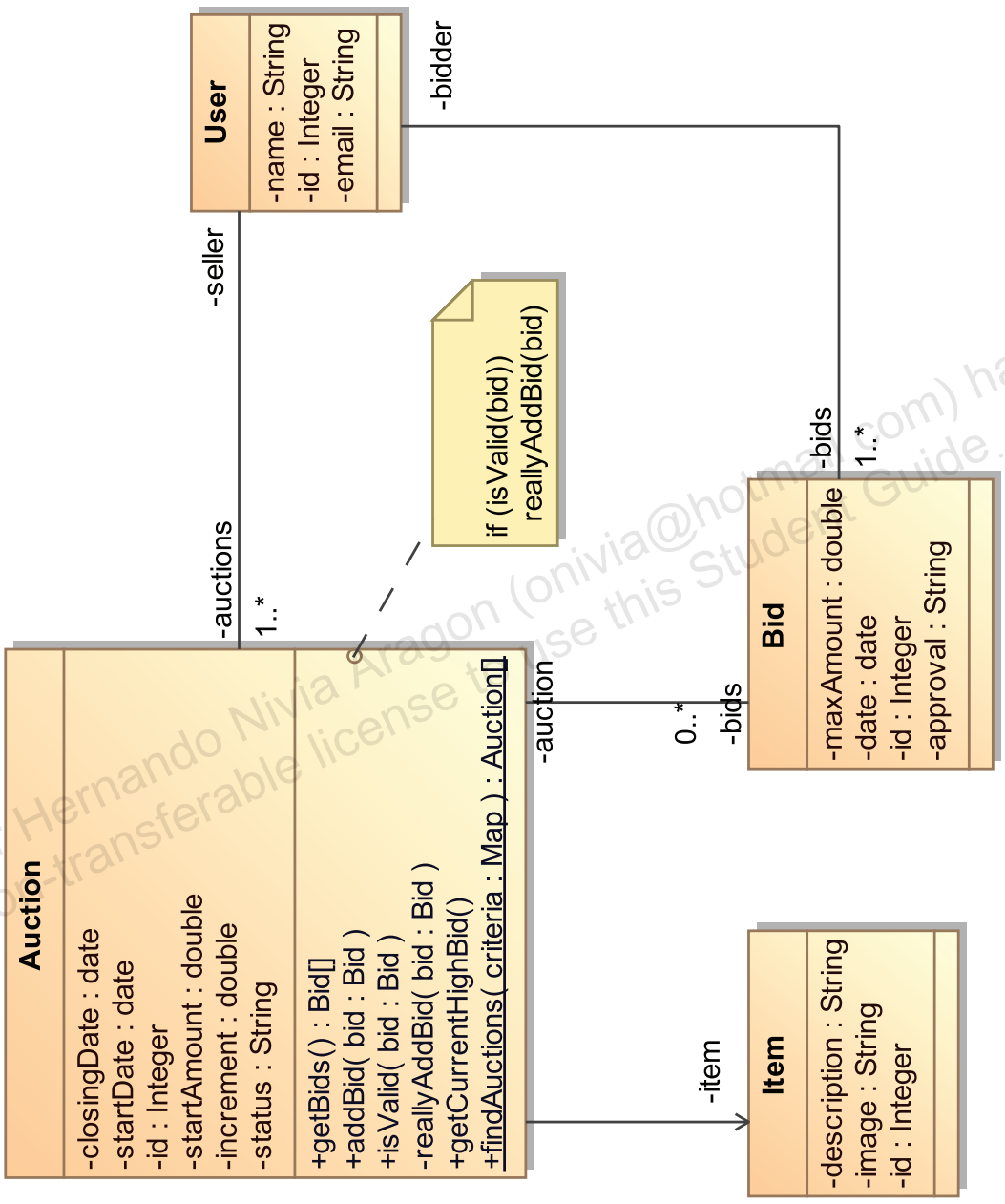
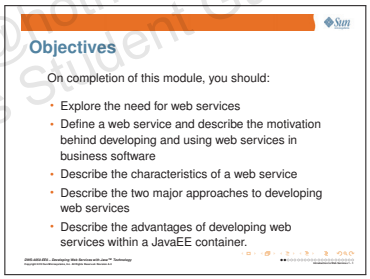


Figure 8: Domain Types for Auction System

Chapter 1

Introduction to Web Services

On completion of this module, you should:

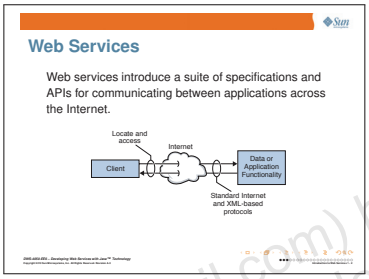


- Explore the need for web services
- Define a web service and describe the motivation behind developing and using web services in business software
- Describe the characteristics of a web service
- Describe the two major approaches to developing web services
- Describe the advantages of developing web services within a JavaEE container.

What Is a Web Service?

A web service is a piece of business logic that is:

- Accessible through standards based Internet protocols, such as HTTP
- Platform independent and language independent



Web services are services offered on the web. In a typical web services scenario, a business application sends a request to a service at a URL. The service receives the request, processes it, and returns a response. An often-cited example of a web service is that of a stock quote service, in which the request asks for the current price of a specified stock and the response returns the stock price. This is one of the simplest forms of a web service.

Web services depend on the ability of parties to communicate with each other even if they are using different information systems. Web services often transmit information represented using XML. XML (eXtensible Markup Language), which makes data portable, is a key technology in addressing this need. Another representation in common use is JSON.

A web service is independent of the hardware or software platform on which it is implemented. It is also independent of the programming language in which it is written. This enables web services-based applications to be loosely coupled, component-oriented, cross-technology implementations. Web services can be used alone, or with other web services, to carry out complex business transactions.

Exploring the Need for Web Services

Web services incorporate existing standards and introduce a suite of specifications and APIs for communicating between application components across the Internet. Web services provide a standardized mechanism for remotely accessing data and application functionality. The World Wide Web Consortium (W3C) describes a web service as "... a software system identified by a Uniform Resource Identifier (URI), whose public interfaces and bindings are defined and described using Extensible Markup Language (XML). Its definition can be discovered by other software systems. These systems then interact with the web service in a manner prescribed by its definition, using XML-based messages conveyed by Internet protocols."

Figure 1.1 illustrates a web service as a technology for accessing data and application functionality over the Internet using standard Internet protocols and XML-based message formats. The Hypertext Transport Protocol (HTTP) is the most commonly used web service transport protocol. A very common XML-based Web services message format is defined by the Simple Object Access Protocol (SOAP) specification.

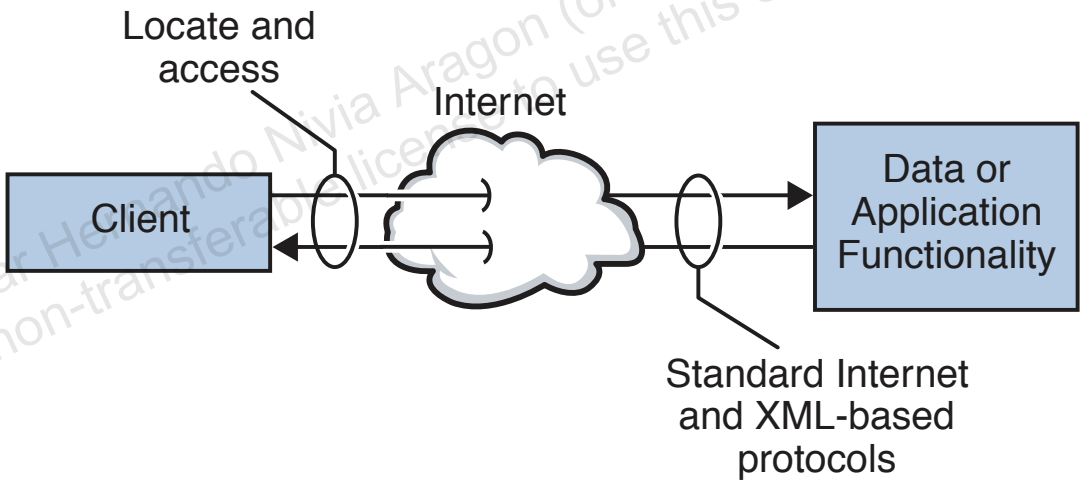


Figure 1.1: A Web Service

The Drive Towards Web Services

Some of the forces driving web service standardization include:

- Roadblocks that prevent business-to-business (B2B) communication from reaching its full potential, such as the following:

  - The cost of developing and administering applications that rely on Common Object Request Broker Architecture (CORBA) or Enterprise JavaBeans™(EJB™) technology, or implement interchange standards such as Electronic Data Interchange (EDI), can be great.
  - The industry had failed to standardize on a common message exchange format. Systems developed using the Distributed Component Object Model (DCOM) architecture cannot directly interoperate with EJB systems, Health Level Seven (HL7) must be adapted to CORBA, and so forth.
  - The cost of integrating existing applications with those from other businesses increases the cost of developing and maintaining software.
- Limitations of business-to-consumer (B2C) capabilities, such as the following:
  - Conventional document-presentation methods have limited functionality.
  - Web application functionality that is designed for human interaction is typically hard or impossible to automate, and cannot be easily aggregated into new capabilities.
- Consolidating the implementation of B2B and B2C capabilities into one framework, such as web services offers gains such as:
  - Speed of implementation – As SOA allows developers to make extensive use of existing resources, the enterprise can build new applications quickly. This speed of implementation both saves money by reducing the amount of expensive engineering resources expended and reduces time to market, thus increasing revenue opportunity.
  - Reduced expenditure on integration technologies – The use of web services to expose core functions from enterprise applications enables the building of composite applications from existing web services.

Challenges of IT Integration

Supply-chain management offers a classic example of the general problems associated with information technology (IT) integration, as follows:

- A chain of producer-consumer relationships can include a dozen or more companies.
- Each company might have its own unique system of managing information about inventory, orders pending, manufacturing processes, and so on.

- Interaction with one's vendors or clients can drain a great deal of staff time as workers manually translate and transcribe information to pass it over these boundaries.

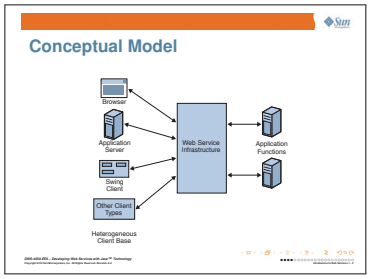
The traditional solutions used to minimize the problems with integrating disparate IT systems are standard message formats and distributed object computing (DOC) frameworks. Each has specific limitations:

- Standard message formats, such as EDI, are strongly typed, prescribing specific message formats for specific types of communication. They are not easily adapted to new domains.
- DOC systems are more flexible to different application designs, but the frameworks themselves are expansive and exclusive.

Neither approach offers the sort of adaptability that business increasingly demands. Each solution is expensive to build. Web services frameworks address these concerns.

Conceptual Model

Conceptually, a web service provides an alternative way of exposing application logic to a heterogeneous client base, as shown in Figure 1.2. Because web services are standards-based, clients running on disparate systems can programmatically access application functionality exposed using web service technology in a standard way, independent of the service implementation details or runtime environment.



Web services enable clients and services to communicate regardless of the object model, programming language, or runtime environment used on either side of the communication link. This is one of the main properties of web service technology that differentiates it from other remote access mechanisms, such as Remote Method Invocation (RMI) and CORBA. The concept of a web service as an alternative way of exposing application functions over the Internet is the web service model described in this course.

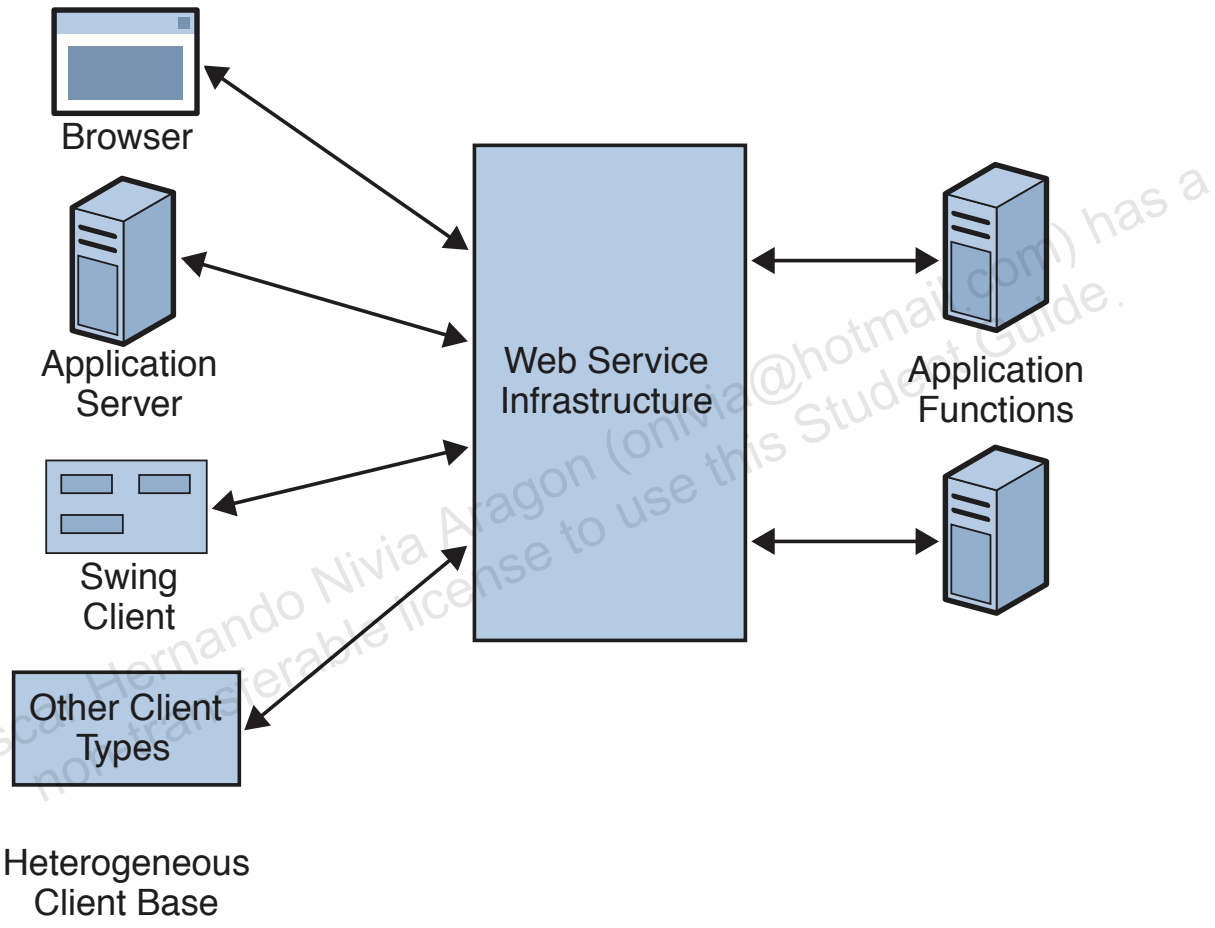


Figure 1.2: Conceptual Model

Exposing Application Functions as a Web Service

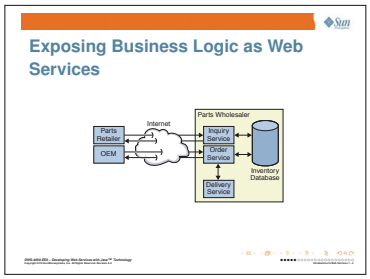


Figure 1.3 shows how a corporation might expose a set of application functions over the Internet as a web service. In this example, a parts wholesaler exposes its inventory system to potential customers, such as parts retailers or an original equipment manufacturer (OEM), as a web service. A customer might use the service to check on the availability and pricing of parts, place an order for the required components, and arrange for delivery of the needed parts to coincide with the anticipated production schedule.

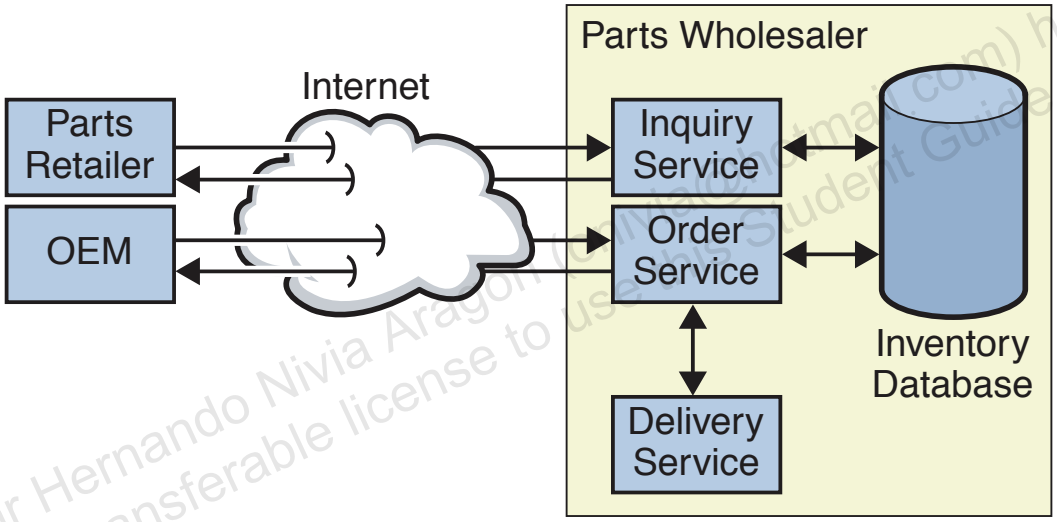
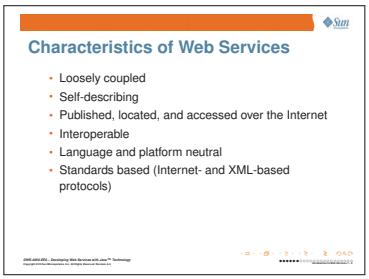


Figure 1.3: Exposing Business Logic as Web Services

Properties of a Web Service

Web services have several general properties:



- Distributed, loosely coupled, self-contained units of functionality
- Self-describing
- Published, located, and accessed over the Internet
- Interoperable
- Language and platform neutral
- Reliant on standard Internet and XML-based protocols, such as HTTP for communication or SOAP for data transfer

A web service delivers functionality previously associated with DOC standards, such as COM, CORBA, and EJB, by leveraging the lightweight infrastructure used by web applications.

- Unlike a CORBA, COM, or EJB implementation, a web service can be invoked by lightweight protocols, such as HTTP.Some of these protocolos, again such as HTTP, are also more likely to pass through firewalls.
- Unlike a web application, a web service can provide an API for programmatic use by other applications and software components, and is not presentation-focused or user-driven.

Comparison With Other Remote Component Access Mechanisms

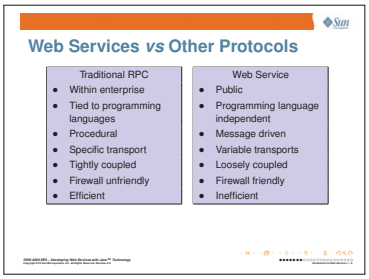


Table 1.1 shows how a web service distinguishes itself from other remote component access mechanisms, such as CORBA, RMI, and RPC over Internet Inter-ORB Protocol (RPC-IIOP), in several important ways.

Because a web service uses standard protocols for communication and data transfer, it is platform and implementation independent. You interact with a web service in a standard way that is independent of the technology used to implement the service, such as the underlying protocols, component models, application programming interfaces, programming language, or operating system. Web service clients do not need to reside on the same platform or operating system

Traditional RPC	Web Service
• Within enterprise	• Public
• Tied to programming languages	• Programming language independent
• Procedural	• Message driven
• Specific transport	• Variable transports
• Tightly coupled	• Loosely coupled
• Firewall unfriendly	• Firewall friendly
• Efficient	• Inefficient

Table 1.1: Web Services vs Other Protocols

as the service they are accessing. In fact, web service clients usually have no knowledge about the implementation and deployment details of a web service beyond those described by the service interface.

Web services are designed to facilitate enterprise-to-enterprise communication. They use transport protocols and message formats that can easily pass through a firewall.

A web service can be a loosely coupled, self-describing technology. A service definition file can describe a web service. A service registry listing can contain such a description, and everything else a service requester needs to know about a web service. Web service requesters can search a service registry for services by using a query mechanism. Service requesters can retrieve the information required to create a web service client and access a service instance from the service's description.

Benefits of Web Services

Web services enable business entities to perform business transactions over the Internet, while minimizing both the investment in IT infrastructure and the amount of pre-arrangement between communicating partners. Web services facilitate direct interactions between software applications and provide seamless interoperability across heterogeneous platforms. Application functions exposed as a web service can be discovered and accessed, programmatically, in a platform and system neutral way, using standard communication protocols and data formats.

A well-designed suite of application and system functions exposed as web services can also streamline application development time. A web service can expose both system and application functions. You can group a collection of system and application functionality from a suite of reusable services into useful applications.

Web services also provide an ideal mechanism for integrating legacy systems running in dissimilar environments in an implementation-neutral way.

Web Services in B2B and B2C Scenarios

Web services are regarded as the best solution for B2B communication for the following reasons:

- HTTP/XML-based software is quicker and cheaper to implement than Distributed Component Object Model (DCOM) or CORBA systems.
- Web services are easily developed as wrappers around existing internal systems, where those exist.
- Web services offer much better interoperability with other businesses' software than do any of the DOC standards.
- Web services that use existing web protocols, such as HTTP, can pass through most corporate firewalls, where DOC protocols are more probably blocked.

Although a web service might not directly address B2C needs, it can simplify the integration and consolidation of B2B and B2C solutions.

Service-Oriented Architecture (SOA) and Web Services

SOA is an architectural style for building software applications that uses services available in a network, such as the web. It promotes loose coupling between software components so that they can be reused, and promotes the creation of services as assemblies of other services. Applications in SOA are built based on services. A service is an implementation of a well-defined business functionality, and such services can then be consumed by clients in different applications or business processes.

SOA allows for the reuse of existing assets where new services can be created from an existing IT infrastructure of systems. In other words, it enables businesses to leverage existing investments by allowing them to reuse existing applications, and promises interoperability between heterogeneous applications and technologies.

SOA-based applications are distributed multitier applications that have presentation, business logic, and persistence layers. Services are the building blocks of SOA applications.

SOA architectures can be realized through web services. However, Web services specifications can add to the confusion of how to best use SOA to solve business problems. For a smooth transition to SOA, managers and developers in organizations should know that:

- SOA is an architectural style that has been around for years. Web services are the preferred way to realize SOA.
- SOA is more than just deploying software. Organizations need to analyze their design techniques and development methodology and partner/customer/supplier relationship.
- Moving to SOA should be done incrementally. This requires a shift in how you compose service-based applications while maximizing existing IT investments.

Limitations of Web Services

Web services are not suitable for all applications. There are costs associated with using a web service. Some potential costs are outlined as follows:

- Increased central processing unit (CPU) utilization – Parsing and processing the XML-based messages at the client and service takes time, in addition to what might be required by other types of remote method invocation.
- Increased network bandwidth – XML-based messages that contain the information required to invoke a service can be larger than passing serialized parameter values.
- Increased memory requirements – Storing the object model generated from an XML document might require additional memory.
- Emerging service support – Development of a service structure for web services, such as support for security and transaction management, is still evolving and might not be sufficient for the application.

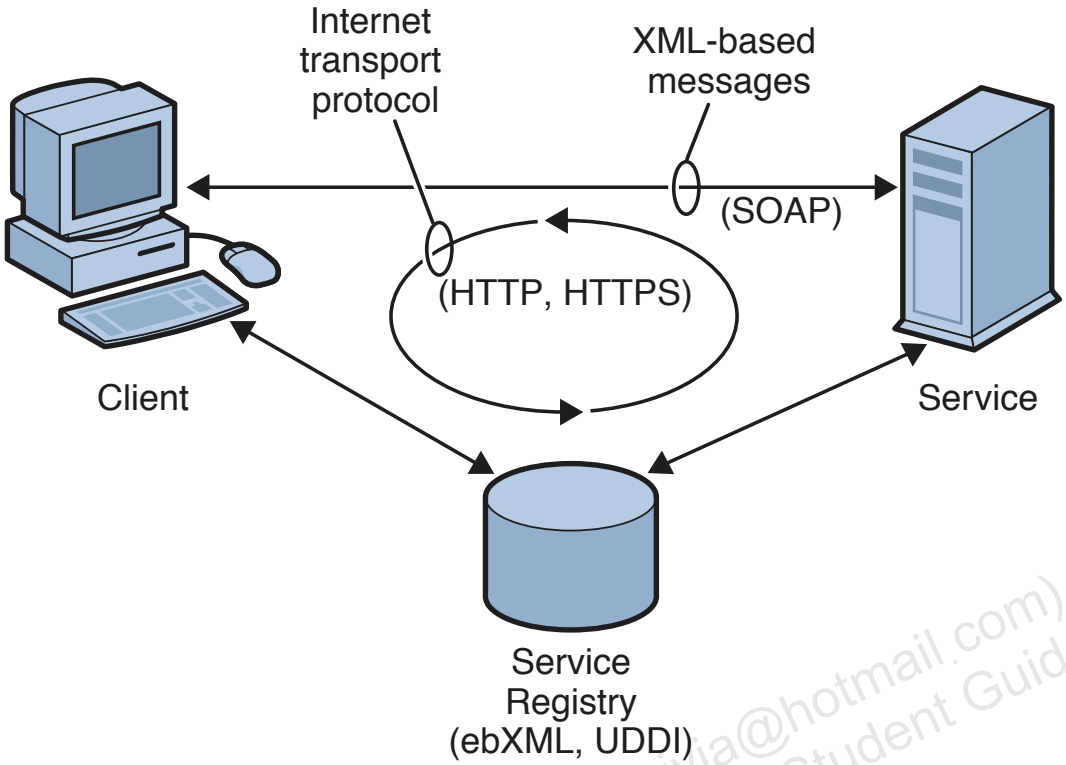


Figure 1.4: Web Service Elements

Characteristics of a Web Service

The following sections describe the elements of a web service, the roles involved in developing and maintaining web services, the life cycle of a web service, and the web service protocol layers.

Web Service Elements

The five basic web service elements are: service, service registry, service client, a transport protocol, and messages. Figure 1.4 shows the basic elements of a web service.

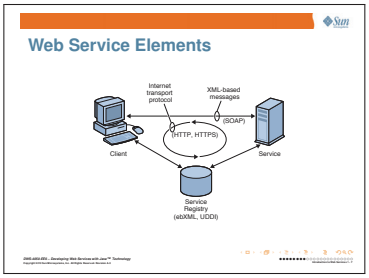


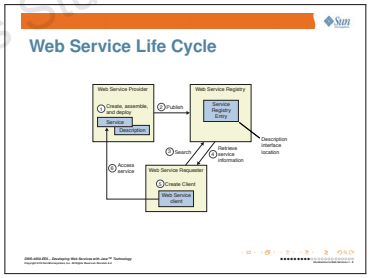
Table 1.2 describes the five basic elements of a web service. These five elements together define the web service model.

Element	Description
Service	A deployed and accessible business process or application function
Service registry	A place where you can find information about available services (not required if all serviceproviders are known to the clients ahead of time)
Service client	A client that accesses a service
messages	Information transmitted between the client and service required to access a service and return the processing results. XML is a common form of representation for messages
Internet transport protocol	The Internet communication protocol, such as HTTP or Hypertext Transport Protocol Secure (HTTPS), used between the client and service.

Table 1.2: Basic Web Service Elements

Web Service Life Cycle

Figure 1.5 shows the life cycle of a web service.



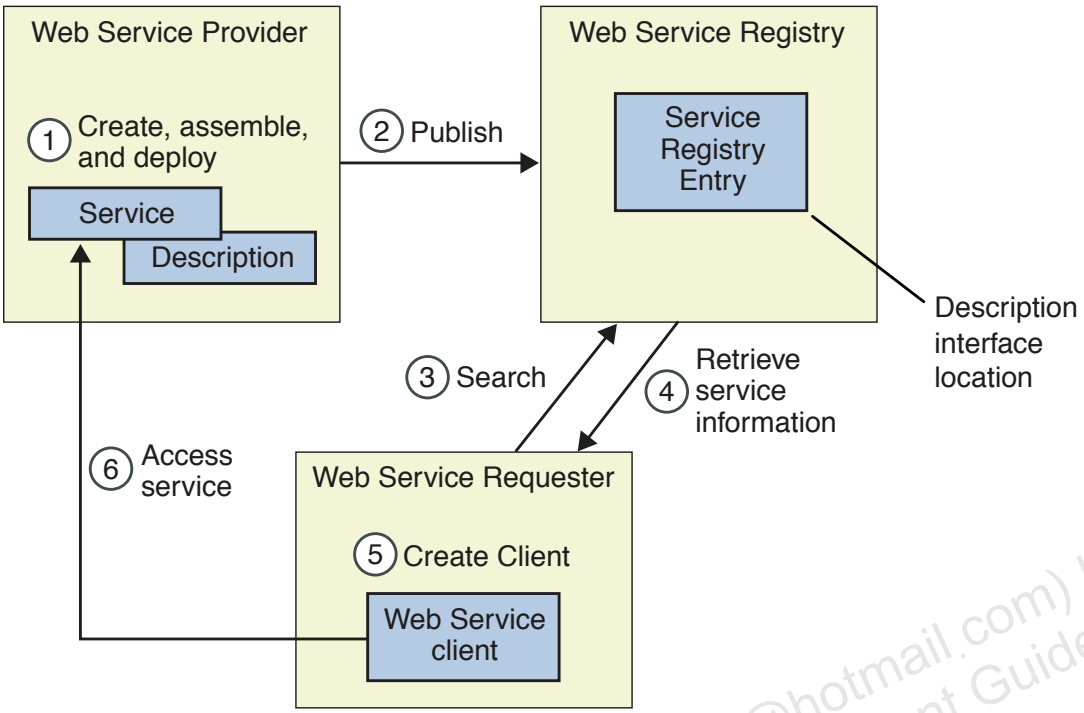


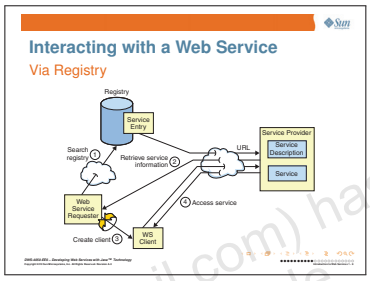
Figure 1.5: Web Service Life Cycle

A web service has the following life-cycle events:

1. A web service provider creates and deploys a web service.
2. A web service consists of a service or application function and the web service infrastructure that make it accessible to remote clients.
3. The web service provider then publishes information about a web service in a service registry - or makes it available to clients directly.
4. Published information might include a description of the service, the service interface definition, and the service instance location. A service provider can also associate a service with a variety of taxonomies or classification schemes to facilitate locating services that satisfy a particular criteria. Alternatively, the provider can place the service information in an accessible location and provide instructions for accessing the service directly to any potential requesters.
5. Web service requesters search a service registry - or some well-known location - for services that meet their needs. If using a service registry, the requester could use a query mechanism provided by a registry browser.
6. After locating an appropriate service, a service requester retrieves information about the service, such as the service location and interface description.

7. The web service requester creates a web service client.
8. A web service client invokes the web service at the location specified in the service description, passing the required parameters in an XML-based message. The service executes using the parameters contained in the incoming message, and returns the processing results to the client in an XML-based response message. Both the client and the service components contain logic to generate and process the XML-based messages.

Figure 1.6 shows how to access a service description from a service registry.



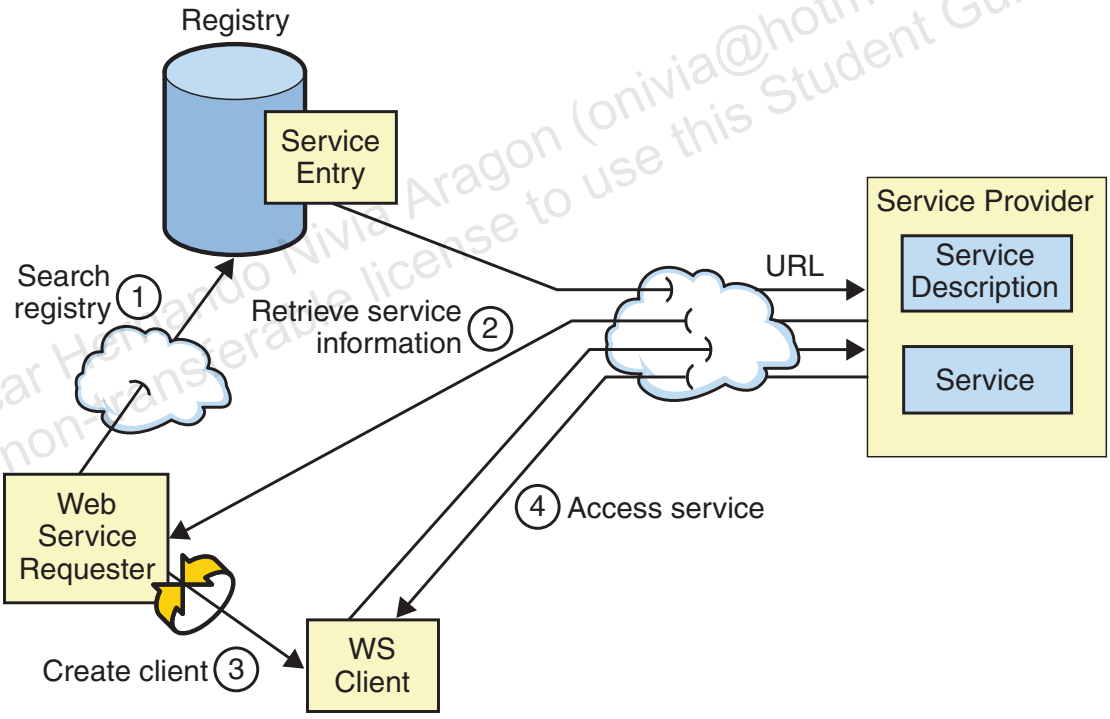


Figure 1.6: Interacting with a Web Service via Registry

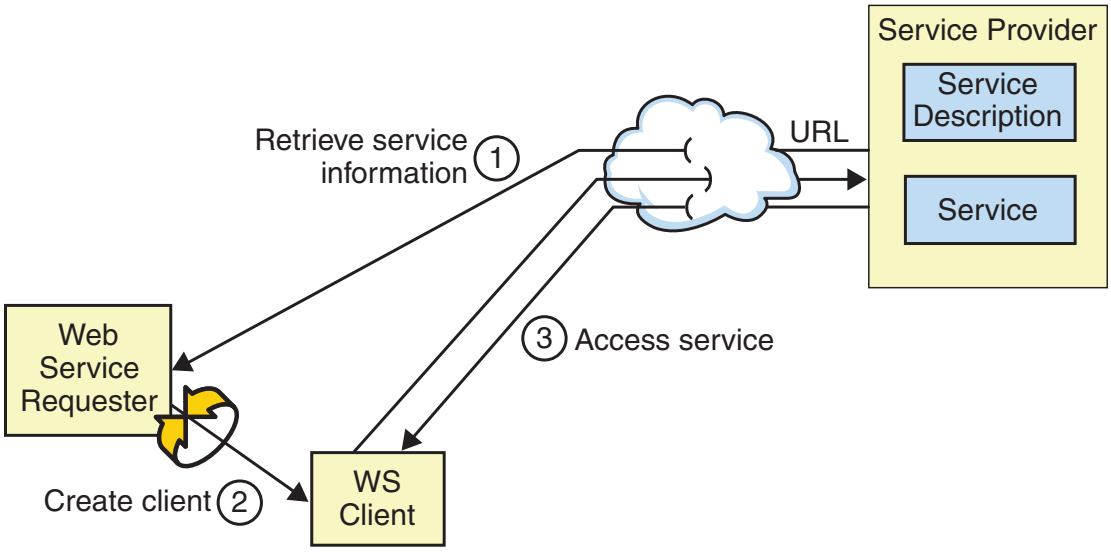
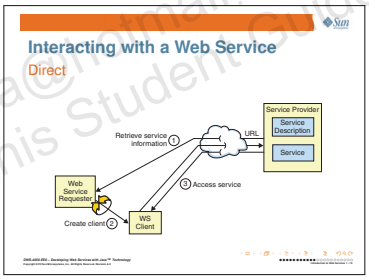


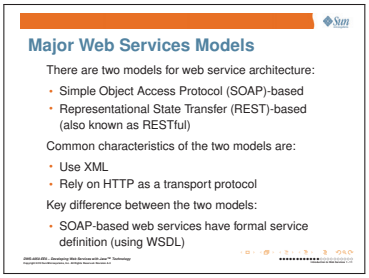
Figure 1.7: Interacting with a Web Service – Direct

Figure 1.7 shows the second method. A web service requester retrieves the web service description directly from the service provider



Major Web Services Models

The two major models for web service architecture include the following:



- SOAP-based web services, which rely on the Simple Object Access Protocol as the primary messaging language. SOAP web services are dominant in the emerging market and are standardized for some purposes by the WS-I Basic Profile.
- XML-based web services, also called REpresentational State Transfer (REST) services, which were established and in use before the term web service was coined.

Both models share the following characteristics:

- They can both use XML.
- They both typically rely on HTTP as a transport protocol.

Role of HTTP in Web Services

HTTP has the following characteristics:

- HTTP is a means by which HTML pages and other resources are requested and retrieved on the web.
- HTTP is a text-based protocol most typically carried over Transmission Control Protocol/Internet Protocol (TCP/IP) that consists of a numbered message, a message header, and an optional message body.
- It is extensible by means of the HTTP header system.
- While limited semantically, HTTP is ubiquitous and also (almost) universally trusted to enter and leave a corporate or other security domain by passing through its firewall.

Although web services are not explicitly founded on HTTP, HTTP is the dominant transport protocol for web services. However, SOAP or other XML messages can travel by other means.

Role of XML in Web Services

XML has the following characteristics:

- XML is a text-based language that allows data of arbitrary type, size, and complexity to be expressed and described in embedded markup.
- XML is somewhat similar in grammar and style to HTML, but is more precise in its descriptive capabilities. XML is better suited for data rendition than graphical presentation.
- XML allows information that might otherwise be stored and transmitted in binary forms, such as relational databases and specific wire formats, to be encoded entirely in text, without loss of data, precision, or metadata. XML's ability to encode complex data structures in pure text makes it suitable for web services which need to pass structures over web protocols such as HTTP.

All web services can use XML. REST services can use XML directly - although they often use JSON instead, due to its simplicity. SOAP is an XML vocabulary used in more complex web services frameworks, such as JAX-WS (the Java API for XML-based Web Services).

XML schemas are a closely related W3C recommendation, but is not required in a web service implementation. XML schemas, like HTTP, are a standard, but not a formal requirement.

Web Service Initiatives, Standards, and Specifications

Several specifications, initiatives, and application programming interfaces are emerging surrounding the developing web service model.

Although the integration of web services into an application framework is evolving as the associated standards and programming models develop and mature, a basic model is emerging.

The current model uses HTTP/HTTPS as the Internet transport protocol. The current model uses a set of standard XML-based protocols to publish and describe a web service, and to define the message and data components of a web service invocation and subsequent response.

Governing Bodies

Several governing bodies and interested groups are participating in the development of web services standards and APIs. The main groups include:

- Organization for the Advancement of Structured Information Systems (OASIS) – OASIS is a not-for-profit, global consortium that drives the development, convergence, and adoption of e-business standards.
- United Nations Centre for the Facilitation of Procedures and Practices for Administration, Commerce and Transport (UN/CEFACT) – This United Nations body establishes worldwide policy and technical development in the area of trade facilitation and electronic business.
- World Wide Web Consortium (W3C) – W3C was founded in October 1994 to lead the World Wide Web to its full potential by developing common protocols that promote its evolution and ensure its interoperability.
- Java Community Process^{TM}(JCP^{TM}) – The JCP is an open organization of international Java developers and licensees whose charter is to develop and revise Java technology specifications, reference implementations, and technology compatibility kits.
- Web Services Interoperability Organization (WS-I) – An open industry effort that promotes web services interoperability across platforms, applications, and programming languages. The WSI is actively involved in:
  - Defining a set of specifications that describe profiles for web service functionality
  - Producing guidelines for ensuring interoperability among web service security mechanisms

  - Developing sample applications that illustrate typical web service usage scenarios
  - Implementing a suite of testing tools for validating the interoperability of a web service

Web Service Initiatives

The primary web service initiatives include the following:

- Sun Java™System
  Sun Java System is Sun's standards-based software vision, architecture, and platform for building and deploying services on demand. It provides a scalable and robust foundation for traditional software applications, as well as current web-based applications, while laying the foundation for the distributed computing models, such as web services.
- .NET
  This Microsoft initiative provides an application development framework, and a set of supporting tools for creating applications and web services.
- Electronic Business using eXtensible Markup Language (ebXML)
  ebXML, sponsored by UN/CEFACT and OASIS, is a modular suite of specifications that enables enterprises of any size and in any geographical location to conduct business over the Internet.

Web Service Specifications and APIs

Table 1.3 lists the specifications and APIs used by the web service model discussed in this course.

The SOAP Specification

Most dominant web service architectures use SOAP as a message exchange format.

- SOAP was initially developed by Microsoft, IBM, and others. The W3C currently develops SOAP.
  - Version 1.1 is in most common use and is a W3C note.
  - Version 1.2 is a W3C-defined specification.

Specification or API	Description
XML	XML is a markup language developed by the W3C. The message and data formats used in the web service model are XML-based.
SOAP	SOAP defines an XML-based structure for passing information, such as messages and data, between a web service and a service client.
Web Service Description Language (WSDL)	Developed by Ariba, IBM, and Microsoft, and submitted to the W3C, WSDL is an XML-based standard for describing the external interface to a web service.
Registry	The Universal Description, Discovery, and Integration (UDDI) and Electronic Business XML (ebXML) registry/repository specification defines the contract for registry functionality and provides APIs that are used to publish and locate information about web services in a registry.

Table 1.3: Web Service Specifications and APIs

  - A separate W3C note defines an enhanced protocol known as SOAP With Attachments. This allows some message information to be transmitted as MIME parts.

- SOAP is a means for encoding remote procedure calls (RPCs) and document-style information as XML.
- SOAP messages can travel where traditional RPCs, in various frameworks, cannot.
  - Any text-bearing protocol can carry SOAP messages.
  - In particular, and as the protocol name implies, SOAP messages can pass through firewalls as HTTP payloads.

The Web Services Description Language

Armed with a WSDL document describing a service, you can implement clients or services with confidence that other participants will operate correctly, understanding the messages that are exchanged. WSDL provides basic metadata about web services as follows:

- WSDL 1.1 is a W3C note. The Web Service Description Working Group is responsible for future versions of WSDL.
- WSDL is an XML vocabulary, a specific XML-based language, with its own standard XML Schema.

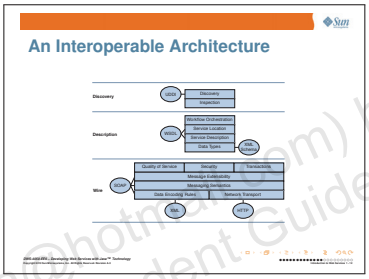
- WSDL allows services to be defined in terms of ports, each of which is of a specific port type. Ports then support a list of operations, which, in turn, map to message types for input and output.
  - Operations are similar to methods in a programming language, with parameters for invocation, and return types.
  - Operations also define basic message flow according to one of four patterns, the most common of which is request-response.

Web Services Interoperability

Other remote communication technologies did not proliferate because disparate systems cannot interact seamlessly. Those other frameworks were not independent of the object model, programming language, or runtime environment used on either side of the communication link.

An Interoperable Architecture

Web services are intended to operate between different implementation languages and platforms. The web services interoperability stack, shown in Figure 1.8, is defined in ascending order of technical sophistication: wire, description, and discovery.



The industry is addressing interoperability based on the requirements at each level of the web services stack. The WS-I Consortium has defined a number of Web Services Profiles, to address interoperability concerns at the different levels of this stack.Working through the stack, the industry has agreed to solve the interoperability problem in the following way:

- The wire stack defines how to pass messages. Message passing relies on SOAP, which is an XML vocabulary, passing over HTTP.
- WSDL manages the description stack for defining and consuming metadata about services. WSDL, in turn, aggregates the XML schema. The ability to define type models and messaging scenarios allows you to regard web services as distributed objects with interfaces.
- The discovery stack facilitates locating services published in some form of the service registry. The web service model uses the UDDI registry model, the ebXML model, and the BizTalk model.

Other standards include:

- WS-Security, WS-Transactions, and WS-ReliableMessaging – These standards correspond to the three wire-stack features.
- ebXML – The ebXML standard defines a more ambitious, high-level framework for discovery, inspection, and workflow orchestration, as do several competing products and standards.

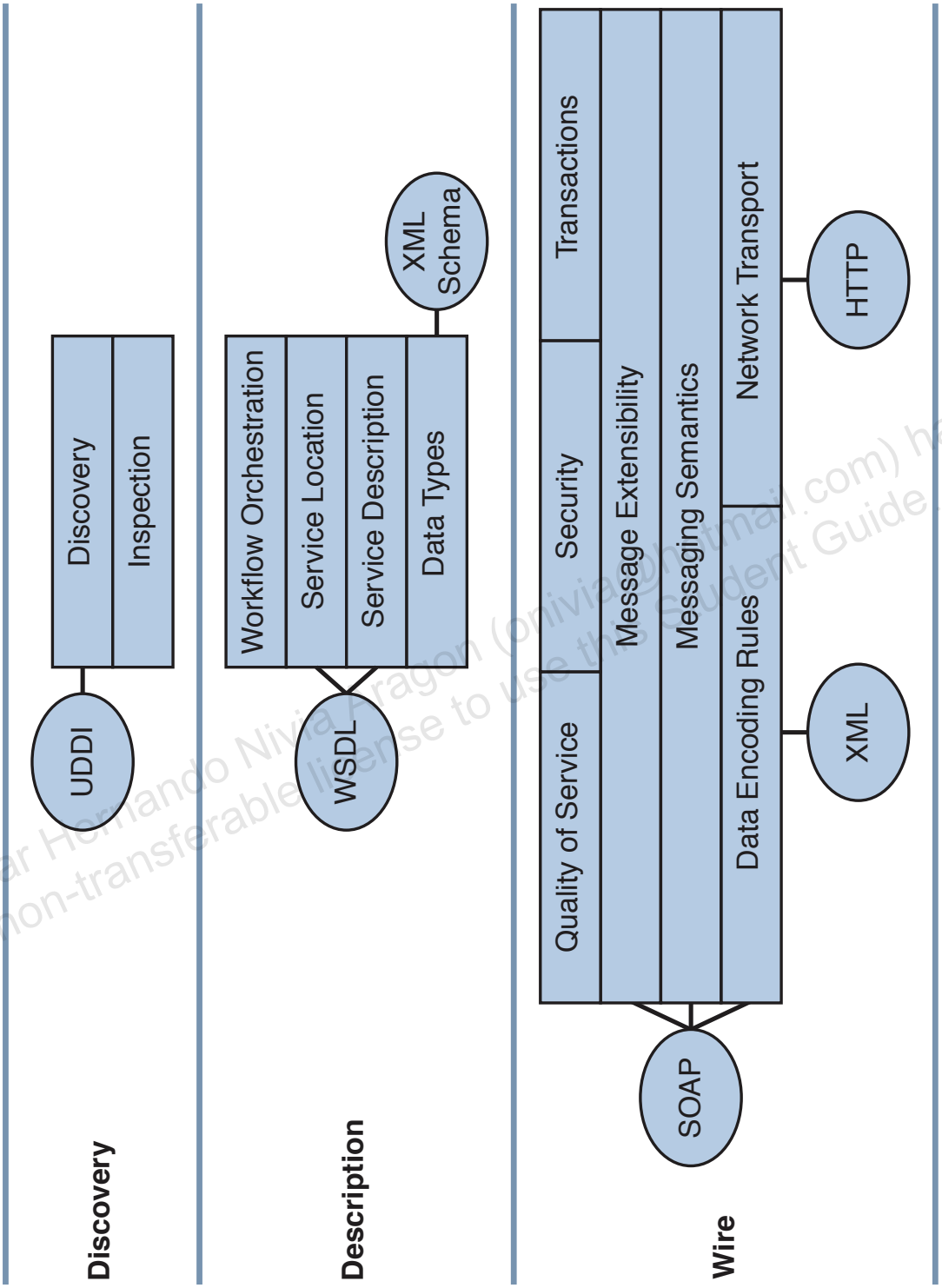


Figure 1.8: An Interoperable Architecture

The WS-I Organization

WS-I is an open industry organization chartered to promote web services interoperability across platforms, operating systems, and programming languages. The organization's diverse community of web service leaders helps customers to develop interoperable web services by providing guidance, recommended practices, and supporting resources. All companies interested in promoting web service interoperability are encouraged to join the effort.

Specifically, WS-I creates, promotes, and supports generic protocols for the interoperable exchange of messages between web services.

Driven by the ambiguities that exist in the web service standards, the WS-I defined a set of common web service parameters that, if followed by all web service implementations, improve interoperability. The WS-I has gathered major corporate and non-commercial participants to agree upon overarching specifications for web services as follows:

- The WS-I does not govern individual languages and protocols, such as SOAP, WSDL, and UDDI.
- The WS-I regards the W3C as the ultimate source for specifications.
- The WS-I focuses on profiles. Profiles are defined as follows:
  - A profile combines certain versions of useful standards.
  - A profile also sets specific rules that address either weaknesses in component specifications or issues of how to combine the collected technology.
  - A tool, application, or set of functional requirements that can be tagged as compliant with a specific WS-I profile.

The WS-I consortium has defined the following profiles:

- Basic Profile
- Attachments Profile
- Simple SOAP Binding Profile
- Basic Security Profile

The most important product of the WS-I is the Basic Profile, which at version 1.1 requires support for the following:

- SOAP 1.1 (over HTTP 1.1)

- WSDL 1.1
- UDDI 2.03
- HTTP over TLS 1.0 or SSL 3.0 (not mandatory)

The WS-I has done additional work regarding web services security and the use of attachments as part of a web service interaction.

WS-I Basic Profile Support

Both Microsoft .NET and Java EE technology implement the Basic Profile, as do most commercial Java tools. Not every service created with a Java EE technology-compliant tool is also compliant with WS-I Basic. However, you can build WS-I-Basic-compliant services.

