Build COMPETENCY
across your TEAM

SYNERGETICS GET IT RIGHT

Microsoft Partner
Gold Cloud Platform
Silver Learning

Webservices (SOAP + REST)

By Smita B Kumar

Topics Overview

Day	Topics
Day 17	Introduction to Web Services and alternatives
	Styles of Web Services
	SOAP
	WSDL
Day 18	Developing SOAP Web Service
	Introduction to REST
	Getting Started with JAX-RS Service
	SOA Concepts

Simple Scenario

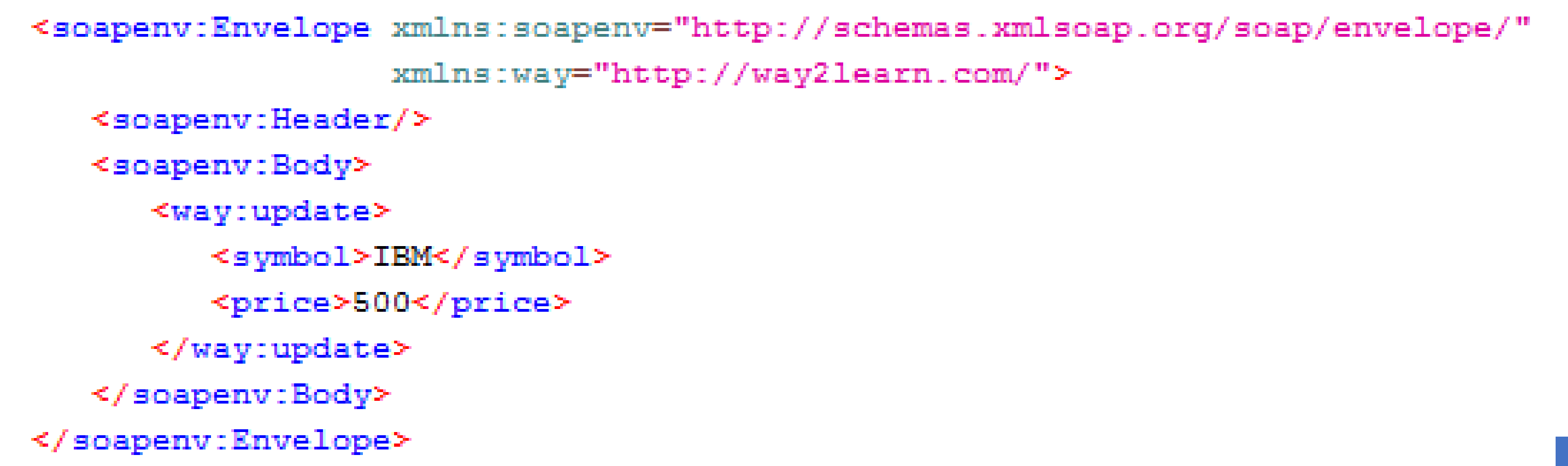
- Want to expose the implementation of below interface as a soap webservice

```
public interface StockQuoteService {

    public  double getPrice(String symbol) ;

    public  void update(String symbol, double price);

}
```

What is a Webservice ?

- What do I mean by a webservice ?
  - Web services are made available from a business's Web server for Web users or other Web-connected programs.
  - Unlike traditional client/server models, Web services do not provide the user with a GUI .
  - Web services instead share business logic, data and processes through a programmatic interface across a network.

SOAP

- XML data which should be exchanged between client and server should be in a particular format.
- SOAP is a standard message protocol which defines the format of exchanged data.
- Sample SC 

```xml
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:way="http://way2learn.com/">
   <soapenv:Header/>
   <soapenv:Body>
      <way:update>
         <symbol>IBM</symbol>
         <price>500</price>
      </way:update>
   </soapenv:Body>
</soapenv:Envelope>
```

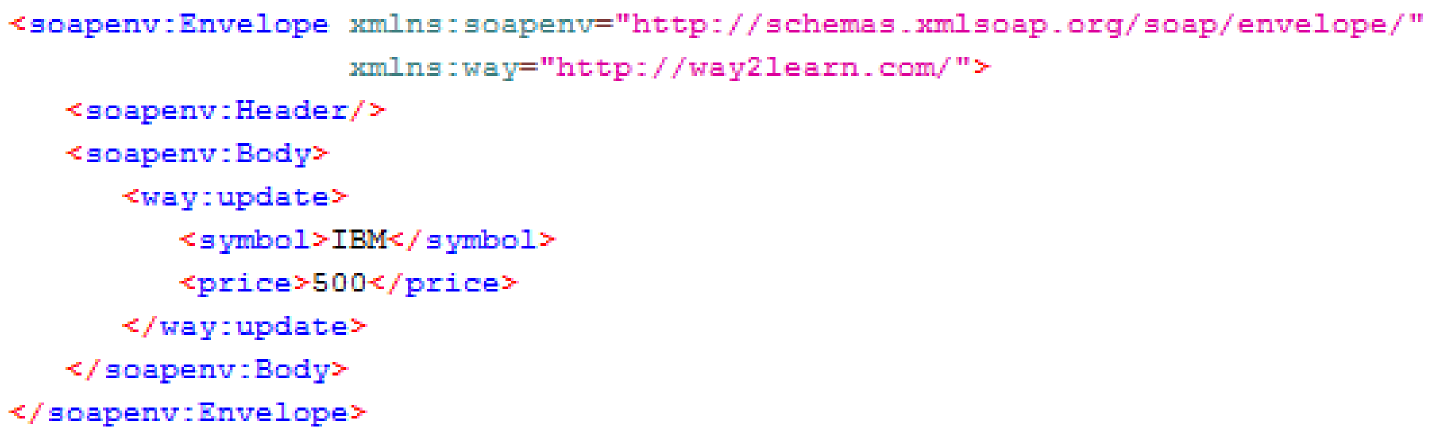
:

Styles of Webservices

- Following are styles of Webservices :
  - Rpc Style
  - Document Style

RPC Style Webservices

- In RPC Style webservices, one input message can contain more than one part.



```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:way="http://way2learn.com/">
   <soapenv:Header/>
   <soapenv:Body>
      <way:update>
         <symbol>IBM</symbol>
         <price>500</price>
      </way:update>
   </soapenv:Body>
</soapenv:Envelope>
```

- If above is the input message for accessing, RPC style webservice, <way:update> identifies operation name and parts are symbol and name

Document Style Webservices

- In Document Style Webservices, input message contain only a single part.

- If the method being invoked takes more than one argument, the single part in input message is a complex type defined in the schema

RPC vs Document style

- In Document style Webservices, the input message tag are defined in XSD. So, the incoming messages **can be validated** against XSD before processing the message

- In RPC style Webservices, the incoming messages does not contain and xsd. So, they **cannot be validated**

- **So, Document style is preferred over RPC Style**

How to describe about our service to client?

- We describe a webservice to client in a WSDL file.
- WSDL stands for webservice definition language
- WSDL is an XML-based language for describing Web services and how to access them.

WSDL

- A WSDL document describes a web service using these major elements:

- **<types>** →The data types used by the web service
- **<message>** →The messages used by the web service
- **<portType>** →The operations performed by the web service **<binding>** →The communication protocols used by the web service

WSDL document

- Main structure of a WSDL document looks like this:

```
<definitions>

<types>
  definition of types........
</types>

<message>
  definition of a message....
</message>

<portType>
  definition of a port.......
</portType>

<binding>
  definition of a binding....
</binding>

</definitions>
```

WSDL Types

- The **<types>** element defines the data types that are used by the web service.
- For maximum platform neutrality, WSDL uses XML Schema syntax to define data types.
-

Example for Wsdl:types

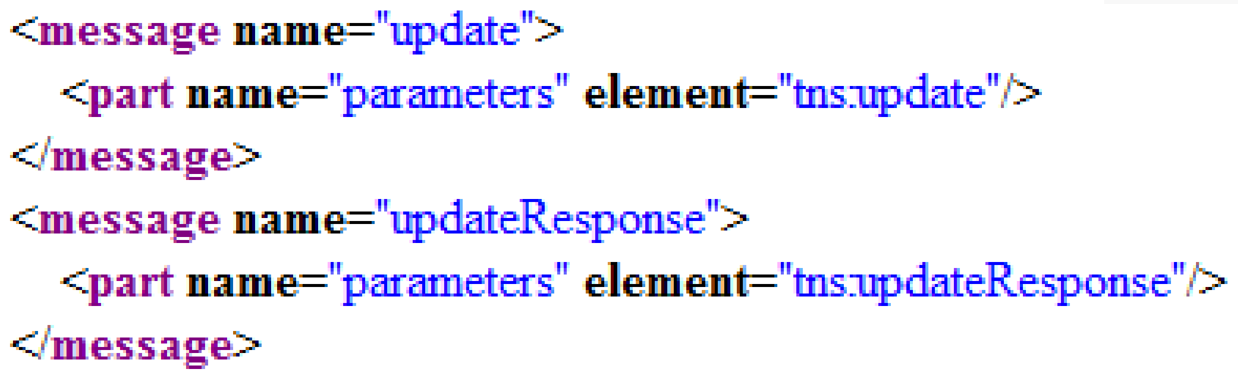
```
<wsdl:types>
     <xs:schema xmlns:tns="http://way2learn.com/" xmlns:xs="http://www.w3.org/2001/XMLSchema"
           version="1.0"   targetNamespace="http://way2learn.com/">

           <xs:element name="update" type="tns:update" />
          <xs:element name="updateResponse" type="tns:updateResponse" />

          <xs:complexType name="update">
          <xs:sequence>
          <xs:element name="arg0" type="xs:string" minOccurs="0"></xs:element>
          <xs:element name="arg1" type="xs:double"></xs:element>
          </xs:sequence>
          </xs:complexType>

          <xs:complexType name="updateResponse">
          <xs:sequence></xs:sequence>
          </xs:complexType>

      </xs:schema>
</wsdl:types>
```

WSDL Messages

- The **<message>** element defines the data elements of an operation.
- Each message can consist of one or more parts.
- The parts can be compared to the parameters of a function call in a traditional programming language.

