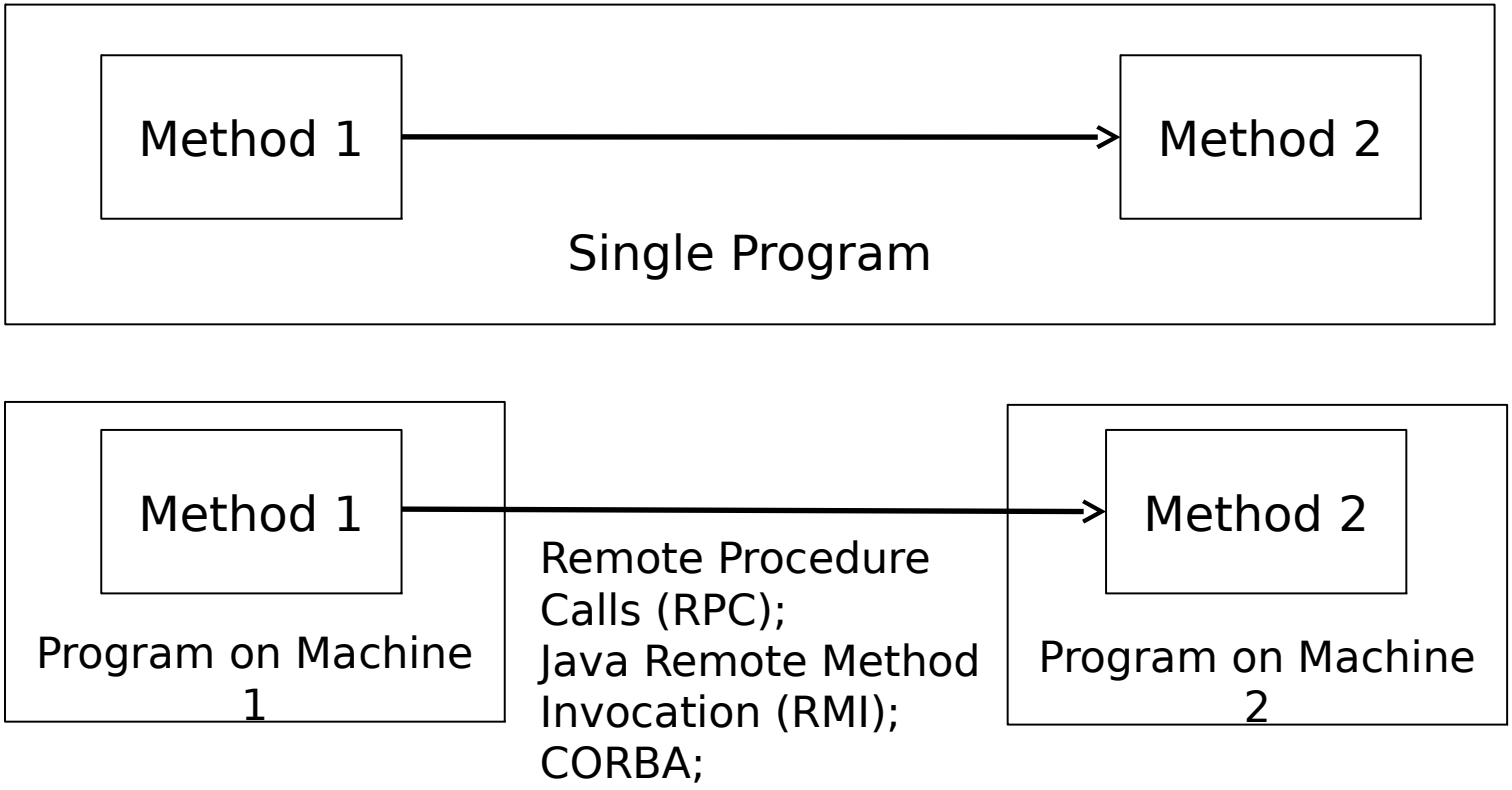
REST

Devdatta Kulkarni

REST

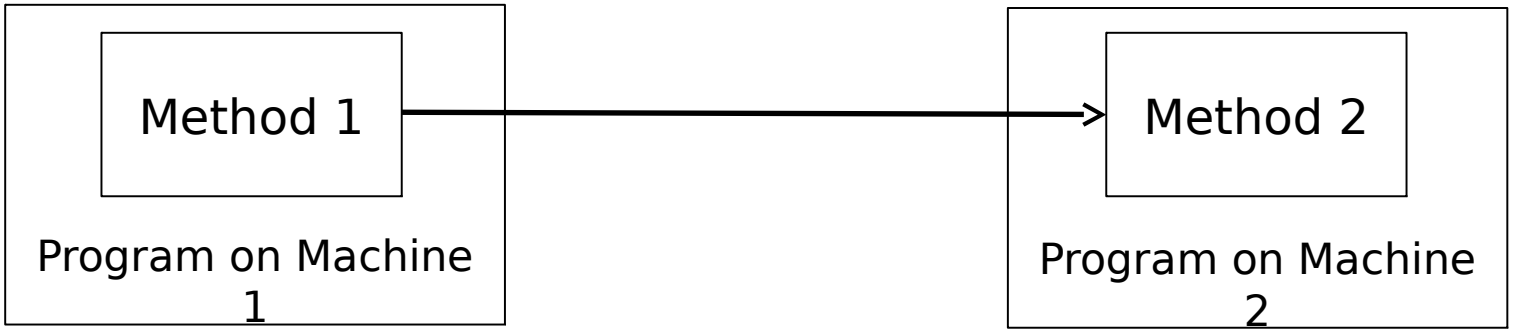
- What is REST?
  - Representational state transfer
  - A methodology for managing and manipulating web resources through their state representations
    - Web resources
      - Resources that are accessible over HTTP; typically available on the web server
    - State representations
      - Values for a resources' attributes
    - Managing and Manipulation
      - Change the resource through its state representation
    - Methodology
      - HTTP methods to achieve above goal
        - » GET, PUT, POST, HEAD, PATCH

Emergence of REST



A Classic Paper:
Implementing remote procedure calls
http://research.cs.wisc.edu/areas/os/Qual/papers/rpc.pdf
Citations: 2455

Emergence of REST



Java RMI; CORBA over WEB (SOAP)

REST:

- Keep transport mechanism HTTP (i.e. keep the WEB part of SOAP)
- Define a “uniform” interface instead of arbitrary method signatures

SOAP: Simple Object Access Protocol

REST example

- Obtaining data from the eavesdrop site:
  - We resorted to opening connection and parsing the HTML using Jsoup
  - But we would not have had to do that if the eavesdrop site had provided a REST API

REST Architectural Principles

- Addressability
- Uniform Constrained Interface
- Representation Oriented
- Communicate Statelessly
- Hypermedia As The Engine of Application State (HATEOS)

Addressability

- Every object and resource in your system is reachable through a *unique identifier*

- Example:
  - Resource: *Course 378*
    - /departments/cs/courses/cs378
  - Resource: *Assignments of CS 378*
    - /../cs/courses/cs378/assignments

Uniform Constrained Interface

- Use finite set of actions
- For HTTP these are:
  - GET
  - POST
  - PUT
  - DELETE
  - PATCH

Why Uniform Constrained Interface?

- Familiarity
  - Each resource is accessible with only the familiar set of actions
    - No need to look up method signatures
      - Great advantage over protocol such as SOAP
- Interoperability
  - HTTP is universal; Language libraries available for HTTP; no need for special client libraries

REST Action semantics

- POST (Also called as ‘create’ (C))
  - Create a new resource
- GET (Also called as ‘read’ (R))
  - Get information about a specific resource
- PUT (Also called as ‘update’ (U))
  - Update an existing resource
- DELETE (D)
  - Delete an existing resource
- Above actions are also called as CRUD operations