The Basic Profile makes a number of implicit assumptions about the style and purpose of the compliant service as follows:

- It is essentially a rigorous profile for document-style services.
- The Basic Profile does not explicitly disallow RPC-style services, but it forbids a number of behaviors common to the current RPC style.
- A lack of solid consensus about rules for RPC-style messages has kept the Basic Profile from embracing them fully.
- RPC-style services are much more ambitious. Because RPC-style services are meant to hide a great deal of inter-object wiring from the SOAP/WSDL programmer, they are much more ambitious in scope than document-style services.

A WSDL document that describes a web service also defines how a web service is bound to the SOAP protocol. The WSDL SOAP binding can be of RPC style or document style. The binding style determines how to transform a WSDL binding to a SOAP message. The two binding styles are covered in Module 6.

Even under WS-I, a component does not have to observe any specific contract to be considered a real web service. Instead, a service can implement ascending levels of sophistication, or stacks. Even now, the WS-I makes no requirements. Instead, WS-I specifies rules to make a specific function interoperable.

WS-I Basic Profile Web Services

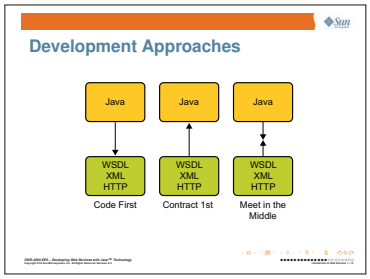
The WS-I Basic Profile specification is a document in which a majority of the specification is targeted at the development tool vendors who are working on vendor-specific implementations of SOAP processors, WSDL parsers, code generators, and so on. The Profile is an effort by those tools and platform vendors to ensure that their respective products will either generate or host interoperable web services instances.

Web service developers and practitioners can follow some of the following best practices:

- Use the available testing tools to ensure that a web service adheres to a profile. A web service adhering to the Basic Profile is interoperable with other services adhering to that Profile.
- Define data types early in the integration cycle.
- Test interoperability of data at every stage of development.
- Keep data types simple for speed and stability.
- Use Message Transmission Optimization Mechanism (MTOM) for transmitting attachments with SOAP messages.

Development Approaches

Web services development has evolved three major approaches to the design an implementation of web services, as illustrated in Figure 1.9:



- code-first, or bottom-up
- contract-first, or top-down
- meet-in-the-middle.

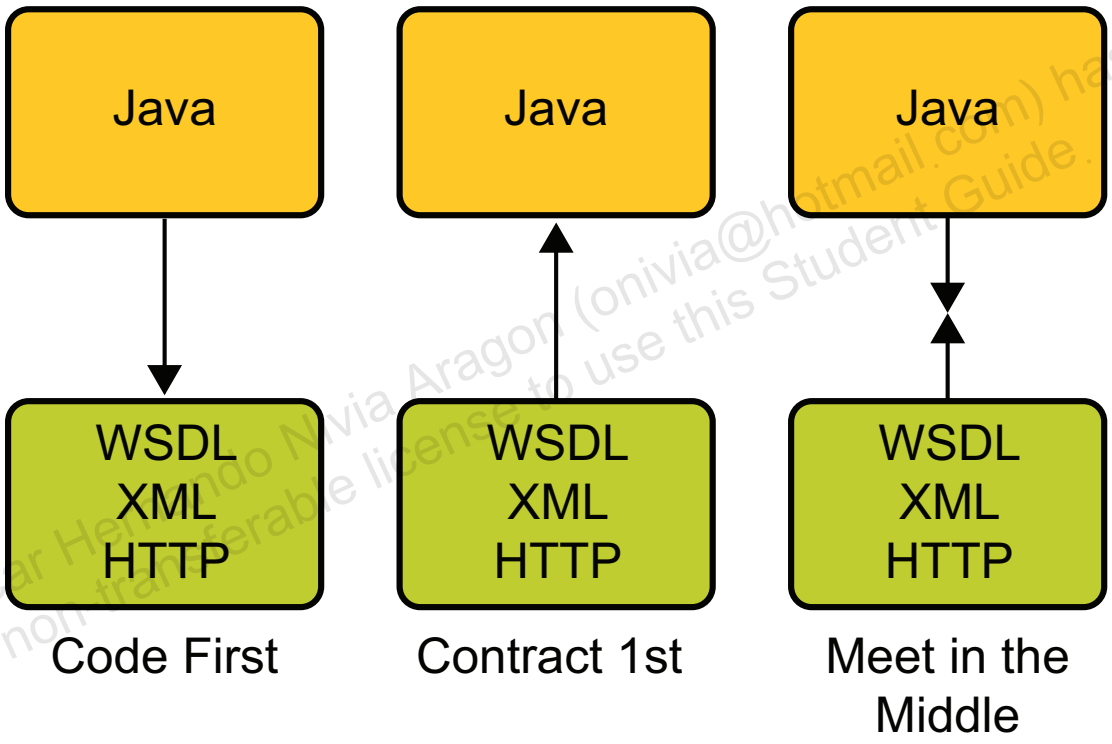
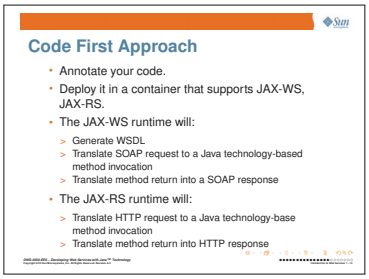


Figure 1.9: Development Approaches

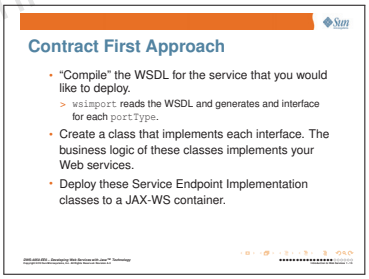
Code First Approach

- Annotate your code.
- Deploy it in a container that supports JAX-WS, JAX-RS.
- The JAX-WS runtime will:
  - Generate WSDL
  - Translate SOAP request to a Java technology-based method invocation
  - Translate method return into a SOAP response
- The JAX-RS runtime will:
  - Translate HTTP request to a Java technology-base method invocation
  - Translate method return into HTTP response



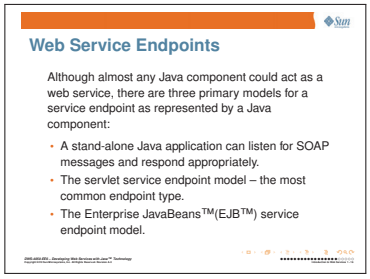
Contract First Approach

- "Compile" the WSDL for the service that you would like to deploy.
  - `wsimport` reads the WSDL and generates and interface for each `portType`.
- Create a class that implements each interface. The business logic of these classes implements your Web services.
- Deploy these Service Endpoint Implementation classes to a JAX-WS container.



Web Service Endpoints

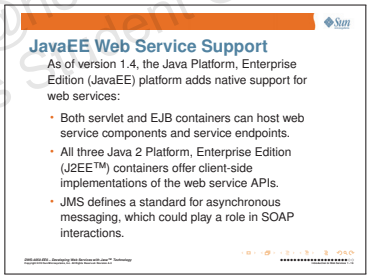
Although almost any Java component could act as a web service, there are three primary models for a service endpoint as represented by a Java component:



- A stand-alone Java application can listen for SOAP messages and respond appropriately.
- The servlet service endpoint model – the most common endpoint type.
- The Enterprise JavaBeans™(EJB™) service endpoint model.

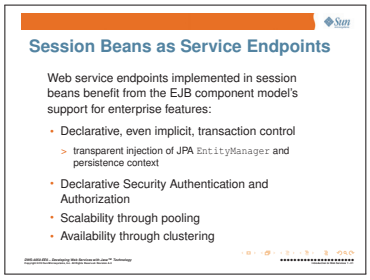
JavaEE Web Service Support

As of version 1.4, the Java™ Platform, Enterprise Edition (JavaEE) platform adds native support for web services:



- Both servlet and EJB containers can host web service components and service endpoints.
- All three Java 2 Platform, Enterprise Edition (J2EE™) containers offer client-side implementations of the web service APIs.
- JMS defines a standard for asynchronous messaging, which could play a role in SOAP interactions.

Web service endpoints implemented in session beans benefit from the EJB component model's support for enterprise features:



- Declarative, even implicit, transaction control
  - transparent injection of JPA `EntityManager` and persistence context
- Declarative Security Authentication and Authorization
- Scalability through pooling
- Availability through clustering

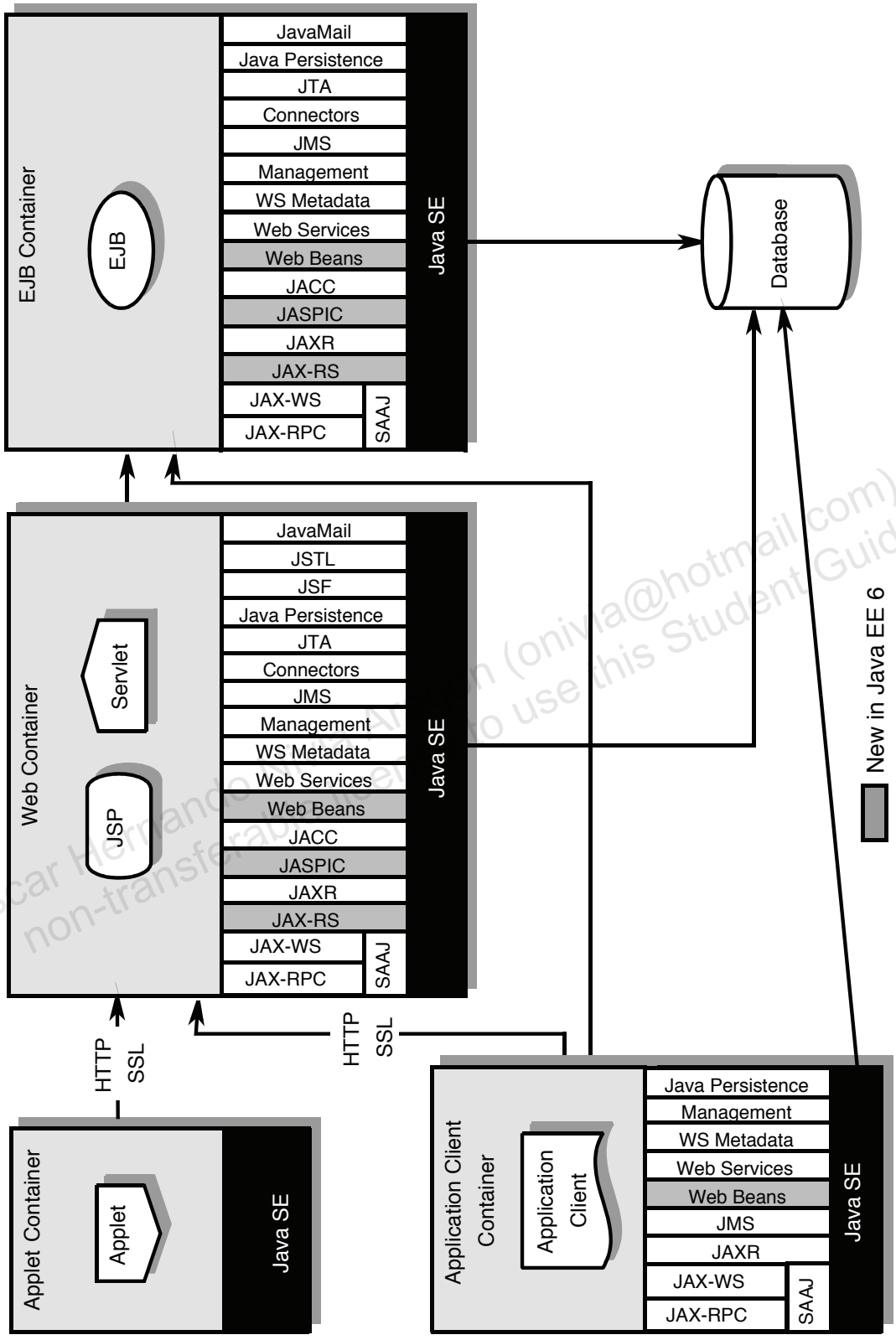


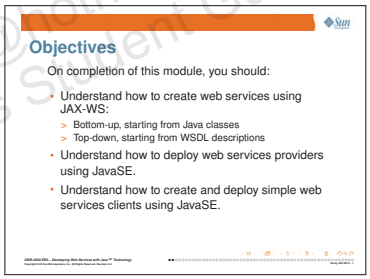
Figure 1.10: JavaEE 6 APIs

Chapter 2

Using JAX-WS

On completion of this module, you should:

- Understand how to create web services using JAX-WS:
  - Bottom-up, starting from Java classes
  - Top-down, starting from WSDL descriptions
- Understand how to deploy web services providers using JavaSE.
- Understand how to create and deploy simple web services clients using JavaSE.

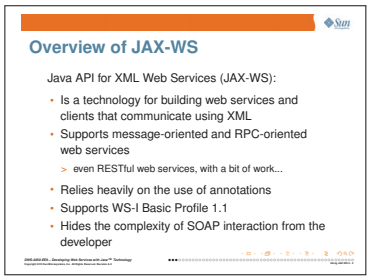


Additional Resources

The following references provide additional information on the topics described in this module:

- JAX-WS Reference Implementation Project home page at java.net
  `http://jax-ws.dev.java.net`
- JAX-WS articles,
  `http://wiki.java.net/bin/view/Javawsxml/JaxwsArticles`
  as of September 2009.
- JAX-WS Users's Guide,
  `https://jax-ws.dev.java.net/jax-ws-201-m1/docs/UsersGuide.html` as of September 2006.
- JAX-WS Annotations
  `http://java.sun.com/webservices/docs/2.0/jaxws/annotations.html` as of November 2005.
- GlassFish Project - Web Services Deployment and Dispatching Home Page,
  `https://glassfish.dev.java.net/javaee5/webservices/dispatch_process.html`

Overview of JAX-WS



JAX-WS is a technology for building web services and clients that communicate using XML. JAX-WS allows you to write document-oriented, as well as RPC-oriented web services. REST-based web services can also be developed using JAX-WS.

JAX-WS is the aggregating component of what is called the integrated Stack (I-Stack). The I-Stack consists of JAX-WS, JAXB, StAX, SAAJ, and Fast Infoset. JAXB is the data-binding component of the stack. StAX is the Streaming XML parser used by the stack. SAAJ is used for its attachment support with SOAP messages to allow handler developers to gain access to the SOAP message using a standard interface. Fast Infoset is a binary encoding of XML infosets and is an efficient alternative to XML.

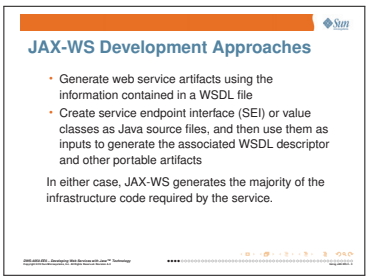
The JAX-WS API hides the complexity of SOAP messages from the application developer.

On the server side, you can specify a Service Endpoint Interface (SEI) by defining methods in an interface written in the Java programming language. You must write a web service implementation class with the desired methods and the required annotations. The Java interface is optional because the web service implementation class implicitly defines an SEI. On the client side, the client code creates a JAX-WS generated proxy (a local object representing the service) and then invokes methods on the proxy.

With JAX-WS, you do not generate or parse SOAP messages. It is the JAX-WS runtime system that converts the API calls and responses to and from SOAP messages.

With JAX-WS, clients and web services have an advantage: the platform independence of the Java programming language. In addition, JAX-WS is not restrictive: a JAX-WS client can access a web service that is not running on the Java platform, and a non-Java client can access a JAX-WS web service. This flexibility is possible because JAX-WS supports WS-I Basic Profile 1.1 (HTTP, SOAP, and WSDL).

You can take one of two different development approaches when you create web services using JAX-WS. You select the approach based on the source language used to define the service semantics. You can:

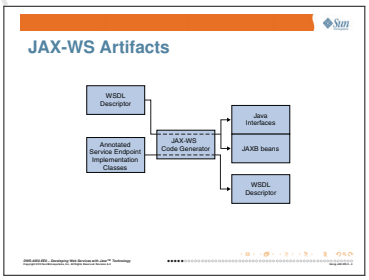


- Start from WSDL – Generate portable web service artifacts, such as a Java service endpoint interface (SEI) and serializable value type classes using the information contained in a WSDL file.
- Start from Java technology – Create a service interface and value classes as Java source files, and then use them as inputs to generate the associated WSDL descriptor.

In either case, JAX-WS generates the majority of the infrastructure code required by the service.

JAX-WS relies on the use of annotations within the implementation classes to help in marshalling/unmarshalling messages.

Figure 2.1 summarizes the key JAX-WS code generation processes, and shows the relationship between the hand-written and generated artifacts.



Regardless of the development approach used, start from Java or start from WSDL, the process is quite similar. The primary difference in the development process is the order in which tools are run:

- When using the start from WSDL approach, you generate the Java technology artifacts using the information contained in the WSDL descriptor.
- When using the start from Java approach, you create and compile the Java source files, such as the service endpoint implementation class, and or the interface which are then used to generate the additional web service artifacts.

Regardless of the approach, you must create the service endpoint implementation class.

In JAX-WS, all artifacts generated by `apt`, `wsimport`, and `wsgen` are portable. Users can create their own deployable WAR file.

The `wsgen` tool processes a compiled service endpoint implementation class and generates the portable artifacts. JAX-W's `wsgen` tool does not generate WSDL at build-time; the WSDL is generated when the endpoint is deployed. However, there is an option on `wsgen` to generate the WSDL for development purposes.

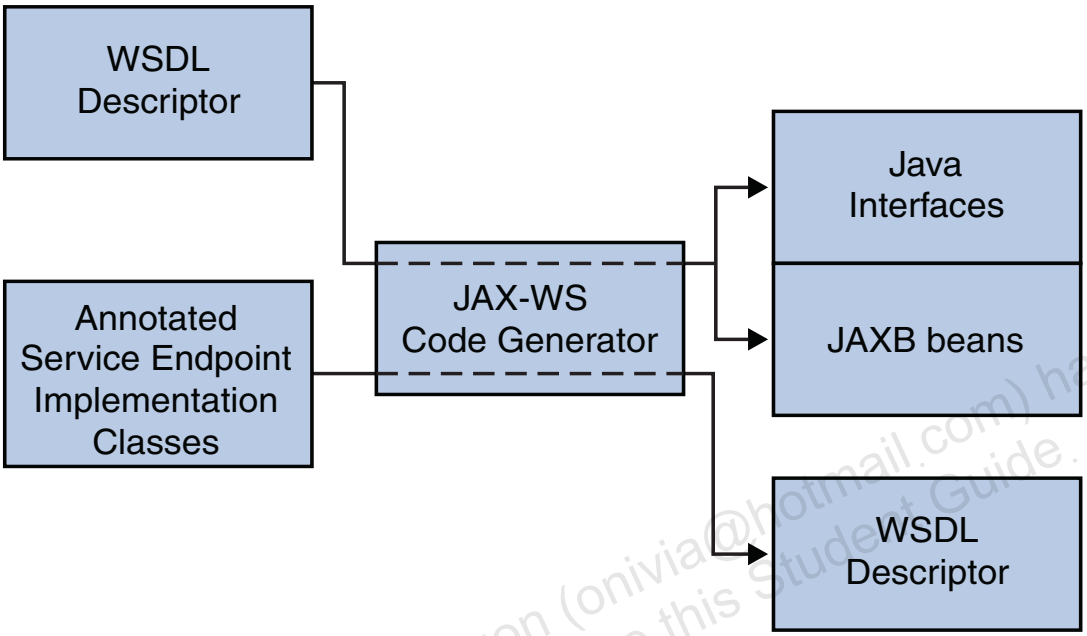


Figure 2.1: JAXWS Artifacts

Table 2.1 shows a summary of how JAX-WS translates WSDL components, including supporting XML schema types, into Java components (and vice-versa).

The mapping is essentially the same for both development approaches, WSDL-to-Java or Java-to-WSDL. The roles of various WSDL, XML schema, and Java components, and their relationships in the mapping, are consistent regardless of the development approach that you use.

The WSDL `binding` element is notably absent from the mapping elements listed in Table 2.1. JAX-WS supports the concrete model, specifically for SOAP over HTTP, by defining a number of supporting classes:

- The supporting classes are primarily responsible for translating primitive values and objects back and forth between SOAP messages and Java technology components.
- JAX-WS generates these classes regardless of the development approach, Java-to-WSDL or WSDL-to-Java.
- The supporting classes obtain some information required to perform their work from the extensibility elements under the WSDL `binding` element.

WSDL Components	Descriptions
WSDL service and port	A web application supporting one service endpoint implementation class for each port.
WSDL portType	A Java service interface with one method for each WSDL operation.
WSDL operation (including message and part)	The method signature, which includes a parameter for each part in the input message, throws an exception for each fault, and returns a primitive or single object for the output message.
XML schema simple types (from the WSDL types section)	This is delegated to JAXB, which does class-based mapping.
XML schema complex types	Java classes known as value types generated by JAXB.

Table 2.1: WSDL-to-Java Technology Component Translations

For example, the messaging style, either RPC or document, and use, either literal or encoded, have an effect on how an element is serialized.

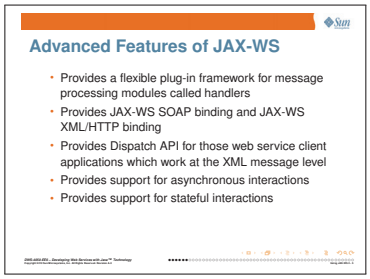
- The service runtime invokes the supporting classes, which operate independently from the service endpoint implementation that has no knowledge of their existence. The classes are discovered by mappings and loaded dynamically at runtime.
- Service APIs, client API, and core APIs for the JAX-WS-based implementation of web services and clients:
  - JAX-WS provides service APIs for service implementations:
    - `javax.xml.ws.Provider`
    - `javax.xml.ws.Endpoint`
    - `javax.xml.ws.WebServiceContext`
  - Core APIs includes the following classes that can be used both by the services and service clients:
    - `javax.xml.ws.Binding`
    - `javax.xml.ws.spi.Provider`
    - `javax.xml.ws.spi.ServiceDelegate`
    - Exceptions – `WebServiceException` , `ProtocolException` , `SOAPFaultException` , `HTTPException`
  - Client APIs are the standard APIs provided to create JAX-WS-based clients. These APIs allow a client to create proxies for remote service endpoints and dynamically construct operation invocations. These client APIs are:

```
* javax.xml.ws.Service
* javax.xml.ws.BindingProvider
* javax.xml.ws.Dispatch
```

JAX-WS takes advantage of Java annotations and aligns with the annotations defined by JSR (181). JAX-WS uses annotations extensively. One of the main difference between JAX-RPC and JAX-WS is the programming model. A JAX-WS-based service uses annotations (such as `@WebService`) to declare web service endpoints. Use of these annotations eliminates the need for deployment descriptors. With JAX-WS, you can have a web service deployed on a Java EE-compliant application server without a single deployment descriptor. Apart from these, other additional features (such as asynchronous callbacks) are also present.

JAX-WS also provides a standard facility to customize the WSDL 1.1-to-Java binding – JAX-WS 2.0 defines an XML-based language that you can use to specify customizations to the WSDL 1.1 that is mapped to Java technology, such as the service endpoint interface class, the method name, parameter name, exception class, and so on. The language is referred to as a binding language or a binding declaration. Using these binding declarations, you can also control certain features, such as asynchrony, provider, wrapper style, and additional headers. For example, a client application can enable asynchrony for a particular `portType` operation or all `portType` operations defined in the WSDL file. These binding declarations can reside in a WSDL file or can live outside as an external file. The binding declarations closely align with the JAXB binding declarations.

In addition to basic support to map the constructs of WSDL to equivalent Java types, JAX-WS provides additional functionality to help developers build sophisticated web services-based applications. Among these additional features:



- Web service endpoints can choose to work at the XML message level by implementing the `Provider` interface. This is achieved by implementing one of
  - `Provider<Source>`
  - `Provider>SOAPMessage>`
  - `Provider<DataSource>`

  The endpoint accesses the message or message payload using this low-level, generic API. All the `Provider` endpoints must have a `@WebServiceProvider` annotation. The `@ServiceMode` annotation is used to convey whether the endpoint wants to access the raw SOAP message

(use `Service.Mode.MESSAGE`) or payload
(use `Service.Mode.PAYLOAD`).

- Provides a flexible plug-in framework for message processing modules called handlers. Handlers are message interceptors that you can plug in to the JAX-WS runtime to do additional processing of the inbound and outbound messages. JAX-WS defines two types of handlers, logical handlers and protocol handlers. Protocol handlers are specific to a protocol and can access or change the protocol-specific aspects of a message. Logical handlers act only on the payload of the message and cannot change any protocol-specific parts (like headers) of a message.
- Provides JAX-WS SOAP binding and JAX-WS XML/HTTP binding. JAX-WS defines an API for programmatic configuration of client-side SOAP binding. You can configure server-side SOAP binding programmatically or by using annotations. SOAP binding is applicable while working in message handlers. JAX-WS XML/HTTP binding provides raw XML over HTTP messaging capabilities with the help of handlers. This is more relevant for REST-based web services.
- Provides a `Dispatch` API for those web service client applications which works at the XML message level. The `Dispatch` API is intended for advanced XML developers who prefer to use XML constructs at the SOAP/XML message level. Use of the `Dispatch` API with JAXB data-bound objects is supported.
- The authors of the WSDL specification discuss a number of modes of interaction between the web service client and the web service provider, and they provide mechanisms to describe which mode a particular operation of a service will use. In practice, web service implementations used to support just the synchronous request-response and synchronous one-way modes. In JAX-WS, the infrastructure adds support for two kinds of asynchronous request-response interactions: client-pull (poll-based) and server-push (notification).
- Although web services were originally defined to be stateless, JAX-WS argues that, when the semantics of the operations on a server maintain conversational state across operations, a stateless implementation has to incur additional overhead just to keep the state. For these scenarios, JAX-WS adds the ability to specify *statefull* web service endpoints.

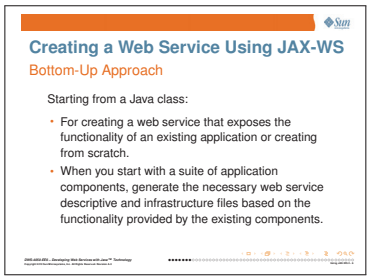
Creating a Web Service Using JAX-WS

There are two development approaches for creating a web service using JAX-WS. When developing a web service endpoint, you can either start from a Java endpoint implementation class or from a Web Services Description Language (WSDL) file. A WSDL document describes the contract between the web service endpoint and the client. A WSDL document can include or import or both XML schema files used to describe the data types used by the web service. When starting from a Java class, the tools generate any portable artifacts, as mandated by the specification. When starting from a WSDL file and schemas, the tools generate a service endpoint interface.

There is a trade-off when starting from a Java class or from a WSDL file. If you start from a Java class, you can make sure that the endpoint implementation class has the desirable Java data types, but you have less control of the generated XML schema. When starting from a WSDL file and schema, you have total control over what XML schema is used, but less control over what the generated service endpoint and the classes it uses will contain.

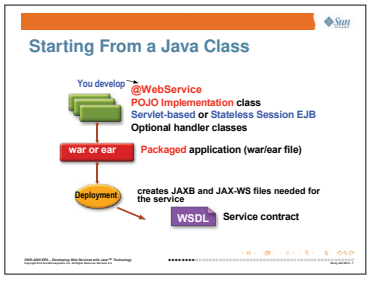
Creating a Web Service Using JAX-WS: Bottom-Up

Sometimes, it is most convenient to define a new web service by starting from a Java class:



- When creating a web service that exposes the functionality of an existing application, or creating one from scratch.
- When you start with a suite of application components, generate the necessary web service descriptive and infrastructure files based on the functionality provided by the existing components.

Figure 2.2 illustrates the steps required to create a web service provider from an existing Java class.



You must provide the JAX-WS tools with a valid endpoint implementation class. This implementation class is the class that implements the desired web service. JAX-WS has a number of restrictions on endpoint implementation classes. A valid endpoint implementation class must meet the following requirements:

- It must carry a `javax.jws.WebService` or a `javax.xml.ws.WebServiceProvider` annotation (see JSR 181).
- It can extend `java.rmi.Remote` either directly or indirectly.
- Any of its methods can carry a `javax.jws.WebMethod` annotation.
- Any method parameters can carry a `javax.jws.WebParam` annotation
- All of its methods can throw `java.rmi.RemoteException` in addition to any service-specific exceptions.
- All method parameters and return types must be compatible with the JAXB 2.0 Java to XML Schema mapping definition.
- A method parameter or return value type must not implement the RMI marker interface, `java.rmi.Remote`, either directly or indirectly.

Figure 2.1 shows a simple POJO service, which will be used to define a web service. The service is implemented by class `AirportManager`. It has a single public method, which will become the only operation on our first web service. Internally, it relies on a data access object – an instance of `AirportDAO` – to manage persistent instances of `Airport`.

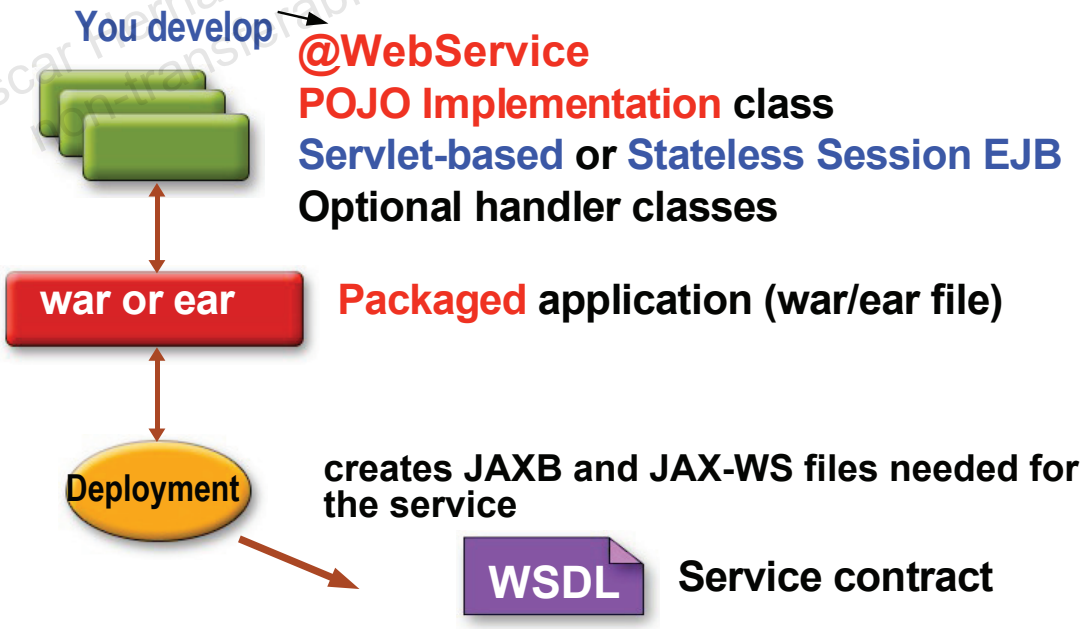
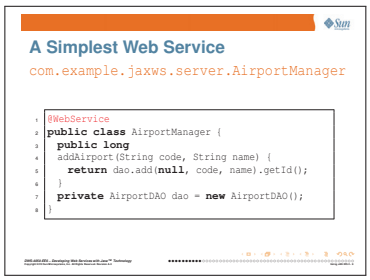
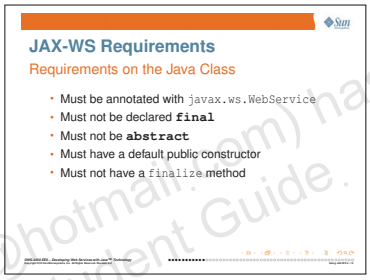


Figure 2.2: Starting From a Java Class



JAX-WS tries to make the definition and deployment of web services as convenient as possible. When configuration is necessary, JAX-WS uses "configuration by default" – JAX-WS will guess at any configuration values that it needs, if they are not provided explicitly. For this reason, it is enough to tell JAX-WS that a POJO class is to be deployed as a web service, by annotating the class with the **`@WebService`** annotation, as shown in Code 2.2.

However, there are a number of requirements on Java classes that one would consider as web services:



- Must be annotated with `javax.ws.WebService`
- Must not be declared `final`
- Must not be **`abstract`**
- Must have a default public constructor
- Must not have a `finalize` method

Similarly, there are requirements on any methods in a class that one would consider as an operation in a web service:

```
1  public class AirportManager {
2    public long
3    addAirport(String code, String name) {
4      return dao.add(null, code, name).getId();
5    }
6    private AirportDAO dao = new AirportDAO();
7  }
```

E-1

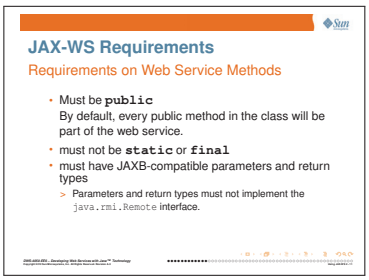
Code 2.1: A Simple Service: `AirportManager`

```
1  @WebService
2  public class AirportManager {
3    public long
4    addAirport(String code, String name) {
5      return dao.add(null, code, name).getId();
6    }
7    private AirportDAO dao = new AirportDAO();
8  }
```

E-2

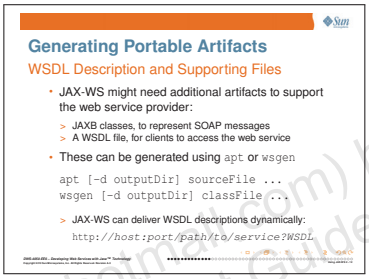
Code 2.2: A Simplest Web Service

- Must be `public`
  By default, every public method in the class will be part of the web service.
- must not be `static` or `final`
- must have JAXB-compatible parameters and return types
  - Parameters and return types must not implement the RMI marker interface, `java.rmi.Remote`.



To generate portable artifacts, use either the `apt` tool or the `wsgen` tool.

Use the `apt` tool to generate the artifacts when you start from Java source files, because the JAX-WS tools then has full access to the source code and can use parameter names that are otherwise not available through the Java reflection APIs. Use the `wsgen` tool to generate the artifacts when you start from Java classes.



The artifacts that these tools will generate include:

- JAXB classes, to represent SOAP messages
- A WSDL file, for clients to access the web service

Both of these tools can be run as command-line applications:

```
apt [-d outputDir] sourceFile ...
wsgen [-d outputDir] classFile ...
```

They can also be run as `ant` tasks. Code 2.3 shows an example of an `ant` task for `apt`.

The WSDL descriptions for web services need not be generated explicitly ahead of time, if they are needed to be consumed by clients. JAX-WS can deliver WSDL descriptions dynamically, using the URL:

```
http://host:port/path/to/service?WSDL
```

```
<apt
    debug="\${debug}"
    verbose="\${verbose}"
    destdir="\${build.classes.home}"
    sourcedestdir="\${build.classes.home}"
    sourcepath="\${basedir}/src">
      <classpath refid="jax-ws.classpath"/>
      <option key="r" value="\${build.home}"/>
      <source dir="\${basedir}/src">
         <include name="**/server/*.java"/>
      </source>
 </apt>
```

Code 2.3: Sample `ant` task for `apt`

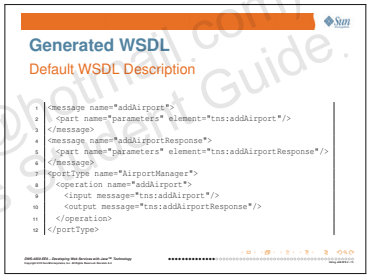
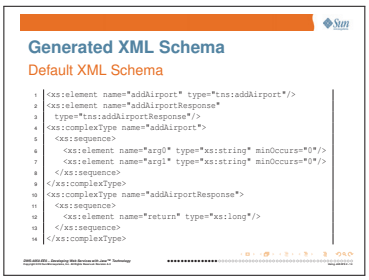


Figure 2.4 shows the WSDL description that JAX-WS would generated for the POJO web service class described above. Note the correlation between the `portType` element and the Java class, and between the `operation` elements and methods in the Java class. Since Java methods are logically request/response (a function call followed by a return back to the caller), the description of the `operation` involves two messages, one each for the request and its response.

```
<message name="addAirport">
  <part name="parameters" element="tns:addAirport"/>
</message>
<message name="addAirportResponse">
  <part name="parameters" element="tns:addAirportResponse"/>
</message>
<portType name="AirportManager">
  <operation name="addAirport">
    <input message="tns:addAirport"/>
    <output message="tns:addAirportResponse"/>
  </operation>
</portType>
```

Code 2.4: Generated WSDL for `AirportManager` Class

Figure 2.5 shows XML schema that is associated with the `AirportManager` service. Note how each of the messages required to describe operations offered by the web service has an XML schema that describes the structure of the message. This description explicitly mentions the root node of the request message, representing the method to call (`addAirport`), with the parameters described as children of that "method" node.

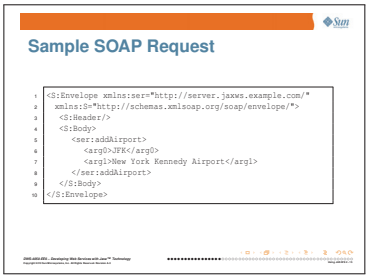



```
1  <xs:element name="addAirport" type="tns:addAirport"/>
2  <xs:element name="addAirportResponse"
3    type="tns:addAirportResponse"/>
4  <xs:complexType name="addAirport">
5    <xs:sequence>
6      <xs:element name="arg0" type="xs:string" minOccurs="0"/>
7      <xs:element name="arg1" type="xs:string" minOccurs="0"/>
8    </xs:sequence>
9  </xs:complexType>
10 <xs:complexType name="addAirportResponse">
11   <xs:sequence>
12     <xs:element name="return" type="xs:long"/>
13   </xs:sequence>
14 </xs:complexType>
```

Code 2.5: XML Schema Generated for `AirportManager`

Figure 2.3 shows a sample SOAP message that a client could direct at this service, and Figure 2.4 shows a sample SOAP response from this service. Figure 2.5 shows the raw HTTP message that the client would have sent to the web service, with the SOAP message embedded within it, and Figure 2.6 shows the raw HTTP response.

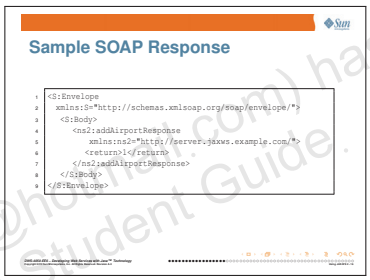
The sample SOAP message shown in Figure 2.3 is the *input* message, sent from client to server when the client attempts to invoke the `addAirport()` method defined by the `AirportManager` service. On line 5 one can see the method to be invoked, represented in the body of the message, with the parameters to the request as children elements (lines 6 and 7).



```
<S:Envelope xmlns:ser="http://server.jaxws.example.com/"
  xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Header/>
   <S:Body>
      <ser:addAirport>
         <arg0>JFK</arg0>
         <arg1>New York Kennedy Airport</arg1>
      </ser:addAirport>
   </S:Body>
</S:Envelope>
```

Figure 2.3: Sample SOAP Request for AirportManager

The sample SOAP message shown in Figure 2.4 is the *output* message, sent from server back to client once the response to the client's request has been computed. Note how this response message describes that it is a response to an `addAirport()` request, only through the `addAirportResponse` element in the body of the message.



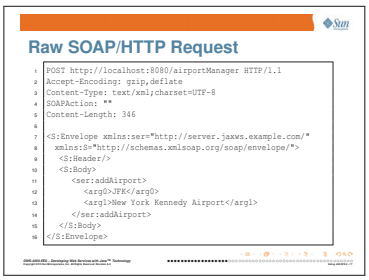
The response message doesn't usually represent explicitly which message it is a response to - the assumption is that the communications between client and server are based on HTTP, the client is holding the HTTP connection open while waiting for the response, and so no one else can use this connection until the client receives its response.

```
<S:Envelope
  xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:addAirportResponse
          xmlns:ns2="http://server.jaxws.example.com/">
         <return>1</return>
      </ns2:addAirportResponse>
   </S:Body>
</S:Envelope>
```

Figure 2.4: Sample SOAP Response from AirportManager

It would be more complex, though perhaps useful, to have the response messages represent explicitly the correspondence between specific request and response messages. This feature is available in JAX-WS via the WS-Addressing extension.

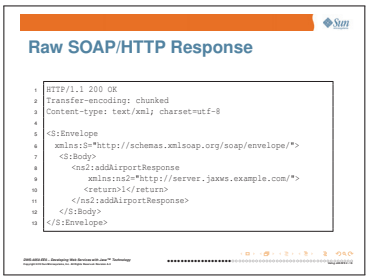
Figure 2.5 shows the raw HTTP message actually sent from client to web service. Note how the SOAP message is embedded within a larger SOAP request, and how there's a separation of responsibilities between the message and transport layers: the SOAP message represents the target object to whom the request is addressed – but not the network address where that target object is found, which is left to the transport layer. In the HTTP request, the target network address is represented via the `POST` URL (see line 1). For convenience, the HTTP request is also allowed to represent the object-level request carried by that HTTP request, in the `SOAPAction` HTPP header (in line 4, set in this example to an empty string).




```
1  POST http://localhost:8080/airportManager HTTP/1.1
2  Accept-Encoding: gzip,deflate
3  Content-Type: text/xml;charset=UTF-8
4  SOAPAction: ""
5  Content-Length: 346
6
7  <S:Envelope xmlns:ser="http://server.jaxws.example.com/"
8    xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
9     <S:Header/>
10    <S:Body>
11       <ser:addAirport>
12          <arg0>JFK</arg0>
13          <arg1>New York Kennedy Airport</arg1>
14       </ser:addAirport>
15    </S:Body>
16 </S:Envelope>
```


Figure 2.5: Raw SOAP/HTTP Request

Figure 2.6 shows the raw HTTP response. The HTTP response message mostly carries the SOAP message that represents the server's answer to this caller's request. All the features of the HTTP protocol are available to JAX-WS, when using HTTP as a transport. In particular, this figure shows how the underlying runtime could use the `"chunked"` encoding, to send an entity of arbitrary length (the SOAP message), without having to compute its overall size up front.



```
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: text/xml; charset=utf-8

<S:Envelope
  xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:addAirportResponse
          xmlns:ns2="http://server.jaxws.example.com/">
         <return>1</return>
      </ns2:addAirportResponse>
   </S:Body>
</S:Envelope>
```

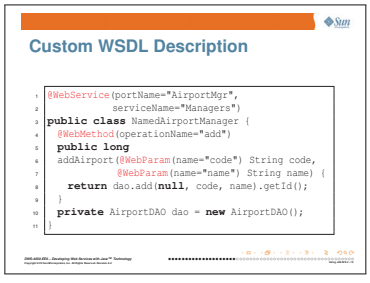
Figure 2.6: Raw SOAP/HTTP Response

The actual "chunked" representation of the HTTP message shown in Figure 2.6 is a little harder to read than the representation used here – this figure shows the result of collapsing the individual chunks back into a more readable representation of the complete entity payload.

Customizing the JAX-WS Web Service

Sometimes, it is necessary to override the default values used by JAX-WS to create the WSDL description for a POJO web service implementation.

The default names for the service, port, and operations in a web service are generated automatically by JAX-WS, based on the names of the corresponding Java constructs – but it is possible to override these defaults, by providing explicit values to the JAX-WS annotations, as show in Figure 2.6.



Code 2.7 and 2.8 show the WSDL and XML schema descriptions generated by JAX-WS for the POJO web service class shown in Code 2.6.

Explicit names to be used in the WSDL definition of an operation – and provided via attributes to the `@WebParam` annotation – are the only values that are typically specified explicitly, every time a bottom-up web service is defined.

The reason is that parameter names are the only thing that is lost, when querying classes for their API via reflection.

Code 2.7 lists the WSDL description that is generated by JAX-WS for the customized class shown in Code 2.6. Note how the two custom name attributes specified in the **@WebService** annotation affect the service bindings for the service in question (in lines 8 and 9), but not the `portType` name (in line 1, which remains the same as the class name). The custom operation name specified does affect the definition for the matching operation (in line 2).


```
1  @WebService(portName="AirportMgr",
2              serviceName="Managers")
3  public class NamedAirportManager {
4    @WebMethod(operationName="add")
5    public long
6    addAirport(@WebParam(name="code") String code,
7               @WebParam(name="name") String name) {
8      return dao.add(null, code, name).getId();
9    }
10   private AirportDAO dao = new AirportDAO();
11 }
```

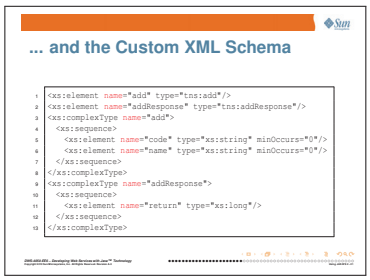
E-3

Code 2.6: Custom WSDL Description: Class `NamedAirportManager`

```
1  <portType name="NamedAirportManager">
2    <operation name="add">
3      <input message="tns:add"/>
4      <output message="tns:addResponse"/>
5    </operation>
6  </portType>
7  <!-- ... -->
8  <service name="Managers">
9    <port name="AirportMgr" binding="tns:AirportMgrBinding">
10     <soap:address
11       location="http://localhost:8080/namedManager"/>
12   </port>
13 </service>
```

Code 2.7: Customized Generated WSDL

Code 2.8 lists the XML schema generated by JAX-WS for the customized class shown in Code 2.6. The `name` attributes in the two **`@WebParam`** annotations are used to define the names for the two elements in the XML schema that will be used to represent both parameters in the SOAP message (in lines 5 and 6), and the `name` attribute in the **`@WebMethod`** annotation defines the name for the representation of the operation in line 1.




```
1  <xs:element name="add" type="tns:add"/>
2  <xs:element name="addResponse" type="tns:addResponse"/>
3  <xs:complexType name="add">
4    <xs:sequence>
5      <xs:element name="code" type="xs:string" minOccurs="0"/>
6      <xs:element name="name" type="xs:string" minOccurs="0"/>
7    </xs:sequence>
8  </xs:complexType>
9  <xs:complexType name="addResponse">
10   <xs:sequence>
11     <xs:element name="return" type="xs:long"/>
12   </xs:sequence>
13 </xs:complexType>
```

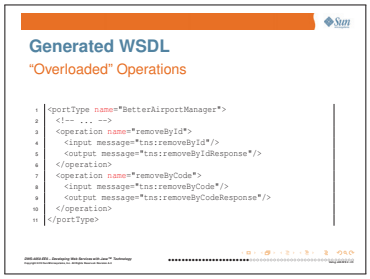
Code 2.8: Customized Generated XML Schema

There is one important difference between the Java and Web Services models: Java supports overloading functions (where several functions on the same instance have the same name, but can be disambiguated based on their parameter lists), while Web Services require that every operation have a unique name.


```
@WebService
public class BetterAirportManager {
  @WebMethod(operationName="removeById")
  public void removeAirport(long id) {
  }
  @WebMethod(operationName="removeByCode")
  public void removeAirport(String code) {
  }
  // ...
}
```

When the POJO web service class has overloaded functions, it is mandatory to use annotations to provide unique names for each overloaded function that will make it into the web service definition, as shown in Figure 2.9.

Figure 2.10 shows the WSDL description generated for the POJO class in Figure 2.9. Note how the WSDL description for the web service simple lists a series of operations, each with its own unique name – since the WSDL specification requires that the name of each operation in a web service be unique.




```
<portType name="BetterAirportManager">
  <!-- ... -->
  <operation name="removeById">
    <input message="tns:removeById"/>
    <output message="tns:removeByIdResponse"/>
  </operation>
  <operation name="removeByCode">
    <input message="tns:removeByCode"/>
    <output message="tns:removeByCodeResponse"/>
  </operation>
</portType>
```

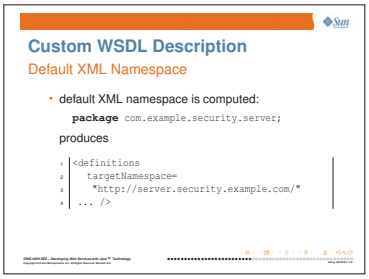
```
1  @WebService
2  public class BetterAirportManager {
3    @WebMethod(operationName="removeById")
4    public void removeAirport(long id) {
5    }
6    @WebMethod(operationName="removeByCode")
7    public void removeAirport(String code) {
8    }
9    // ...
10 }
```

E-4

Code 2.9: Capturing Overloaded Methods

```
1  <portType name="BetterAirportManager">
2    <!-- ... -->
3    <operation name="removeById">
4      <input message="tns:removeById"/>
5      <output message="tns:removeByIdResponse"/>
6    </operation>
7    <operation name="removeByCode">
8      <input message="tns:removeByCode"/>
9      <output message="tns:removeByCodeResponse"/>
10   </operation>
11 </portType>
```

Code 2.10: Generated WSDL – Overloaded Operations



In XML, every new vocabulary that is defined should be assigned its own *namespace*. This way, we can all avoid naming collisions, and we can document which vocabulary we're using, every time.

When JAX-WS creates an XML vocabulary for the WSDL definition for a POJO web service, it computes a (hopefully) unique namespace for it based on the original POJO class: package `com.example.security.server` produces the target namespace `"http://server.security.example.com/"`, as shown in Figure 2.7.

```
<definitions
 targetNamespace="http://server.security.example.com/"
 ... />
```

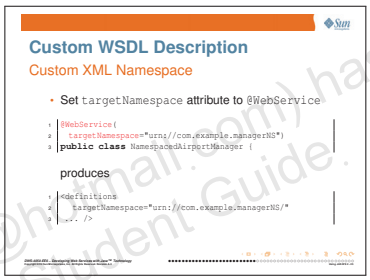
Figure 2.7: Target Namespace for Application Elements

```
@WebService(
  targetNamespace="urn://com.example.managerNS")
public class NamespacedAirportManager {
```

E-6

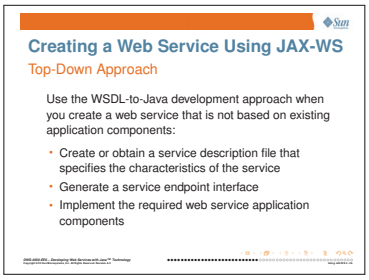
Code 2.11: Custom WSDL Description: Custom Namespace

To override this default namespace, set the `targetNamespace` attribute of `@WebService`, as shown in Code 2.11.



Creating a Web Service Using JAX-WS: Top-Down

Use the WSDL-to-Java development approach when you create a web service that is not based on existing application components.



This top-down approach requires the following steps:

1. Create or obtain a service description file that specifies the characteristics of the service, such as service method name, service parameters, and return values.
2. Generate a service endpoint interface.
3. Implement the required web service application components to satisfy the criteria specified in the service description.
4. Create a WAR file and deploy.