Example for message definition

- What does tns:update refer to



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

Build COMPETENCY
across your TEAM

SYNERGETICS
GET IT RIGHT

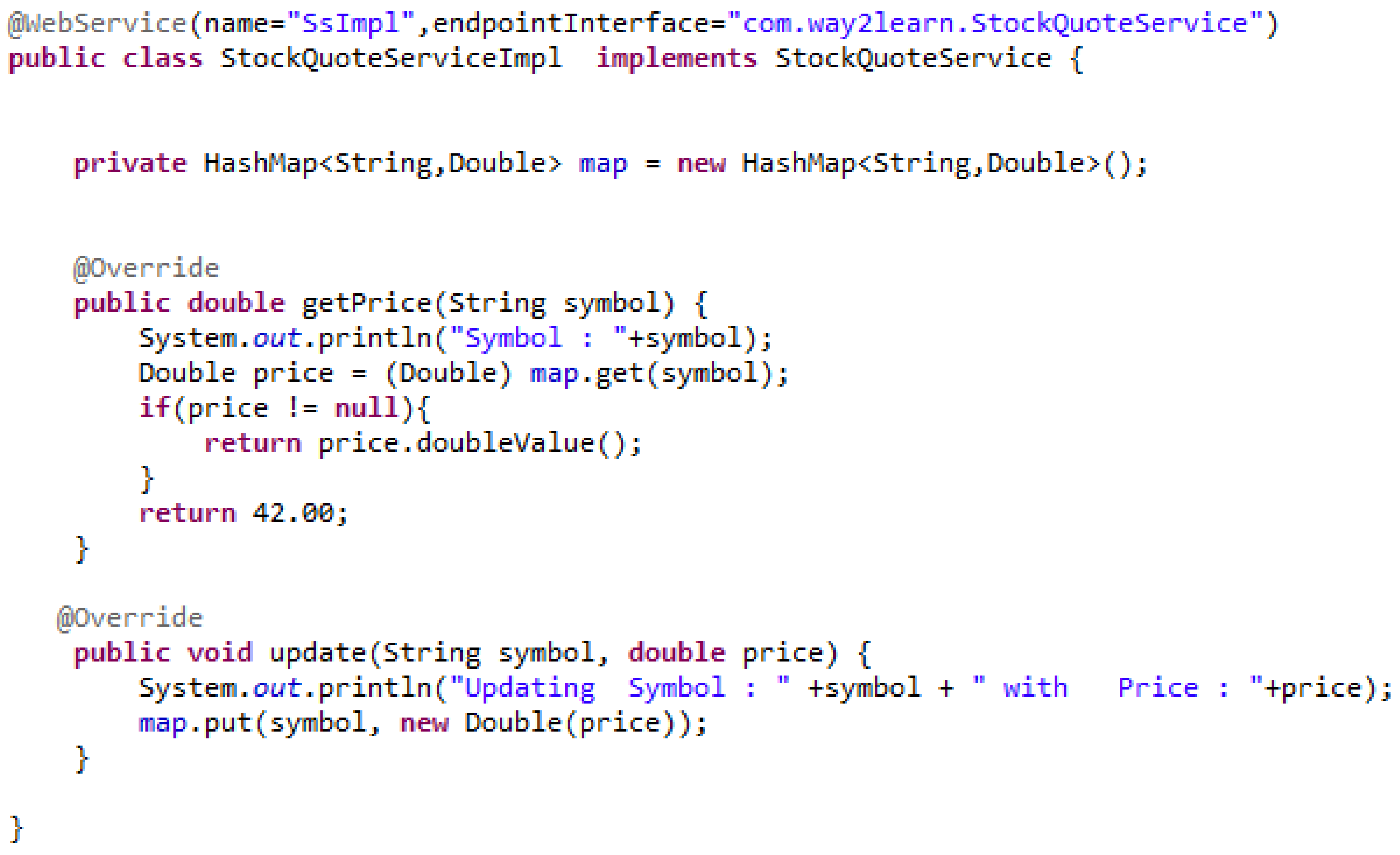
Microsoft Partner
Gold Cloud Platform
Silver Learning

Code First Webservice

Example

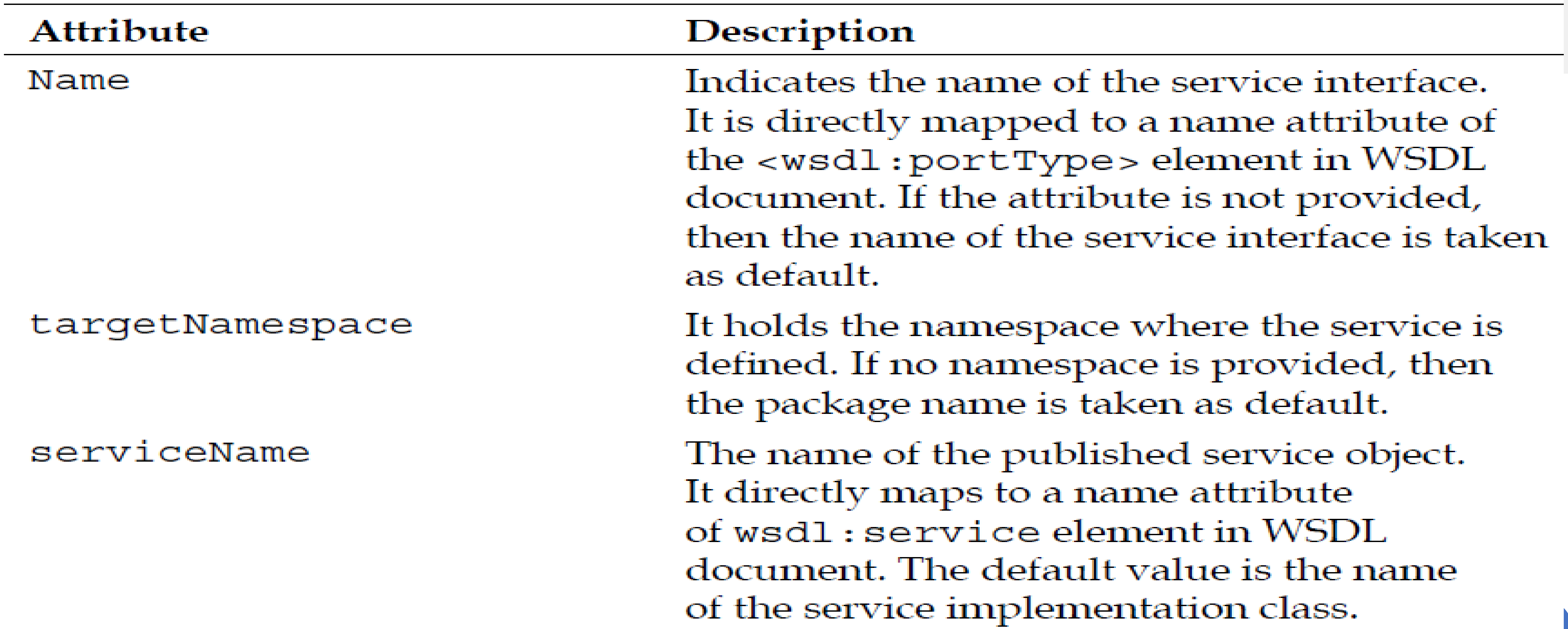
```
@WebService
public interface StockQuoteService {

    public  double getPrice(String symbol);

    public  void update(String symbol, double price) ;

}
```

Example



```
@WebService(name="SsImpl",endpointInterface="com.way2learn.StockQuoteService")
public class StockQuoteServiceImpl  implements StockQuoteService {


    private HashMap<String,Double> map = new HashMap<String,Double>();


    @Override
    public double getPrice(String symbol) {
        System.out.println("Symbol : "+symbol);
        Double price = (Double) map.get(symbol);
        if(price != null){
            return price.doubleValue();
        }
        return 42.00;
    }

   @Override
    public void update(String symbol, double price) {
        System.out.println("Updating  Symbol : " +symbol + " with   Price : "+price);
        map.put(symbol, new Double(price));
    }

}
```

@WebService attributes



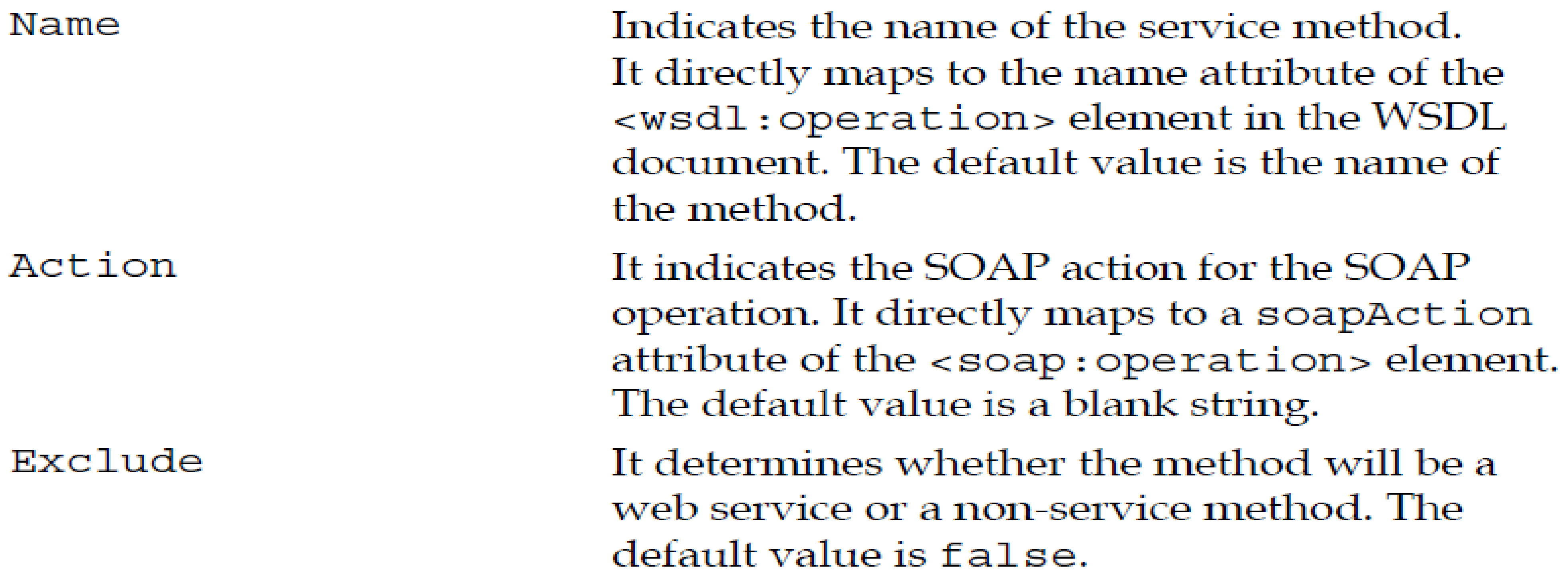
Attribute	Description
`Name`	Indicates the name of the service interface. It is directly mapped to a name attribute of the `<wsdl:portType>` element in WSDL document. If the attribute is not provided, then the name of the service interface is taken as default.
`targetNamespace`	It holds the namespace where the service is defined. If no namespace is provided, then the package name is taken as default.
`serviceName`	The name of the published service object. It directly maps to a name attribute of `wsdl:service` element in WSDL document. The default value is the name of the service implementation class.

