BUILDING REST WITH JERSEY

banuprakashc@yahoo.co.in

Agenda

- Understand REST
- Building REST using Jersey SE environment
- Building REST using Servlet 2.x and 3.x environment
- Understand HTTP verbs
- Understand produces and consumes
- Understand how to test REST
- Understand different media representation
- Customize JSON using ContextResolver
- MessageBodyWriters and Readers

The world before REST

- Many different ‘standards’:
  - RMI, SOAP, Corba, DCOM
- From many different parties:
  - Sun, Microsoft, IBM, OASIS, OMG
- Caused many problems:
  - Bad interoperability.
  - Reinvent the wheel.
  - Vendor ‘lock---in’.

And then came REST!

"Representational State Transfer (REST) is a style of software architecture for distributed hypermedia systems such as the World Wide Web"

REST is based on a set of constraints

1. Client-server
   - Separate clients and servers.
2. Stateless server
   - Each request from a client contains all the information necessary to service the request.
3. Cacheable
   - Clients can cache responses, responses must indicate if this is allowed.
4. Uniform interface
   - There is a uniform interface between clients and servers.
5. Layered System
   - Must allow concepts such as load balancers, proxies and firewalls.
6. Code-On-Demand (optional)
   - Client can request code from server and execute it.

Resources

- Fundamental to REST is the concept of resource.
- A resource is anything that can be accessed or manipulated.
- Examples of resources include “videos,” “blog entries,” “user profiles,” “images,” and even tangible things such as persons or devices.
- Resources are typically related to other resources.
  - For example, in an ecommerce application, a customer can place an order for any number of products. In this scenario, the product resources are related to the corresponding order resource. It is also possible for a resource to be grouped into collections. Using the same ecommerce example, “orders” is a collection of individual “order” resources.

Representation

- RESTful resources are abstract entities.
- The data and metadata that make a RESTful resource needs to be serialized into a representation before it gets sent to a client.
- This representation can be viewed as a *snapshot* of a resource's state at a given point in time.
- Now, when a developer writing a native mobile application requests product details, the ecommerce application might return those details in XML or JSON format.
- In both scenarios, the clients didn't interact with the actual resource—the database record-holding product details. Instead, they dealt with its representation.

XML and JSON

- Favor JSON support as the default, but unless the costs of offering both JSON and XML are staggering, offer them both.
- Ideally, let consumers switch between them by just changing an extension from .xml to .json

HTTP Verbs

- ***GET***
- The HTTP GET method is used to retrieve (or read) a representation of a resource. In the “happy” (or non-error) path, GET returns a representation in XML or JSON and an HTTP response code of 200 (OK). In an error case, it most often returns a 404 (NOT FOUND) or 400 (BAD REQUEST).
- Examples:
  - *GET http://www.example.com/customers/12345*
  - *GET http://www.example.com/customers/12345/orders*
- According to the design of the HTTP specification, GET (along with HEAD) requests are used only to read data and not change it. Therefore, when used this way, they are considered *safe*

HTTP Verbs

- ***PUT***
  - PUT is most-often utilized for update capabilities, PUT-ing to a known resource URI with the request body containing the newly-updated representation of the original resource.
  - However, PUT can also be used to create a resource in the case where the resource ID is chosen by the client instead of by the server. In other words, if the PUT is to a URI that contains the value of a non-existent resource ID. Again, the request body contains a resource representation
  - Examples:
    - *PUT http://www.example.com/customers/12345*
    - *PUT http://www.example.com/customers/12345/orders/98765*
    - *PUT http://www.example.com/buckets/secret_stuff*
  - On successful update, return 200 (or 204 if not returning any content in the body) from a PUT. If using PUT for create, return HTTP status 201 on successful creation