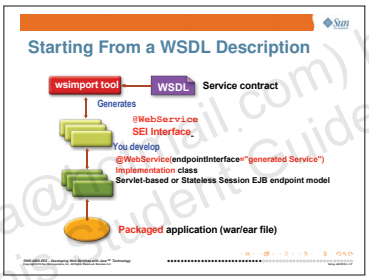
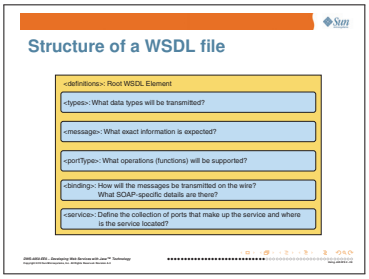


Figure 2.8 illustrates this approach.

The first step is to define the set of services to be supplied, including the list of operations supported by each service. This description is written in terms of the Web Service Description (or Definition) Language, WSDL. Figure 2.9 illustrates the contents of such a description.



The elements in a WSDL description include:

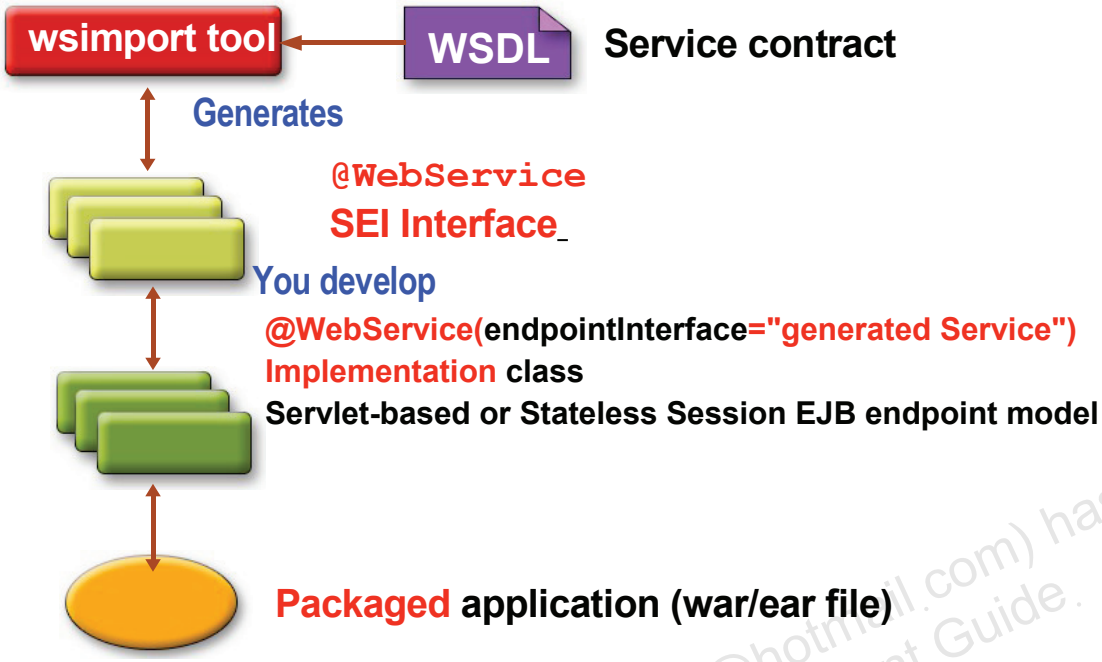


Figure 2.8: Starting From a WSDL Description

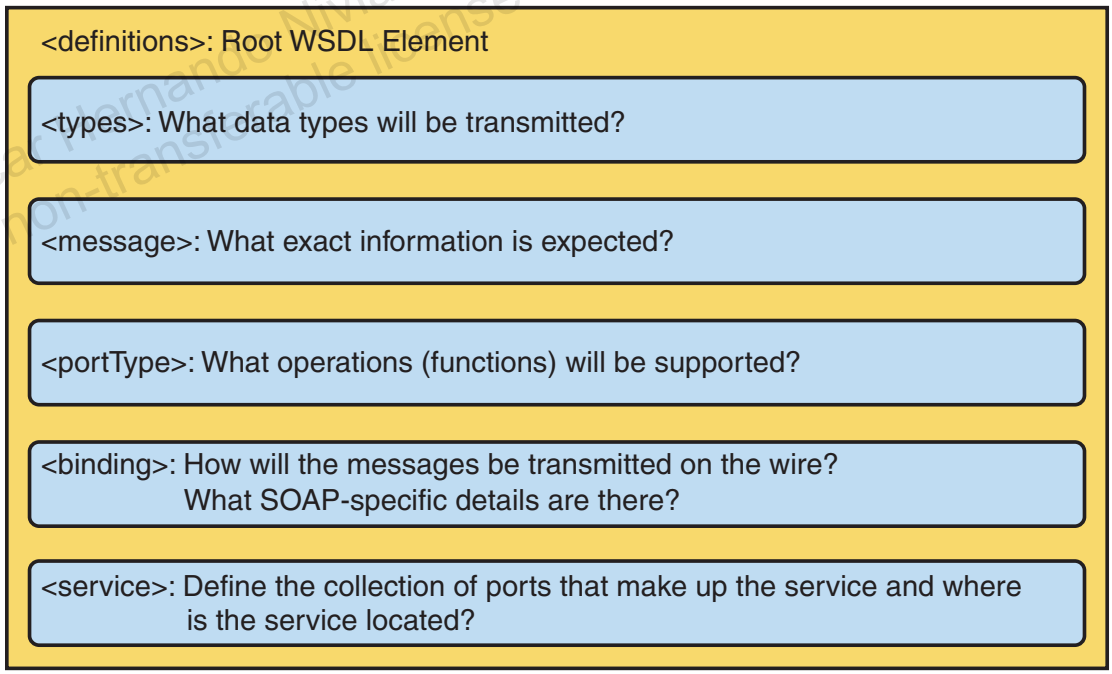
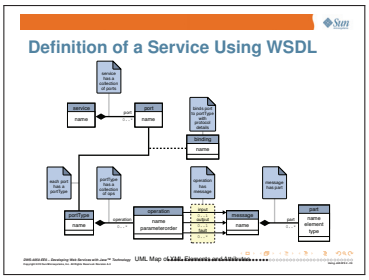


Figure 2.9: Structure of a WSDL file

`definitions`

Container for a service description. All global declaration of namespaces that should be in the document are done here.



`types` Holds one or more schema subelements that follow the syntax of a schema document. The data types to be used under the `message` element are defined.

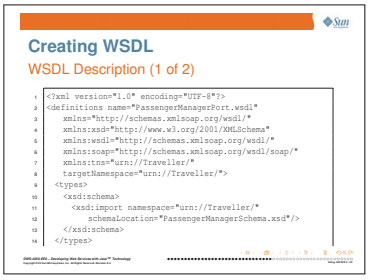
`message` Used to explain the data that is exchanged between a web service and a client and refers to the content in the `types` element. Each `message` element has one or more `part` subelements that define the individual piece of data within the message.

`portType` Specifies the set of actions or methods provided by a single endpoint of a web service. This is similar to providing method signatures. For each of the methods, you have an `operation` element within the `portType` element describing and uniquely identifying each method of the web service. Each operation element has `input` and `output` subelements describing the input value and response value respectively.

Figure 2.10 illustrates this structure.

Writing a WSDL Description of a Service

The first step is to define services and their operations in WSDL. Code 2.12 shows the definition of a service called `PassengerManager`, which will offer a single operation `addPassenger`, to add a new user to a directory. The WSDL description of the service is a little more verbose than the equivalent description in Java – but the WSDL specification provides a richer language to describe services.



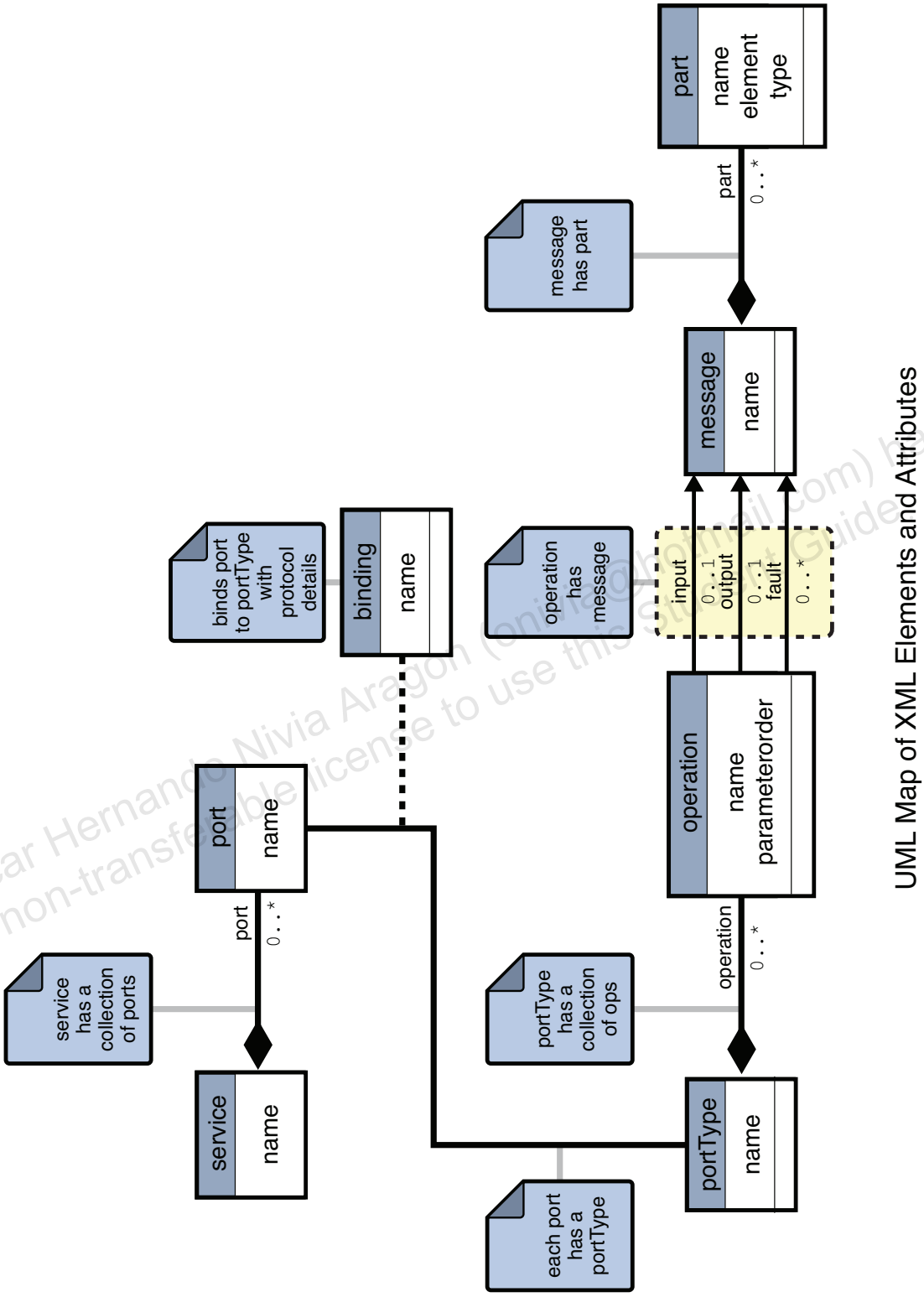


Figure 2.10: Definition of a Service Using WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="PassengerManagerPort.wsdl"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="urn://Traveller/"
    targetNamespace="urn://Traveller/">
  <types>
    <xsd:schema>
      <xsd:import namespace="urn://Traveller/"
          schemaLocation="PassengerManagerSchema.xsd"/>
    </xsd:schema>
  </types>
  <message name="addPassengerRequest">
    <part name="params" element="tns:addPassenger"/>
  </message>
  <message name="addPassengerResp">
    <part name="params" element="tns:addPassengerResponse"/>
  </message>
  <portType name="PassengerManager">
    <operation name="addPassenger">
      <input name="in1" message="tns:addPassengerRequest"/>
      <output name="out1" message="tns:addPassengerResp"/>
    </operation>
  </portType>
</definitions>
```

E-7

Code 2.12: `PassengerManager portType`


```
<xsd:schema id="PassengerManagerSchema.xsd"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="urn://Traveller/"
    elementFormDefault="qualified"
    targetNamespace="urn://Traveller/">
  <xsd:element name="addPassenger">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="firstName" type="xsd:string"/>
        <xsd:element name="lastName" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
```

E-8

Code 2.13: XML Schema Type for `addPassenger`

The data manipulated by the `PassengerManager` service – and in particular, the parameters required by the `addPassenger` operation – are described using XML schema types. Code 2.13 shows this XML schema. Note again how the XML schema described here captures explicitly the name of the operation that is requested along with the parameters to each call.

Concrete information about the service actually deployed that matches this description goes in the `bindings` element of the WSDL description. We could list the address where the service is to be found, for example, as shown in Code 2.14, line 13.


```
1  <binding name="binding" type="tns:PassengerManager">
2    <soap:binding style="document"
3      transport="http://schemas.xmlsoap.org/soap/http"/>
4    <operation name="addPassenger">
5      <soap:operation/>
6      <input><soap:body use="literal"/></input>
7      <output><soap:body use="literal"/></output>
8    </operation>
9  </binding>
10 <service name="PassengerManagerService">
11   <port name="PassengerManager" binding="tns:binding">
12     <soap:address
13       location="http://localhost:8080/passengerManager"/>
14   </port>
15 </service>
```


Code 2.14: WSDL Bindings for `PassengerManager`

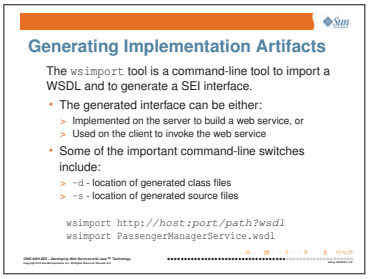
Generating JAX-WS Artifacts

The `wsimport` tool generates JAX-WS portable artifacts, such as:

- Service endpoint interface (SEI)
- Service
- Exception class mapped from `wsdl:fault` (if any)

- Async Response Bean derived from response `wsdl:message` (if any)
- JAXB generated value types (mapped Java classes from schema types)

The `wsimport` tool can be launched in a shell (or command-line); there is also an ant task to import and compile the WSDL. The generated interface can either be implemented on the server to build a web service or can be used on the client to invoke the web service. Some of the important command-line switches include the following:



-d The output location of generated class files

-s The output location of generated source files

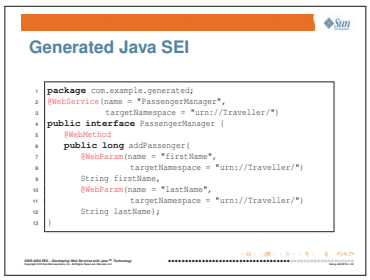
The following example shows the use of the `wsimport` command. In the following command, the JAX-WS artifacts of the WSDL file, as mentioned in the URL, are generated and saved in the `gen` folder:

```
wsimport -d gen http://localhost:8080/app/service?wsdl
```

A defined mapping exists between the elements in a WSDL file and Java interfaces, classes, methods, and arguments. The following list describes this mapping:

- A `wsdl:definitions` element and its associated `targetNamespace` attribute maps to a Java package.
- A `wsdl:portType` maps to a Java service endpoint interface in a mapped package.
- Each `wsdl:operation` is mapped to a method in the corresponding Java service endpoint interface.
- The `wsdl:input`, `wsdl:output`, and `wsdl:fault` elements are mapped to method parameters and Java exceptions respectively.
- The contents of `wsdl:types` are passed to JAXB along with any additional type or element declaration required to map other WSDL constructs to a Java interface.
- Binding of a port type to a protocol can bring changes in the mapping of a port type to a Java interface, such as *SOAP binding* and *MIME binding*.

Figure 2.15 shows the Java class that `wsimport` would create for the WSDL description show in Figure 2.12. Note how the generated code is a little more verbose than the service that we implemented bottom-up, back in Figure 2.2. The reason is that we allowed JAX-WS to use defaults values for several features of our service provider, while JAX-WS specifies all those values explicitly here.

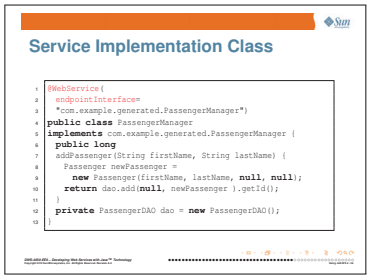



```
1  package com.example.generated;
2  @WebService(name = "PassengerManager",
3                targetNamespace = "urn://Traveller/")
4  public interface PassengerManager {
5      @WebMethod
6      public long addPassenger(
7          @WebParam(name = "firstName",
8                    targetNamespace = "urn://Traveller/")
9          String firstName,
10         @WebParam(name = "lastName",
11                   targetNamespace = "urn://Traveller/")
12         String lastName);
13 }
```

E-10

Code 2.15: Generated Java SEI

To implement the service endpoint interface, you must provide a **`@WebService`** annotation on the implementation class with an `endpointInterface` attribute specifying the qualified name of the endpoint interface class generated for you. The example in Figure 2.16 shows a possible implementation class. Note (in lines 2-3) that it is not enough to have the implementation class simply implement the SEI interface – the **`@WebService`** annotation must explicitly indicate which interface is the service endpoint interface.



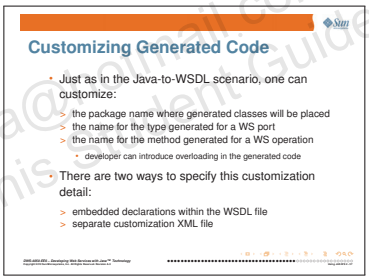
Just as in the Java-to-WSDL scenario, one can customize:

```
1  @WebService(
2    endpointInterface=
3    "com.example.generated.PassengerManager")
4  public class PassengerManager
5  implements com.example.generated.PassengerManager {
6    public long
7    addPassenger(String firstName, String lastName) {
8      Passenger newPassenger =
9        new Passenger(firstName, lastName, null, null);
10     return dao.add(null, newPassenger ).getId();
11   }
12   private PassengerDAO dao = new PassengerDAO();
13 }
```

E-13

Code 2.16: Service Implementation Class

- the package name where generated classes will be placed
- the name for the type generated for a WS port
- the name for the method generated for a WS operation
  - developer can introduce overloading in the generated code



There are two ways to specify this customization detail:

- embedded declarations within the WSDL file
- separate customization XML file

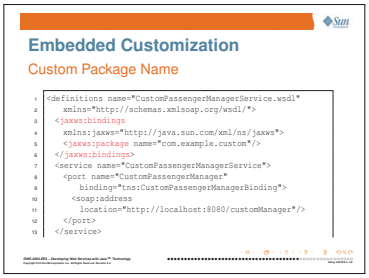
When embedding customization declarations within the WSDL file, the location (or *context*) of that declaration within the WSDL description matters - the context for a declaration defines what is affected by that declaration.

When specifying customizations in a separate customization file, the WSDL element that is affected by an individual customization element is explicitly identified as part of each customization element via XPath expressions that refer to the original WSDL description.

Code 2.17 shows an embedded declaration to customize the package name to be used in the generated code. Code 2.18 shows an embedded declaration to customize the Java class to be used. Code 2.19 show an embedded declaration

to customize the name of the Java method to be used. Code 2.20 shows the Java code generated when all three customizations are used.

Code 2.17 shows an embedded declaration to customize the package name to be used in the generated code. Note how this customization declaration directly as a chile of the `definitions` element – the package name customization applies to everything defined by the WSDL file.

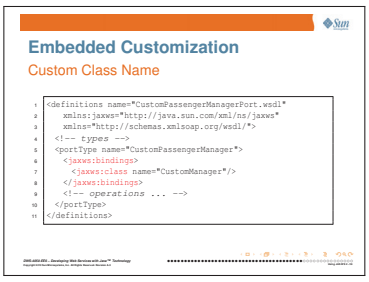



```
1  <definitions name="CustomPassengerManagerService.wsdl"
2      xmlns="http://schemas.xmlsoap.org/wsdl/">
3    <jaxws:bindings
4      xmlns:jaxws="http://java.sun.com/xml/ns/jaxws">
5      <jaxws:package name="com.example.custom"/>
6    </jaxws:bindings>
7    <service name="CustomPassengerManagerService">
8      <port name="CustomPassengerManager"
9          binding="tns:CustomPassengerManagerBinding">
10       <soap:address
11         location="http://localhost:8080/customManager"/>
12     </port>
13   </service>
```

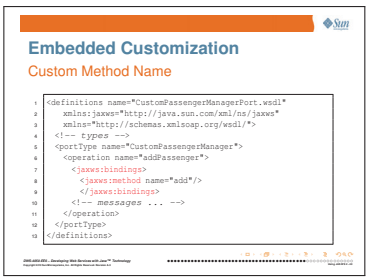
E-14

Code 2.17: Embedded Customization: Package

Code 2.18 shows an embedded declaration to customize the Java class to be used. This customization appears in the context of the specific `portType` element that is to be represented by the class named: a class named `CustomDirectory` (line 6) will be used to represent the web service described by the `portType` in lines 4 to 9.



Code 2.19 show an embedded declaration to customize the name of the Java method to be used: a Java method with name `add()` (line 7) will be used to represent the WSDL operation named `addPassenger`, since the customization declaration is embedded within that operation (lines 5 to 10).



```
<definitions name="CustomPassengerManagerPort.wsdl"
    xmlns:jaxws="http://java.sun.com/xml/ns/jaxws"
    xmlns="http://schemas.xmlsoap.org/wsdl/">
  <!-- types -->
  <portType name="CustomPassengerManager">
    <jaxws:bindings>
      <jaxws:class name="CustomManager"/>
    </jaxws:bindings>
    <!-- operations ... -->
  </portType>
</definitions>
```

E-15

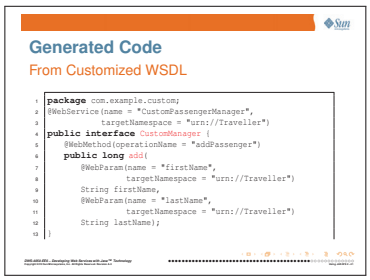
Code 2.18: Embedded Customization: Class

```
<definitions name="CustomPassengerManagerPort.wsdl"
    xmlns:jaxws="http://java.sun.com/xml/ns/jaxws"
    xmlns="http://schemas.xmlsoap.org/wsdl/">
  <!-- types -->
  <portType name="CustomPassengerManager">
    <operation name="addPassenger">
      <jaxws:bindings>
        <jaxws:method name="add"/>
        </jaxws:bindings>
      <!-- messages ... -->
    </operation>
  </portType>
</definitions>
```

E-15

Code 2.19: Embedded Customization: Method

Code 2.20 shows the Java code generated when all three customizations are used.



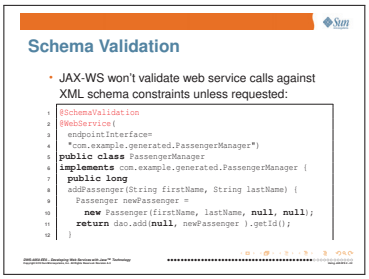

```
1  package com.example.custom;
2  @WebService(name = "CustomPassengerManager",
3              targetNamespace = "urn://Traveller")
4  public interface CustomManager {
5      @WebMethod(operationName = "addPassenger")
6      public long add(
7          @WebParam(name = "firstName",
8                    targetNamespace = "urn://Traveller")
9          String firstName,
10         @WebParam(name = "lastName",
11                   targetNamespace = "urn://Traveller")
12         String lastName);
13 }
```

E-16

Code 2.20: Class Generated From Customized WSDL

Schema Validation

We mentioned earlier that the ability to validate input to a web service via schema validation was one of the features that made the top-down approach attractive. There is a catch, however: JAX-WS will not validate web service calls against schema constraints unless explicitly requested. Code 2.21 shows how this is done.

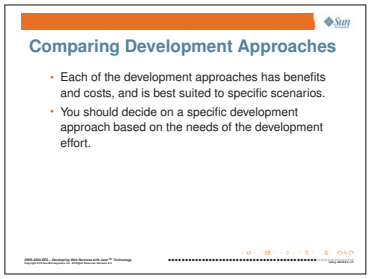



```
1  @SchemaValidation
2  @WebService(
3    endpointInterface=
4    "com.example.generated.PassengerManager")
5  public class PassengerManager
6  implements com.example.generated.PassengerManager {
7    public long
8    addPassenger(String firstName, String lastName) {
```

Code 2.21: How to Enable Schema Validation

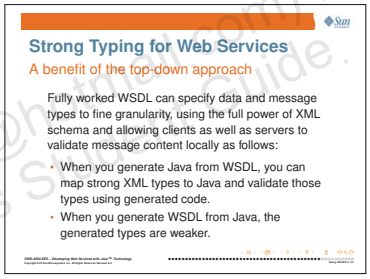
Comparing Development Approaches

Each of the two development approaches, starting from Java class and starting from WSDL, has benefits and costs associated with it. You should decide on a specific development approach based on the needs of the development effort. Each approach offers its own advantages best suited to specific scenarios.



Strong Typing for Web Services

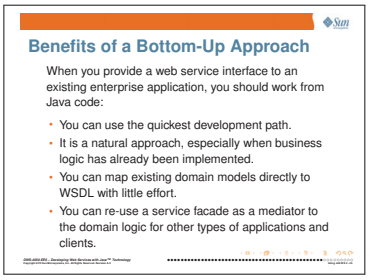
The web services architecture is open and allows many different types of participants; therefore, strong typing is especially important for smooth operation.



Fully worked WSDL can specify data and message types to fine granularity, using the full power of XML schema and allowing clients as well as servers to validate message content locally as follows:

- When you generate WSDL from a Java class, the generated types are weaker. Java code generated from WSDL for the client reflects this. Thus, validation logic that is encoded in the service is lost, and invalid content must be caught by the service on receipt of the message, which is more expensive and cumbersome than client-side pre-validation.
- When you generate a Java class from WSDL, you can map strong XML types to Java and validate those types using generated code. For example, XML schema enumerations can be mapped to a Java class and a Java class can be mapped to XML schema enumerations. Working from WSDL-defined enumerations allows all parties to share the knowledge that only certain values are allowed for certain fields.

Benefits and Costs of Starting from a Java Development Approach



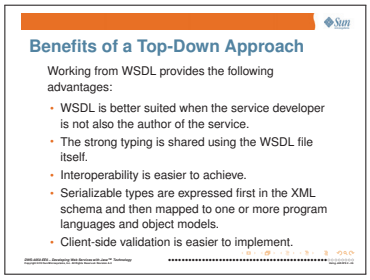
When you define the semantics of the service first – such as when you provide a web service interface to an existing enterprise application – you should work from Java code. When you create web services, working from existing Java components provides the quickest development path and the shortest time to market. Using Java components as the basis for web service development is a natural approach, especially when business logic has already been implemented and merely needs to be wrapped by the web service elements.

Existing domain models can be mapped directly to WSDL with little effort. You can leverage existing domain classes and might even use some of the classes directly as source classes for generating serializers, WSDL descriptors, and so forth. However, to be usable, the semantics of the domain class must fulfill the JAX-WS mapping requirements. Otherwise, you must wrap existing domain classes using a new JAX-WS service and value types.

You can create a set of SEIs and value types as a common façade over the domain logic and expose these SEIs and value types as services. The façade acts as a coherent statement of service semantics, expressed clearly and simply in the Java programming language. A service façade can be reused as a mediator to the domain logic for web services and for web applications, and other local and remote client types.

Benefits and Costs of Starting from WSDL Development Approach

The starting from WSDL development approach is most suitable when you build a web service from scratch and the web service elements are the fundamental component model. You should also use this development approach when your primary focus is on client-service interoperability.



Working from WSDL provides the following advantages:

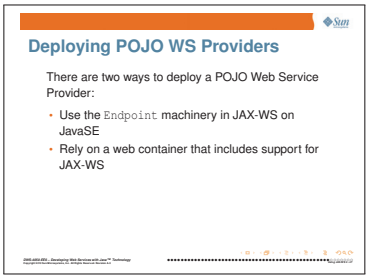
- WSDL is better suited to situations in which the service developer is not the author of the service semantics and workflow, such as when implementing to a certain industry's convention for web services.
- The strong typing is shared using the WSDL file itself. Starting from WSDL and XML schema types puts strong types where they can be shared by services and clients. Each generation of XML or WSDL-to-Java mapping or the reverse tends to weaken the types involved.
- Working with WSDL offers you the best interoperability because it allows you to author the WSDL file directly and gives you control over data type and encoding choices.
- Serializable types are expressed first in XML schema, and then mapped to one or more programming languages and object models.
- Client-side validation is easier to implement.

However, integration of WSDL-generated code with domain logic can be unwieldy. Generating a proper set of domain classes from WSDL imposes a dependency between the domain code and the WSDL and the WSDL-to-Java mapping. This is problematic when the domain logic is needed outside the web service context, perhaps for a web, enterprise, or standalone application. In this situation, the integration can offset some of the potential benefits of using a WSDL-to-Java development approach.

In practice, preserving multi-tier service architecture can be a challenge when you use the starting from WSDL development approach. You are often tempted to take shortcuts from the SEI classes to persistent resources. This happened with early web development, resulting in a large generation of two-tier web applications.

Deploying POJO Web Service Providers

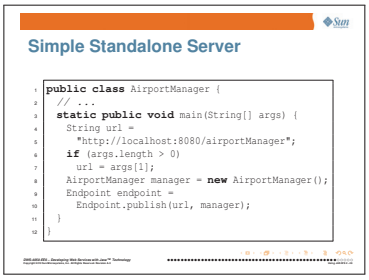
The default implementation of a web service relies on SOAP over HTTP – which means an infrastructure that supports HTTP is going to be required, in order to deploy web services.



There are common ways to deploy a POJO Web Service Provider:

- Since Java 6, Java has a simple web server built into JavaSE. This built-in server is not recommended for production use – but it will work well to test web services. JAX-WS comes with the machinery necessary to deploy POJO web services to this built-in web server, via the `Endpoint` class.
- Rely on a web container that includes support for JAX-WS. Since JAX-WS implements the support it provides to deploy web services in terms of a JAX-WS servlet, pretty much any servlet-compliant web container will suffice.

Figure 2.2 illustrated a very simple web service provider, implemented as a JAX-WS POJO web service. Code 2.22 shows the minimal structure needed to deploy a JAX-WS web service, based only on JAX-WS and the basic web container implementation built into JavaSE 6.

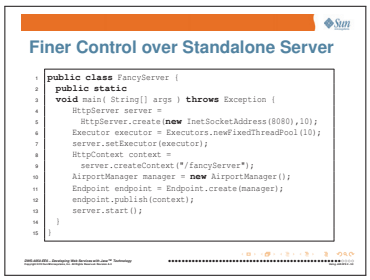



```
1  public class AirportManager {
2    // ...
3    static public void main(String[] args) {
4      String url =
5        "http://localhost:8080/airportManager";
6      if (args.length > 0)
7        url = args[1];
8      AirportManager manager = new AirportManager();
9      Endpoint endpoint =
10       Endpoint.publish(url, manager);
11   }
12 }
```

E-2

Code 2.22: Simple Standalone Server

Even though the web container built into JavaSE 6 is not recommended for production use, it is still possible to configure to some extent the machinery it will use behind the things. For example, it is possible to configure the thread pool that this internal web container will use to process incoming requests. Code 2.23 shows an example that fine-tunes the configuration of the internal JavaSE web container, to deploy a JAX-WS web service on it.




```
1  public class FancyServer {
2    public static
3    void main( String[] args ) throws Exception {
4        HttpServer server =
5          HttpServer.create(new InetSocketAddress(8080),10);
6        Executor executor = Executors.newFixedThreadPool(10);
7        server.setExecutor(executor);
8        HttpContext context =
9          server.createContext("/fancyServer");
10       AirportManager manager = new AirportManager();
11       Endpoint endpoint = Endpoint.create(manager);
12       endpoint.publish(context);
13       server.start();
14   }
15 }
```

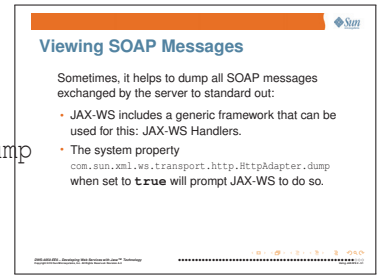
E-12

Code 2.23: Finer Control over Standalone Server

Debugging Web Service Interactions

Sometimes, it helps to dump all SOAP messages exchanged by the server to standard out:

- JAX-WS includes a generic framework that can be used for this: JAX-WS Handlers.
- The system property `com.sun.xml.ws.transport.http.HttpAdapter.dump` when set to `true` will prompt JAX-WS to do so.



There are a number of strategies for testing web service provider implementations:

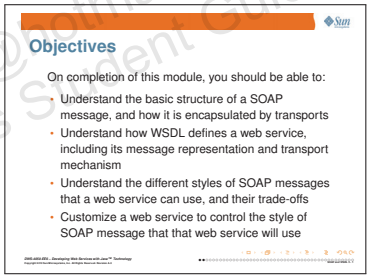
- In GlassFish, `http://`*`hostname`*`/`*`webService`*`?Tester` generates a web page that can be used to test the web service specified by the URI.
  - This only works for operations whose arguments are simple types.
- The SoapUI plugin for NetBeans/Eclipse will generate test suites for all operations in a web service
- A custom web services client.

Chapter 3

SOAP and WSDL

On completion of this module, you should be able to:

- Understand the basic structure of a SOAP message, and how it is encapsulated by transports
- Understand how WSDL defines a web service, including its message representation and transport mechanism
- Understand the different styles of SOAP messages that a web service can use, and their trade-offs
- Customize a web service to control the style of SOAP message that that web service will use



Relevance

The following questions are relevant to understanding SOAP:

- Having HTTP, why is another protocol (SOAP) needed?
- What type of information is required to invoke an application function?
- What type of information is passed between a service and service client during a web service exchange?
- What characteristics are important when choosing the message format used in a service interaction?

The following questions are relevant to understanding WSDL:

- Why do you need to describe a web service?
- What distinguishes a web service from other types of remote component access mechanisms, such as RMI and CORBA?
- What information is typically required to invoke a method on a remote application component?
- What is an appropriate format for a file that describes the functionality available from a web service?

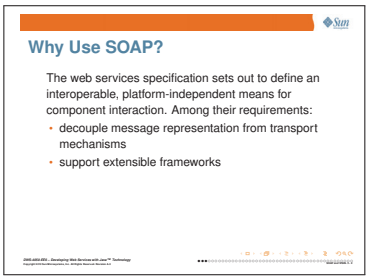
Additional Resources

The following references provide additional information on the topics described in this module:

- The Sun web services home page, `http://java.sun.com/webservices`.
- The WS-I home page, `http://ws-i.org`.
- The W3C home page, `http://www.w3.org`.
- Steve Graham, Simeon Simeonov, Toufic Boubez, Doug Davis, Glen Daniels, Yuichi Nakamura, Ryo Neyama. Building Web Services with Java. SAMS Publishing: 2001.
- W3C Note on SOAP Messages with Attachments, `http://www.w3.org/TR/SOAP-attachments`
- Article by Mc Carthy James, “Reap the Benefits of Document Style Web Services,” `http://www-128.ibm.com/developerworks/webservices/library/ws-docstyle.html`.
- The Sun web services home page, `http://java.sun.com/webservices`
- The W3C page on WSDL 1.1, `http://www.w3.org/TR/wsdl`
- Chappel Dave, Jewell Tyler, Java Web Services. O’Reilly, March 2002.
- The WS-I home page, `http://www.ws-i.org`
- Article by Sameer Tyagi, “Patterns and Strategies for Building Document-Based Web Services”, `http://java.sun.com/developer/technicalArticles/xml/jaxrpcpatterns/`

SOAP

Basic Structure of a SOAP Message



You can pass information between applications using only XML and HTTP. However, this approach has shortcomings due to a lack of commonly understood and standard practices.

SOAP was developed to address some of these issues and limitations. SOAP is a lightweight, text-based wire protocol used to encapsulate serialized data in an XML wrapper. SOAP provides the following features:

- Data encoding rules
- An extensible packaging structure for messages
- A strategy for binding SOAP messages to a transport protocol

These features provide a message exchange infrastructure for bridging the gap between heterogeneous applications or application components. With SOAP, data is encoded in a human-readable XML format, unlike the binary format used by other types of data encoding schemes.

Originally, SOAP was an acronym which stood for “Simple Object Access Protocol”. As of the SOAP v.1.2 specification, however, they consider it to be just a name.

The SOAP Specification

SOAP defines an XML vocabulary for messaging.

- SOAP version 1.1 is the most widely used version.
- Most SOAP-based tools also support the SOAP with Attachments specification that defines how to include MIME attachments with SOAP messages.

- SOAP Version 1.2 is cleaner, has better web integration, is more versatile, and is faster:
  - Cleaner – Easy to understand processing and extensibility models, increased interoperability.
  - Better web integration – Better integration with XML standards and the architecture of the web.
  - More versatile – Binding framework provides protocol independence.
  - Faster – Based on XML Infoset, allowing performance optimization.

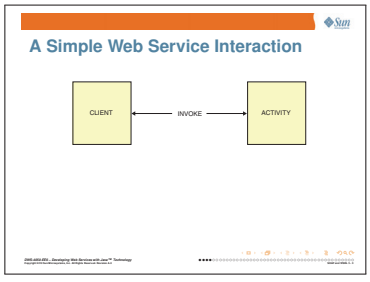
You can obtain detailed information about the changes and benefits of SOAP 1.2 from: `http://www.w3.org/2003/06/soap11-soap12`.

A SOAP message must be bound to a transport protocol for transport across the wire. You can bind a SOAP message to any transport protocol that supports the transmission of XML-based messages. HTTP is the most common and easiest protocol.

The WS-I Basic Profile 1.1 requires SOAP 1.1 over HTTP. Although HTTP 1.1 is preferred, version 1.0 is allowed.

Messaging Over the Web

SOAP fosters the development of component-based web services, extending the DOC model. A SOAP-based service can be considered an API exposed over the web and addressable at a single URI. The URI becomes a portal for RPC interactions.



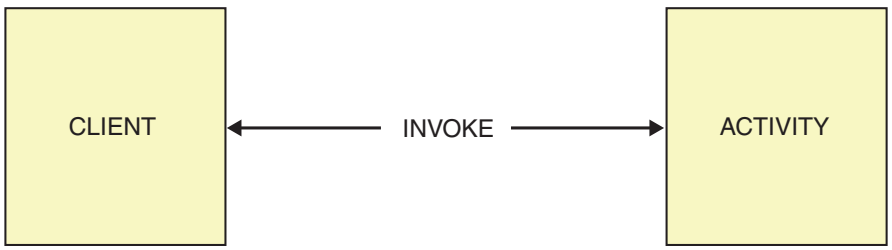


Figure 3.1: A Simple Web Service Interaction

SOAP defines a common message format for inter-application communication. Figure 3.1 shows the interaction between a client and web service. The interaction uses SOAP messages passed over an HTTP request.

A benefit of using SOAP/HTTP messaging is the ability of SOAP messages to pass through firewalls that exist along the message path.

- This is a common and desirable feature of web services that use HTTP as a transport protocol.
- The inability to seamlessly pass through firewalls has been troublesome for other distributed-computing infrastructures, such as DCOM, CORBA, and Java RMI.

In fact, the Java RMI transport layer supports an approach similar to that of SOAP: the Java RMI transport layer can wraps the RMI message payload in an HTTP request, as a fallback, if the standard RMI port cannot be reached.

Nodes

The participants of a SOAP message exchange are described in terms of nodes.

The SOAP node types include the following:

- Initial sender – A SOAP initial sender creates and transmits a SOAP message.
- Ultimate receiver – A SOAP ultimate receiver at the message destination processes the SOAP message, including the header, if applicable, and body components. The ultimate receiver can generate a SOAP response or fault message.
- Intermediary – Nodes, sometimes called message handlers, can also exist between the initial sender and ultimate receiver nodes. SOAP intermediaries can only process the header components of a SOAP message. Intermediary nodes can be both SOAP senders and receivers. Intermediary nodes receive messages from upstream nodes and can send the message to the next node in the chain after they have processed the message.

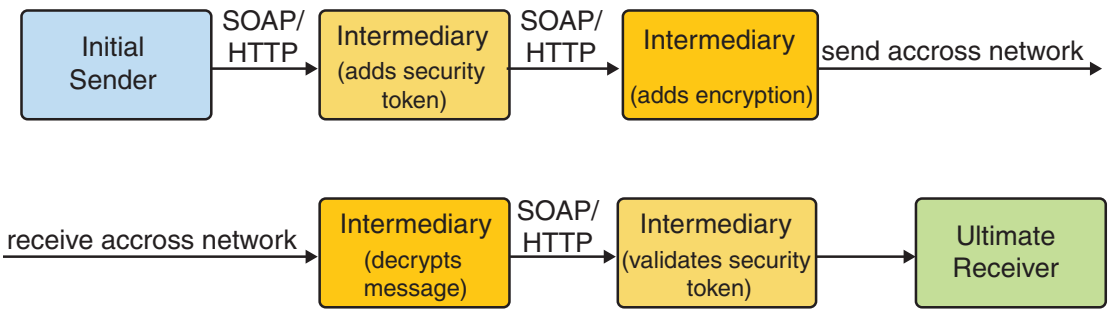
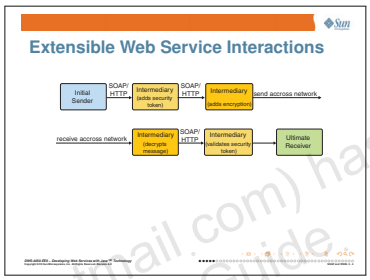


Figure 3.2: Extensible Web Service Interactions

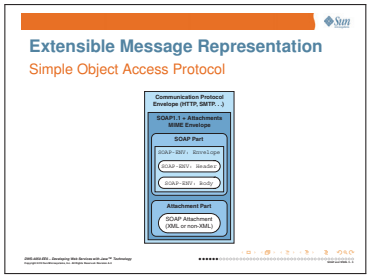


Intermediaries play a vital role in implementing security in distributed systems. You can ensure the scalability of distributed systems by the use of intermediaries.

Intermediaries can provide value-added services in a distributed system. A few examples are:

- Securing message exchanges, typically when messages pass through untrustworthy domains, such as using HTTP/SMTP on the Internet. SOAP messages can be secured by passing them through an intermediary that first encrypts them and then digitally signs them. On the receiving side, an intermediary performs the inverse operations, checking the digital signature and, if it is valid, decrypting the message.
- Providing message-tracing facilities. Tracing allows the recipient of messages to find out the exact path that the message went through, complete with detailed timings of arrivals and departures to and from intermediaries along the way. This information is valuable for tasks, such as measuring quality of service (QoS), auditing systems, and identifying scalability bottlenecks.

SOAP Message Format



The SOAP standards defines a hierarchical model for the structure of a SOAP message. The SOAP message fundamentally consists of a single XML element called the SOAP envelope. The SOAP envelope includes the following additional elements:

- The SOAP header that contains peripheral information that affects the processing of the message.

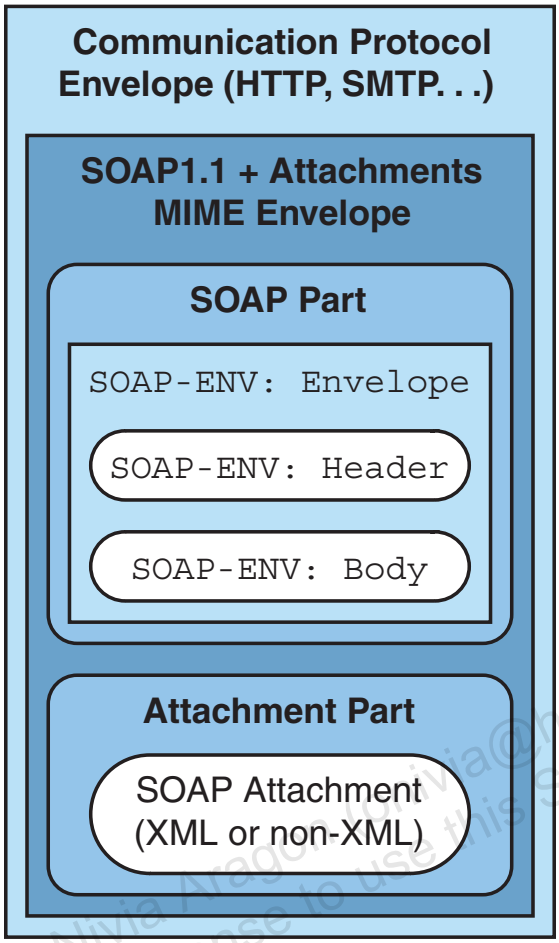
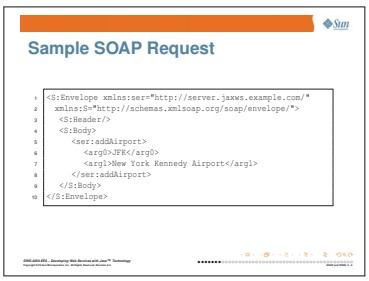


Figure 3.3: Extensible Message Representation

- The SOAP body that contains the core message content.

Attachments are supported outside the SOAP envelope. The transport protocol, such as HTTP, manages attachments used to transmit the SOAP message.

Figure 3.4 shows a sample SOAP request message. The SOAP envelope shows includes both header and body elements, although the header element is empty. The body element contains a representation of the request, as an XML structure. XML namespaces are used to distinguish SOAP elements from application-specific elements.