@WebService attributes

`wsdlLocation`	It indicates the location of WSDL document in the form of URL.
`endpointInterface`	This attribute is used by the service implementation class. It specifies the fully qualified name of the service interface which will be implemented by the service implementation class.
`portName`	It indicates the name of the endpoint where the service is published. It directly maps to a name attribute of the `<wsdl:port>` element in the WSDL document.

javax.jws.WebMethod

- The **@WebMethod** annotation supports the following attributes:



`Name`	Indicates the name of the service method. It directly maps to the name attribute of the `<wsdl:operation>` element in the WSDL document. The default value is the name of the method.
`Action`	It indicates the SOAP action for the SOAP operation. It directly maps to a `soapAction` attribute of the `<soap:operation>` element. The default value is a blank string.
`Exclude`	It determines whether the method will be a web service or a non-service method. The default value is `false`.

Publishing a Webservice

```java
public class Server {

    public static void main(String[] args) {
        StockQuoteServiceImpl stockQuoteServiceImpl= new StockQuoteServiceImpl();
        Endpoint.publish("http://localhost:6060/ss", stockQuoteServiceImpl);
        System.out.println("Server Started");

    }
}
```

Looking at wsdl

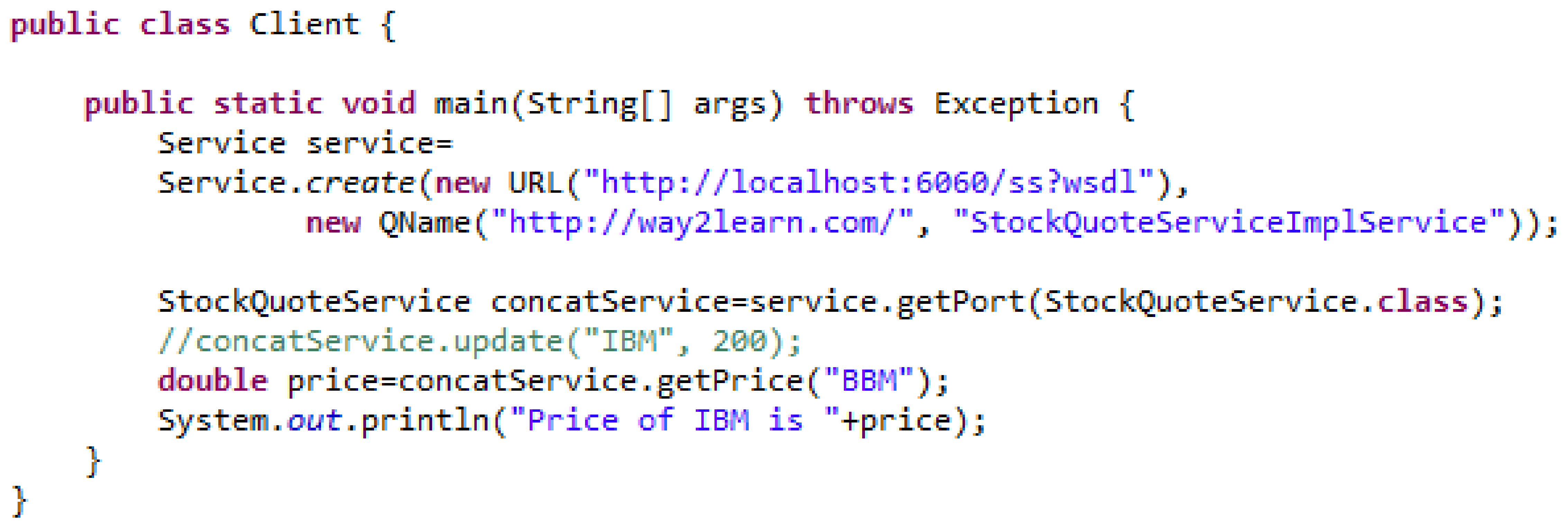
- Give a request to the following URL :

- http://localhost:6060/ss?wsdl

SOAP UI

- Send a request using SOAP UI

Consuming a webservice



```java
public class Client {

    public static void main(String[] args) throws Exception {
        Service service=
        Service.create(new URL("http://localhost:6060/ss?wsdl"),
                new QName("http://way2learn.com/", "StockQuoteServiceImplService"));

        StockQuoteService concatService=service.getPort(StockQuoteService.class);
        //concatService.update("IBM", 200);
        double price=concatService.getPrice("BBM");
        System.out.println("Price of IBM is "+price);
    }
}
```

Build COMPETENCY
across your TEAM

SYNERGETICS
GET IT RIGHT

Microsoft Partner
Gold Cloud Platform
Silver Learning

Introduction to REST

Overview

- REST is not protocol-specific .

- When people talk about REST, they usually mean
- REST over HTTP.

- REST uses HTTP extensively.

- Non-RESTful technologies like SOAP and WS-* use HTTP strictly as a transport protocol and thus use a very small subset of its capabilities.

SYNERGETICS GET IT RIGHT | Microsoft Partner Gold Cloud Platform Silver Learning

RESTful Architectural Principles

RESTful Architectural Principles

- you don’t have an “action” parameter in your URI and use only the methods of HTTP for your web services.

- Following are the HTTP methods :
  - GET
  - PUT
  - DELETE
  - POST
  - HEAD
  - OPTIONS

GET

- GET is a read-only operation. It is used to query the server for specific information.
- It is both an *idempotent and safe operation*
- Idempotent means that no matter how many times you apply the operation, the result is always the same.
- Safe means that invoking a GET does not change the state of the server at all.

PUT

- PUT requests that the server store the message body sent with the request under the location provided in the HTTP message.

- It is usually modeled as an insert or update.

- It is also idempotent.

- When using PUT, the client knows the identity of the resource it is creating or updating. It is idempotent because sending the same
- PUT message more than once has no effect on the underlying service.

- An analogy is an MS Word document that you are editing. No matter how many times you click the Save button, the file that stores your document will logically be the same document.

DELETE

- DELETE is used to remove resources.
- It is idempotent as well.

POST

- POST is the only nonidempotent and unsafe operation of HTTP.
- Each POST method is allowed to modify the service in a unique way.
- You may or may not send information with the request.
- You may or may not receive information from the response.

HEAD

- HEAD is exactly like GET except that instead of returning a response body, it returns only a response code and any headers associated with the request.

OPTIONS

- OPTIONS is used to request information about the communication options of the resource you are interested in.

- It allows the client to determine the capabilities of a server and a resource without triggering any resource action or retrieval.

Example

```
public class Customer {
   private int id;
   private String firstName;
   private String lastName;
   private String street;
   private String city;
   private String state;
   private String zip;
   private String country;
```

Model the URIs

- In a RESTful system, **endpoints are usually referred to as *resources and are identified using a URI.***
- ***URIs satisfy the addressability*** requirements of a RESTful service.
- Here is a list of URIs that will be exposed in our system:
- /customers
- /customers/{id}
- The /customers URI represents all the customers in our system.
- To access an individual customer ,we will use a pattern: /customers/{id}.

SYNERGETICS GET IT RIGHT | Microsoft Partner Gold Cloud Platform Silver Learning

Defining the Data Format

Read and Update Format

```
<customer id="771">
   <link rel="self" href="http://example.com/customers/771"/>
   <first-name>Bill</first-name>
   <last-name>Burke</last-name>
   <street>555 Beacon St.<street>
   <city>Boston</city>
   <state>MA</state>
   <zip>02115</zip>
</customer>
```

Create Format

```
<customer
    <first-name>Bill</first-name>
    <last-name>Burke</last-name>
    <street>555 Beacon St.<street>
    <city>Boston</city>
    <state>MA</state>
    <zip>02115</zip>
</customer>
```

Assigning HTTP Methods

Assigning HTTP Methods

- It is crucial that we do not assign functionality to an HTTP method that supersedes the specification-defined boundaries of that method.
- For example, **an HTTP GET on a particular resource should be read only.**
- It should not change the state of the resource it is invoking on. Intermediate services like a proxy-cache, or your browser rely on you to follow the semantics of HTTP strictly so that they can perform built-in tasks like caching effectively.
- If you do not follow the definition of each HTTP method strictly, clients
- and administration tools cannot make assumptions about your services and your system becomes more complex.

Assigning HTTP Methods

- To get a list of Customers, the remote client will call an HTTP GET
- on the URI of the object group it is interested in.

- An example request would look like the following:
- **GET /customers HTTP/1.1**

- One problem with this bulk operation is that we may have thousands of Customers in our system and we may overload our client and hurt our response time.

