Data Externalisation + REST/SOAP web services

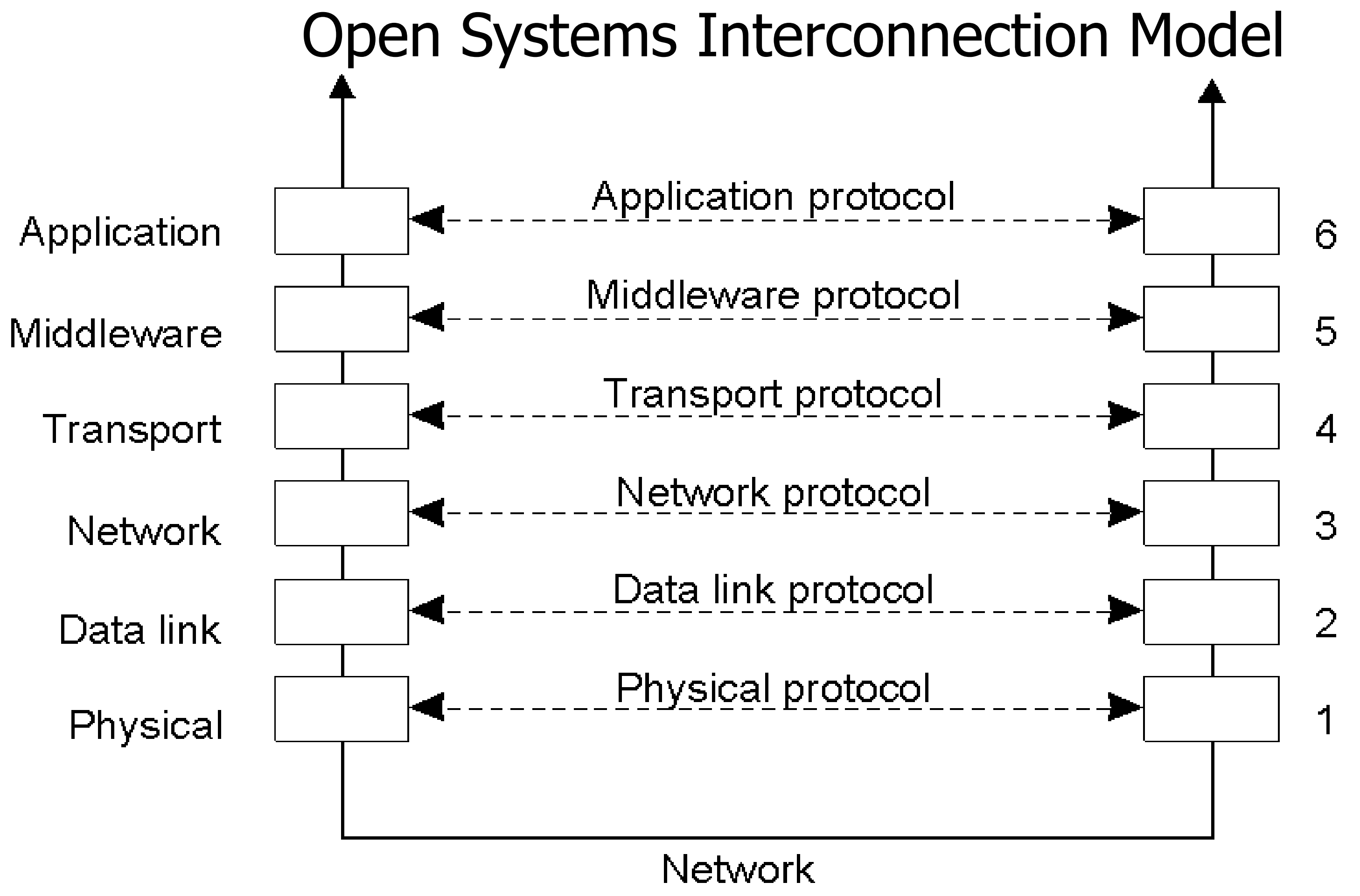
❖ ***Topics***

- Middleware
- CORBA CDR
- XML Object Binding
- JSON
- Java Serialization
- JNDI and Naming Services
- JAX-RS/Jersey
- SOAP
- Describing web services (WSDL/WADL)

Network Prog & Middleware

- A ***software layer*** that lies between the operating system and the applications on each site of the system.
- A software layer that:
  - masks the ***heterogeneity*** of systems
  - provides a ***convenient programming*** abstraction
  - provides protocols for providing general-purpose services to more specific applications, eg.
    - authentication protocols
    - authorization protocols
    - distributed commit protocols
    - distributed locking protocols
    - high-level communication protocols
      - remote procedure calls (RPC)
      - remote method invocation (RMI)
- ***Used to "glue" heterogeneous remote systems together.***

Middleware and the OSI Model



Middleware Prog Models

- **Remote Calls**
  - Remote Procedure Calls (RPC)
  - Distributed objects and Remote Method Invocation (RMI)
    - eg. Java RMI
- **Common Object Request Broker Architecture (CORBA)**
  - Cross-language Remote Method Invocation. Old and infrequently used these days.
- **Web Services**
  - XML-based service discovery, binding and invocation.
- Other programming models:
  - remote event notification.
  - remote SQL access.
  - distributed transaction processing.
- Usually provided by an ***application server*** in an n-tier client-server environment.

External Data Representation

- ***Interprocess communication*** is the passing of messages from one process to another.
  - May be on the same host, may not.
- Challenge to devise an architecture that enables applications in heterogenous environments to communicate and work together.
  - ***Homogenous***: relatively straight forward – COM/DOM on Windows, Pipes in Unix, RMI in Java.
- A platform and language neutral, vendor-agnostic framework for IPC has been the philosopher's stone of distributed computing.
- *We will look at the following standards for data representation:*
  - CORBA Common Data Representation (CDR)
  - Java Serialization
  - XML/JSON Object Binding
- These data representation formats form the basis of the technologies which we will discuss later.

External Data Representation

- ❑ Why can binary invocations work in different environments?
  - ***Byte ordering variants*** for the ordering of integers – Big-endian (MSD comes first) v Little-Endian (LSD comes first).
  - Range and size of ***primitive types*** (ints, double etc..) differ from vendor to vendor.
  - ***Character sets*** also differ – Unicode v ASCII v EBCDIC.

Irrespective of the form of communication, the data structures must be "flattened" (converted to a sequence of bytes) before transmission and rebuilt on arrival.

- ❑ ***Marshalling***: process of taking a collection of data items and assembling into a form suitable for the transmission of a message.
- ❑ ***Unmarshalling***: process of dissembling them on arrival to produce an equivalent collection of data items at the destination.

CORBA CDR

- A transfer syntax, mapping from data types defined in Open Management Group (OMG) Interface Definition Language (IDL) to a bicanonical, low-level representation for transfer between agents.
  - Can represent all of the datatypes that can be used as ***arguments and return values*** in remote invocations in CORBA.
- ***Variable Byte Ordering***
  - Machines with a common byte order may exchange messages without byte swapping.
  - For different byte orders, the message originator determines the message byte order, and the receiver swaps bytes to match its native ordering.
  - Each General Inter-Object Request Broker Protocol (GIOP) message (and CDR encapsulation) contains a flag that indicates the appropriate byte order.
- The types of the data structures and the basic data items are described in the CORBA ***Interface Definition Language (IDL).***

CORBA CDR

- The flattened form represents a Person struct with value: {'Smith', 'London', 1934}

index in sequence of bytes	← 4 bytes →	
0–3	5	length of string
4–7	"Smit"	'Smith'
8–11	"h___"	
12–15	6	length of string
16–19	"Lond"	'London'
20-23	"on__"	
24–27	1934	unsigned long

- NOTE: CORBA CDR is a very old format that is rarely used anymore. Shown for information purposes only.

XML Object (data) Binding

- ***XML databinding***: ***mapping*** an instance of an *XML Schema* into the appropriate ***object*** model (set of classes and types which represent the data). Allows applications (usually data-centric) to manipulate data that has been serialized as XML in a way that is more natural than using the Document Object Model (DOM) or the Simple API for XML (SAX).
- Why not use DOM or SAX?
  - May not want to access the structure of an XML document.
  - These APIs can be tedious when working with typed content.
- ***Databinding:***
  - Applications deal with the ***correct representation*** of the data.
  - Provides for strong ***type checking***.
- Popular XML object binding APIs include JAXB and MOXy.

XML Databinding & Marshalling

- ***Marshalling Framework***
  - Handles the conversion of Java objects to and from XML.
  - Uses descriptors to obtain information about a class and it's fields.
  - It is designed to work with any "bean-like" class (get/set methods).
  - Uses reflection to create descriptors on the fly.
  - Can use ***mapping file*** to override defaults.
- ***Marshalling API***
  - ***Marshaller*** converts objects to XML instances. Can marshal to a Writer or DocumentHandler.
  - ***Unmarshaller*** converts XML instances to objects. Can marshal from a Reader, or EventProducer. Can return a DocumentHandler.

Object/Databinding Frameworks

- ***Castor:*** An open source data-binding framework for Java.
  - ***Greek Mythology***: Castor and Pollux, the twin sons of Leda and brothers of Helen and Clytemnestra, who were transformed by Zeus into the constellation Gemini.
  - Can map ***XML to Java/RDBs (via JDO) and vice-versa***.
  - Can marshall/unmarshall XML to Java.
  - Java to SQL binding also.
  - https://castor-data-binding.github.io/castor/
- ***JAXB:*** Sun Microsystem's Java Architecture for XML Binding
  - Provides a ***binding compiler*** and a runtime framework mapping.
- ***EclipseLink MOXy:*** Comprehensive open-source Java persistence solution addressing relational, XML, JSON, and database web services.
  - https://www.eclipse.org/eclipselink/

Simple Example

```
package nuig;
public class Person implements java.io.Serializable {
    private String name = null;
    private java.util.Date dob = null;
    public Person(){ super();}
    public Person(String name){
        this.name  = name;
    }
    public Date getDateOfBirth(){ return dob; }
    public String getName() { return name; }
    public void setDateOfBirth(Date dob) {
        this.dob = dob;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

Simple Example Using JAXB

To marshal the Person object to an xml file:

```
Person person = new Person("Michael Murphy");
person.setDateOfBirth(new Date(1970, 7, 7));
JAXBContext jaxbContext     = JAXBContext.newInstance( Person.class );
Marshaller jaxbMarshaller   = jaxbContext.createMarshaller();
OutputStream os = new FileOutputStream( “person.xml" );
jaxbMarshaller.marshal( person, os );
```

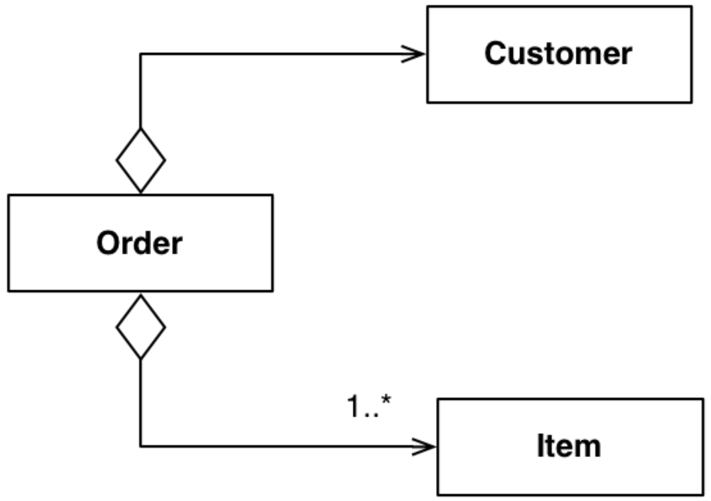
To unmarshal an XML file at a given URL to a Java object:

```
URL url = new URL( "http://localhost:8080/person.xml" );
Person person = (Person) jaxbUnmarshaller.unmarshal( url );
```

XML Representation

- *Consider the following example of representing object state in XML:*

