RESTful Services in Nutshell

Based on the original slides of Michael Arnwine: Service Oriented Architecture (SOA) and “Restful” Service.

Based on the original slides of Bill Burke: REST and JAX-RS

REST Concept

- Actually only the difference is how clients access our service. Normally, a service will use SOAP, but if you build a REST service, clients will be accessing your service with a different architectural style (calls, serialization like JSON, etc.).

- REST uses some common HTTP methods to insert/delete/update/retrieve information which is below:

- **GET** - Requests a specific representation of a resource
- **PUT** - Creates or updates a resource with the supplied representation
- **DELETE** - Deletes the specified resource
- **POST** - Submits data to be processed by the identified resourc

What is REST?

- REpresentational State Transfer
  - PhD by Roy Fielding
  - The Web is the most successful application on the Internet
  - What makes the Web so successful?
- Addressable Resources
  - Every “thing” should have an ID
  - Every “thing” should have a URI
- Constrained interface
  - Use the standard methods of the protocol
  - HTTP: GET, POST, PUT, DELETE
- Resources with multiple representations
  - Different applications need different formats
  - Different platforms need different representations (XML + JSON)
- Communicate statelessly
  - Stateless application scale

- Every “thing” has a URI

Addressability

`http://sales.com/customers/323421`
`http://sales.com/customers/32341/address`

- From a URI we know
  - The protocol (How do we communicate)
  - The host/port (Where it is on network)
  - The resource path(What resource are we communicating with

Describing a URI

http://sales.com/customers/323421

/customers/{customer-id}

- Human readable URIs: Desired but not required
- URI Parameters

http://sales.com/customers?zip=49009

- Query parameters to find other resources

http://sales.com/cars/mercedes/amg/e55;color=black

- Matrix parameters to define resource attributes

Implications of a Uniform Interface

- Intuitive
  - You know what operations the resource will support
- Predictable behavior
  - GET - readonly and idempotent. Never changes the state of the resource
  - PUT - an idempotent insert or update of a resource. Idempotent because it is repeatable without side effects.
  - DELETE - resource removal and idempotent.
  - POST - non-idempotent, “anything goes” operation
- Clients, developers, admins, operations know what to expect
  - Much easier for admins to assign security roles
  - For idempotent messages, clients don’t have to worry about duplicate messages.

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.“

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.
•More human readable and testable (harder to test SOAP with just a browser).
•Don't need to use XML (well, you kind of don't have to for SOAP either but it hardly makes sense since you're already doing parsing of the envelope).
•Libraries have made SOAP (kind of) easy. But you are abstracting away a lot of redundancy underneath as I have noted. Yes, in theory, SOAP can go over other transports so as to avoid riding atop a layer doing similar things, but in reality just about all SOAP work you'll ever do is over HTTP.

REST Data Elements

- Resources and Resource Identifiers
  - Uniform Interface (GET, PUT, POST, DELETE)
  - Resource Oriented
  - Simple and simple is beautiful

HTTP	**Method**	**CRUD**	**Desc.**
POST	CREATE	Create	-
GET	RETRIEVE	Retrieve	Safe,Idempotent,Cacheable
PUT	UPDATE	Update	Idempotent
DELETE	DELETE	Delete	Idempotent

REST Core Idiologies

- Simple is better
- The web works and works well
- Some web services should follow the “way of the web”.