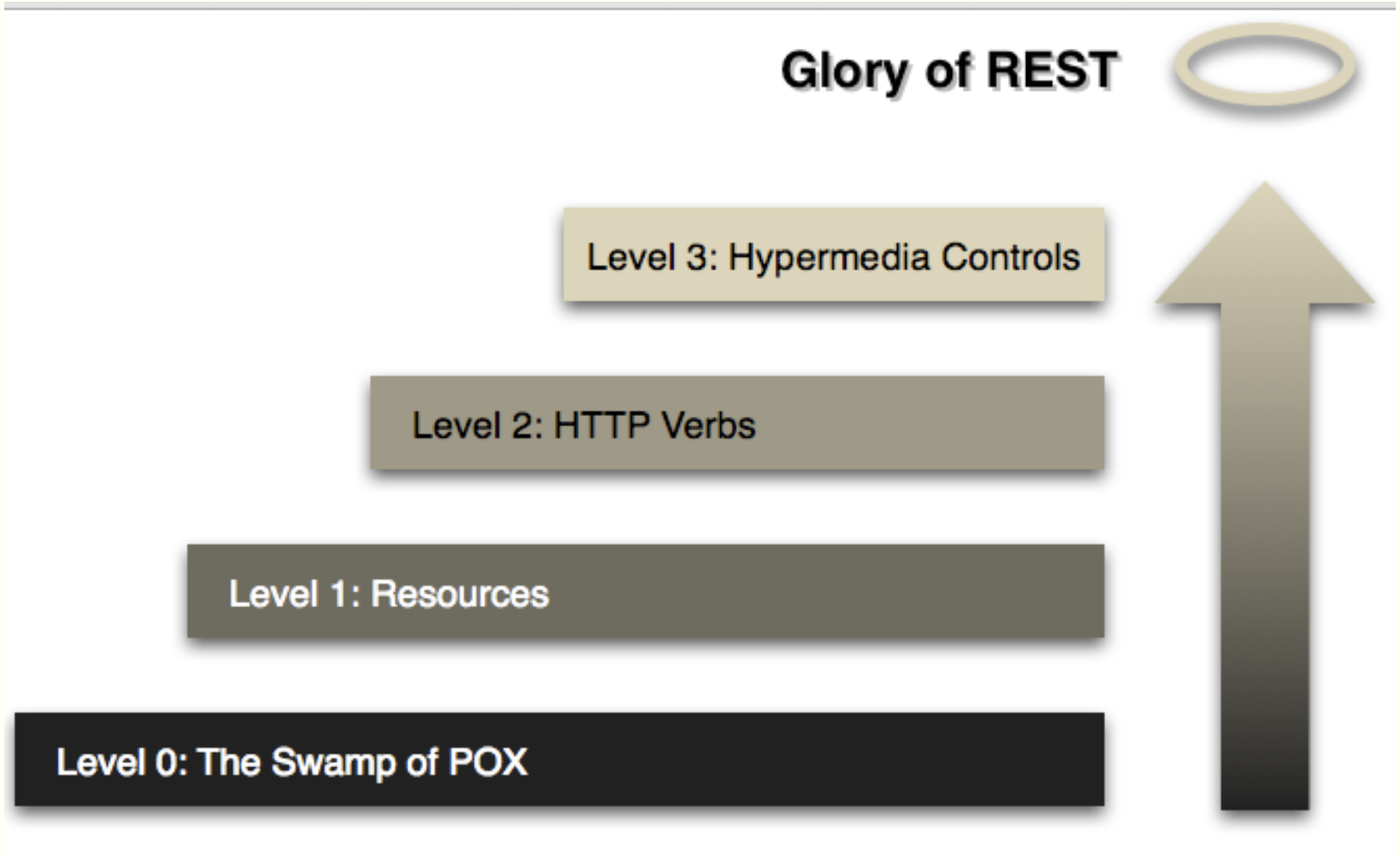
HTTP Verbs

- ***POST***
  - The POST verb is most-often utilized for creation of new resources. In particular, it's used to create *subordinate* resources. That is, subordinate to some other (e.g. parent) resource. In other words, when creating a new resource, POST to the parent and the service takes care of associating the new resource with the parent, assigning an ID (new resource URI), etc.
  - Examples:
    - *POST http://www.example.com/customers*
    - *POST http://www.example.com/customers/12345/orders*
  - On successful creation, return HTTP status 201, returning a *Location* header with a link to the newly created resource with the 201 HTTP status.
  - POST is neither *safe* or *idempotent*.

HTTP Verbs

- ***DELETE***
  - DELETE is pretty easy to understand. It is used to delete a resource identified by a URI.
  - Examples:
    - *DELETE http://www.example.com/customers/12345*
    - *DELETE http://www.example.com/customers/12345/orders*
    - *DELETE http://www.example.com/buckets/sample*
  - On successful deletion, return HTTP status 200 (OK) along with a response body, perhaps the representation of the deleted item (often demands too much bandwidth), or a wrapped. Either that or return HTTP status 204 (NO CONTENT) with no response body.

Richardson Maturity Model



Richardson Maturity Model

- Level 0:
  - the Swamp of POX (plain old XML) - at this level, we're just using HTTP as a transport.
  - You could call SOAP a Level 0 technology. It uses HTTP, but as a transport.
  - It's worth mentioning that you could also use SOAP on top of something like JMS with no HTTP at all. SOAP, thus, is not RESTful. It's only just HTTP-aware.
- Level 1:
  - Resources - at this level, a service might use HTTP URIs to distinguish between nouns, or entities, in the system.
  - For example, you might route requests to /customers, /users, etc.
  - XML-RPC is an example of a Level 1 technology: it uses HTTP, and it can use URIs to distinguish endpoints.
  - Ultimately, though, XML-RPC is not RESTful: it's using HTTP as a transport for something else (remote procedure calls).

Richardson Maturity Model

- Level 2:
  - HTTP Verbs
  - At this level, services take advantage of native HTTP qualities like headers, status codes and distinct URIs
- Level 3:
  - Hypermedia Controls
  - HATEOAS ("HATEOAS" -> "Hypermedia as the Engine of Application State") design pattern. Hypermedia promotes service longevity by decoupling the consumer of a service from intimate knowledge of that service’s surface area and topology.
  - It describes REST services.
  - The service can answer questions about what to call, and when.

Jersey RESTful Web Services framework

- Jersey RESTful Web Services framework is open source, production quality, framework for developing RESTful Web Services in Java that provides support for JAX-RS APIs

- Jersey framework is more than the JAX-RS Reference Implementation. Jersey provides it's own API that extend the JAX-RS toolkit with additional features and utilities to further simplify RESTful service and client development.