```
<order number="12566">
    <customer code="1">
        <name>Patrick Sarsfield</name>
        <dob>7/7/1660</dob>
        <address1>Ballyneety</address1>
        <address2>Pallasgreen</address2>
        <county>Limerick</county>
        <phone>087-12345678</phone>
    </customer>
    <details>
        <item partNumber="QB-111-AA" partName="Match Lock Musket" qty="1"
            price="99.99"/>
        <item partNumber="QB-233-AB" partName="Pike" qty="2" price="4.00"/>
        <item partNumber="QB-134-CC" partName="Bayonet" qty="2" price="35.00"/>
        <item partNumber="QB-454-ZA" partName="Sword" qty="1" price="350.00"/>
    </details>
</order>
```



JSON

- *JSON (JavaScript Object Notation) is an open standard format for transmission of human-readable key/value pairs.*
  - Douglas Crockford (2001) and IETF RFC 7158.
  - Arose from requirement for stateful, real-time web browser / server communication, independent of plugins.
- ***JSON basic types:***
  - ***Number:*** signed decimal number that may contain a fractional part and may use exponential E notation.
  - ***String:*** sequence of zero or more Unicode characters
  - ***Boolean:*** true / false
  - ***Array:*** ordered list of zero or more values. Uses square bracket notation with elements being comma-separated.
  - ***Object:*** unordered associative array (key/value pairs). Delimited with curly brackets. Uses commas to separate each pair. Keys are distinct strings.

JSON Representation of State

```
{
 "order": {
  "-number": "12566",
  "customer": {
   "-code": "1",
   "name": "Patrick Sarsfield",
   "dob": "7/7/1660",
   "address1": "Ballyneety",
   "address2": "Pallasgreen",
   "county": "Limerick",
   "phone": "087-12345678"
  },
  "details": {
   "item": [
    {
     "-partNumber": "QB-111-AA",
     "-partName": "Match Lock Musket",
     "-qty": "1",
     "-price": "99.99"
    },
    {
     "-partNumber": "QB-233-AB",
     "-partName": "Pike",
     "-qty": "2",
     "-price": "4.00"
    },
    {
     "-partNumber": "QB-134-CC",
     "-partName": "Bayonet",
     "-qty": "2",
     "-price": "35.00"
    },
    {
     "-partNumber": "QB-454-ZA",
     "-partName": "Sword",
     "-qty": "1",
     "-price": "350.00"
    }
   ]
  }
 }
}
```

Java Serialization

- The method of externalisation used in the Java language.
  - Powerful and flexible.
- ***Serialization***: saving the current state of an object to a stream.
- ***Deserialization***: Restoring an equivalent object from that stream.
- Stream functions as a container for the object.
  - Includes a partial representation of the object's internal structure, including variable types, names, and values.
- May be transient (RAM-based) or persistent (disk-based).
  - ***Transient*** container may be used to prepare an object for transmission from one computer to another (e.g. over a socket).
  - ***Persistent*** container, such as a file on disk, allows storage of the object after the current session is finished.

Serializable and Externalizable

- Objects must be an instance of a class that implements either the ***Serializable*** or ***Externalizable*** interface (java.io lib).
  - Can only save data associated with an object's variables.
  - Depend on the class definition being available to the JVM at reconstruction time.
- Serializable interface relies on the Java runtime default mechanism to save an object's state.
  - ***writeObject()*** used to serialize an object (ObjectOutputStream class & ObjectOutput interface).
  - Use ***write<datatype>()*** method to write a primitive value.
  - ***readObject()*** used to deserilaize (ObjectInputStream class)
  - Use ***read<datatype>()*** method to read primitives.
- ***Any object references are also written to the stream*** (otherwise lots of null pointers and exceptions…)

Serializable and Externalizable

- ***Externalizable*** interface specifies that the implementing class will handle the serialization on its own.
  - Doesn't rely on the default runtime mechanism.
- Includes which fields get written (and read), and in what order.
- Class must define a ***writeExternal()*** method to write out the stream, and a corresponding ***readExternal()*** method to read the stream.
  - Inside of these methods the class calls ***ObjectOutputStream writeObject()***, ***ObjectInputStream readObject()***, and any necessary ***write<datatype>()*** and ***read<datatype>()*** methods, for the desired fields.
- ***Note:*** *writeExternal() and readExternal()* ***must be declared public***.
  - Increases risk that a rogue object could use them to determine the format of the serialized object.

Serialization - Hiding Data

- Might want to ***prevent certain fields from being stored*** in the serialized object.
  - Serialization allows us to specify that some of fields do not get saved or restored.
  - Done by placing the keyword ***transient*** before the data type in the variable declaration.
- ***Note: static fields are not serialized (written out), and so cannot be deserialized (read back in) – why?***
- Can also override *writeObject()* of *Serializable* to hide data.
  - Will also need to override readObject(). Similar to *Externalizable*.

Serialization - Versioning

- What happens when an object format has been superseded by a new, different version of the class?
  - Need to check during deserialization for backward compatibility.
- Changes to classes specified using a version number.
  - A specific class variable, serialVersionUID (representing the Stream Unique Identifier, or SUID) used to specify the earliest version of the class that can be deserialized.
- The SUID is declared as follows:

```
static final long serialVersionUID = 2L;
```

- ***SUID is a measure of backward compatibility.***
  - Same SUID can be used for multiple representations of a class, as long as newer versions can still read the older versions (interfaces).
- JVM ***automatically assigns a default SUID*** (unless done explicitly). *Secure Hash Algorithm (SHA).*
  - Consists of a hash of class name, interfaces, methods, and fields.

Serialization - Versioning

- Can check the SUID of a class at runtime by querying the JVM:

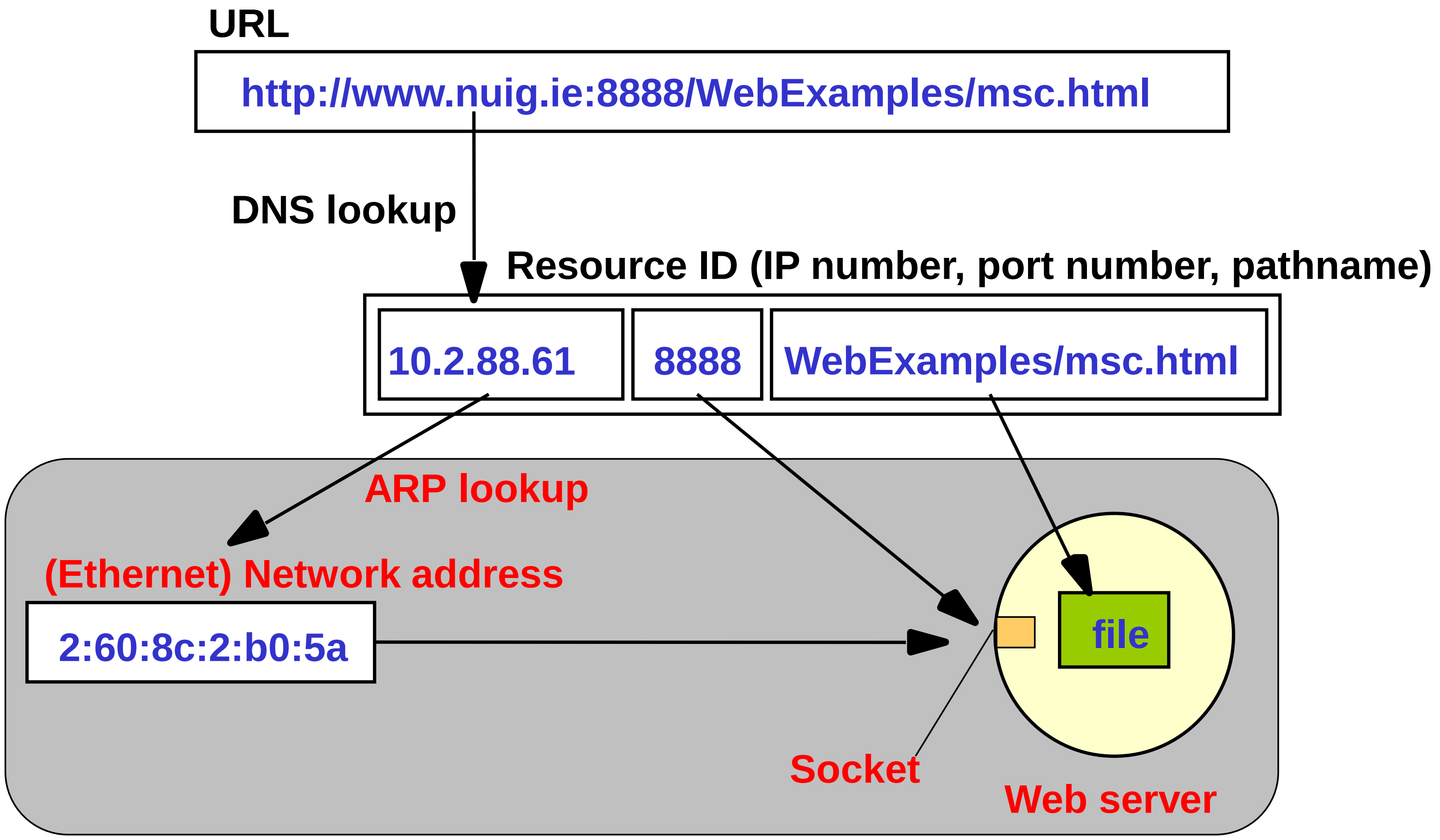
```
ObjectStreamClass myObject = ObjectStreamClass.lookup(
Class.forName( "MyClass" ) );
long theSUID = myObject.getSerialVersionUID();
```

- Can compare the SUID of a restored *Externalizable* object to the class SUID just obtained.
  - If mismatch occurs, we should take appropriate action.
- Java runtime will check the SUID for a *Serializable* object – if we override *readObject()* we'll have to implement this ourselves.
- Problems with forward compatibility – older object with fewer fields. Converse is not the case – default values can be added:
  - ***Deleting a field***, or changing it from non-static or non-transient to static or transient.
  - Changing the position of classes in a ***hierarchy***.
  - Changing the ***data type*** of a primitive field.
  - Changing the ***interface*** for a class from Serializable to Externalizable (or vice-versa).

Naming Services

- Fundamental facility in any computer system.
  - Maps names to objects and object to names.
  - File, email and DNS systems.
- We're primarily interested in mapping names to remote object references – principle remains the same. A naming system provides a ***naming service*** for naming-related operations.
  - Naming service is accessed through it's own interface.
  - ***DNS*** offers a name service that maps machine names to IP addresses.
  - ***Lightweight Directory Access Protocol (LDAP)*** offers a name service that maps LDAP names to entries.
- To look up an object in a naming system, you supply the ***name*** for that object.
  - Naming convention determines the syntax for valid name.
  - E.g. ***cn=Patrick Mannion, o=NUIG, c=Ireland***

Domain Name Service


URL
http://www.nuig.ie:8888/WebExamples/msc.html
DNS lookup
Resource ID (IP number, port number, pathname)
10.2.88.61
8888
WebExamples/msc.html
ARP lookup
(Ethernet) Network address
2:60:8c:2:b0:5a
file
Socket
Web server

Naming Services

- The association of a ***name*** with an object is called a ***binding***.
  - E.g. a file name is bound to a file, a machine name to an IP address.
- Naming service may not store object directly – must be ***stored by reference***.
  - A ***pointer or reference*** to the object is placed inside the naming service, e.g. a file handle.
- A ***context*** is a set of name to object bindings.
  - Every context has an ***associated naming convention***.
  - Context ***provides a lookup*** (resolution) operation that returns the object.
  - May ***provide operations*** for binding names, unbinding and listing bound names.
  - A name in one context object ***can be bound to another context*** object (a subcontext) that has the same naming convention.
  - e.g. a unix subdirectory named relative to another directory.

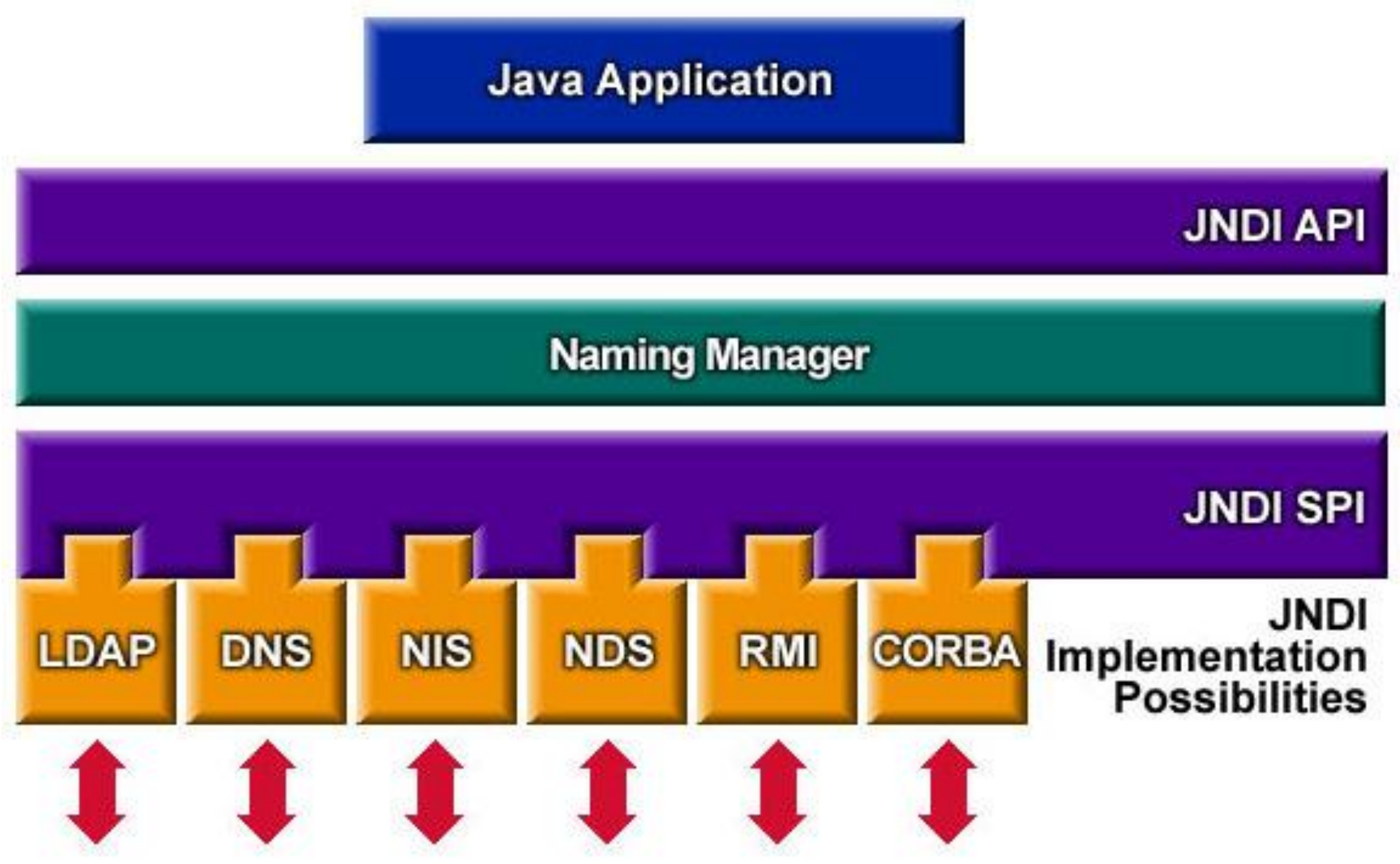
Naming Services

- A naming system therefore can be thought of as ***connected set of contexts of the same type***.
  - Same naming convention and common set of operations.
- A ***namespace*** is the set of names in a naming system.
  - May vary from DNS domains to 128-bit object GUIDs.
- Naming services can be extended with a ***directory service***.
  - Associates names with objects but also ***allows objects to have attributes***. (can search for objects based on attributes).
  - Attributes consist of an ***attribute identifier and a set of attribute*** values (name/value pairs).
- A directory is a connected set of directory objects.
  - A directory service provides operations for creating, adding, removing and modifying the attributes of directory objects.
- Examples of directory services include the Novell Directory Service (NDS) and Sun's Network Information Service (NIS).

Java Naming & Dir Interface

- JNDI is an API that provides ***naming and directory*** functionality to applications written in Java.
  - Designed to be ***independent*** of any specific directory service implementation.
  - Can access a variety of directory services in a common way.
- Architecture consists of an ***API and Service Provider Interface*** (SPI).
  - SPI enables different naming & directory services to be plugged in transparently.
- Available with the <u>***1.3 SDK***</u> and later releases.
- <u>Includes service providers for</u>:
  - LDAP, CORBA Common Object Services (COS) Name Service and Java Remote Method Invocation RMI Registry.
  - Main ***packages*** are javax.naming, javax.naming.directory, javax.naming.event, javax.naming.ldap and javax.naming.spi.

JNDI Hierarchy



JNDI Naming Package

- Contains classes and interfaces for accessing naming services.
- ***Context***: core interface for looking up, binding/unbinding, renaming objects and creating and destroying *subcontexts*.
  - Most commonly used operation is ***lookup(java.lang.String)***.
  - Returns the object bound to the lookup name.

    ```
    Printer printer = (Printer) ctx.lookup("HP-2400");
    printer.print(report);
    ```
- ***Binding***: a tuple containing the name of the bound object, the name of the object's class, and the object itself.
  - ***listBindings()*** returns an enumeration of name-to-object bindings.
- ***InitialContext***: provides a starting point for naming and directory operations.
  - All naming and directory operations are performed *relative to a context*. No absolute roots.

JNDI Directory Package

- Allows applications to retrieve attributes associated with objects stored in the directory and to search for objects using specified attributes.
- ***DirContext***: represents a *directory context*.
  - Defines methods for examining and updating attributes associated with a directory object.
  - ***getAttributes(<name>)***: retrieves the attributes associated with a directory object (for which you supply the name).
  - ***modifyAttributes(<name>, <modifications>)***: Add, replace, or remove attributes and/or attribute values.
  - ***search(<name>, <matching attributes>)***: Searches in a single context for objects that contain a specified set of attributes.

Static / Dynamic Object Binding

- Usually we know what objects we want to use at design time (***static binding***).
  - E.g. Person p = new Person();
- ***Dynamic binding***: Code executed to perform a given operation is determined at run time.
  - An expression may denote an object which may have ***more than one possible class*** and that class can only be determined at run time.
  - Binding done "on the fly". Allows us to build powerful applications.
- ***Implemented using interfaces.***
  - Allows us to harness the power of ***polymorphism*** in an OO environment. Even distributed polymorphism!
  - ***Interfaces are the key*** to understanding how may object binding technologies work – Java Interfaces, CORBA IDL, COM ODL and WSDL (web services).

RESTful Web Services

- ❑ *Representation State Transfer (REST) defines a set of architectural principles for exploiting system resources.*
  - Focuses on how resources are ***addressed*** and ***accessed*** over HTTP by a wide range of heterogeneous clients.
  - Simpler architectural model supplanting SOAP/WSDL.
  - Based on PhD thesis of Mark Fielding (2000) at University of California, Irvine. *Architectural Styles and the Design of Network-based Software Architectures*.
- ❑ Most important concept in REST are ***resources***. Identified by a global ID, typically a URI.
  - RESTful client applications use HTTP GET, POST, PUT & DELETE methods to manipulate a resource or a collection of resources.
  - Different MIME (Multipurpose Internet Mail Extensions) types for response return data, including XML, JSON and ATOM.
- ❑ Apache Jersey JSR 311/JAX-RS implementation available for Java and Tomcat.
  - Uses annotations to make development and deployment easier.

REST

- "*Representational State Transfer is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through an application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use.*" **– Roy Fielding**

REST vs. SOAP/RPC

- REST – is an architectural style, not a standard (in contrast to SOAP and RPC, which are protocols for inter-process communication)
- In SOAP/RPC systems, the design emphasis is on **verbs**. What operations can be invoked on the system? E.g. getUser(), addUser(), deleteUser(), getLocation(), updateLocation()
- In RESTful systems, the design emphasis is on **nouns**, i.e. **resources,** what their representation is and how they are affected by the standard set of HTTP methods e.g. User resource, Location resource
- Could develop an XML representation for User and Location and then clients and servers would operate on these documents to exchange information via the standard set of HTTP methods.

REST vs. SOAP/RPC

- REST does not dictate the message format; SOAP/RPC must use XML.
- Therefore RESTful services may be used to service requests from a diverse range of clients without using any specific libraries on the client side. SOAP/RPC require libraries so that method calls may be understood and forwarded to the correct object.
- RESTful web services can service both web browsers and custom clients written in any arbitrary programming language.
- E.g. Consider a request for a list of customers. A RESTful service could return the customer details in marshalled into XML/JSON format if the caller is a custom application, or format the details in a HTML table if the caller is a web browser (more on this later!)

RESTful Principles

- *REST is based on principles that encourage RESTful applications to be simple, lightweight, and fast:*
  - ***Resource Identification through URI:*** every resource is given a unique identifier. Provides a global addressing space for resource and service discovery.
  - ***Uniform Interface:*** reuse exisiting HTTP methods (GET, POST, PUT, DELETE). Simple and ubiquitous.
  - ***Self-descriptive Messages:*** Decouple resources from their representation. Provide access in variety of formats (HTML, XML, plain text, PDF, JPEG, JSON).
  - ***Stateful Interactions through Hyperlinks:*** All interactions with a resource are *stateless* (request messages are self-contained).
    - *Stateful interactions use explicit state transfer* techniques (URI rewriting, cookies, and hidden form fields).
    - State can also be *embedded in response messages* to point to valid future states of the interaction.
- Stateless interaction allows greater ***scalability*** of an application.

URI vs URL

- A Uniform Resource Identifier (URI) in a RESTful web service is a link to a resource e.g. randomsite.com/orderinfo
- A Uniform Resource Locator (URL) is a specific form of URI that defines how a specific resource may be accessed. Unlike a URI, the URL defines how the resource can be obtained e.g. http://randomsite.com/orderinfo
- Note that all URLs are also URIs, but that not all URIs are URLs! (analogy: birds can fly, but not everything that can fly is a bird)

Why REST?

- Less overhead (no SOAP envelope to wrap every call in)
- Less duplication (HTTP already represents operations like DELETE, PUT, GET, etc. that have to otherwise be represented in a SOAP envelope).
- More standardized - HTTP operations are well understood and operate consistently. Some SOAP implementations can get finicky (as we saw!).
- More human readable and easier to test (harder to test SOAP with just a browser).
- Don't need to use XML (but can use it if convenient).
- SOAP can go over other transports besides HTTP, so as to avoid riding atop a layer doing similar things, but in reality most SOAP messages over HTTP. Can just use REST and HTTP methods instead.

Why REST?

- If you have a URI that points to a service, you know exactly which methods are available on that resource.
- You don’t need an IDL-like file describing which methods are available.
- You don’t need stubs or interfaces.
- All you need is an HTTP client library.
- If you have a document that is composed of links to data provided by many different services, you already know which method to call to pull in data from those links.
- If your web service is exposed over HTTP, there is a very high probability that people who want to use your service will be able to do so without any additional requirements beyond being able to exchange the data formats the service is expecting.
- CORBA or SOAP require vendor-specific client libraries as well as lots of IDL- or WSDL-generated stub code.

Why REST?

- **Scalability**:
  - When surfing the Internet, the second time you browse to a specific page it may load more quickly. This is because browsers can cache already visited pages and images.
  - HTTP has a rich and configurable protocol for defining caching semantics.
  - Because GET is a read method that is both idempotent (the result is always the same) and safe, browsers and HTTP proxies can cache responses to servers, and this can save a huge amount of network traffic to a web service.
  - By adding HTTP caching semantics to web services, caching policies for clients may be defined (using the Cache-Control: response header). This includes e.g. how long a given cached response is valid for

REST & HTTP Methods

- REST requires developers to use HTTP methods only, in a way that is consistent with their definition. This basic REST design principle gives a one-to-one mapping between CRUD (create, read, update, delete) operations and HTTP methods:
  - Use POST to create a resource on the server
  - Use GET to retrieve an existing resource on the server
  - Use PUT to change the state of a resource/update a resource
  - Use DELETE to remove or delete a resource

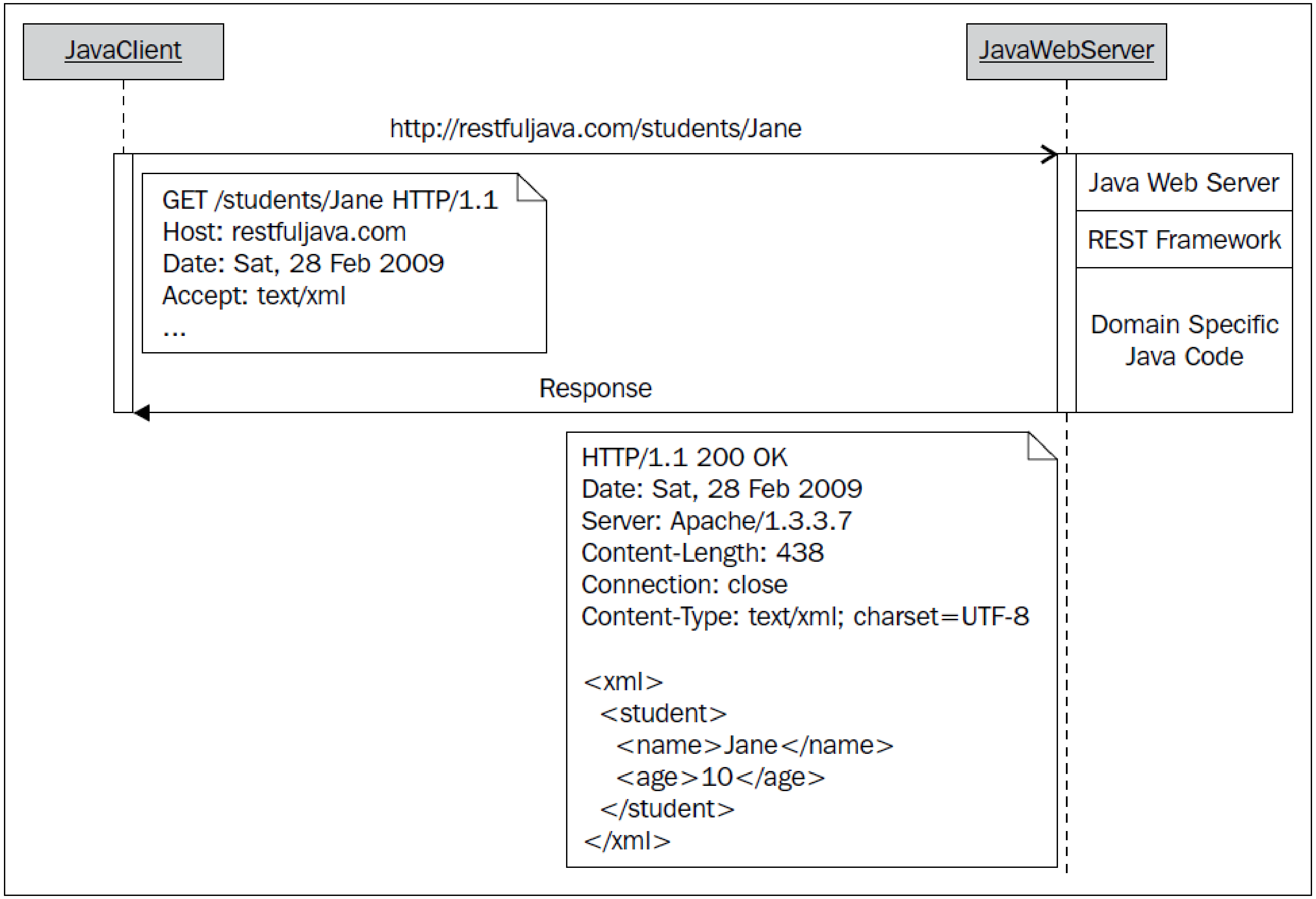
REST & HTTP Methods

- Consider the following XML representation of a student:

```
<student>
    <name>Jane</name>
    <age>10</age>
    <link>/students/Jane</link>
</student>
```

- We want to use the usual HTTP methods to perform CRUD operations on students using this XML format
- RESTful service will be available at http://www.examplesite.com/students

GET Example

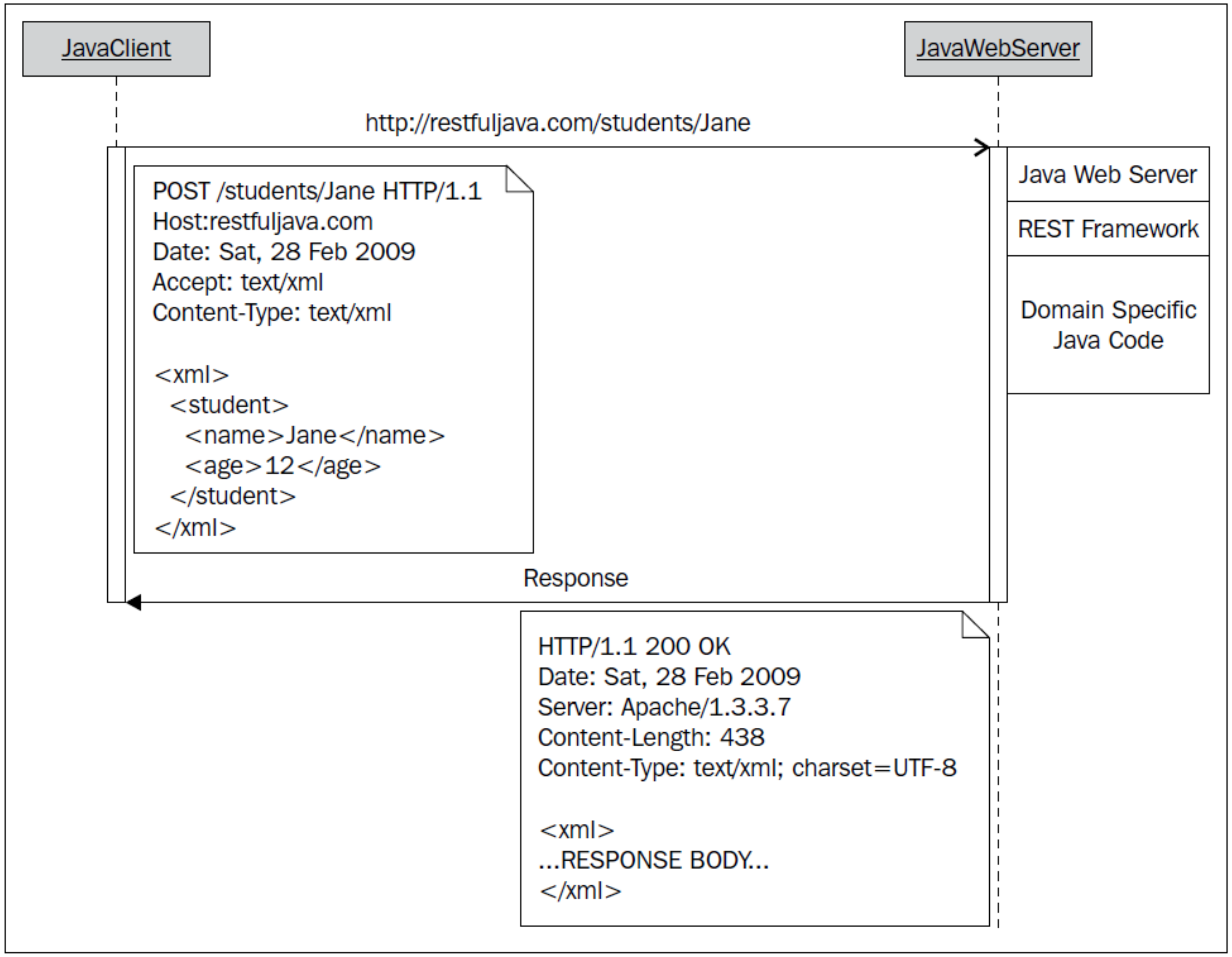

JavaClient
JavaWebServer
http://restfuljava.com/students/Jane
Java Web Server
REST Framework
Domain Specific Java Code
Response

```
GET /students/Jane HTTP/1.1
Host: restfuljava.com
Date: Sat, 28 Feb 2009
Accept: text/xml
...
```

```
HTTP/1.1 200 OK
Date: Sat, 28 Feb 2009
Server: Apache/1.3.3.7
Content-Length: 438
Connection: close
Content-Type: text/xml; charset=UTF-8

<xml>
  <student>
    <name>Jane</name>
    <age>10</age>
  </student>
</xml>
```

POST Example


JavaClient
JavaWebServer
http://restfuljava.com/students/Jane
Java Web Server
REST Framework
Domain Specific Java Code
Response

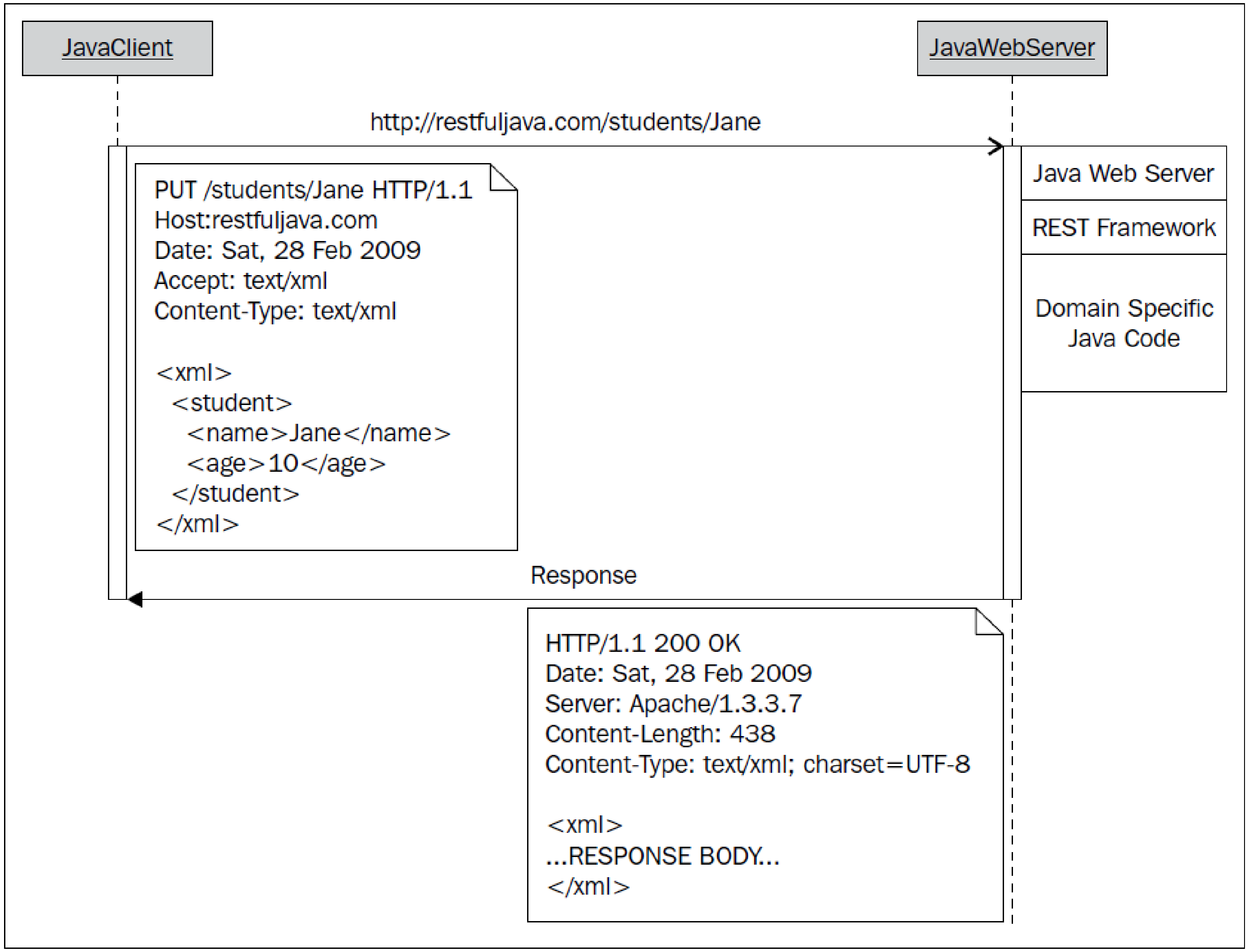
```
POST /students/Jane HTTP/1.1
Host:restfuljava.com
Date: Sat, 28 Feb 2009
Accept: text/xml
Content-Type: text/xml