RESTful Services

- Resources as URI
  - Use unique URI to reference every resource on your API
- Operations as HTTP Methods
  - GET – Queries
  - POST – Queries
  - PUT, DELETE – Inset, Update and delete
- Connectedness and Discoverability
  - Like the Web, HTTP Responses contains links to other resources

REST API EXAMPLE: Delicious

URL	http://del.icio.us/api/[username]/bookmarks/	
Method	GET	
Querystring	tag=	Filter by tag
	dt=	Filter by date
	start=	The number of the first bookmark to return
	end=	The number of the last bookmark to return
Returns	200 OK & XML (delicious/bookmarks+xml)	
	401 Unauthorized	
	404 Not Found	

REST API EXAMPLE: Delicious

URL	http://del.icio.us/api/[username]/bookmarks/
Method	POST
Request Body	XML (delicious/bookmark+xml)
Returns	201 Created & Location
	401 Unauthorized
	415 Unsupported Media Type

REST API EXAMPLE: Delicious

URL	http://del.icio.us/api/[username]/bookmarks/[hash]
Method	DELETE
Returns	204 No Content
	401 Unauthorized
	404 Not Found

Designing services with a Uniform Interface

- When in doubt, define a new resource
- /orders
  - GET - list all orders
  - POST - submit a new order
- /orders/{order-id}
  - GET - get an order representation
  - PUT - update an order
  - DELETE - cancel an order
- /orders/average-sale
  - GET - calculate average sale
- /customers
  - GET - list all customers
  - POST - create a new customer
- /customers/{cust-id}
  - GET - get a customer representation
  - DELETE- remove a customer
- /customers/{cust-id}/orders
  - GET - get the orders of a customer

Resources with Multiple Representations

- HTTP Headers manage this negotiation
  - CONTENT-TYPE: specifies MIME type of message body
  - ACCEPT: comma delimited list of one or more MIME types the client would like to receive as a response
  - In the following example, the client is requesting a customer representation in either xml or json format

```
GET /customers/33323
ACCEPT: application/xml,application/json
```

- Preferences are supported and defined by HTTP specification

```
GET /customers/33323
ACCEPT: text/html;q=1.0,
        application/json;q=0.5;application/xml;q=0.7
```

What is JSON?

- JavaScript Object Notation
- Lightweight syntax for representing data
- Easier to “parse” for JavaScript client code
- Alternative to XML in AJAX applications

[{"Email":"bob@example.com","Name":"Bob"},{"Email
":"mark@example.com","Name":"Mark"},{"Email":"j
ohn@example.com","Name":"John"}]

EXAMPLES:

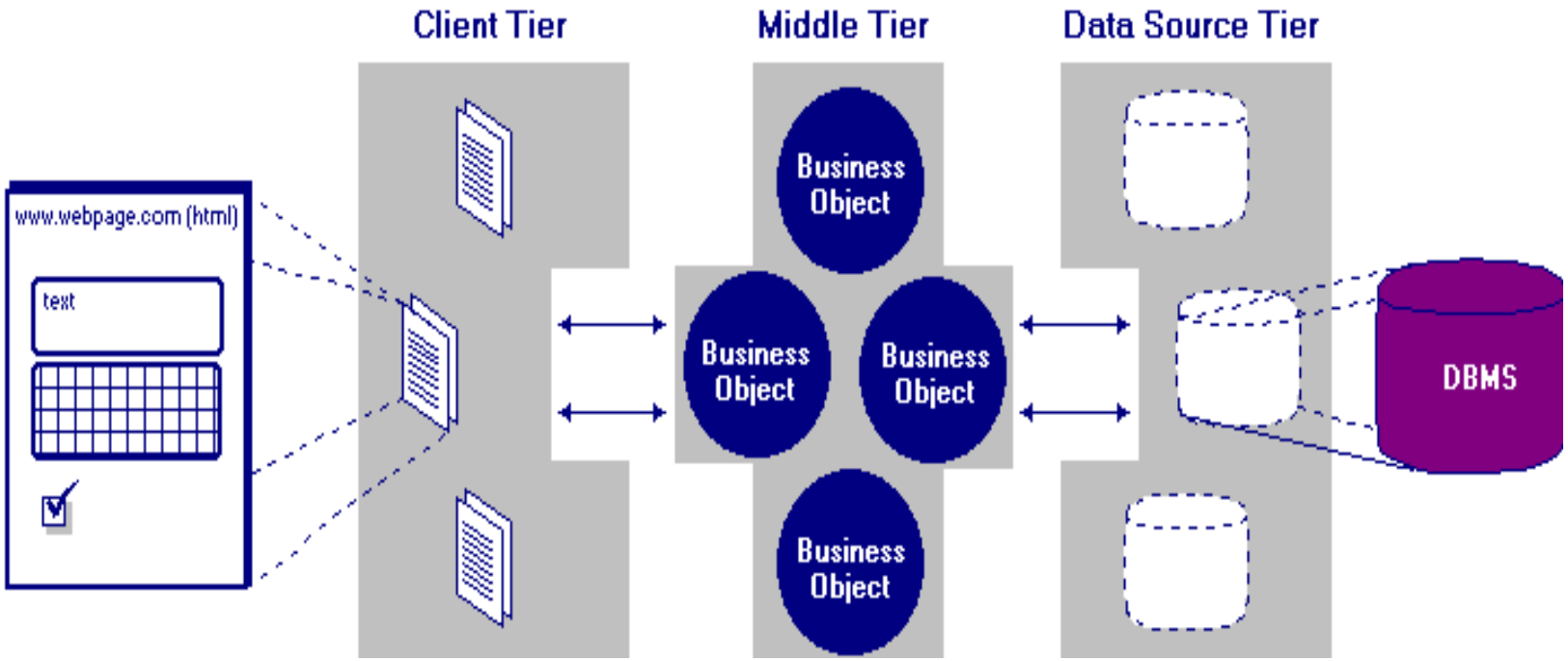
Publish and Consume REST Services

EXAMPLES of REST APIs:

Facebook Graph API
Google Custom Search API
Yahoo
… and a lot more …

n-Tiers Architecture

- SOAP and REST based web-services enable the 3-tier architecture to be extended into n-tiers.



JAX-RS

- JCP Specification
  - Lead by Sun, Marc Hadley
  - Currently in public draft (which means final draft right around the corner)
- Annotation Framework
- Dispatch URI's to specific classes and methods that can handle requests
- Allows you to map HTTP requests to method invocations
- IMO, a beautiful example of the power of parameter annotations
- Nice URI manipulation functionality

JAX-RS Annotations

- @Path
  - Defines URI mappings and templates
- @ProduceMime, @ConsumeMime
  - What MIME types does the resource produce and consume
- @GET, @POST, @DELETE, @PUT, @HEADER
  - Identifies which HTTP method the Java method is interested in

JAX-RS Parameter Annotations

- @PathParam
  - Allows you to extract URI parameters/named URI template segments
- @QueryParam
  - Access to specific parameter URI query string
- @HeaderParam
  - Access to a specific HTTP Header
- @CookieParam
  - Access to a specific cookie value
- @MatrixParam
  - Access to a specific matrix parameter
- Above annotations can automatically map HTTP request values to
  - String and primitive types
  - Class types that have a constructor that takes a String parameter
  - Class types that have a static valueOf(String val) method
  - List or Arrays of above types when there are multiple values
- @Context
  - Access to contextual information like the incoming URI

JAX-RS Resource Classes

- JAX-RS annotations are used on POJO classes
- The default component lifecycle is per-request
  - Same idea as @Stateless EJBs
- Root resources identified via @Path annotation on class

JAX-RS

```
@Path(“/orders”)
public class OrderService {

    @Path(“/{order-id}”)
    @GET
    @ProduceMime(“application/xml”)
    String getOrder(@PathParam(“order-id”) int id) {
     …
    }
}
```

Default Response Codes

- GET and PUT
  - 200 (OK)
- DELETE and POST
  - 200 (OK) if content sent back with response
  - 204 (NO CONTENT) if no content sent back

