WebStore #maven

JAX-WS SOAP WS, Stateful Session Bean, PMD, JavaDoc

Óbuda University, Java Enterprise Edition
John von Neumann Faculty of Informatics
Lab 8

Dávid Bedők
2018-03-03
v1.0

SOAP webservices

- 1998, 2000 (v1.1), 2003 (v1.2 W3C recommendation)
- **Simple Object Access Protocol** (`SOAP`), but after the version 1.2 we do not use the 'expanded' form
- For an `XML SOAP` **Request** message an `XML SOAP` **Response** message will arrived (an `XSD` describes the type information)
- **Web Services Description Language** (`WSDL`) describes the communication and the rules
- In general it is sent via `HTTP(S)` (as in case of the *RESTful* webservices), but this is only a frequent option here
- Highly extensible, great and sophisticated, guided by standards
- Encryption and digital signature supprt - **Web Services Security** (`WS-Security`, `WSS`)
- **Universal Description Discovery and Integration** (`UDDI`), webservice registration in finders

Top-Down vs. Bottom-Up approach

- **Top-down** approach: we create a `WSDL` then we are generated the required Java classes (*stubs* (`XSD`) and *service* classes based on the `WSDL`).
- **Bottom-up** approach: we create Java code (*stubs*, *services* with annotations) then we are generate a `WSDL`.

Regarding the client

To create a client application we only need the `WSDL` (which is an XML document, so that file is the key for the cross-language behavior). This file contains the XSD(s) which grants the type-safe communication between the client and the server.

Configuration
SOAP message format (four combinations)

- **Encoding Style/Communication Style** (named attribute in the WSDL)
  - **Document/Message-Oriented**: freely formable XML content
  - **RPC**: more bound (more detailed often) but easier to process by machines
- **Encoding Models** (named attribute in the WSDL)
  - **Literal**: the content must fit the type information of the XSD (*XSD validation*). We can easily transform this message to something else with XSLT transformation (e.g.: create XHTML documents for a webpage/web-application)
  - **Encoded**: only predefined XSD types can be used (cumbersome validation)

In case of *Document style* we have the change to validate the *SOAP Response*, but the same thing in case of *RPC* is very circumstantial. We will choose the `Document/literal` SOAP message format in the blueprint project.

Configuration
Parameter Stlye

▷ **Parameter Style** (different references inside the `WSDL`)

- **BARE**: Do not wrap anything in the `SOAP` messages. If we would like readable results we have to create wrapper classes for the parameters and the return values.
- **WRAPPED**: the parameters of the requests and the responses will be wrapped around the messages (more transparent but it generates bigger `SOAP` messages)

We will choose the `WRAPPED` *Parameter Style* in the blueprint project.

WSDL overview

Based on the implemented webstore blueprint

```
<wsdl:definitions [..]>
  <wsdl:types>
    <xs:schema xmlns:xs="http://www.w3.org/200
         xmlns:tns="http://www.qwaevisz.hu/We
         targetNamespace="http://www.qwaevisz
         version="1.0">[..]</xs:schema>
  </wsdl:types>
  <wsdl:message name="[..]">[..]</wsdl:message
  <wsdl:portType name="WebStore">
    <wsdl:operation name="[..]">
      <wsdl:input message="[..]" name="[..]"></wsdl:input>
      <wsdl:output message="[..]" name="[..]"></wsdl:output>
      <wsdl:fault message="[..]" name="[..]"></wsdl:fault>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="WebStoreServiceSoapBinding" type="tns:WebStore">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="[..]">
      <soap:operation soapAction="[..]" style="document"/>
      <wsdl:input name="[..]"><soap:body use="literal"/></wsdl:input>
      <wsdl:output name="[..]"><soap:body use="literal"/></wsdl:output>
      <wsdl:fault name="[..]"><soap:fault name="[..]" use="literal"/></wsdl:fault>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="WebStoreService">
    <wsdl:port binding="tns:WebStoreServiceSoapBinding" name="WebStorePort">
      <soap:address location="http://localhost:8080/webstore/WebStoreService"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

The parts of the WSDL refer to each other. The service refers the binding, the binding refers the portType and within that the operation(s), the portType's operation(s) refer(s) the message(s) and the message(s) refer(s) the items of the types (XSD of the WSDL).

SOAP messages - Request and Response

```xml
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:web="http://www.qwaevisz.hu/WebStore">
   <soapenv:Header/>
   <soapenv:Body>
      <web:ListAllProducts/>
   </soapenv:Body>
</soapenv:Envelope>
```

SOAP Request

The http://schemas.xmlsoap.org/soap/envelope/ namespace defines the basic elements and attributes (e.g.:. Envelope, Header, Body, etc.), while the http://www.qwaevisz.hu/WebStore namespace defines the application's domain (e.g.: ListAllProducts, ListAllProductsResponse).