```
<S:Envelope xmlns:ser="http://server.jaxws.example.com/"
  xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Header/>
   <S:Body>
      <ser:addAirport>
         <arg0>JFK</arg0>
         <arg1>New York Kennedy Airport</arg1>
      </ser:addAirport>
   </S:Body>
</S:Envelope>
```

Figure 3.4: Sample SOAP Message

```
<S:Envelope
  xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:addAirportResponse
          xmlns:ns2="http://server.jaxws.example.com/">
         <return>1</return>
      </ns2:addAirportResponse>
   </S:Body>
</S:Envelope>
```

Figure 3.5: Sample SOAP Response

Figure 3.5 shows a sample SOAP response. SOAP responses also include header and body elements; the application-level response is embedded in the body element.

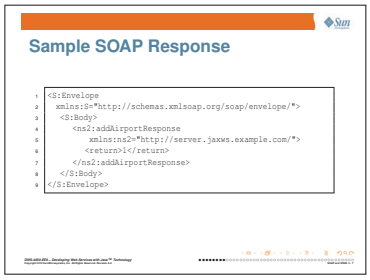


Table 3.1 summarizes the XML namespaces associated with SOAP.

Specification	Namespace
SOAP 1.1	http://schemas.xmlsoap.org/soap/envelope/
	http://schemas.xmlsoap.org/soap/encoding/
SOAP 1.2	http://www.w3.org/2001/09/soap-envelope

Table 3.1: SOAP Namespaces

The SOAP `Envelope` Element

The SOAP `Envelope` is the top-level element of the XML document representing a SOAP message. The SOAP `Envelope` element has the following characteristics:

- It typically defines namespace prefixes for the entire message, although this is not required.
- It declares the data-encoding style for the message.
- It contains an optional `Header` element and a mandatory `Body` element.

The SOAP envelope acts as a wrapper around the message content. It is the interface between the supporting protocol and XML. The SOAP envelope facilitates the transmission of potentially complex XML information over a simple, trusted protocol, such as HTTP or SMTP.

The SOAP `Header` Element

The SOAP `Header` element is a single optional XML element placed as a child of the SOAP `Envelope`:

- The header can hold any element or attribute content from non-SOAP namespaces.
- Each direct child of the header element is called a header entry.
- All immediate child elements of the SOAP `Header` element must be XML namespace-qualified.

Headers extend the SOAP model and also extend the semantics of each particular message.

The SOAP `Body` Element

The SOAP message `Body` is a single required element placed as a child of the SOAP envelope. The SOAP `Body` element holds the basic message content, such as method name to invoke and arguments, or response name and results.

Information intended for the ultimate message destination is included in the message body element.

The structure of the body element is similar to that of the `Header` element:

- Any element or attribute content from non-SOAP namespaces is allowed.
- Immediate children are defined as entries, but the body content can be of any arbitrary structure.
  - In an RPC-style encoding, typically one body entry identifies the invoked method. This body entry has individual child elements for each method argument.
  - In a document style encoding, the body can hold document-style information, such as an XHTML-encoded presentation fragment or document.
- The one standard body entry type is the SOAP fault.

SOAP Faults

The SOAP `Fault` element is a standard body entry type used for error and exception reporting:

- The SOAP `Fault` element appears as a child of the `Body` element.
- There can be only one fault entry. The SOAP specification permits mixing a single fault entry with other response information.

The SOAP Fault has four child elements, including the following:

- The `faultcode` element expresses an identifier of the fault condition or reason, intended for machine processing.
- The `faultstring` element provides a human-readable description of the fault condition.
- The `faultactor` element can be used to identify what actor in the message path caused the problem. This element is optional.
- The `detail` element is an open-content element allowing the addition of application-specific details in the fault message. This element is optional.

Transport Protocols for SOAP

SOAP can travel over a number of common Internet protocols, such as HTTP, SMTP, and FTP.

SOAP Over HTTP

HTTP is the most commonly used transport protocol for SOAP messaging. HTTP offers the following advantages:

- It is ubiquitous.
- It is scalable and robust.
- It connects two live endpoints over a network.
- It naturally supports a request/response messaging style.
- It offers extensible headers, which are helpful to the SOAP model.
- It supports attachments.
- It offers a role-based authentication model.

Secure SOAP exchanges are possible over HTTPS connections. HTTPS is not the only means of securing SOAP communications, but it is a relatively simple mechanism to implement; it is therefore a commonly accepted solution.

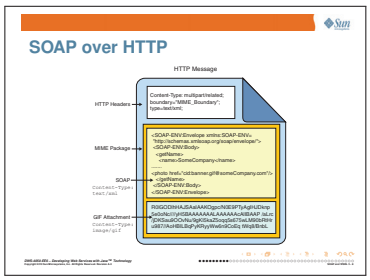
A limitation when using HTTP as the communications protocol for exchanging SOAP messages is that it must be possible to deliver the SOAP message as part of the HTTP communication. This means, for example, that HTTP `POST` messages can be used to deliver SOAP messages, since `POST` messages have a payload area. HTTP `GET` messages, on the other hand, would not be acceptable as a means to deliver SOAP messages – they only have a URI, but no payload.

In SOAP 1.1, the value of `Content-Type` in an HTTP message exchange is the string `text/xml`, which indicates that the payload is an XML document. In SOAP 1.2, it is represented as

```
Content-Type: application/soap+xml; charset=utf-8.
```

There are two MIME assignments for XML data. These are: `application/xml` and `text/xml` (both from RFC 3023).
Because of the wide variety of documents that can be expressed using an XML syntax, additional MIME types are needed to differentiate between types of content. XML-based formats add a suffix of `+xml` to the MIME type – for example, `application/xhtml+xml` (RFC 3236).

A SOAP message typically travels as the payload of a transport protocol. Figure 3.6 illustrates how a SOAP message can be embedded in the the payload of a HTTP message. In this figure, the larger HTTP message is encoded using the MIME Multipart specification, to include both the SOAP envelope proper, and some of the information associated with that message, captured as a MIME attachment.



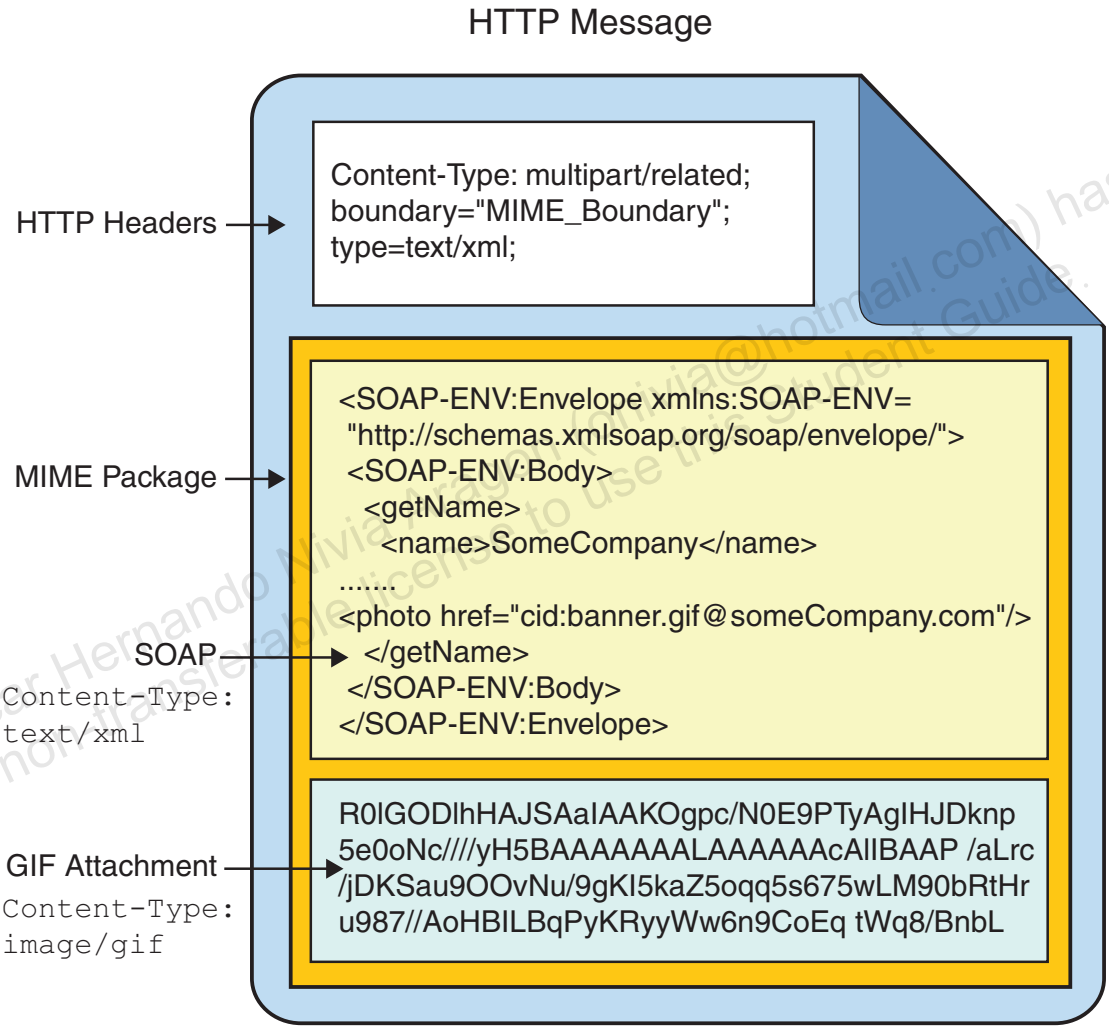


Figure 3.6: SOAP Message embedded in HTTP

```
1  POST http://localhost:8080/airportManager HTTP/1.1
2  Accept-Encoding: gzip,deflate
3  Content-Type: text/xml;charset=UTF-8
4  SOAPAction: ""
5  Content-Length: 346
6
7  <S:Envelope xmlns:ser="http://server.jaxws.example.com/"
8    xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
9     <S:Header/>
10    <S:Body>
11       <ser:addAirport>
12          <arg0>JFK</arg0>
13          <arg1>New York Kennedy Airport</arg1>
14       </ser:addAirport>
15    </S:Body>
16 </S:Envelope>
```

Figure 3.7: SOAP Request Embedded in HTTP

Figure 3.7 shows the sample SOAP request message shown before, but this time embedded inside an HTTP request. Note how some of the information associated with the request (like the target of the request) is captured in the HTTP header section (the argument to `POST`, in line 1), rather than inside the SOAP envelope itself.

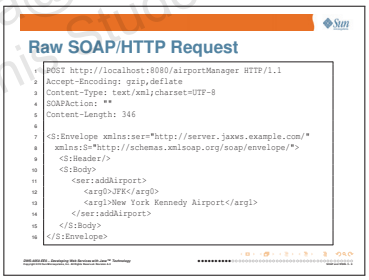
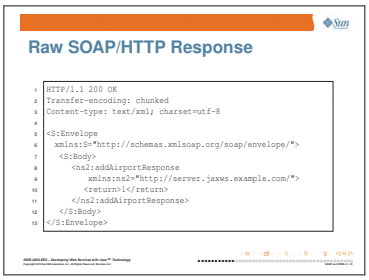


Figure 3.8 shows the embedded HTTP response.



SOAP Over SMTP

The standard protocols for electronic mail can combine to effectively pass SOAP messages. These protocols include the following:

- SMTP
- Post Office Protocol (POP)
- Internet Message Access Protocol (IMAP)

Using mail protocols to transmit SOAP messages offers a slightly different feature set than transmitting SOAP messages using HTTP:

```
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: text/xml; charset=utf-8

<S:Envelope
  xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:addAirportResponse
          xmlns:ns2="http://server.jaxws.example.com/">
         <return>1</return>
      </ns2:addAirportResponse>
   </S:Body>
</S:Envelope>
```

Figure 3.8: SOAP Response Embedded in HTTP

- Mail protocols are naturally asynchronous, supporting a one-way messaging style.
- Mail protocols use a mailbox model instead of connecting two live endpoints.
- Mail protocols can be used to multicast messages.
- Though asynchronous, mail protocols offer reliable messaging, including retries and notices of delivery failure.
- Like HTTP, mail protocols are scalable and robust, support attachments, and have extensible headers.

SOAP Over FTP

FTP is another option for transporting SOAP messages. FTP has the following features and characteristics.

- FTP communications are naturally one-way.
- Unlike HTTP, FTP uses a publish-and-subscribe model.
- FTP uses a fine-grained authentication capability based on resources.

FTP can be useful for simple one-way messaging using web services. Access to FTP servers can be authenticated. Further, role-based restrictions can be applied to particular directories on the FTP server. When using FTP, SOAP messages are mapped onto the files that are being transferred. Typically, the file names indicate the target of the SOAP message.

WSDL

A web service description can be explained as:

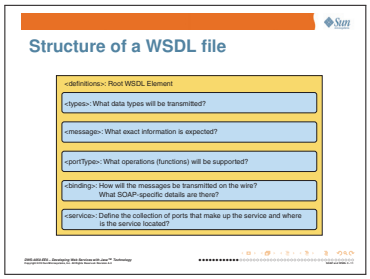
- Containing a set of specifications for the service requester
- Containing a formal mechanism to let the service requester know what has to be done to invoke the web service
- Containing exactly what message format needs to be delivered to what network address in order to invoke a web service
- Enabling software tool developers to provide tools to automate the development of code to invoke a web service

A web service client developer can leverage the benefits of a standard service description format to simplify the development task and generate more robust and reliable code. A standard IDL for web services provides the following three primary benefits:

- Validation of message content at runtime - Validation can save both the service and the client time and effort, by pushing errors to the front of the process and by producing error messages about invalid content.
- Dynamic service invocation at runtime – SOAP intermediaries, such as firewalls, resource managers, filters, and routers, can act intelligently on messages if they are given clear metadata about the message format and contents.
- Automatic code generation during the development phase – Generating code, during the development phase typically reduces maintenance. The generated code could be static stubs and skeletons, and other dedicated components that support or use a web service.

A standard IDL for web services can help facilitate all of these web service functions. An IDL needs to go beyond a schema language, however, and needs to recognize the patterns prevalent in SOAP messaging, common structures, and behaviors. For example, an IDL needs to identify which XML elements are method names and which are arguments to methods or return values. The IDL also needs to organize request and response messages into an entity similar to a method signature on a Java class.

Primary Elements Contained in a WSDL File

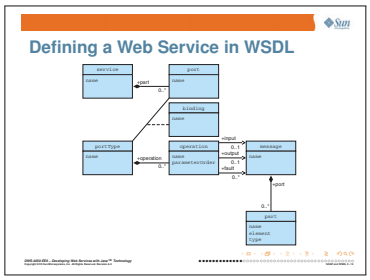


WSDL is designed to serve as a standard mechanism for defining the functionality exposed by a web service. WSDL is an XML vocabulary for describing web services. A WSDL descriptor defines the following elements:

- SOAP message types and their valid content, including supporting data types – Details of the data formats and protocols necessary to access the service's operations.
- Operations that can include one or more possible message types in a scenario, such as one message type for a request and one for a response.
- Details of the protocol-specific network address, such as URL.
- Services themselves, as components that support one or more operations at defined endpoints (URIs). Operations are further grouped into port types that are roughly analogous to interfaces in other DOC models and some programming languages.

A WSDL descriptor defines:

- Two models that can be used to generate or to validate messages.
- Provides an abstract model of messages, port types, operations, and so forth
- Provides a concrete model that binds the abstract model to a specific messaging protocol, such as SOAP
- A target namespace, just like an XML schema. Definitions in the rest of the descriptor populate that namespace, much like XML schema components.



A WSDL document is used to describe a web service using standardized XML. The elements of WSDL are shown in Figure 3.9, while Figure 3.10 shows the relationship between WDSL elements in a UML diagram..

The elements in a WSDL description include:

`definitions` Container for a service description. All global declaration of namespaces that should be in the document are done here.

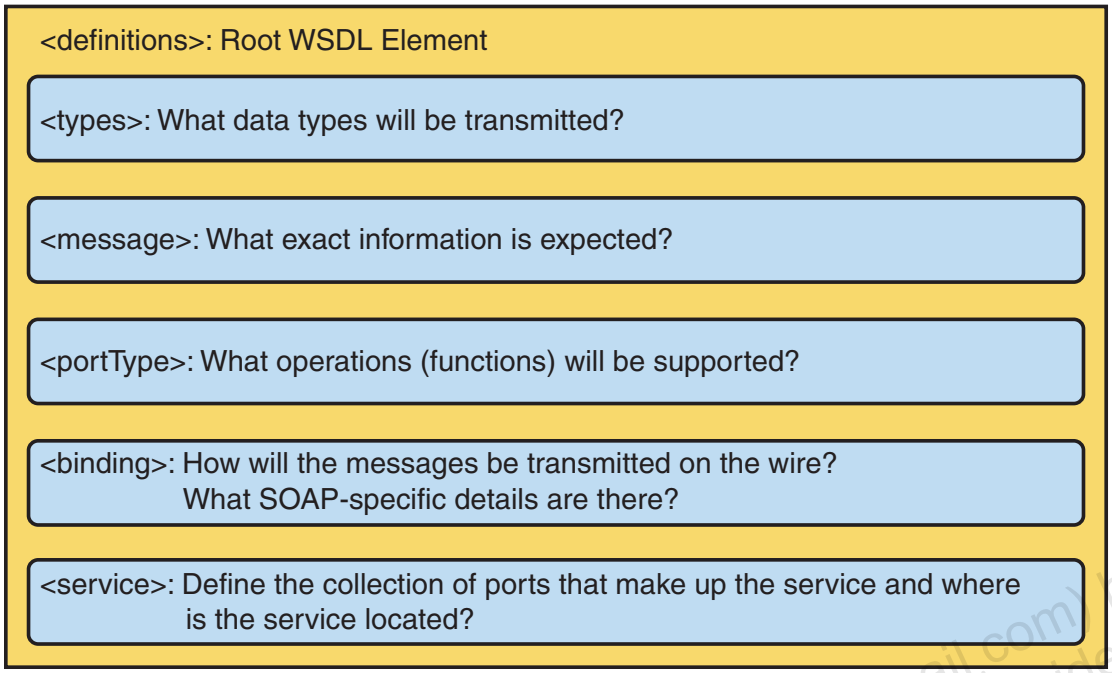


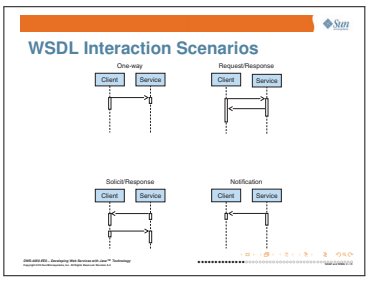
Figure 3.9: Structure of a WSDL File

`types` Holds one or more schema subelements that follow the syntax of a schema document. The data types to be used under the `message` element are defined.

`message` Used to explain the data that is exchanged between a web service and a client and refers to the content in the `types` element. Each `message` element has one or more `part` subelements that define the individual piece of data within the message.

`portType` Specifies the set of actions or methods provided by a single endpoint of a web service. This is similar to providing method signatures. For each of the methods, you have an `operation` element within the `portType` element describing and uniquely identifying each method of the web service. Each operation element has `input` and `output` subelements describing the input value and response value respectively.

The value of `input` and `output` is dependent on the messaging mechanism of the operations. There are fours kinds of interaction discussed in WSDL 1.0 (and the number goes up to 8 in WSDL 2.0):



- Request-response – The client sends a request and expects a synchronous response. Both `input` and `output` elements are defined.

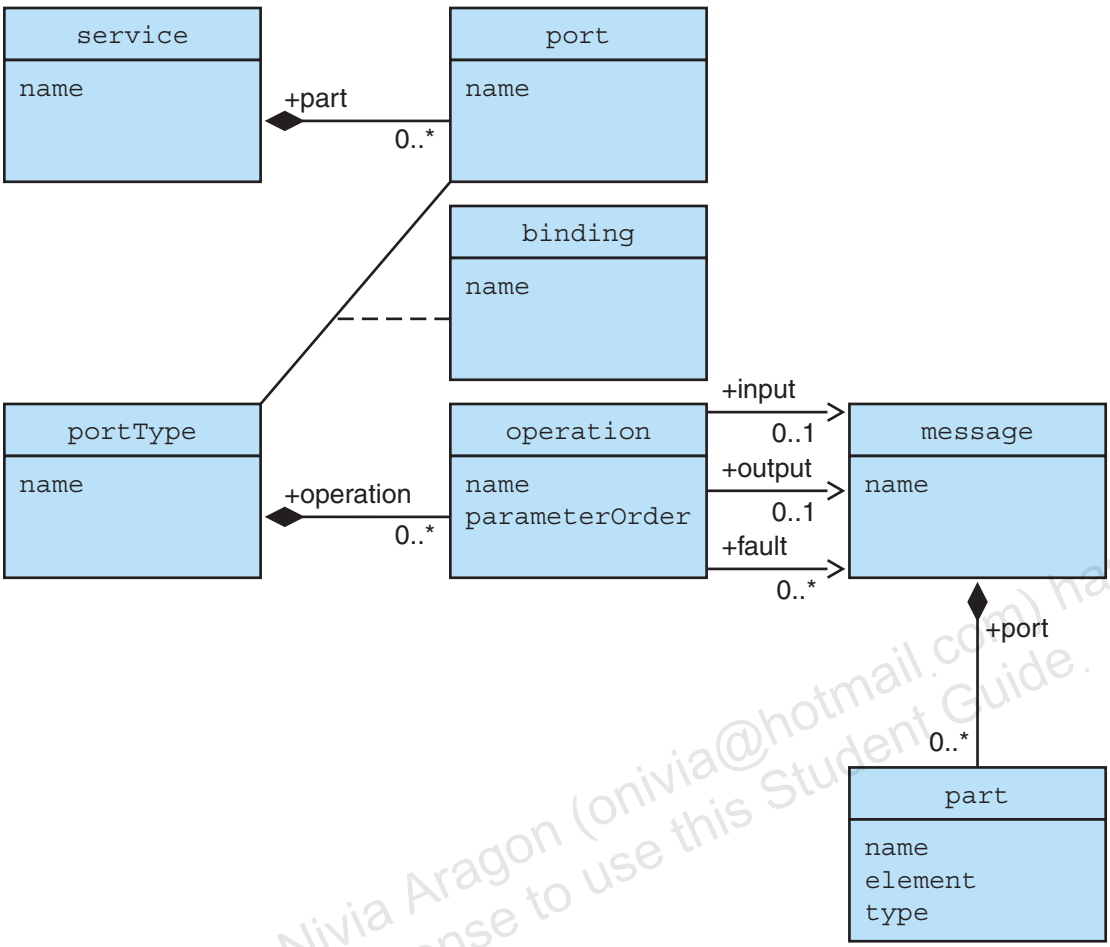


Figure 3.10: WSDL Elements as UML

- Solicit-response – The web service solicits a response from the client, expecting a response. Both `input` and `output` elements are used, but the `output` (solicit) element is placed before the `input` (response) element.
- One-way – This is a one-way invocation with no response. The `input` element is present but no `output` element.
- Notification – The web service sends a one-way message to the client without expecting a response. The `output` element is present but no `input` element.

Figure 3.11 shows the characteristics of each messaging scenario as defined by WSDL.

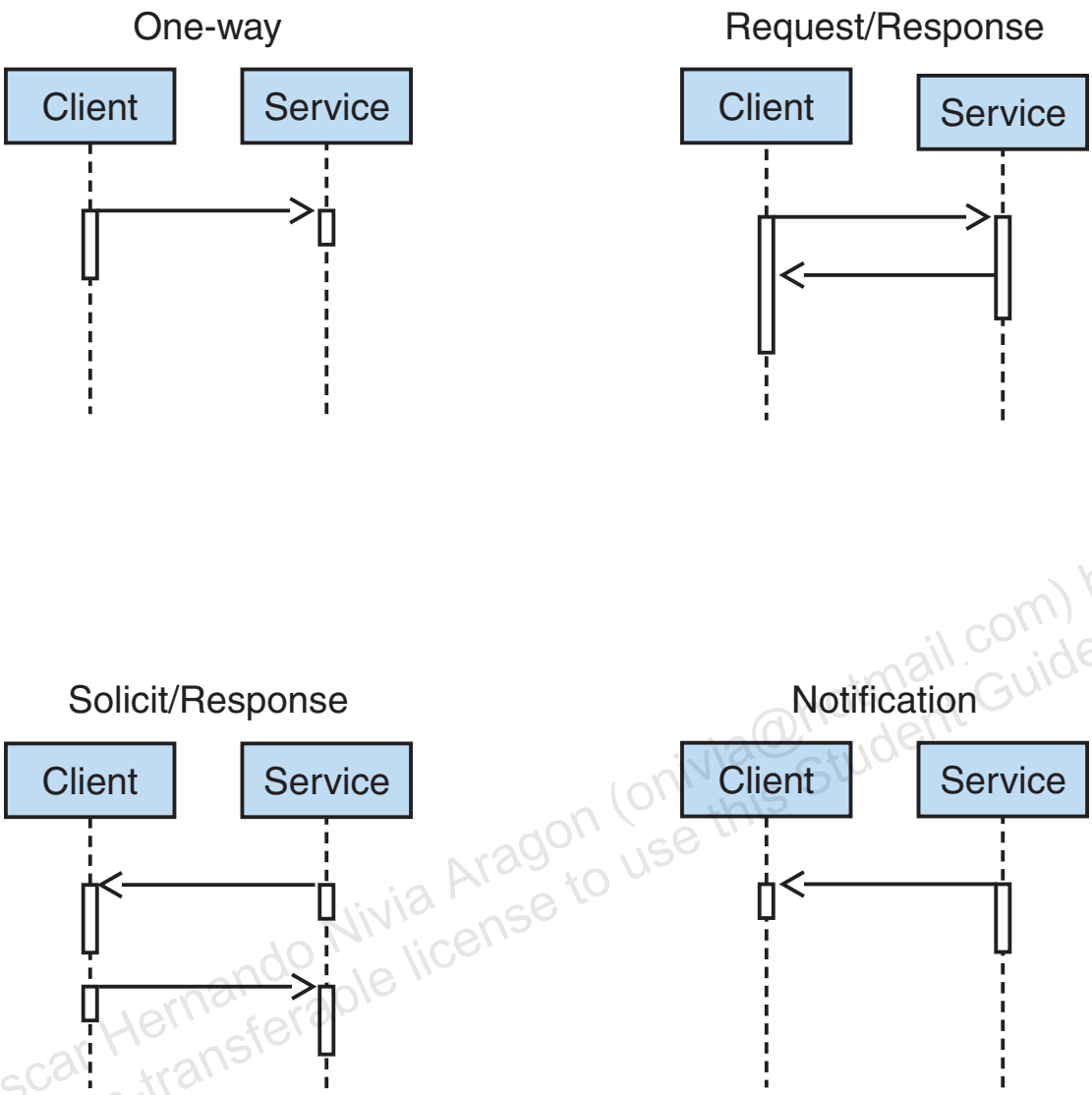
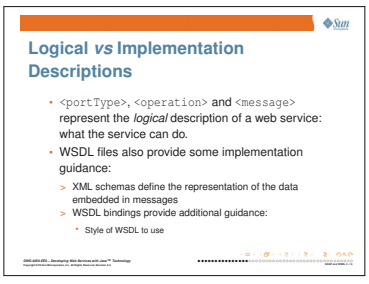


Figure 3.11: WSDL Interaction Scenarios

WSDL identifies a task that a service might provide for a client using the `operation` element. The WSDL `operation` element:



- Is analogous to the method or member function in object-oriented programming languages.
- Identifies a different `message` element for each possible message in the invocation. Each component message is given a designation as either `input`, `output`, or `fault`.

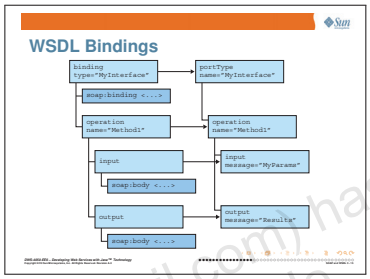
Thus, `<portType>`, `<operation>` and `<message>` represent the *logical* description

of a web service: what the service can do.

WSDL files also provide some implementation guidance:

- XML schemas define the representation of the data embedded in messages
- WSDL bindings provide additional guidance: style of WSDL to use

The `binding` element is a concrete protocol and a data format specification for the `portType` element. You provide one of the standard binding extensions: HTTP, SOAP, MIME, or create one of your own. Each protocol has its own format. For example, HTTP has a simple header/body format, and SOAP, which exists inside HTTP, has its own standard format. Figure 3.12 illustrates the structure of the `binding` element.



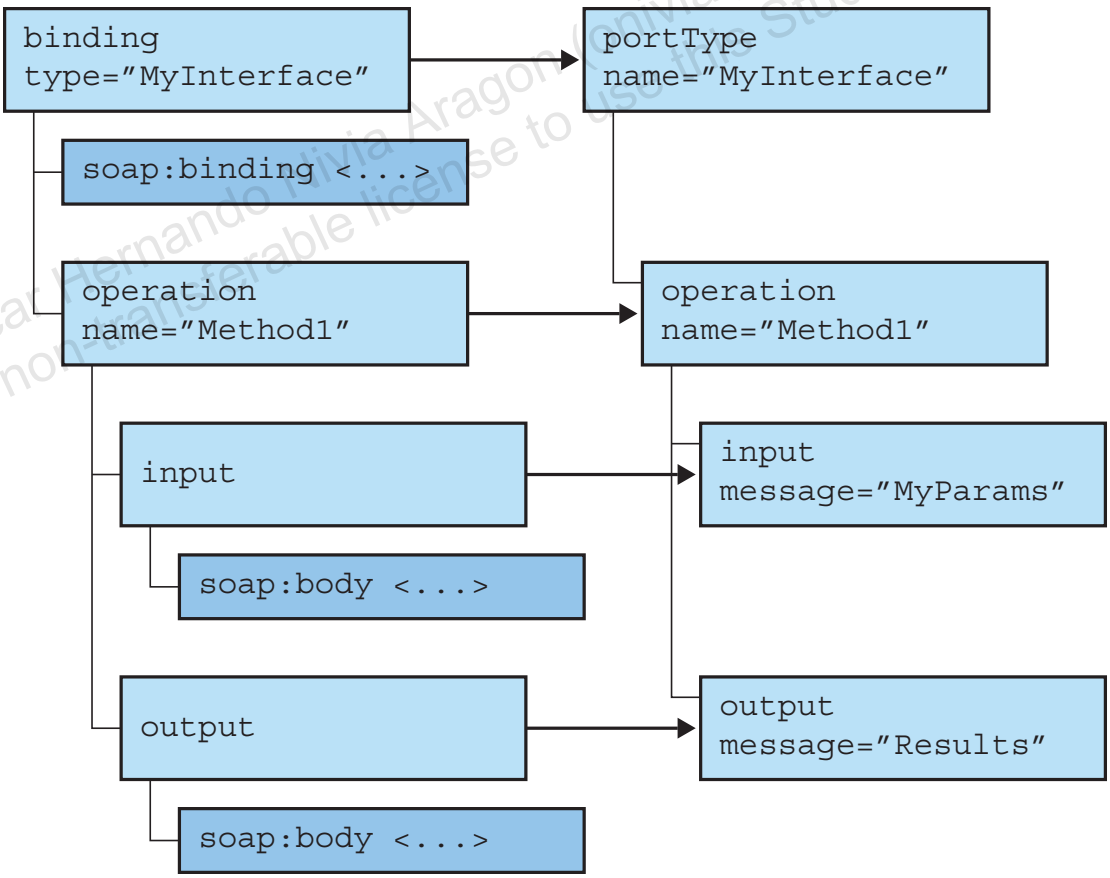


Figure 3.12: WSDL `binding` Element

The `binding` component has the specific purpose of extending the abstract service model. The `binding` element can be referenced by a `port` element which, in turn, references a specific `portType`. In this way, the `binding` element connects the two, and adds protocol-specific information. The binding is the primary means of extension, but not the only one. Extensibility elements can appear elsewhere as well.

Figure 3.12 shows that the `binding` element's child elements are operations. These do not define new operations for the port type. Rather, each child element contains enough information to precisely identify an operation already defined in the port type. These child elements are then combined with extensibility elements to add information specific to a target protocol.

For SOAP-based services, the WSDL descriptor includes elements specific to SOAP messaging. These elements bind the abstract semantic definitions described thus far to a concrete SOAP implementation. Extensibility elements in this binding include the following:

- The `soap:binding` element extends the abstract binding by defining the specific SOAP transport, such as SOAP over HTTP, and the messaging style as either document or RPC. Other, more specific extensibility elements can also define messaging style, overriding anything defined at a higher level.
- The `soap:operation` element extends the abstract operation with attributes, such as the related SOAP action URI.
- The `soap:body` element extends an abstract message type by defining encoded or literal use, encoding rules, and other attributes.
- The `soap:fault` element extends abstract fault messages the same way the `soap:body` element extends input and output messages.
- The `soap:address` element can be used to assign a real URI to an abstract component, such as a service.

The `service` element describes a service as a collection of one or more `port` elements. Each `port` element provides the URL location of the service endpoint.

The `port` element has a `name` attribute that provides a unique identifier for this endpoint. It also contains a `binding` attribute that references the name of a binding element contained within the same WSDL document or in an imported one.

The `port` element has a `soap:address` subelement. The `soap:address` element identifies the URL of the web service. If this service used a different binding extension, this element would be different as well. There can be a single web service that is accessed through multiple URLs.

Variations of WSDL

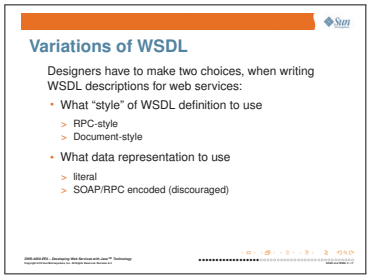
We have seen several examples of SOAP messages, like the one shown in Figure 3.13. We just discussed that a web service must have a description of what the service can do, and of how clients interact with the service, written in terms of WSDL.

One of the responsibilities of the WSDL description of a service is to state what the expected structure for SOAP requests and responses is supposed to be. Should the web service accept more than one kind of structure for its SOAP messages? Even if there is only one structure that would be valid – is there more than one way to describe these SOAP messages?

```
<S:Envelope
    xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:ser="http://server.jaxws.example.com/">
   <S:Header/>
   <S:Body>
      <ser:addAirport>
         <code>JFK</code>
         <name>New York John F. Kennedy Airport</name>
      </ser:addAirport>
   </S:Body>
</S:Envelope>
```

Figure 3.13: Sample SOAP Message

You can characterize a web service by the data or information model that the service implements. The data or information model describes the type of data, contained within the messages, that is passed between a service and service client during a web service interaction. There are two basic information or messaging models: *parameter style* (or RPC style), and *document style*.

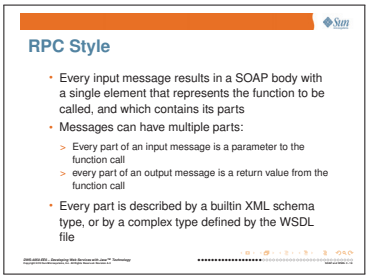


Designers have to make two choices, when writing WSDL descriptions for web services:

- What "style" of WSDL definition to use
  - RPC-style
  - Document-style
- What data representation to use

- literal
- SOAP/RPC encoded (discouraged)

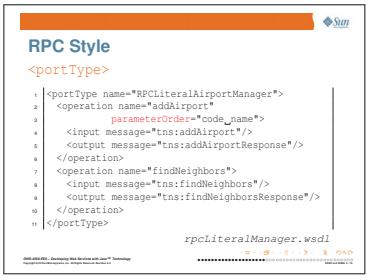
Parameter Style



A parameter-style (also known as RPC-style) web service responds to a request submitted as an XML-formatted procedure call with parameters. The web service executes the procedure call specified by the request and returns the result to the service client. Parameter-style services use SOAP over HTTP as a transport for remote procedure calls and the XML content of the SOAP message tends to follow a more rigid pattern.

- There is just one body entry, identifying the procedure to be invoked.
- The body element has a list of child elements, each of which is an argument to that procedure. Arguments for complex types, if supported, might have child elements that describe the sub-elements of the complex type. Thus, components, methods, and arguments are expressed as service URI, body entry, and child element respectively.
- Every part can be described by a built-in XML schema type, or by a complex type defined by the WSDL file
- The result of an RPC-style service invocation is encoded in a SOAP response in much the same way.

Code 3.1, 3.2, and 3.3 show fragments of a WSDL file that describes a service that uses RPC style. Figure 3.14 shows an example message, given that definition.



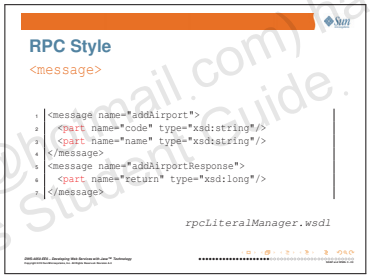
Code 3.1 shows the fragment of the WSDL description for a web service where the `portType` description is found. This description includes two operations, `addAirport` and `findNeighbors`, both based on the request/response message pattern. The description for `addAirport` (lines 2 to 6) includes both request and response messages, described by the two XML `message` elements named.

```
1  <portType name="RPCLiteralAirportManager">
2    <operation name="addAirport"
3                parameterOrder="code␣name">
4      <input message="tns:addAirport"/>
5      <output message="tns:addAirportResponse"/>
6    </operation>
7    <operation name="findNeighbors">
8      <input message="tns:findNeighbors"/>
9      <output message="tns:findNeighborsResponse"/>
10   </operation>
11 </portType>
```

E-18

Code 3.1: WSDL Definition, RPC-Style

Code 3.2 shows the fragment of the WSDL description that describes the two messages associated with the `addAirport` operation. The description for the `addAirport` message (lines 1 to 4) indicates that this message will carry two parameters, corresponding to the new airport's code and name.

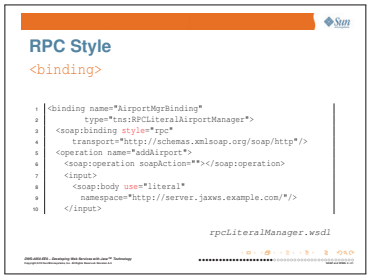



```
1  <message name="addAirport">
2    <part name="code" type="xsd:string"/>
3    <part name="name" type="xsd:string"/>
4  </message>
5  <message name="addAirportResponse">
6    <part name="return" type="xsd:long"/>
7  </message>
```

E-18

Code 3.2: WSDL Message Definition, RPC-Style

Code 3.3 shows the fragments of the WSDL description of the service corresponding to the logical to physical binding. This section specifies, among other things, that the description for this service is using the RPC/literal style (lines 3 and 8) for the `addAirport` operation. This choice must be captured explicitly, so that readers of this description will interpret it correctly.



```
<binding name="AirportMgrBinding"
         type="tns:RPCLiteralAirportManager">
  <soap:binding style="rpc"
      transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="addAirport">
    <soap:operation soapAction=""></soap:operation>
    <input>
      <soap:body use="literal"
        namespace="http://server.jaxws.example.com/"/>
    </input>
```

E-18

Code 3.3: WSDL Bindings Definition, RPC-Style

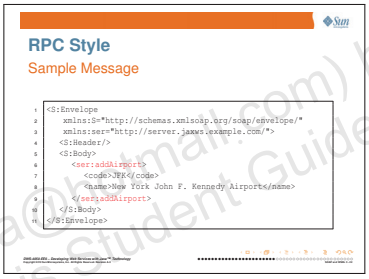


Figure 3.14 shows an example message, given the definition above. Note how the payload for the message consists of an `addAirport` XML element with children for the `code` and `name` of the airport – even though the description for the outgoing request message only mentioned the two parameters. This is due to the `RPC/literal` specification in the `binding` element in the WSDL description: it instructed readers to consider the payload of the message to consist of an XML element that represents the operation itself, which would then include as children the elements required by the `message` description. This requirement that an XML element corresponding to the operation be included was *implicit* in the specification of the `RPC/literal` encoding.

```
<S:Envelope
    xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:ser="http://server.jaxws.example.com/">
   <S:Header/>
   <S:Body>
      <ser:addAirport>
         <code>JFK</code>
         <name>New York John F. Kennedy Airport</name>
      </ser:addAirport>
   </S:Body>
</S:Envelope>
```

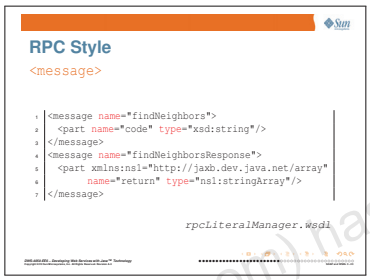
Figure 3.14: Sample SOAP Message, RPC Style

The RPC literal style represents simple data (Java primitives, for example) as string values that are presented as text elements within the XML-based SOAP body. However, the RPC literal style is not limited to such simple data: if a

more complex type needs to be transferred as part of a SOAP message, an XML schema type is added to the WSDL description of the service in order to describe that complex type.

Figure 3.4 shows an example of a WSDL message that includes a more complex type – an array of strings. Figure 3.5 shows an example of the XML schema definition that is introduced to describe that type. Figure 3.15 shows a sample SOAP message that would be generated from this description.

Figure 3.4 shows the two `message` elements of the second operation, `findNeighbors`. This second operation returns a more complex type: an array of strings. When using the RPC/literal style, such complex types must be described in XML schema form, so that the web services runtime can marshall and unmarshall values of those types.




```
<message name="findNeighbors">
  <part name="code" type="xsd:string"/>
</message>
<message name="findNeighborsResponse">
  <part xmlns:ns1="http://jaxb.dev.java.net/array"
        name="return" type="ns1:stringArray"/>
</message>
```

E-18

Code 3.4: WSDL Description of Message with Complex Type

Code 3.5 shows an XML schema type definition that could be used to describe the array of strings returned by the `findNeighbors` operation. Since the application developers define the XML schema type, they control the representation that will be used: in this example, the array of strings is represented in XML by a series of `item` elements, one for each string.

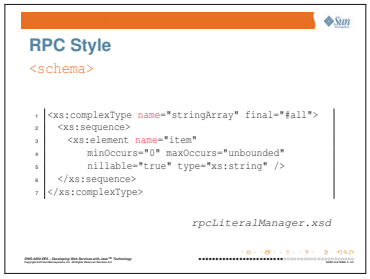
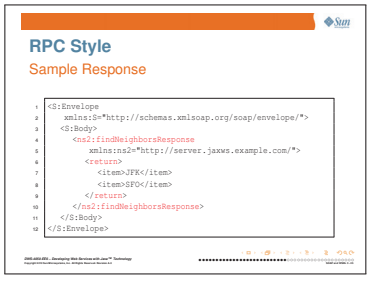


Figure 3.15 shows a sample return message that could be produced as the result of a call on the `findNeighbors` operation.



```
1   <xs:complexType name="stringArray" final="#all">
2     <xs:sequence>
3       <xs:element name="item"
4           minOccurs="0" maxOccurs="unbounded"
5           nillable="true" type="xs:string" />
6     </xs:sequence>
7   </xs:complexType>
```

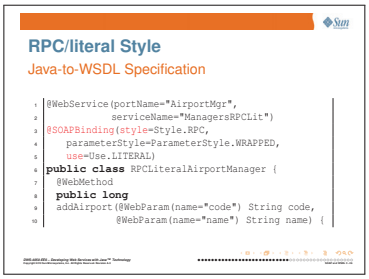
E-20

Code 3.5: WSDL Schema Type for Complex Type

```
1  <S:Envelope
2      xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
3     <S:Body>
4        <ns2:findNeighborsResponse
5            xmlns:ns2="http://server.jaxws.example.com/">
6           <return>
7              <item>JFK</item>
8              <item>SFO</item>
9           </return>
10       </ns2:findNeighborsResponse>
11    </S:Body>
12 </S:Envelope>
```

Figure 3.15: Sample RPC/literal SOAP Message with Complex Value

In all the examples so far, we have seen how developers can specify that a web service use RPC/literal style through its WSDL definition. However, that is not the only way to do so – developers who build web services bottom-up would rather be able to specify this style during the creation of the Java implementation of the web service (since they do not author the WSDL description for their service, at all).

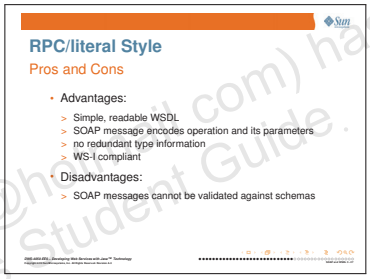


JAX-WS supports specifying the style that the web service will use through annotations on the web service implementation class, as shown in Figure 3.6.

```
1  @WebService(portName="AirportMgr",
2              serviceName="ManagersRPCLit")
3  @SOAPBinding(style=Style.RPC,
4      parameterStyle=ParameterStyle.WRAPPED,
5      use=Use.LITERAL)
6  public class RPCLiteralAirportManager {
7    @WebMethod
8    public long
9    addAirport(@WebParam(name="code") String code,
10              @WebParam(name="name") String name) {
```

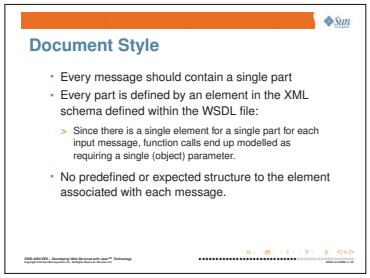
E-21

Code 3.6: Specifying RPC/literal in Java



- Advantages:
  - Simple, readable WSDL
  - SOAP message encodes operation and its parameters
  - no redundant type information
  - WS-I compliant
- Disadvantages:
  - SOAP messages cannot be validated against schemas

Document Style



A document-style service uses the information contained within a document to determine how to process a specific request. For example, a travel booking service client might present a user with a form that collects trip information from the user, such as start and end dates, destination, itinerary, method of travel, and lodging preferences. When the client submits this information in document form to the booking service, the service determines how to process the request by parsing the document and interpreting some or all of the request information.

Document-style messaging includes the message content using XML documents and XML fragments passed over HTTP:

- The XML content of a document-style service might contain a complete or partial result set from a database query, information on a product order, configuration information for an out-of-band communication, and so forth.
- The content is essentially free-form XML.

Document-style web services are loosely coupled and document-driven. The client sends the parameter to the web service as an XML document, instead of a discrete set of parameter values to the web service. The input XML document can also be described in WSDL. Unlike RPC-style web service, document-style web services need not follow call/response semantics. The web service receives the entire document, processes it, and might not return a response message. The SOAP body of a document style carries one or more XML documents within its body. The protocol places no constraint on how that document needs to be structured, which is handled at the application level. A document-style web service can provide asynchronous processing.

Code 3.7, 3.8, 3.9, and 3.10 show the description of a document-styled web service.

The logical description of a document-style web service can be found in Code 3.7. Note that, at this level, there is no difference between this description, and the RPC/literal description of the web service shown above.


```
1  <portType name="DocumentLiteralAirportManager">
2    <operation name="addAirport">
3      <input message="tns:addAirport"/>
4      <output message="tns:addAirportResponse"/>
5    </operation>
6    <operation name="findNeighbors">
7      <input message="tns:findNeighbors/">
8      <output message="tns:findNeighborsResponse"/>
9    </operation>
10 </portType>
```

E-23

Code 3.7: WSDL Description of Document-style Service

Code 3.8 shows the description of the request/response messages required for each of the two operations `addAirport` and `findNeighbors`. Note how the description for the request message for the `addAirport` operation (lines 1 to 3) does differ from the RPC/literal description shown before: the document-style description for this request message consists of a single part, of XML schema type `tns:airport`.


```
1  <message name="addAirport">
2    <part name="airport" element="tns:airport"/>
3  </message>
4  <message name="addAirportResponse">
5    <part name="addAirportResponse"
6          element="tns:addAirportResponse"/>
7  </message>
8  <message name="findNeighbors">
9    <part name="code" element="tns:code"/>
10 </message>
11 <message name="findNeighborsResponse">
12   <part name="findNeighborsResponse"
13         element="tns:findNeighborsResponse"/>
14 </message>
```

Code 3.8: Message Description in Document-style Service

The message description can be written in terms of a single part because the XML schema type specified, shown in Code 3.9 (lines 7 to 14) is itself a complex type: it describes that there will be two elements, `code` and `name`, both strings. The application developer has complete control over the content to be embedded as the body of the SOAP envelope.

The WSDL description must capture the requirement that this description must be understood according to the document/literal style. This is described in the `binding` section of the WSDL description, as show in Code 3.10.

```
<xs:element name="addAirportResponse" type="xs:long"/>
<xs:element name="airport" nillable="true"
    type="tns:airport"/>
<xs:element name="code" nillable="true" type="xs:string"/>
<xs:element name="findNeighborsResponse" nillable="true"
    type="ns1:stringArray"/>
<xs:complexType name="airport">
 <xs:complexContent>
  <xs:sequence>
    <xs:element name="code" type="xs:string" minOccurs="0"/>
    <xs:element name="name" type="xs:string" minOccurs="0"/>
  </xs:sequence>
 </xs:complexContent>
</xs:complexType>
```

E-25

Code 3.9: XML Schema Type for Document-style Service


```
<binding name="AirportMgrBinding"
         type="tns:DocumentLiteralAirportManager">
  <soap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="addAirport">
    <soap:operation soapAction=""/>
    <input><soap:body use="literal"/></input>
    <output><soap:body use="literal"/></output>
  </operation>
  <operation name="findNeighbors">
    <soap:operation soapAction=""/>
    <input><soap:body use="literal"/></input>
    <output><soap:body use="literal"/></output>
  </operation>
</binding>
```

E-23

Code 3.10: WSDL Bindings for Document-style Service

```
<S:Envelope xmlns:ser="http://server.jaxws.example.com/"
    xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Header/>
   <S:Body>
      <ser:airport>
         <code>LGA</code>
         <name>New York La Guardia Airport</name>
      </ser:airport>
   </S:Body>
</S:Envelope>
```

Figure 3.16: Sample Message for Document-style Service

Figure 3.16 shows a sample message, given this definition. Note how the message obeys the WSDL description of this message to the letter: only the information explicitly mentioned in the XML schema type for the request message is included in the body of this request. As a result, it may impossible to determine which method the client meant to invoke, given just the information in the message.

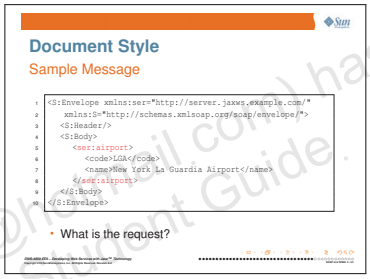


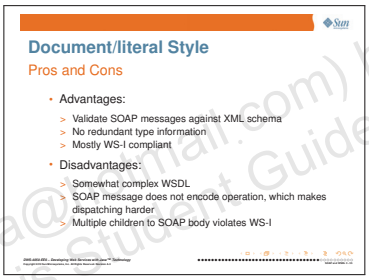
Figure 3.11 shows how a Java programmer could specify that a web service use the document/literal style, when building the web service bottom-up. Note how, in addition to the two annotations values that indicate that this web service should use (i) document style, and (ii) literal, there is a third value that is set in the example to `ParameterStyle.BARE`.

We saw earlier how the document-style literal can make it challenging to figure out what method the client mean to invoke. The value `BARE` states that this (original) style is to be used, even so. However, this issue was significant enough that a second option can be requested, using the value `WRAPPED` instead. We will discuss this other option below.

```
1  @WebService(portName="DocLiteralAirportMgr",
2              serviceName="ManagersDoc")
3  @SOAPBinding(style=Style.DOCUMENT,
4      parameterStyle=ParameterStyle.BARE,
5      use=Use.LITERAL)
6  public class DocumentLiteralAirportManager {
7    @WebMethod
8    public long
9    addAirport(@WebParam(name="airport")
10                Airport airport ) {
```

E-26

Code 3.11: Java-based Specification of Document/literal Style



Developers need to consider the advantages and disadvantages of the different styles that can be used, when defining web services. Factors to consider about the document style include:

- In document style, the benefits of XML can be used to describe an important corporate-level document. In RPC, you use XML to encode the method calls and parameters; therefore, there is a limitation in incorporating all the advanced business rules.
- Document style does not follow strict rules. Because there are no specified rules, you can modify and enhance the XML schema without producing an impact on the client application. In RPC, any change in any method greatly impacts all clients consuming the RPC service.
- Document style is suitable for asynchronous messaging for reliable and guaranteed delivery, scalability, and better performance.
- Document-style makes object exchange flexible. There are no constraints on the design of the object being exchanged between the server and client. Both ends can design, implement, and process the objects independent of any encoding scheme.

In general, advantages of the document/literal approach include:

- Validate SOAP messages against XML schema
- No redundant type information
- Mostly WS-I compliant

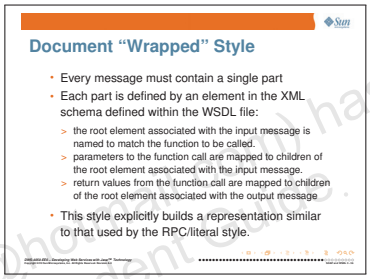
Disadvantages of the document/literal approach include:

- Somewhat complex WSDL
- SOAP message does not encode operation, which makes dispatching harder
- Multiple children to SOAP body violates WS-I

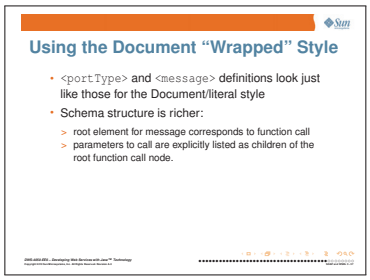
The document-literal "wrapped" style was introduced to address these shortcomings.

Document "Wrapped" Style

The document/literal "wrapped" style aims to combine the best of the rpc/literal and document/literal styles, without sharing any of their drawbacks. In the document/literal "wrapped" style:



- Every message must contain a single part
- Each part is defined by an element in the XML schema defined within the WSDL file:
  - the root element associated with the input message is named to match the function to be called.
  - parameters to the function call are mapped to children of the root element associated with the input message.
  - return values from the function call are mapped to children of the root element associated with the output message
- This style explicitly builds a representation similar to that used by the RPC/literal style.
- `<portType>` and `<message>` definitions look just like those for the Document/literal style
- Schema structure is richer:
  - root element for message corresponds to function call
  - parameters to call are explicitly listed as children of the root function call node.



Code 3.12, 3.13, and 3.14 show fragments of the WSDL description for a web service that uses the document/literal "wrapped" style.

The overall logical description (the `portType`) for a document/literal "wrapper" description is no different than the document/literal "bare" one – both describe operations in terms of messages, which are in turn defined by XML schema types. The difference between the two approaches lies in the detail captured in their respective XML schema types. Code 3.12 shows the descriptions for the two messages associated with the `addAirport` operation, when using the document/literal "wrapped" style. Note that they are exactly the same as for the "bare" description.

```
1 <message name="addAirport">
2   <part name="parameters"
3   element="tns:addAirport"/>
4 </message>
5 <message name="addAirportResponse">
6   <part name="parameters"
7   element="tns:addAirportResponse"/>
8 </message>
```

E-28

Code 3.12: WSDL Message Description Using Document/literal "wrapped"

Code 3.13 shows the XML schema types used for the two messages for the `addAirport` operation. Note how the XML schema definition for the request message is written in terms of an XML element (`addAirport`, in line 1) whose content is defined by the XML schema type in lines 4 to 9). As a result, the SOAP body will contain a single element corresponding to the operation intended (`addAirport`, in this case), with children corresponding to the parameters to the request.

Just as before, the WSDL description for the web service must specify the style used within it, so that readers can interpret the description appropriately. There is a catch, however: the document/literal "wrapped" style is not contemplated in the WSDL 1.1 specification. To specify that this is the style in user, the developer must fall back on a JAX-WS extension to standard WSDL, as shown in lines 5 to 8 of Code 3.14. Without using this extension, the WSDL description would be just that of the document/literal "bare" style.