Response Object

- JAX-RS has a Response and ResponseBuilder class
  - Customize response code
  - Specify specific response headers
  - Specify redirect URLs
  - Work with variants

```
@GET
Response getOrder() {
   ResponseBuilder builder = Response.status(200);
   builder.type("text/xml")
          .header("custom-header", "33333");
   return builder.build();
}
```

RESTful Service Example:

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.core.Response;

@Path("/users")
public class UserRestService {

        @GET
        @Path("{year}/{month}/{day}")
        public Response getUserHistory(
                        @PathParam("year") int year,
                        @PathParam("month") int month,
                        @PathParam("day") int day) {

           String date = year + "/" + month + "/" + day;

           return Response.status(200)
                .entity("getUserHistory is called, year/month/day : " + date)
                .build();

        }
```

java.net.URL RESTful Client:

```
URL url = new URL("http://localhost:8080/RESTfulExample/json/product/post");
HttpURLConnection conn = (HttpURLConnection) url.openConnection();
conn.setDoOutput(true);
conn.setRequestMethod("POST");
conn.setRequestProperty("Content-Type", "application/json");

String input = "{\"qty\":100,\"name\":\"iPad 4\"}";

OutputStream os = conn.getOutputStream();
os.write(input.getBytes());
os.flush();

if (conn.getResponseCode() != HttpURLConnection.HTTP_CREATED) {
        throw new RuntimeException("Failed : HTTP error code : "
                + conn.getResponseCode());
}

BufferedReader br = new BufferedReader(new InputStreamReader(
                (conn.getInputStream())));

String output;
System.out.println("Output from Server .... \n");
while ((output = br.readLine()) != null) {
        System.out.println(output);
}

conn.disconnect();
```

Apache HttpClient:

```
DefaultHttpClient httpClient = new DefaultHttpClient();
HttpGet getRequest = new HttpGet(
        "http://localhost:8080/RESTfulExample/json/product/get");
getRequest.addHeader("accept", "application/json");

HttpResponse response = httpClient.execute(getRequest);

if (response.getStatusLine().getStatusCode() != 200) {
        throw new RuntimeException("Failed : HTTP error code : "
           + response.getStatusLine().getStatusCode());
}

BufferedReader br = new BufferedReader(
         new InputStreamReader((response.getEntity().getContent())));

String output;
System.out.println("Output from Server .... \n");
while ((output = br.readLine()) != null) {
        System.out.println(output);
}

httpClient.getConnectionManager().shutdown();
```

Jersey Client (jersey-client.jar):

```
Client client = Client.create();

WebResource webResource = client
   .resource("http://localhost:8080/RESTfulExample/rest/json/metallica/get");

ClientResponse response = webResource.accept("application/json")
   .get(ClientResponse.class);

if (response.getStatus() != 200) {
   throw new RuntimeException("Failed : HTTP error code : "
        + response.getStatus());
}

String output = response.getEntity(String.class);

System.out.println("Output from Server .... \n");
System.out.println(output);
```

JAXB Annotations:

```java
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "customer")
public class Customer {

	String name;
	int pin;

	@XmlElement
	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	@XmlAttribute
	public int getPin() {
		return pin;
	}

	public void setPin(int pin) {
		this.pin = pin;
	}

}
```

Produces:

```xml
<customer pin="value">
	<name>value</name>
</customer>
```

JAX-RS Service that Returns XML:

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import com.mkyong.Customer;

@Path("/xml/customer")
public class XMLService {

	@GET
	@Path("/{pin}")
	@Produces(MediaType.APPLICATION_XML)
	public Customer getCustomerInXML(@PathParam("pin") int pin) {

		Customer customer = new Customer();
		customer.setName("mkyong");
		customer.setPin(pin);

		return customer;

	}

}
```

OAuth2.0

OAuth 2.0 is a relatively simple protocol and a developer can integrate with Google's OAuth 2.0 endpoints without too much effort. In a nutshell, you register your application with Google, redirect a browser to a URL, parse a token from the response, and send the token to the Google API you wish to access.