<xml>
  <student>
    <name>Jane</name>
    <age>12</age>
  </student>
</xml>
```

```
HTTP/1.1 200 OK
Date: Sat, 28 Feb 2009
Server: Apache/1.3.3.7
Content-Length: 438
Content-Type: text/xml; charset=UTF-8

<xml>
...RESPONSE BODY...
</xml>
```

PUT Example


JavaClient
JavaWebServer
http://restfuljava.com/students/Jane
Java Web Server
REST Framework
Domain Specific Java Code
Response

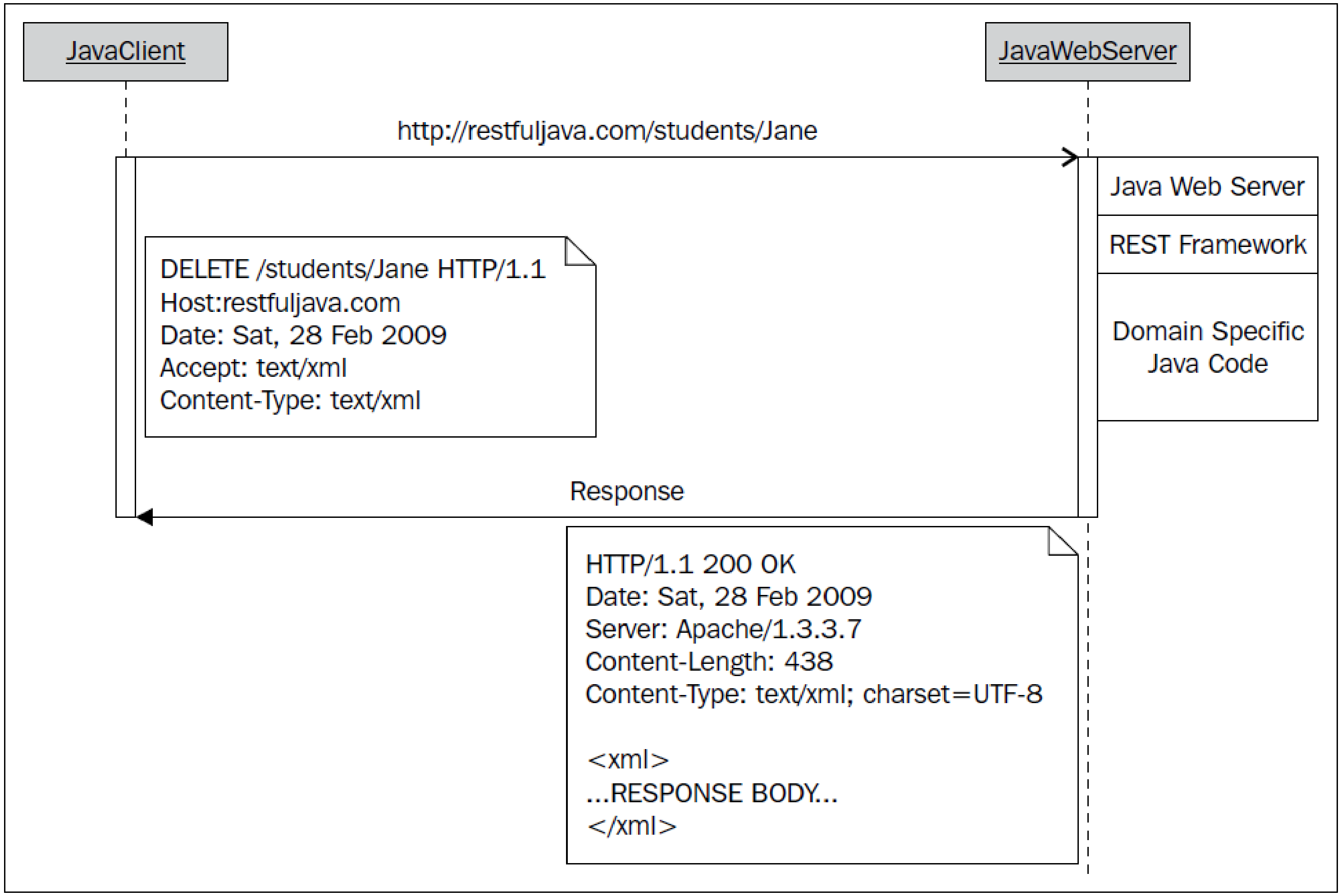
```
PUT /students/Jane HTTP/1.1
Host:restfuljava.com
Date: Sat, 28 Feb 2009
Accept: text/xml
Content-Type: text/xml

<xml>
  <student>
    <name>Jane</name>
    <age>10</age>
  </student>
</xml>
```

```
HTTP/1.1 200 OK
Date: Sat, 28 Feb 2009
Server: Apache/1.3.3.7
Content-Length: 438
Content-Type: text/xml; charset=UTF-8

<xml>
...RESPONSE BODY...
</xml>
```

DELETE Example


JavaClient
JavaWebServer
http://restfuljava.com/students/Jane
Java Web Server
REST Framework
Domain Specific Java Code
Response

```
DELETE /students/Jane HTTP/1.1
Host:restfuljava.com
Date: Sat, 28 Feb 2009
Accept: text/xml
Content-Type: text/xml
```

```
HTTP/1.1 200 OK
Date: Sat, 28 Feb 2009
Server: Apache/1.3.3.7
Content-Length: 438
Content-Type: text/xml; charset=UTF-8

<xml>
...RESPONSE BODY...
</xml>
```

JAX-RS

- JAX-RS: Java API for RESTful Web Services (JAX-RS) is a Java programming language API specification which allows web services to be developed according to the Representational State Transfer (REST) architectural pattern.
- Official part of Java EE 6 since JAX-RS v1.1
- https://github.com/eclipse-ee4j/jaxrs-api
- JAX-RS makes heavy use of annotations (syntactic metadata that can be added to Java source code). Annotations have no direct effect on the operation of the code they annotate, but contain compile-time, deployment-time and run-time information.
- "*Oracle sees REST as key to modernizing future versions of Java EE for microservices and the cloud*" https://www.infoworld.com/article/3163136/java/oracle-bets-java-ee-future-on-rest-apis.html

JAX-RS Implementations

- **Jersey:** RESTful Web Services framework is an open source, framework for developing RESTful Web Services in Java, that provides support for JAX-RS APIs and serves as a JAX-RS (JSR 311 & JSR 339) Reference Implementation.
  - https://eclipse-ee4j.github.io/jersey/
- **Apache CXF**
- **RESTEasy**
- **Restlet**
- **IBM JAX-RS**
- Etc..

JAX-RS Annotations

- *Java API for RESTful Web Services (JAX-RS) provides annotations to facilitate the mapping of a POJO as a RESTful web resource.*

Annotation	Description
@Path	Specifies the relative path for a resource class or method.
@GET etc..	@GET, @PUT, @POST, @DELETE and @HEAD specify the HTTP request type of a resource.
@Produces	Specifies the response Internet MIME types, i.e. return types.
@Consumes	specifies the accepted request MIME types, i.e. parameters.

  - @GET used to access a resource, @POST to add a new resource, @PUT to update a resource and @DELETE to remove a resource.
- REST uses basic HTTP methods in the manner in which they were originally intended to be used!
  - Shouldn't use a GET request with a query string to update a resource (done frequently in web applications).

JAX-RS Annotations

- *JAX-RS specifies extra annotations for method parameters.*
  - All @*Param annotations take a key of some form which is used to look up the value required.

Annotation	Description
@PathParam	Binds a method parameter to a path segment.
@QueryParam	Binds a method parameter to the value of an HTTP query parameter.
@MatrixParam	Binds a method parameter to the value of an HTTP matrix parameter.
@HeaderParam	Binds a method parameter to an HTTP header value.
@CookieParam	Binds a method parameter to a cookie value.
@FormParam	Binds a method parameter to a form value.
@DefaultValue	Default value for the above bindings when the key is not found.
@Context	Returns the entire context of the object.(for example @Context HttpServletRequest request)

Jersey HelloWorld

- URL http://localhost:8080/REST_Lab/webapi/hello

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/hello")
public class HelloResource {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String sayHello() {
        return "Hello World!";
    }
}
```

Jersey Hello {name}

- URL http://localhost:8080/REST_Lab/webapi/hello/name/Patrick

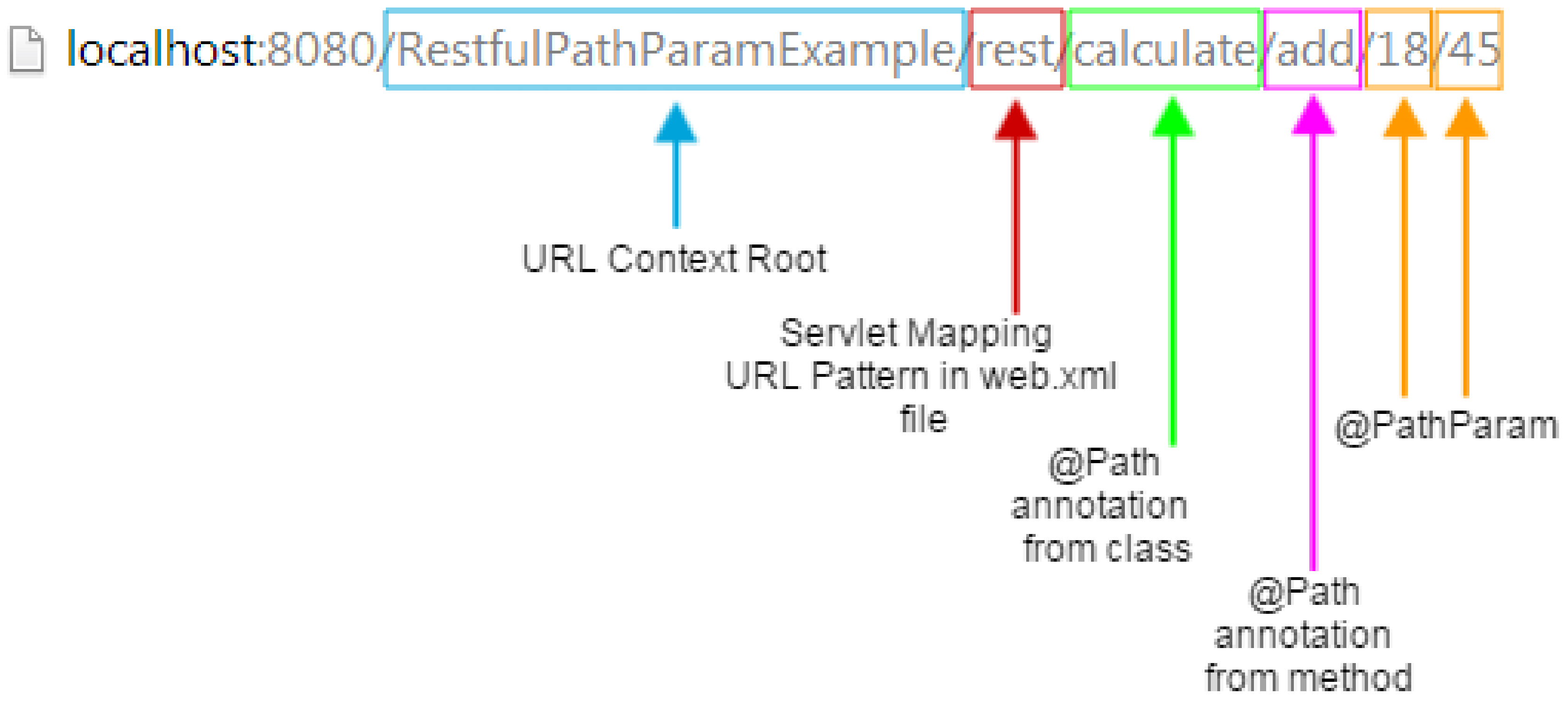
```
@Path("/hello")
public class HelloResource {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String sayHello() {
        return "Hello World!";
    }

    @GET
    @Path("/name/{value}")
    @Produces(MediaType.TEXT_PLAIN)
    public String sayHello(@PathParam("value") String value) {

        return "Hello " + value;
    }
}
```