- So we may use :

- GET /customers**?startIndex=0&size=5 HTTP/1.1**

Obtaining Individual entities

- /customers/{id}

- Eg :
- GET   /customers/232   HTTP/1.1

SYNERGETICS GET IT RIGHT | Microsoft Partner Gold Cloud Platform Silver Learning

Creating with PUT

- The HTTP definition of PUT states that it **can be used to create or update a resource** on the server
- Eg:
- PUT /customers/112 HTTP/1.1
  >
- Content-Type: application/xml

```
<customer
    <first-name>Bill</first-name>
    <last-name>Burke</last-name>
     <street>555 Beacon St.<street>
     <city>Boston</city>
     <state>MA</state>
     <zip>02115</zip>
  </customer>
```

Creating with PUT

- PUT is required by the specification to send a **response code of 201, “Created”** if a new resource was created on the server as a result of the request.
- The HTTP specification also states that **PUT is idempotent.**
- Our PUT is idempotent, because no matter how many times we tell the server to “create” our Order, the **same bits are stored** at the /customers/112 location.

Creating with PUT

- Sometimes a PUT request will fail and the client won't know if the request was delivered and processed at the server.
- Idempotency guarantees that it's OK for the client to retransmit the PUT operation and not worry about any adverse side effects.
- The **disadvantage of using PUT to create resources is that the client has to provide the unique ID** that represents the object it is creating.
- If we want ID to be generated by the server, we can use POST.

Creating With POST

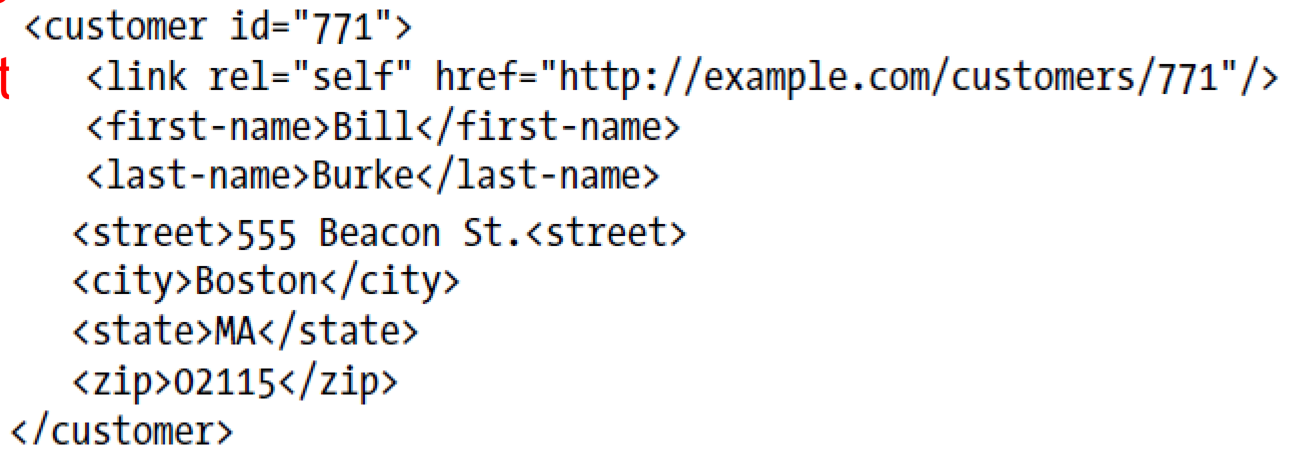
- To create a Customer with POST, the client sends a representation of the new object it is creating to the parent URI of its representation,
- leaving out the numeric target ID.

- POST /customers HTTP/1.1
- Content-Type: ication/xml

```
<customer >
   <first-name>Bill</first-name>
   <last-name>Burke</last-name>
    <street>555 Beacon St.<street>
    <city>Boston</city>
    <state>MA</state>
    <zip>02115</zip>
  </customer>
```

Creating With POST

- What if the client wants to edit, update, or cancel the order it just posted?
- What is the ID of the new order?
- To resolve this issue, we will add a bit of additional information to the HTTP response message.
- HTTP/1.1 201 Created
- Content-Type: application/xml
- Location: htt



```
<customer id="771">
   <link rel="self" href="http://example.com/customers/771"/>
   <first-name>Bill</first-name>
   <last-name>Burke</last-name>
  <street>555 Beacon St.<street>
  <city>Boston</city>
  <state>MA</state>
  <zip>02115</zip>
</customer>
```

Creating With POST

- HTTP requires that if POST creates a new resource that it respond with a code of 201, “Created” (just like PUT).

- The Location header in the response message provides a URI
- to the client so it knows where to further interact with the Customer that was created, i.e., if the client wanted to update the Customer.

- **It is optional whether the server sends the representation of the newly created Order with the response.**

Updating a Customer

- PUT /customers/117 HTTP/1.1
- Content-Type: application/xml

```
<customer id="117">
    <first-name>Bill</first-name>
    <last-name>Burke</last-name>
    <street>555 Beacon St.<street>
    <city>Boston</city>
    <state>MA</state>
    <zip>02115</zip>
 </customer>
```

- When a resource is updated with PUT, the HTTP specification requires that you send **a response code of 200, “OK” and a response message body** or **a response code of 204,**
- **“No Content” without any response body**

Deleting a customer

- The client simply invokes the DELETE method on the exact URI that represents the object we want to remove.
- When a resource is removed with DELETE, the HTTP specification requires that you send a response code of **200, “OK” and a response message body or a response code of 204, “No Content”** without any response body.

Cancelling an Order

- While removing an object wipes it clean from our databases, cancelling only changes the state of the Order and retains it within the system.

- How should we model such an operation?

Overloading the meaning of DELETE

- Cancelling an Order is very similar to removing it.

- One thing we could do is add an extra query parameter
- to the request:

- DELETE /orders/233?cancel=true HTTP/1.1

- It is not good RESTful design as you are changing the meaning of the uniform interface.
- Using a query parameter in this way is actually creating **a mini-RPC mechanism.**
- HTTP specifically states that **DELETE is used to delete a resource from the server, not cancel it.**

States versus operations

- When modelling a RESTful interface for the operations of your object model, you should ask yourself a simple question:

- **is the operation a state of the resource?**

- If you answer yes to this question, the operation should be modeled **within the data format.**

States versus operations

- Since the state of being cancelled is modeled in the data format, we can now use our already defined mechanism of updating an Order to model the cancel operation.
- For example, we could PUT this message to our service:

```
PUT /orders/233 HTTP/1.1
Content-Type: application/xml

<order id="233">
   <total>$199.02</total>
   <date>December 22, 2008 06:56</date>
   <cancelled>true</cancelled>
...
</order>
```

Using Subresource

- What if we expanded on our cancel example by saying that we wanted a way to clean up all cancelled orders?
- We can't really model purging the same way we did cancel.
- To solve this problem, **we model this operation as a sub resource of /orders** and we trigger a purging by doing a POST on that resource. For example:
- **POST /orders/purge HTTP/1.1**

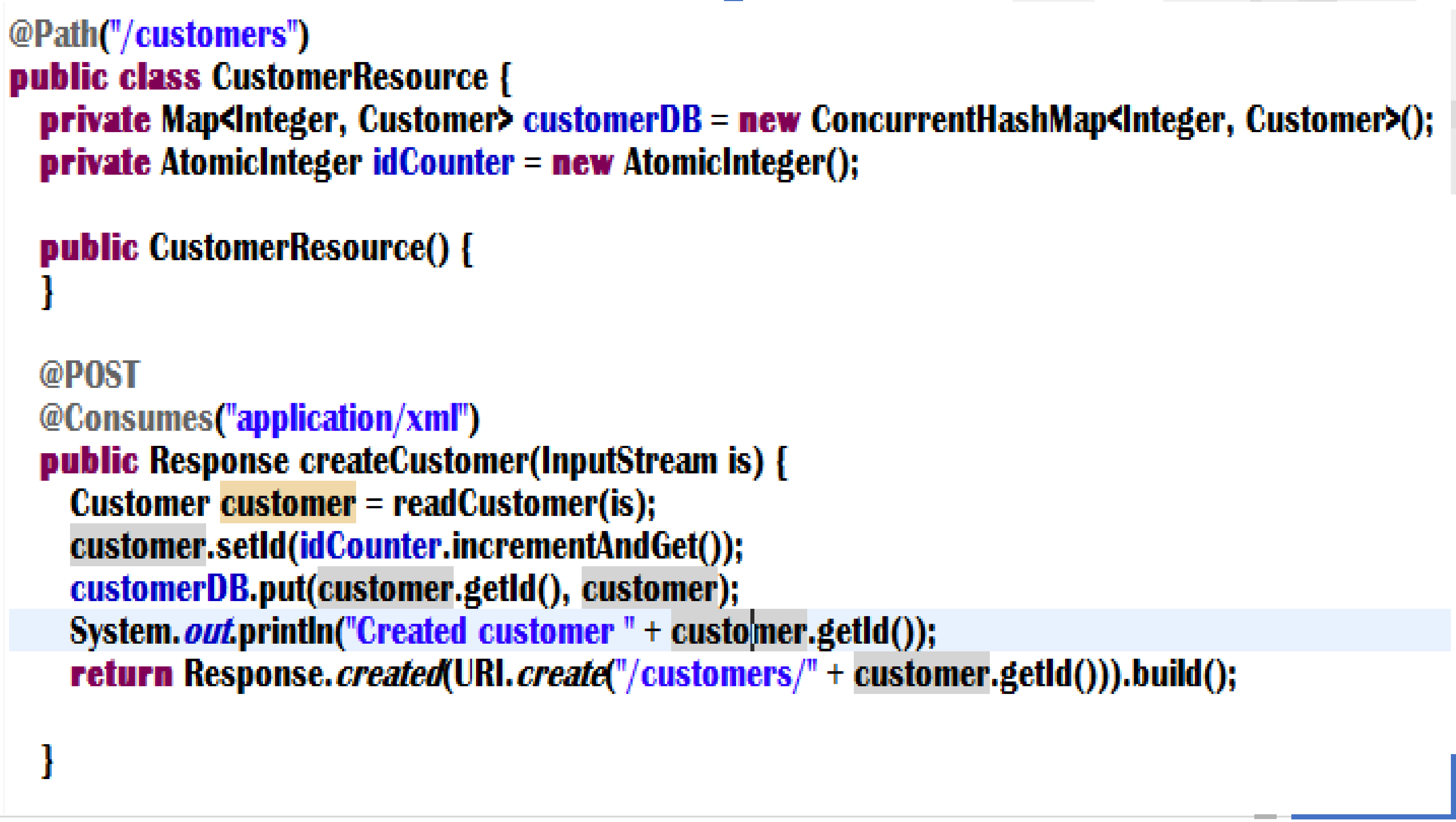
Using Subresource

- An interesting side effect of this is that because purge is now a URI, we can evolve its interface over time.