```
<xs:element name="addAirport" type="tns:addAirport"/>
<xs:element name="addAirportResponse"
  type="tns:addAirportResponse"/>
<xs:complexType name="addAirport">
  <xs:sequence>
    <xs:element name="code" type="xs:string"
                minOccurs="0"/>
    <xs:element name="name" type="xs:string"
                minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="addAirportResponse">
  <xs:sequence>
    <xs:element name="return" type="xs:long"/>
  </xs:sequence>
</xs:complexType>
```

E-30

Code 3.13: XML Schema for Web Service Using Document/literal "wrapped"


```
<binding name="WrappedAirportManagerPortBinding"
   type="tns:AirportManager">
  <soap:binding style="document"
     transport="http://schemas.xmlsoap.org/soap/http"/>
  <jaxws:bindings
    xmlns:jaxws="http://java.sun.com/xml/ns/jaxws">
    <jaxws:enableWrapperStyle>
      true
    </jaxws:enableWrapperStyle>
  </jaxws:bindings>
  <operation name="addAirport">
    <soap:operation soapAction=""/>
    <input><soap:body use="literal"/></input>
    <output><soap:body use="literal"/></output>
  </operation>
</binding>
```

E-28

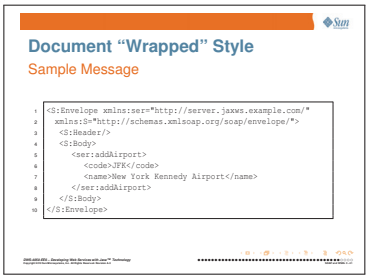
Code 3.14: WSDL Bindings Using Document/literal "wrapped"

```
<S:Envelope xmlns:ser="http://server.jaxws.example.com/"
  xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Header/>
   <S:Body>
      <ser:addAirport>
         <code>JFK</code>
         <name>New York Kennedy Airport</name>
      </ser:addAirport>
   </S:Body>
</S:Envelope>
```

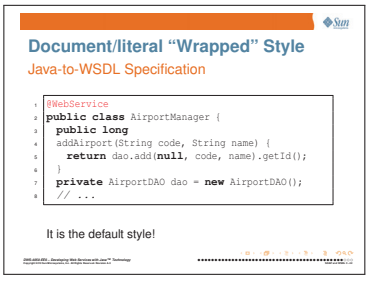
Figure 3.17: Sample Message Using Document/literal "wrapped"

Note that, technically speaking, the document/literal "wrapped" style is really just a specific example of the document/literal style: it is a description that captured the content of the SOAP body in terms of specific XML schema types. It just so happens that this XML schema type structure explicitly includes an element corresponding to the operation to invoke.

Figure 3.17 shows a sample message generated based on this description. Note that the actual payload of a web service request described according to the document/literal "wrapped" style may be indistinguishable from the payload that would have been produced had the developer used the RPC/literal style, instead. The "wrapped" style simply captures *explicitly* the effect that the RPC/literal style will achieve *implicitly*.



It is also possible to build a web service bottom-up, and yet use the document/literal "wrapped" style. Figure 3.15 shows an example of the Java implementation of a web service provider defined bottom-up. This example will result in a web service that uses the "wrapped" style – even though there is nothing to declare that it should be so in the source code. In fact, document/literal "wrapped" style is the default style for JAX-WS.



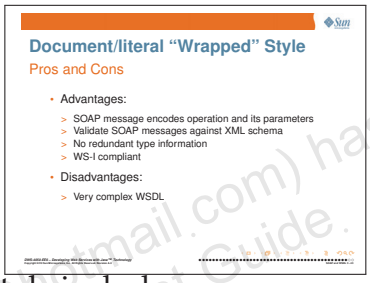
Advantages of the document/literal "wrapped" style include:

```
1 @WebService
2 public class AirportManager {
3   public long
4   addAirport(String code, String name) {
5     return dao.add(null, code, name).getId();
6   }
7   private AirportDAO dao = new AirportDAO();
8   // ...
```

E-2

Code 3.15: Java-to-WSDL Using Document/literal "wrapped"

- SOAP message encodes operation and its parameters
- Validate SOAP messages against XML schema
- No redundant type information
- WS-I compliant

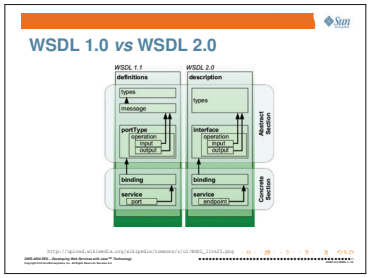


Disadvantages of the document/literal "wrapped" style include:

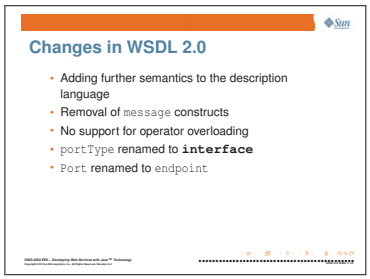
- Very complex WSDL

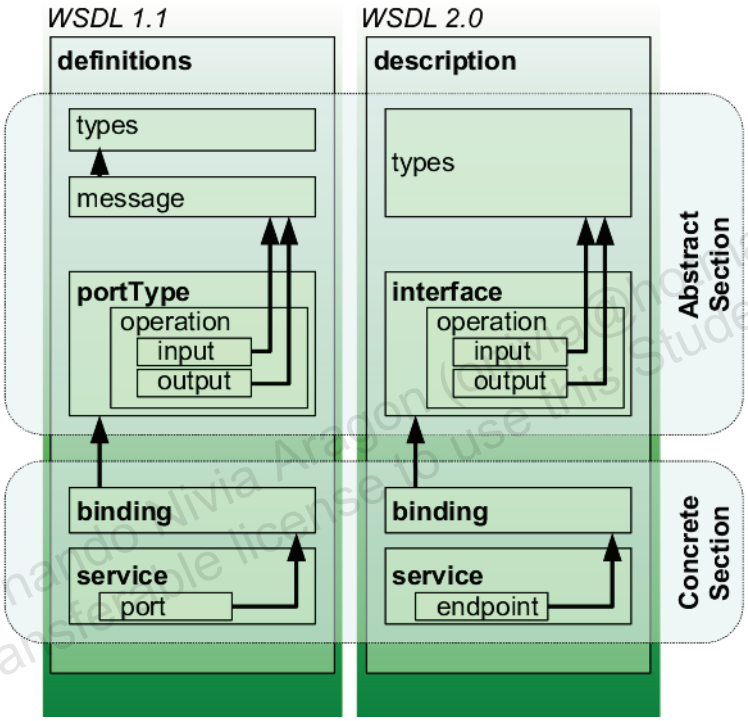
Evolution of WSDL

The specification for the XML vocabulary that the web services standards use to describe web services has already reached version 2.0. In our course, we still discuss WSDL 1.1, since that is the level of the specification supported by most web service infrastructures (such as JAX-WS) – but a quick mention of the direction in which WSDL is evolving is in order. Figure 3.18 illustrates the differences between the WSDL 1.x and the WSDL 2.0 specifications. The differences between the two specs include:



- Adding further semantics to the description language
- Removal of `message` constructs
- No support for operator overloading
- `portType` renamed to `interface`
- `Port` renamed to `endpoint`





http://upload.wikimedia.org/wikipedia/commons/c/c2/WSDL_11vs20.png

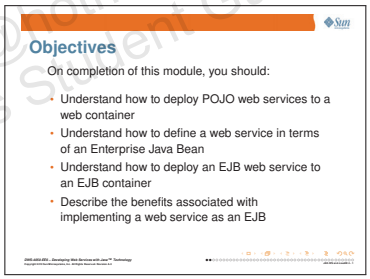
Figure 3.18: Representation of concepts defined by WSDL 1.1 and WSDL 2.0 documents

Chapter 4

JAX-WS and JavaEE

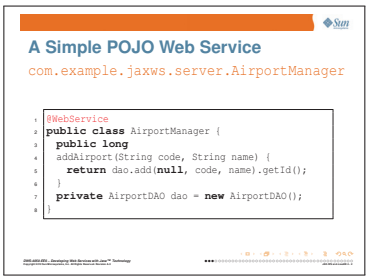
On completion of this module, you should:

- Understand how to deploy POJO web services to a web container
- Understand how to define a web service in terms of an Enterprise Java Bean
- Understand how to deploy an EJB web service to an EJB container
- Describe the benefits associated with implementing a web service as an EJB

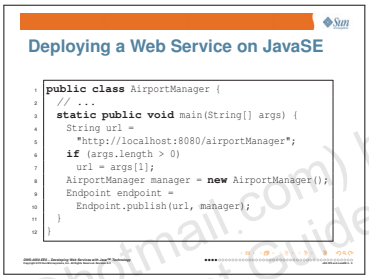


Deploying a Web Service to a Web Container

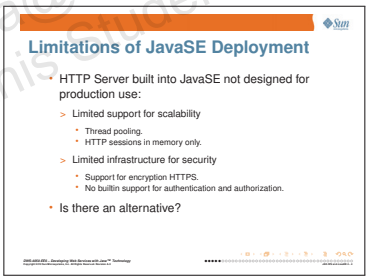
In Chapter 2, we saw what a very simple JAX-WS-based web service looks like (see Code 2.2).



Also in Chapter 2, we saw how a JAX-WS web service could be deployed without too much effort to the web server built into JavaSE 6 (see Code 2.22).

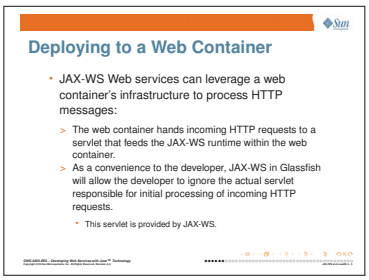


Although enough to deploy JAX-WS-based web services on, the HTTP Server built into JavaSE was not designed for production use; it was only intended for limited-scale testing of web resources (including JAX-WS service providers). Among the limitations associated with that web server, we find:



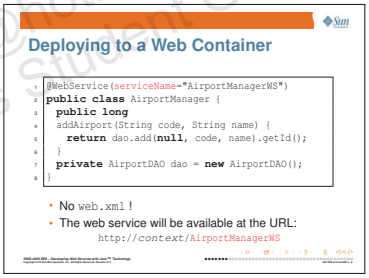
- It provides limited support for scalability.
  Production web servers provide features to enable them to handle high client loads, like thread pooling, to minimize the amount of effort it takes to run concurrent requests, and mechanisms for "passivating" and "activating" HTTP sessions, to avoid the penalty of keeping all sessions in memory, just in case they are used once again.
- It provides a limited infrastructure for security.
  Production web containers offer a number of features to support authentication and authorization constraints on web resources. Although it provides support to securing channels via HTTPS, the HTTP Server built into Java does not provide other features typical of production web servers, such as a variety of authentication schemes, or authorization constraints.

JAX-WS Web services can take advantage of a web container's infrastructure to process web services HTTP messages. To plug into any web container, JAX-WS includes a `Servlet` that acts as the entry point from the web container to the JAX-WS runtime:



- The web container hands incoming HTTP requests to the JAX-WS servlet, which feeds the request into the JAX-WS runtime.
- The response produced by the JAX-WS-based web service is return to the JAX-WS, which lets the JAX-WS servlet package the response into the body of the HTTP response, before forwarding it to the client.
- As a convenience to the developer, JAX-WS in GlassFish will allow the developer to ignore the actual servlet responsible for initial processing of incoming HTTP requests.

Code 4.1 shows a simple JAX-WS-based service that can be deployed to a web container. In general, nothing new needed to be done – a POJO JAX-WS web service can be deployed to a web container just as it can be deployed to the built-in web server within JavaSE 6. The machinery to support this is all provided by the JAX-WS runtime.



When deploying JAX-WS-based web services to a web container, no `web.xml` configuration need be specified:

- In JavaEE6, web containers support `web.xml` *fragments*, which can be used to deploy web components completely transparently to a web application. Prior to JavaEE 6, it might be necessary to add the names of the JAX-WS-based web services to the `web.xml` configuration – but they are listed as servlets, even though they are not, to make their deployment as convenient as possible.
- Web services are deployed so that they are reachable via URIs based on their *service name*. The web service shown in Code 4.1 will be available at the URL:

  `http://context/`**`AirportManagerWS`**

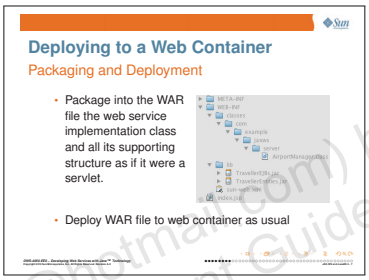
When deploying JAX-WS-based web services into a web container, the JAX-WS-based web services are just packaged into a web application to be deployed into that web container. Once deployed, they are available.

```
1  @WebService(serviceName="AirportManagerWS")
2  public class AirportManager {
3    public long
4    addAirport(String code, String name) {
5      return dao.add(null, code, name).getId();
6    }
7    private AirportDAO dao = new AirportDAO();
8  }
```

E-31

Code 4.1: Deploying a JAX-WS Service to a Web Container

- Package into the WAR file the web service implementation class and all its supporting structure as if it were a servlet.
- Deploy WAR file to web container as usual



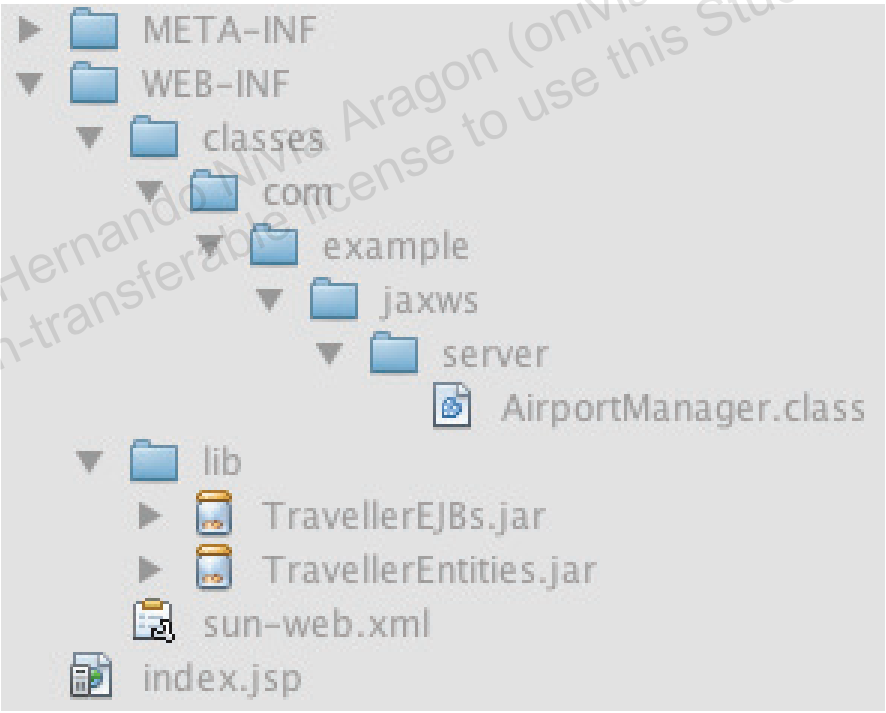
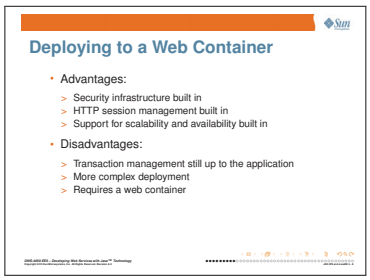


Figure 4.1: Structure of WAR file with a JAX-WS-based web service
fig:Structure of WAR file with a JAX-WS-based web service



- Advantages:
  - Security infrastructure built in
  - HTTP session management built in
  - Support for scalability and availability built in
- Disadvantages:
  - Transaction management still up to the application
  - More complex deployment
  - Requires a web container

Authentication and Authorization

Any collection of best practices for security must address these security principles:

- Maintaining Corporate Security Policies
- Self-Preservation
- Defense in Depth
- Least Privilege
- Compartmentalization
- Proportionality

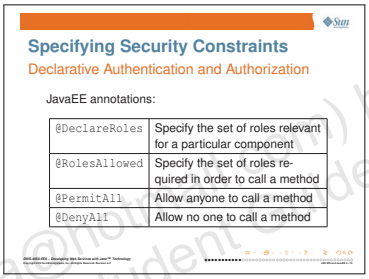
Authentication involves proving the identity of an individual or enterprise. The authentication process typically requires a user, also known as a principal, to provide credentials when requesting a service function. At the low end, user credentials might consist of an enciphered user name and password that the system validates against a private database. At the high end, user credentials might contain a digital signature that demonstrates the user's knowledge of the private-key associated with a certified digital identity. Conversely, an enterprise application might also prove its authenticity to prospective clients.

A well-designed authentication mechanism also helps protect a business from liability associated with a transaction when the service requester denies having used the service. This is known as non-repudiation.

Traditional security solutions provided by the existing Java EE architecture might offer the best and most interoperable security strategies for web services deployed to the Java EE platform, such as the following:

- Web services deployed as Java EE web applications can use traditional web-based security solutions.
- Web services deployed as EJB applications can leverage the robust, role-based, and potentially fine-grained authorization policy supported by the EJB container.
- The Java EE server can secure URLs, to which SOAP messages might be addressed.
- You can configure services to use HTTPS, which provides a secure means of authenticating users and encrypting message content.

You can use the annotations in Table 4.1 to specify method permissions. You can apply these annotations to a bean class or to a method. If applied to a bean class, they apply to all applicable business methods of the bean class.

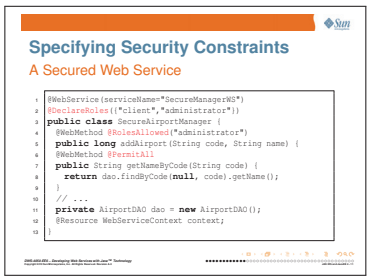


- `PermitAll`
  The `PermitAll` annotation enables all roles to execute the method(s).
- `DenyAll`
  The `DenyAll` annotation specifies that no security roles are permitted to execute the specified method(s).
- `RolesAllowed`
  The`RolesAllowed` annotation is a list of security role names to be mapped to the security roles that are permitted to execute the specified method(s). The `RolesAllowed` annotation can be applied to all methods of a class or to selected methods of a class.

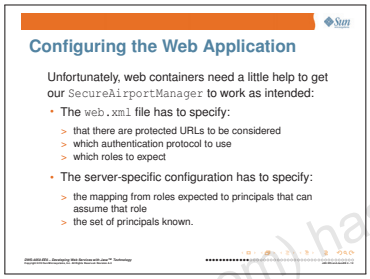
`@DeclareRoles`	Specify the set of roles relevant for a particular component
`@RolesAllowed`	Specify the set of roles required in order to call a method
`@PermitAll`	Allow anyone to call a method
`@DenyAll`	Allow no one to call a method

Table 4.1: Authorization Annotations

Code 4.2 shows a fragment of a web service implementation class that includes annotations for authorization. The annotations and their semantics mirror those of the EJB specification: the developer can specify what types of users can access the different operations provided by the web service, delegating the enforcing of these constraints to the JAX-WS runtime.



Unfortunately, the presence of authentication and authorization annotations on JAX-WS-based web services does not transparently configure the features that support this within the web container. Web containers need a little help to get our `SecureAirportManager` to work as intended:



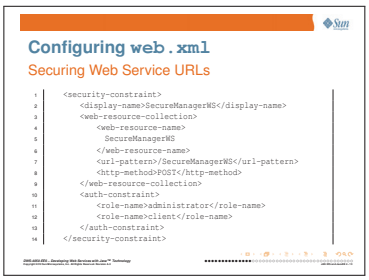
- The `web.xml` file has to specify:
  - that there are protected URLs to be considered
  - which authentication protocol to use
  - which roles to expect
- The server-specific configuration has to specify:
  - the mapping from roles expected to principals
  - the set of principals known.


```
1  @WebService(serviceName="SecureManagerWS")
2  @DeclareRoles({"client","administrator"})
3  public class SecureAirportManager {
4    @WebMethod @RolesAllowed("administrator")
5    public long addAirport(String code, String name) {
6    @WebMethod @PermitAll
7    public String getNameByCode(String code) {
8      return dao.findByCode(null, code).getName();
9    }
10   // ...
11   private AirportDAO dao = new AirportDAO();
12   @Resource WebServiceContext context;
13 }
```

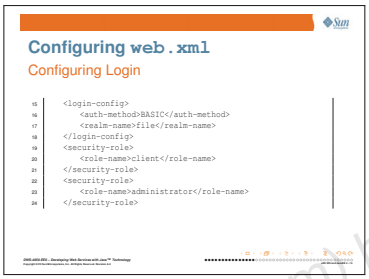
E-32

Code 4.2: A Secured Web Service

Code 4.3 illustrates this strategy. Note how the specification of which users have access to which URLs in the web application that this web service belongs to mirrors the same restrictions when applied to web resources: from the perspective of the web container around the web service, the web service is just another web resource to control.



Security roles are declared in the `web.xml` deployment descriptor, using the `<security-role>` element. This element lives at the first level of the `web.xml` file, as a direct child of the `<web-app>` element. In many cases, it is created and edited using tools in your chosen environment, but the specification of the naked XML is simple too, as shown in Code 4.4 below. The example shows a fragment of `web.xml` illustrating the declaration of the security roles "client" and "administrator" (in lines 10 to 13).



The fragment in Code 4.4 also shows how the `web.xml` must specify the approach to be used by the underlying server to authenticate users. Java EE web containers can use password challenges or client certificates in any of four standardized forms to implement this task. The three approaches that can be used by a web service are:

- BASIC – HTTP Basic authentication. This mechanism is standardized,

```
1  <security-constraint>
2      <display-name>SecureManagerWS</display-name>
3      <web-resource-collection>
4          <web-resource-name>
5            SecureManagerWS
6          </web-resource-name>
7          <url-pattern>/SecureManagerWS</url-pattern>
8          <http-method>POST</http-method>
9      </web-resource-collection>
10     <auth-constraint>
11         <role-name>administrator</role-name>
12         <role-name>client</role-name>
13     </auth-constraint>
14 </security-constraint>
```

E-33

Code 4.3: Securing Resource URLs

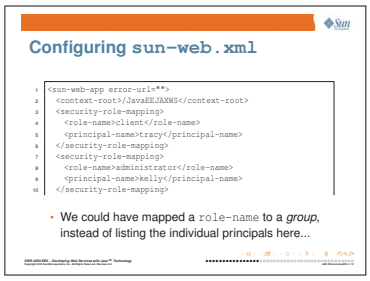
```
15      <login-config>
16          <auth-method>BASIC</auth-method>
17          <realm-name>file</realm-name>
18      </login-config>
19      <security-role>
20          <role-name>client</role-name>
21      </security-role>
22      <security-role>
23          <role-name>administrator</role-name>
24      </security-role>
```

Code 4.4: Configuring Login

based on a username and password challenge, and requires the use of encryption to maintain wire security.

- DIGEST – This mechanism modifies the BASIC approach by providing encryption of the password in transit. However, the other disadvantages remain and it too is unpopular.
- CLIENT CERTIFICATE – In this mechanism, the client is required to provide a public key certificate, signed by an authority trusted by the web container. The web container then challenges the client browser to digitally sign a piece of data. If the signature provided by the client verifies against the certificate, then the identity of the client has been authenticated. This can be a very secure means of authentication provided that the certificates and keys are properly handled, but is usually only used in business to business transactions.

Mapping users to roles is not part of the JavaEE specification, and so it is dependent on the web-container in use. In the case of GlassFish, this occurs in the vendor-specific deployment descriptor file named `sun-web.xml`. This file contains one `<security-role-mapping>` element for each role that will exist; this element specifies the name of the role and the names of all the users that are members of that role. Code 4.5 shows an example of this configuration file.

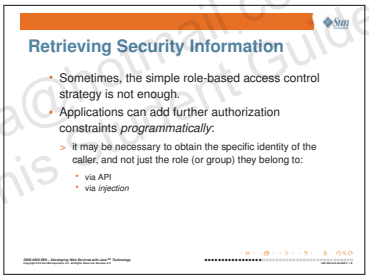


```
1  <sun-web-app error-url="">
2    <context-root>/JavaEEJAXWS</context-root>
3    <security-role-mapping>
4      <role-name>client</role-name>
5      <principal-name>tracy</principal-name>
6    </security-role-mapping>
7    <security-role-mapping>
8      <role-name>administrator</role-name>
9      <principal-name>kelly</principal-name>
10   </security-role-mapping>                                E-35
```

Code 4.5: Fragment of Sample `sun-web.xml` File

Retrieving Security Information Programmatically

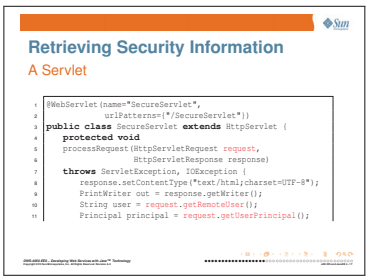
Sometimes, the simple role-based access control strategy is not enough. Applications can add further authorization constraints *programmatically*:



- it may be necessary to obtain the specific identity of the caller, and not just the role (or group) they belong to:
  - via API
  - via *injection*

Programmatic authorization is the responsibility of the bean developer. The following methods in the `HttpServletRequest` support programmatic authorization:

- `boolean isUserInRole(String role)`
- `Principal getUserPrincipal()`
- `void login(String user, String password)`
- `void logout()`

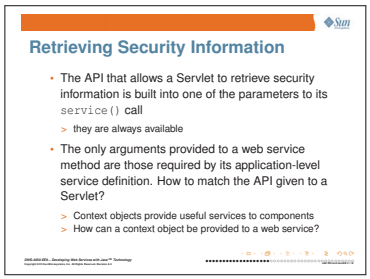


Programmatic authorization is more expressive than the declarative approach, but is more cumbersome to maintain, and because of the additional complexity, more error prone. In particular, declarative authorization controls access at a role level while programmatic authorization can selectively permit or block access to principals belonging to a role. Code 4.6 shows a code fragment from a web service that uses programmatic authentication.

```
1  @WebServlet(name="SecureServlet",
2              urlPatterns={"/SecureServlet"})
3  public class SecureServlet extends HttpServlet {
4      protected void
5      processRequest(HttpServletRequest request,
6                     HttpServletResponse response)
7      throws ServletException, IOException {
8          response.setContentType("text/html;charset=UTF-8");
9          PrintWriter out = response.getWriter();
10         String user = request.getRemoteUser();
11         Principal principal = request.getUserPrincipal();
```

E-36

Code 4.6: Programmatic Authentication and Authorization



In the case of the servlet infrastructure, the API that allows a servlet to retrieve security information is built into one of the parameters to its `service()` call – and so it is always available to the servlet, every time it is called.

The only arguments provided to a web service method are those required by its application-level service definition – so the authors of the JAX-WS specification had to work out some means to provide this API to the JAX-WS service provider conveniently. To do so, they borrow a page from other specifications that had run into similar issues: servlets needed to interact with the web container they are deployed into, applets needed to interact with the application container they are deployed into (the web browser), enterprise beans needed interact with the EJB container they are deployed into ... In all these cases, the approach taken was the same – and it is also the approach taken here: some type of `Context` supports interaction between the component and its container.

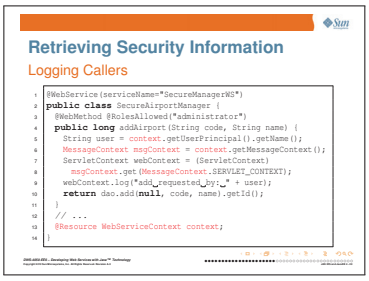
In our particular case, the JAX-WS specification introduces the notion of a `WebServiceContext` that the web service provider can retrieve and use – and the standard way to provide resources to a component the resources it requires is to us *dependency injection*. Code 4.7 shows an example of this.


```
1  @WebService(serviceName="SecureManagerWS")
2  public class SecureAirportManager {
3    @WebMethod @RolesAllowed("administrator")
4    public long addAirport(String code, String name) {
5      String user = context.getUserPrincipal().getName();
6      return dao.add(null, code, name).getId();
7    }
8    // ...
9    @Resource WebServiceContext context;
10 }
```

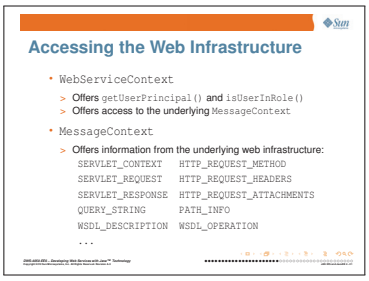

Code 4.7: Dependency Injection and `WebServiceContext`

`WebServiceContext` also offers access to the underlying `MessageContext`.

Code 4.8 shows an example where the JAX-WS-based web service needs to obtain authentication information, in order to log the identity of the caller issuing every request processed by the web service provider.



`WebServiceContext` offers a relatively simple API – consistent with the other context objects used elsewhere in Java. However, the JAX-WS specification allows application developers to ask for far more detailed information than that available directly via `WebServiceContext`. The alternate type that offers this richer API is `MessageContext`.



`WebServiceContext` offers these API functions:

- `getUserPrincipal()` and `isUserInRole()`

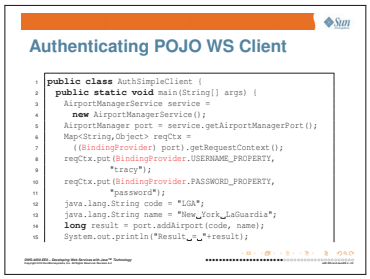
```
1  @WebService(serviceName="SecureManagerWS")
2  public class SecureAirportManager {
3    @WebMethod @RolesAllowed("administrator")
4    public long addAirport(String code, String name) {
5      String user = context.getUserPrincipal().getName();
6      MessageContext msgContext = context.getMessageContext();
7      ServletContext webContext = (ServletContext)
8        msgContext.get(MessageContext.SERVLET_CONTEXT);
9      webContext.log("add requested by: " + user);
10     return dao.add(null, code, name).getId();
11   }
12   // ...
13   @Resource WebServiceContext context;
14 }
```

E-32

Code 4.8: Logging Callers in JAXWS

- Getter to access the underlying `MessageContext`

`MessageContext` offers information from the underlying web infrastructure:



When the web service requires authentication data in order to authorize requests, the JAX-WS client-side framework will need to obtain authentication data from the application, in order to pass it along. JAX-WS proxy objects implement the `BindingProvider` interface, which describes methods that allow the client application to configure the proxy, providing information such as this authentication data. Code 4.9 shows how a client application could use this interface to provide username and password information to the proxy (in lines 6 to 11).

Property Type	Property Name	Property Type	Description
Standard	MESSAGE_OUTBOUND_PROPERTY	Boolean	The message direction – true for outbound messages, false for inbound. Handlers may use this property to determine if the processing is on an outbound or inbound message.
	INBOUND_MESSAGE_ATTACHMENTS	(java.util.Map)	Attachments to an inbound message. These can be used to acquire the attachments in inbound message.
	OUTBOUND_MESSAGE_ATTACHMENTS	(java.util.Map)	Attachments to an outbound message. A proxy can use these to send attachments not described through WSDL MIME binding.
HTTP	HTTP_REQUEST_METHOD	(java.lang.String)	The name of the HTTP method with which a request is made; for example, GET, POST, or PUT.
	HTTP_REQUEST_HEADERS	(java.util.Map)	The HTTP headers for the request message.
	QUERY_STRING	(String)	The HTTP query string for the request message.
	PATH_INFO	(String)	Extra path information associated with the request URL. This information follows the base url path, but precedes the query string.
	HTTP_RESPONSE_CODE	(java.lang.Integer)	The HTTP response status code for the last invocation.
	HTTP_RESPONSE_HEADERS	(java.util.Map)	The HTTP response headers.

Table 4.2: Properties Available via `MessageContext`

Property Type	Property Name	Property Type	Description
In Servlet Container	SERVLET_CONTEXT	(javax.servlet.ServletContext)	The ServletContext object of the Web application that contains the Web endpoint.
	SERVLET_REQUEST	(javax.servlet.http.HttpServletRequest)	The HTTPServletRequest object associated with the request currently being served.
	SERVLET_RESPONSE	(javax.servlet.http.HttpServletResponse)	The HTTPServletResponse object associated with the request currently being served.
Optional	WSDL_DESCRIPTION	(java.net.URI)	A resolvable URI that may be used to obtain access to the WSDL for the endpoint.
	WSDL_SERVICE	(javax.xml.namespace.Qname)	The name of the service being invoked.
	WSDL_PORT	(javax.xml.namespace.Qname)	The name of the port to which the current message is addressed.
	WSDL_INTERFACE	(javax.xml.namespace.Qname)	The name of the port type to which the current message belongs.
	WSDL_OPERATION	(javax.xml.namespace.Qname)	The name of the WSDL operation with which the current message is associated.

http://www.javaworld.com/javaworld/jw-02-2007/jw-02-handler.html

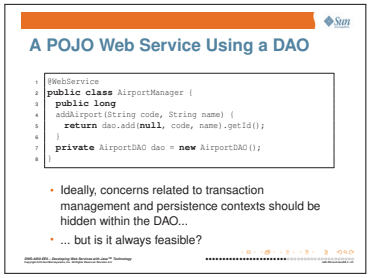
Table 4.3: Properties Available via `MessageContext`

```
1  public class AuthSimpleClient {
2    public static void main(String[] args) {
3      AirportManagerService service =
4        new AirportManagerService();
5      AirportManager port = service.getAirportManagerPort();
6      Map<String,Object> reqCtx =
7        ((BindingProvider) port).getRequestContext();
8      reqCtx.put(BindingProvider.USERNAME_PROPERTY,
9                 "tracy");
10     reqCtx.put(BindingProvider.PASSWORD_PROPERTY,
11                "password");
12     java.lang.String code = "LGA";
13     java.lang.String name = "New York LaGuardia";
14     long result = port.addAirport(code, name);
15     System.out.println("Result = "+result);                E-139
```

Code 4.9: Authenticating POJO WS Client

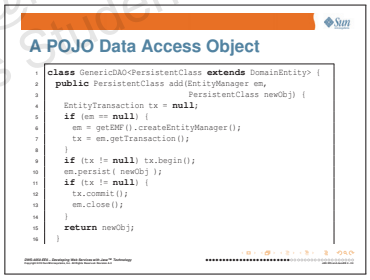
Creating a Web Service From an EJB

Limitations of POJO Web Services



The original implementation of the `AirportManager` web service, shown in Code 2.2, was implemented as a POJO web service provider – a simple Java object that did what was needed.

Ideally, concerns related to transaction management and persistence contexts should be hidden within the DAO – so that our web service provider would not have to be concerned with this. If we could achieve this, we would have a model that exhibited better decoupling between business and integration tier – and reducing coupling is always a good thing. However, ... It is not always feasible...

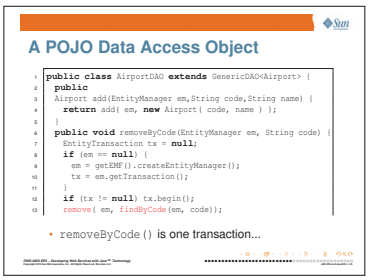


Code 4.10 shows a fragment of the implementation of a `GenericDAO` class – a POJO implementation of a DAO that attempts to encapsulate JPA-related knowledge, such as management of entity managers and transactions, away from its clients. Ideally, our `AirportManager` could simply delegate this knowledge to the `GenericDAO`.

```
1  class GenericDAO<PersistentClass extends DomainEntity> {
2    public PersistentClass add(EntityManager em,
3                               PersistentClass newObj) {
4      EntityTransaction tx = null;
5      if (em == null) {
6        em = getEMF().createEntityManager();
7        tx = em.getTransaction();
8      }
9      if (tx != null) tx.begin();
10     em.persist( newObj );
11     if (tx != null) {
12       tx.commit();
13       em.close();
14     }
15     return newObj;
16   }
```


Code 4.10: A POJO Data Access Object

Code 4.11 shows a simple operation – `removeByCode` – which does successfully encapsulate management of the entity manager and associated transaction: all the work required for `removeByCode` is internal to that function. In particular, `removeByCode()` is one transaction...

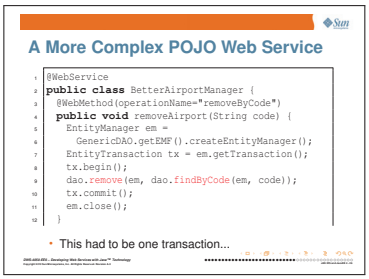



```
1  public class AirportDAO extends GenericDAO<Airport> {
2    public
3    Airport add(EntityManager em,String code,String name) {
4      return add( em, new Airport( code, name ) );
5    }
6    public void removeByCode(EntityManager em, String code) {
7      EntityTransaction tx = null;
8      if (em == null) {
9        em = getEMF().createEntityManager();
10       tx = em.getTransaction();
11     }
12     if (tx != null) tx.begin();
13     remove( em, findByCode(em, code));
```

E-42

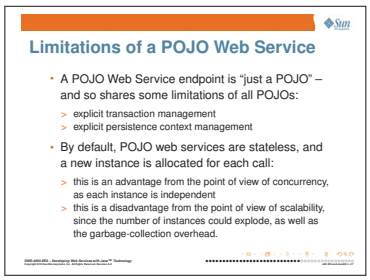
Code 4.11: A Simple Operation in a DAO

However - a more complex operation in our Data Access Object is enough to demonstrate the limitations to this approach to implementing our business and integration objects. Code 4.12 shows a more complex operation – `removeAirport()` – which has to perform a number of tasks in order to remove the airport it's given as a parameter. The only way to implement this function as a single transaction requires that `AirportManager` accept responsibility for managing both entity manager and transaction – it is the only way it can share those across the multiple steps required to achieve it responsibility in `removeAirport()`.



But increasing coupling between `AirportManager` and `GenericDAO` is poor design...

A POJO Web Service endpoint is "just a POJO" – and so shares some limitations of all POJOs: explicit transaction management and explicit persistence context management. By default, POJO web services are stateless, and a new instance is allocated for each call:

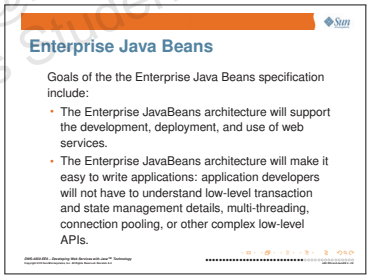


- this is an advantage from the point of view of concurrency, as each instance is independent
- this is a disadvantage from the point of view of scalability, since the number of instances could explode, along with the garbage-collection overhead.

Enterprise Java Beans

Goals of the the Enterprise Java Beans specification include:

- The Enterprise JavaBeans architecture will support the development, deployment, and use of web services.
- The Enterprise JavaBeans architecture will make it easy to write applications: application developers will not have to understand low-level transaction and state management details, multi-threading, connection pooling, or other complex low-level APIs.

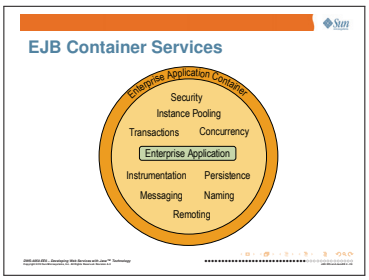



```
1  @WebService
2  public class BetterAirportManager {
3    @WebMethod(operationName="removeByCode")
4    public void removeAirport(String code) {
5      EntityManager em =
6        GenericDAO.getEMF().createEntityManager();
7      EntityTransaction tx = em.getTransaction();
8      tx.begin();
9      dao.remove(em, dao.findByCode(em, code));
10     tx.commit();
11     em.close();
12   }
```

E-4

Code 4.12: A More Complex POJO Web Service

The JavaEE specification asks what kinds of machinery is common across application, and can be implemented as more or less boilerplate logic – then incorporates those into a framework to develop and deploy enterprise applications.



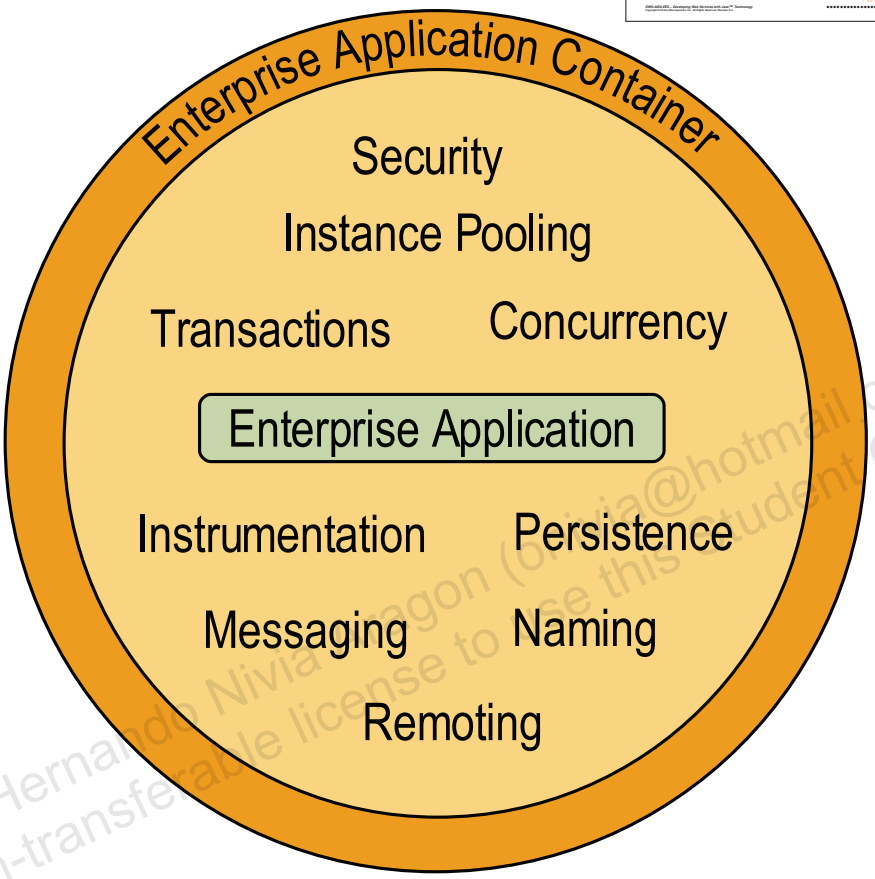


Figure 4.2: EJB Container Services

The following services, illustrated in Figure 4.2, are offered by JavaEE application servers:

- Concurrency Services
  Any running thread that relies on a resource outside the JVM, implementation must wait for a response from that service before allowing the thread to continue operation. Without the creation of new threads to run that same process concurrently, numerous requests for the same operation can create a significant bottleneck in an enterprise application without taxing the resources of the hosting server. Therefore, the usage of a multithreaded environment for long running processes and processes that access external resources is highly advisable.

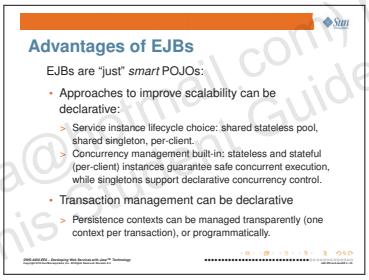
When implementing a multithreaded environment, great care needs to be taken to ensure the integrity of any shared state within the running application. This is accomplished in Java technology with the use of object locking. Object locks ensure that threads take turns accessing a piece of code around which an object lock might have been created. These segments of code, referred to as synchronized blocks, should be as short as possible.

- Transaction Services
  Transactions allow you to treat a sequence of operations as a single atomic operation. It is often desirable to have an all-or-nothing approach to business logic. With a transaction, if you encounter a failure in one of the steps of your atomic sequence, you can undo each of the previous steps in your sequence, in reverse order, back to the point where your transaction began.
- Security Services
  Certainly, securing access to sensitive resource that are accessible using a public web interface is a good idea. However, it is also important to secure the access of any distributed components that make themselves available for remote execution.
- Instance Pooling Services
  There are certain circumstances where the instantiation of objects can be an expensive process from a resource standpoint. In those circumstances, not having a pool of pre-instantiated objects to pull from can result in a sluggish application. Objects that benefit from pre-instantiation into an object pool are those that either involve the connection to external resources or involve the generation of source code.
- Messaging Services
  Messaging-oriented services provide the framework for the delivery of asynchronous out-of-band messages. Messaging systems should have a high degree of fault tolerance to lessen the likelihood of losing a message in an outage condition. The ability to persist messages that have been submitted for delivery, as well as an intelligent handshake process for the production and consumption of messages within the context of a transaction, are important concerns.
- Naming Services One of the core motivations behind architecting for an enterprise application is the ability to distribute components to separate servers for hosting. Scaling a single server might not be the most cost effective way to handle high load in all tiers of the SunTone AM.
  For objects in two distinct JVM heap spaces to properly communicate with each other, the calling object needs a proxied handle to the item with which it wants to communicate. That handle directs the caller to the appropriate location of the callee in the callee's JVM heap space. The

mechanism for managing the registration and lookup of these handles is referred to as a naming service.

- Instrumentation Services
  Java Management eXtensions (JMX) provides a facility for the runtime configuration and stateful monitoring of your application components. The classes that manage the state of a component are referred to as Managed Beans (MBeans). MBeans are instrumented and monitored though an MBean Server.
- Persistence Services
  Java EE 5 application servers provide an implementation of the Java Persistence Architecture (JPA). JPA is an Object/Relational Mapping (ORM) technology that provides a framework for the management of application state within an RDBMS.

EJBs are "just" *smart* POJOs – but those "smarts" allow application developers to delegate some of the application-independent, boilerplate logic to the JavaEE application server. For example, approaches to improve scalability can be declarative – describe which strategy you would like to see realized, among the choices offered by the JavaEE container, then let the JavaEE container actually implement it:

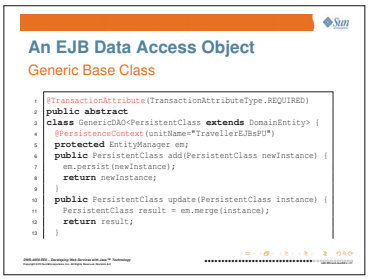


- Service instance lifecycle choice: shared stateless pool, shared singleton, or per-client.
- Concurrency management built-in: stateless and stateful (per-client) instances guarantee safe concurrent execution, while singletons support declarative concurrency control.

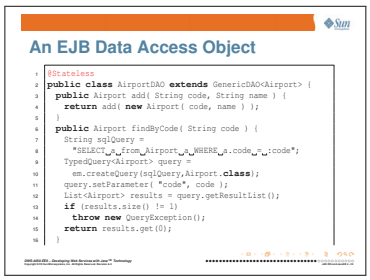
Also, transaction management can be declarative:

- Persistence contexts can be managed transparently (one context per transaction), or programmatically.

Code 4.13 revisits the Data Access Object classes – except that, instead of implementing them as POJOs, we now choose to implement them as EJBs. As a result, implementation logic that we used to have code ourselves into the application, can now be delegated to the machinery provided by the application server.



Code 4.14 shows a version of the `Airport` Data Access Object, implemented as a stateless session bean. As a result of this choice of implementation, the DAO gains abilities such as dependency injection of helper instances, and automatic transaction management.




```
1
2  @TransactionAttribute(TransactionAttributeType.REQUIRED)
3  public abstract
4  class GenericDAO<PersistentClass extends DomainEntity> {
5    @PersistenceContext(unitName="TravellerEJBsPU")
6    protected EntityManager em;
7    public PersistentClass add(PersistentClass newInstance) {
8      em.persist(newInstance);
9      return newInstance;
10   }
11   public PersistentClass update(PersistentClass instance) {
12     PersistentClass result = em.merge(instance);
13     return result;
14   }
```

E-52

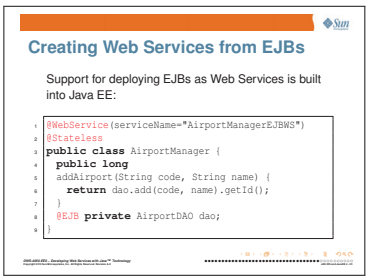
Code 4.13: A Data Access Object – Revisited

```
1  @Stateless
2  public class AirportDAO extends GenericDAO<Airport> {
3    public Airport add( String code, String name ) {
4      return add( new Airport( code, name ) );
5    }
6    public Airport findByCode( String code ) {
7      String sqlQuery =
8        "SELECT a from Airport a WHERE a.code = :code";
9      TypedQuery<Airport> query =
10       em.createQuery(sqlQuery,Airport.class);
11     query.setParameter( "code", code );
12     List<Airport> results = query.getResultList();
13     if (results.size() != 1)
14       throw new QueryException();
15     return results.get(0);
16   }
```

E-55

Code 4.14: An EJB Data Access Object – AirportDAO

The web services specifications are orthogonal to the EJB specification. To define an EJB to also be a web service, we just have to say so – and support for deploying EJBs as Web Services is built into Java EE. Code 4.15 shows an example of a JAX-WS web service, now implemented as an EJB.