Application Deployment and Runtime Environments

- Jersey supports wide range of server environments from lightweight http containers up to full-fledged Java EE servers.
- The way how the application is published depends on whether the application shall run in a Java SE environment or within a container

Setting up Maven for JavaSE

- Dependencies

```
<properties>
    <jersey.version>2.14</jersey.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.glassfish.jersey</groupId>
            <artifactId>jersey-bom</artifactId>
            <version>${jersey.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

<dependencies>
    <dependency>
        <groupId>org.glassfish.jersey.containers</groupId>
        <artifactId>jersey-container-grizzly2-http</artifactId>
    </dependency>
```

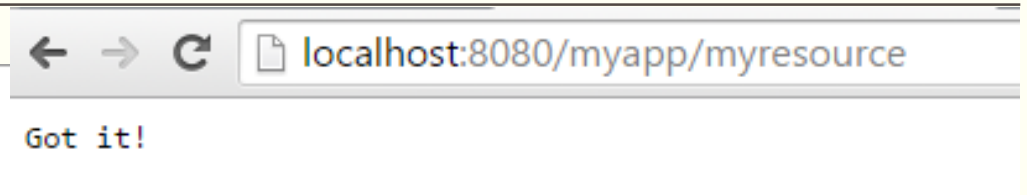
Setting up Maven for JavaSE

- mvn exec:java

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>2.5.1</version>
            <inherited>true</inherited>
            <configuration>
                <source>1.7</source>
                <target>1.7</target>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>exec-maven-plugin</artifactId>
            <version>1.2.1</version>
            <executions>
                <execution>
                    <goals>
                        <goal>java</goal>
                    </goals>
                </execution>
            </executions>
            <configuration>
                <mainClass>com.sample.Main</mainClass>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Resource

```java
package com.sample.service;

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

/**
 * @author Banu Prakash
 * Root resource (exposed at "myresource" path)
 */
@Path("myresource")
public class MyResource {

    /**
     * Method handling HTTP GET requests. The returned object will be sent
     * to the client as "text/plain" media type.
     *
     * @return String that will be returned as a text/plain response.
     */
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String getIt() {
        return "Got it!";
    }
}
```



Start Grizzly HTTP server exposing JAX-RS

```java
/**
 * @author Banu Prakash
 * Main class.
 */
public class Main {
    // Base URI the Grizzly HTTP server will listen on
    public static final String BASE_URI = "http://localhost:8080/myapp/";

     * Starts Grizzly HTTP server exposing JAX-RS resources defined in this application.
    public static HttpServer startServer() {
        // create a resource config that scans for JAX-RS resources and providers
        // in com.sample.service package
        final ResourceConfig rc = new ResourceConfig().packages("com.sample.service");

        // create and start a new instance of grizzly http server
        // exposing the Jersey application at BASE_URI
        return GrizzlyHttpServerFactory.createHttpServer(URI.create(BASE_URI), rc);
    }

    public static void main(String[] args) throws IOException {
        final HttpServer server = startServer();
        System.out.println(String.format("Jersey app started with WADL available at "
                + "%sapplication.wadl\nHit enter to stop it...", BASE_URI));
        System.in.read();
        server.stop();
    }
}
```

Deployment Environments

JDK HttpServer

```
URI baseUri = UriBuilder.fromUri("http://localhost:9998").build();
ResourceConfig config = new ResourceConfig(new MyApplication().getClasses());
HttpServer server =
        JdkHttpServerFactory.createHttpServer(baseUri, config);
```

GrizzlyHttpServer

```
HttpServer server = GrizzlyHttpServerFactory
        .createHttpServer(baseUri, config);
```

Jetty HTTP Server

```
Server server = JettyHttpContainerFactory.createServer(baseUri, config);
```

JAX-RS Application Model

- JAX-RS provides a deployment agnostic abstract class Application for declaring root resource and provider classes, and root resource and provider singleton instances.
- A Web service may extend this class to declare root resource and provider classes

```
public class MyApplication extends Application {

    @Override

    public Set<Class<?>> getClasses() {
        Set<Class<?>> s = new HashSet<Class<?>>();
        s.add(CustomerResource.class);
        s.add(ProductResource.class);
        return s;

    }

}
```

ResourceConfig - Jersey's own implementations of Application class

- Compared to Application, the ResourceConfig provides advanced capabilities to simplify registration of JAX-RS components, such as scanning for root resource and provider classes in a provided classpath or a in a set of package names etc.

```
public class MyApplication extends ResourceConfig {

    public MyApplication() {

        packages(“com.banu.services;com.banu.resources");

    }

}
```

Jersey service provider contract (SPI) classes.