- For example, maybe GET /orders/purge returns a document that
- states the last time a purge was executed and which orders were deleted.

- What if we wanted to add some criteria for purging as well?
  - Form parameters could be passed stating that we only want to purge orders older than a certain date.

Example



```java
@Path("/customers")
public class CustomerResource {
   private Map<Integer, Customer> customerDB = new ConcurrentHashMap<Integer, Customer>();
   private AtomicInteger idCounter = new AtomicInteger();

   public CustomerResource() {
   }

   @POST
   @Consumes("application/xml")
   public Response createCustomer(InputStream is) {
      Customer customer = readCustomer(is);
      customer.setId(idCounter.incrementAndGet());
      customerDB.put(customer.getId(), customer);
      System.out.println("Created customer " + customer.getId());
      return Response.created(URI.create("/customers/" + customer.getId())).build();

   }
```

Example

```
@GET
@Path("{id}")
@Produces("application/xml")
public StreamingOutput getCustomer(@PathParam("id") int id) {
   final Customer customer = customerDB.get(id);
   if (customer == null) {
      throw new WebApplicationException(Response.Status.NOT_FOUND);
   }
   return new StreamingOutput() {
      public void write(OutputStream outputStream) throws IOException, WebApplicationException {
         outputCustomer(outputStream, customer);
      }
   };
}
```

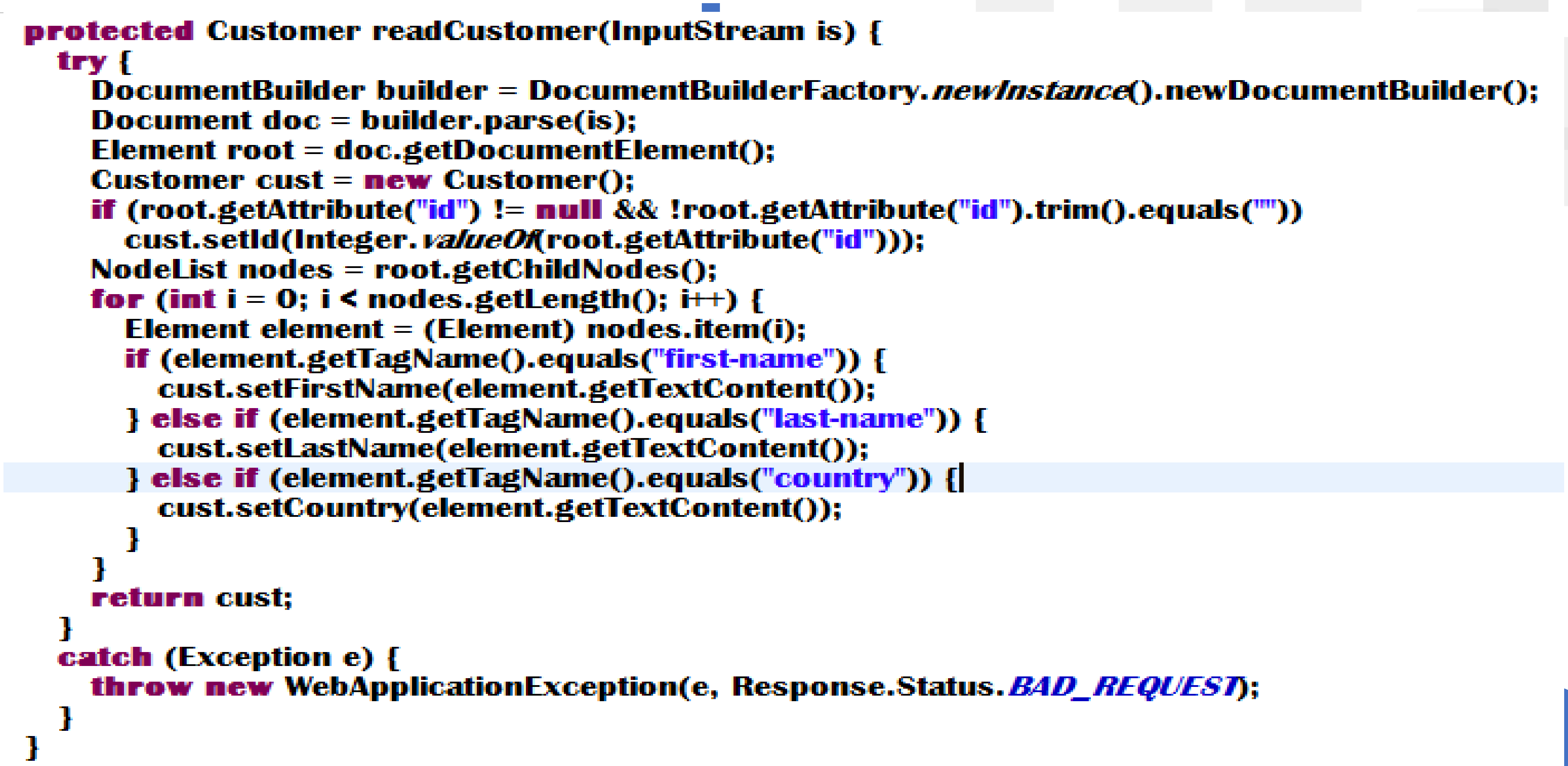
Example

```
@PUT
@Path("{id}")
@Consumes("application/xml")
public void updateCustomer(@PathParam("id") int id, InputStream is) {
   Customer update = readCustomer(is);
   Customer current = customerDB.get(id);
   if (current == null) throw new WebApplicationException(Response.Status.NOT_FOUND);

   current.setFirstName(update.getFirstName());
   current.setLastName(update.getLastName());
   current.setStreet(update.getStreet());
   current.setState(update.getState());
   current.setZip(update.getZip());
   current.setCountry(update.getCountry());
}
```

Example

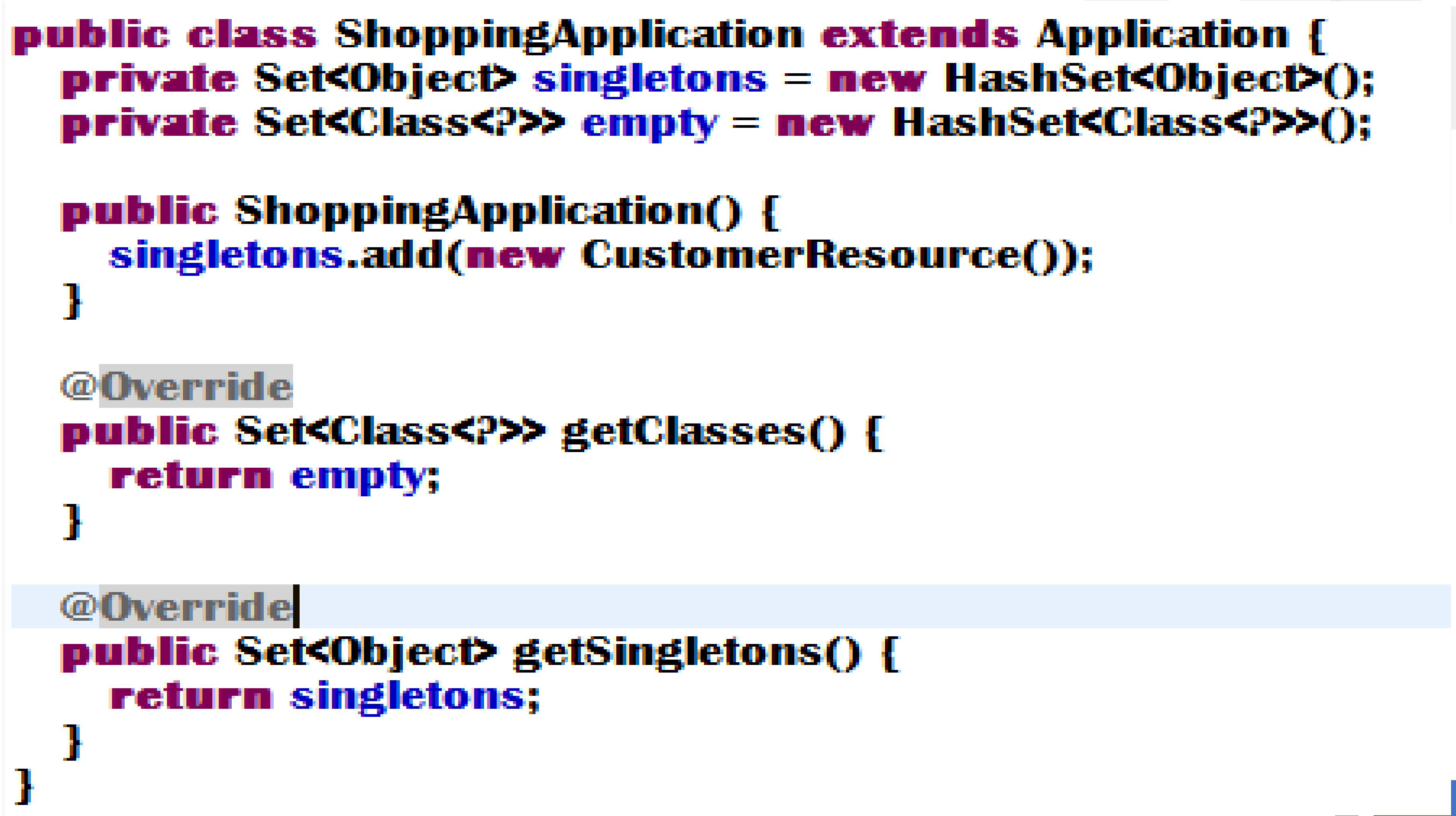
```
protected void outputCustomer(OutputStream os, Customer cust) throws IOException {
   PrintStream writer = new PrintStream(os);
   writer.println("<customer id=\"" + cust.getId() + "\">");
   writer.println("   <first-name>" + cust.getFirstName() + "</first-name>");
   writer.println("   <last-name>" + cust.getLastName() + "</last-name>");
   writer.println("   <street>" + cust.getStreet() + "</street>");
   writer.println("   <city>" + cust.getCity() + "</city>");
   writer.println("   <state>" + cust.getState() + "</state>");
   writer.println("   <zip>" + cust.getZip() + "</zip>");
   writer.println("   <country>" + cust.getCountry() + "</country>");
   writer.println("</customer>");
}
```