```xml
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
      <ns2:ListAllProductsResponse xmlns:ns2="http://www.qwaevisz.hu/WebStore">
         <Product brand="SONY" productName="ZD9 4K HDR" price="1499"/>
         <Product brand="SONY" productName="SD85 4K HDR" price="1299"/>
         <Product brand="SONY" productName="XD83 4K HDR" price="1299"/>
         <Product brand="PHILIPS" productName="40PFH5500 Smart Led" price="999"/>
         <Product brand="PANASONIC" productName="TX-40CS620E LED " price="1350"/>
         <Product brand="PANASONIC" productName="TX-58DX800E" price="699"/>
      </ns2:ListAllProductsResponse>
   </soap:Body>
</soap:Envelope>
```

SOAP Response

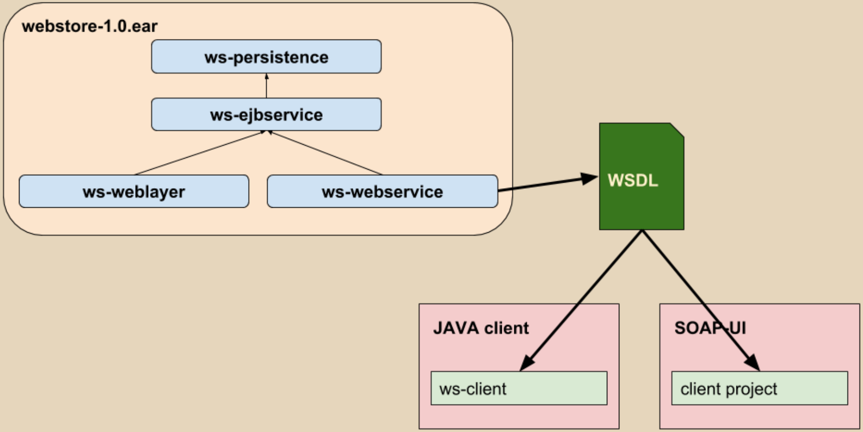
Task: Create a webstore application where you can buy various brands TV. The application will store the (stock) items in memory and do not deal with the authentication of the users.
Build a `SOAP` **webservice** which able to **query the stock items** and an other one which **handles the user's webbasket**.
The webbasket stores items (brand, name, price) during the lifetime of he *user-session*. If the user buys from the same item more than one, change only the quantity in his/her basket.

`[gradle|maven]\jboss\webstore`

- The customer may set an identifier for the basket and he/she can retrieve that value later (only demonstration purpose). If we try to reach a nameless basket the system gives us a **SOAP Fault** error message.
- We will implement a **Stateful Session Bean** to handle the state of the webbasket.
- We are going to use the **SOAP-UI** application as a SOAP client application, but we will write a custom standalone **Java client** application as well.

Project structure

Location of the modules



Persistence layer

ws-persistence project

```java
@Local
public interface StoreHolder {
  void create(Product product);
  Product read(String name);
  List<Product> readAll(String nameTer
  List<Product> readAll(Brand brand);
  List<Product> readAll();
  void delete(String name);
}
```

The `StoreHolder` **Singleton Session Bean** stores (and initializes) the *domain* types (`Brand` and `Product`), and the `PersistenceService` **Stateless Session Bean** ensures access for the services of the EJB layer.

StoreHolder.java