```
1 @WebService(serviceName="AirportManagerEJBWS")
2 @Stateless
3 public class AirportManager {
4   public long
5   addAirport(String code, String name) {
6     return dao.add(code, name).getId();
7   }
8   @EJB private AirportDAO dao;
9 }
```

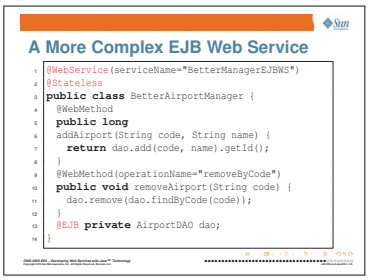
E-38

Code 4.15: Creating Web Services from EJBs

Note how `AirportManager` obtains its DAO via *injection* – whether/how that `AirportDAO` is shared is transparent!

Transaction Management

Our `AirportManager` POJO web service had to be implemented to be responsible for managing entity managers (and their persistence contexts), and transactions explicitly. As we saw in the example in Code 4.12 it is possible to hide this control, at least some of the time – but there comes a time where the application level object needs to take control (of the transaction, for instance).



Code 4.16 shows an alternate implementation of the `AirportManager` service – but, this time, as a stateless session bean. Note in lines 9-11, for instance, how responsibility for entity managers and their associated persistence context, and for transaction management, are now hidden within the Data Access Object EJB .

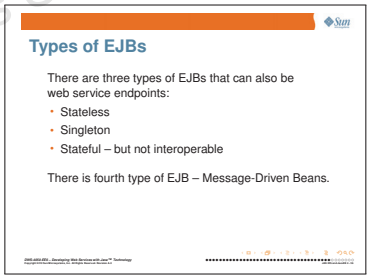
```
1  @WebService(serviceName="BetterManagerEJBWS")
2  @Stateless
3  public class BetterAirportManager {
4    @WebMethod
5    public long
6    addAirport(String code, String name) {
7      return dao.add(code, name).getId();
8    }
9    @WebMethod(operationName="removeByCode")
10   public void removeAirport(String code) {
11     dao.remove(dao.findByCode(code));
12   }
13   @EJB private AirportDAO dao;
14 }
```

E-39

Code 4.16: A More Complex EJB Web Service

Scalability

There are three types of EJBs that can also be web service endpoints:

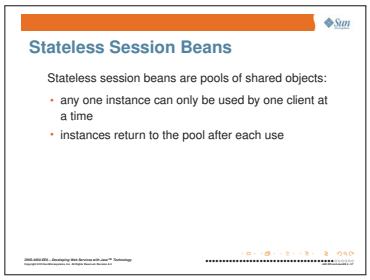


- Stateless
- Singleton
- Stateful – but not interoperable

There is a fourth type of EJB – Message-Driven Beans – but they are not useful as implementations of JAX-WS-based web services.

Stateless Session Beans

A stateless session bean has the following characteristics:



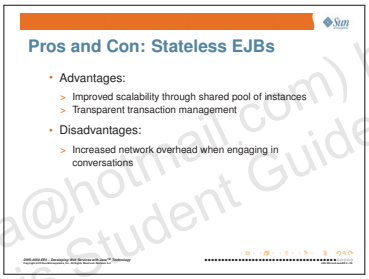
- The bean does not retain client-specific information.
- A client might not get the same session bean instance.

- Any number of client requests can be handled by the same session bean instance. This has a profound impact on performance.

Given these semantics, stateless session beans are often implemented as pools of shared objects:

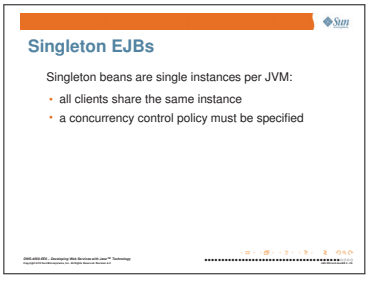
- any one instance can only be used by one client at a time
- instances return to the pool after each use

Code 4.16 shows our `AirportManager` as a stateless session bean.



- Advantages:
  - Improved scalability through shared pool of instances
  - Implicit transaction management
- Disadvantages:
  - Increased network overhead when engaging in conversations

Singleton Session Beans



Singleton session beans represent an implementation of the singleton pattern. Singleton session beans can be used to provide shared access and concurrent access across clients.

A singleton session bean is instantiated once per application, and exists for the lifecycle of the application. In cases where the container is distributed over many virtual machines, each application will have one bean instance of the singleton for each JVM.

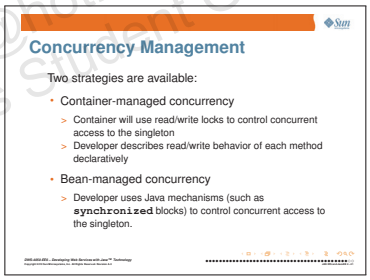
Since singleton beans are shared and used concurrently, a concurrency control policy must be specified.

Code 4.17 shows our `AirportManager` as a singleton session bean.

```
1  @WebService(serviceName="SingletonManagerEJBWS")
2  @Singleton
3  public class SingletonAirportManager {
4    @WebMethod
5    public long addAirport(String code, String name) {
6      return dao.add(code, name).getId();
7    }
8    @WebMethod(operationName="removeByCode")
9    public void removeAirport(String code) {
10     dao.remove(dao.findByCode(code));
11   }
12   @WebMethod
13   public String getNameByCode(String code) {
14     return dao.findByCode(code).getName();
15   }
```

E-40

Code 4.17: Singleton Web Service EJBs



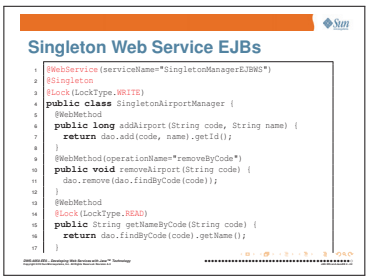
Concurrency is a term used to describe some form of parallel processing, or processing that occurs at the same time, within an application. Concurrency is commonly used to increase throughput and responsiveness of an application.

Singleton session beans support concurrency at the object level: multiple clients can simultaneously invoke methods of the same singleton session bean instance. The only way to do this safely, is to control access to the shared singleton instance.

Two strategies are available:

- Container-managed concurrency
  - Container will use read/write locks to control concurrent access to the singleton
  - Developer describes read/write behavior of each method declaratively
- Bean-managed concurrency
  - Developer uses Java mechanisms (such as `synchronized` blocks) to control concurrent access to the singleton.

In singleton beans that utilize container-managed concurrency, the container controls concurrent access to the bean instance based on method-level locking metadata. Specifically, the `javax.ejb.Lock` and `javax.ejb.LockType` annotations are used to specify the concurrent access management strategy of the singleton's business methods.



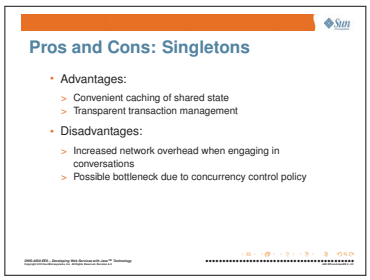
The `@Lock` annotation informs the container that a method requires concurrency management. There are two locking strategies defined by constants included in `LockType`: `javax.ejb.LockType.READ` and `javax.ejb.LockType.WRITE`.

Choosing between the `LockType` depends on the operational characteristics of the business method. If the business method can support concurrent access, such as returning the value of a variable, the method should be annotated with `@Lock(READ)`.

Code 4.18 shows our `SingletonAirportManager` with a possible concurrency control specification built in.

```
1  @WebService(serviceName="SingletonManagerEJBWS")
2  @Singleton
3  @Lock(LockType.WRITE)
4  public class SingletonAirportManager {
5    @WebMethod
6    public long addAirport(String code, String name) {
7      return dao.add(code, name).getId();
8    }
9    @WebMethod(operationName="removeByCode")
10   public void removeAirport(String code) {
11     dao.remove(dao.findByCode(code));
12   }
13   @WebMethod
14   @Lock(LockType.READ)
15   public String getNameByCode(String code) {
16     return dao.findByCode(code).getName();
17   }
```


Code 4.18: Singleton Web Service EJB with Concurrency Policy



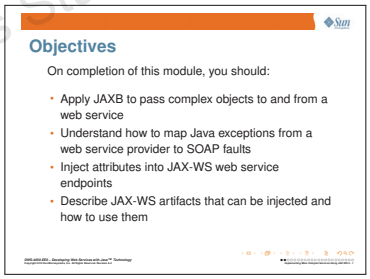
- Advantages:
  - Convenient caching of shared state
  - Implicit transaction management
- Disadvantages:
  - Increased network overhead when engaging in conversations
  - Possible bottleneck due to concurrency control policy

Chapter 5

Implementing More Complex Services Using JAX-WS

On completion of this module, you should:

- Apply JAXB to pass complex objects to and from a web service
- Understand how to map Java exceptions from a web service provider to SOAP faults
- Inject attributes into JAX-WS web service endpoints
- Describe JAX-WS artifacts that can be injected and how to use them



Additional Resources

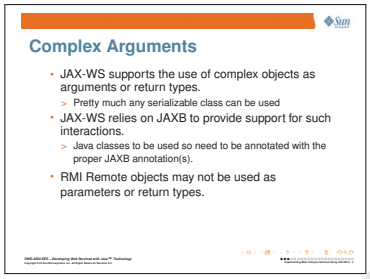
The following references provide additional information on the topics described in this module:

- Ayyappan Gandhirajan, "User Defined Exceptions: Improve Error Handling in Web Services," `http://www.developer.com/services/article.php/3493491`.
- Ping Wang and Russell Butek, "Web services programming tips and tricks: Exception Handling with JAX-RPC," 2004, `http://www-128.ibm.com/developerworks/webservices/library/ws-tip-jaxrpc.html`.

Complex Arguments and Return Values

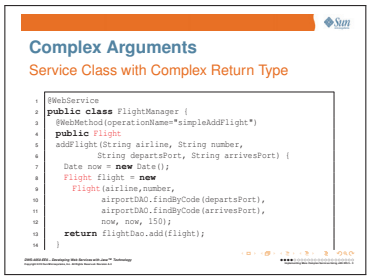
In our examples to date, we have always used "simple" Java types (Java primitives, or Strings) for parameter and return types for our web services. However, there is no rule that actually limits us to just those types.

JAX-WS delegates the mapping of Java programming language types to and from XML definitions to JAXB. Application developers don't need to know the details of these mappings, but they should be aware that not every class in the Java language can be used as a method parameter or return type in JAX-WS.



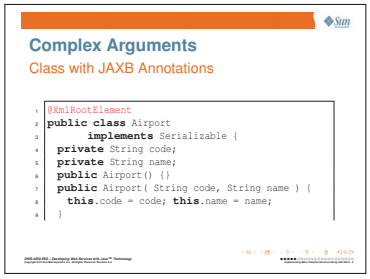
- JAX-WS supports the use of complex objects as arguments or return types.
  - Pretty much any serializable class can be used
- JAX-WS relies on JAXB to provide support for such interactions.
  - Java classes to be used so need to be annotated with the proper JAXB annotation(s).
- RMI Remote objects may not be used as parameters or return types.

Figure 5.1 shows an example of a web service implementation class that returns a complex Java type (a class with a number of members) by value from calls to its `simpleAddFlight()` operation.

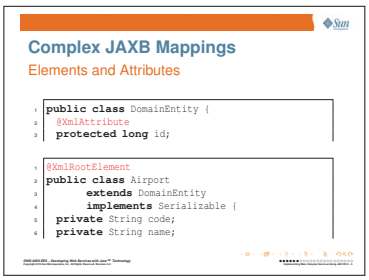


In this example, the web service returns the persistent object that is used within the business tier directly out to its caller. This might be dangerous, if the client is in fact in a different tier, as it might expose internal business logic to that external client.

JAX-WS relies on JAXB to map Java objects to the XML representation that is used to encode them into SOAP messages. JAXB is another Java technology that relies on annotations and "configuration by default". The simplest possible JAXB-enabled class would simply state that it can be represented in XML form - as shown in Code 5.2.



JAXB supports a number of options, as far as customizing the mapping between the representation of an entity in the Java programming language as opposed to its representation in XML Schema form. For example, Java represents information about an entity as properties of its class. XML Schema, on the other hand, have two different means of representing information about a person: children elements, and attributes of the element. Figure 5.3 shows how one could customize the Java to XML mapping used by JAXB to represent domain data about airports as elements of the XML node for the airport, while representing internal data (the internal id associated with each airport entity) as an attribute on that XML node.




```
1  @WebService
2  public class FlightManager {
3    @WebMethod(operationName="simpleAddFlight")
4    public Flight
5    addFlight(String airline, String number,
6              String departsPort, String arrivesPort) {
7      Date now = new Date();
8      Flight flight = new
9        Flight(airline,number,
10             airportDAO.findByCode(departsPort),
11             airportDAO.findByCode(arrivesPort),
12             now, now, 150);
13     return flightDao.add(flight);
14   }
```

E-58

Code 5.1: Service Class with Complex Return Type

```
1  @XmlRootElement
2  public class Airport
3          implements Serializable {
4    private String code;
5    private String name;
6    public Airport() {}
7    public Airport( String code, String name ) {
8      this.code = code;  this.name = name;
9    }
```

E-67

Code 5.2: Class with JAXB Annotations

```
1  public class DomainEntity {
2    @XmlAttribute
3    protected long id;
```

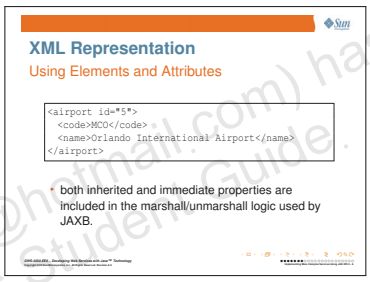
E-69

```
1  @XmlRootElement
2  public class Airport
3          extends DomainEntity
4          implements Serializable {
5    private String code;
6    private String name;
```

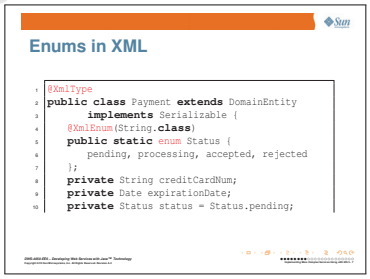
E-67

Code 5.3: Mapping Elements and Attributes

Figure 5.4 shows how an `Airport` entity is represented in XML, given the definition for class `Airport` shown in Figure 5.3.



Some Java constructs require special handling, including enumerated types and instances of `java.util.Date`. Enumerated types capture in Java the notion of a property with a fixed set of possible values associated with it – but such enumerations of values could be captured in one of two ways:



- using the symbolic labels for each value, or
- using numbers corresponding to the position of the values in the set.

JAXB can represent enumerations in XML form using either of these two choices – developers can use annotations to specify which representation to use, as show in Figure 5.5.

```
<airport id="5">
  <code>MCO</code>
  <name>Orlando International Airport</name>
</airport>
```

Code 5.4: XML Representation

```
@XmlType
public class Payment extends DomainEntity
        implements Serializable {
    @XmlEnum(String.class)
    public static enum Status {
        pending, processing, accepted, rejected
    };
    private String creditCardNum;
    private Date expirationDate;
    private Status status = Status.pending;
```

E-71

Code 5.5: Enums in XML

Code 5.6 shows the XML Scheme representation that corresponds to the `Payment` class presented in Figure 5.5.

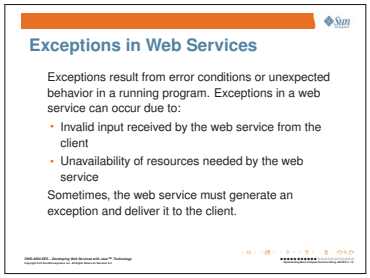
```
  <xs:complexType name="domainEntity">
    <xs:sequence/>
    <xs:attribute name="id" type="xs:long" use="required"/>
    <xs:attribute name="version" type="xs:int"/>
  </xs:complexType>
  <xs:complexType name="payment">
  <xs:complexContent>
    <xs:extension base="tns:domainEntity">
    <xs:sequence>
      <xs:element name="ticket" type="tns:ticket"/>
      <xs:element name="creditCardNum" type="xs:string"/>
      <xs:element name="bankName" type="xs:string"/>
      <xs:element name="expirationDate" type="xs:dateTime"/>
      <xs:element name="status" type="tns:status"/>
    </xs:sequence>
    </xs:extension>
  </xs:complexContent>
  </xs:complexType>
  <xs:simpleType name="status">
    <xs:restriction base="xs:string">
      <xs:enumeration value="pending"/>
      <xs:enumeration value="processing"/>
      <xs:enumeration value="accepted"/>
      <xs:enumeration value="rejected"/>
    </xs:restriction>
  </xs:simpleType>
```

E-60

Code 5.6: `FlightManager` XML Schema

Exception Handling

In general, an exception results from an error condition or an unexpected behavior in a running program. Exceptions in a web service can occur due to:



- Invalid input received by the web service from the client
- Unavailability of resources needed by the web service

The web service can recover from some of the errors. However, for other errors, the web service must generate an exception and deliver it to the client. Even the exception code must be encapsulated in an XML document and delivered to the client in the form of web services that use XML for transportation. Web services are language independent so the exception objects are never passed between client and server. The client gets either an XML fault element or an HTTP header with an error code.

A web service can and will attempt to recover from some of the errors. Exception objects cannot be passed as Java language objects to a non-Java client, so for portability the exception details can be serialized as XML elements, instead. The `fault` element is the XML representation of an exception in SOAP-based services. Faults are received and handled not just by Java clients, but by clients implemented in any language.

Mapping WSDL-to-Java Exception Classes

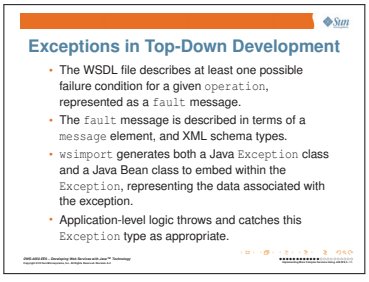
When working with web services, there can be differences between the SOAP-defined types used by the service and the Java-defined types used by the client application. To handle these different types, a client of a web service cannot use the normal approach of importing remote classes. Instead, the client must map the WSDL types to Java types to obtain the parameter and return types used by the service. Once the types are mapped, the client has the correct Java types to use in its code.

```
<message name="addPassengerFault">
  <part name="parameters" element="tns:addPassengerFault"/>
</message>
<portType name="SaferPassengerManager">
  <operation name="addPassenger">
    <input name="input1" message="tns:addPassengerRequest"/>
    <output name="out1" message="tns:addPassengerResponse"/>
    <fault name="fault1" message="tns:addPassengerFault"/>
  </operation>
</portType>
```

Code 5.7: WSDL Code Fragment – `fault` Message

```
  <xsd:element name="addPassengerFault">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="duplicateCode" type="xsd:string"/>
        <xsd:element name="duplicateName" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
```

Code 5.8: XML Schema Type Fragment – `fault` Message



When working WSDL-to-Java, the developer first defines the potential failures associated with specific operations as `fault` messages in the description of each operation. These `fault` messages then get mapped to Java exception classes. For language-independence, however, JAX-WS does not want to map `fault` messages simply to Java exceptions – since another language might not represent transactions at all. Instead, the default mapping used by JAX-WS maps a `fault` message to two types, a Java exception, and a Java bean that is associated with that exception, and which holds any data that the application might have about that exception.

The WSDL code fragments shown in Code 5.7 and Code 5.8 capture the way in which `fault` messages are introduced. Code 5.9 and Code 5.10 show the code generated by JAX-WS for this fault message.

```
@XmlType(name = "", propOrder = {
    "duplicateCode",
    "duplicateName"
})
@XmlRootElement(name = "addPassengerFault")
public class AddPassengerFault {
    @XmlElement(required = true)
    protected String duplicateCode;
    @XmlElement(required = true)
    protected String duplicateName;
```

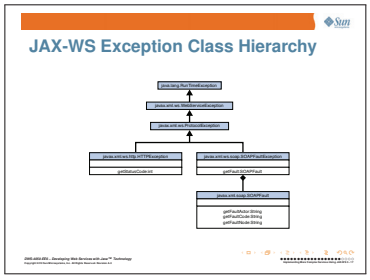
Code 5.9: Fault Type Generated by JAX-WS

```
@WebFault(name = "addPassengerFault",
          targetNamespace = "urn://Traveller/")
public class AddPassengerFault_Exception
        extends Exception {
```

Code 5.10: Exception Type Generated by JAX-WS for Fault

The JAX-WS API Exception Classes

The JAX-WS API provides a family of predefined exception classes. These are classes that the service should commonly use to communicate exceptions to a client accessing the service. The JAX-WS API provides a family of predefined exception classes, such as:



- `javax.xml.ws.WebServiceException`
  A runtime exception that is thrown by methods in JAX-WS APIs when error occurs due to incorrect configuration or incorrect parameters on the client-side. It is the base exception class for all JAX-WS API runtime exceptions.
- `javax.xml.ws.ProtocolException`
  A base class for exceptions related to a specific protocol binding, such as SOAP or HTTP. The following subclasses are used to communicate protocol-level fault information to clients:
  - `javax.xml.ws.soap.SOAPFaultException`
    A subclass of `ProtocolException` that carries SOAP-specific information from SOAP-based web services to clients.
  - `javax.xml.ws.http.HTTPException`

A subclass of `ProtocolException` that carries HTTP-specific information from REpresentational State Transfer (REST)-ful web services to clients.

Figure 5.1 shows the hierarchy of JAX-WS exception classes.

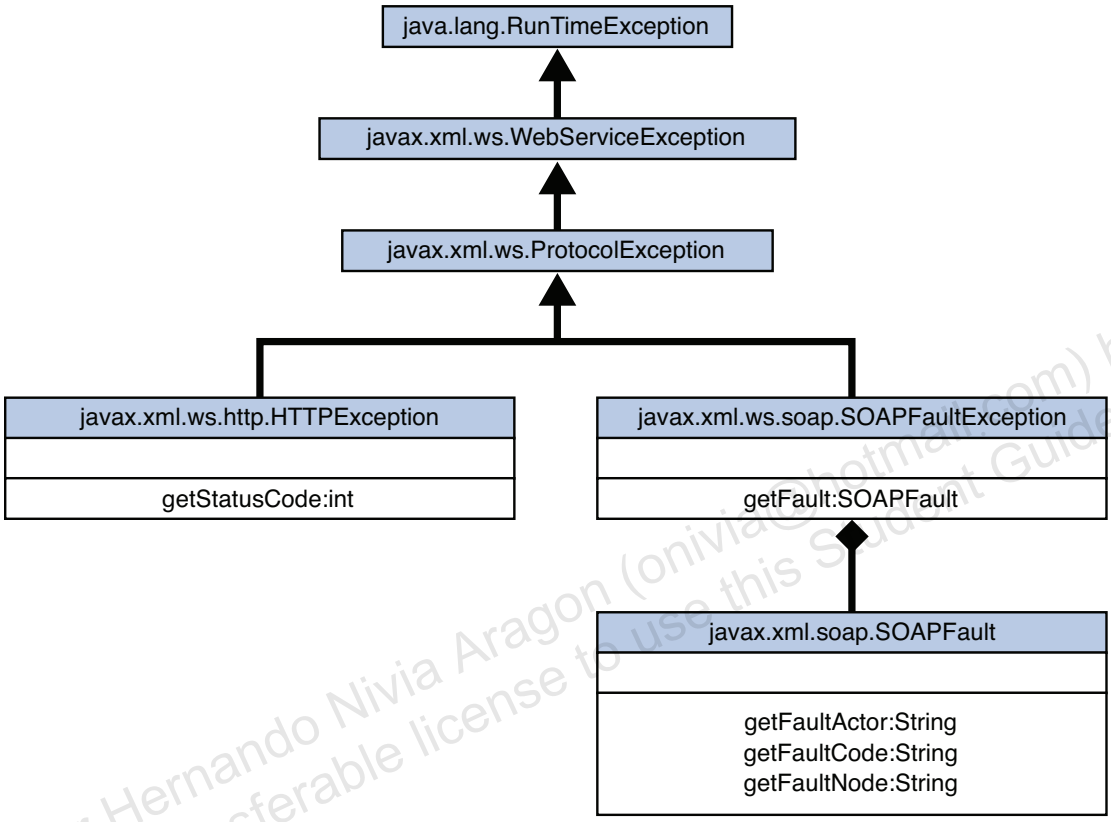


Figure 5.1: JAX-WS Exception Class Hierarchy

The `SOAPFaultException` Object

The `SOAPFaultException` object represents a SOAP 1.1 or 1.2 fault.
A `SOAPFaultException` object wraps a SAAJ `SOAPFault` that manages the SOAP-specific representation of faults. The `createFault` method of the `SOAPFactory` creates an instance of `SOAPFault` for use with the constructor. `SOAPBinding` contains an accessor for the instance of `SOAPFactory` used by the binding instance. The value of `getFault` is the only part of the exception used when serializing a SOAP fault.

The server constructs a `SOAPFaultException` object to generate exceptions for the client use. The `SOAPFaultException` class defines the following constructor:

```
public SOAPFaultException(javax.xml.soap.SOAPFault fault)
```

```
1 public void getWeatherStatus() {
2   try {
3     // business logic to access a web service method
4   } catch (SOAPFaultException sfe) {
5     SOAPFault fault = sfe.getFault();
6   } catch (Exception ex) {
7     ex.printStackTrace();
8   }
9 }
```

Code 5.11: Client-Side `SOAPFaultException` Exception Handling

This constructor constructs a `SOAPFaultException` object with fault as a parameter that represents the SOAPFault object.

The HTTPException Object

The `HTTPException` represents an XML/HTTP fault. Because there is no standard format for faults or exceptions in XML/HTTP messaging, only the HTTP status code is captured and carried to the client.

The `HTTPException` object is useful for REST-ful web services because it uses HTTP as an application-layer protocol rather than transport. You just need to throw a new `HTTPException` object with HTTP status code 500 as the argument for the client use. The client can retrieve the status code and respond appropriately.

Using Predefined Exception Classes in Web Services

When an exception occurs as a result of a protocol-level fault, the web service must ensure that the exception is an instance of the appropriate subclass of `ProtocolException`. For example, for all the SOAP-related faults, the appropriate `ProtocolException` subclass is `SOAPFaultException` , and for XML/HTTP related faults, it is `HTTPException`.

Code 5.11 illustrates the client-side `SOAPFaultException` handling. In the try block of the web service client method, you write the business logic to call the web service. In the catch block, you write the code to handle the exception by making use of the `SOAPFaultException` and `SOAPFault` classes.

Code 5.12 illustrates the server-side `SOAPFaultException` handling. In the web service method, you write the business logic of the web service, which might

```
1  public String getConditions(String locationName)
2  throws WebServiceException {
3    if( some problem ) {
4      try {
5        SOAPFactory factory = SOAPFactory.newInstance();
6        SOAPFault fault = factory.createFault();
7        fault.setFaultActor("fault actor");
8        fault.setFaultCode(factory.createName("local name",
9                                              "prefix", "uri"));
10       fault.setFaultString("fault string");
11       throw new SOAPFaultException(fault);
12     } catch (SOAPException se) {
13       throw new WebServiceException(se.getMessage());
14     }
15   }
16 }
```

Code 5.12: Server-side `SOAPFaultException` Exception Handling

generate an exception while being accessed from a web service client. You make use of the `SOAPFactory` and `SOAPFault` class along with the `SOAPFaultException`, `SOAPException` and `WebServiceException` classes to handle the exception.

Using Custom-Defined Exception Classes in Web Services

Just like any Java or Java EE application, a web service application can encounter an error condition while processing a client request. A web service application needs to properly catch any exceptions thrown by an error condition and propagate these exceptions. For a Java application running in a single virtual machine, you can propagate exceptions up the call stack until reaching a method with an exception handler that handles the type of exception thrown. To put it another way, for non-Web service Java EE and Java applications, you can continue to throw exceptions up the call stack, passing along the entire stack trace, until reaching a method with an exception handler that handles the type of exception thrown. You can also write exceptions that extend or inherit other exceptions.

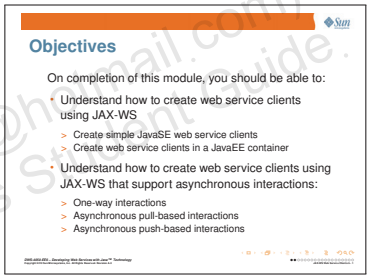
However, throwing exceptions in web service applications has additional constraints that impact the design of the service endpoint. When considering how the service endpoint handles error conditions and notifies clients of errors, you must keep in mind these points:

- Similar to requests and responses, exceptions are also sent back to the client as part of the SOAP messages.
- Your web service application should support clients running on non-Java platforms that might not have the same, or even similar, error-handling mechanisms as the Java exception-handling mechanism.

Chapter 6

JAX-WS Web Service Clients

On completion of this module, you should be able to:



- Understand how to create web service clients using JAX-WS
  - Create simple JavaSE web service clients
  - Create web service clients in a JavaEE container
- Understand how to create web service clients using JAX-WS that support asynchronous interactions:
  - One-way interactions
  - Asynchronous pull-based interactions
  - Asynchronous push-based interactions

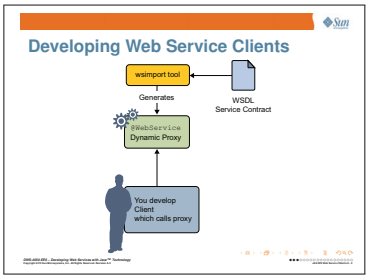
Web Service Clients

Clients can take advantage of web services to obtain a wide range of functions or services. The client does not have to know how the service is implemented or even who provides it. The client primarily cares about the functionality (the service) provided by the web service. Examples of web services include order tracking services, information look-up services, and credit card validation services. Various clients running on different types of platforms can all access these web services.

Interoperability is the primary advantage for using web services as a means of communication. Web services give clients the ability to interoperate with almost any type of system and application, regardless of the platform on which the system or application runs. In addition, the client can use a variety of technologies for this communication.

Web services use HTTP as the transport protocol, which enables clients to operate with systems through firewalls. The service's WSDL document enables clients and services that use different technologies to map and convert their respective data objects. For services and clients that are based on JAX-WS, the JAX-WS runtime handles this mapping transparently. The interoperability provided by well-designed web services means that the services can be accessed from a wide variety of client types operating in different environments and created using various development tools and programming languages.

Although the web service client does not need to know anything about the implementation of the web service it will call on, it does need to know something about how to interact with that service. This information is provided to the client via the WSDL description of the service, which contains a description of the service provided:



- A list of the operations provided, with parameters and return types.
- Security and other restrictions associated with how the interaction with the service must take place.

Since the WSDL description of the service is both platform- and language-independent, it must be translated into a form that can help the implementation of the web service client. In JAX-WS, this translation is performed by `wsimport`, which produces a Java implementation of proxies to the web services described by the WSDL file, which can then be used by the client application to interact

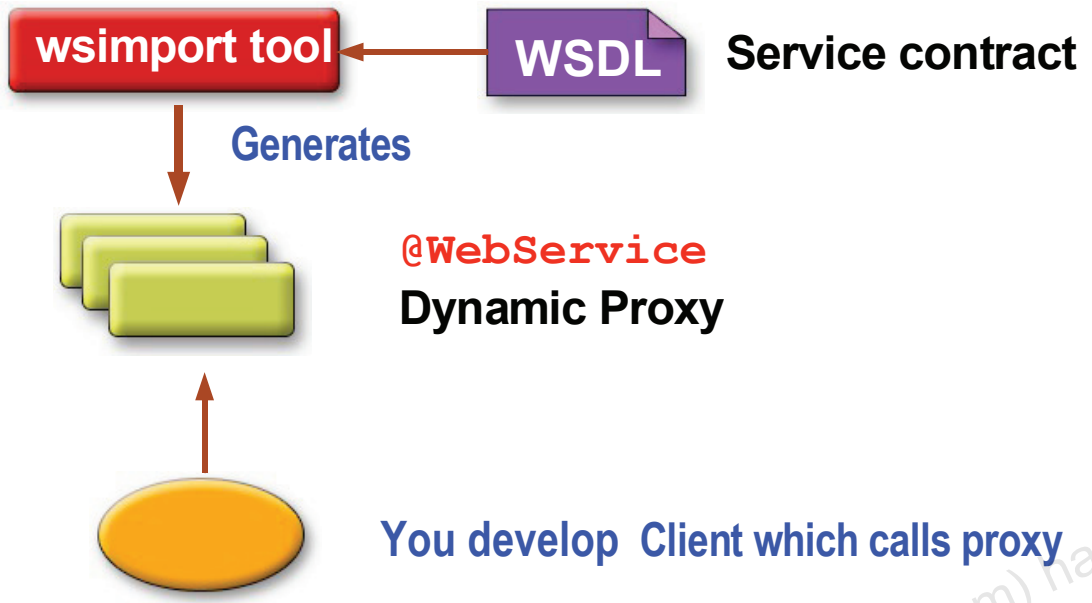


Figure 6.1: Developing Web Service Clients

with those services. Application client developers simply use these proxies, as they would use any other Java objects available to them.

Figure 6.1 illustrates this process.

The `wsimport` tool can be run in a couple of different ways:

- As a command-line tool: `wsimport` *wsdlFileOrURL*
  Arguments to this tool include:

  -b To specify customization files to incorporate.

  -d Target directory for generated code.

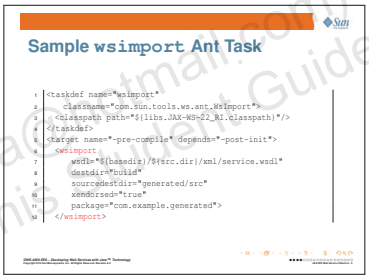
  -keep To keep source version of generated code.

- As an `ant` task. Code 6.1 lists an example of what such an `ant` task might look like. This ant task accepts arguments equivalent to those accepted by the command-line tool.

```
<taskdef name="wsimport"
    classname="com.sun.tools.ws.ant.WsImport">
  <classpath path="${libs.JAX-WS-22_RI.classpath}"/>
</taskdef>
<target name="-pre-compile" depends="-post-init">
  <wsimport
      wsdl="${basedir}/${src.dir}/xml/service.wsdl"
      destdir="build"
      sourcedestdir="generated/src"
      xendorsed="true"
      package="com.example.generated">
  </wsimport>
```

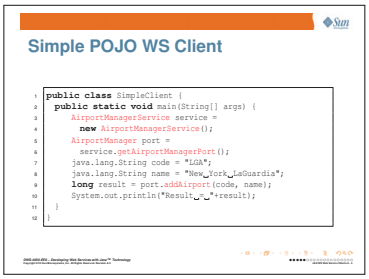
Code 6.1: Fragment of client-side `ant` build script.

Code 6.1 shows an example of the `ant` tasks that could be used to include the generation of client-side artifacts as part of building an application. The `wsimport` task is not built into `ant`, so `taskdef` is used to define it. The `wsimport` task then generates the artifacts needed.



Using the `xendorsed` option shown, along with the `classname` option, may be required to ensure that the right version of `wsimport` is used. JAX-WS has been built into Java since JDK 1.6 – but the version built in may not be the lastest.

Once the client-side code to support interaction with a web service is generated, the client-side logic to interact with that service is straightforward: the client just needs to obtain a reference to the proper web service proxy (known as a *port*), then use it as it would any other Java object. The one catch is that these proxy objects cannot just be created via `new` expressions – a factory type (known as the *service* type) is provided. Code 6.2 lists a simple Java client.



Developers can build web services either "contract-first" (starting from WSDL descriptions of the services), or "code-first" (starting from the Java implementation of the web service provider). One of the advantages to starting from the WSDL description of the service is that it is possible to be more precise about the data values that are acceptable to the service: while Java may be able to

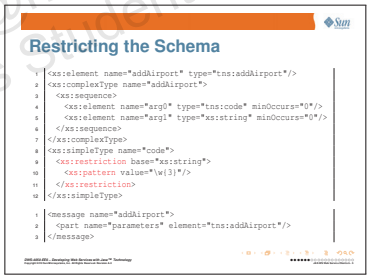
```
1  public class SimpleClient {
2    public static void main(String[] args) {
3        AirportManagerService service =
4          new AirportManagerService();
5        AirportManager port =
6          service.getAirportManagerPort();
7        java.lang.String code = "LGA";
8        java.lang.String name = "New York LaGuardia";
9        long result = port.addAirport(code, name);
10       System.out.println("Result = "+result);
11   }
12 }
```

E-74

Code 6.2: Simple POJO WS Client

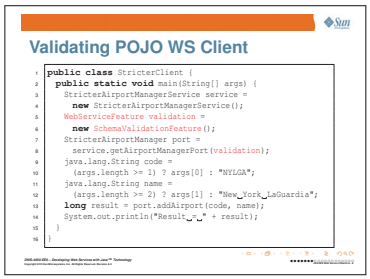
describe an expected value as a `String`, the WSDL description can use XML schema restrictions to further narrow what constitutes legal values.

Code 6.3 lists what the WSDL description of a service could look like, when restricting what legal values can be, using XML schema types. The code shown includes one of the two messages that make up a legal operation, and the XML schema type for the value to be used in that message. The important point is that the type is not just a string: the XML restriction only allows strings that are made up of exactly three letters!



The default behavior of the JAX-WS runtime is to skip validation, even if the WSDL description of the service includes such restrictions.

However, it is possible to enable such validation, if required. Code 6.4 shows how this feature is enabled: the JAX-WS API has a mechanism to represent *features* present in JAX-WS; version of the `getPort()` function admit parameters that are the features to be enabled for this particular port: the instance of `SchemaValidationFeature` obtained in line 6 is used to request validation in the port created in line 8.



Browser-based applications rely on Java EE components, which act as clients of the web service. They run in either an Enterprise JavaBeans (EJB) container or a web container, and these containers manage the client environment. The Java

```
<xs:element name="addAirport" type="tns:addAirport"/>
<xs:complexType name="addAirport">
  <xs:sequence>
    <xs:element name="arg0" type="tns:code" minOccurs="0"/>
    <xs:element name="arg1" type="xs:string" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
<xs:simpleType name="code">
  <xs:restriction base="xs:string">
    <xs:pattern value="\w{3}"/>
  </xs:restriction>
</xs:simpleType>
```

E-75

```
<message name="addAirport">
  <part name="parameters" element="tns:addAirport"/>
</message>
```

E-76

Code 6.3: Restricting the Schema

```
public class StricterClient {
  public static void main(String[] args) {
    StricterAirportManagerService service =
      new StricterAirportManagerService();
    WebServiceFeature validation =
      new SchemaValidationFeature();
    StricterAirportManager port =
      service.getAirportManagerPort(validation);
    java.lang.String code =
      (args.length >= 1) ? args[0] : "NYLGA";
    java.lang.String name =
      (args.length >= 2) ? args[1] : "New York LaGuardia";
    long result = port.addAirport(code, name);
    System.out.println("Result = " + result);
  }
}
```

E-78

Code 6.4: Validating POJO WS Client

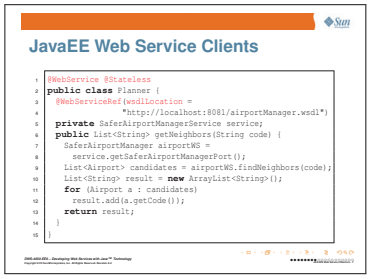
```
@Stateless @WebService
public class UserDirectoryBean
  implements UserDirectoryRemote {
    @WebServiceRef
    (wsdlLocation = "http://localhost/securityHelper?WSDL")
    private HelperService service;
    public boolean testPassword(String value)
      throws Exception {
        com.example.security.server.Helper port =
          service.getHelperPort();
        double result = port.passwordStrength(value);
        return result > 0.75;
    }
}
```

E-79

Code 6.5: JavaEE Web Service Clients

EE platform provides advantages, such as declarative security, transactions, and instance management.

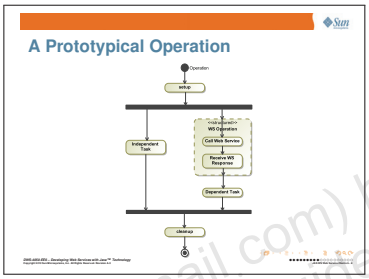
The Java EE platform provides an environment that supports client application access to web services. In a Java EE environment, the deployment descriptors declaratively define the client-side web service configuration information. The deployer can change this configuration information at deployment. In addition, the Java EE platform handles the underlying work for creating and initializing the access to the web services.



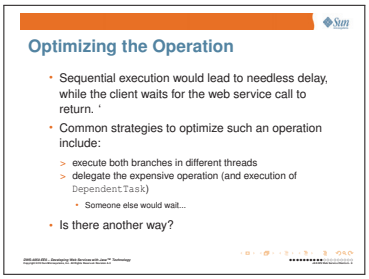
Code 6.5 lists a simple JavaEE web service client. Just as JavaSE clients, the JavaEE client uses a service type as a factory to obtain the port instances that will serve as proxies for the remote web services. However, service types are obtained differently: the service type is injected automatically, for any field in any managed class (such as EJBs, or servlets) annotated with the **@WebServiceRef** annotation. The **wsdlLocation** attribute to the annotation allows the code to specify where the WSDL description for the service can be found.

Asynchronous Interactions

The JAX-WS Web Services framework is designed to make interactions with web services as transparent as possible: on the server side, the web service implementation class could be effectively just a POJO; on the client side, the code interacts with the web service via a proxy object that is meant to be indistinguishable from an API perspective from the original web service implementation class.



Things become more interesting, however, when one imagines a more complex interaction between client and server, in which the client could need to perform several tasks that may be independent of each other, and where one of those could be a call to a remote web service. Figure 6.2 illustrates such a complex interaction, in which the client has three tasks to perform: one is a sequence involving a call to a remote web service that is followed by a second task that depends on the result of that web service call, while the other is a completely independent task.



Sequential execution would lead to needless delay for the client, while it waits for the web service call to return – during this time, the client could have been executing third independent task. Common strategies to optimize such an operation include:

- Execute both branches in different threads
- Delegate the expensive operation (and execution of `DependentTask`) to someone else – but the client may end up having to wait for its delegate, anyway.

The JAX-WS framework supports an API that is designed to optimize execution of such tasks.

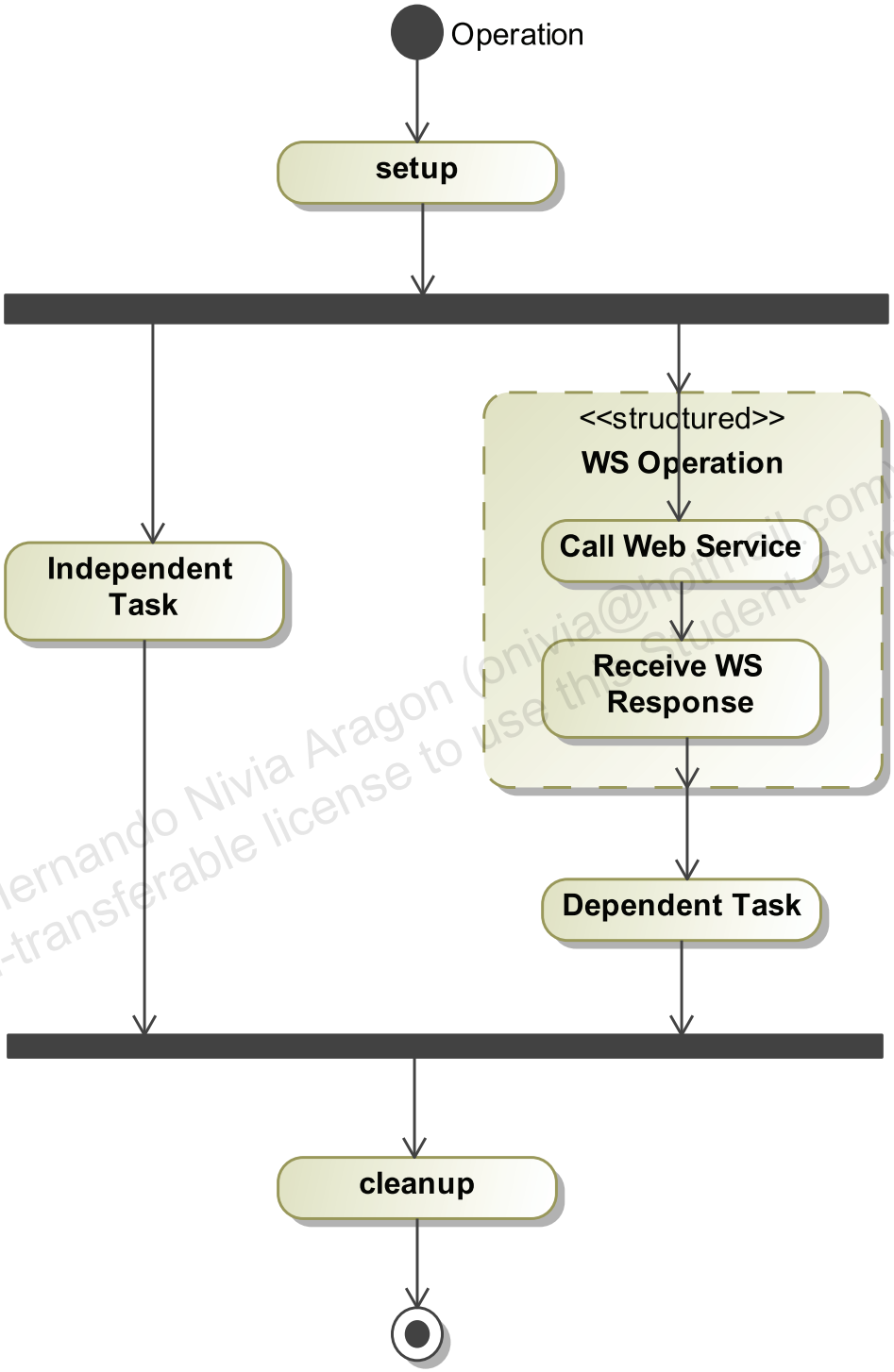
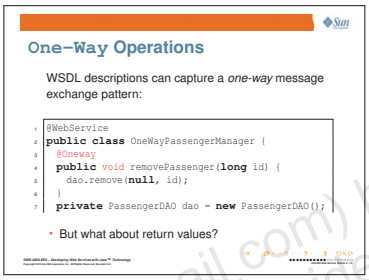


Figure 6.2: A Prototypical Operation

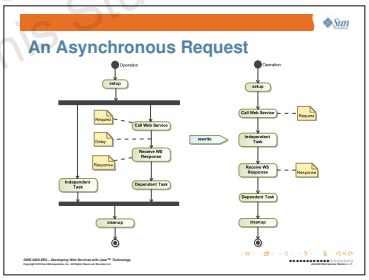
```
1  @WebService
2  public class OneWayPassengerManager {
3    @Oneway
4    public void removePassenger(long id) {
5      dao.remove(null, id);
6    }
7    private PassengerDAO dao = new PassengerDAO();
```

E-80

Code 6.6: One-Way Operations



WSDL descriptions can capture a *one-way* message exchange pattern. In such interactions, the client issues a request to the server, but there is no reply from the server provider back to the client. In such an interaction, there may not be a reason for the client to wait, since there is no answer to be delivered. Code 6.6 shows a web service provider who makes a "one-way" assertion about a particular operation on that service.



The simplest way to optimize the operation illustrated in Figure 6.2 is to notice that the delay to avoid is the one that is introduced between the call to the remote webs service, and the receipt of the return value it produces. The web service proxy represents the call and its return as a single Java method call - but in such a representation, there is no opportunity to incorporate additional logic between those two steps. If the web service interaction could be represented as two separate actions, one could then schedule the execution of that independent task between the call to the remote service and the receipt of its return value, as illustrated in Figure 6.3.

The ability to separate call to remote service from receipt of return value is one of the features provided by the JAX-WS asynchronous client API.

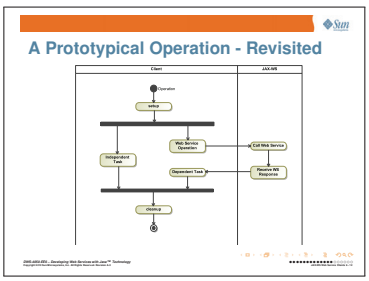


Figure 6.4 is another way to describe the prototypical interaction first introduced in Figure 6.2 above. In Figure 6.4, however, the figure illustrates that the call to the web service that looks like an atomic operation to the application level logic, is in fact implemented as a call to the remote service by the JAX-WS runtime, which then has to wait for the return value to arrive, in order to deliver it to the caller.

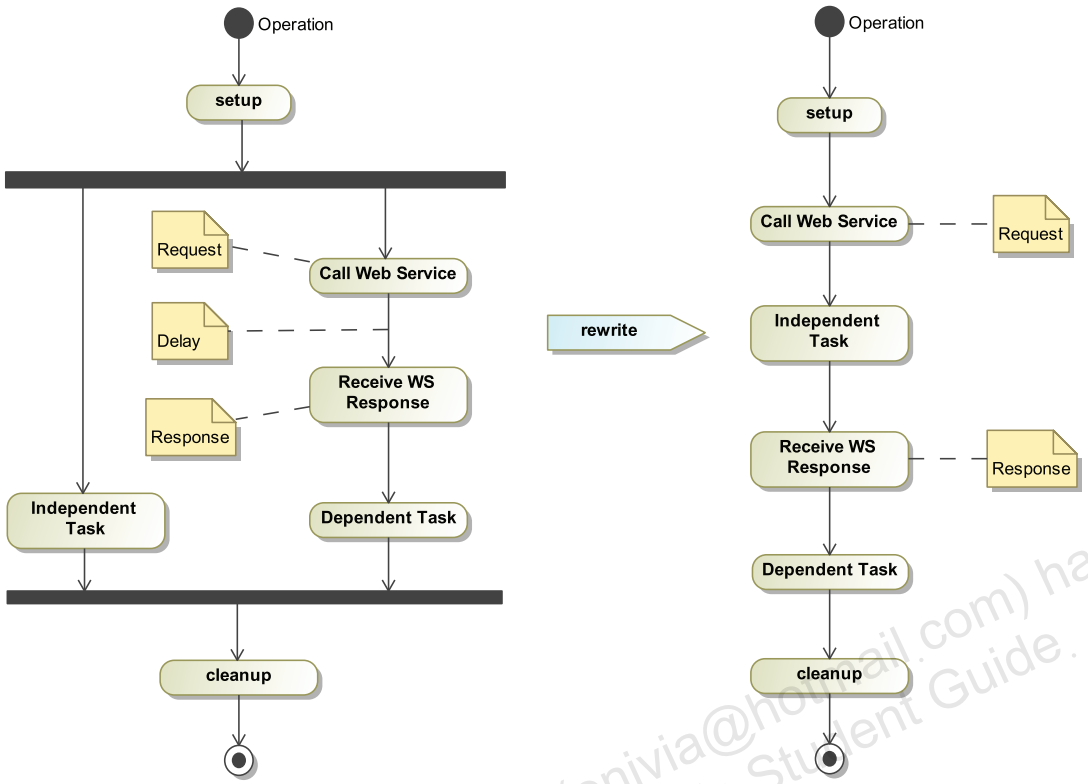
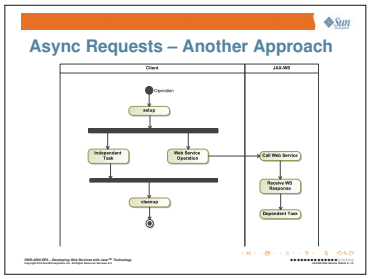


Figure 6.3: An Asynchronous Request

The client would only need to wait for that return value, if it needed to proceed then with the dependent task. If no such task existed, however, the client could just delegate the call to the web service to the JAX-WS runtime, and simply proceed with the independent task, without any need to wait for the remote web service to complete. This scenario can be captured in WSDL, by describing the operation in terms of a single input message, without a matching output message.



There is other alternative available, to optimize the task first presented in Figure 6.2: there may be a dependent task that requires the result returned by the call to the web service – but perhaps that dependent task is not tied to any further processing for the original task. In such a case, one could imagine that, rather than requiring that the JAX-WS runtime notify the caller that the remote call has returned a result, so that the dependent task can then be executed, the caller could turn things around: delegate execution of the de-

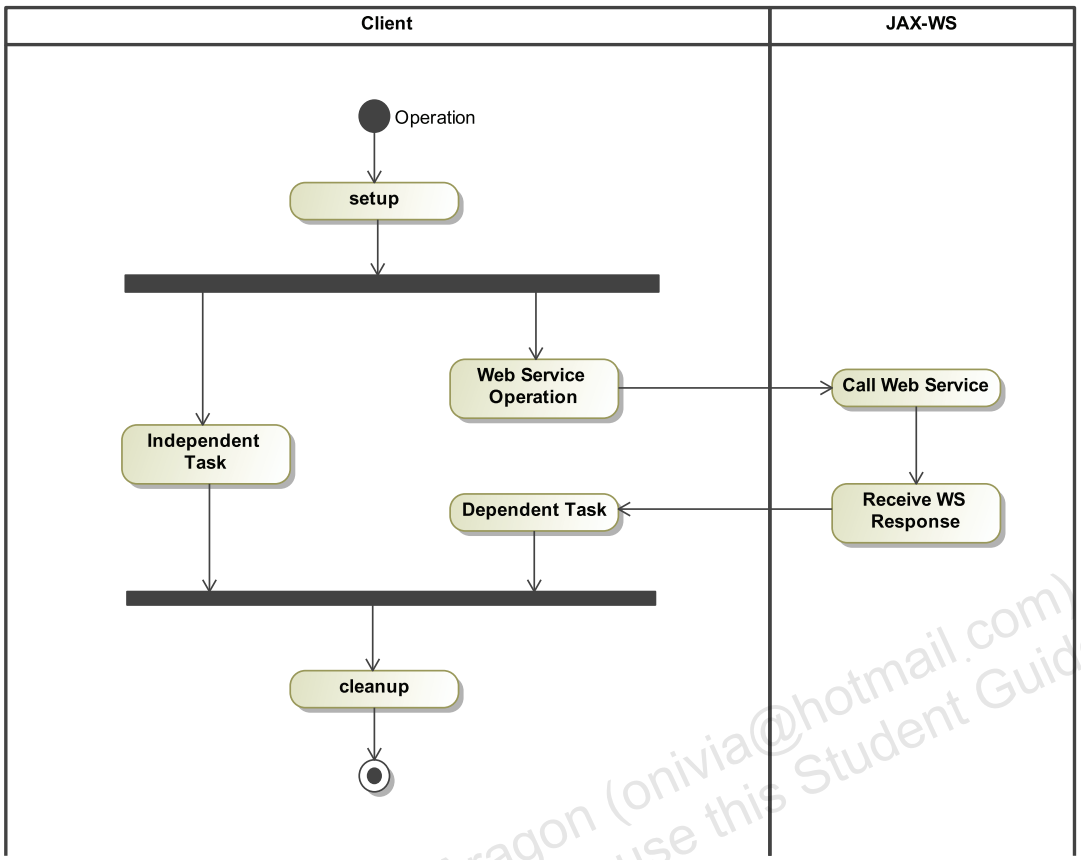
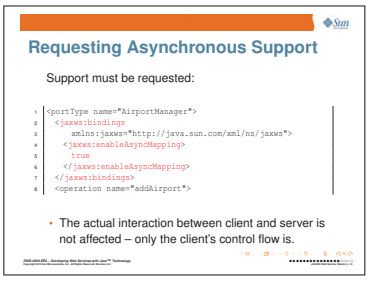


Figure 6.4: A Prototypical Operation – Revisited

pendent task to the JAX-WS runtime, when that remote call eventually returns, without involving the caller any further. This other alternative is illustrated in Figure 6.5.

The JAX-WS asynchronous API also provides support for this last approach.



An important observation about the JAX-WS asynchronous API is that it is purely a client-side framework: calls to the remote service execute as they always would, over on the server side. Only the way in which the client invokes that remote service is affected by whether (and how) the client-side application uses this asynchronous API. Because it is purely a client-side phenomenon, the WSDL description of the web service is not normally affected by the way in which the client uses that service. However – JAX-WS requires to be notified if support for this asynchronous API needs to be incorporated into the client-side artifacts it will generate. This notification can be incorporated into the

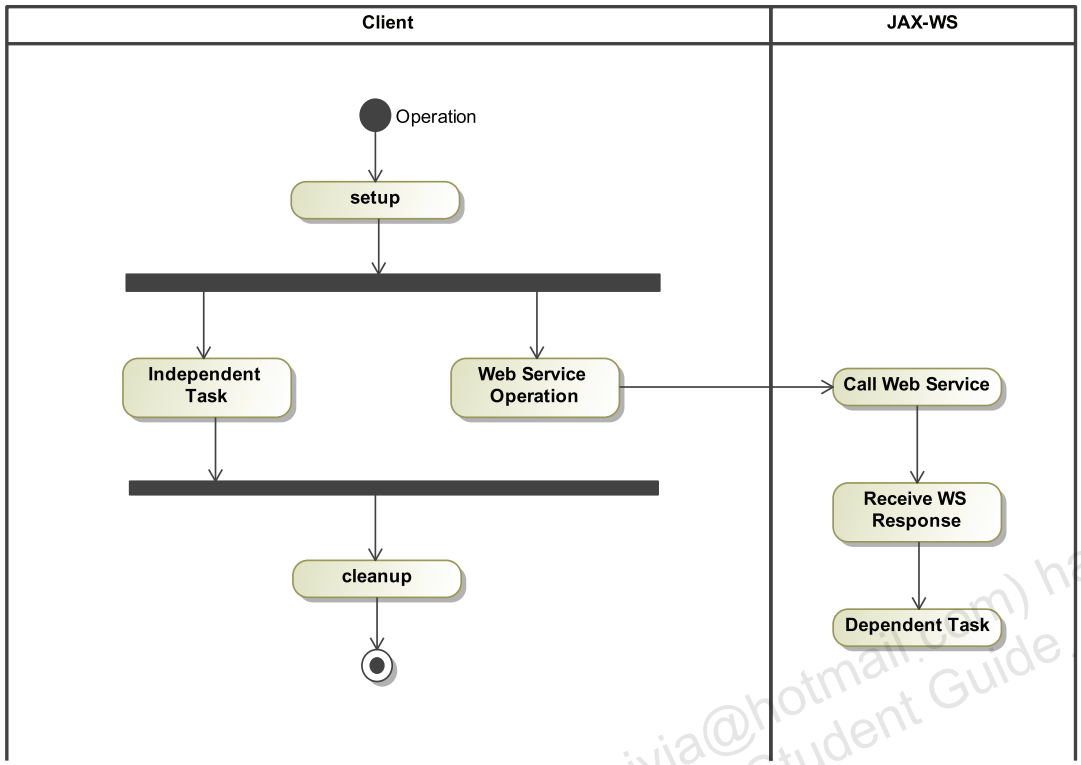
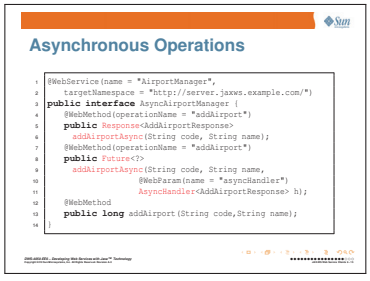


Figure 6.5: Async Requests – Another Approach

WSDL description as a client-side extension, to be ignored on the server side – as shown in Code 6.7. This WSDL code fragment could appear:

- As part of the WSDL description of the service on the server side, ignored on the server, but delivered to any clients who request the WSDL description directly from the server.
- As part of the WSDL description given to clients, separate from the WSDL description used on the server side.
- As a WSDL customization, incorporated by the client "on top of" the original WSDL description provided by the server.

When the WSDL description of a web service includes a request to enable asynchronous interactions with the web service, the Java interface generated as a client-side artifact to describe the proxy class is extended with two additional methods, which capture the scenarios described above.

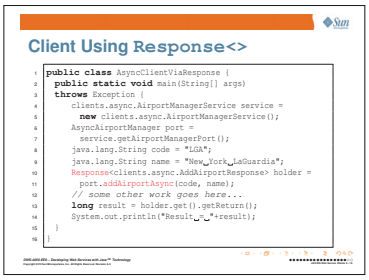