- Registering SPI implementations using ResourceConfig subclass

```
public class MyApplication extends ResourceConfig {
    public MyApplication() {
            register(org.glassfish.jersey.server.validation.ValidationFeature.class);
            register(org.glassfish.jersey.server.spring.SpringComponentProvider.class);
            register(org.glassfish.jersey.grizzly2.httpserver.GrizzlyHttpContainerProvider
            .class);
 }
}
```

Testing

- Test
  - mvn test

```java
public class MyResourceTest {

    private HttpServer server;
    private WebTarget target;

    @Before
    public void setUp() throws Exception {
        // start the server
        server = Main.startServer();
        // create the client
        Client c = ClientBuilder.newClient();
        target = c.target(Main.BASE_URI);
    }

    @After
    public void tearDown() throws Exception {
        server.stop();
    }

    /**
     * Test to see that the message "Got it!" is sent in the response.
     */
    @Test
    public void testGetIt() {
        String responseMsg = target.path("myresource").request().get(String.class);
        assertEquals("Got it!", responseMsg);
    }
}
```

Servlet-based Deployment

- Servlet 2.x Container
  - In Servlet 2.5 environment, you have to explicitly declare the Jersey container Servlet in your Web application's web.xml deployment descriptor file.

```
<web-app>
    <servlet>
        <servlet-name>MyApplication</servlet-name>
        <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
        <init-param>
            ...
        </init-param>
    </servlet>
    ...
    <servlet-mapping>
        <servlet-name>MyApplication</servlet-name>
        <url-pattern>/myApp/*</url-pattern>
    </servlet-mapping>
    ...
</web-app>
```

Servlet-based Deployment

- Servlet 2.x Container
- Configuring Jersey container Servlet or Filter to use custom Application subclass

```
<init-param>
    <param-name>javax.ws.rs.Application</param-name>
    <param-value>com.banu.resource.MyApplication</param-value>
</init-param
```

Servlet-based Deployment

- Servlet 2.x Container
- Jersey package scanning

```
<init-param>
    <param-name>jersey.config.server.provider.packages</param-name>
    <param-value>com.banu.resources, com.sample.service</param-value>
</init-param>
<init-param>
    <param-name>jersey.config.server.provider.scanning.recursive</param-name>
    <param-value>false</param-value>
</init-param>
```

Selecting concrete resource and provider classes

```
<param-name>jersey.config.server.provider.classnames</param-name>
```

Servlet-based Deployment

- Servlet 2.x Container using Filter

```
<web-app>
    <filter>
        <filter-name>MyApplication</filter-name>
        <filter-class>org.glassfish.jersey.servlet.ServletContainer</filter-class>
    </filter>
  <filter-mapping>
        <filter-name>MyApplication</filter-name>
        <url-pattern>/myApp/*</url-pattern>
    </filter-mapping>
</web-app>
```

Servlet-based Deployment

- Servlet 3.x Container [Descriptor-less deployment]

```
@ApplicationPath("resources")
public class MyApplication extends ResourceConfig {
    public MyApplication() {
        packages(“com.banu.resources;com.banu.services");
    }
}
```

Representations and Java Types

- Following is a short listing of the Java types that are supported out of the box with respect to supported media type:
- All media types (*/*)
  - byte[]
  - java.lang.String
  - java.io.Reader (inbound only)
  - java.io.File
  - javax.activation.DataSource
  - javax.ws.rs.core.StreamingOutput (outbound only)
- XML media types (text/xml, application/xml and application/...+xml)
  - javax.xml.transform.Source
  - javax.xml.bind.JAXBElement
  - Application supplied JAXB classes (types annotated with @XmlRootElement or@XmlType)
- Form content (application/x-www-form-urlencoded)
  - MultivaluedMap<String,String>
- Plain text (text/plain)
  - java.lang.Boolean
  - java.lang.Character
  - java.lang.Number

JSON

- Jersey integrates with the following modules to provide JSON support:
  - MOXy
    - JSON binding support via MOXy is a default and preferred way of supporting JSON binding in your Jersey applications since Jersey 2.0.
    - When JSON MOXy module is on the class-path, Jersey will automatically discover the module and seamlessly enable JSON binding support via MOXy in your applications.
  - Java API for JSON Processing (JSON-P)
  - Jackson
  - Jettison

Adding Moxy dependencies

- **Default Jersey Entity Providers is available for** XML
  - JAXBElement (text/xml, application/xml and media types of the form application/*+xml)

- For JSON add dependencies

```
<dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-moxy</artifactId>
</dependency>
```

JAXB,XML and JSON

```java
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {

    private int id;

    @XmlElement(name="first-name")
    private String firstName;

    @XmlElement(name="last-name")
    private String lastName;

    private Address address;

    @XmlElement(name="phone-number")
    private List<PhoneNumber> phoneNumbers;

}
```

```java
@XmlAccessorType(XmlAccessType.FIELD)
public class Address {

    private String street;

}
```

```java
@XmlAccessorType(XmlAccessType.FIELD)
public class PhoneNumber {

    @XmlAttribute
    private String type;

    @XmlValue
    private String number;

}
```

XML

```xml
<customer>
    <id>123</id>
    <first-name>Jane</first-name>
    <last-name>Doe</last-name>
    <address>
        <street>123 A Street</street>
    </address>
    <phone-number type="work">555-1111</phone-number>
    <phone-number type="cell">555-2222</phone-number>
</customer>
```

JSON

```json
{
   "customer": {
      "id": 123,
      "first-name": "Jane",
      "last-name": "Doe",
      "address": {
         "street": "123 A Street"
      },
      "phone-number": [
         {
            "@type": "work",
            "$": "555-1111"
         },
         {
            "@type": "cell",
            "$": "555-2222"
         }
      ]
   }
}
```