REST Action semantics

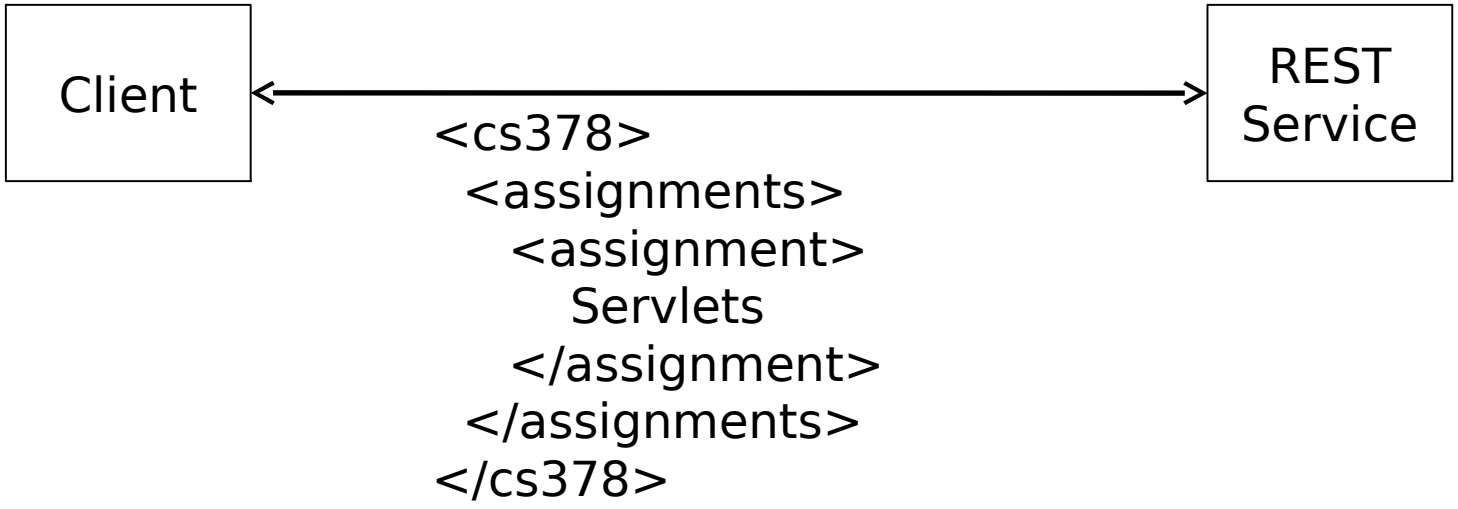
- Idempotent actions
  - Side-effect of N > 0 identical requests on the web application/REST service is the same as for a single request.
  - GET, HEAD, PUT, DELETE are idempotent
    - http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html
- GET – Getting a resource multiple times does not change the state of the web app/REST service
- PUT – Executing the PUT operation with the same data multiple times does not change the state of that resource
- DELETE – Deleting a resource once is same as deleting it multiple times (responses will be different for first and other deletes but the state of web app will be same in both cases)
- What about POST?

REST Action semantics (cont.)

- Idempotent sequences
  - A sequence is idempotent if a single execution of the entire sequence always yields a result that is not changed by a re-execution of all, or part, of that sequence.
    - Idempotent sequence: <GET, GET, GET>, <PUT>, <DELETE>, <DELETE, DELETE>
    - Non-idempotent sequences: <PUT, PUT> (where data is different for each PUT action)
- Are action semantics dependent on returned status?
  - After initial DELETE, subsequent DELETE invocations will result in 404 (Not Found)
  - Does the returned status code determine the semantics of the action?
    - No. Action semantics are only dependent on the state of web application system and not on the returned status

Representation Oriented

- REST service is addressable through a specific URI and representations are exchanged between the client and service



Stateless communication

- No client session data is stored on server
- Server only records and manages the state of the resources it exposes
- If any session data is required, it has to be maintained by the client and sent with each request
  - We already know techniques for this
    - Cookies
    - URLRewriting
    - Request body parameters

HATEOAS

- Hypermedia As The Engine Of Application State (HATEOAS)
  - Discoverability
    - One resource points to other resources

```
<cs378>
  <assignments>
    <assignment>
      <name>Caching Proxy Server</name
      <link>cachingproxy</link>
    </assignment>
  </assignments>
</cs378>
```

How to design REST APIs?

