

COPENHAGEN BUSINESS ACADEMY



RESTful Web Services

Lars Mortensen

RESTful Web services:

https://en.wikipedia.org/wiki/Representational_state_transfer

A **Web Service** is a method of communication between two electronic devices over a network.

It is a software function provided at a network address over the web, with the service always on.

The W3C defines a Web service generally as:

A software system designed to support interoperable machine-to-machine interaction over a network.

RESTful Web Services

- HTTP transport protocol
- No specific Data Protocol



We will focus only on this during the: AP Degree in Computer Science (Datamatiker)

REpresentational State Transfer

Representational

Clients possess the information necessary to identify, modify, and/or delete a web resource.

State

All resource state information is stored on the client.

Transfer

Client state is passed from the client to the service through HTTP.

- Stateless
- Cacheable
- Client-server
- Layered System (Resources are decoupled from their representation)
- Uniform interface
- Code on Demand (optional)

Web service APIs that adhere to these REST Architectural Constraints are called RESTful APIs.

State dependencies limit and restrict scalability, so:

- REST interactions store no client context on the server between requests.
- The client holds session state.
- All information necessary to service the request is contained in the URL, query parameters, body or headers.

Clients can cache the responses.

The responses must define themselves as, cacheable or not, to prevent the client from sending the inappropriate data in response to further requests.

The clients and the server are separated from each other:

- The client is not concerned with the data storage thus the portability of the client code is improved
- The server is not concerned with the client interference, thus the server is simpler and easy to scale.

Layered System -1

Resources decoupled from their representation

- At any time clients cannot tell if they are connected to the end server or to an intermediate.
- Neither can Clients see (and should not consider), the technologies used to implement a REST API



Layered System -2

Resources decoupled from their representation

When resources are decoupled from their representation their content can be accessed in a variety of formats:

What Clients See via the API

- **JSON**
- **XML**
- **HTML**
- **Plain text**
- **PDF**
- **JPEG**



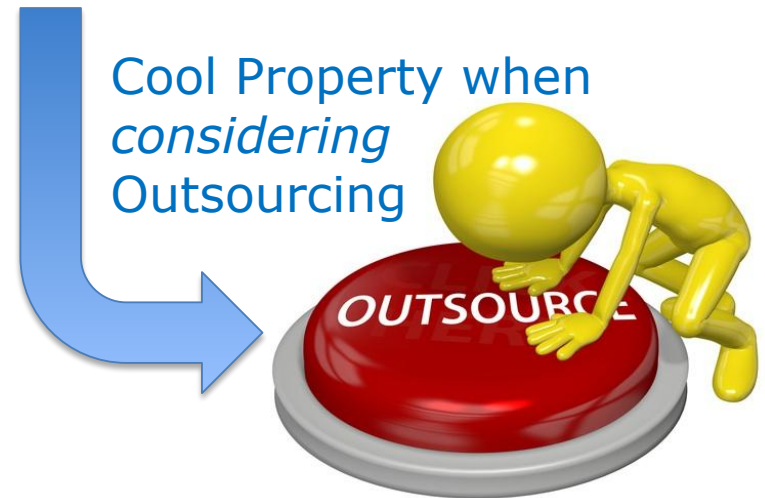
Internal Representation

- **Classes (Java, C# etc.)**
- **Language Data Structures**
- **Business Logic**
- **Database Tables**
- **Exceptions**
- **...**

Uniform interface

Individual resources are identified using URLs.

- The uniform interface constraint is fundamental to the design of any REST service.
- The uniform interface **simplifies** and **decouples** the architecture, which enables each part to evolve independently.



The four constraints for the Uniform Interface

- Identification of resources
- Manipulation of resources through these representations
- Self-descriptive messages
- Hypermedia as the engine of application state ([HATEOAS](#))

Identification of resources

Every interesting resource has its own unique URI. This URI can be used to request an instance of the resource
The HTTP verbs comprise a major portion of the identification part

Use:

- **POST** to create a resource on the server
- **GET** to retrieve a resource
- **PUT** To change the state of a resource or to update it
- **DELETE** to remove or delete a resource

Manipulation of resources through representations

- An identified resource can be returned in various formats such as HTML, XML, **JSON**, PNG, SVG, etc. These formats are *representations* of the identified resource. The list of possible formats ([Media Types](#)) understood by clients and servers is constrained and each format is well-defined.
- REST applications may support more than one representation of the same resource at the same URI. REST applications allow clients to indicate which representation of a resource they wish to receive via the **Accept HTTP Header** that is passed by the client to the server with each request for a resource

Self Descriptive Message -1

REST Web service URIs should be intuitive to the point where they are easy to guess.

Think of a URI as a kind of self-documenting interface that requires little, if any, explanation or reference for a developer to understand, similar to how you can (should) guess what a function does only by its name.

Self Descriptive Message -2

One way to achieve this level of usability is to define directory structure-like URIs. This type of URI is hierarchical, rooted at a single path, and branching from it are sub paths that expose the service's main areas.

