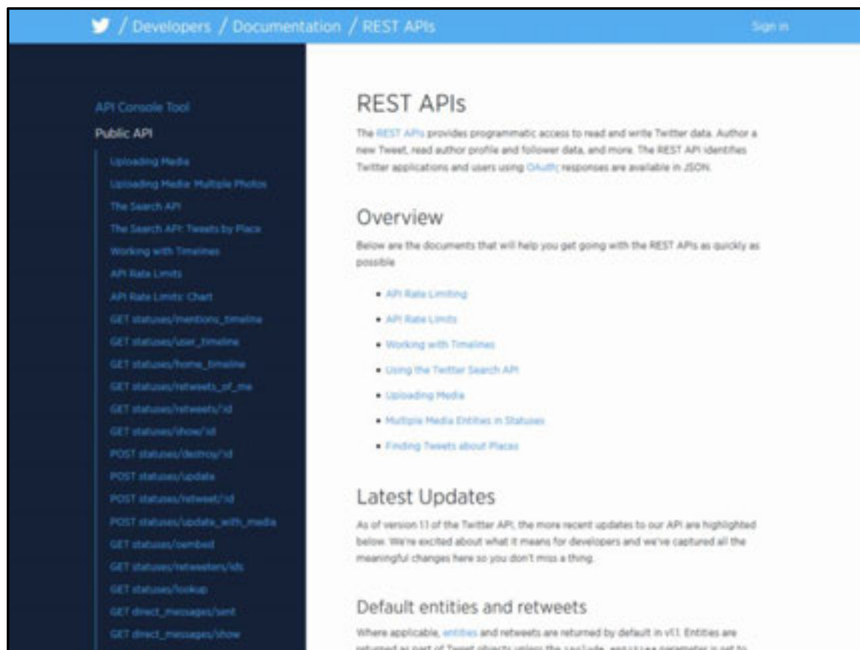
Web services



Modern web applications and web sites are not "islands". They need to communicate with each other and share information.

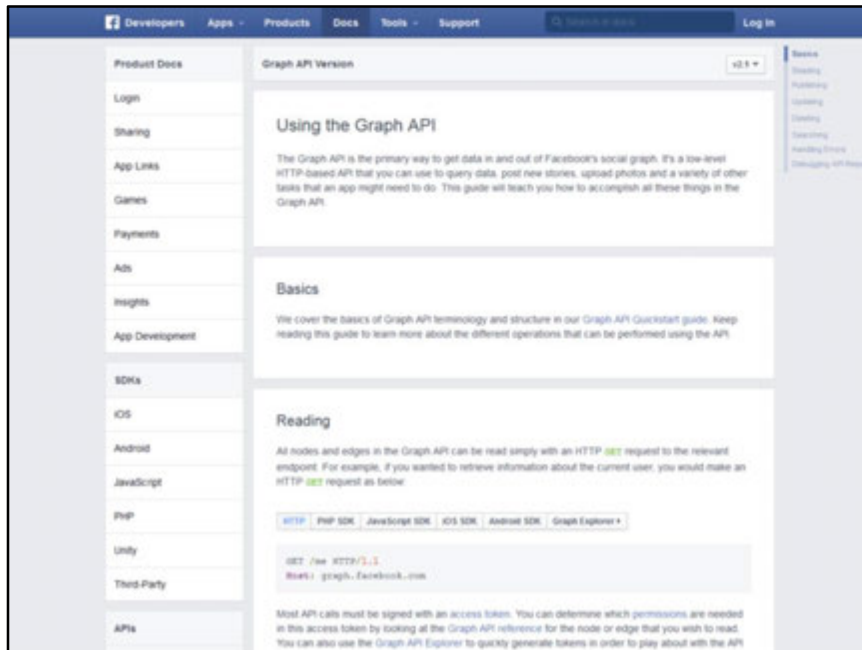
For example, when you develop a web application, you may need to do some of the following:

- Connect to a social network to authenticate a user
- Connect to a social network to send a post/message on behalf of your users
- Connect to a billing gateway to charge a credit card
- Connect to a phone gateway to make or receive phone calls
- Connect to a search engine to perform custom searches
- Connect to a analysis service to transform or analyze images or sound
- Connect to a video service to transform video
- And so on…