- How to go about designing a REST API?

- A recipe

Recipe for designing REST APIs

- From the problem statement:
  - Step 1: Identify potential *resources* in the system
  - Step 2: Map *actions* to the identified resources
  - Step 3: Identify actions that cannot be mapped to resources
  - Step 4: Refine resource definitions by adding new resources in the resource model if required
  - Step 5: Introduce resource representations
  - Step 6: Re-map actions by establishing resource hierarchies
  - Step 7: Revisit above steps if required

REST API for UT course listings

- Queries
  - List all departments
  - List all courses within a department
  - Find information about a particular course, such as the instructor, pre-requisites, meeting time

Designing REST API

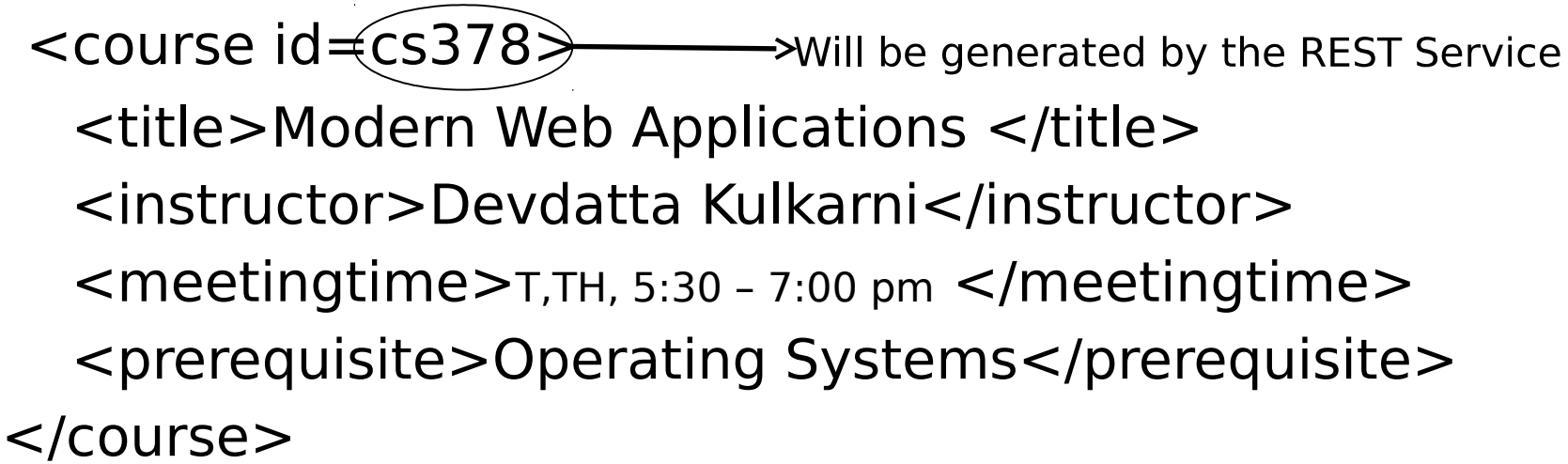
- Step 1: Identify the *resources*
  - Hint: Look for nouns
    - course, department, instructor, prerequisite, meeting time
- Steps 2 and 3: Map actions to resources
  - Action that can be mapped to above resources
    - Find information about a course
      - GET /course/{course_id}
        - {course_id} is a placeholder parameter
  - Action that cannot be mapped
    - List all courses within a department
    - Why?
      - Above identified resources are useful to get information about a specific resource
      - No resource available which would be appropriate to be used in a GET call to obtain listing of *all* courses

Designing REST API

- Step 4: Refine resource model
  - Introduce following resources
    - *courses* (plural form of course), *departments* (plural form of department)
- What about following entities?
  - instructor, pre-requisites, meeting time
  - Should we model these as separate resources or should we model them as *attributes* of the course resource?
  - Depends on how we intend to use them
    - If we are planning to support CRUD operations on them then they should be modeled as a separate resource
    - If not, we can keep them as attributes on the *course* resource