JAX-RS @PathParam



localhost:8080/RestfulPathParamExample/rest/calculate/add/18/45
URL Context Root
Servlet Mapping URL Pattern in web.xml file
@Path annotation from class
@Path annotation from method
@PathParam

Sample web.xml file

```
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
    <servlet>
        <servlet-name>Jersey Web Application</servlet-name>
        <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
        <init-param>
            <param-name>jersey.config.server.provider.packages</param-name>
            <param-value>ie.nuig.sw.REST_Lab</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Jersey Web Application</servlet-name>
        <url-pattern>/webapi/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Preserving Instances

- By default, Jersey will create a new instance of a resource class to handle each new request
- The @Singleton annotation may be used to ensure that the same instance of the resource class is maintained at all times

```
@Path("/hellosingleton")
@Singleton
public class HelloSingleton {

    int timesCalled = 0;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String sayHelloPlain() {
        timesCalled++;
        return "Hello World number: " + timesCalled;
    }
}
```

JAX-RS Response MIME Types

- The @Produces() method annotation allows the type of response content for a method to be specified
- e.g. @Produces(MediaType.TEXT_PLAIN) which we have seen so far
- Can also specify the MIME type directly e.g. @Produces("text/plain")
  - Error prone as the MIME type must be input manually
  - Better solution is to use javax.ws.rs.core.MediaType
- More on MIME types: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types

javax.ws.rs.core.MediaType

- The class javax.ws.rs.core.MediaType provides abstractions for different MIME types
- Advantage – compiler can check if the MediaType specified is valid
- MediaTypes which are available include:
  - MediaType.TEXT_PLAIN
  - MediaType.TEXT_HTML
  - MediaType. APPLICATION_XML
  - MediaType. APPLICATION_JSON
  - MediaType. APPLICATION_XHTML_XML
  - MediaType. APPLICATION_SVG_XML
  - etc...
- Full listing of MediaTypes is available at: https://javaee.github.io/javaee-spec/javadocs/javax/ws/rs/core/MediaType.html

Response based on Accept Type

- Recall the example from earlier: You wish to retrieve a list of customer details from a web service.
  - With SOAP/RPC/RMI the response must be of the appropriate message type and (potentially) interpreted by a library on the client side
  - A RESTful service could return the customer details in XML/JSON format if the caller is a custom application, or format the details in a HTML table if the caller is a web browser
- How can this be achieved? Set the correct value in the HTTP Accept: header field. Browsers are (usually) set up to request HTML (i.e. Accept: text/html). For a custom application we could request some other format (e.g. Accept: application/xml)
- Note that if your service does not support any of the requested formats (i.e. does not have a method with the correct @Produces annotation) Tomcat will return a HTTP 406 Status – Not Acceptable e.g. "The resource identified by this request is only capable of generating responses with characteristics not acceptable according to the request "accept" headers."

HTTP MIME Accept Types

- Example MIME Accept types:
  - text/html
  - text/plain
  - application/xml
  - application/json
  - image/gif
  - image/jpeg
  - etc...

- Listing of default Accept field values for common browsers: https://developer.mozilla.org/en-US/docs/Web/HTTP/Content_negotiation/List_of_default_Accept_values

Response based on Accept Type

```
@Path("/helloaccepttypes")
public class HelloAcceptTypes {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String sayHelloPlain() {
        return "Hello World!";
    }

    @GET
    @Produces(MediaType.TEXT_HTML)
    public String sayHelloHTML() {
        return "<html><body><h1>Hello World!</h1></body></html>";
    }
}
```

The example above shows a RESTful Hello World service, which produces a different response type depending on which Accept: type is specified by the client

JAX-RS and Java Interfaces

- As well as applying JAX-RS annotations directly on the Java class that implements a service, JAX-RS also allows you to define a Java interface that contains all your JAX-RS annotation metadata, instead of applying all your annotations to your implementation class.
- As we saw with RMI, interfaces are a great way to scope out how your services are accesed. With an interface, you can write something that defines what your RESTful API will look like along with what Java methods they will map to without writing any business logic
- This approach also stops business logic from being “polluted” with numerous annotations – greater code readability.
- Interface-based approach also convenient if the same business logic must be accessible via e.g. SOAP

JAX-RS and Java Interfaces

```
public interface HelloInterface {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String sayHello();

    @GET
    @Path("/name/{value}")
    @Produces(MediaType.TEXT_PLAIN)
    public String sayHello(@PathParam("value") String value);

}
@Path("/hello")
public class HelloResource implements HelloInterface {

    @Override
    public String sayHello() {
        return "Hello World!";
    }
    @Override
    public String sayHello(String value) {
        return "Hello " + value;
    }
}
```

JAX-RS and Java Interfaces

```
import ...;

@Path("/customers")
public interface CustomerResource {

   @POST
   @Consumes("application/xml")
   public Response createCustomer(InputStream is);

   @GET
   @Path("{id}")
   @Produces("application/xml")
   public StreamingOutput getCustomer(@PathParam("id") int id);

   @PUT
   @Path("{id}")
   @Consumes("application/xml")
   public void updateCustomer(@PathParam("id") int id, InputStream is);
}
```

JAX-RS and Java Interfaces

- No JAX-RS annotations are needed within the implementing class. All metadata is confined to the interface, and business logic is less cluttered – greater readability.
- Remember though that the @Path annotation must be applied **to the implementing class, not to the interface** (otherwise Jersey won't be able to find your implementation)
- If you need to, you can override the metadata defined in your interfaces by reapplying annotations within your implementation class (not recommended, as the whole reason for using an interface is to remove metadata from the implementation).
- Note that JAX-RS also permits full class inheritance hierarchies, so we could e.g. use abstract classes to provide a default implementation of a method, which could be overwritten if desired.

Deploying a JAX-RS Service

- JAX-RS applications may be deployed within a Java EE–certified application server (e.g. JBoss) or a standalone Servlet 3 container (e.g. Tomcat). Also supported by cloud platforms such as Google App Engine.
- Can write a class which extends javax.ws.rs.core.Application. This class tells our application server which JAX-RS components we want to register and make available (similar idea to registering objects with the RMI registry).
- In the lab we will use Eclipse EE + a Jersey Maven Archetype to simplify configuration, and deploy our Jersey applications on Tomcat (TomEE) for testing.

javax.ws.rs.core.Application

```
package javax.ws.rs.core;

import java.util.Collections;
import java.util.Set;

public abstract class Application {
   private static final Set<Object> emptySet = Collections.emptySet();

   public abstract Set<Class<?>> getClasses();


   public Set<Object> getSingletons() {
      return emptySet;
   }

}
```

javax.ws.rs.core.Application

- The **getClasses()** method returns a list of JAX-RS service classes (and providers). When the JAX-RS vendor implementation determines that an HTTP request needs to be delivered to a method of one of these classes, an instance of it will be created for the duration of the request and thrown away. You are delegating the creation of these objects to the JAX-RS runtime.
- The **getSingletons()** method returns a list of JAX-RS service objects (and providers). The application programmer is responsible for creating and initializing these objects.
- These two methods tell the JAX-RS vendor which services are to be deployed

Registering a JAX-RS component

```
import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;
import java.util.HashSet;
import java.util.Set;