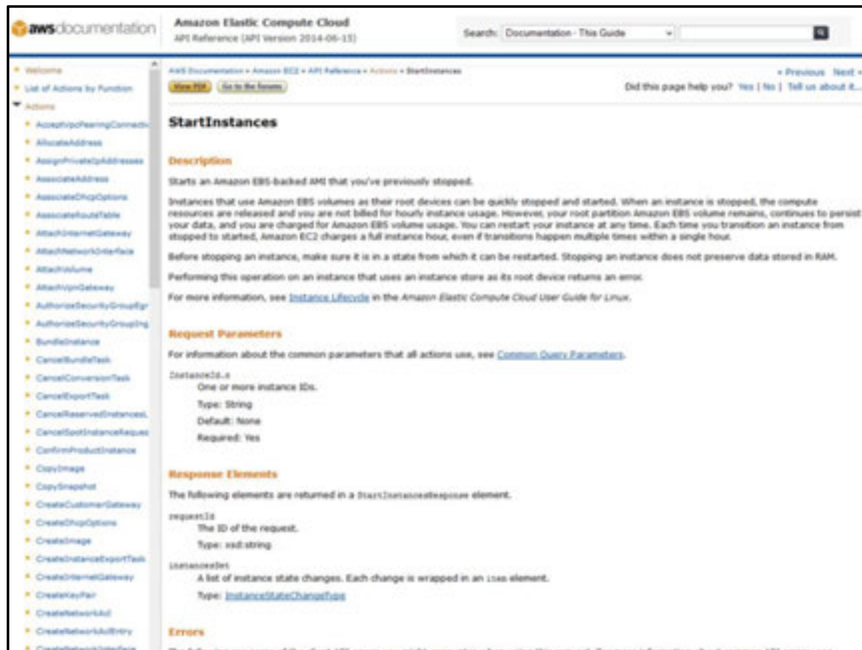
Conversely, if you are providing a novel service or technology, or if your site managed valuable data, other developers (or your users) may want to connect to your service.

Today, the typical way of exchanging information or allowing remote method calls is via web services.

Twitter, for example, provides a rich API for checking statuses, searching, posting and retrieving user timelines.



Facebook provides an API to extract information from the Facebook social graph.



Amazon provides web services that allow you to automatically manage all of their technologies. For example, you can use the StartInstances web service to automatically launch a new web server on Amazon EC2.

The Amazon bookstore also comes with a comprehensive API.

Twilio provides a web service for making and receiving telephone calls (and SMSes).

Stripe provides a credit card processing gateway.

Pin payments also provides a billing gateway.

Remote methods

Accessing remote services

Network protocols:
- RPC (Remote Procedure Call)
- RMI (Java Remote Method Invocation)
- CORBA (Common Object Request Broker Architecture)
- RMI over IIOP (RMI over Internet Inter-ORB Protocol)

How can these websites to communicate with each other?

This is an old problem that goes back to the early days of computer networks: how can code running on one computer take advantage of code running on another computer?

One approach could be to use standard information exchange protocols of the internet (e.g., email/SMTP or HTTP).

However, if we are building a distributed system, it may be preferable to use a method invocation model that mimics our programming languages. That is, we would like to be able to call a method on a remote computer as though it is running on the same computer.

This idea is called a "remote procedure call" (RPC). The idea goes back to the 1980s. Specialized network protocols were developed to help build 'distributed systems'.
http://en.wikipedia.org/wiki/Remote_procedure_call

In Java, remote procedure calls were first implemented using "RMI". RMI is an object-

oriented approach to remote procedure calls. It allows objects (not just methods) to be accessed over a network. A special "stub" is used on the client: it is a local object that acts like the remote object. It uses the network to communicate with the RMI server. On the server, a "skeleton" receives requests from the client stub and passes the request on to the "real" object. Any results are returned back to the client via the skeleton and the stub.

CORBA is a complex standard for distributed systems. It is a cross-platform RPC/RMI technology.

With the advent of CORBA, Java provided a version of RMI that is compatible with the CORBA "Inter-ORB protocol". In fact, whenever you use a remote interface of an Enterprise JavaBean, your EJB is compatible with CORBA.

All of these approaches used custom low-level protocols. They are complex but generally efficient and quite powerful.

Challenges

- Performance, efficiency and bandwidth
- Security
- Reliability
- Cross-platform and cross-language support
- References and garbage collection
- Naming and directory services

These are some of the problems faced by the creators of RPC technologies.

Developing solutions to these problems creates a lot of complexity.

RPC technologies try to hide the details of the network. However, the price of hiding these challenges and details is complexity.