Designing REST API

- Step 5: Introduce resource representations



Designing REST API

- Step 6: Re-map actions to resources
  - List all departments
    - GET /departments
  - List all courses within a department
    - GET /departments/{department_id}/courses
      - E.g.: GET /departments/cs/courses
  - Find information about a particular course
    - GET /departments/{department_id}/courses/{course_id}
      - E.g.: GET /departments/cs/courses/cs378

REST API to REST Service

- What is a REST Service?
  - A service implemented to support REST-based access to resources maintained by a web application

Java REST Specification

- The Java API for RESTful Web Services (JAX-RS)
  - https://jsr311.java.net/nonav/releases/1.1/spec/spec.html
- JAX-RS implementations
  - Jersey (https://jersey.java.net/)
    - Reference implementation by SUN
  - RESTEasy (http://resteasy.jboss.org/)
    - From Redhat Jboss
      - Bill Burke, author of RESTful Java book is the project lead
  - Restlet (http://restlet.com/)
  - Spring's in-built support

JAX-RS Concepts

- Resources
- Resource methods
- Annotations
  - @Path
  - @Consumes
  - @Produces
  - :
  - :
- Application Class

Resources

- Using JAX-RS, a Web resource is implemented as a *resource class* and requests are handled by *resource methods*
- Resource class
  - It is a Java class that uses *JAX-RS annotations* to implement a corresponding Web resource
  - A resource class needs to have at least one method annotated with @Path or a request method designator (@GET, @POST, etc.)

Resource Life-cycle

- By default a new *resource class instance* is created for each request to the resource
  - Per-request model
- Life-cycle
  - First the constructor is called
  - Then, any requested dependencies are injected
  - Then, the appropriate method is invoked
  - Finally, the object is made available for garbage collection

Resource Constructors

- Root resource classes are instantiated by the JAX-RS runtime and MUST have a public constructor for which JAX-RS runtime can provide all parameter values
  - Zero argument constructor is permissible under this rule
- Non-root resource classes are instantiated by application and do not require such a public constructor
- A public constructor MAY include parameters annotated with one of the following:
  - @Context, @Header-Param, @CookieParam, @MatrixParam, @QueryParam, @PathParam
    - However, per-request information may not make sense in a constructor

Resource Methods

- These are methods of a resource class annotated with a *request method designator*
- Examples of request method designators defined by JAX-RS
  - @GET, @POST, @PUT, @DELETE, @HEAD
- Visibility
  - Only *public* methods may be exposed as resource methods

Resource Methods

- Parameters
  - Annotated parameters
    - Method parameters can be annotated to allow injecting of values from the request
    - It is possible to specify a *default value* for a parameter
  - Non-annotated parameters
    - These are called *entity parameters*
    - The value of an entity parameter is mapped from the request entity body
  - Resource methods can have *at most one* non-annotated parameter

Return Type

- Resource methods may return following types:
  - void
    - Results in empty entity body with a 204 status code
  - Response
    - Entity body of the result is mapped from *entity* property of the Response
    - Status code of the result is mapped from *status* property of the Response
  - A Java type
    - Entity body of the result is mapped from the Java class
  - GenericEntity
    - Represents a response entity of a generic type

Declaring Media Capabilities

- Application classes can declare supported request and response media types using the @Consumes and @Produces annotations

- These annotations can be applied to a resource method or a resource class

Application Class

- This class tells our application server which JAX-RS components we want to register with the JAX-RS runtime

- In our code we need to implement a class that extends:

  javax.ws.rs.core.Application class

Application Class

- Has two methods
  - public Set<Class<?>> getClasses()
    - Get a set of root resource *classes*
    - Default life-cycle for resource class instances is ‘per-request’
  - public Set<Object> getSingletons()
    - Get a set of *objects* that are *singletons* within our application
      - These objects are shared across different requests
- Difference between getClasses() and getSingletons()
  - http://stackoverflow.com/questions/18254555/jax-rs-getclasses-vs-getsingletons