```
<portType name="AirportManager">
  <jaxws:bindings
      xmlns:jaxws="http://java.sun.com/xml/ns/jaxws">
    <jaxws:enableAsyncMapping>
      true
    </jaxws:enableAsyncMapping>
  </jaxws:bindings>                                    E-81
  <operation name="addAirport">
```

Code 6.7: Requesting Asynchronous Support

```
@WebService(name = "AirportManager",
    targetNamespace = "http://server.jaxws.example.com/")
public interface AsyncAirportManager {
  @WebMethod(operationName = "addAirport")
  public Response<AddAirportResponse>
    addAirportAsync(String code, String name);
  @WebMethod(operationName = "addAirport")
  public Future<?>
    addAirportAsync(String code, String name,
                    @WebParam(name = "asyncHandler")
                    AsyncHandler<AddAirportResponse> h);
  @WebMethod
  public long addAirport(String code,String name);
}                                                      E-83
```

Code 6.8: Asynchronous Operations



Code 6.9 shows a simple JAX-WS client that uses the simpler asynchronous API: the client invokes a method on the web service using one of the async calls (line 11), and obtains immediately a <Response> wrapper around the (eventual) return value (line 10). This call returns to the caller immediately, though, while in the background the JAX-WS client runtime initiates the call to the remote and waits for the answer. While the JAX-WS runtime is waiting, however, the client is free to perform other work (line 12). Eventually, the client could retrieve the answer (line 13), and continue work with that answer now available.

```
1  public class AsyncClientViaResponse {
2    public static void main(String[] args)
3    throws Exception {
4        clients.async.AirportManagerService service =
5          new clients.async.AirportManagerService();
6        AsyncAirportManager port =
7          service.getAirportManagerPort();
8        java.lang.String code = "LGA";
9        java.lang.String name = "New York LaGuardia";
10       Response<clients.async.AddAirportResponse> holder =
11         port.addAirportAsync(code, name);
12       // some other work goes here...
13       long result = holder.get().getReturn();
14       System.out.println("Result = "+result);
15   }
16 }
```

E-85

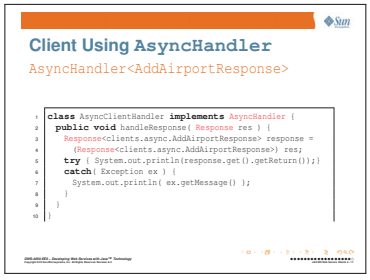
Code 6.9: Client Using Response

```
1  class AsyncClientHandler implements AsyncHandler {
2    public void handleResponse( Response res ) {
3      Response<clients.async.AddAirportResponse> response =
4        (Response<clients.async.AddAirportResponse>) res;
5      try { System.out.println(response.get().getReturn());}
6      catch( Exception ex ) {
7        System.out.println( ex.getMessage() );
8      }
9    }
10 }
```

E-86

Code 6.10: Client Using AsyncHandler – AsyncHandler

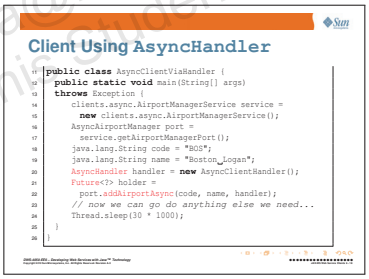
In order to delegate the dependent task to the JAX-WS runtime altogether, the application must define a class that implements `AsyncHandler`, to capture that dependent task. Code 6.10 is a very simple example of such a handler: it just received the value returned from the service, when available, and prints it out: the call to `get()` in line 5 will block until the answer arrives, if necessary.



```
11 public class AsyncClientViaHandler {
12   public static void main(String[] args)
13   throws Exception {
14       clients.async.AirportManagerService service =
15         new clients.async.AirportManagerService();
16       AsyncAirportManager port =
17         service.getAirportManagerPort();
18       java.lang.String code = "BOS";
19       java.lang.String name = "Boston Logan";
20       AsyncHandler handler = new AsyncClientHandler();
21       Future<?> holder =
22         port.addAirportAsync(code, name, handler);
23       // now we can go do anything else we need...
24       Thread.sleep(30 * 1000);
25   }
26 }
```

E-86

Code 6.11: Client Using AsyncHandler – Client

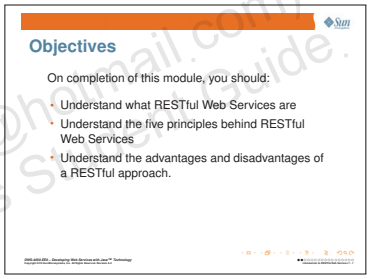


Code 6.11 shows another JAX-WS client – but one that uses the more sophisticated asynchronous API, in which the caller delegates the dependent task to the JAX-WS runtime. In this example, the client creates an instance of an `AsyncHandler` (in line 20), and passes it as the extra argument to the call to the web service in line 22. That call to web service will return immediately, allowing the client to proceed to do whatever else may be required. While that other processing goes on, the JAX-WS runtime will wait for the return value to be delivered back to the client. When that happens, the JAX-WS runtime will invoke the `AsyncHandler` provided, with the value returned as a parameter, to execute the dependent task represented by that handler.

Chapter 7

Introduction to RESTful Web Services

On completion of this module, you should:



- Understand what RESTful Web Services are.
- Understand the five principles behind RESTful Web Services.
- Understand the advantages and disadvantages of a RESTful approach.

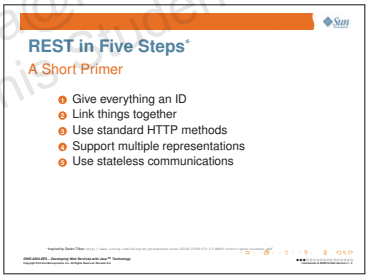
What are RESTful Web Services?

RESTful web services are services that are built to work best on the web. Representational State Transfer (REST) is an architectural style that specifies constraints, such as the uniform interface, that if applied to a web service induce desirable properties, such as performance, scalability, and modifiability, that enable services to work best on the Web.

In the REST architectural style, data and functionality are considered resources, and these resources are accessed using Uniform Resource Identifiers (URIs), typically links on the web. The resources are acted upon by using a set of simple, well-defined operations.

The REST architectural style constrains an architecture to a client-server architecture, and is designed to use a stateless communication protocol, typically HTTP. In the REST architecture style, clients and servers exchange representations of resources using a standardized interface and protocol.

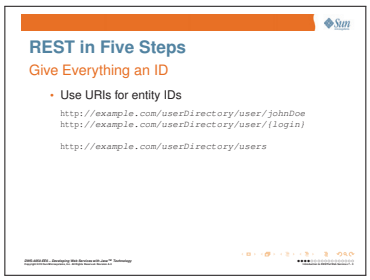
These principles encourage RESTful applications to be simple, lightweight, and have high performance:



1. Give everything an ID
2. Link things together
3. Use standard HTTP methods
4. Support multiple representations
5. Use stateless communications

Inspired by Stefan Tilkov:
http://www.innoq.com/blpresentations/2008/2008-03-13-REST-Intro-QCon-London.pdf

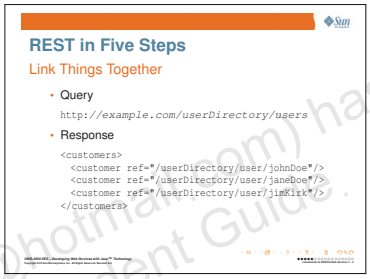
A RESTful Web service exposes a set of resources which identify the targets of the interaction with its clients. Resources are identified by URIs, which provide a global addressing space for resource and service discovery:




```
http://example.com/userDirectory/users

http://example.com/userDirectory/user/johnDoe
http://example.com/userDirectory/user/{login}
```

RESTful web services are expected to obey the principle of "progressive disclosure" – a web service should never return large, complex representations of resources at once. The proper thing to do is to return a "summary" representation of the resources in question, with links that will allow the client to navigate its way to the specific detail they need – much like people do, when they surf the web. For example, the query:
`http://example.com/userDirectory/users`
should result in




```
<customers>
  <customer ref="/userDirectory/user/johnDoe"/>
  <customer ref="/userDirectory/user/janeDoe"/>
</customers>
```

Resources are manipulated using a fixed set of four create, read, update, delete operations: PUT, GET, POST, and DELETE. PUT creates a new resource, which can be then deleted using DELETE. GET retrieves the current state of a resource in some representation. POST transfers a new state onto a resource.

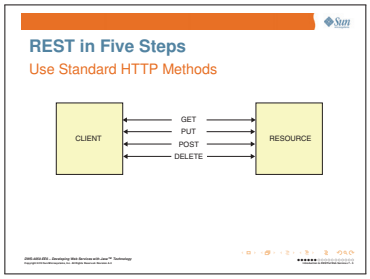
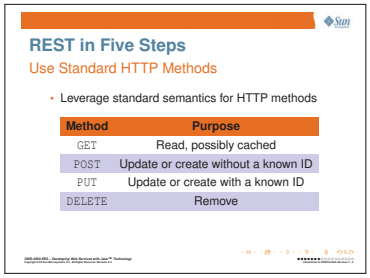


Table 7.1 indicates what the standard intent behind each HTTP method is. RESTful web services are expected to abide by these semantics.



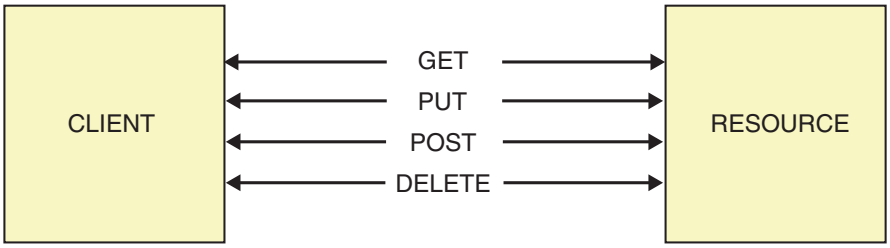
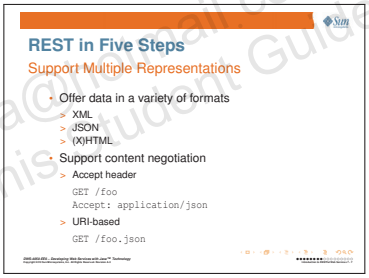


Figure 7.1: Standard HTTP Methods

Method	Purpose
GET	Read, possibly cached
POST	Update or create without a known ID
PUT	Update or create with a known ID
DELETE	Remove

Table 7.1: Standard Semantics for HTTP Methods

Resources are decoupled from their representation so that their content can be accessed in a variety of formats (such as HTML, XML, plain text, PDF, JPEG, JSON, and others). Metadata about the resource is available and used, for example, to control caching, detect transmission errors, negotiate the appropriate representation format, and perform authentication or access control.



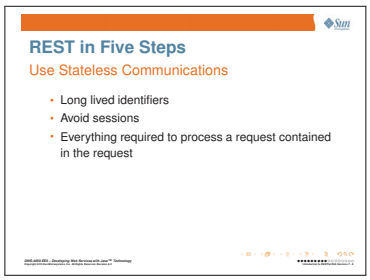
- Offer data in a variety of formats
  - XML
  - JSON
  - (X)HTML
- Support content negotiation
  - Accept header

```
GET /foo
Accept: application/json
```

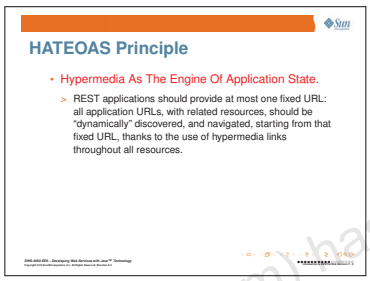
  - URI-based

```
GET /foo.json
```

Every interaction with a resource is stateless: request messages are self-contained. Stateful interactions are based on the concept of *explicit state transfer*. Several techniques exist to exchange state, such as URI rewriting, cookies, and hidden form fields. State can be embedded in response messages to point to valid future states of the interaction



Since the server-side to RESTful interactions is stateless, it is natural to expect that the client will therefore be responsible for maintaining conversational state. The RESTful approach to building web services – the RESTful *architecture* – has a principle that it believes ought to control how such state is represented:

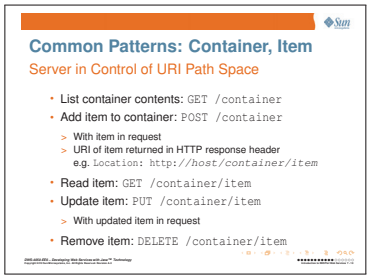


> Hypermedia as the engine of application state

often shortened to its acronym: HATEOAS.

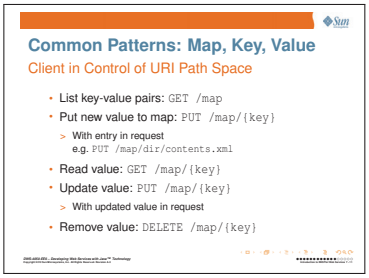
HATEOAS is one of the most important, and often misled/ignored, principles behind a RESTful architecture. HATEOAS could also be named *hypermedia-based navigation*. It means that a REST application should provide at most one fixed URL: all application URLs, with related resources, should be "dynamically" discovered, and navigated, starting from that fixed URL, thanks to the use of hypermedia links throughout all resources – much like people navigate the world-wide web.

The above quoted from the blog at:
`http://sbtourist.blogspot.com/2009/01/jax-rs-and-hateoas.html`.



- List container contents:
  `GET /container`
- Add item to container:
  `POST /container`
  - With item in request
  - URI of item returned in HTTP response header
    e.g. `Location: http://host/container/item`

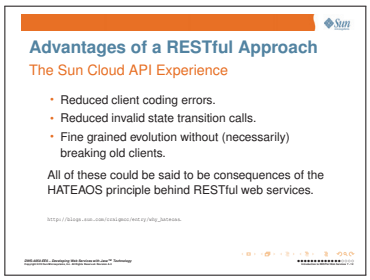
- Read item: `GET /container/item`
- Update item: `PUT /container/item`
  - With updated item in request
- Remove item: `DELETE /container/item`



- List key-value pairs: `GET /map`
- Put new value to map: `PUT /map/{key}`
  - With entry in request
    e.g. `PUT /map/dir/contents.xml`
- Read value: `GET /map/{key}`
- Update value: `PUT /map/{key}`
  - With updated value in request
- Remove value: `DELETE /map/{key}`

Advantages and Disadvantages

On his blog, Craig McClanagan (one of the developers behind the Sun Cloud API), writes about some of the significant, practical, short term benefits that group gained from taking a RESTful approach for that API:



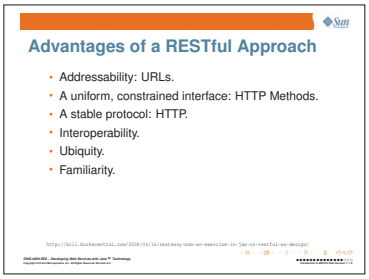
- Reduced client coding errors:
  Looking back at all the REST client side interfaces that they have built, about 90% of the bugs have been in the construction of the right URIs for the server. Typical mistakes are leaving out path segments, getting them in the wrong order, or forgetting to URL encode things. All this goes away when the server hands you exactly the right URI to use for every circumstance.

- Reduced invalid state transition calls:
  When the client decides which URI to call and when, they run the risk of attempting to request state transitions that are not valid for the current state of the server side resource. For example, it's not allowed to "start" a virtual machine (VM) until it's been "deployed". The server knows about URIs to initiate each of the state changes (via a POST), but the representation of the VM lists only the URIs for state transitions that are valid from the current state. This makes it extremely easy for the client to understand that trying to start a VM that has not been deployed yet is not legal, because there will be no corresponding URI in the VM representation.

- Fine grained evolution without (necessarily) breaking old clients:
  At any given time, the client of any REST API is going to be programmed with some assumptions about what the system can do. But, if one document has a restriction to "pay attention to only those aspects of the representation that you know about", plus a server side discipline to add things later that do not disrupt previous behavior, you can evolve APIs fairly quickly without breaking all clients, or having to support multiple versions of the API simultaneously on your server. Compare this with something like SOAP where the syntax of your representations is versioned (in the WSDL), so you have to mess with the clients on every single change.

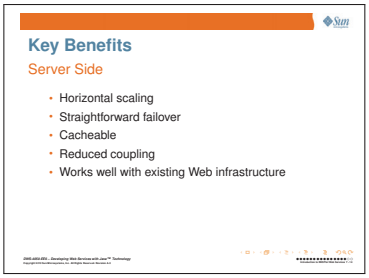
All of these could be said to be consequences of the HATEAOS principle behind RESTful web services.

The above section taken from Craig McClanahan's blog at:
http://blogs.sun.com/craigmcc/entry/why_hateoas.



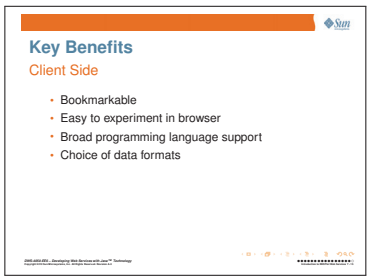
- Addressability: URLs.
- A uniform, constrained interface: HTTP Methods.
- A stable protocol: HTTP.
- Interoperability.
- Ubiquity.
- Familiarity.

http://bill.burkecentral.com/2008/06/16/
resteasy-mom-an-exercise-in-jax-rs-restful-ws-design/

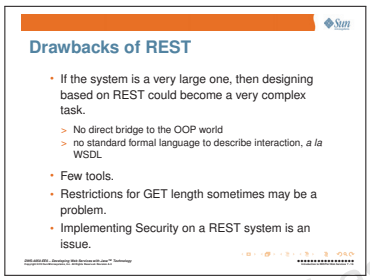


REST architectures are based on the same assumptions that underlie the world-wide web, and so exhibit similar characteristics. REST architectures lead to better (horizontal) scaling, since the stateless nature of interactions means the server need not keep any expensive resources (memory, network connections) tied down to any particular client. Also as a consequence of their stateless nature, failover across servers is also straightforward: each client request will carry all the conversational state required to process that request, so any server in the cluster could process it. Since REST architectures leverage the semantics of HTTP methods, caching the result of operations is also straightforward: `GET` requests are by definition cacheable, others are not.

REST architectures follow the HATEOAS principle, which leads to reduced coupling between client and server: every response from the server carries to the client the set of operations that are legal at that point in their conversation, which reduces the amount of knowledge about the interaction that the client is responsible for up-front.



- Bookmarkable
- Easy to experiment in browser
- Broad programming language support
- Choice of data formats



- If the system is a very large one, then designing based on REST could become a very complex task.
  - No direct bridge to the OOP world
  - no standard formal language to describe interaction, *a la* WSDL
- Few tools.
- Restrictions for GET length sometimes may be a problem.
- Implementing Security on a REST system is an issue.

Jan Algermissen wrote on the Jersey users mailing list:

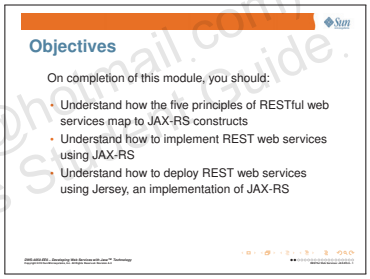
> One of REST's primary aims is to enable the creation of systems that have a very long lifetime (decades) and can be evolved in a decentralized fashion without the need for server owners to consult all the client owners to ask what clients might break by what change of the server.
>
> The startup cost for achieving this is considerably high (proper media type design, proper client programming model) and the associated integration task might not warrant that endeavor: the lifetime might be too short, the decentralization aspect might be too small, etc.
>
> RESTful systems have the property that a person, faced with the task of changing a service implementation really does in no way need to consider what any client out there might be doing – as long as the following principles are followed:
>
> - Keep resource semantics stable and make URIs cool ('cool URIs don't change')
> - Conform to the media type specs
>
> That is a huge advantage because developers need no communication whatsoever with their client owners to change a service.

From a message from Jan Algermissen, `algermissen1971@mac.com` to the Jersey users mailing list, `users@users.dev.java.net` on Thu, Feb 18, 2010 at 5:32 AM.

Chapter 8

RESTful Web Services: JAX-RS

On completion of this module, you should:



- Understand how the five principles of RESTful web services map to JAX-RS constructs
- Understand how to implement REST web services using JAX-RS
- Understand how to deploy REST web services using Jersey, an implementation of JAX-RS

Additional Resources

The following references provide additional information on the topics described in this module:

- JAX-RS Specification, at:
  `https://jsr311.dev.java.net/nonav/releases/1.1/spec/spec.html`.
- Jersey User Guide, at:
  `https://jersey.dev.java.net/nonav/documentation/latest/user-guide.html`.
- Jersey web site, at:
  `https://jersey.dev.java.net`.

JAX-RS

JAX-RS defines a set of Java APIs for the development of Web services built according to the Representational State Transfer (REST) architectural style. Goals of the JAX-RS specification include:

- POJO-based:
  The API will provide a set of annotations and associated classes/interfaces that may be used with POJOs in order to expose them as Web resources. The specification will define object lifecycle and scope.
- HTTP-centric:
  The specification will assume HTTP is the underlying network protocol and will provide a clear mapping between HTTP and URI elements and the corresponding API classes and annotations. The API will provide high level support for common HTTP usage patterns and will be sufficiently flexible to support a variety of HTTP applications including WebDAV and the Atom Publishing Protocol.
- Format independence:
  The API will be applicable to a wide variety of HTTP entity body content types. It will provide the necessary pluggability to allow additional types to be added by an application in a standard manner.
- Container independence:
  Artifacts using the API will be deployable in a variety of Web-tier containers. The specification will define how artifacts are deployed in a Servlet container and as a JAX-WS Provider.
- Inclusion in Java EE:
  The specification will define the environment for a Web resource class hosted in a Java EE container and will specify how to use Java EE features and components within a Web resource class.

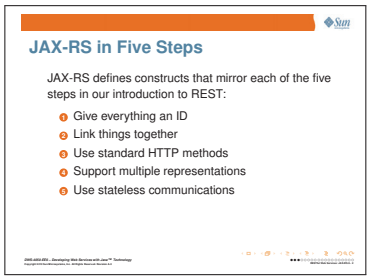
Non-goals of the JAX-RS specification include:

- Support for Java versions prior to J2SE 5.0:
  The API will make extensive use of annotations and will require J2SE 5.0 or later.
- Description, registration and discovery:
  The specification will neither define nor require any service description, registration or discovery capability.
- Client APIs:
  The specification will not define client-side APIs. Other specifications are expected to provide such functionality.

- HTTP Stack:
  The specification will not define a new HTTP stack. HTTP protocol support is provided by a container that hosts artifacts developed using the API.
- Data model/format classes:
  The API will not define classes that support manipulation of entity body content, rather it will provide pluggability to allow such classes to be used by artifacts developed using the API.

Mapping REST Principles to JAX-RS Constructs

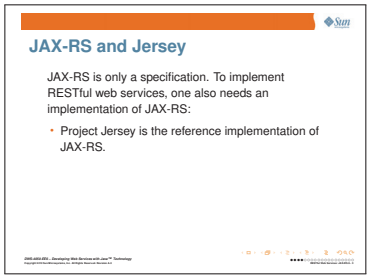
JAX-RS defines constructs that mirror each of the five steps in our introduction to REST:



1. Give everything an ID
2. Link things together
3. Use standard HTTP methods
4. Support multiple representations
5. Use stateless communications

The JAX-RS API uses Java programming language annotations to simplify the development of RESTful web services. Developers decorate Java programming language class files with HTTP-specific annotations to define resources and the actions that can be performed on those resources. Jersey annotations are run-time annotations, therefore, runtime reflection will generate the helper classes and artifacts for the resource, and then the collection of classes and artifacts will be built into a web application archive (WAR).

JAX-RS is only a specification. To implement RESTful web services, one also needs an implementation of JAX-RS:

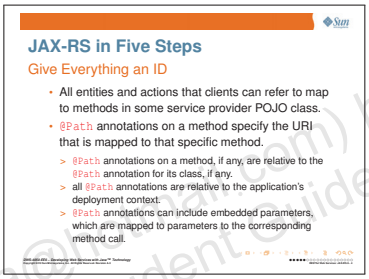


- Project Jersey is the reference implementation of JAX-RS.
- Other implementations of JAX-RS include:
  - Apache CXF
  - RESTeasy
  - RESTlet

```
1  @Path("/airports")
2  public class AirportRM {
3    // ...
4    @Path("/numAirports")
5    public String getNumAirports() {
6      List<String> codes = dao.getAllCodes( null );
7      return
8        (codes == null) ? "0" : "" + codes.size();
9    }
```

E-87

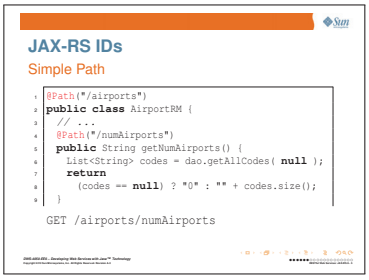
Code 8.1: JAX-RS IDs – Simple Path



- All entities and actions that clients can refer to map to methods in some service provider POJO class.
- **@Path** annotations on a method specify the URI that is mapped to that specific method.
  - **@Path** annotations on a method, if any, are relative to the **@Path** annotation for its class, if any.
  - all **@Path** annotations are relative to the application's deployment context.
  - **@Path** annotations can include embedded parameters, which are mapped to parameters to the corresponding method call.

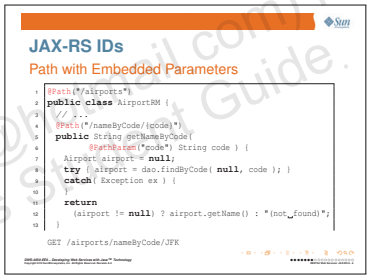
A **@Path** value may or may not begin with a `'/'`, it makes no difference. Likewise, by default, a **@Path** value may or may not end in a `'/'`, it makes no difference. Thus, request URLs will both be matched, regardless of whether they start or end in a `'/'`.

In the simplest scenarios, request URLs are matched against the **@Path** annotations of root resources and their methods, in order to select the proper service provider to instantiate, and the proper method in that provider to invoke, to process each request. Figure 8.1 lists a very simple JAX-RS root resource. Given this class, the request `GET /airports/numAirports` would result in a call to `getNumAirports()` on line 5.



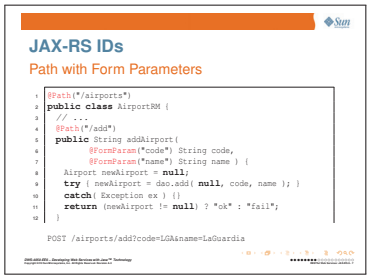
```
1  @Path("/airports")
2  public class AirportRM {
3    // ...
4    @Path("/nameByCode/{code}")
5    public String getNameByCode(
6            @PathParam("code") String code ) {
7      Airport airport = null;
8      try { airport = dao.findByCode( null, code ); }
9      catch( Exception ex ) {
10     }
11     return
12       (airport != null) ? airport.getName() : "(not found)";
13   }
```


Code 8.2: JAX-RS Path with Embedded Parameters



URI *path templates* are URIs with variables embedded within the URI syntax. These variables are substituted at runtime in order for a resource to respond to a request based on the substituted URI. Variables are denoted by curly braces. Figure 8.2 lists a root resource with a method associated with such a path: `"/nameByCode/{code}"` on line 4. When processing the request `GET /airports/nameByCode/JFK`, JAX-RS would bind the parameter `code` to the string `"JFK"`, in order to invoke the method `getNameByCode()`. Template variables have to match the attributes of **`@PathParam`** annotations, so that JAX-RS can map URI values against method parameters.

Parameters of a resource method may be annotated with parameter-based annotations to extract information from a request. In Code 8.2, the **`@PathParam`** parameter is used to extract a path parameter from the path component of the request URL that matched the path declared in **`@Path`**. There are six types of parameters you can extract for use in your resource class: query parameters, URI path parameters, form parameters, cookie parameters, header parameters, and matrix parameters.



`@FormParam` is slightly special because it extracts information from a request representation that is of the MIME media type `"application/x-www-form-urlencoded"` and conforms to the encoding specified by HTML forms. This parameter is very useful for extracting information that is `POST`ed by HTML forms. Figure 8.3 shows how a method could extract the

```
1  @Path("/airports")
2  public class AirportRM {
3    // ...
4    @Path("/add")
5    public String addAirport(
6            @FormParam("code") String code,
7            @FormParam("name") String name ) {
8      Airport newAirport = null;
9      try { newAirport = dao.add( null, code, name ); }
10     catch( Exception ex ) {}
11     return (newAirport != null) ? "ok" : "fail";
12   }
```

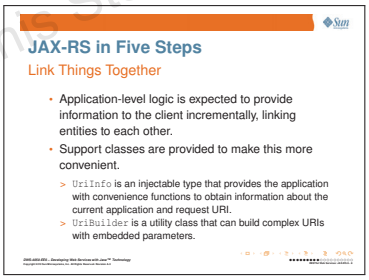
E-87

Code 8.3: JAX-RS Path with Form Parameters

parameters `code` and `name` from the POSTed form data in a request, such as:

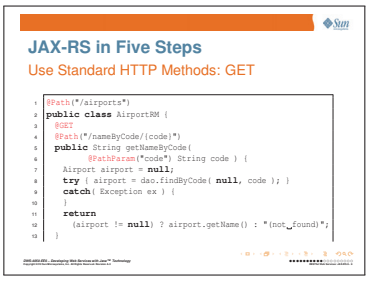
```
POST /airports/add?code=LGA&name=LaGuardia
```

Application-level logic is expected to provide information to the client incrementally, linking entities to each other – and capturing conversational state according to the HATEOAS principle. JAX-RS defines some support classes are provided to make this more convenient:



- `UriInfo` is an injectable type that provides the application with convenience functions to obtain information about the current application and request URI.
- `UriBuilder` is a utility class that can build complex URIs with embedded parameters.

`@PUT`, **`@POST`**, `@DELETE`, and `@HEAD` are resource method designator annotations defined by JAX-RS and which correspond to the similarly named HTTP methods. In the example in Figure 8.4, the annotated Java method will process HTTP GET requests: it will return the name of an airport, given its three-letter code.



The behavior of a resource is determined by which of the HTTP methods the resource is responding to.

```
1  @Path("/airports")
2  public class AirportRM {
3    @GET
4    @Path("/nameByCode/{code}")
5    public String getNameByCode(
6            @PathParam("code") String code ) {
7      Airport airport = null;
8      try { airport = dao.findByCode( null, code ); }
9      catch( Exception ex ) {
10     }
11     return
12       (airport != null) ? airport.getName() : "(not found)";
13   }
```

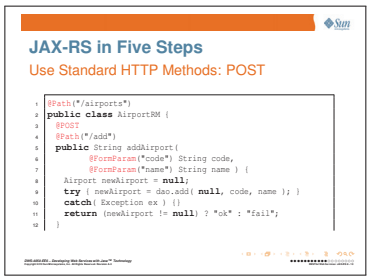
E-87

Code 8.4: Use Standard HTTP Methods: GET

```
1  @Path("/airports")
2  public class AirportRM {
3    @POST
4    @Path("/add")
5    public String addAirport(
6            @FormParam("code") String code,
7            @FormParam("name") String name ) {
8      Airport newAirport = null;
9      try { newAirport = dao.add( null, code, name ); }
10     catch( Exception ex ) {}
11     return (newAirport != null) ? "ok" : "fail";
12   }
```

E-87

Code 8.5: Use Standard HTTP Methods: POST

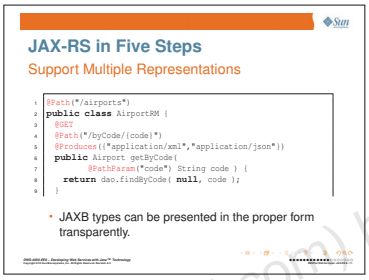


In the example in Figure 8.5, the annotated Java method will process HTTP POST requests: it will add a new airport to the system, given its three-letter code and its name.

Methods decorated with request method designators must return `void`, a Java programming language type, or a `javax.ws.rs.core.Response` object. Methods that need to provide additional metadata with a response should return an instance of `Response`. The `ResponseBuilder` class provides a convenient way to create a `Response` instance using a builder pattern.

By default the JAX-RS runtime will automatically support the methods `HEAD` and `OPTIONS`, if not explicitly implemented. For `HEAD` the runtime will invoke the implemented `GET` method (if present) and ignore the response entity (if set). For `OPTIONS` the the `Allow` response header will be set to the set of HTTP methods support by the resource. In addition, Jersey will return a WADL document describing the resource.

The **`@Produces`** annotation is used to specify the MIME media types of representations a resource can produce and send back to the client. In the example in Figure 8.6, the Java method `getByCode()` (starting on line 3) will produce representations identified by the MIME media type `"text/plain"`. **`@Produces`** can be applied at both the class and method levels.




```
1  @Path("/airports")
2  public class AirportRM {
3    @GET
4    @Path("/byCode/{code}")
5    @Produces({"application/xml","application/json"})
6    public Airport getByCode(
7            @PathParam("code") String code ) {
8      return dao.findByCode( null, code );
9    }
```

E-87

Code 8.6: JAX-RS Support for Multiple Representations

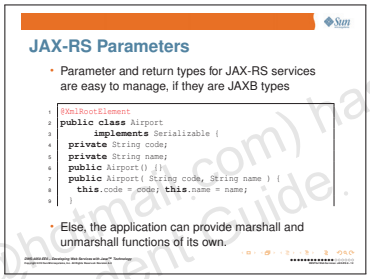
If a resource class is capable of producing more that one MIME media type then the resource method chosen will correspond to the most acceptable media type as declared by the client. More specifically, the `Accept` header of the HTTP request declared what is most acceptable.

The examples in the course refer explicitly to MIME media types for clarity. It is possible to refer to constant values, which may reduce typographical errors – see the constant field members of `MediaType`.

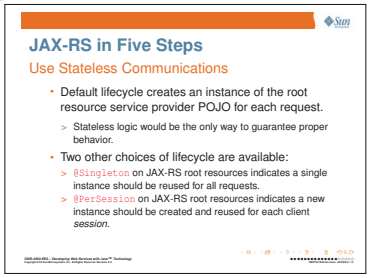
```
1 @XmlRootElement
2 public class Airport
3         implements Serializable {
4   private String code;
5   private String name;
6   public Airport() {}
7   public Airport( String code, String name ) {
8     this.code = code;  this.name = name;
9   }
```


Code 8.7: JAX-RS Parameters

- Parameter and return types for JAX-RS services are easy to manage, if they are JAXB types
- Else, the application can provide marshall and unmarshall functions of its own.



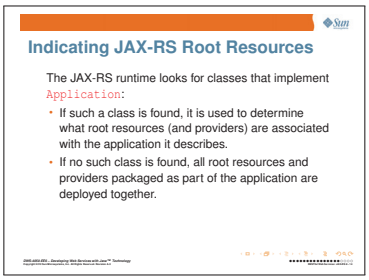
- Default lifecycle creates an instance of the root resource service provider POJO for each request.
  - Stateless logic would be the only way to guarantee proper behavior.



- Two other choices of lifecycle are available:
  - **`@Singleton`** on JAX-RS root resources indicates a single instance should be reused for all requests.
  - **`@PerSession`** on JAX-RS root resources indicates a new instance should be created and reused for each client *session*.

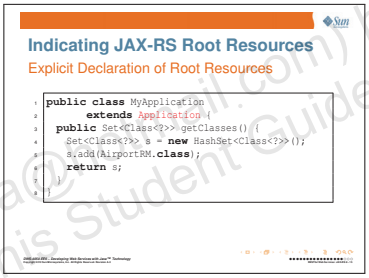
Deploying a JAX-RS Web Service Provider

At application startup, the JAX-RS runtime will look for classes that implement **`Application`**:

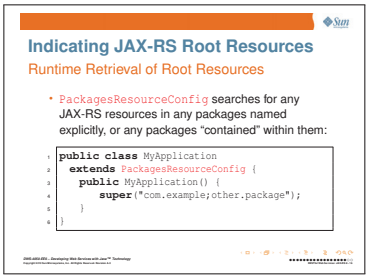


- If such a class is found, it is used to determine what root resources (and providers) are associated with the application it describes.
- If no such class is found, all root resources and providers packaged as part of the application are deployed together.

JAX-RS provides the deployment agnostic abstract class **`Application`** for declaring root resource and provider classes; it can also specify root resource and provider singleton instances. A Web service may extend this class to declare root resource and provider classes, as shown in Figure 8.8.



Alternatively, it is possible to reuse one of Jersey's implementations that scans for root resource and provider classes given a classpath or a set of package names. Such classes are automatically added to the set of classes that are returned by `getClasses()`. For example, the code in Figure 8.9 scans for root resources in the two Java packages `com.example` and `other.package`, and in any sub-packages therein.



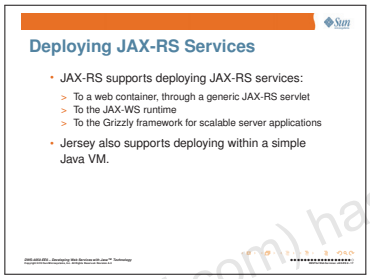

```
1  public class MyApplication
2          extends Application {
3    public Set<Class<?>> getClasses() {
4      Set<Class<?>> s = new HashSet<Class<?>>();
5      s.add(AirportRM.class);
6      return s;
7    }
8  }
```

E-90

Code 8.8: Explicit Declaration of Root Resources

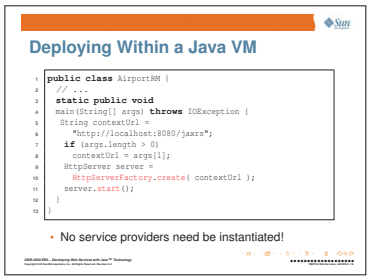
```
1 public class MyApplication
2   extends PackagesResourceConfig {
3     public MyApplication() {
4         super("com.example;other.package");
5     }
6 }
```

Code 8.9: Runtime Retrieval of Root Resources



- JAX-RS supports deploying JAX-RS services:
  - To a web container, through a generic JAX-RS servlet
  - To the JAX-WS runtime
  - To the Grizzly framework for scalable server applications
- Jersey also supports deploying within a simple Java VM.

The simplest approach to deploy a Jersey-based web service is to simple take advantage of the HTTP server built into the Java SE platform since JDK 1.6, as shown in Figure 8.10. The Jersey runtime handles the configuration of the service, and the dispatch of incoming requests to resource instances, relying on that built-in web container for HTTP processing.



```
1  public class AirportRM {
2    // ...
3    static public void
4    main(String[] args) throws IOException {
5     String contextUrl =
6        "http://localhost:8080/jaxrs";
7      if (args.length > 0)
8        contextUrl = args[1];
9      HttpServer server =
10       HttpServerFactory.create( contextUrl );
11     server.start();
12   }
13 }
```

E-87

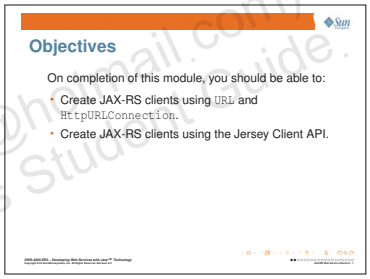
Code 8.10: Deploying Within a Java VM

Chapter 9

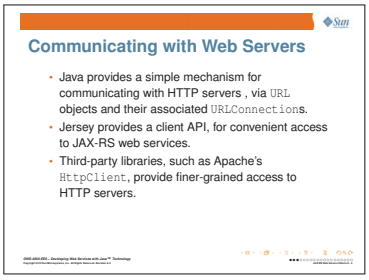
JAX-RS Web Service Clients

On completion of this module, you should be able to:

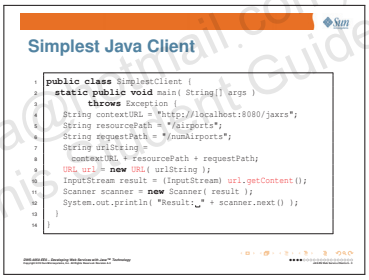
- Create JAX-RS clients using `URL` and `HttpURLConnection`.
- Create JAX-RS clients using the Jersey Client API.



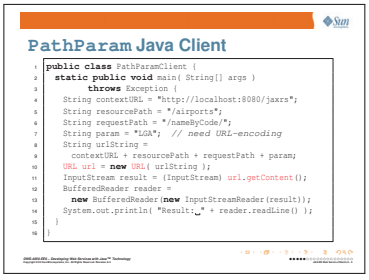
Writing JAX-RS Clients Using HttpURLConnection



- Java provides a simple mechanism for communicating with HTTP servers, via `URL` objects and their associated `URLConnection`s.
- Jersey provides a client API, for convenient access to JAX-RS web services.
- Third-party libraries, such as Apache's `HttpClient`, provide finer-grained access to HTTP servers.



REST interactions are simply HTTP request and response messages, as far as the actual communication between client and server sides is concerned. The simplest way to write a RESTful client in Java then is just to issue the proper HTTP request from Java, and read the response produced by the server. In the case of HTTP `GET` requests, the Java code to do this is straightforward: `java.net.URL` objects can be used to represent URIs, and to issue requests for those resources. The `getContent()` function in `URL` will open a connection to the appropriate server, issue the HTTP request, and return an instance of an `InputStream` to allow the application to read the response produced by the remote server. Figure 9.1 illustrates such a client.



RESTful requests with parameters may be more complex to handle – but it depends on how those parameters are to be encoded as part of the request. In the simpler approach, the parameters are embedded in the URL path itself. In this case, to the actual interaction, this is just another request for an (slightly longer) URI – so the same Java support as before is all that is required. Figure 9.2 illustrates such a client.