Web services

HTTP-based:
- XML-RPC (XML Remote Procedure Call)
- SOAP (Simple Object Access Protocol)
- REST (Representational State Transfer)

With the rise of the world-wide-web and the domination of HTTP as the most popular protocol for accessing remote resources, RPC technology adapted to take advantage of the HTTP protocol.

There are some advantages in using a HTTP-based protocol for creating remote method calls:
- HTTP is simple and easy to implement
- The popularity of the web means that most programming languages already support HTTP
- HTTP is often the easiest technology for bypassing corporate firewalls (everyone needs HTTP to browse the web, so port 80 is unlikely to be blocked by a firewall) – this means that accessing a web service does not require developers to get through bureaucracy or corporate politics
- HTTP has established technologies and techniques for caching and load-balancing
- HTTP is well understood by developers
- HTTP can be tested using a web browser

In a sense, a HTTP-based approach does not try to hide the challenges of RPC. It leaves the complexity to the developer to solve. Perhaps this isn't such a big problem

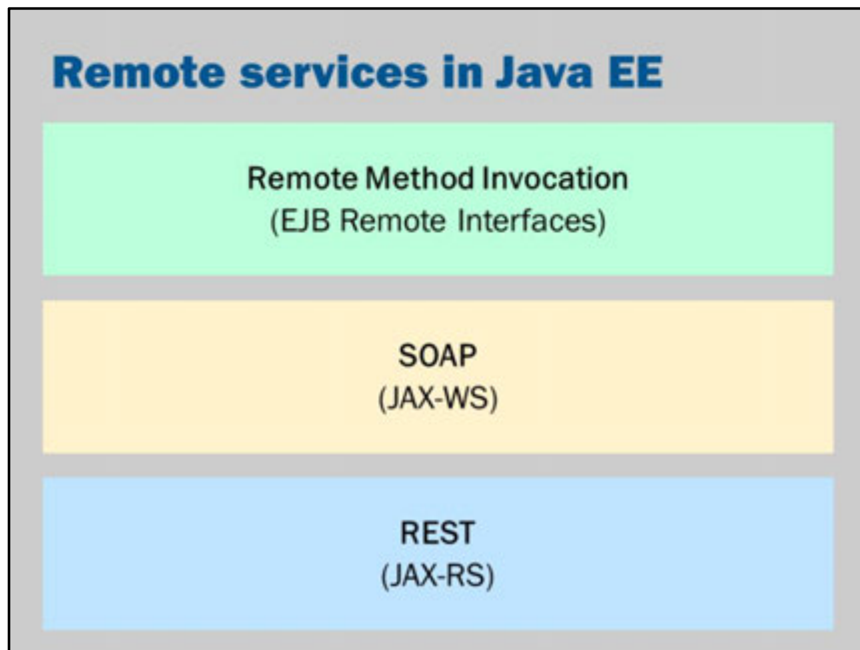
because ultimately networks do fail and no matter how sophisticated the underlying RPC technology, there is no way to send data through a disconnected network cable.

Three main approaches to handling remote method calls over HTTP have arisen:

XML-RPC provides a classic RPC-style architecture and is a simple protocol that works. It is based on sending and receiving simple XML documents.

SOAP (Simple Object Access Protocol) is also XML-based. However, despite its name, it is a quite complex technology. It has "all the bells and whistles" and perhaps its power has caused it to overshadow XML-RPC. That is, SOAP seems to have been much more successful than XML-RPC.

REST is not a standardized technology. It is a set of conventions for treating remote method invocations like requests for web pages. In REST, standard HTTP methods are used to POST or GET XML or JSON content from a server. While REST lacks standards and few frameworks or automatic tools, its simplicity seems to have driven rapid adoption. Most new web services on the internet are using a RESTful architecture.



These are the three main approaches to creating and using web services in Java EE.

We will not cover RMI / EJB in this lecture, as we already covered it in previous lectures.
To create a remote interface for an EJB, you simply add a @Remote interfaces to your EJB.

JAX-WS is the Java technology for creating XML-based Web Services.

JAX-RS is the Java technology for creating XML-based (and JSON-based) RESTful Web Services.

JAX-WS

SOAP

Simple Object Access Protocol
- W3C specification
- XML-based protocol

Combines multiple technologies:
- XML and XML Schema, used for encoding all data
- HTTP, the transport protocol used for connections (other transports are also allowed)
- SOAP, a one-way message exchange technology
- SOAP RPC, a uniform representation for RPC style request/response messaging
- WSDL, a language to describe the interface
- UDDI, a protocol for accessing a registry of web services

SOAP is a complex set of standards and specifications.