JAX-RS Applications

- A JAX-RS application is packaged as a Servlet in a .war file
  - The *Application* subclass and resource classes are packaged in WEB-INF/classes
  - The required libraries are packaged in WEB-INF/lib
  - When using a JAX-RS aware servlet container, the *servlet-class* element of web.xml should name the *application-supplied* subclass of Application class

JAX-RS Applications

- When using a non-JAX-RS aware servlet container, the *servlet-class* element of web.xml should name the *JAX-RS implementation-supplied Servlet class*.

- The application-supplied subclass of Application is identified using an *context-param* with a param name of *javax.ws.rs.Application*

Examples

- RESTful Java with JAX-RS 2.0
  - Example code available at:
    - https://github.com/oreillymedia/restful_java_jax-rs_2_0
    - Once you clone it and unzip the file, the code examples are available in:
      - /examples/oreilly-jaxrs-2.0-workbook/

Running Examples

- Run from command line using Maven
  - Check out chapters 17 and 18 of the “RESTful Java” book
  - Note: Chapter 18 has good discussion about “pom.xml”

Additional Example

- https://github.com/devdattakulkarni/ModernWebApps/tree/master/REST

- Setting up and running the project in Eclipse.

  1) Import project as follows:
  - In "Package Explorer" view: - Import -> Existing Projects into Workspace

  2) Build: - Project -> Build Project

  3) Run on Server

  4) Try out following urls:
  - http://localhost:8080/assignment4/ut/courses
  - http://localhost:8080/assignment4/ut/helloworld

Running RESTEasy examples in Eclipse

- Precautions to take:
  - Check that the "Java Build Path" for your project has Maven Dependencies set in the Deployment Assembly.
  - How to check (or set it)?
  - Right click on project in the Package Explorer view
  - Choose *Build Path*
  - Choose *Configure Build Path*
  - Choose *Deployment Assembly*
  - Choose *Add*
  - Choose *Java Build Path Entries*
  - Choose *Maven Dependencies*

References

- http://www.infoq.com/articles/springmvc_jsx-rs
- http://stackoverflow.com/questions/80799/jax-rs-frameworks
- http://karanbalkar.com/2013/09/tutorial-54-getting-started-with-resteasy-using-eclipse/
- http://docs.jboss.org/resteasy/docs/3.0.1.Final/userguide/html/Maven_and_RESTEasy.html

References

- http://www.javacodegeeks.com/2011/01/restful-web-services-with-resteasy-jax.html
- http://docs.jboss.org/resteasy/docs/3.0.5.Final/userguide/html_single/#javax.ws.rs.core.Application
- http://howtodoinjava.com/2013/07/30/jaxb-exmaple-marshalling-and-unmarshalling-list-or-set-of-objects/
- https://access.redhat.com/solutions/55793

Annotations

Spring vs. JAX-RS

- Spring Annotations
  - @RequestMapping, @Controller, @Component, @PathVariable, @RequestHeader, @RequestBody, @ResponseBody, etc.
- JAX-RS Annotations
  - @Path, @PathParam, @MatrixParam, @HeaderParam, @FormParam, @CookieParam, etc.

Spring vs. JAX-RS

- Why different annotations?
  - Different use cases
    - Spring started with supporting ‘user-in-the-loop’ model-view-controller web applications
    - JAX-RS focuses on REST API based web services
- Spring Annotations are tied to spring framework
  - Lock-in to Spring
- JAX-RS Annotations are part of a specification
  - In theory, application code is portable across different JAX-RS implementations

Parameters

Scope of Path Parameters

- If a named URI path parameter is repeated by different @Path expressions, the @PathParam annotation will always reference the final path parameter

```
@Path(“/customers/{id}”)
    public class CustomerResource {
        @Path(“/address/{id}”)
        @GET
        public String getAddress(@PathParam(“id”) String addressId)
    }
```