```java
@Local
public interface PersistenceService {
  void create(Product product) throws PersistenceException;
  Product read(String name) throws PersistenceException;
  List<Product> readAll(String nameTerm) throws PersistenceException;
  List<Product> readAll(Brand brand) throws PersistenceException;
  List<Product> readAll() throws PersistenceException;
  void delete(String name) throws PersistenceException;
}
```

PersistenceService.java

Datamodel

ws-persistence and ws-ejbservice project

The `Basket` class contains the business methods to handle the webbasket (`increment()`, `decrement()`, `getSize()`, `getTotal()`).

- ▷ Persistence layer
  - `Brand`: PHILIPS, SONY, PANASONIC
  - `Product`: **brand** (`Brand`), **name** (`String`), **price** (`int`)
- ▷ EJB Service layer
  - `BrandStub`: PHILIPS, SONY, PANASONIC
  - `ProductStub`: **brand** (`BrandStub`), **name** (`String`), **price** (`int`)
  - `BasketItem`: **product** (`ProductStub`), **quantity** (`int`)
  - `Basket`: **identifier** (`String`), **items** (`List<BasketItem>`)
  - `ServiceError`: **code** (`int`), **message** (`String`)

XML annotations

The *stubs* are decorated with the **XML Bind annotations**, because we have to serialize these instances into SOAP XML messages.
`@XmlAccessorType(XmlAccessType.FIELD)`: The annotations are defined on the fields.
`@XmlAccessorType(XmlAccessType.PROPERTY)`: The annotations are defined on the *getter* methods.

XML Binding

```
@XmlAttribute(name = "productName")
private String name;
```

The name of the product is an example for the FIELD based XML binding. The SOAP business name differs from the source code names (different naming rules). With the @XmlAttribute annotation we can create XML attribute of course, where the widespread spelling is *camelCase*. We cannot use @XmlAttribute annotation in complex types.

```
@XmlAttribute(name = "total")
public int getTotalPrice() {..}
```

We can present the total value of the basket with a **computed field** in the XML message. Here we can use only the PROPERTY based binding.

```
@XmlElement(name = "Items")
public List<BasketItem> getItems() {..}
```

In case of lists, arrays or sets we can use the @XmlElement annotation. According to the current config this name will be the group name of the items. The widespread spelling of the *elements* is the *CamelCase* writing, which differs from the rules of the Java naming.

EJB Service - Error handling

ws-ejbservice project

We are going to create **SOAP Fault custom** message (XML) from the state of the ServiceError. Of course the EJB layer will throws an exception (ServiceException) in case of error, in which we will pack an error code (WebStoreError enum: IDENTIFIER, PERSISTENCE, PRODUCT, BASKET).

Dependency between layers

Towards the public interfaces (SOAP) the EJB service layer will create stubs from the Product and Brand instances (as we have known before, with the help of the ProductConverter *Stateless Session Bean*). Notice that regardless of the persistence layer's type (currently we store the data in memory), the structure of the EJB service layer remained the same. At this time the conversion seems to us *overkill* (or the entire *persistence* layer at all), with that little technique we will able to introduce the RDBMS storage at this application without rewriting the service layer (**the storage layer does not depend on the logical layer**).

EJB Service - Services

ws-ejbservice project

Stateful Session Bean

```java
@Local
public interface WebBasketService {
  void setIdentifier(String identifier) throws ServiceException;
  String getIdentifier() throws ServiceException;
  int getBasketSize() throws ServiceException;
  void addItem(String productName) throws ServiceException;
  void removeItem(String productName) throws ServiceException;
  Basket getContent() throws ServiceException;
}
```

WebBasketService.java

Stateless Session Bean

```java
@Local
public interface StoreService {
  void add(ProductStub product) throws ServiceException;
  ProductStub get(String name) throws ServiceException;
  List<ProductStub> list(String nameTerm) throws ServiceException;
  List<ProductStub> list(BrandStub brand) throws ServiceException;
  List<ProductStub> getAll() throws ServiceException;
  void remove(String name) throws ServiceException;
}
```

StoreService.java

Stateful Session Bean

- The implementation class has the `@Stateful` annotation.
- The class may contain **instance fields**. The states of these fields are interpreted per *sessions*. If the same *session* calls an other method of the same `SFSB`, the *EJB container* will give an instance with the same state to serve the request.
- The method which has the `@PostConstruct` annotation will be executed when a new *session* calls a business method inside the bean.
- The method which has the `@Remove` annotation will be executed when the *session* becomes *invalid* or it does not reachble any more (e.g.: *timeout*).

State management

Beside the Stateful Session Bean the Singleton Session Bean is also able to store state, but the lifecycle of that is similar to the Stateless Session Bean, because here we do not need **activation** and **passivation**.

Session Bean

States and Lifecycles

- ▷ In case of Stateless and Singleton Session Beans or Message-Driven Beans
  - **Does not exist** state
  - **Ready** state (ready to execute a business method via *proxy*)
- ▷ In case of Stateful Session Beans
  - **Does not exist** state
  - **Ready** state (active, ready to execute a business method)
  - **Passive** state
    - The *container* writes the state of the bean to 'secondary storage' from memory (it would be useful later but currently the *user* does not use it).
    - `@PrePassivate` and `@PostActivate` methods can be created

`@PostConstruct` and `@PreDestroy` methods can be created anywhere

WebStoreService - Frame

ws-webservice project

WSDL: http://localhost:8080/webstore/WebStoreService?wsdl