```
1  public class SimplestClient {
2    static public void main( String[] args )
3            throws Exception {
4      String contextURL = "http://localhost:8080/jaxrs";
5      String resourcePath = "/airports";
6      String requestPath = "/numAirports";
7      String urlString =
8        contextURL + resourcePath + requestPath;
9      URL url = new URL( urlString );
10     InputStream result = (InputStream) url.getContent();
11     Scanner scanner = new Scanner( result );
12     System.out.println( "Result: " + scanner.next() );
13   }
14 }
```

E-91

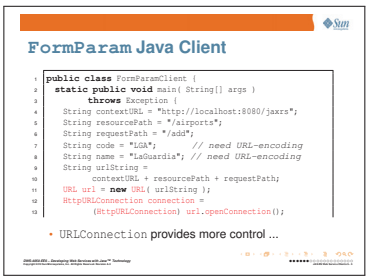
Code 9.1: Simplest JAX-RS Java Client


```
1  public class PathParamClient {
2    static public void main( String[] args )
3            throws Exception {
4      String contextURL = "http://localhost:8080/jaxrs";
5      String resourcePath = "/airports";
6      String requestPath = "/nameByCode/";
7      String param = "LGA";      // need URL-encoding
8      String urlString =
9        contextURL + resourcePath + requestPath + param;
10     URL url = new URL( urlString );
11     InputStream result = (InputStream) url.getContent();
12     BufferedReader reader =
13       new BufferedReader(new InputStreamReader(result));
14     System.out.println( "Result: " + reader.readLine() );
15   }
16 }
```

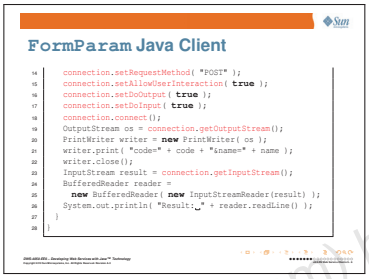
E-92

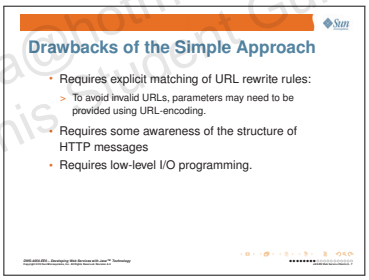
Code 9.2: `PathParam` Java Client

Things are more complicated if the web service expects to receive information via the request entity (that is, in the body of the request), rather than in to the URL path itself – the way an HTML form encodes information when using `POST`. Class `java.net.URL` offers an `HttpURLConnection`, which can be used to gain finer-grained control over the communications between client and server.



The `HttpURLConnection` instance allows the application access to both input and output streams between client and server. With the output stream, the application can encode the content to include in the (entity) body of the request explicitly, as shown in Code 9.3. Note that the representation of the information to be encoded into the body of the request is the application's responsibility.





- Requires explicit matching of URL rewrite rules:
  - To avoid invalid URLs, parameters may need to be provided using URL-encoding.
- Requires some awareness of the structure of HTTP messages
- Requires low-level I/O programming.

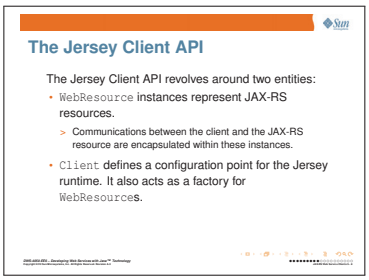
```
14 public class FormParamClient {
15   static public void main( String[] args )
16           throws Exception {
17     String contextURL = "http://localhost:8080/jaxrs";
18     String resourcePath = "/airports";
19     String requestPath = "/add";
20     String code = "LGA";           // need URL-encoding
21     String name = "LaGuardia"; // need URL-encoding
22     String urlString =
23            contextURL + resourcePath + requestPath;
24     URL url = new URL( urlString );
25     HttpURLConnection connection =
26            (HttpURLConnection) url.openConnection();
27     connection.setRequestMethod( "POST" );
28     connection.setAllowUserInteraction( true );
29     connection.setDoOutput( true );
30     connection.setDoInput( true );
31     connection.connect();
32     OutputStream os = connection.getOutputStream();
33     PrintWriter writer = new PrintWriter( os );
34     writer.print( "code=" + code + "&name=" + name );
35     writer.close();
36     InputStream result = connection.getInputStream();
37     BufferedReader reader =
38       new BufferedReader( new InputStreamReader(result) );
39     System.out.println( "Result: " + reader.readLine() );
40   }
41 }
```


E-93

Code 9.3: FormParam Java Client

Writing JAX-RS Clients Using the Jersey Client API

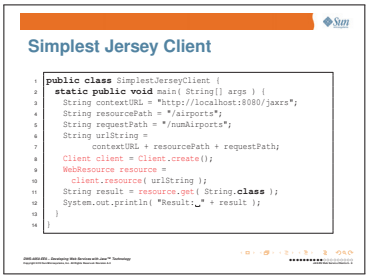
The client API is high-level API that reuses many aspects of the JAX-RS API. It is designed to be quick, easy to use and is especially useful and productive when developing tests for Web services.



A key concept of REST is the uniform interface and this is encapsulated in the `WebResource` class, which implements `UniformInterface`. A request is built up and when the corresponding HTTP method on the `UniformInterface` is invoked the request is sent to the Web resource and the returned response is processed.

The Java types that may be used for request and response entities are not restricted to String. The client API reuses the same infrastructure as JAX-RS server-side. Thus the same types that can be used on the server-side can also be used on the client-side, such as JAXB-based types. The set of Java types supported can be extended by providers that implement `MessageBodyReader` and `MessageBodyWriter`.

Code 9.4 shows how simple it can be to use the Jersey client API to issue requests to RESTful web services. `Client` is a factory for `WebResource` instances and is also used for configuring the properties of connections and requests. `WebResource` instances are the encapsulation of a Web resource, capable of building requests to send to the Web resource and processing responses returned from the Web resource.



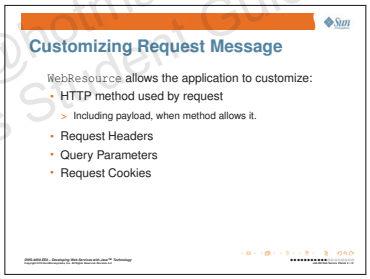
The creation of a `Client` instance is an expensive operation and the instance may make use of and retain many resources. It is therefore recommended that a `Client` instance be reused for the creation of `WebResource` instances that require the same configuration settings.

```
1  public class SimplestJerseyClient {
2    static public void main( String[] args ) {
3      String contextURL = "http://localhost:8080/jaxrs";
4      String resourcePath = "/airports";
5      String requestPath = "/numAirports";
6      String urlString =
7             contextURL + resourcePath + requestPath;
8      Client client = Client.create();
9      WebResource resource =
10       client.resource( urlString );
11     String result = resource.get( String.class );
12     System.out.println( "Result: " + result );
13   }
14 }
```

E-94

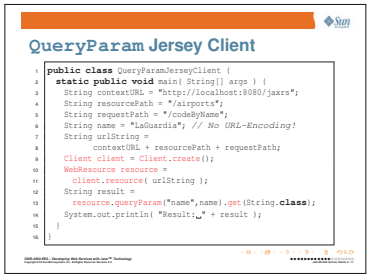
Code 9.4: Simplest Jersey Client

`WebResource` allows the application to customize:



- HTTP method used by request
  - Including payload, when method allows it.
- Request Headers
- Query Parameters
- Request Cookies

The advantages of the Jersey client API start to show as the complexity of the request increases. Code 9.5 shows a client that needs to issue a request to a web service that expects to receive parameters to the requests embedded as query parameters. The client need not know how to do this explicitly, any more: `WebResource` provides methods to add query parameters to a request itself.

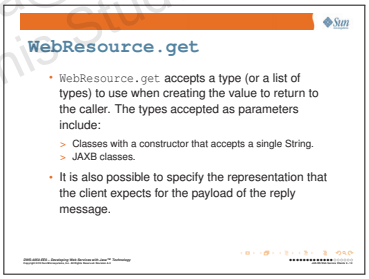


```
1  public class QueryParamJerseyClient {
2    static public void main( String[] args ) {
3      String contextURL = "http://localhost:8080/jaxrs";
4      String resourcePath = "/airports";
5      String requestPath = "/codeByName";
6      String name = "LaGuardia";  // No URL-Encoding!
7      String urlString =
8             contextURL + resourcePath + requestPath;
9      Client client = Client.create();
10     WebResource resource =
11       client.resource( urlString );
12     String result =
13       resource.queryParam("name",name).get(String.class);
14     System.out.println( "Result: " + result );
15   }
16 }
```

E-95

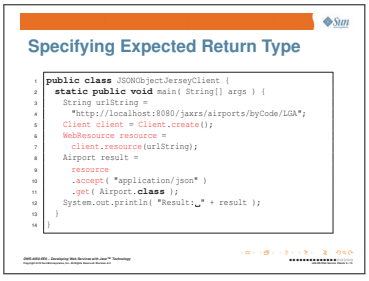
Code 9.5: `QueryParam` Jersey Client

`WebResource.get()` accepts a type (or a list of types) to use when creating the value to return to the caller. The types accepted as parameters include: classes with a constructor that accepts a single String, and JAXB classes. It is also possible to specify the representation that the client expects for the payload of the reply message.



Accepting information as the response to a RESTful request would have been complicated, when the client was limited to `HttpURLConnection`: the information would have arrived at the client as text, perhaps with an associated MIME type, and the client might have needed to process it appropriately to obtain the application-level type that it needed.

When using the Jersey client API, on the other hand, the task is much simpler: to obtain a return value from a RESTful web service call, the client simply declares the type of information it expects to receive (line 10 in Code 9.6), and the Jersey client runtime takes care of building the appropriate client-side representation of the information received from the service.

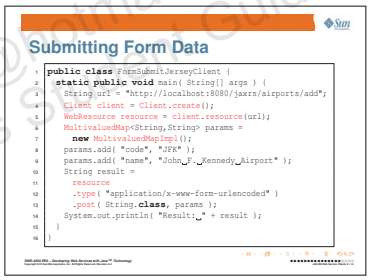


```
1  public class JSONObjectJerseyClient {
2    static public void main( String[] args ) {
3      String urlString =
4        "http://localhost:8080/jaxrs/airports/byCode/LGA";
5      Client client = Client.create();
6      WebResource resource =
7        client.resource(urlString);
8      Airport result =
9        resource
10       .accept( "application/json" )
11       .get( Airport.class );
12     System.out.println( "Result: " + result );
13   }
14 }
```

E-96

Code 9.6: Specifying Expected Return Type

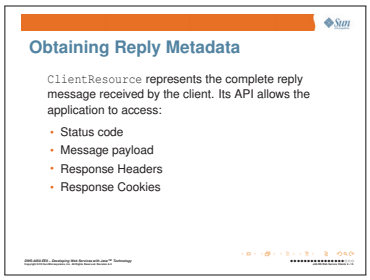
Submitting data could have similar complexity - the client might have needed to understand how to represent the information to be sent, in the entity body of the request. Using the Jersey client API, sending information to the server is straightforward, as shown in the example in Code 9.7: the client just declares the type of information to send (line 12), then passes an instance of the appropriate type as a parameter to the HTTP method invocation function (the call to `post()` in this case, in line 13).



`MultivaluedMap` and `MultivaluedMapImpl` are convenience types provided by Jersey to support maps with multiple values for the same key, such as might be needed to represent checkbox data in HTML forms.

A type of `ClientResponse` declared for the response entity may be used to obtain the status, headers and response entity received as part of the server's response to a request.

If any type, other than `ClientResponse`, is declared and the response status is greater than or equal to 300 then a `UniformInterfaceException` will be thrown, from which the `ClientResponse` instance can be accessed.



```
1  public class FormSubmitJerseyClient {
2    static public void main( String[] args ) {
3      String url = "http://localhost:8080/jaxrs/airports/add";
4      Client client = Client.create();
5      WebResource resource = client.resource(url);
6      MultivaluedMap<String,String> params =
7        new MultivaluedMapImpl();
8      params.add( "code", "JFK" );
9      params.add( "name", "John F. Kennedy Airport" );
10     String result =
11       resource
12       .type( "application/x-www-form-urlencoded" )
13       .post( String.class, params );
14     System.out.println( "Result: " + result );
15   }
16 }
```

E-97

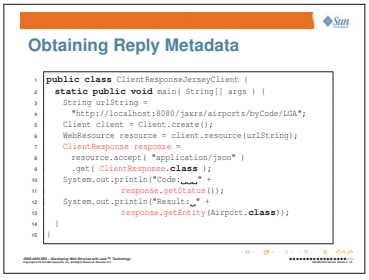
Code 9.7: Submitting Form Data

```
1  public class ClientResponseJerseyClient {
2    static public void main( String[] args ) {
3      String urlString =
4        "http://localhost:8080/jaxrs/airports/byCode/LGA";
5      Client client = Client.create();
6      WebResource resource = client.resource(urlString);
7      ClientResponse response =
8        resource.accept( "application/json" )
9        .get( ClientResponse.class );
10     System.out.println("Code:   " +
11                        response.getStatus());
12     System.out.println("Result: " +
13                        response.getEntity(Airport.class));
14   }
15 }
```

E-98

Code 9.8: Obtaining Reply Metadata – Client

Code 9.8 shows a client that is using an instance of `ClientResponse` (obtained in line 9) to obtain metadata information about the response.

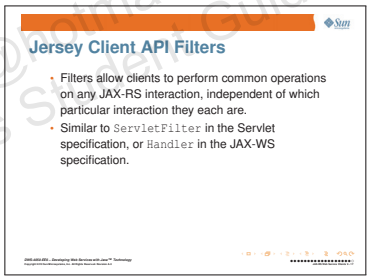


```
1  public class LoggingClient {
2    static public void main( String[] args ) {
3      String urlString =
4        "http://localhost:8080/jaxrs/airports/byCode/LGA";
5      Client client = Client.create();
6      client.addFilter( new LoggingFilter() );
7      WebResource resource = client.resource(urlString);
8      Airport result =
9        resource
10       .accept( "application/json" )
11       .get( Airport.class );
12     System.out.println( "Result: " + result );
13   }
14 }
```

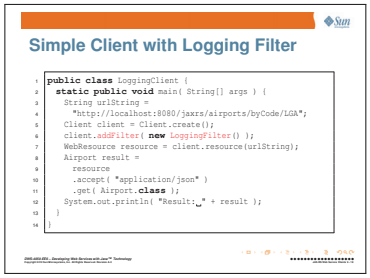
E-99

Code 9.9: Simple Client with Logging Filter

- Filters allow clients to perform common operations on any JAX-RS interaction, independent of which particular interaction they each are.
- Similar to `ServletFilter` in the Servlet specification, or `Handler` in the JAX-WS specification.



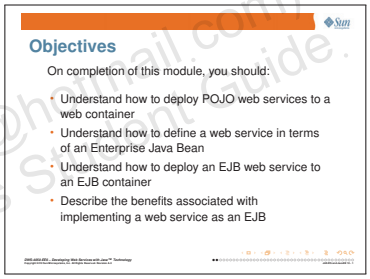
Code 9.9 shows an example of a client configuring a filter to use during request processing. The `LoggingFilter` class is a convenience filter class provided by the Jersey client API, which simply logs every request and response to a default JDK logger.



Chapter 10

JAX-RS and JavaEE

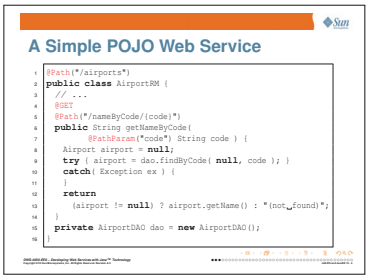
On completion of this module, you should:



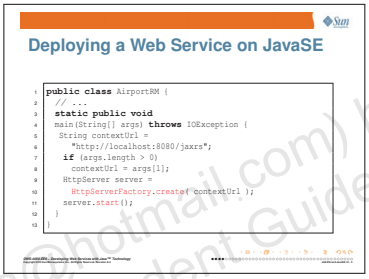
- Understand how to deploy POJO web services to a web container
- Understand how to define a web service in terms of an Enterprise Java Bean
- Understand how to deploy an EJB web service to an EJB container
- Describe the benefits associated with implementing a web service as an EJB

Deploying a Web Service to a Web Container

Code 10.1 shows a simple web service, implemented using JAX-RS (and Jersey). Note how JAX-RS resources in general have no coupling with the specifics of the web container they might be deployed to – JAX-RS resources are just plain Java objects that can act as RESTful web service providers.



Code 10.2 shows the simplest way to deploy a JAX-RS web service provider, using the Jersey runtime. In this example, all the logic required to tie the JAX-RS resource to the underlying web container – in this case, the web server built into Java SE 6 – is handled transparently by the Jersey runtime.




```
1  @Path("/airports")
2  public class AirportRM {
3    // ...
4    @GET
5    @Path("/nameByCode/{code}")
6    public String getNameByCode(
7            @PathParam("code") String code ) {
8      Airport airport = null;
9      try { airport = dao.findByCode( null, code ); }
10     catch( Exception ex ) {
11     }
12     return
13       (airport != null) ? airport.getName() : "(not found)";
14   }
15   private AirportDAO dao = new AirportDAO();
16 }
```

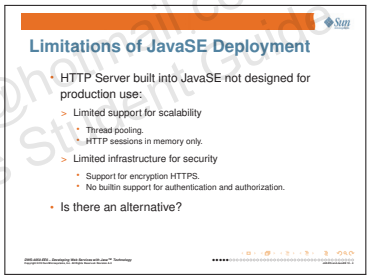
E-87

Code 10.1: A Simple POJO Web Service Using JAXRS

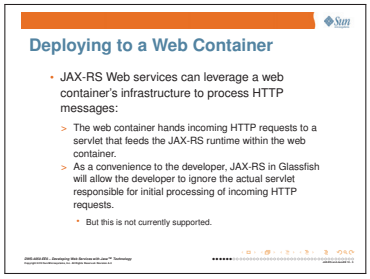
```
1  public class AirportRM {
2    // ...
3    static public void
4    main(String[] args) throws IOException {
5     String contextUrl =
6        "http://localhost:8080/jaxrs";
7      if (args.length > 0)
8        contextUrl = args[1];
9      HttpServer server =
10       HttpServerFactory.create( contextUrl );
11     server.start();
12   }
13 }
```

E-87

Code 10.2: Deploying a Web Service on JavaSE Using JAXRS

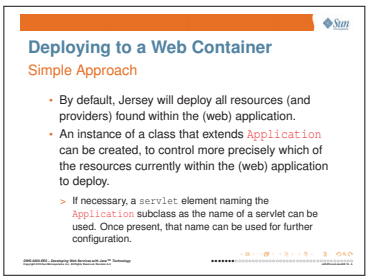


- HTTP Server built into JavaSE not designed for production use:
  - Limited support for scalability
    - Thread pooling.
    - HTTP sessions in memory only.
  - Limited infrastructure for security
    - Support for encryption HTTPS.
    - No built-in support for authentication and authorization.
- Is there an alternative?



- JAX-RS Web services can leverage a web container's infrastructure to process HTTP messages:
  - The web container hands incoming HTTP requests to a servlet that feeds the JAX-RS runtime within the web container.
  - As a convenience to the developer, JAX-RS in GlassFish will allow the developer to ignore the actual servlet responsible for initial processing of incoming HTTP requests.
    - But this is not currently supported.

By default, deploying JAX-RS based web services is straightforward: all the developer needs to do is package the JAX-RS resources and providers, along with all their supporting structure, as part of the application. At application startup, the JAX-RS runtime will look for classes that implement **`Application`**:



- If such a class is found, it is used to determine what resources and providers are associated with the application it describes.
- If no such class is found, all resources and providers packaged as part of the application are deployed together.

When the JAX-RS resources have authorization constraints associated with them, JAX-RS developers run into the same limitation that JAX-WS had to address: the JAX-RS runtime relies on the web container to obtain authentication information – but this means that the web container must be configured to require authentication data when receiving requests that will be forwarded to the JAX-RS runtime to process.

In the case of JAX-RS, the set of resources to be deployed as RESTful web services can be described by an instance of a child of the **`Application`** class. JAX-RS leverages this description, in order to capture authentication constraints for the underlying web container: the presence of such a child as part of a web application will act as though there was a servlet defined as part of the web application whose name is the fully qualified name for that child of **`Application`**. Figure 10.1 illustrates what this might have looked like. However, lines 2 to 6 need not appear in the `web.xml` configuration file – the runtime will act as if such as declaration had been there, anyway.

Once such a servlet is known to be part of the application, it can be used to declare, in `web.xml`, security authentication constraints on attempts to access that servlet – which translate into authentication constraints on attempts to invoke the corresponding RESTful web services.

```
1 <web-app ...>
2   <servlet>
3     <servlet-name>
4       com.example.jaxrs.resources.MyApplication
5     </servlet-name>
6   </servlet>
```

Figure 10.1: Simple Approach to Deploying a JAX-RS Web Service

There are two alternative approaches to configuring JAX-RS-based web services as (part of) a web application, both involving more explicit configuration of the Jersey runtime within the web application. Behind the scenes, Jersey implements the bridge from HTTP processing to calls to the RESTful web services via the servlet `com.sun.jersey.spi.container.servlet.ServletContainer`.

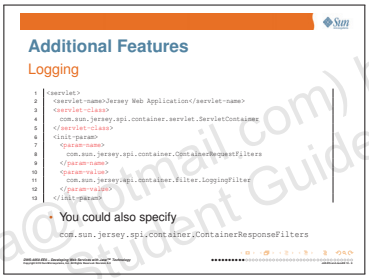
In the first of these two approaches, the developer explicitly configures this Jersey servlet in `web.xml`. Since this is just another servlet, as far as the web container is concerned, the developer can also configure the URL path (or, for JAX-RS, the URL pattern) that the container will forward to this servlet, along with any authentication configuration that might be appropriate. When using this approach, the developer must specify the child of **`Application`** that defines the resources and providers to be offered as RESTful web services. The class name is captured as the value of the servlet initialization parameter with name `"javax.ws.rs.`**`Application`**`"`. This approach is illustrated in Code 10.3.

```
1  <servlet>
2    <servlet-name>ServletAdaptor</servlet-name>
3    <servlet-class>
4     com.sun.jersey.spi.container.servlet.ServletContainer
5    </servlet-class>
6    <init-param>
7      <param-name>javax.ws.rs.Application</param-name>
8       <param-value>com.example.jaxrs.MyApp</param-value>
9    </init-param>
10 </servlet>
11 <servlet-mapping>
12   <servlet-name>ServletAdaptor</servlet-name>
13   <url-pattern>/resources/*</url-pattern>
14 </servlet-mapping>
```


Code 10.3: JAX-RS Deployment: Alternative Approach #1

The URL pattern specified in line 13 of Code 10.3 is used to define a path prefix that distinguishes incoming URL requests that map to calls to web services from other URL requests within the web application. This notion can also be captured using a `@ApplicationPath` annotation to the child class of **`Application`**, when relying on the default configuration of the JAX-RS resources described above.

A second alternative provides a more convenient approach: instead of requiring that that developer provide a child class of **`Application`** that explicitly enumerates all the JAX-RS resources and providers that are to be considered part of the application, the configuration of the Jersey dispatch servlet also supports the ability to search a list of package names; all resources and providers found therein would be considered part of the application. The property name `"com.sun.jersey.config.property.packages"` is used to specify this list of package names; the package names in the list must be separated by whitespace, or `;` (semi-colon). This second approach is illustrated in Code 10.4.



JAX-RS relies on the web container handle HTTP requests, in order to dispatch them to JAX-RS resources to process. Since the container is part of picture, the developer can use other features that the web container offers - such as *servlet filters* to monitor what goes on when requests are received. For convenience, Jersey provides an equivalent mechanism within the framework:

- `ContainerRequestFilter` types get a chance to examine incoming requests, before the Jersey runtime dispatches them to the appropriate resource.
- `ContainerResponseFilter` types get a chance to examine the outgoing response, before Jersey hands it off to the web container, to deliver back to the client.

```
1  <servlet>
2    <servlet-name>Jersey Web Application</servlet-name>
3    <servlet-class>
4      com.sun.jersey.spi.container.servlet.ServletContainer
5    </servlet-class>
6    <init-param>
7      <param-name>
8        com.sun.jersey.config.property.packages
9      </param-name>
10     <param-value>com.example;other.package</param-value>
11   </init-param>
12 </servlet>
```

Code 10.4: JAX-RS Deployment: Alternative Approach #2

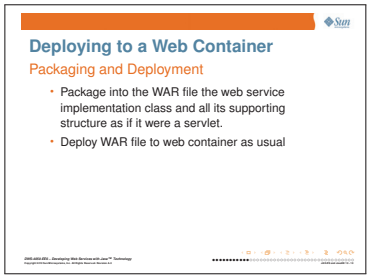
```
<servlet>
  <servlet-name>Jersey Web Application</servlet-name>
  <servlet-class>
    com.sun.jersey.spi.container.servlet.ServletContainer
  </servlet-class>
  <init-param>
    <param-name>
      com.sun.jersey.spi.container.ContainerRequestFilters
    </param-name>
    <param-value>
      com.sun.jersey.api.container.filter.LoggingFilter
    </param-value>
  </init-param>
```

Code 10.5: Logging in JAX-RS

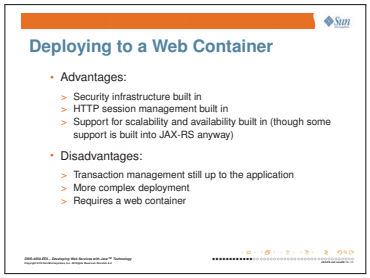
Code 10.5 shows an example of a web application configuration file that installs a logging filter, provided by Jersey by default, to monitor incoming requests.

You could also specify filters on the outgoing response back to the client, via the parameter name:
`com.sun.jersey.spi.container.ContainerResponseFilters`.

- Package into the WAR file the web service implementation class and all its supporting structure as if it were a servlet.
- Deploy WAR file to web container as usual



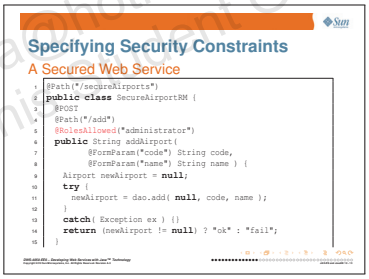
- Advantages:
  - Security infrastructure built in
  - HTTP session management built in
  - Support for scalability and availability built in (though some support is built into JAX-RS anyway)



- Disadvantages:
  - Transaction management still up to the application
  - More complex deployment
  - Requires a web container

- Best practices for security address these security principles:
  - Maintaining Corporate Security Policies
  - Self-Preservation
  - Defense in Depth
  - Least Privilege
  - Compartmentalization
  - Proportionality
- Least Privilege requires authentication and authorization constraints on services.

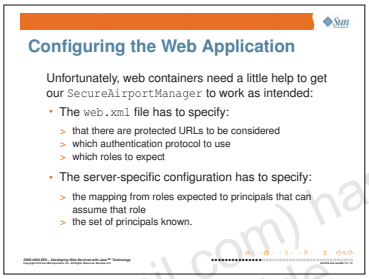
JAX-RS also supports the standard JavaEE security annotations to indicate what roles, if any, are allowed to invoke operations on JAX-RS resources. Code 10.6 shows a JAX-RS root resource which includes a `@RolesAllowed` annotation, to indicate that only users in the `administrator` role can invoke the `addAirport` operation on this resource.




```
1  @Path("/secureAirports")
2  public class SecureAirportRM {
3    @POST
4    @Path("/add")
5    @RolesAllowed("administrator")
6    public String addAirport(
7            @FormParam("code") String code,
8            @FormParam("name") String name ) {
9      Airport newAirport = null;
10     try {
11       newAirport = dao.add( null, code, name );
12     }
13     catch( Exception ex ) {}
14     return (newAirport != null) ? "ok" : "fail";
15   }
```

Code 10.6: A Secured JAX-RS Web Service

Note that this example is not a best practice, when building RESTful web services: clients of this service would invoke this `addAirport` operation by issuing a `POST` to `/secureAirports/add` – but the best practice in REST is not to encode the action itself in the URL: URLs should name resources, not activities. A better example, in this sense, might just `POST` to `/secureAirports` – line 4 in this code sample would just be taken out.

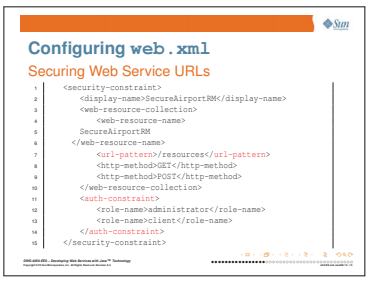


All of the three approaches described above capture basic information about the RESTful web services that are to be offered as part of a web application: in particular, they capture the URLs that will map to calls to these services. Regardless of which approach is used, however, additional configuration is needed, to enable the web container to require authentication data in order to allow incoming requests to these RESTful web services to proceed. This additional configuration is the traditional `web.xml` configuration of authentication requirements for URL resources, just as it would be required for securing access to web application servlets, or JAX-WS web services, deployed within the web container. The `web.xml` file has to specify:

- that there are protected URLs to be considered
- which authentication protocol to use
- which roles to expect

The server-specific configuration file has to specify, in addition:

- the mapping from roles expected to principals that can assume that role
- the set of principals known.



Code 10.7 illustrates what the XML elements in `web.xml` might look like, to capture the requirement that only users authenticated to belong to specific groups may access the URLs associated with the JAX-RS resources. Note how line 5 refers to the prefix associated with all JAX-RS resources in this web application, and then lines 6 to 8 specify the set of HTTP methods to be restricted (which include all the HTTP methods that the JAX-RS resources might respond to).

```
1  <security-constraint>
2    <display-name>SecureAirportRM</display-name>
3    <web-resource-collection>
4      <web-resource-name>
5        SecureAirportRM
6      </web-resource-name>
7      <url-pattern>/resources</url-pattern>
8      <http-method>GET</http-method>
9      <http-method>POST</http-method>
10   </web-resource-collection>
11   <auth-constraint>
12     <role-name>administrator</role-name>
13     <role-name>client</role-name>
14   </auth-constraint>
```

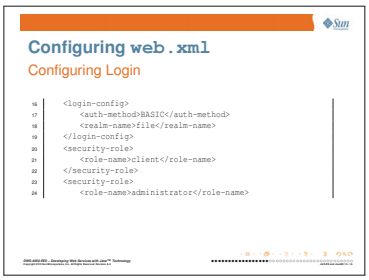
E-100

Code 10.7: Securing Web Service URLs

```
16 <login-config>
17   <auth-method>BASIC</auth-method>
18   <realm-name>file</realm-name>
19 </login-config>
20 <security-role>
21   <role-name>client</role-name>
22 </security-role>
23 <security-role>
24   <role-name>administrator</role-name>
```

Code 10.8: Configuring Login in `web.xml`

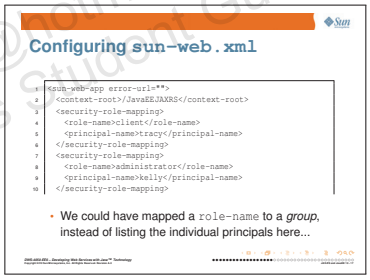
These requirements are applied at the level of the web container – but the JAX-RS runtime will perform a second check, at the level of the individual method being called.



Security roles are declared in the `web.xml` deployment descriptor, using the `<security-role>` element. This element lives at the first level of the `web.xml` file, as a direct child of the `<web-app>` element. In many cases, it is created and edited using tools in your chosen environment, but the specification of the naked XML is simple too, as shown in Code 10.8. The example shows a fragment of `web.xml` illustrating the declaration of the security roles "client" and "administrator" (in lines 20 to 24).

The fragment in Code 10.8 also shows how the `web.xml` must specify the approach to be used by the underlying server to authenticate users. Java EE web containers can use password challenges or client certificates in any of four standardized forms to implement this task. The three approaches that can be used by a web service are:

- BASIC – HTTP Basic authentication. This mechanism is standardized, based on a username and password challenge, and requires the use of encryption to maintain wire security.
- DIGEST – This mechanism modifies the BASIC approach by providing encryption of the password in transit. However, the other disadvantages remain and it too is unpopular.
- CLIENT CERTIFICATE – In this mechanism, the client is required to provide a public key certificate, signed by an authority trusted by the web container.



Mapping users to roles is not part of the JavaEE specification, and so it is dependent on the web-container in use. In the case of GlassFish, this occurs in the vendor-specific deployment descriptor file named `sun-web.xml`. This file contains one `<security-role-mapping>` element for each role that will exist; this element specifies the name of the role and the names of all the users that are members of that role. Code 10.9 shows an example of this configuration file.

We could have mapped a `role-name` to a *group*, instead of listing the individual principals here...

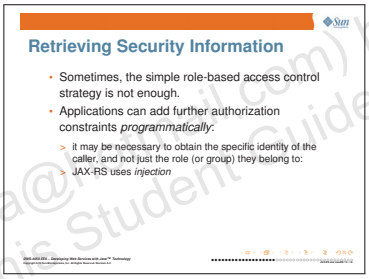
```
1  <sun-web-app error-url="">
2    <context-root>/JavaEEJAXRS</context-root>
3    <security-role-mapping>
4      <role-name>client</role-name>
5      <principal-name>tracy</principal-name>
6    </security-role-mapping>
7    <security-role-mapping>
8      <role-name>administrator</role-name>
9      <principal-name>kelly</principal-name>
10   </security-role-mapping>
```


Code 10.9: Configuring `sun-web.xml`

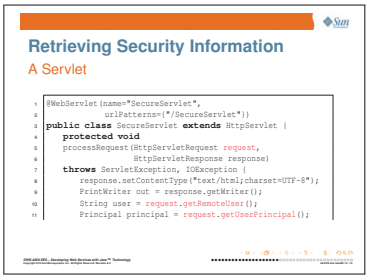
```
1  @WebServlet(name="SecureServlet",
2                 urlPatterns={"/SecureServlet"})
3  public class SecureServlet extends HttpServlet {
4      protected void
5      processRequest(HttpServletRequest request,
6                     HttpServletResponse response)
7      throws ServletException, IOException {
8          response.setContentType("text/html;charset=UTF-8");
9          PrintWriter out = response.getWriter();
10         String user = request.getRemoteUser();
11         Principal principal = request.getUserPrincipal();
```

E-106

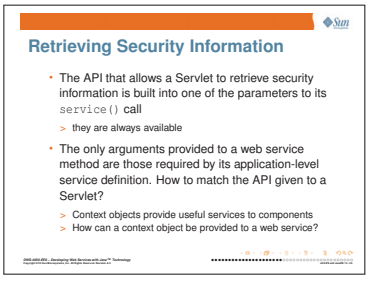
Code 10.10: Retrieving Security Information in a Servlet



Sometimes, the simple role-based access control strategy is not enough. In these cases, applications can specify additional authorization constraints *programmatically*. To do so, it may be necessary to obtain the specific identity of the caller, and not just the role (or group) they belong to.



Programmatic authorization is more expressive than the declarative approach, but is more cumbersome to maintain, and because of the additional complexity, more error prone. In particular, declarative authorization controls access at a role level while programmatic authorization can selectively permit or block access to principals belonging to a role. Code 10.10 shows a code fragment from a web service that uses programmatic authentication.



The API that allows a Servlet to retrieve security information is built into one of the parameters to its `service()` call – so it is always available. On the other hand, the only arguments provided to a web service method are those required by its application-level service definition:

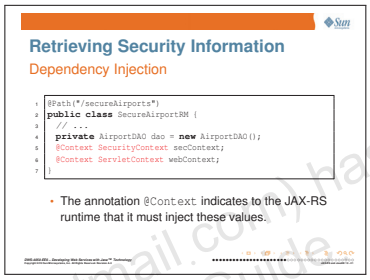
- Context objects provide useful services to components
- How can a context object be provided to a web service?

```
1 @Path("/secureAirports")
2 public class SecureAirportRM {
3   // ...
4   private AirportDAO dao = new AirportDAO();
5   @Context SecurityContext secContext;
6   @Context ServletContext webContext;
7 }
```

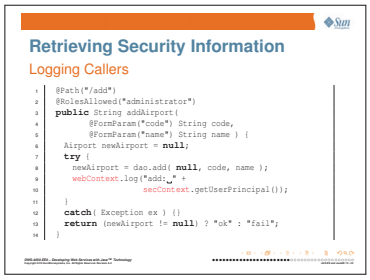
E-102

Code 10.11: Dependency Injection in JAXRS

Code 10.11 illustrates how JAX-RS resources would request that the runtime provide them with the instances that would allow the resources to obtain authentication information. Since JAX-RS resources rely on the web container infrastructure for processing HTTP requests, they can request access to the `ServletContext` that the underlying web container provides, as shown in line 6. Given this instance, JAX-RS resources can use the same (security) API that servlets use. As a convenient to developers, however, JAX-RS offers a `SecurityContext`, which offers more convenient access to the security information available for the current caller. Line 5 shows how a JAX-RS resource would ask for this instance.



Code 10.12 shows the same example shown before, but in the context of JAX-RS: a web service that logs to the we container log the identity of the caller, each time the service is called. The web service uses the `ServletContext` API to add entries to the web container log, but it uses the `SecurityContext` type provided by JAX-RS to retrieve the caller's identity to log.

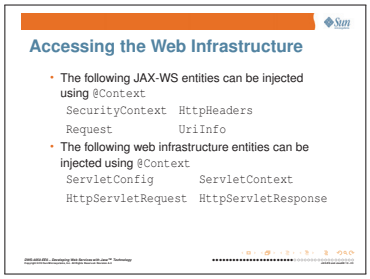


- The following JAX-WS entities can be injected using `@Context`

  `SecurityContext` `HttpHeaders`
  `Request` `UriInfo`

- The following web container entities can be injected using `@Context`

  `ServletConfig` `ServletContext`
  `HttpServletRequest` `HttpServletResponse`



```
1   @Path("/add")
2   @RolesAllowed("administrator")
3   public String addAirport(
4           @FormParam("code") String code,
5           @FormParam("name") String name ) {
6     Airport newAirport = null;
7     try {
8       newAirport = dao.add( null, code, name );
9       webContext.log("add: " +
10                      secContext.getUserPrincipal());
11    }
12    catch( Exception ex ) {}
13    return (newAirport != null) ? "ok" : "fail";
14  }
```

E-102

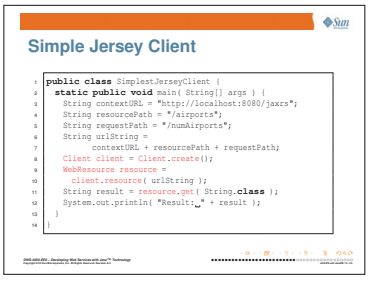
Code 10.12: Logging Callers in JAX-RS

```
1  public class SimplestJerseyClient {
2    static public void main( String[] args ) {
3      String contextURL = "http://localhost:8080/jaxrs";
4      String resourcePath = "/airports";
5      String requestPath = "/numAirports";
6      String urlString =
7              contextURL + resourcePath + requestPath;
8      Client client = Client.create();
9      WebResource resource =
10       client.resource( urlString );
11     String result = resource.get( String.class );
12     System.out.println( "Result: " + result );
13   }
14 }
```

E-94

Code 10.13: Simple Jersey Client

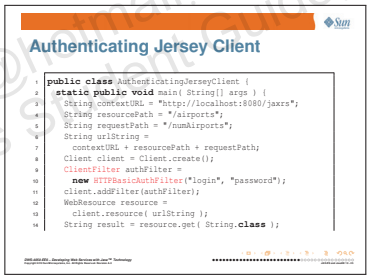
Code 10.13 shows a simple web service client application, written using the Jersey Client API. To contact a restricted web service, however, the client application would have to incorporate authentication information into the web service requests.



```
1  public class AuthenticatingJerseyClient {
2    static public void main( String[] args ) {
3      String contextURL = "http://localhost:8080/jaxrs";
4      String resourcePath = "/airports";
5      String requestPath = "/numAirports";
6      String urlString =
7        contextURL + resourcePath + requestPath;
8      Client client = Client.create();
9      ClientFilter authFilter =
10       new HTTPBasicAuthFilter("login", "password");
11     client.addFilter(authFilter);
12     WebResource resource =
13       client.resource( urlString );
14     String result = resource.get( String.class );
```

E-117

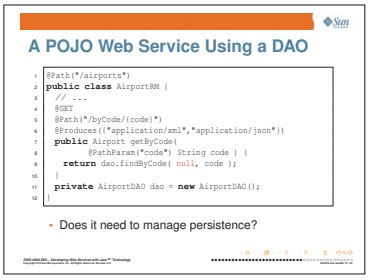
Code 10.14: Authenticating Jersey Client



Code 10.14 illustrates the approach taken by the Jersey Client API to incorporate authentication information into requests: the client application associates an authentication filter (the instance of `HTTPBasicAuthFilter` with the `Client`, and this filter incorporates the appropriate HTTP header into every request. The encoding required by the HTTP standard to protect the authentication data as it travels over the network is handled by this filter.

Creating a Web Service From an EJB

Our original implementation of the `AirportManagerRM` web service, shown in Code 10.15, was implemented as a POJO web service provider – a simple Java object that did what was needed.

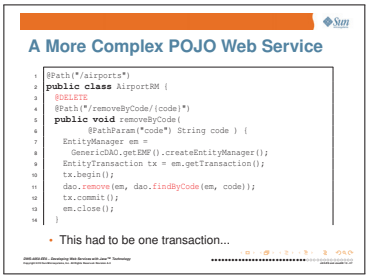



```
1  @Path("/airports")
2  public class AirportRM {
3    // ...
4    @GET
5    @Path("/byCode/{code}")
6    @Produces({"application/xml","application/json"})
7    public Airport getByCode(
8            @PathParam("code") String code ) {
9      return dao.findByCode( null, code );
10   }
11   private AirportDAO dao = new AirportDAO();
12 }
```

E-87

Code 10.15: A POJO Web Service Using a DAO

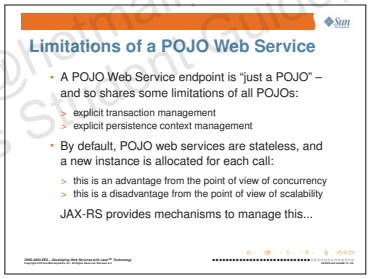
Ideally, concerns related to transaction management and persistence contexts should be hidden within the DAO – so that our web service provider would not have to be concerned with this. These are the same concerns that were raised before, in Section 4, when discussing JAX-WS web services. The solution, in the case of JAX-RS, will be the same: turn the JAX-RS resources into EJBs, to leverage the facilities built into the JavaEE EJB container.



```
1  @Path("/airports")
2  public class AirportRM {
3    @DELETE
4    @Path("/removeByCode/{code}")
5    public void removeByCode(
6            @PathParam("code") String code ) {
7      EntityManager em =
8        GenericDAO.getEMF().createEntityManager();
9      EntityTransaction tx = em.getTransaction();
10     tx.begin();
11     dao.remove(em, dao.findByCode(em, code));
12     tx.commit();
13     em.close();
14   }
```

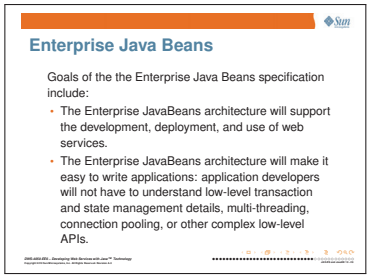
E-108

Code 10.16: A More Complex POJO Web Service



- A POJO Web Service endpoint is "just a POJO" – and so shares some limitations of all POJOs:
  - explicit transaction management
  - explicit persistence context management
- By default, POJO web services are stateless, and a new instance is allocated for each call:
  - this is an advantage from the point of view of concurrency
  - this is a disadvantage from the point of view of scalability

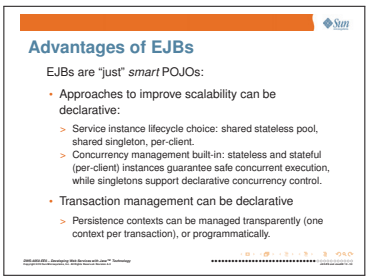
  JAX-RS provides mechanisms to manage this...



Goals of the the Enterprise Java Beans specification include:

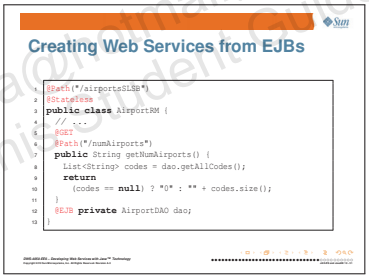
- The Enterprise JavaBeans architecture will support the development, deployment, and use of web services.
- The Enterprise JavaBeans architecture will make it easy to write applications: application developers will not have to understand low-level transaction and state management details, multi-threading, connection pooling, or other complex low-level APIs.

EJBs are "just" *smart* POJOs:



- Approaches to improve scalability can be declarative:
  - Service instance lifecycle choice: shared stateless pool, shared singleton, per-client.
  - Concurrency management built-in: stateless and stateful (per-client) instances guarantee safe concurrent execution, while singletons support declarative concurrency control.
- Transaction management can be declarative
  - Persistence contexts can be managed transparently (one context per transaction), or programmatically.

The web services specifications are orthogonal to the EJB specification. To define an EJB to also be a web service (or vice versa), we just have to say so – and support for deploying EJBs as Web Services is built into Java EE. Code 10.17 shows an example of a JAX-WS web service, now implemented as an EJB.



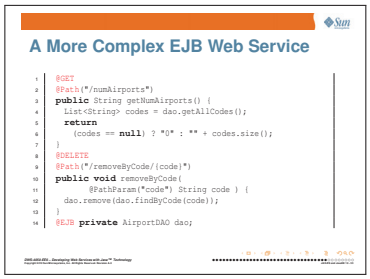

```
1  @Path("/airportsSLSB")
2  @Stateless
3  public class AirportRM {
4    // ...
5    @GET
6    @Path("/numAirports")
7    public String getNumAirports() {
8      List<String> codes = dao.getAllCodes();
9      return
10       (codes == null) ? "0" : "" + codes.size();
11   }
12   @EJB private AirportDAO dao;
13 }
```

E-111

Code 10.17: Creating Web Services from EJBs

Transaction Management

Code 10.18 shows an alternate implementation of the `AirportManagerRM` resource – but, this time, as a stateless session bean. Note how responsibility for entity managers and their associated persistence context, are now hidden within the Data Access Object EJB. Transaction management is hidden, too, as it is something that this implementation can delegate to the EJB container surrounding it.




```
1   @GET
2   @Path("/numAirports")
3   public String getNumAirports() {
4     List<String> codes = dao.getAllCodes();
5     return
6       (codes == null) ? "0" : "" + codes.size();
7   }
8   @DELETE
9   @Path("/removeByCode/{code}")
10  public void removeByCode(
11          @PathParam("code") String code ) {
12    dao.remove(dao.findByCode(code));
13  }
14  @EJB private AirportDAO dao;
```

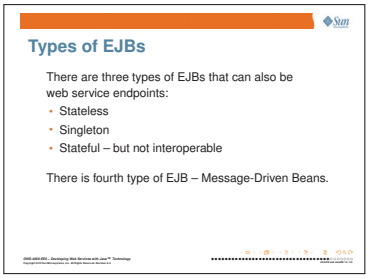
E-111

Code 10.18: A More Complex EJB Web Service

Scalability

There are three types of EJBs that can also be web service endpoints:

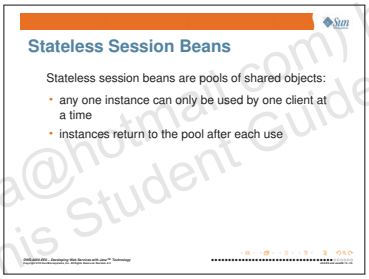
- Stateless
- Singleton
- Stateful – but not interoperable

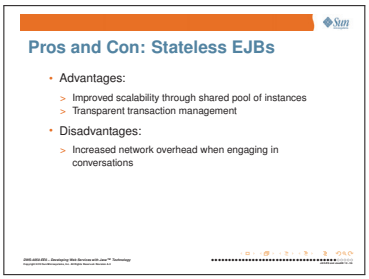


There is fourth type of EJB – Message-Driven Beans.

Stateless session beans are pools of shared objects:

- any one instance can only be used by one client at a time
- instances return to the pool after each use

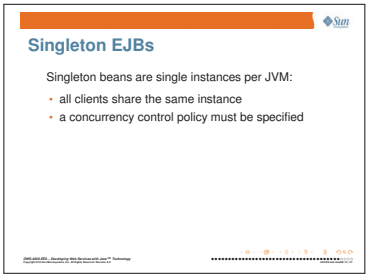




- Advantages:
  - Improved scalability through shared pool of instances
  - Transparent transaction management
- Disadvantages:
  - Increased network overhead when engaging in conversations

Singleton beans are single instances per JVM:

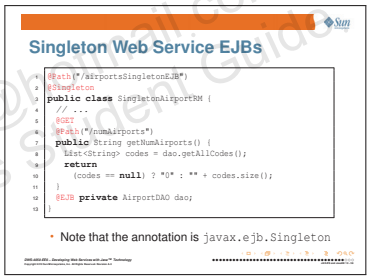
- all clients share the same instance
- a concurrency control policy must be specified



```
1  @Path("/airportsSingletonEJB")
2  @Singleton
3  public class SingletonAirportRM {
4    // ...
5    @GET
6    @Path("/numAirports")
7    public String getNumAirports() {
8      List<String> codes = dao.getAllCodes();
9      return
10       (codes == null) ? "0" : "" + codes.size();
11   }
12   @EJB private AirportDAO dao;
13 }
```

E-114

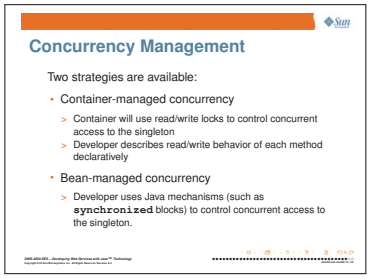
Code 10.19: Singleton Web Service EJBs



Singleton session beans represent an implementation of the singleton pattern. Singleton session beans can be used to provide shared access and concurrent access across clients.

A singleton session bean is instantiated once per application, and exists for the lifecycle of the application. In cases where the container is distributed over many virtual machines, each application will have one bean instance of the singleton for each JVM. This is equivalent to the effect of the **@Singleton** annotation offered by JAX-RS – but the EJB Singleton offers more sophisticated (declarative) concurrency control.

Code 10.19 shows the `AirportManagerRM` resource, as a singleton session bean.



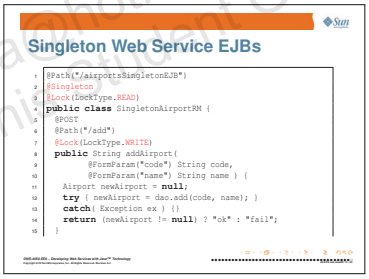
Two strategies are available:

- Container-managed concurrency
  - Container will use read/write locks to control concurrent access to the singleton
  - Developer describes read/write behavior of each method declaratively
- Bean-managed concurrency
  - Developer uses Java mechanisms (such as `synchronized` blocks) to control concurrent access to the singleton.

```
1  @Path("/airportsSingletonEJB")
2  @Singleton
3  @Lock(LockType.READ)
4  public class SingletonAirportRM {
5    @POST
6    @Path("/add")
7    @Lock(LockType.WRITE)
8    public String addAirport(
9            @FormParam("code") String code,
10           @FormParam("name") String name ) {
11     Airport newAirport = null;
12     try { newAirport = dao.add(code, name); }
13     catch( Exception ex ) {}
14     return (newAirport != null) ? "ok" : "fail";
15   }
```

E-114

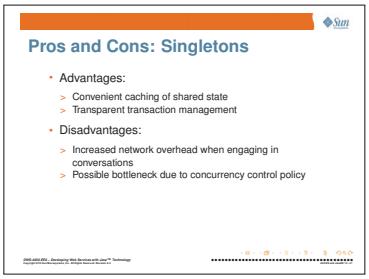
Code 10.20: Concurrency Policy for Singleton Web Service EJBs using JAX-RS



In singleton beans that utilize container-managed concurrency, the container controls concurrent access to the bean instance based on method-level locking metadata. Specifically, the `javax.ejb.Lock` and `javax.ejb.LockType` annotations are used to specify the concurrent access management strategy of the singleton's business methods.

The `@Lock` annotation informs the container that a method requires concurrency management. There are two locking strategies defined by constants included in `LockType`: `javax.ejb.LockType.READ` and `javax.ejb.LockType.WRITE`.

Code 10.20 shows our `SingletonAirportManagerRM` resource with a possible concurrency control specification built in.



- Advantages:
  - Convenient caching of shared state
  - Transparent transaction management
- Disadvantages:
  - Increased network overhead when engaging in conversations
  - Possible bottleneck due to concurrency control policy