Example



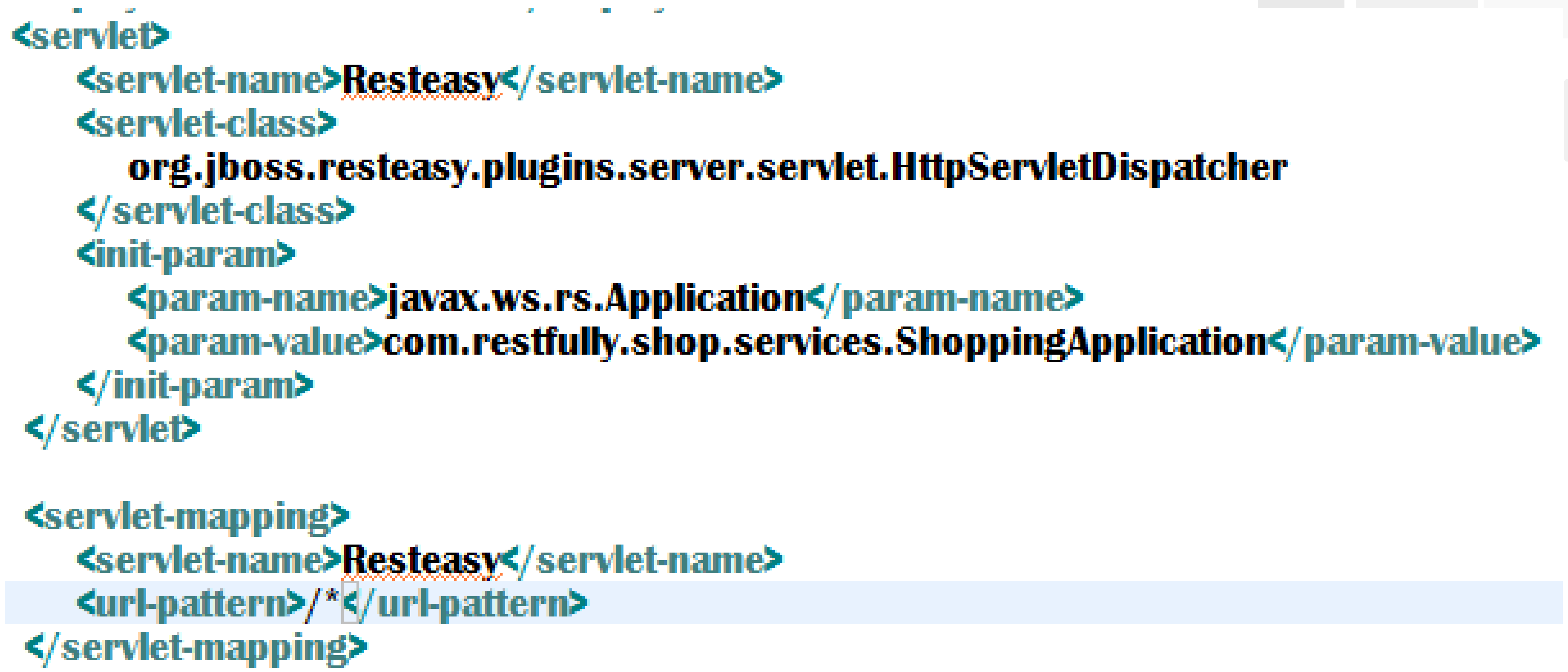
```
protected Customer readCustomer(InputStream is) {
  try {
    DocumentBuilder builder = DocumentBuilderFactory.newInstance().newDocumentBuilder();
    Document doc = builder.parse(is);
    Element root = doc.getDocumentElement();
    Customer cust = new Customer();
    if (root.getAttribute("id") != null && !root.getAttribute("id").trim().equals(""))
      cust.setId(Integer.valueOf(root.getAttribute("id")));
    NodeList nodes = root.getChildNodes();
    for (int i = 0; i < nodes.getLength(); i++) {
      Element element = (Element) nodes.item(i);
      if (element.getTagName().equals("first-name")) {
        cust.setFirstName(element.getTextContent());
      } else if (element.getTagName().equals("last-name")) {
        cust.setLastName(element.getTextContent());
      } else if (element.getTagName().equals("country")) {
        cust.setCountry(element.getTextContent());
      }
    }
    return cust;
  }
  catch (Exception e) {
    throw new WebApplicationException(e, Response.Status.BAD_REQUEST);
  }
}
```

Shopping Application



```
public class ShoppingApplication extends Application {
   private Set<Object> singletons = new HashSet<Object>();
   private Set<Class<?>> empty = new HashSet<Class<?>>();

   public ShoppingApplication() {
      singletons.add(new CustomerResource());
   }

   @Override
   public Set<Class<?>> getClasses() {
      return empty;
   }

   @Override
   public Set<Object> getSingletons() {
      return singletons;
   }
}
```

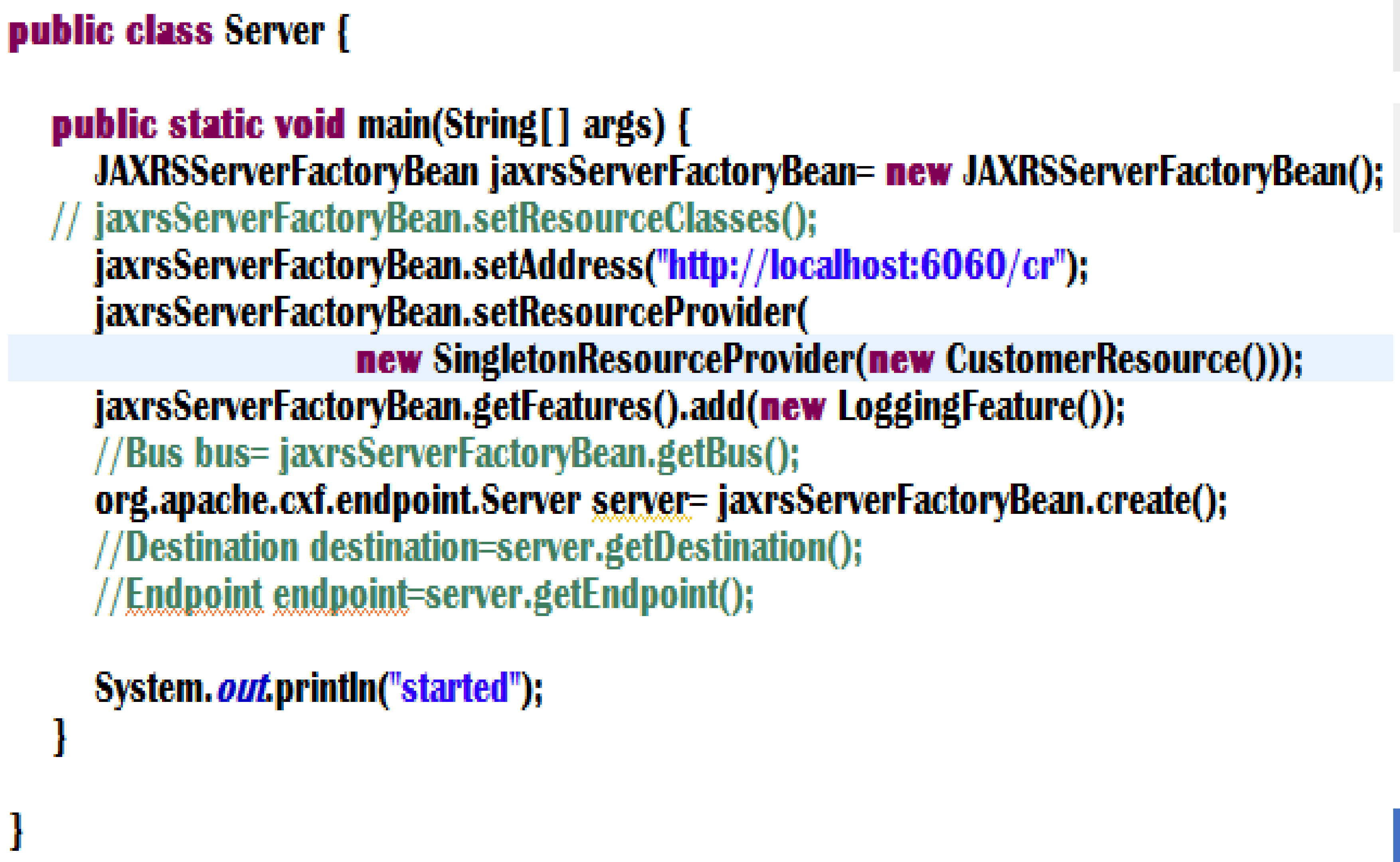
xml



```
<servlet>
    <servlet-name>Resteasy</servlet-name>
    <servlet-class>
        org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher
    </servlet-class>
    <init-param>
        <param-name>javax.ws.rs.Application</param-name>
        <param-value>com.restfully.shop.services.ShoppingApplication</param-value>
    </init-param>
</servlet>

<servlet-mapping>
    <servlet-name>Resteasy</servlet-name>
    <url-pattern>/*</url-pattern>
</servlet-mapping>
```

Using CXF without server

Using CXF without server



```
public class Server {

    public static void main(String[] args) {
        JAXRSServerFactoryBean jaxrsServerFactoryBean= new JAXRSServerFactoryBean();
    // jaxrsServerFactoryBean.setResourceClasses();
        jaxrsServerFactoryBean.setAddress("http://localhost:6060/cr");
        jaxrsServerFactoryBean.setResourceProvider(
                        new SingletonResourceProvider(new CustomerResource()));
        jaxrsServerFactoryBean.getFeatures().add(new LoggingFeature());
        //Bus bus= jaxrsServerFactoryBean.getBus();
        org.apache.cxf.endpoint.Server server= jaxrsServerFactoryBean.create();
        //Destination destination=server.getDestination();
        //Endpoint endpoint=server.getEndpoint();

        System.out.println("started");
    }

}
```

Using CXF without server

- The **JAXRSServerFactoryBean** creates a Server inside CXF which starts listening for requests on the URL specified.