If we do *GET /customers/123/address/456*, the addressId is bound to 456

@MatrixParam

```
@Path(“/{make}”)
public class CarResource {
    @GET
    @Path(“/{model}/{year}”)
    @public Jpeg getPicture
        (@PathParam(“make”) String make,
         @PathParam(“model”) String model,
         @MatrixParam(“color”) String color)
}
```

GET /mercedes/e55;color=black/2006

@MatrixParam

- Example
  - Ex05_1 (O'reilly book)
    - Change Junit version in pom.xml from 4.1 to 4.4

@FormParam

```
@POST
public void createCustomer
   (@FormParam(“firstname”) String first,
    @FormParam(“lastname”)
   ) {
}
```

Example: ex05_02

Programmatic URI Information

```
public interface UriInfo {
   public String getPath();
   public List<PathSegment> getPathSegments();
   public MultivaluedMap<String, String> getPathParameters();
   :
}
@Path(“/cars/{make}”)
public class CarResource {
    @GET
    @Path(“/{model}/{year}”)
    public Jpeg getPicture(@Context UriInfo info)
}
```

Spring Beans vs @BeanParam

- Injecting Spring Beans and RESTEasy
  - http://docs.jboss.org/resteasy/docs/1.1.GA/userguide/html/RESTEasy_Spring_Integration.html
  - http://www.mkyong.com/webservices/jax-rs/resteasy-spring-integration-example/

@BeanParam

```
public class CustomerInput {
    @FormParam(“first”)
    String firstName;

    @FormParam(“last”)
    String lastName;

    @HeaderParam(“Content-Type”)
    String contentType;
}

@POST
public void createCustomer(@BeanParam CustomerInput newCust)
```

@BeanParam

- The JAX-RS runtime will introspect the @BeanParam parameter’s type for injection annotations and then set them as appropriate

- Using Beans is a great way to aggregate information instead of having a long list of method parameters

Type Conversion

Type Conversion

- How to convert from String representation within an HTTP request into a specific Java type?

- JAX-RS can convert String data into any Java type, provided that it matches one of the following criteria:
  - It is a primitive type (int, short, float, double, byte, char, and boolean)
  - It is a Java class that has a constructor with a single String parameter
  - It is a Java class that has a static method named *valueOf()* that takes a single String argument and returns an instance of the class
  - It is a java.util.List<T>, java.util.Set<T>, java.util.SortedSet<T>

Custom parameter conversion

- Custom conversion of HTTP parameters to Java objects
  - JAX-RS provides two interfaces
    - *ParamConverter*
    - *ParamConverterProvider*

```
public interface ParamConverter<T> {
    public T fromString(String value)
    public String toString(T value)
}
```

Custom parameter conversion

```
public interface ParamConverterProvider {
    public <T> ParamConverter<T> getConverter(
        Class<T> rawType,
        Type genericType,
        Annotation annotations[]
    )
}
```

The class that implements this interface needs to be registered with your *Application* deployment class

Content Handling

- Unmarshalling
- Marshalling

- Unmarshalling
  - Converting from the over-the-wire representation of data into an in-memory representation
- Marshalling
  - Converting from the in-memory representation to a representation that is suitable for over-the-wire transmission

JAXB

- JAXB is an annotation framework that maps Java classes to XML and XML schema

- Example: ex06_1

Custom Marshalling

- JAX-RS allows writing your own handlers for performing marshalling/unmarshalling actions

```
public interface MessageBodyWriter<T> {
    boolean isWriteable()
    long getSize()
    void writeTo()
}

public interface MessageBodyReader<T> {
    boolean isReadable()
    T readFrom()
}
```

Exception Handling

- Application code is allowed to throw any Exception
- Thrown exceptions are handled by the JAX-RS runtime if you have registered an exception mapper
- An exception mapper can convert an exception to an HTTP response
- If the thrown exception is not handled by a mapper, it is propagated and handled by the container

Exception Handling

- JAX-RS provides the javax.ws.rs.WebApplicationException
- This can be thrown by application code and automatically processed by JAX-RS without having to write an explicit mapper
- When JAX-RS sees that a WebApplicationException has been thrown, it catches the exception and calls its getResponse() method