`http://www.myservice.org/discussion/topics/{topic}`

With this structure, it's easy to pull up discussion threads just by typing something after /topics/:

- **`http://www.myservice.org/discussion/topics/football`**
- **`http://www.myservice.org/discussion/topics/golf`**
- **`http://www.myservice.org/discussion/topics/biking`**
- **`http://www.myservice.org/discussion/topics/volleyball`**

Resource identification through URIs cphbusiness

Semantic URL (RESTful URL)



Non-semantic URL

`http://example.com/index.php?page=name`

`http://example.com/index.php?page=consulting/marketing`

`http://example.com/products?category=2&pid=25`

`http://example.com/cgi-bin/feed.cgi?feed=news&frm=rss`

`http://example.com/services/index.jsp?category=legal&id=patents`

`http://example.com/kb/index.php?cat=8&id=41`



Semantic URL

`http://example.com/name`

`http://example.com/consulting/marketing`

`http://example.com/products/2/25`

`http://example.com/news.rss`

`http://example.com/services/legal/patents`

`http://example.com/kb/8/41`

http://en.wikipedia.org/wiki/Semantic_URL

Developing RESTful Web Services with Java

RESTful APIs will be the fundamental Server Building Blocks for all the designs we will build for the rest of the semester.

Since REST relies on HTTP, everything that can handle HTTP request can be used to provide a REST service.

- Servlets + GSON has allowed us to provide REST API's
- Java however, provides a specification **JAX-RS** that provides support in creating Web Services

Developing RESTful Web Services with JAX-RS cphbusiness

- JAX-RS: Java API for RESTful Web Services (JAX-RS) is a Java programming language API spec that provides support in creating web services according to the Representational State Transfer (REST) architectural pattern
- The JAX-RS API uses Java programming language **annotations** to simplify the development of RESTful web services.
- JAX-RS annotations are runtime annotations; therefore, runtime reflection will generate the helper classes and artifacts for the resource.

Jersey, the reference implementation from Oracle

There are several implementation of the JAX-RS specification. We will use the reference implementation Jersey .

User [Guide](#)

API [Documentation](#)

JAX-RS with Tomcat and Maven

```
<dependency>
  <groupId>org.glassfish.jersey.bundles</groupId>
  <artifactId>jaxrs-ri</artifactId>
  <version>2.23.2</version>
</dependency>
```

REST with JAX-RS

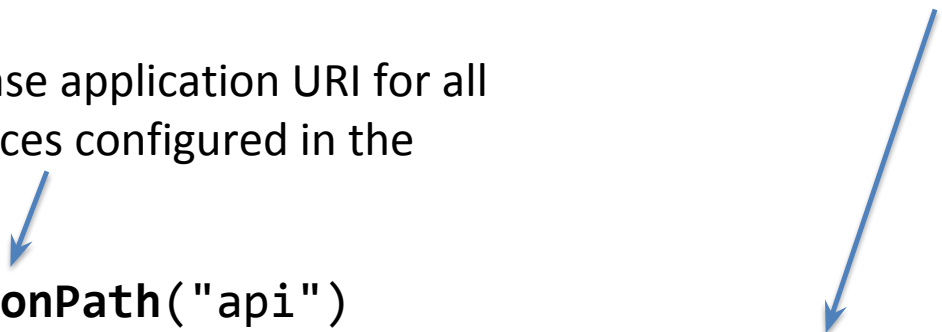


A quick demo before we start

JAX-RS Application Model

JAX-RS provides a deployment agnostic abstract class [Application](#) for declaring **root resource** and **provider** classes etc. A Web service (as is done via the NetBeans Wizard) may extend this class to declare root resource and provider classes

Define the base application URI for all JAX-RS resources configured in the application



```
@ApplicationPath("api")
public class MyApplication extends Application {
    @Override
    public Set<Class<?>> getClasses() {
        Set<Class<?>> s = new HashSet<Class<?>>();
        s.add(HelloWorldResource.class);
        return s;
    }
}
```

JAX-RS annotations

```
@Path("/helloworld")
public class HelloWorldResource {
    @GET
    @Produces("text/plain")
    public String getMessage() {
        return "Hello World";
    }

    @POST
    @Consumes("text/plain")
    public void postMessage(String message) {
        // Store the message
    }
}
```

The Java class will be hosted at the URI path
"/helloworld"

The Java method will process HTTP GET requests

The method will produce content
identified by the MIME Media type
"text/plain"

The Java method will process HTTP POST requests

The method can consume content
identified by the MIME Media type
"text/plain"

The @Path Annotation

The @Path annotation identifies the URI path template to which the resource responds and is specified at the class or method level of a resource.

The @Path annotation's value is a partial URI path template relative to the base URI of the server on which the resource is deployed, the context root of the application, and the URL pattern to which the JAX-RS runtime responds.

@Path("/helloworld")

```
public class HelloWorldResource {  
    @GET  
    @Produces("text/plain")
```



Parameters of a resource method may be annotated with parameter-based annotations to extract information from a request

We can extract the following types of parameters for use in a resource class:

- **Query**
- **URI path**
- **Form**
- Cookie
- Header
- Matrix

Query parameters For Non-Semantic URLs



```
@GET
@Produces("text/plain")
@Path("/QueryDemo")
public String getText(    @QueryParam("id") int id,
                        @QueryParam("name") String name,
@DefaultValue("Teacher") @QueryParam("position") String position)
{
    return id+ " : "+ name + " : " + position;
}
```

URI Path Templates For Semantic URLs



URI path templates are URIs with variables embedded within the URI syntax. These variables are substituted at runtime in order for a resource to respond to a request based on the substituted URI. Variables are denoted by braces ({ and }).

```
@Path("/users/{username}")
```

To obtain the value of the user name, the `@PathParam` annotation may be used on the method parameter of a request method

```
@Path("/users/{username}")
public class UserResource {
    @GET
    @Produces("text/xml")
    public String getUser(@PathParam("username") String userName){
        ...
    }
}
```



All the Details 😊

<http://docs.oracle.com/javaee/7/tutorial/doc/jaxrs002.htm>