You can view the specification here:
http://www.w3.org/TR/soap/

SOAP RPC

Request

```
<?xml version="1.0" encoding="UTF-8"?>
<s:Envelope
    xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Header/>
  <s:Body>
    <chat:addMessage
        xmlns:chat="http://server.chat.aip.uts.edu.au/">
      <arg0>0</arg0>
      <arg1>Hey there!</arg1>
    </chat:addMessage>
  </s:Body>
</s:Envelope>
```

Response

```
<?xml version="1.0" encoding="UTF-8"?>
<s:Envelope
    xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Header/>
  <s:Body>
    <chat:addMessageResponse
        xmlns:chat="http://server.chat.aip.uts.edu.au/">
      <return>
        <id>1</id><message>Greetings!</message>
        <timestamp>2014-10-04T16:23:44.706+10:00</timestamp>
      </return>
      <return>
        <id>2</id><message>Hey there!</message>
        <timestamp>2014-10-04T16:24:33.672+10:00</timestamp>
      </return>
    </chat:addMessageResponse>
  </s:Body>
</s:Envelope>
```

This is an example of the body of a SOAP request and response.

Each message is XML.
Each message is in "envelope" that contains a header and then the SOAP message body.

While you can build XML SOAP messages directly, they are more typically created using toolkits or wizards that come with your programming language or development environment.

i.e., the typical developer does not need to understand these low-level details

Creating a SOAP web service

```
package au.edu.uts.aip.chat.server;

import java.util.*;
import javax.jws.*;

@WebService
public class Chat {

  @WebMethod
  public Collection<Message> addMessage(int lastSeen,
                                        String message) {
    // implementation...
  }

  @WebMethod
  public Collection<Message> getMessages(int lastSeen) {
    // implementation...
  }

}
```

On Java EE, creating a SOAP web service is as simple as adding annotations to a class. Java EE does all the work of ensuring that the SOAP service is created: it publishes a WSDL file, it sets up the web service and it serializes/deserializes the parameters of method calls.

Generated WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions>
  ...
  <portType name="Chat">
    <operation name="addMessage">
      <input wsam:Action="http://jaxwschat/Chat/addMessageRequest"
   message="tns:addMessage"/>
      <output wsam:Action="http://jaxwschat/Chat/addMessageResponse"
   message="tns:addMessageResponse"/>
    </operation>
    ...
  </portType>
  <binding name="ChatPortBinding" type="tns:Chat">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
   style="document"/>
    <operation name="addMessage">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
    ...
  </binding>
  <service name="ChatService">
    <port name="ChatPort" binding="tns:ChatPortBinding">
      <soap:address location="http://chat/Chat/ChatService"/>
    </port>
  </service>
</definitions>
```

When you deploy a JAX-WS service, you can write a WSDL interface definition file or you can let the application server automatically generate one for you.
This is an excerpt of an automatically generated WSDL file.

Java web service client

```
// Get the client
ChatService service = new ChatService();
Chat chat = service.getChatPort();

// Send a message
chat.addMessage(-1, "Hello from Java");

// Show the results
for (Message message : chat.getMessages(-1)) {
   System.out.println(message.getText());
}
```

The generated WSDL file can then be used to generate Java class files that serve as a client to the web service.

Here's an example of code that uses the web services.

Note that, aside from getting the Chat object from the ChatService, it is basically identical to calling local methods on the local computer.

.NET web service client

```
// Get the client
ChatClient chat = new ChatClient();

// Send a message
chat.addMessage(-1, "Hello from C#");

// Show the results
foreach (var message in chat.getMessages(1))
    Console.WriteLine(message.text);
```

That same Java class can be used from .NET or C#.

This is code that I wrote using C# to conect to the Java web service.

Note, of course, that the reverse is also possible.
It is just as easy to connect to a Python web service using Java.

Python web service client

```
from suds.client import Client

// Get the client
client = Client('http://chat/Chat/ChatService?WSDL')

// Send a message
client.service.addMessage(-1, 'Hello from Python')