Service Example

```
package com.banu.resource;

import java.util.ArrayList;

@Path("/bookService")
public class BookService {

    private static List<Book> list = new ArrayList<>();
    @POST
    @Consumes("application/xml")
    public Response addBook(Book book) {
        list.add(book);
        return Response.status(201).entity("Book  " + book.getId() + " added").build();
    }

    @GET
    @Produces("application/json")
    public List<Book> getBooks() {
        return list;
    }


}
```

Adding JSON support

```
package com.banu.service;

import javax.ws.rs.ApplicationPath;

import org.glassfish.jersey.moxy.json.MoxyJsonFeature;
import org.glassfish.jersey.server.ResourceConfig;

@ApplicationPath("resource")
public class MyApplication extends ResourceConfig {

    public MyApplication() {
        packages("com.banu.resource");
        // Providers - JSON.
        register(MoxyJsonFeature.class);
    }
}
```

- Execute on Tomcat 7+ or any other containers supporting 3.0

Test client

```
public static void main(String[] args) {

    Client client = ClientBuilder.newClient().register(JacksonFeature.class);

    WebTarget target = client.target("http://localhost:8080/jersey_web").path(
            "resource/bookService");

    Response response = target.request().post(
            Entity.entity(new Book(4545, "344sf", 445.55),
                    MediaType.APPLICATION_XML));
    System.out.println(response.readEntity(String.class));

    response = target.request(MediaType.APPLICATION_JSON).get();

    System.out.println(response.readEntity(List.class));
}
```

ContextResolver

```
@Provider
public class MOXyJsonContextResolver implements ContextResolver<MoxyJsonConfig> {
    private final MoxyJsonConfig config;
     public MOXyJsonContextResolver() {
         config = new MoxyJsonConfig()
            .setAttributePrefix("@")
            .property(JAXBContextProperties.JSON_WRAPPER_AS_ARRAY_NAME, true)
            .property(JAXBContextProperties.JSON_INCLUDE_ROOT, true);
    }
    @Override
    public MoxyJsonConfig getContext(Class<?> objectType) {
        return config;
    }
}

With Context-Resolver
{"customer":{"id":1,"name":"Raj","phoneNumbers":[{"@type":"mobile","value":"343434"},
{"@type":"home","value":"12345"}]}}

Without:
{"id":1,"name":"Raj","phoneNumbers":{"phoneNumber":[{"type":"mobile","value":"343434"},
{"type":"home","value":"12345"}]}}
```

ContextResolver complete listing

```
@ApplicationPath("resource")
public class MyApplication extends ResourceConfig {

    public MyApplication() {
        packages("com.banu.resource");
        // Providers - JSON.
        register(MoxyJsonFeature.class);
        register(JsonConfiguration.class);
    }

    @Provider
    public static class JsonConfiguration implements
            ContextResolver<MoxyJsonConfig> {

        @Override
        public MoxyJsonConfig getContext(final Class<?> type) {
            final MoxyJsonConfig config = new MoxyJsonConfig();
            config.setFormattedOutput(true);
            config.setAttributePrefix("@")
                    .setValueWrapper("value")
                    .property(JAXBContextProperties.JSON_INCLUDE_ROOT, true)
                    .property(JAXBContextProperties.JSON_WRAPPER_AS_ARRAY_NAME,
                            true);
            return config;
        }
    }
}
```

JAX-RS: POST form data

```
@POST
  @Produces(MediaType.TEXT_HTML)
  @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
  public void addEmployee(
            @FormParam("eid") int eid,
            @FormParam("name") String ename,
             @Context HttpServletResponse response) throws IOException{
}
```

Jersey Client

POST form data

```
Form f = new Form();
f.param("id", "77");
f.param("name", "Electronics");
f.param("description", "FM Radio, Transistors etc.");
Entity<Form> e = Entity.entity(
                    f, MediaType.APPLICATION_FORM_URLENCODED);
```

Response r=

```
ClientBuilder.newClient()
.target("http://localhost:8080/restapp/category").request().post(e);
```

JAX-RS: POST form data using

- MultivaluedMap

```
@POST
@Consumes("application/x-www-form-urlencoded")
@Path("/form")
public Response getFormParams(MultivaluedMap<String, String> formParams) {
    StringBuilder builder = new StringBuilder("/Context results");
    builder.append(" ");
    for(Entry<String, List<String>> entry: formParams.entrySet()) {
        builder.append("Key :" + entry.getKey());
        builder.append("values :");
        for(String s : entry.getValue()) {
            builder.append(s);
        }
    }
    return Response.status(200).entity(builder.toString()).build();
}
```

JAX-RS: POST form data

- BeanParam

```java
public class Product {

    @FormParam("id")
    private int id;
    @FormParam("name")
    private String name;
    @FormParam("price")
    private double price;
```

```java
@POST
@Consumes("application/x-www-form-urlencoded")
@Path("/formBean")
public Response createProduct(@BeanParam Product product) {
    StringBuilder builder = new StringBuilder("/formBean results");
    builder.append(" ");
    builder.append(product.getId() + "," + product.getName() +"," + product.getPrice());
    return Response.status(200).entity(builder.toString()).build();
}
```

