This page last changed on Mar 13, 2011 by patrik_nordwall.

Sculptor REST Tutorial

This hands-on tutorial will walk you through the steps of how to use the REST support in Sculptor. Sculptor makes it very easy to expose restful services by using conventions and easy delegation to Services to reduce boilerplate coding. Spring MVC is the underlaying REST framework. Spring configuration for html, json and xml representations are generated.

Table of Contents:

Setup Project

Run Part1 of 2. Hello World Tutorial (CSC) to create maven and Eclipse projects.

Modify pom.xml to make it a web application

```
<packaging>war</packaging>
```

Add some dependencies to pom.xml

```
<dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
      <version>${spring.version}</version>
</dependency>
<dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>2.5</version>
</dependency>
<dependency>
   <groupId>org.codehaus.jackson</groupId>
   <artifactId>jackson-core-lgpl</artifactId>
   <version>1.4.3</version>
</dependency>
<dependency>
   <groupId>org.codehaus.jackson</groupId>
   <artifactId>jackson-mapper-lgpl</artifactId>
   <version>1.4.3</version>
```

```
        </dependency>
        <dependency>
          <groupId>org.springframework</groupId>
          <artifactId>org.springframework.oxm</artifactId>
          <version>${spring.version}</version>
        </dependency>
        <dependency>
           <groupId>com.thoughtworks.xstream</groupId>
           <artifactId>xstream</artifactId>
           <version>1.3.1</version>
        </dependency>
```

Add jetty plugin in pom.xml

```
            <plugin>
                  <groupId>org.mortbay.jetty</groupId>
                  <artifactId>maven-jetty-plugin</artifactId>
                  <version>6.1.11</version>
                  <configuration>
                        <webAppConfig>
                              <contextPath>/</contextPath>
                        </webAppConfig>
                        <port>8888</port>
                        <stopKey>STOP</stopKey>
                        <stopPort>8889</stopPort>
                        <systemProperties>
                              <!-- enable easy JMX connection to JConsole -->
                              <systemProperty>
                                    <name>com.sun.management.jmxremote</name>
                                    <value />
                              </systemProperty>
                              <systemProperty>
                                    <name>jetty.port</name>
                                    <value>8888</value>
                              </systemProperty>
                        </systemProperties>
                        <scanIntervalSeconds>30</scanIntervalSeconds>
                  </configuration>
            </plugin>
```

Initial Model

Add the following to model.btdesign:

```
Resource HelloWorldResource {
      String hello(String msg, ModelMap modelMap) return="helloworld/hello";
}
```

Generate code with `mvn generate-sources`. Look at the generated HelloWorldResource.java. The @RequestMapping path and HTTP method are defined by convention, i.e. the path is the name of the resource and the HTTP method is by default GET. You can also explicitly define these in the model:

```
String hello(String msg, ModelMap modelMap) GET path="/world";
```

Complete the code in HelloWorldResource with something like this (most of it should already be there):

```
@Override
@RequestMapping(value = "/helloWorld", method = RequestMethod.GET)
public String hello(@RequestParam("msg") String msg, ModelMap modelMap) {
    modelMap.addAttribute("result", msg);
    return "helloworld/hello";
}
```

The `return` attribute in the model indicates that this operation will show a view with specified name. This also result in a generated JSP file in `src/main/webapp/WEB-INF/jsp/helloworld/hello.jsp`.
The JSP is only generated once and you need to complete it. Maybe showing the msg.

```
<p>
${result}
</p>
```

Try it by starting jetty with `mvn jetty:run` and browse to http://localhost:8888/rest/helloWorld?msg=Hi

Servlet and Spring configuration is generated in web.xml and rest-servlet.xml. By default, 3 content type representations are supported; html, xml and json. When testing xml and json it is convenient to use cURL. Try this from console:

```
curl http://localhost:8888/rest/helloWorld.json?msg=Hi

curl http://localhost:8888/rest/helloWorld.xml?msg=Hi
```

CRUD Resource

Let's move on to a simple CRUD service. Add the following to model.btdesign:

```
Resource PlanetResource {
        show => PlanetService.findById;
        String createForm;
        create => PlanetService.save;
        delete => PlanetService.delete;
        showAll => PlanetService.findAll;
}

Service PlanetService {
        findById => PlanetRepository.findById;
        findAll => PlanetRepository.findAll;
        save => PlanetRepository.save;
        delete => PlanetRepository.delete;
}

Entity Planet {
        String name
        int diameter
        Repository PlanetRepository {
                findById;
                save;
                delete;
                findAll;
        }
```

```
}
```

Generate code. No hand written code is needed.