- setResourceClasses() is for root resources only, use setProvider() or **setProviders() for @Provider-annotated classes.**

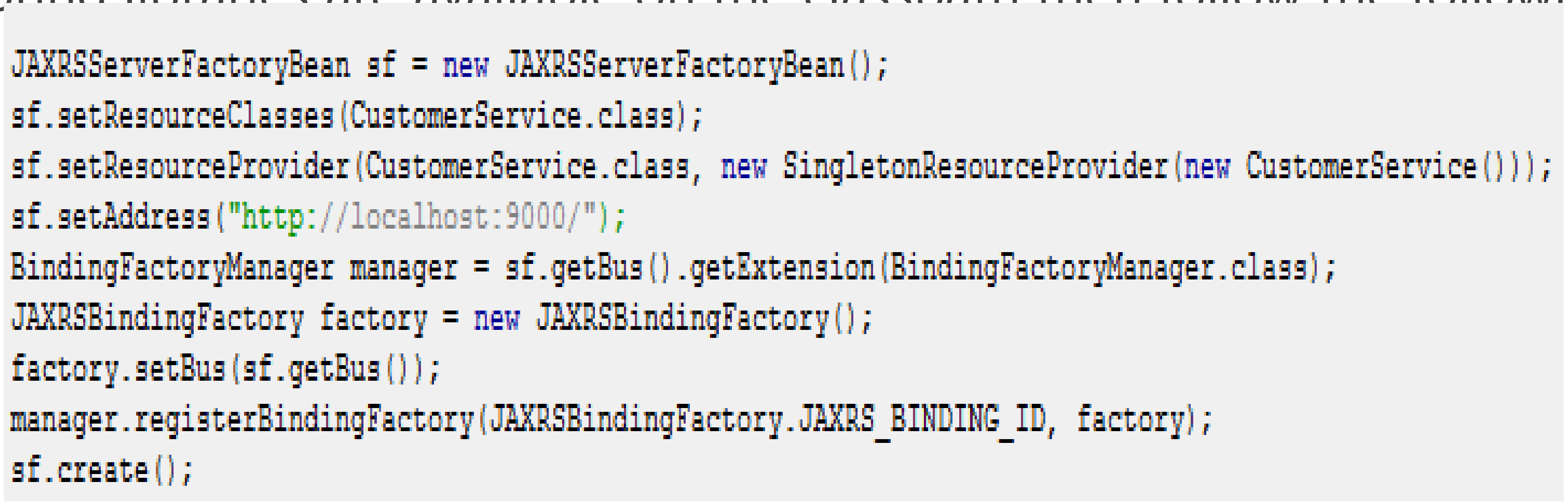
- By default, the JAX-RS runtime is responsible for the lifecycle of resource classes, **default lifecycle is per-request**. You can set the lifecycle to singleton by using following line:

```
jaxrsServerFactoryBean.setResourceProvider(
                    new SingletonResourceProvider(new CustomerResource()));
```

Configuring JAX-RS endpoints programmatically without Spring

- Even though no Spring is explicitly used in the previous configuration, it is still used by default to have various CXF components registered with the bus such as transport factories.

- If no Spring libraries are available on the classpath then follow the following exampl



```
JAXRSServerFactoryBean sf = new JAXRSServerFactoryBean();
sf.setResourceClasses(CustomerService.class);
sf.setResourceProvider(CustomerService.class, new SingletonResourceProvider(new CustomerService()));
sf.setAddress("http://localhost:9000/");
BindingFactoryManager manager = sf.getBus().getExtension(BindingFactoryManager.class);
JAXRSBindingFactory factory = new JAXRSBindingFactory();
factory.setBus(sf.getBus());
manager.registerBindingFactory(JAXRSBindingFactory.JAXRS_BINDING_ID, factory);
sf.create();
```

SOA Concepts

- SOA is an architecture approach for defining, linking, and integrating reusable business services that have clear boundaries and are self-contained with their own functionalities. Within this type of architecture, you can orchestrate the business services in business processes. Adopting the concept of services—a higher-level abstraction that's independent of application or infrastructure IT platform and of context or other services—SOA takes IT to another level, one that's more suited for interoperability and heterogeneous environments.

Introducing Terminology

- Business Processes
- Services
- Service-Oriented Architecture

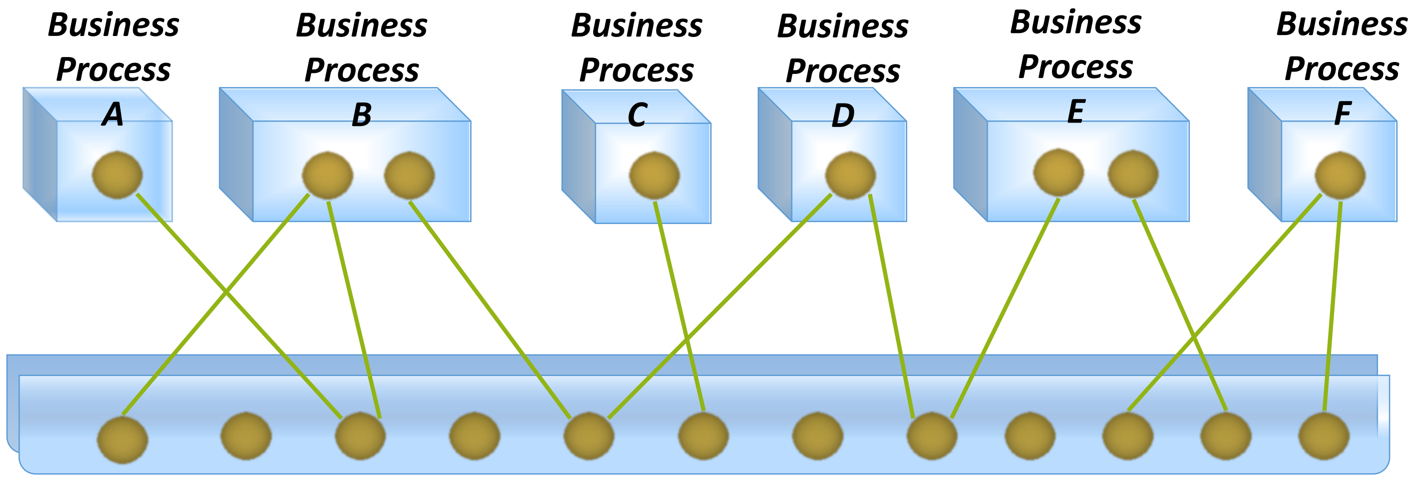
Services

- Physically independent software programs with distinct design characteristics that facilitate
  - Loose coupling
  - Reusability
  - Composition
  - Discovery

  through a standardized service contract.

Service Compositions

- Coordinated aggregate or assembly of services to provide the functionality required to automate a specific business task or process.
- Services with functional contexts that are agnostic to any one business process are capable of participating in multiple service compositions.



