RESTful Web services with JAX-RS

Paul.Sandoz@Sun.Com
Sun Microsystems
Overall Presentation Goal

Learn how to build RESTful Web services using the JAX-RS API
REST style

Set of constraints you apply to the architecture of a distributed system to induce desirable properties
Application of REST style to architecture of services that utilize Web standards: URIs, HTTP, MIME, HTML, XML, Atom, RDF, ...
Desirable properties

Composition
Loose coupling
Scalability
Serendipity
Simplicity
REST Empowers

Developers can make informed and predictable engineering decisions but...
REST is no free lunch

Developers need to understand REST and Web to GET the best out of it
JSR 311: JAX-RS

Standard Java-based annotation-driven API to HELP developers build RESTful Web services
JSR 311: JAX-RS

- Web services (applications) not APIs have the property of being RESTful, or not
- JAX-RS brings key REST data elements to the fore as Java artifacts
- Guides novice developers
- Intuitive for advanced developers
- 1.0 released in October 2008
- 1.1 to align with Java EE 6
Five open source implementations

• Apache CXF
• JBoss RESTEasy
• Jersey. (The reference implementation.)
• Restlet
• Triaxrs
REST: Uniform Interface constraint

Every service is constrained to the same interface

UNIX pipes & files, Plan9
REST: Uniform Interface constraint

For a **resource**, identified by a **URI**

A client exchanges with the **resource** via **HTTP requests** and **responses** using a **fixed set of HTTP methods**

One or more **representations**, identified by **media types**

The contents of which **link to further resources**.
REST: Uniform Interface constraint

A client exchanges with the resource via HTTP requests and responses using a fixed set of HTTP methods.

One or more representations, identified by media types, link to further resources.

HTML with GET and HTML forms with POST for:
- Searching
- Buying books
- Listening to music
- Paying tax
- Social networking
- Deploying applications to the “cloud”
JAX-RS Example

```java
@Path("widgets/{id}"")
@Produces("application/widgets+xml")
@Consumes("application/widgets+xml")
public class WidgetResource {

    private Widget w;

    public WidgetResource(@PathParam("id") String id) {
        this.w = locateRecord(id);
    }

    @GET
    Widget getWidget() {
        return w;
    }

    @PUT
    Widget updateWidget(Widget update) {
        w = processUpdate(update);
        return w;
    }
}
```
A resource is a Java class

```java
@Path("widgets/{id}\")
@Produces("application/widgets+xml")
@Consumes("application/widgets+xml")
public class WidgetResource {

    private Widget w;

    public WidgetResource(@PathParam("id") String id) {
        this.w = locateRecord(id);
    }

    @GET
    Widget getWidget() {
        return w;
    }

    @PUT
    Widget updateWidget(Widget update) {
        w = processUpdate(update);
        return w;
    }
}
```
@Path("widgets/{id}\")
@Produces("application/widgets+xml")
@Consumes("application/widgets+xml")
public class WidgetResource {

    private Widget w;

    public WidgetResource(@PathParam("id") String id) {
        this.w = locateRecord(id);
    }

    @GET
    Widget getWidget() {
        return w;
    }

    @PUT
    Widget updateWidget(Widget update) {
        w = processUpdate(update);
        return w;
    }
}
... using a fixed set of methods

```java
@Path("widgets/{id}")
@Produces("application/widgets+xml")
@Consumes("application/widgets+xml")
public class WidgetResource {
    private Widget w;
    
    public WidgetResource(@PathParam("id") String id) {
        this.w = locateRecord(id);
    }

    @GET
    Widget getWidget() {
        return w;
    }

    @PUT
    Widget updateWidget(Widget update) {
        w = processUpdate(update);
        return w;
    }
}
```
@Path("widgets/{id}")
@Produces("application/widgets+xml")
@Consumes("application/widgets+xml")
public class WidgetResource {

    private Widget w;

    public WidgetResource(@PathParam("id") String id) {
        this.w = locateRecord(id);
    }

    @GET
    Widget getWidget() {
        return w;
    }

    @PUT
    Widget updateWidget(Widget update) {
        w = processUpdate(update);
        return w;
    }
}
@Path("widgets/{id}")
@Produces("application/widgets+xml")
@Consumes("application/widgets+xml")
public class WidgetResource {

    private Widget w;

    public WidgetResource(@PathParam("id") String id) {
        this.w = locateRecord(id);
    }

    @GET
    Widget getWidget() {
        return w;
    }

    @PUT
    Widget updateWidget(Widget update) {
        w = processUpdate(update);
        return w;
    }
}
• Matching URLs
  • With @Path and @PathParam
• Consuming POST requests
  • With @POST and @FormParam
• Producing and consuming with JAXB
• Content negotiation with @Consumes and @Produces
• Building URIs with UriBuilder
• Building responses with Response
Advanced features

• Add your own Java types with `MessageBodyReader` and `MessageBodyWriter`

• Map exceptions to `Response` with `ExceptionMapper`
Alignment with Java EE 6

• 3.1 EJBs as resource classes
  • No interface required
  • In the war file

• More portable deployment with Servlet 3.0

• WebBeans 1.0 for a component model: life-cycle and injection
Concluding statement

Have a play: be actively RESTful
Playing with JAX-RS using Jersey

- GET started
- GET the JAX-RS overview guide
- GET the dependencies
- GET the samples
- Blogs: Paul, Marc, Jakub, Aquarium
- JAX-RS API, JAX-RS specification, Jersey API
- Jersey available
- 1.0 and 1.0.1: Glassfish v2/v3 update centre
- 1.0: NetBeans 6.5
Thanks for your attention!

Paul.Sandoz@Sun.Com
users@jersey.dev.java.net
http://blogs.sun.com/sandoz