Start jetty or wait for jetty reload. Open browser at http://localhost:8888/rest/planet and try the simple CRUD functionality. Note that the generated JSPs are working for this simple example, but they are only intended to be a starting point for writing your own user interface.

You can try the xml and json representations also. For example, to POST data for a new Planet

```
curl -i -H "Content-Type: application/json" -X POST -d '{"name":"Earth4","diameter":1275632000}' http://
localhost:8888/rest/planet
```

Note that there are two ways to POST to the resource, either with the data in the request body as above, or with form data, as in the html form. Form data can be produced with cURL also:

```
curl -i --data 'name=Earth&diameter=1275632000' http://localhost:8888/rest/planet
```

Retrieve the new Planet as xml:

```
curl http://localhost:8888/rest/planet/1.xml
```

Note that fully qualified class names are used in the xml. Probably not what you want. Easily fixed, by adding the following to your sculptor-generator.properties @XStreamAlias will be generated.

```
generate.xstream.annotation=true
```

There is a shortcut for generating CRUD operations. The above example can be made shorter by using the the `scaffold` keyword.

```
Resource PlanetResource {
	scaffold
}

Entity Planet {
	scaffold
	String name
	int diameter
}
```

A note regarding update - PUT

It is easy to define operation for PUT:

```
Resource PlanetResource {
	show => PlanetService.findById;
	String udpateForm;
```

```
        update => PlanetService.save;
    }
```

However, this will not work if the underlaying domain object (Planet) is an Entity or ValueObject, because those have some attributes that are not possible to modify (no setter methods). id, uuid, key and other not changeable attributes. This means that the data that is PUT can't be automatically mapped to the domain object instance. Therefore, for update you must use DataTransferObject with all attributes defined and do the mapping manually.

```
    Resource PlanetResource {
        show => PlanetService.findById;
        String updateForm throws PlanetNotFoundException;
        update(@PlanetForm planetForm) throws PlanetNotFoundException;
    }

    DataTransferObject PlanetForm {
        String name
        int diameter
        IDTYPE id
        Long version
    }
```

In the Resource gap code you need to write the mapping between Entity and DataTransferObject.

```
  @Override
  @RequestMapping(value = "/planet/{id}/form", method = RequestMethod.GET)
  public String updateForm(@PathVariable("id") Long id, ModelMap modelMap) throws
PlanetNotFoundException {
      PlanetForm planetForm = new PlanetForm();
      Planet planet = getPlanetService().findById(serviceContext(), id);
      planetForm.setDiameter(planet.getDiameter());
      planetForm.setId(planet.getId());
      planetForm.setName(planet.getName());
      planetForm.setVersion(planet.getVersion());
      modelMap.addAttribute("planetForm", planetForm);
      return "planet/update";
  }

  @Override
  @RequestMapping(value = "/planet", method = RequestMethod.PUT)
  public String update(@RequestBody PlanetForm planetForm) throws PlanetNotFoundException {
      Planet planet = getPlanetService().findById(serviceContext(), planetForm.getId());
      planet.setVersion(planetForm.getVersion());
      planet.setName(planetForm.getName());
      planet.setDiameter(planetForm.getDiameter());
      getPlanetService().save(serviceContext(), planet);
      return "redirect:/rest/planet/" + planetForm.getId();
  }
```

Data Transfer Object

When you provide a service to external parties you probably want to expose DataTransferObjects (DTO) instead of domain objects, to have the freedom of changing the internal implementation without having to change the published API. You should also use DTOs when the external representation need to be different than the internal domain, which is often the case. The drawback is of course the additional mapping layer between DTOs and domain objects.