```
package hu.qwaevisz.webstore.webservice;
[..]
@WebService(name = "WebStore", serviceName = "WebStoreService", targetNamespace =
     "http://www.qwaevisz.hu/WebStore")
@SOAPBinding(style = Style.DOCUMENT, use = Use.LITERAL, parameterStyle =
     ParameterStyle.WRAPPED)
public class WebStoreService {
  [..]
  @EJB
  private StoreService storeService;

  @WebMethod(action = "http://www.qwae
       "GetProduct")
  @WebResult(name = "Product")
  public ProductStub getProduct(@WebParam(name = "Name") String name) throws
       WebStoreServiceException {
    try {
      return this.storeService.get(name);
    } catch (ServiceException e) {
      throw new WebStoreServiceException(e.getMessage(), e.getError());
    }
  }
  [..]
}
```

We can set the namespace with the help of the **@WebService** annotation, and configure the service with the **@SOAPBinding** annotation.

WebStoreService.java

WebStoreService - GetProduct

ws-webservice project

The method which has the @WebMethod annotation will be an operation/message pair in the webservice. We can configure it via the attributes of the annotation (e.g.: it's name in the WSDL). If the name of the *operation* is GetProduct (and this will be a independent *element* in the *SOAP Request*), than the response in the *SOAP Response* will be inside a GetProductResponse element.

```
@WebMethod(action = "http://www.qwaevisz.hu/WebStore/getProduct", operationName =
     "GetProduct")
@WebResult(name = "Product")
public ProductStub getProduct(@WebParam(name = "Name") String name) throws
     WebStoreServiceException {
  [..]
}
```

The @WebResult annotation will be the *wrapper* of the return value. The ProductStub instance will be closed into a Product element in the *SOAP Response*. If we do not specify the @WebResult, the 'return' will be its default value (the @XmlRootElement annotation does not matter in that case). With the @WebParam annotation you can define the XML element names of the *request* (in the example we will need a Name element to define the name of the product).

SOAP messages - GetProduct

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:web="http://www.qwaevisz.hu/WebStore">
   <soapenv:Header/>
   <soapenv:Body>
      <web:GetProduct>
         <Name>SD85 4K HDR</Name>
      </web:GetProduct>
   </soapenv:Body>
</soapenv:Envelope>
```

SOAP Request

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
      <ns2:GetProductResponse xmlns:ns2="http://www.qwaevisz.hu/WebStore">
         <Product brand="SONY" productName="SD85 4K HDR" price="1299"/>
      </ns2:GetProductResponse>
   </soap:Body>
</soap:Envelope>
```

The brand, productName and price attribute names are come from the ProductStub's *XML Bind annotations*.

SOAP Response

Handling error cases

If the *WebMethod* throws an exception than a **SOAP Fault** will be generated. Otherwise the instance of the return value will be serialized as an XML.

WSDL - Retrieve a product

Type information

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
     xmlns:tns="http://www.qwaevisz.hu/WebStore" >
  <xs:element name="GetProduct" type="tns:GetProduct"/>
  <xs:element name="GetProductResponse" type="tns:GetProductResponse"/>
  <xs:element name="Product" type="tns:productStub"/>
  <xs:complexType name="GetProduct">
    <xs:sequence>
      <xs:element minOccurs="0" name="Name" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="GetProductResponse">
    <xs:sequence>
      <xs:element minOccurs="0" ref="tns:Product"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="productStub">
    <xs:sequence/>
    <xs:attribute name="brand" type="tns:brandStub"/>
    <xs:attribute name="productName" type="xs:string"/>
    <xs:attribute name="price" type="xs:int" use="required"/>
  </xs:complexType>
  <xs:simpleType name="brandStub">
    <xs:restriction base="xs:string">
      <xs:enumeration value="PHILIPS"/>
      <xs:enumeration value="SONY"/>
      <xs:enumeration value="PANASONIC"/>
    </xs:restriction>
  </xs:simpleType>
  [..]
</xs:schema>
```

The type information comes partly from the `http://www.w3.org/2001/XMLSchema` namespace elements (e.g.: `int`, `string`) and party from the application's own `http://www.qwaevisz.hu/WebStore` namespace.