Exception Handling

- Example: ex07_1
  - Exception Mapper is annotated with @Provider

HATEOAS

- Data format provides extra information on how to change state of your application
- Atom Links:

```
<link rel="next"
      href=
http://example.com/customers?start=2&size=2
      type="application/xml" />
```

- Example: ex10_1

HATEOAS

- Advantages
  - Location transparency
    - Server can change the links without clients having to know about them
  - Logical name to a state transition

    href="http://example.com/customers?start=2&size=2"

Caching

- Conditional GETs
  - Server sends ‘max-age’ and ‘Last-Modified’ headers
  - Client sends ‘If-Modified-Since’ header
  - Server sends either
    - 304 - ‘Not-Modified’
    - 200 – new representation

Caching

- Conditional GETs
  - Server sends ‘Etag’
  - Client sends Etag value in ‘If-None-Match’ header
  - Server sends either
    - 304 - ‘Not-Modified’
    - 200 – new representation

Caching

```
public interface Request {
....
ResponseBuilder
evaluatePreconditions(EntityTag eTag)

}
```

Reading

- Chapters
  - 1, 2, 3, 4, 5, 6, 7, 10, 11
- RESTful Java with JAX-RS 2.0
  - Chapters 1 and 2
- Java for Web Applications
  - Chapter 13

References

- http://www.hascode.com/2011/09/rest-assured-vs-jersey-test-framework-testing-your-restful-web-services/
- http://www.baeldung.com/2011/10/13/integration-testing-a-rest-api/
- http://www.javacodegeeks.com/2011/10/java-restful-api-integration-testing.html
- http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

REST Service Example

- REST API for BitBucket
  - http://restbrowser.bitbucket.org/
- Resources
  - User, Privileges, Follows, Repositories, Issues, Milestones

Field and Bean Properties

- When a resource class is instantiated, the values of field and bean properties annotated with one of the following annotations are set according to the semantics of the annotation:
  - @QueryParam: Extracts the value of a URI query parameter (http://localhost:8080/classes?name=cs378)
  - @PathParam: Extracts the value of a URI template parameter (http://localhost:8080/classes/cs378/)
  - @CookieParam: Extracts the value of a cookie
  - @HeaderParam: Extracts the value of a header
  - @MatrixParam: Extracts the value of a URI matrix parameter (http://localhost/ut/dept;name=cs/classes;name=cs378/)
  - @Context: Injects an instance of a supported resource
    - UriInfo, HttpHeader, Request

Field and Bean injection

- Injection of properties occurs at object creation time
- Therefore, use of annotations on resource class fields and bean properties is only supported for the default per-request resource class lifecycle
- A JAX-RS implementation is required to set these properties only for root resources
  - It is the responsibility of the application to set the properties for sub resources

Environment

- The container-managed resources available to a JAX-RS root resource class depend on the environment in which it is deployed.

- Resources that are available for JAX-RS apps deployed within a Servlet container
  - ServletConfig, ServletContext, HttpServletRequest, HttpServletResponse

Context

- JAX-RS provides facilities for obtaining and processing information about the application deployment context and the context of the individual requests

- Available contexts for resource classes
  - UriInfo, HttpHeaders, Request, SecurityContext

Context Parameter: UriInfo

```
@GET
@Produces(“text/plain”)
public String listQueryParams(@Context UriInfo info)
{
    StringBuilder buf = new StringBuilder();
    for(String param:
info.getQueryParameters().keySet()) {
      buf.append();
   }
   return buf.toString();
}
```

Context Parameter: HttpHeaders

- public String listHeaderNames(@Context HttpHeaders headers)

Context Parameter: Request

```
@PUT
public Response updateAccount(@Context Request request,
Account accnt) {
	EntityTag tag = getCurrentTag();
	 ResponseBuilder responseBuilder =
request.evaluatePreconditions(tag);
	if (responseBuilder != null) {
		return responseBuilder.build();
	}
	else {
		return updateAccount(accnt);
	}
}
```