JAX-RS:

- HttpHeaders

```
@GET
@Path("/context")
public Response get(@Context HttpHeaders hh) {
    MultivaluedMap<String, String> headerParams = hh.getRequestHeaders();
   // Map<String, Cookie> pathParams = hh.getCookies();
    StringBuilder builder = new StringBuilder("/Context results");
    builder.append(" ");
    for(Entry<String, List<String>> entry: headerParams.entrySet()) {
        builder.append("Key :" + entry.getKey());
        builder.append("values :");
        for(String s : entry.getValue()) {
            builder.append(s);
        }
    }
    return Response.status(200).entity(builder.toString()).build();
}
```

JAX-RS

- QueryParam

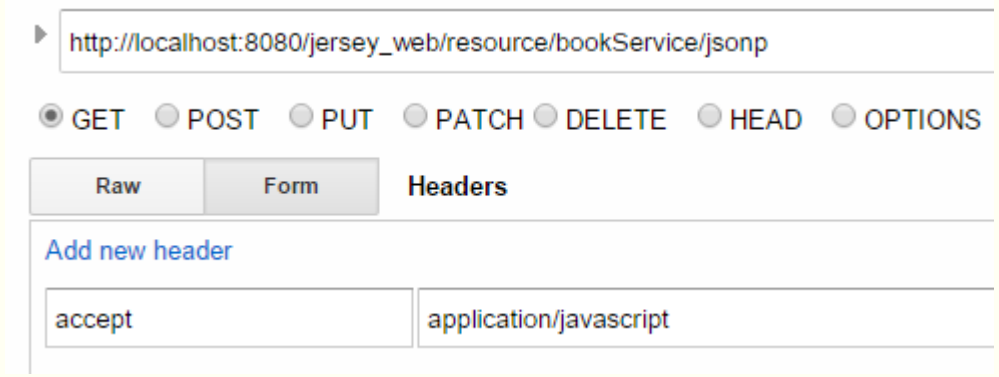
```
//http://localhost:8080/FirstJerseyEx/resources/product/query/?id=2
@GET
@Path("/query")
@Produces(MediaType.APPLICATION_JSON)
public Product getProductUsingQuery(@DefaultValue("1") @QueryParam("id") int id) {
    System.out.println("getProductUsingQuery(@QueryParam(\"id\") int id) called");
    if (id <= 0) {
        throw new NotFoundException(Response
                .status(Response.Status.NOT_FOUND)
                .entity("Product, " + id + ", is not found")
                .build());
    }

    return ProductRepo.getProduct(id);
}
```

JSONP [**JSON with Padding Support]**

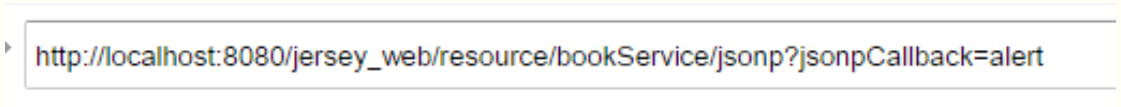
- JSONP is necessary when you need to provide cross domain support.
- The main difference between JSON and JSONP is that JSONP returns its datastructure wrapped in a piece of javascript which will be evaluated by the browser.
- Jersey provides out-of-the-box support for JSONP - JSON with padding.
- User's request has to contain Accept header with one of the JavaScript media types defined:
  - Acceptable media types compatible with @JSONP are:
    - application/javascript, application/x-javascript, application/ecmascript, text/javascript, text/x-javascript, text/ecmascript, text/jscript.

JSONP example

```java
@GET
@Path("/jsonp")
@Produces({"application/json", "application/javascript"})
@JSONP(callback = "eval", queryParam = "jsonpCallback")
public Book getSimpleJSONP() {
    return new Book(1, "JSONP Book", 100.00);
}
```




```
eval({
   "book" : {
      "@id" : 1,
      "title" : "JSONP Book",
      "price" : 100.0
   }
})
```




```
alert({
   "book" : {
      "@id" : 1,
      "title" : "JSONP Book",
      "price" : 100.0
   }
})
```

MessageBodyWriter

- **Evolve your REST representation with a MessageBodyWriter**
  - it's one of the most important building blocks if you plan on evolving your representation of your REST resources.
  - A JAX-RS implementation already supports a number of conversions from types to different representation. The classes responsible for the conversions are all implemented as MessageBodyWriter's and Readers

- The REST style
  - Before looking at the implementation we should have a mechanism to make a distinction between the different representations. This can be done by the putting our version number in the MIME type. MIME types allow for a vendor namespace where we can create all our private representations. This is how the MIME type could look like:

Accept	application/vnd.banu.labs.app.v1.company+xml

  - You are free to define the structure between the **application/vnd.** and the **+xml**.