Business Process Agnostic Services

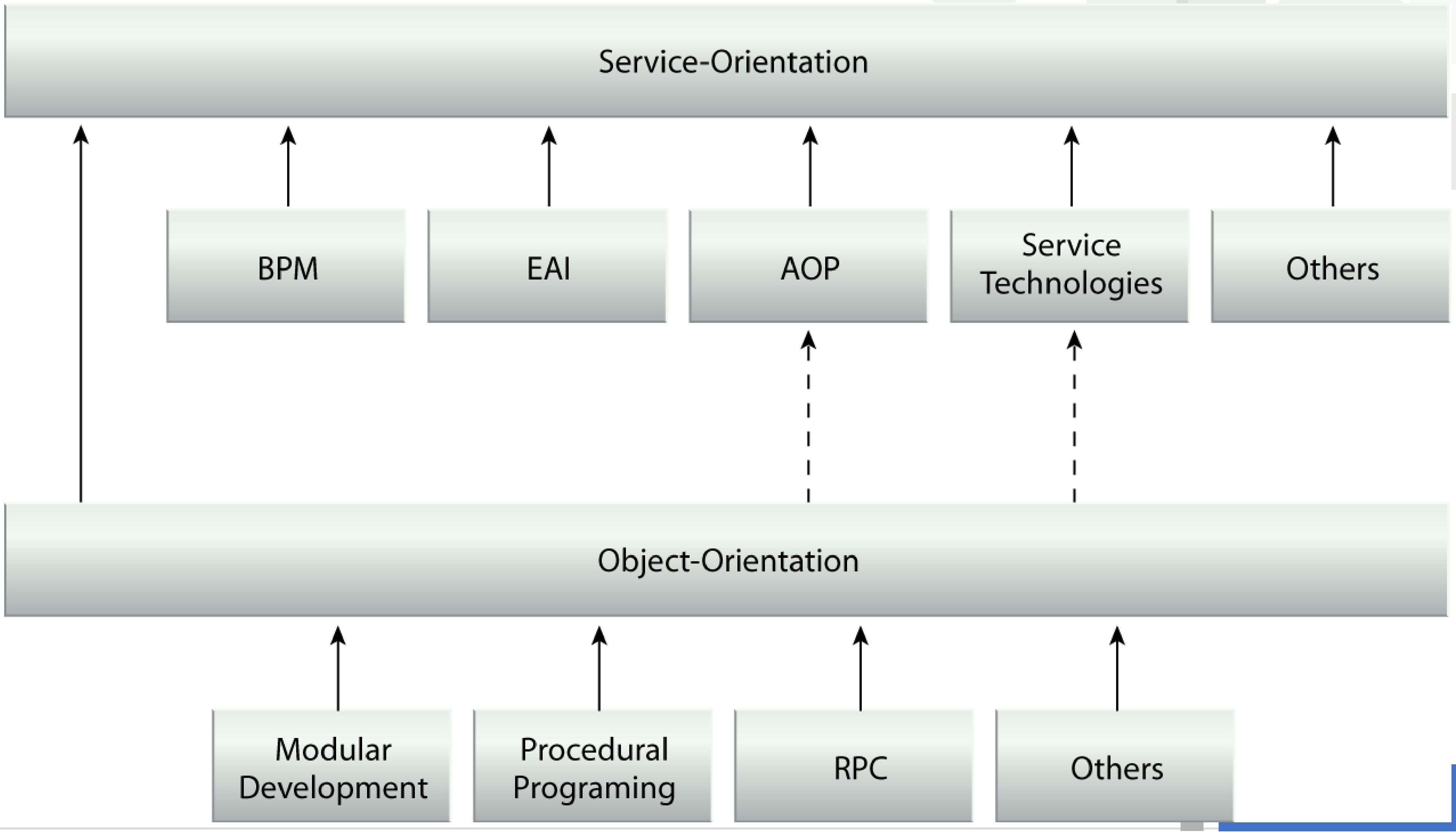
Service Inventory

- An independently standardized and governed collection of complementary services.
- A single, enterprise-wide inventory, or
  multiple service inventories, each of which may be individually standardized, governed, and supported by its own service-oriented technology architecture.
- Typically created through top-down delivery processes.
- Application of service-oriented design principles and custom design standards through a service inventory is important to establish a high degree of inter-service interoperability.

Service-Oriented Architecture

- Technology architecture designed to support service-oriented solution logic, comprising services and service compositions.
- Standards-based platform that allows services to be developed, discovered and consumed by each other, to facilitate the creation of an orchestrated business process.

Origins of SOA



Benefits of SOA

- Loosely-coupled architecture
- Promotes architectural composability
- Increases quality of service
- Based on open standards
- Supports vendor diversity
- Increases reusability
- Promotes organizational agility

SOA Technology Platform

- The most common technology platform for realization of SOA is Web Services.
- Web Service, basic standards:
  - XML
  - WSDL: Web Services Description Language
  - XSD: XML Schema Definition Language
  - SOAP: Simple Object Access Protocol
  - UDDI: Universal Description, Discovery, and Integration
  - WS-I Basic Profile

WS-* Extensions

- WS-Addressing
- WS-ReliableMessaging
- WS-Policy
- WS-Metadata Exchange
- WS-Security
- And many more

Build COMPETENCY
across your TEAM

SYNERGETICS GET IT RIGHT

Microsoft Partner
Gold Cloud Platform
Silver Learning

SOA Principles

Addition Reading

Principles of SOA

- Standardized service contract
- Service loose coupling
- Service abstraction
- Service reusability
- Service autonomy
- Service statelessness
- Service discoverability
- Service composability

Standardized Service Contract

- Services express their purpose and capabilities via a service contract.
- In the form of:
  - WSDL definition
  - XML schema definition
  - WS-Policy description
  - Non-technical SLA, such as availability, response time usage statistics
- Service contracts should be optimized, appropriately granular, and standardized to ensure that the endpoints established by the services are consistent, reliable and governable.

Service Loose Coupling

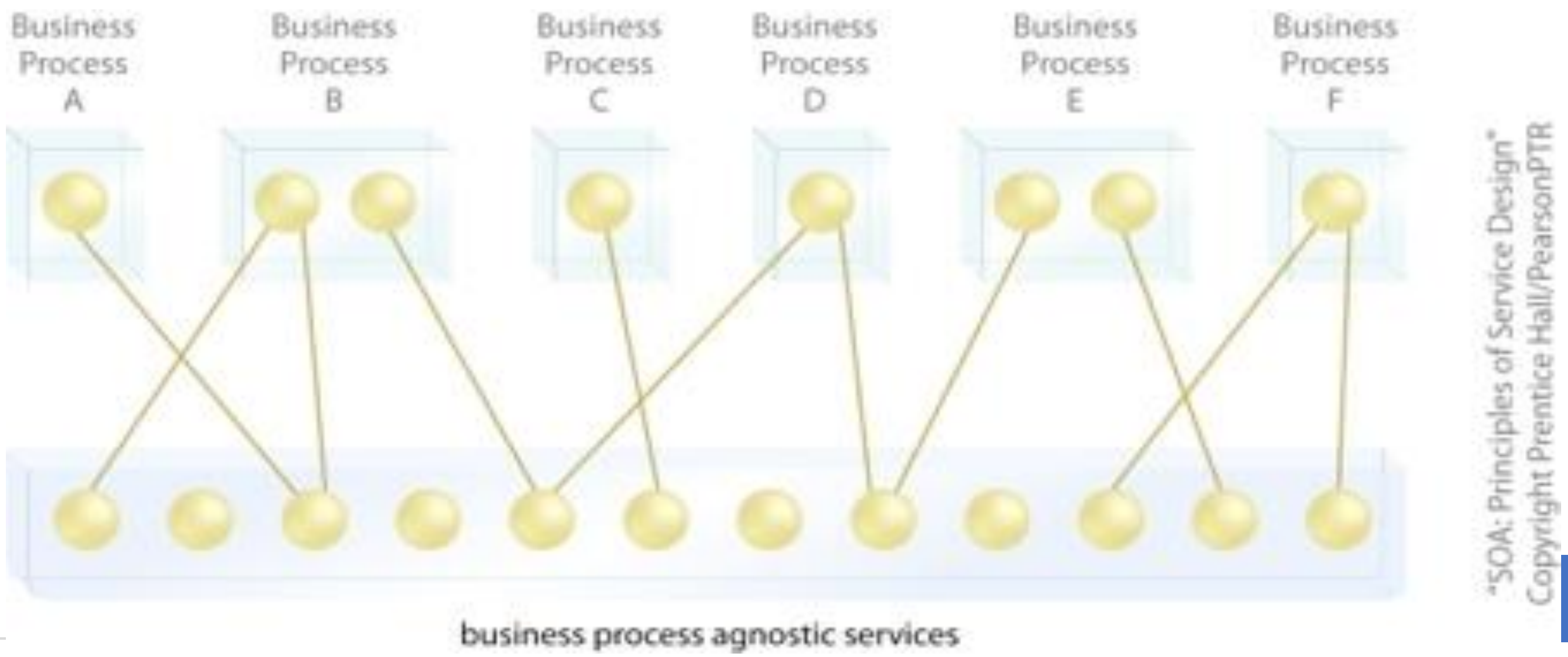
- Service logic and implementation should be designed and evolved independently while still guaranteeing baseline interoperability with consumers.
- Different levels of service coupling:
  - Service contract and underlying logic may be coupled to a parent business process
  - Service consumers are coupled to the service contract
  - Service logic is implemented in a proprietary vendor technology
  - Service logic may be coupled with multiple services it may need to compose
  - Service logic may be coupled to various resources that are part of the overall implementation environment.

Service Abstraction

- Service abstraction enables loose coupling.
- A service may encapsulate:
  - A legacy system
  - Custom components
  - Other services

Service Reusability

- Designing services that are agnostic to a specific business process increases their reusability.



Service Autonomy

- Autonomy of a service is its ability to carry out its core service logic independently.

Service Statelessness

- Management of excessive state information can compromise the availability of a service.

Service Discoverability

- Services are supplemented with communicative meta data by which they can be effectively discovered and interpreted.

Service Composability

- Services should be capable of participating as effective composition members, even if they don't need to be immediately enlisted in a composition.
- To solve a problem, its solution logic is decomposed into services. However, the ultimate, strategic benefits comes from the ability to continually recompose these services to solve additional problems in the future.
- Complex service composition requires anticipating service design to avoid massive retro-fitting efforts.

Challenges Introduced by SOA

- Design Complexity
  - Increased complexity of both the architecture and the design of individual services.
  - Increased performance considerations from the use of agnostic services.
  - Reliability issues of services at peak concurrent usage times.
  - Single point of failure for multiple processes from excessive reuse of agnostic services.
  - Service contract versioning issues.
  - Impact of potentially redundant service contracts.
- Need for design standards
  - Need to enforce their compliance
  - Resistance to standardisation from architects and developers.

Challenges Introduced by SOA

- Top-Down Requirements:
  - Need to design planned services, their relationships, boundaries, etc. upfront.
  - Imposes a significant amount of upfront analysis effort involving many business analysts and architects.
  - Time and budget constraints in creating such a blueprint upfront.
  - Counter-agile delivery.
- Governance Demands:
  - Body of solution logic for a service does not belong to any one process. The team that delivers a service solution may not continue to own the service logic as it gets repeatedly used by other solutions, processes and compositions.
  - Therefore, special governance structures may be required, introducing new roles, processes and even departments.

SOA Project Lifecycle Stages

- SOA Adoption Planning
- Service Inventory Analysis
- Service-Oriented Analysis (Service Modeling)
- Service-Oriented Design (Service Contract)
- Service Logic Design
- Service Development
- Service Testing
- Service Deployment and Maintenance
- Service Usage and Monitoring
- Service Versioning and Retirement

SYNERGETICS GET IT RIGHT | Microsoft Partner Gold Cloud Platform Silver Learning

Q & A

Contact: smitakumar@synergetics-india.com

Build COMPETENCY
across your TEAM

SYNERGETICS
GET IT RIGHT

Microsoft Partner
Gold Cloud Platform
Silver Learning

Thank You