WSDL - Retrieve a product

Define messages

```
<wsdl:message name="GetProduct">
  <wsdl:part element="tns:GetProduct" name="parameters"></wsdl:part>
</wsdl:message>
<wsdl:message name="GetProductResponse">
  <wsdl:part element="tns:GetProductResponse" name="pa
</wsdl:message>
<wsdl:message name="WebStoreServiceException">
  <wsdl:part element="tns:WebStoreServiceFault"
       name="WebStoreServiceException"></wsdl:part>
</wsdl:message>
<wsdl:portType name="WebStore">
  <wsdl:operation name="GetProduct">
    <wsdl:input message="tns:GetProduct" name="GetProduct"></wsdl:input>
    <wsdl:output message="tns:GetProductResponse"
         name="GetProductResponse"></wsdl:output>
    <wsdl:fault [..] />
  </wsdl:operation>
</wsdl:portType>
<wsdl:binding name="WebStoreServiceSoapBinding" type="tns:WebStore">
  <wsdl:operation name="GetProduct">
    <soap:operation soapAction="http://www.qwaevisz.hu/WebStore/getProduct"
         style="document"/>
    <wsdl:input name="GetProduct">
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output name="GetProductResponse">
      <soap:body use="literal"/>
    </wsdl:output>
    [..]
  </wsdl:operation>
</wsdl:binding>
```

Therse refer the type definition.

Because of the perspicuity the example does not contain the WSDL parts of the error handling.

SOAP WebService - Error handling
SOAP Fault

If an error occurs at the server side it would be elegant if we will send a **SOAP Fault** message as a response. The *SOAP Fault* marks the event (`faultcode` and `faultstring`), and it may give details optionally (`detail`). Latter could be a result of the `XML` serialization of the exception class.

```xml
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
      <soap:Fault>
         <faultcode>soap:Server</faultcode>
         <faultstring>Product does not exist in the catalog (name: lorem).</faultstring>
         <detail>
            <ns2:WebStoreServiceFault xmlns:ns2="http://www.qwaevisz.hu/WebStore">
               <code>20</code>
               <message>Product does not exist in the catalog (name: lorem).</message>
            </ns2:WebStoreServiceFault>
         </detail>
      </soap:Fault>
   </soap:Body>
</soap:Envelope>
```

ServiceError DTO

ws-ejbservice project

```java
package hu.qwaevisz.webstore.ejbservice.domain;
[..]
@XmlRootElement(name = "ServiceError")
public class ServiceError implements Serializable {

  private final int code;
  private final String message;

  public ServiceError() { this(0, null); }

  public ServiceError(final int code, final String message) { [..] }

  @XmlElement(name = "code")
  public int getCode() {
    return this.code;
  }

  @XmlElement(name = "message")
  public String getMessage() {
    return this.message;
  }

  @Override
  public String toString() {
    return "ServiceError [code=" + this.code + ", message=" + this.message + "]";
  }
}
```

JAX-B: Be default the XML serialization uses @XmlAccessorType(XmlAccessType.PROPERTY) configuration (so we put the annotation to the *getter* methods).

ServiceError.java

Custom SOAP Fault message

ws-webservice project

```java
package hu.qwaevisz.webstore.webservice.exception;
[..]
@WebFault(name = "WebStoreServiceFault", targetNamespace =
      "http://www.qwaevisz.hu/WebStore")
public class WebStoreServiceException extends Exception {
   private final ServiceError faultInfo;
   public WebStoreServiceException(String message, ServiceError
         faultInfo) {
      super(message);
      this.faultInfo = faultInfo;
   }
   public WebStoreServiceException(St
         faultInfo, Throwable cause) {
      super(message, cause);
      this.faultInfo = faultInfo;
   }
   public ServiceError getFaultInfo()
      return this.faultInfo;
   }
}
```

If the exception has a `@WebFault` annotation and the `@WebMethod` throws an instance of the that exception it will be caused a *SOAP Fault* message.