// Show the results
for x in client.service.getMessages(-1):
  print x.text
```

In Python, is it is even easier to connect to a SOAP web service.

This code here connects to the Java-based JAX-WS web service.

Note, of course, that the reverse is also possible.
It is just as easy to connect to a Python web service using Java.

JAX-RS, the Java API for RESTful Web Services is as easy to use as SOAP (or, perhaps even easier).

Simplicity

Perhaps the success of simple internet protocols can be explained by Metcalfe's "law":

- The value of a telecommunications network is proportional to the square of the number of connected users of the system

https://en.wikipedia.org/wiki/Metcalfe%27s_law

REST

REST

REpresentational State Transfer:

- Architectural style (not a protocol)
- Based around resources, identified by URIs
- Resources can have multiple representations (media types), typically JSON and XML
- Standard HTTP methods are used:
  - *GET to fetch*
  - *POST to create*
  - *PUT to update*
  - *DELETE to remove*
- Communication is stateless

The concept of REST was described in Roy Fielding's doctoral dissertation:
Fielding, R. (2000) *Architectural Styles and the Design of Network-based Software Architectures*, PhD Dissertation, University of California, Irvine.
Available online here:
http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

SOAP vs REST

SOAP is based around method invocations:
- Call addMessage
- Call getMessage
- Call getMessages

REST is based around HTTP methods:
- POST to http://myserver/api/message
- GET http://myserver/api/message/3
- GET http://myserver/api/message

You could argue that in theory, there's no difference: it is just data set over the network connection with a particular structure.

However, in practice, by using the same technologies as a web browser, REST seems to be simpler to use and understand.

SOAP defines a standard for encoding method calls in XML to send over HTTP.
REST essentially says "let's just use HTTP directly".

REST benefits

- Simplicity
- Scalability (Statelessness and Cache-ability)
- Portability
- Uniformity
- Compatibility

XML

```
<?xml version="1.0"?>
<messages>
  <message>
    <id>1</id>
    <message>Greetings!</message>
    <timestamp>
      2014-10-04T16:23:44.706+10:00
    </timestamp>
  </message>
  <message>
    <id>1</id>
    <message>Greetings!</message>
    <timestamp>
      2014-10-04T16:23:44.706+10:00
    </timestamp>
  </message>
</messages>
```

This is a simple XML document.
When using REST, there is not fixed structure for the messages.
You choose an XML document that meets the business requirements of your application.
This XML document might be one created by a RESTful web service.

One thing to note here – that will come up later – is that an XML document has only one root element.
The whole document has to be wrapped in a single element.
In this case it is <messages></messages>

JSON

```
{
  "messages": [
    {"id": 1,
     "message": "Greetings!",
     "timestamp": "2014-10-04T16:23:44.706+10:00"},
    {"id": 2,
     "message": "Hey there!",
     "timestamp": "2014-10-04T16:24:33.672+10:00"}
  ]
}
```

JSON = JavaScript Object Notation
- JSON files are JavaScript expressions
- Possible to parse using eval(), but this is poor practice
- Not all JavaScript expressions are valid JSON

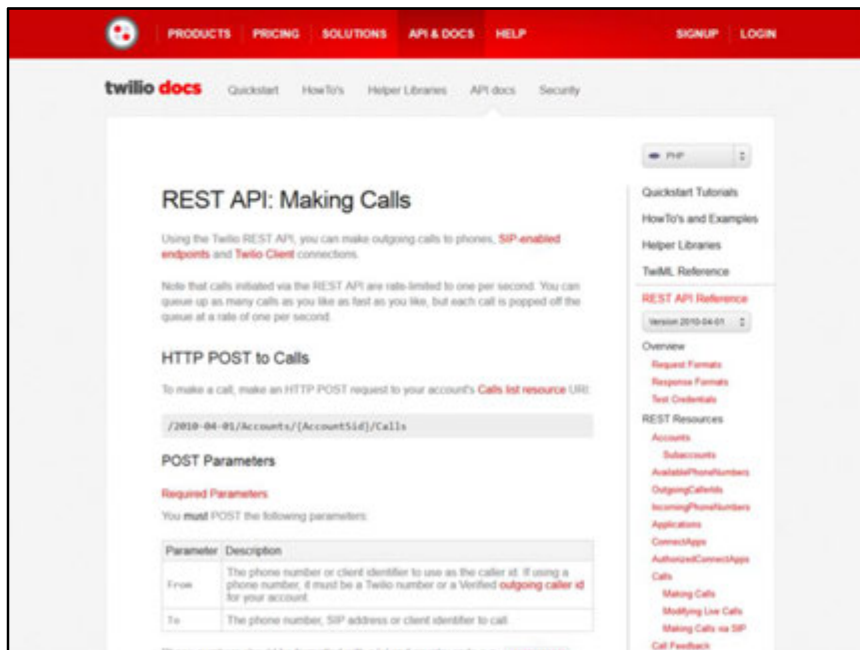
This is one way that the document on the previous slide might be converted into JSON.

http://en.wikipedia.org/wiki/JSON
http://json.org/

What are some advantages of JSON when compared to XML?
- JSON is very simple
- JSON works well with JavaScript and web applications
- JSON is easy to parse, particularly in JavaScript
- JSON uses less space than XML (smaller files)
- JSON does not require schemas

What are some disadvantages of JSON when compared to XML?
- JSON lacks schemas (no way to enforce a structure or to validate the file)
- JSON tends to work less effectively in languages other than JavaScript
- JSON lacks the sophisticated tools of XML (such as XSLT, validators, rendering pipelines, specialized databases, programming language integration)

- JSON is not particularly good with large and complex documents
- JSON is not a good format for rich textual documents



There is no fixed standard for how REST works.
You need to look at the documentation to understand what to do.

Here's an example from Twilio:
It tells us that to make a phone call you need to make a HTTP post request to /2010-04-01/Accounts/{AccountSid}/Calls.
You need to replace the {AccountSid} with your account details, and then you encode the From and To numbers into the posted form data.

Don't worry about the particular details.
The key point is that the documentation will tell you exactly what you need to do and what format the request and response will be in.

In SOAP this is handled automatically using WSDL.
In REST, you have to read the documentation.

Creating a service

It is very easy to create a RESTful web service.

Initial configuration