@ApplicationPath("/services")
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;
    }
}
```

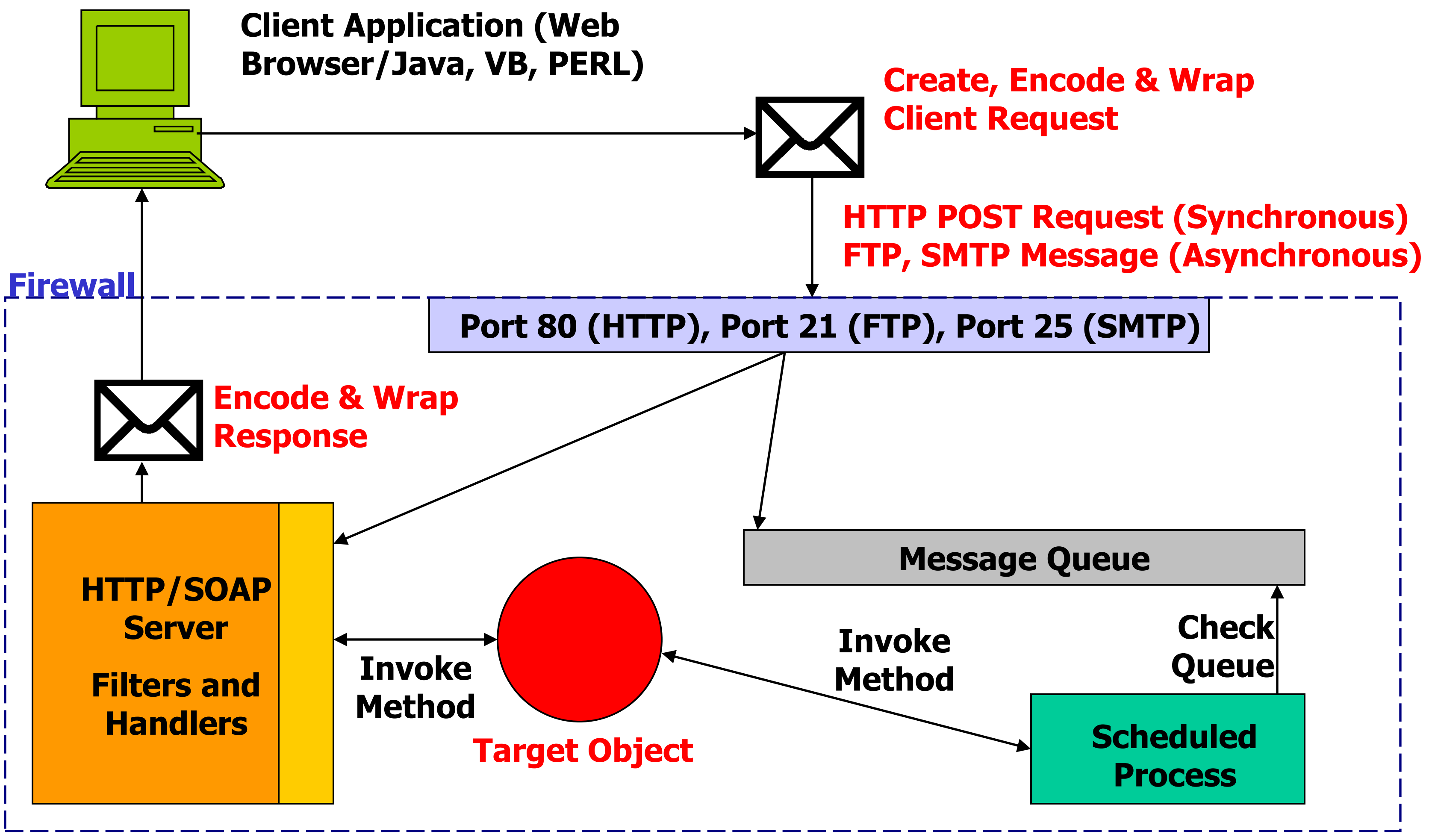
SOAP

- An ***XML-based*** inter-process communication protocol.
  - Can be used in a ***variety of messaging systems*** and delivered via different transport protocols – HTTP, FTP, SMTP and message queues like MSMQ (Microsoft Message Queuing), IBM's MQ or JMS (Java Message Service).
- Enables client applications to easily connect to remote services and invoke remote methods.
- Originally an acronym for *Simple Object Access Protocol*. Now SOAP just means SOAP…
  - Platform and language independent.
- SOAP implementations available for Java, COM, PERL, C#, Python etc.
- *The SOAP 1.2 specification (w3c.org) defines three major parts:*
  - SOAP Envelope Specification
  - Data Encoding Rules
  - RPC Conventions
- NOTE: Most new web app development uses RESTful web services, however SOAP is still also widely used (e.g. by Amazon Web services)

The SOAP Specification

- **The SOAP Envelope Specification**
  - The SOAP XML envelope defines specific ***rules for encapsulating data*** being transferred between hosts.
  - Includes ***application specific data*** like the method name to invoke, parameters and return values.
  - Can include information like who should process the envelope and how to encode error messages.
- **Data Encoding Rules**
  - SOAP includes its own set of conventions for encoding data types.
  - Based on a ***W3C XML Schema*** standard.
- **RPC Conventions**
  - SOAP can be used in a variety of messaging systems, including one-way and two-way messaging.
  - For two-way, SOAP defines a simple convention for representing RPCs and responses.
  - Enables a client to specify a remote method name, parameters, invoke the method a receive a response.

SOAP Overview


Client Application (Web Browser/Java, VB, PERL)
Create, Encode & Wrap Client Request
HTTP POST Request (Synchronous)
FTP, SMTP Message (Asynchronous)
Firewall
Port 80 (HTTP), Port 21 (FTP), Port 25 (SMTP)
Encode & Wrap Response
Message Queue
HTTP/SOAP Server
Filters and Handlers
Invoke Method
Target Object
Invoke Method
Check Queue
Scheduled Process

SOAP vs REST

- https://smartbear.com/blog/test-and-monitor/understanding-soap-and-rest-basics/
- REST – is an architectural style, not a standard (in contrast to SOAP and XML-RPC, which are protocols for inter-process communication)
- In SOAP/RPC systems, the design emphasis is on verbs. What operations can be invoked on the system? E.g. getUser(), addUser(), deleteUser(), getLocation(), updateLocation()
- In RESTful systems, the design emphasis is on nouns, i.e. resources, what their representation is and how they are affected by the standard set of HTTP methods e.g. User resource, Location resource
- REST does not dictate the message format; SOAP/XML-RPC must use XML.

The SOAP Request

- The client request must include the name of the method to invoke and any required parameters:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <SOAP-ENV:Body>
          <nuig:getName
          xmlns:nuig="http://www.nuig.ie/ns/"
          SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
                    <accountNum xsi:type="xsd:string">A1221</accountNum>
          </nuig:getName>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The SOAP Request

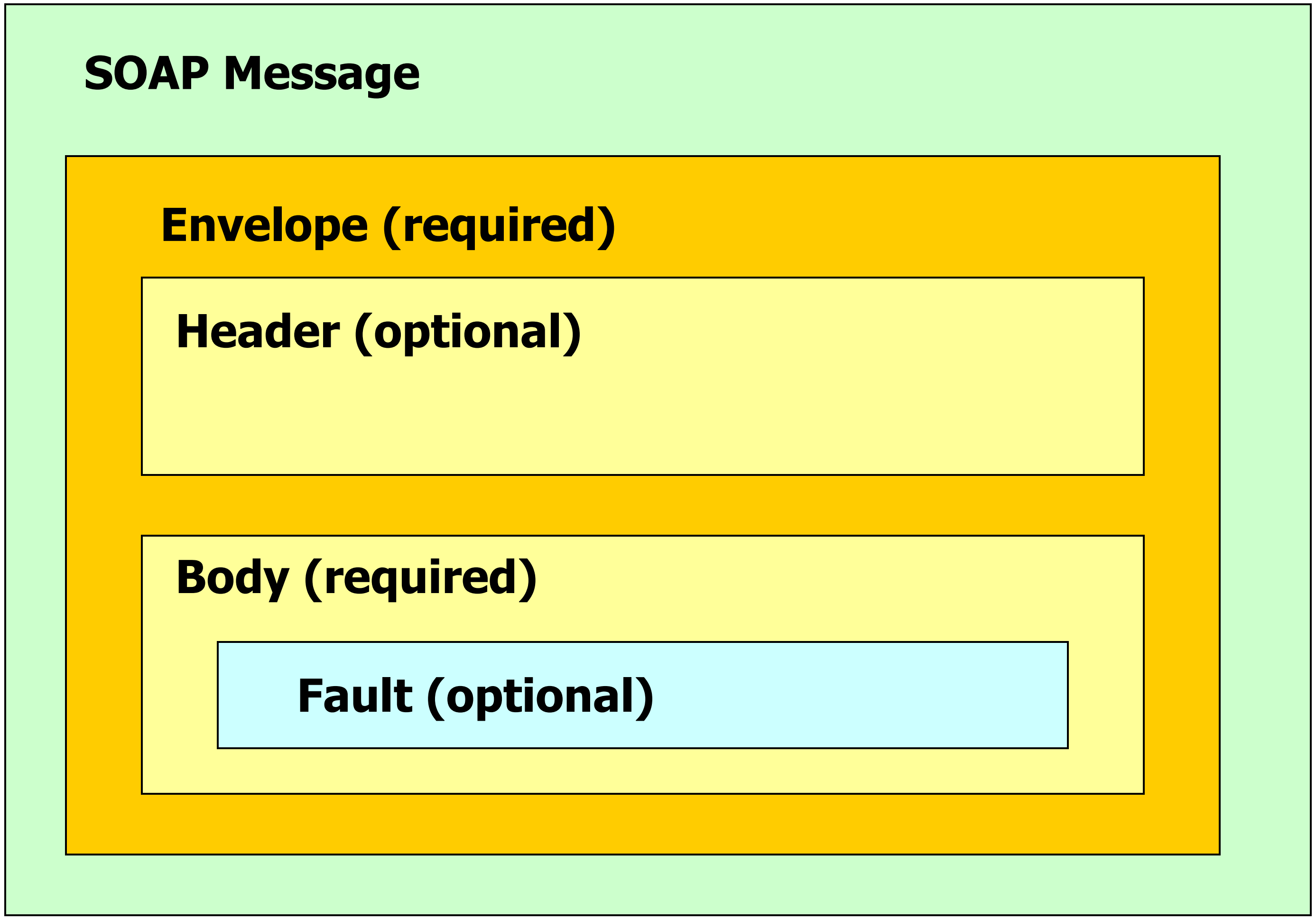
- A total of ***four namespaces*** are used.
- Disambiguate the following identifiers:
  - SOAP Envelope (***SOAP-ENV***)
  - Data encoding (***xsd, xsi***)
  - Application-specific identifiers (***nuig***).
- Enables application ***modularity and flexibility*** (if specifications change in the future).
- The ***body encapsulates the main payload***:
  - The only element is ***the getName()*** method.
  - Each parameter to the method appears as a ***subelement***.
  - Each parameter subelement ***must specify the xsd datatype***.
- When sending SOAP messages over HTTP, a ***SOAPAction*** header indicates the intent of the request.
  - Can determine the nature of a request without looking at payload.
  - Can be used by firewalls to block requests or dispatch to specific SOAP servers.

The SOAP Response

❑ The response to the ***getName()*** request looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <SOAP-ENV:Body>
        <nuig:getNameResponse
        xmlns:nuig="http://www.nuigalway.ie/ns/"
        SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
            <return xsi:type="xsd:string">Michael Monaghan</return>
        </nuig:getNameResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Elements of a SOAP Message



The SOAP Envelope

- Each SOAP message has one root envelope element.
  - ***No conventional form of versioning*** is used (eg. 1.1, 1.2).
- Instead SOAP version determined by the XML ***namespace*** used.
- The version number must be referenced within the envelope element:

  <**SOAP-ENV:Envelope**

  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"

  …
- ***SOAP 1.1*** uses *http://schemas.xmlsoap.org/soap/envelope/*
- ***SOAP 1.2*** uses *http://www.w3.org/2003/05/soap-envelope*
- Any other namespace used is a versioning error.
- ***Note: The SOAP specification prohibits the use of DTD references and XML Processing Instructions.***
- More differences between SOAP versions: https://www.ibm.com/support/knowledgecenter/en/SSAW57_8.5.5/com.ibm.websphere.nd.doc/ae/cwbs_soapverdiffs.html

The SOAP Header

- Header element is ***optional***.
- Used for *specifying additional application-level requirements*.
  - E.g. Digital signatures for password-protected services.
- Not widely used. As SOAP matures, header elements provide a open mechanism for authentication, transaction management, payment authorization etc.
- ***Two header attributes:***
  - ***Actor attribute:***
    - Can define a ***message path*** (a list of SOAP service nodes) that can perform some intermediate processing.
    - Node then forwards message to next node in chain.
    - Can use to ***specify the recipient*** of a SOAP message.
  - ***mustUnderstand attribute:***
    - Indicates whether an actor header element is optional or mandatory.

The SOAP Header

- ***mustUnderstand attribute…***
  - Indicates whether a header element is ***optional or mandatory***.
  - If set to true, the ***recipient must understand and process*** the header attribute according to its defined semantics or return a fault.
  - ***SOAP 1.1*** uses 1 and 0 as mustUnderstand values.
  - ***SOAP 1.2*** uses boolean trues/1/false/0.

```
<SOAP-ENV:Header>
    <nuig:paymentAccount
          xmlns:nuig="urn:nuig-computing" SOAP-ENV:mustUnderstand="true">
                    877197789ADF
    </nuig:paymentAccount>
</SOAP-ENV:Header>
```

SOAP Errors - The Fault Element

- Error message from a SOAP application is carried ***inside a Fault element***.
  - Fault element must appear as an element ***within the Body element***.
  - A Fault element ***can only appear once*** in a SOAP message.
- ***Fault element has the following sub elements:***
  - ***<faultcode>:*** A code identifying the error.
  - ***<faultstring>:*** The error as a string.
  - ***<faultactor>:*** Who caused the error.
  - ***<detail>:*** Specific error information .
- ***Common Error Messages:***
  - ***VersionMismatch:*** Invalid namespace for the SOAP Envelope element
  - ***MustUnderstand:*** A child element of the Header element, with the mustUnderstand attribute set to "1", was not understood
  - ***Client:*** message was incorrectly formed or contained incorrect information
  - ***Server:*** Problem with the server so the message could not proceed

SOAP Encoding

- Although the W3C encourages the use of SOAP encoding rules, this is **not a requirement**.
  - Can choose a ***different encoding schema*** if the need arises.
- The encoding style of a SOAP message is ***set using the SOAP-ENV:encodingStyle attribute***.
  - ***SOAP 1.1*** uses schemas.xmlsoap.org/soap/encoding
  - ***SOAP 1.2*** uses www.w3.org/2003/05/soap-encoding
- Scalar types use the ***simple types*** from the XML Schema specification. `<accountNum xsi:type="xsd:string">A1221</accountNum>`
- ***Complex types*** (arrays and structs):
  - xsi:type="nuig:array"
  - Structs contain multiple values, so each element is specified with a unique accessor name.
- Can use a custom encoding style. This is called ***literal encoding***.

SOAP Encoding – Scalar Types

Simple Type	Example
string	Hello world!
base64Binary	GpM7
hexBinary	0FB7
integer	-126789, -1, 0, 126789
positiveInteger	1, 126789
negativeInteger	-126789, -1
nonNegativeInteger	0, 1, 126789
nonPositiveInteger	-126789, -1, 0
decimal	-1.23, 0, 123.4, 1000.00
boolean	true, false, 1, 0
time	13:20:00.000
dateTime	1999-05-31-T13:20:00.000
duration	P1Y2M3DT10H30M12.3S
date	1999-05-31
name	ShipTo
QName	po:IrishAddress
AnyURI	http://www.example.com
ID*	121
IDREF*	121

SOAP Encoding – Compound Types

- Arrays defined by specific set of rules – ***have to specify element type and array size***.
  - SOAP ***supports multidimensional arrays***, but not all implementations…
- Arrays must be specified as an ***xsd:type*** of ***Array***. Must also include an ***arrayType*** attribute.
  - ***arrayType specifies size and type***, e,g. arrayType="xsd:double[10]" or even arrayType="xsd:string[5,5]".

```
<return
    xmlns:ns2="http://www.w3.org/2003/05/soap-encoding"
    xsi:type="ns2:Array" ns2:arrayType="xsd:double[2]">
    <item xsi:type="xsd:double">12.99</item>
    <item xsi:type="xsd:double">23.56</item>
<return>
```

- Structs ***contain multiple values***, but each element is specified with a ***unique accessor element***.

SOAP Encoding – Compound Types

- The following part of a SOAP response demonstrates the use of a ***struct*** for a stock request.