The message attribute will be the `faultstring` in the *SOAP Fault* (you can change that in your application of course).

The XML serialized return value of the `getFaultInfo()` *getter* method will be the detail content of the *SOAP Fault* (you have to use that method name).

WebStoreServiceException.java

SmartBear: SoapUI

Function and End-to-End test application/platform

- https://www.soapui.org/
- Version: **v5.4.0**
- Install: next-next-finish
- Free, open-source, but there is a commercial Pro variation as well
- Function/End-to-End testing
  - SOAP webservices
  - RESTful webservices
  - JMS support (integrated **HermesJMS** library)
- high level *security* support (encryption, digital signatures, etc.)

SmartBear: SoapUI

Create SOAP client based on the WSDL

File | New SOAP project

- ▷ Project name: **WebStore**
- ▷ Initial WSDL: `http://localhost:8080/webstore/WebStoreService?wsdl`
- ▷ Create sample requests for all operations
- ▷ Creates a TestSuite for the imported WSDL

That is all?

Yes, that is it. The `WSDL` contains all the necessary information, an fully functional client application may by generated. There is nothing special about this, we can do the same with a `CLI` command. The `WSDL` can be loaded from a file (offline generation).

WebStoreService - AddProduct

ws-webservice project

```
@WebMethod(action = "http://www.qwaevisz.hu/WebStore/addProduct",
    operationName = "AddProduct")
public void addProduct(@WebParam(name = "Product") ProductStub
    product) throws WebStoreServiceException {
  [..]
}
```

```
<web:AddProduct>
  <Product brand="PHILIPS" productName="55PFT6510/12 3D SMART
      Ambilight" price="2499"/>
</web:AddProduct>
```

SOAP Request

```
<ns2:AddProductResponse
    xmlns:ns2="http://www.qwaevisz.hu/WebStore"/>
```

SOAP Response

Online purchase process - I

SOAP Request

```
<SetBasketIdentifier>
  <Identifier>42</Identifier>
</SetBasketIdentifier>
```

```
<GetBasketIdentifier/>
```

```
<AddItemToBasket>
  <ProductName>SD85 4K
      HDR</ProductName>
</AddItemToBasket>
```

SOAP Response

```
<SetBasketIdentifierResponse />
```

```
<GetBasketIdentifierResponse>
  <Identifier>42</Identifier>
</GetBasketIdentifierResponse>
```

```
<AddItemToBasketResponse />
```

We buy two peaces from one of the TV (so we send the same message twice) and put another one into our basket as well.

Stateful service in the background → You do not need to add the `Identifier` in the later requests (this behavior is attempted to simulate this request). The container keeps the state for you per user-sessions. In case of *Stateless* service we have to store the data for example in an in-memory database, and we have to use the identifier in each request.

Online purchase process - II

SOAP Request

```
<GetBasketSize/>
```

```
<GetBasketContent/>
```

SOAP Response

```
<GetBasketSizeResponse>
  <BasketSize>2</BasketSize>
</GetBasketSizeResponse>
```

```
<GetBasketContentResponse>
  <Basket identifier="42" total="3448">
    <Item quantity="2" total="2598">
      <Product brand="SONY"
          productName="SD85 4K HDR"
          price="1299"/>
    </Item>
    <Item quantity="1" total="850">
      <Product brand="PHILIPS"
          productName="55PUH6400 UHD Smart
          Led" price="850"/>
    </Item>
  </Basket>
</GetBasketContentResponse>
```

Java WebService client application

The webservices (both *SOAP* and *RESTful*) are language independent technologies. In case of SOAP WS we only need the WSDL to create a client in any programming language. *SoapUI* is written by Java, but we only use the WSDL to generate a client application. To create a SOAP WS client application we only need to send XML documents to an *endpoint*, typically over HTTP.
But it is even easier to create a client if we get to know the `wsimport` CLI tool, which is part of the JDK.

```
> cd ws-client
> wsimport -s src/main/java -d bin -keep -p
    hu.qwaevisz.webstore.client.generated
    http://localhost:8080/webstore/WebStoreService?wsdl
```

The **ws-client** *project* will be the client application. We would like to generate the client classes of the given WSDL under the given package name (`hu.qwaevisz.webstore.client.generated`) and into the given `src/main/` *source* directory.