```
package au.edu.uts.aip.chat.server;

import javax.ws.rs.*;
import javax.ws.rs.core.*;

@ApplicationPath("api")
public class ApplicationConfig extends Application {

}
```

First, you need to have a concrete subclass of Application in your project.
The @ApplicationPath annotation, tells JAX-RS the base URL for all of your web resources.

Creating a resource

```java
package au.edu.uts.aip.chat.server;

import java.util.*;
import javax.ws.rs.*;

@Path("message")
public class Messages {

  @GET
  public List<Message> getMessages() {
    // implementation
  }

  @POST
  public List<Message> addMessage(
      @FormParam("lastSeen") int lastSeen,
      @FormParam("message") String message) {
    // implementation
  }

  @GET
  @Path("since/{id}")
  public List<Message> getMessages(@PathParam("id") int lastSeen) {
    // implementation
  }

  @GET
  @Path("{id}")
  public Message getMessage(@PathParam("id") int id) {
   // implementation
  }

}
```

A resource is created by annotating a class or method with @Path.
You can specify the HTTP methods using @GET, @POST, @PUT and so on.

JAX-RS annotations allow you to receive parameters in the message body, as form parameters, in the path, in the headers, in cookies, as query parameters or as matrix parameters.

(A matrix parameter is a parameter encoded in the URL, like a query parameter but separated by semicolons: http://www.w3.org/DesignIssues/MatrixURIs.html)

JAX-RS annotations

Content Type	`@Consumes("*/xml")`
	`@Consumes(MediaType.APPLICATION_XML)`
	`@Produces(MediaType.APPLICATION_JSON)`

Parameter	`@CookieParam("JSESSIONID")`
	`@HeaderParam("Accept")`
	`@PathParam("id")`
	`@QueryParam("name")`
	`@FormParam("name")`

Response	`return "Hello, world!";`
	`return object;`
	`return Response.ok(object,...).build();`
	`return Response.seeOther(uri).build();`
	`return Response.serverError();`

JAX-B XML binding

JAX-RS needs to convert your returned objects into an appropriate XML or JSON document to return to the user.

This is done by XML binding. JAX-RS can do the conversion automatically.

This XML binding is achieved using another technology called JAXB.

XML bindings

```
@XmlRootElement
public class Message implements Serializable {

  public int getId() {
    return id;
  }

  public String getMessage() {
    return message;
  }

  public Date getTimestamp() {
    return timestamp;
  }

  ...