MessageBodyWriter

- Add new methods to handle the new conversions, and add the MIME type to the @Produces annotation, in your resource class.

```
@GET
@Path("{id}")
@Produces({"application/vnd.banu.labs.app.v1.company+xml",MediaType.APPLICATION_JSON})
public Employee getEmployeeCustom(@PathParam("id")int  id ) {
  return new Employee(1,"Banuprakash");
}
```

MessageBodyWriter

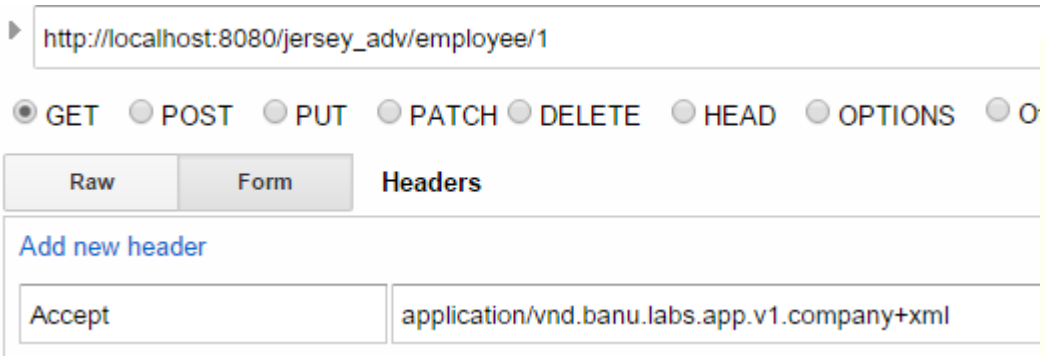
```java
@Provider
public class CustomMessageBodyWriter implements MessageBodyWriter<Object> {

    /*
     * The getSize is called before the writeTo method, to determine the size
     * upfront. If you don't know the size before the conversion, just return
     * -1.
     */
    @Override
    public long getSize(Object arg0, Class<?> arg1, Type arg2,
            Annotation[] arg3, MediaType arg4) {
        return -1;
    }

    /*
     * Here you write the code, to detect if this is the correct writer for the
     * object
     */
    @Override
    public boolean isWriteable(Class<?> clazz, Type type, Annotation[] ann,
            MediaType mediaType) {
        System.out.println("isWritable...");
        if ("com.sample.entity".equals(clazz.getPackage().getName())
                && mediaType.getType().equals("application")
                && mediaType.getSubtype().matches(
                        "vnd\\.banu\\.labs\\.app\\.v1\\..*\\+xml")) {
            return true;
        }
        return false;
    }
```

MessageBodyWriter

```
@Override
public void writeTo(Object object, Class<?> clazz, Type type,
        Annotation[] ann, MediaType mediaType,
        MultivaluedMap<String, Object> map, OutputStream out)
        throws IOException, WebApplicationException {
    try {
        JAXBContext context = JAXBContext.newInstance(Employee.class);
        context.createMarshaller().marshal(object, out);
        context.createMarshaller().marshal(object, System.out);
    } catch (JAXBException e) {
        e.printStackTrace();
    }
}
```

MessageBodyWriter register

```java
@ApplicationPath("/")
public class MyApplication extends ResourceConfig {


    public MyApplication() {
        packages("com.sample.resource");
        // Providers - JSON.
        register(MoxyJsonFeature.class);
        register(JsonConfiguration.class);
        register(new CustomMessageBodyWriter());
    }
```




```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<emp id="1">
   <name>Banuprakash</name>
</emp>
```

MessageBodyReader

- In order to de-serialize the entity of MyBean on the server or the client, we can implement a custom MessageBodyReader<T>
- MessageBodyReader.isReadable
  - It defines the method isReadable() which has a very simliar meaning as method isWriteable() in MessageBodyWriter<T>. The method returns true if it is able to de-serialize the given type.
- MessageBodyReader.readFrom
  - The readForm() method provides a handle to the entity input stream from which the entity bytes should be read and de-serialized into a Java entity which is then returned from the method.

MessageBodyWorkers

- In case you need to directly work with JAX-RS entity providers, you would need to perform quite a lot of steps. You would need to choose the appropriate MessageBodyWriter<T> based on the type, media type and other parameters.
- To remove this burden from developers, Jersey exposes a proprietary public API that simplifies the manipulation of entity providers.
- The API is defined by MessageBodyWorkers interface and Jersey provides an implementation that can be injected using the @Context injection annotation.
- The interface declares methods for selection of most appropriate MessageBodyReader<T> and MessageBodyWriter<T> based on the rules defined in JAX-RS spec, methods for writing and reading entity

MessageBodyWorkers

```
@Context
private MessageBodyWorkers workers;

@GET
@Produces(MediaType.APPLICATION_XML)
public String getEmployeeXML() {
    Employee emp = new Employee(2, "Kavitha");
    ByteArrayOutputStream baos = new ByteArrayOutputStream();

    // get most appropriate MBW
    final MessageBodyWriter<Employee> messageBodyWriter =
            workers.getMessageBodyWriter(Employee.class, Employee.class,
                    new Annotation[]{}, MediaType.APPLICATION_XML_TYPE);

    try {
        // use the MBW to serialize emp into baos
        messageBodyWriter.writeTo(emp,
            Employee.class, Employee.class, new Annotation[] {},
            MediaType.APPLICATION_XML_TYPE, new MultivaluedHashMap<String, Object>(),
            baos);
    } catch (IOException e) {
        throw new RuntimeException(
            "Error while serializing employee.", e);
    }

    final String stringXmlOutput = baos.toString();
    return stringXmlOutput;
}
```