```
<SOAP-ENV:Body>
    <ns1:getProductResponse
     xmlns:ns1="urn:nuig.productService"
     SOAP-ENV:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
      <return xmlns:ns2="urn:nuig" xsi:type="ns2:product">
                <name xsi:type="xsd:string">Red Hat Linux</name>
                <price xsi:type="xsd:double">84.99</price>
                <description xsi:type="xsd:string">
                        Red Hat Linux 9.2 Operating System
                </description>
                <sku xsi:type="xsd:string">A358185</sku>
      </return>
    </ns1:getProductResponse>
</SOAP-ENV:Body>
```

SOAP Encoding – Literal Encoding

- You are ***not required* to use the SOAP encoding style**.
  - May want to embed a complete or partial XML document into a SOAP envelope.
- Literal encoding requires that ***you*** specify an encoding style.
  - Use the namespace ***http://xml.apache.org/xml-soap/literalxml*** for literal encoding with Apache SOAP.

```
<ns1:getProductResponse
 xmlns:ns1="urn:nuig.myLiteralProductService"
 SOAP-ENV:encodingStyle="http://xml.apache.org/xml-soap/literalxml">
  <return>
        <name xsi:type="xsd:string">Red Hat Linux</name>
        <price xsi:type="xsd:double">84.99</price>
        <description xsi:type="xsd:string">
              Red Hat Linux 9.2 Operating System
        </description>
        <sku xsi:type="xsd:string">A358185</sku>
  </return>
</ns1:getProductResponse>
```

Some SOAP Implementations

- ***Apache SOAP 2.3.1 (ws.apache.org/soap/)***
  - Open source java implementation of the SOAP protocol. Based on the ***IBM SOAP4J*** implementation.
- ***Apache Axis 1.2 (xml.apache.org/soap/)***
  - Axis is a follow-on to Apache SOAP. Operates as a stand-alone server or servlet container plugin. Extensive support for WSDL.
- ***Java Web Services Developer Pack 1.2 (java.sun.com/webservices/)***
  - Extensive library of resources for WS. Includes ***saaj.jar*** (SOAP with Attachments API for Java) and ***XML and Web Services Security v1.0***.
- ***Microsoft SOAP Toolkit 2.0 (msdn.microsoft.com/soap)***
  - ***COM implementation*** of the SOAP protocol for VB, C, C# or any other COM-compliant language.
- ***SOAP::Lite for PERL (www.soaplite.com)***
  - ***PERL implementation*** of the SOAP protocol. Includes WDSL and UDDI support.
- **Apache CXF** https://cxf.apache.org/

Describing Web Services

- Structured markup languages (e.g. XML, YAML variants) are typically used to describe the functionality of web services
- Description languages include: WSDL, WADL, RAML, OpenAPI etc. etc.
- Files containing the markup may be made available to potential users of a service, using e.g. a HTTP GET request + response
- They perform a similar role to that of Remote Interfaces in Java RMI. The client has a complete description of the available functionality, without knowing any of the implementation details
- Many automated tools are available to generate client code stubs/applications directly from a service descriptor, e.g.
  - Apache CXF WSDL to Java converter (http://cxf.apache.org/docs/wsdl-to-java.html)
  - Apache CXF WADL to Java converter (http://cxf.apache.org/docs/jaxrs-services-description.html)
- Automated code generation saves a significant amount of effort for the developer

Web Services Description Language

- A W3C working draft that defines a web service using a common XML grammar.
  - V2.0 26 June 2007 - https://www.w3.org/TR/wsdl20/
- WSDL describes four key aspects of a service:
  - ***Interfaces***: Describes all the public methods of the object or application be exposed as a web service.
  - ***Data Types***: The data types of the parameters for a remote method and the return value(s) for that method.
  - ***Binding***: Information describing the network transport being used.
  - ***Service Location***: The network address of the specified web service.
- *A binding contract between a service requestor and a service provider.*
  - A client application can use a WSDL document to locate a web service and invoke any public methods available.

WSDL Specification

- The WSDL specification defines eight principal elements:
  - ***<definitions>*** The root element of a WSDL document. It defines the name of the web service and any namespaces used.
  - ***<types>*** Describes all the data-types used between the service requestor and service provider. The W3C XML Schema specification is used by default.
  - ***<message>*** Describes a one-way message. The message can be a single request or response defining the message name, parameters and return values.
  - ***<portType>*** Wraps multiple messages as a service. The name of the port type is one of the parameters passed by the client SOAP request.
  - ***<binding>*** Defines how the web service will be implemented on the wire. The description includes the type of request, normally RPC, as well as encoding and namespace definitions.

WSDL Specification

  - ***<service>*** Defines the address for invoking the specified service. This is implemented as a URL that points to the WSDL document.
  - ***<documentation>*** A utility element that allows for a human-readable description of the purpose of any WSDL element.
  - ***<import>*** Enables a WSDL document to import other WSDL documents and XML schemas. This facilitates developers in creating more modular WSDL documents.
- *WSDL performs the same role in a web services architecture that IDL and Java Interfaces perform in CORBA and Java RMI respectively.*
- WSDL normally generated using wizards/command-line tools.
- Example of a simple WDSL document describing a SOAP web service: https://www.tutorialspoint.com/wsdl/wsdl_example.htm

Amazon Product Advertising API

- "Amazon has developed a world-class web service that millions of customers use every day. As a developer, you can build Product Advertising API applications that leverage this robust, scalable, and reliable technology. You get access to a lot of the data used by Amazon including the items for sale, customer reviews, seller reviews, as well as most of the functionality you see on Amazon.com, such as finding items, displaying customer reviews, and product promotions. Product Advertising API operations open the doors to Amazon's databases so that you can take advantage of Amazon's sophisticated e-commerce data and functionality. Build your own web store to sell Amazon items or your own items."
- https://docs.aws.amazon.com/AWSECommerceService/latest/DG/Welcome.html
- http://webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl

Describing RESTful Web Services

- In contrast to SOAP, numerous different options available, e.g.
  - WSDL
  - Web Application Description Language (WADL) (https://javaee.github.io/wadl/)
  - OpenAPI Specification (https://www.openapis.org/)
  - RESTful API Modeling Language (RAML) (https://raml.org/)
  - etc.

Describing RESTful Web Services

- ❑ Using WSDL:
  - https://www.ibm.com/developerworks/webservices/library/ws-restwsdl/
- ❑ Describing JAX-RS/Jersey services using WADL:
  - https://jersey.github.io/documentation/latest/wadl.html
- ❑ Generating Java code from a WADL document:
  - https://www.javacodegeeks.com/2012/01/wadl-in-java-gentle-introduction.html
- ❑ Using OpenAPI:
  - https://app.swaggerhub.com/help/tutorials/openapi-3-tutorial
- ❑ Using RAML:
  - https://raml.org/developers/raml-100-tutorial

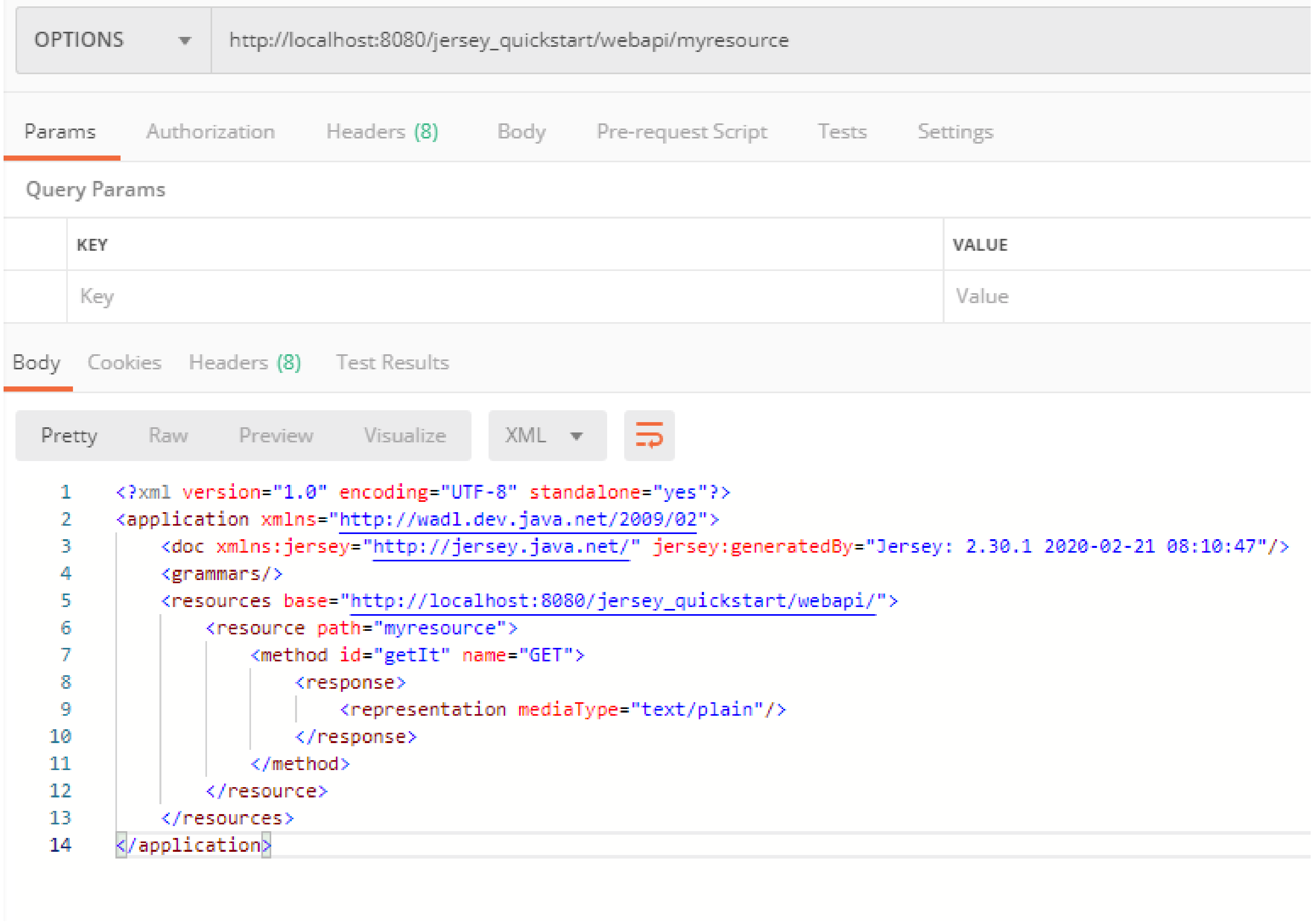
Describing JAX-RS / Jersey RESTFUL services with WADL

Can navigate in a web browser to
http://localhost:8080/jersey_quickstart/webapi/application.wadl
WADL is automatically generated by Jersey

https://docs.huihoo.com/jersey/2.13/wadl.html

```
<application xmlns="http://wadl.dev.java.net/2009/02">
    <doc xmlns:jersey="http://jersey.java.net/" jersey:generatedBy=
    "Jersey: 2.30.1 2020-02-21 08:10:47"/>
    <doc xmlns:jersey="http://jersey.java.net/" jersey:hint="This is
    simplified WADL with user and core resources only. To get full WADL
    with extended resources use the query parameter detail. Link:
    http://localhost:8080/jersey_quickstart/webapi/application.wadl?detail=
    true"/>
    <grammars/>
    <resources base="http://localhost:8080/jersey_quickstart/webapi/">
    <resource path="myresource">
    <method id="getIt" name="GET">
    <response>
    <representation mediaType="text/plain"/>
    </response>
    </method>
    </resource>
    </resources>
```

Generated WADL for HTTP OPTIONS Method with Postman



```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<application xmlns="http://wadl.dev.java.net/2009/02">
    <doc xmlns:jersey="http://jersey.java.net/" jersey:generatedBy="Jersey: 2.30.1 2020-02-21 08:10:47"/>
    <grammars/>
    <resources base="http://localhost:8080/jersey_quickstart/webapi/">
        <resource path="myresource">
            <method id="getIt" name="GET">
                <response>
                    <representation mediaType="text/plain"/>
                </response>
            </method>
        </resource>
    </resources>
</application>
```