Client application

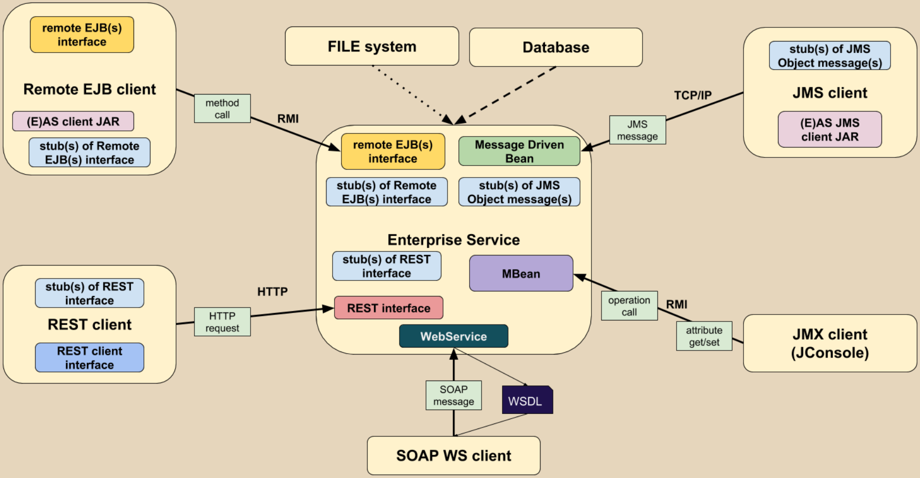
ws-client project

```
URL endpoint = new
    URL("http://localhost:8080/webstore/WebStoreService?wsdl");

WebStoreService service = new WebStoreService(endpoint);
WebStore webStore = service.getWebStorePort();

try {
  List<ProductStub> products = webStore.listAllProducts();

  for (ProductStub product : products) {
    System.out.println("Product: " + product.getBrand() + ":" +
        product.getProductName() + " ( price: " +
        product.getPrice() + " $ )");
  }



} catch (WebStoreServiceException e) {
  e.printStackTrace();
}
```

We would not need the given URL in that example because when we generatged the client classes we use the original WSDL with the right endpoint URL. But the end-point URL is ofter removed from the WSDL for security reason, so the client application has to set it as you see in the example.

Application.java

Communication types



WebLogic variant

Modifications

The application could work without major modification under *WebLogic* as well. Only the following should be considered:

- use **JDK logging** instead of **log4j**
- weblogic expects that the `@Remote` lifecycle method will be *business* method as well so we need to put that abstract method in the interface of the *Stateful Session Bean*

```
[..]
@Local
public interface WebBasketService {
  [..]

  void remove();
}
```

WebBasketService.java

PMD

Source code analyzer

- https://pmd.github.io/
- Version : **v6.1.0**
- Supported languages: Java, JavaScript, PLSQL, Apache Velocity, XML, XSL
- Documentation : https://pmd.github.io/pmd-6.1.0/
- Each **Rule** is part of a **RuleSet**.
- There are predefined *RuleSets* for Java, but we can create custom ones as well.
- *Rule* : name, description, priority, example, implementation class and optional set of properties (e.g.: `CyclomaticComplexity` *rule* has a '`reportLevel`' *Integer property*).

1. Change absolutely required. Behavior is critically broken/buggy.
2. Change highly recommended. Behavior is quite likely to be broken/buggy.
3. Change recommended. Behavior is confusing, perhaps buggy, and/or against standards/best practices.
4. Change optional. Behavior is not likely to be buggy, but more just flies in the face of standards/style/good taste.
5. Change highly optional. Nice to have, such as a consistent naming policy for package/class/fields.

PMD and Maven integration

root project

```
<project [..]>
  <properties>
    <version.plugin.pmd>3.8</version.plugin.pmd>
  </properties>
  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-pmd-plugin</artifactId>
          <version>${version.plugin.pmd}</version>
          <configuration>
            <rulesets>
              <ruleset>/rulesets/java/basic.xml</ruleset>
              <ruleset>/rulesets/java/braces.xml</ruleset>
            </rulesets>
          </configuration>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>
</project>
```

All the rule of the given ruleset must be valid for all projects. The execute the *pmd* validation run the following command: 'mvn clean package pmd:pmd'.