Filters and Interceptors

- Filters can be used when you want to modify any request or response parameters like headers
- There are filters on the server side and the client side.
  - Server filters:
    - ContainerRequestFilter
    - ContainerResponseFilter
  - Client filters:
    - ClientResponseFilter
    - ClientResponseFilter

Filter example

```
public class MyReponseFilter implements ContainerResponseFilter {

    @Override
    public void filter(ContainerRequestContext request,
            ContainerResponseContext response) throws IOException {
        MultivaluedMap<String, Object> map = response.getHeaders();
        response.getHeaders().add("Powered By", "ACME :-)");
        // Cross-Origin Headers
        response.getHeaders().add("Allow-cross-origin","*");

    }

}
```

Interceptor

- Interceptors are intended to manipulate entities, via manipulating entity input/output streams
- There are two kinds of interceptors, ReaderInterceptor and WriterInterceptor.

```
public class MyInterceptor implements WriterInterceptor {

    @Override
    public void aroundWriteTo(WriterInterceptorContext wic)
            throws IOException, WebApplicationException {
        Object obj = wic.getEntity();
        if(obj instanceof String){
            String s = (String) obj;
            s = "Changed Data:" + s;
            wic.setEntity(s);
        }
        wic.proceed();
    }

}
```

Register filters and Interceptors

```
@ApplicationPath("/")
public class MyApplication extends ResourceConfig {


    public MyApplication() {
        packages("com.sample.resource");
        register(MoxyJsonFeature.class);
        register(JsonConfiguration.class);
        register(new CustomMessageBodyWriter());
        register(MyInterceptor.class);
        register(MyReponseFilter.class);
    }
```

Jersey Test Framework

- Current implementation of Jersey Test Framework supports the following set of features:
  - pre-configured client to access deployed application
  - support for multiple containers - grizzly, jdk, jetty
  - able to run against any external container
  - automated configurable traffic logging

```
<dependency>
    <groupId>org.glassfish.jersey.test-framework.providers</groupId>
    <artifactId>jersey-test-framework-provider-grizzly2</artifactId>
    <version>2.14</version>
</dependency>
```

`HttpServer` from Oracle JDK is another supported test container.

```
1  <dependency>
2      <groupId>org.glassfish.jersey.test-framework.providers</groupId>
3      <artifactId>jersey-test-framework-provider-jdk-http</artifactId>
4      <version>2.14</version>
5  </dependency>
```

JerseryTest example

```java
/**
 * @author Banu Prakash
 *
 */
public class HelloWorldResourceTest extends JerseyTest {

    @Override
    protected Application configure() {
        enable(TestProperties.LOG_TRAFFIC);
        enable(TestProperties.DUMP_ENTITY);
        return new ResourceConfig(HelloWorldResource.class,PoweredByResponseFilter.class,CORSFilter.class);
    }

    @Test
    public void testConnection() {
        Response response = target("hello").request().get();
        assertEquals(200, response.getStatus());
    }

    @Test
    public void testClientStringResponse() {
        final String hello = target("hello").request().get(String.class);
        assertEquals("Hello", hello);
    }
```

JerseryTest example

```
@Test
public void testMissingResourceNotFound() {
    Response response;
    response = target("hello").path("arbitrary").request().get();
    assertEquals(404, response.getStatus());
}

@Test
public void testGet() {
    TestBean testBean = target("hello").path("json")
            .request(MediaType.APPLICATION_JSON_TYPE).get(TestBean.class);
    assertEquals(testBean, new TestBean("a", 1));
}

@Test
public void roundTripTest() {
    TestBean testBean = target("hello")
            .path("json")
            .request(MediaType.APPLICATION_JSON_TYPE)
            .post(Entity.entity(new TestBean("Banu", 1),
                    MediaType.APPLICATION_JSON_TYPE), TestBean.class);

    assertEquals(testBean, new TestBean("Banu", 1));
}
```

JerseryTest example

```
@Test(expected=WebApplicationException.class)
public void testGetException() {
     target("hello/ex/443")
            .request(MediaType.APPLICATION_JSON_TYPE).get(TestBean.class);
     fail("Should throw exception");
}

@Test
public void testPoweredByFilter() {
    Response response = target("hello").request()
            .header("Accept", MediaType.TEXT_PLAIN).options();
    System.out.println(response.getHeaderString("X-Powered-By"));
    assertEquals("Novell :-)", response.getHeaderString("X-Powered-By"));
}
```

```
@Test
public void testInterceptor() {
    TestBean testBean = target("hello")
            .path("json").register(TestInterceptor.class)
            .request(MediaType.APPLICATION_JSON_TYPE)
            .post(Entity.entity(new TestBean("Banu", 1),
                    MediaType.APPLICATION_JSON_TYPE), TestBean.class);
        System.out.println(testBean.getName());
    assertEquals(testBean, new TestBean("Changed_item_Banu", 1));
}
```