You have the choice of implementing the mapping in the Resource, as described in the section about Update/PUT, or in a separate Service. Using a separate Service has the benefit that it can be used for other communication protocols also, such as Spring remoting, which is supported by Sculptor.

It can be defined like this in the model:

```
Resource PlanetResource {
	show => PlanetService.findById;
	String createForm;
	create => PlanetService.save;
	updateForm;
	update => PlanetService.save;
	delete => PlanetService.delete;
	showAll => PlanetService.findAll;
}

Service PlanetService {
	inject @InternalPlanetService

	@PlanetDTO findById(IDTYPE id) throws PlanetNotFoundException;
	List<@PlanetDTO> findAll;
	@PlanetDTO save(@PlanetDTO planet);
	delete(@PlanetDTO planet);
}

DataTransferObject PlanetDTO {
	String name
	int diameter
	IDTYPE id
	Long version
}

Service InternalPlanetService {
	findById => PlanetRepository.findById;
	findAll => PlanetRepository.findAll;
	save => PlanetRepository.save;
	delete => PlanetRepository.delete;
}

Entity Planet {
	String name
	int diameter
	Repository PlanetRepository {
		findById;
		save;
		delete;
		findAll;
	}
}
```

This means that you need to write minimal hand written code in the Resource class. The hand written mapping between DTOs and domain objects is implemented in the service (PlanetService). Dozer is a tool that might be useful for doing the mapping.

Additional Spring MVC Features

Spring MVC has powerful possibilities to declare the methods in the controllers (resources). Some of these are supported by Sculptor, others require that you write the method yourself in the gap class of the resource.

To define matching headers you can use the hint `headers`.

```
Resource PlanetResource {
    asText(Long id) path="/planet/{id}" hint="headers=content-type=text/*";
    asPng(Long id) path="/planet/{id}" hint="headers=content-type=image/png";
}
```

You can narrow path mappings through parameter conditions and that is supported with hint `params`. The sample from Spring MVC reference would look like this:

```
findPet(String ownerId, String petId, Model model) path="/owners/{ownerId}/pets/{petId}"
 hint="params=myParam=myValue";
```

There are some types in the method signature that are special, and useful when you need more fine grained control of the request/response. All these can be defined as parameters of the Resource operations in the model.

- Model, ModelMap, Map
- BindingResult
- HttpServletRequest, HttpServletResponse, HttpSession, WebRequest
- Locale
- java.io.InputStream, java.io.Reader, java.io.OutputStream, java.io.Writer
- java.security.Principal
- org.springframework.validation.Errors

Example:

```
Resource OrderResource {
    show(String orderId, ModelMap modelMap, HttpServletRequest req);
}
```

Hypermedia

So far we have not talked about hypermedia, i.e. navigation by following links. Hypermedia as the engine of application state (HATEOAS) was coined to describe a core tenet of the REST architectural style. One can claim that without using links it is not restful. You have to decide if links are important for your application. Sculptor and Spring MVC doesn't have any special support for links, since they are very application specific and should be defined in runtime depending on things like user's roles and current state. It would be wrong to try to define the rules for the links in the static `model.btdesign`.

A quick example, inspired by the Restbucks sample in the excellent book REST in Practice, of how to define a Resource with links. The links are simply part of the resource representations.

```
Resource OrderResource {
    create(@Order order);
    show(String orderId, ModelMap modelMap, HttpServletRequest req);
}

DataTransferObject Order {
    - List<Item> items
    String location
    Double cost
    String status
    - List<Link> links
}
```

```
DataTransferObject Item {
        String milk
        String cupSize
        String drink
}

DataTransferObject Link {
        String rel required
        String uri required
}
```

Customization

Conventions

The annotations that are generated for the controller methods are by default based on conventions. These conventions are not hard coded. They are defined in sculptor-default-generator.properties, which makes it easy for you to redefine and add your own conventions.