pom.xml

PMD rule set descriptors

pmd-bin-[version].zip | pmd-bin-[version].zip | lib | pmd-java-6.1.0.jar | rulesets | java | *.xml

```xml
<ruleset name="Basic"
    xmlns="http://pmd.sourceforge.net/ruleset/2.0.0">
  <description>The Basic ruleset..</description>
  [..]
  <rule name="JumbledIncrementer"
      language="java"
      since="1.0"
      message="Avoid modifying .."
      class="net.sourceforge.pmd.lang.rule.XPathRule"
      externalInfoUrl="https://pmd.github.io/..">
    <description>Avoid jumbled..</description>
    <priority>3</priority>
    <properties>
      <property name="xpath">
        <value><![CDATA[..]]></value>
      </property>
    </properties>
    <example><![CDATA[..]]></example>
  </rule>
</ruleset>
```

Copy these XML files into the rulesets directory of the *root* Gradle *project* (e.g.: rulesets/basic.xml).

basic.xml

PMD fine tuning

ws-ejbservice project

We can create an own *ruleset* file with ease if we pick up existing rules and rule sets and we change its properties if needed.

```xml
<?xml version="1.0"?>
<ruleset name="Custom ruleset"
    xmlns="http://pmd.sf.net/ruleset/1.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://pmd.sf.net/ruleset/1.0.0
         http://pmd.sf.net/ruleset_xml_schema.xsd"
    xsi:noNamespaceSchemaLocation="http://pmd.sf.net/ruleset_xml_schema.xsd">

  <description>Custom ruleset</description>

  <rule ref="[..]"/>
  <rule ref="[..]"/>
  <rule ref="[..]"/>
</ruleset>
```

ruleset.xml

PMD fine tuning

Define own *ruleset* based on existing ones

```xml
<rule ref="rulesets/basic.xml" />

<rule ref="rulesets/unusedcode.xml/UnusedLocalVariable" />

<rule ref="rulesets/braces.xml">
  <exclude name="ForLoopsMustUseBraces" />
</rule>

<rule ref="rulesets/basic.xml/ForLoopShouldBeWhileLoop"
  message="IMPORTANT! This for loop could be simplified to a
      while loop">
  <priority>1</priority>
</rule>

<rule ref="rulesets/codesize.xml/CyclomaticComplexity">
  <properties>
    <property name="reportLevel" value="10" />
  </properties>
</rule>
```

ws-ejbservice | config | pmd | ruleset.xml

javadoc

The `javadoc` is a built-in document creation tool of the `JDK` (`[JDK-HOME]/bin`). We have to add the location of the source code (`-sourcepath`), the package (`com.ericsson.webstore.dummy`) what we want to describe. By default it will create HTML documents via *oracle java standard doclet*).

```
> javadoc -sourcepath ws-dummy/src/main/java
    hu.qwaevisz.webstore.dummy
```

If we need outer sources/libraries to compile to the source files (e.g.: we use a custom *doclet* or any third party library) then we have to add these dependencies (`-classpath`) as well.

```
> javadoc -classpath ws-common/build/classes/main -sourcepath
    ws-dummy/src/main/java hu.qwaevisz.webstore.dummy
```

javadoc and Maven integration

create javadoc with the help of Maven

```
<project [..]>
  [..]
  <properties>
    <version.plugin.javadoc>2.10.4</version.plugin.javadoc>
  </properties>
  [..]
  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-javadoc-plugin</artifactId>
          <version>${version.plugin.javadoc}</version>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>
</project>
```

pom.xml

```
> mvn site
```

Doclet

- **Doclets** are programs written in the Java programming language that use the **doclet API** to specify the content and format of the output of the `javadoc` tool.
- By default, the `javadoc` tool uses the *'standard' doclet* provided by Sun to generate API documentation in HTML form.
- The real power of the `javadoc` is to create an own *doclet* to generate an up-to-date documentation.

The path of the doclet's source can be set with the `docletpath` argument (we may use *jar* files as well[1]), the value of the `doclet` argument will be the *full qualified name* of the doclet class.

```
> javadoc -classpath ws-common\build\classes\main -docletpath
    ws-doclet\build\classes\main -doclet
    hu.qwaevisz.webstore.doclet.WebStoreDoclet -sourcepath
    ws-dummy\src\main\java -ws-filename ws.xml
    hu.qwaevisz.webstore.dummy
```

[1] If you would like to list more than one *jar* file you have to use ";" separation character in case of *win*, and ":" character in case of *nix*.