}
```

The key annotation here is @XmlRootElement.
This tells JAXB that the class can be used as the root element of an XML document.

You need the @XmlRootElement to tell JAXB that the object can be converted into XML.
From there, JAXB can do the rest.

Generated XML

```
<message>
  <id>1</id>
  <message>Hello, World</message>
  <timestamp>
    2015-07-14T15:03:44.263+10:00
  </timestamp>
</message>
```

XML bindings

```
@XmlRootElement
public class Message implements Serializable {

  @XmlAttribute
  public int getId() {
    return id;
  }

  @XmlValue
  public String getMessage() {
    return message;
  }

  @XmlAttribute(name = "time-stamp")
  public Date getTimestamp() {
    return timestamp;
  }

  ...

}
```

The @XmlAttribute tells JAX-B to store the value as an attribute (rather than a separate XML element)
The @XmlValue tells JAX-B to store the value directly as the text inside the element.

Generated XML

```
<message id="1"
         time-stamp="2015-07-14T15:03:44.263+10:00">
  Hello, World
</message>
```

So, this is the resulting XML:

Notice that the message text is now directly inside the <message> element (it is no longer in a sub-element).

Notice that the id and time-stamp are stored as attributes, rather than separate <id> and <timestamp> elements.

JSON bindings

```
@XmlRootElement
public class Message implements Serializable {

  public int getId() {
    return id;
  }

  public String getMessage() {
    return message;
  }

  @XmlElement(name = "time-stamp")
  public Date getTimestamp() {
    return timestamp;
  }

  ...

}
```

JSON bindings aren't officially part of the Java EE standards yet (JSONB is coming soon).
However, GlassFish is able to use XML bindings to customize the conversion of your objects into JSON.
The same JAXB annotations can be used to customize JSON.
However, JAXB annotations are not required.

If you're using JSON with JAXB, it is best to just stick to @XmlElement.
JSON does not have the concept of values or attributes.

Generated JSON

```
{"id":1,
 "message":"Hello, World",
 "time-stamp":"2015-07-14T15:09:45.031"}
```

JAX-RS clients

Calling a RESTful web service

Create your JAXB classes for binding, then:

1. Obtain an instance of javax.ws.rs.client.Client
2. Configure client with the target (the URL)
3. Create a request
4. Invoke the request
5. Close the client

JAX-RS provides a "fluent" API to chain these actions together

JAX-RS is easy to use. In the following slides, you'll see some examples.

The client API provided by JAX-RS is referred to as a “fluent” API.

This means that you do not need to call methods one-after-the-other, like this:

```
Thing x = new Thing();
x.setStart(1);
x.setEnd(2);
ThingHelper helper = x.getHelper("asdf");
Result result = helper.getResult();
```

Instead, a “fluent” API is designed to allow methods to be conveniently chained together in a way that might also read like “normal” English.

```
Result result = x.from(1).to(2).withLabel("asdf").result();
```

Calling a web service

```
String target = "http://www.example.com/test";

Client client = ClientBuilder.newClient();

String result =
    client.target(target)
          .request(MediaType.APPLICATION_JSON)
          .get(String.class);

// handle result

client.close();
```

Get a client:
Client client = ClientBuilder.newClient();

Set the target to the web service URL:
client.target(target)

Create a request (setting the desired content type):
.request(MediaType.APPLICATION_JSON)

Invoke a HTTP GET request and returning an object of type String:
.get(String.class);

Close the client when finished:
client.close();

Calling a web service

```
String target = "http://www.example.com/test";
Request request = ...;

Client client = ClientBuilder.newClient();

Result result =
    client.target(target)
          .request(MediaType.APPLICATION_JSON)
          .post(Entity.json(request),
                Result.class);

// handle result

client.close();
```

Get a client:
Client client = ClientBuilder.newClient();

Result result =

Set the target to the web service URL:

```
client.target(target)
```

Create a request (setting the desired content type):

```
.request(MediaType.APPLICATION_JSON)
```

Invoke a HTTP POST request, passing in the body of the post message a request object encoded using JSON and returning an object of type Result:

```
.post(Entity.json(request),
   Result.class);
```

Close the client when finished:
client.close();

Calling a web service

```
String target = "http://www.example.com/api/message";

Client client = ClientBuilder.newClient();

Message result = client.target(target)
                       .path("{id}")
                       .resolveTemplate("id", 1)
                       .request()
                       .get(Message.class);

// handle result

client.close();
```

Get a client:
Client client = ClientBuilder.newClient();

Set the target to the web service URL:
Message result = client.target(target)

Use a sub-path of the target URL (i.e., we're using path parameters):
.path("{id}")

Resolve the sub-path parameter (i.e., the target is now http://www.example.com/api/message/1):
.resolveTemplate("id", 1)

Create a request:
.request()

Use the HTTP GET method and convert the resulting XML or JSON into an instance of the Message class:
.get(Message.class);

Close the client when finished:

```
client.close();
```

Calling a web service

```
String target = "http://www.example.com/api/message";

Form form = new Form();
form.param("message", currentMessage);
form.param("lastSeen", String.valueOf(lastSeen));

Client client = ClientBuilder.newClient();

List<Message> result =
    client.target(target)
          .request(MediaType.APPLICATION_XML)
          .post(Entity.form(form),
                new GenericType<List<Message>>() {});

// handle result

client.close();
```

Construct the parameters using a HTML-form based submission:
Form form = new Form();
form.param("message", currentMessage);
form.param("lastSeen", String.valueOf(lastSeen));

Get a client:
Client client = ClientBuilder.newClient();

Set the target to the web service URL:
List<Message> result = client.target(target)

Create a request (setting the desired content type):
.request(MediaType.APPLICATION_XML)

Use the HTTP POST method, passing in the HTML form parameter and convert the resulting XML or JSON into a list of instances of the Message class:
.post(Entity.form(form),
new GenericType<List<Message>>() {});

Close the client when finished:

```
client.close();
```

Advanced note:

The reason for GenericType is to deal with limitations of generic types in Java.

Read the documentation for more information:
https://docs.oracle.com/javaee/7/api/javax/ws/rs/core/GenericType.html

The key thing to notice is that the code looks like this:

```
   new GenericType<List<Message>>() {}
```

and NOT like this:

```
   new GenericType<List<Message>>()
```

The extra curly-brackets {} at the end are used to create a subclass.
That subclassing is essential because Java can recover generic type information from subclasses.
It can't recover generic type information from instances.

JAX-RS (Server-side) Lifecycle

Instance per Request	`@Path("message")`
	`public class Messages {`
	`@GET`
	`public List<Message> getMessages() {`
	`// ...and so on...`

Pooled EJB	`@Stateless`
	`@Path("message")`
	`public class Messages {`
	`@GET`
	`public List<Message> getMessages() {`
	`// ...and so on...`

Session scoped	`@SessionScoped`
	`@Path("message")`
	`public class Messages {`
	`@GET`
	`public List<Message> getMessages() {`
	`// ...and so on...`

JAX-RS doesn't define a particular lifecycle.

The specification says this:

By default a new resource class instance is created for each request to that resource. First the constructor (see
Section 3.1.2) is called, then any requested dependencies are injected (see Section 3.2), then the appropriate
method (see Section 3.3) is invoked and finally the object is made available for garbage collection.

An implementation MAY offer other resource class lifecycles, mechanisms for specifying these are outside
the scope of this specification. E.g. an implementation based on an inversion-of-control framework may
support all of the lifecycle options provided by that framework.

The key point is the first sentence:

By default a new resource class instance is created for each request to that resource

On Glassfish, you can use the EJB and CDI lifecycles that are permitted by the JAX-RS specifications ("An implementation MAY offer other resource class lifecycles…").

Bonus slides

SOAP vs REST

SOAP:
- Security
- Transactions
- Reliable messaging
- Sophisticated tool-chains
- Standardization
- Description languages and schemas
- Complexity
- Single-instance by default

REST:
- Simplicity
- Not only XML
- Very flexible (an architectural style, rather than a formal protocol)
- More compatibility
- Browser-friendly
- AJAX-friendly
- Instance-per-request by default

RESTful services appear to be the primary approach to web-services development today.
For a new, public-facing web service, a RESTful API is probably most preferred by developers.

However, in existing organizations, SOAP may be preferred because of its additional features.

http://stackoverflow.com/questions/19884295/soap-vs-rest-differences
http://spf13.com/post/soap-vs-rest

Eight fallacies

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

These are listed as classic fallacies that programmers new to distributed computing are said to be prone to make.

Essentially, it is tempting to pretend that a remote procedure call is exactly the same as a local procedure call.
However, no network is perfect and no framework can solve all these problems.
Ultimately, you need to be conscious of the fact that your applications are distributed applications and you need to be prepared for failures in ways that you might not have expected.

https://blogs.oracle.com/jag/resource/Fallacies.html
http://en.wikipedia.org/wiki/Fallacies_of_Distributed_Computing

A detailed explanation:
http://www.rgoarchitects.com/Files/fallacies.pdf