The complete set of properties for this:

```
# In the 'path' and 'return' you can use the following placeholders, which are replaced in generation time
# ${resourceName}
# ${operationName}
# ${p0}, ${p1}, ${p2} ... - operation parameter names
# In the 'return' you can use the following placeholders, which are replaced in runtime
# {id}
rest.default.path=/${resourceName}
rest.default.httpMethod=GET
#rest.default.return=redirect:/

rest.save.path=/${resourceName}
rest.save.httpMethod=POST
rest.save.return=redirect:/rest/${resourceName}/{id}
rest.create.path=/${resourceName}
rest.create.httpMethod=POST
rest.create.return=redirect:/rest/${resourceName}/{id}
rest.update.path=/${resourceName}
rest.update.httpMethod=PUT
rest.update.return=redirect:/rest/${resourceName}/{id}

rest.createForm.path=/${resourceName}/form
rest.createForm.httpMethod=GET
rest.createForm.return=${resourceName}/create

rest.updateForm.path=/${resourceName}/{id}/form
rest.updateForm.httpMethod=GET
rest.updateForm.return=${resourceName}/update

rest.delete.path=/${resourceName}/{id}
rest.delete.httpMethod=DELETE
rest.delete.return=redirect:/rest/${resourceName}

rest.findById.path=/${resourceName}/{${p0}}
rest.findById.httpMethod=GET
rest.findById.return=${resourceName}/show
rest.findByKey.path=/${resourceName}/{${p0}}
rest.findByKey.httpMethod=GET
rest.findByKey.return=${resourceName}/show
```

```
rest.show.path=/${resourceName}/{${p0}}
rest.show.httpMethod=GET
rest.show.return=${resourceName}/show

rest.findAll.path=/${resourceName}
rest.findAll.httpMethod=GET
rest.findAll.return=${resourceName}/showlist
rest.showAll.path=/${resourceName}
rest.showAll.httpMethod=GET
rest.showAll.return=${resourceName}/showlist
```

Note that you can redefine individual properties in your sculptor-generator.properties to adjust the conventions.

You can define your own conventions. For example, in case you would like to define a generic filter operation. Add the following to your sculptor-generator.properties:

```
rest.filter.path=/${resourceName}/filter/{value}
rest.filter.httpMethod=GET
rest.filter.return=${resourceName}/showlist
```

In the model you can then simply use:

```
Resource PlanetResource {
	filter(String value) => PlanetService.findAllMatching;
}

Service PlanetService {
	List<@PlanetDTO> findAllMatching(String value);
}
```

You need to implement the findAllMatching method yourself in the Service, but the delegation mechanism in the resource is completely generated.

XStreamAlias

You should add the following property to your sculptor-generator.properties to avoid fully qualified class names the xml representations. @XStreamAlias will be generated.

```
generate.xstream.annotation=true
```

JAXB

By default, XStream is used for the XML representation. You can use JAXB instead. In `rest-servlet.xml` you can remove the comment for jaxb2Marshaller and replace references to xstreamMarshaller with jaxb2Marshaller. Note that you must define all classes that are to be supported in the classesToBeBound definition of the Jaxb2Marshaller.

JAXB annotations are generated in DataTransferObjects. If you expose other types of domain objects and use JAXB you can turn on generation of JAXB annotations with these properties in sculptor-generator.properties.

```
generate.xml.bind.annotation.valueObject=true
```

```
generate.xml.bind.annotation.entity=true
generate.xml.bind.annotation.basicType=true
generate.xml.bind.annotation.domainEvent=true
generate.xml.bind.annotation.commandEvent=true
```

Separate module

This tutorial explained how to adjust the business tier project to include the Resources and web app configuration. If you like you can define this in a separate project and reference the services in the business tier services with the ordinary import mechanism in model.btdesign.

It is also possible to combine JSF web presentation tier project with REST. In that case you need to define the following properties in sculptor-gui-generator.properties.

```
generate.resource=true
generate.restWeb=true
```

Location of webapp directory

By default the web configuration and jsp pages are generated in src/main/webapp. You can change that by defining `outlet.webroot.dir` in src/main/resources/generator/Workflow.mwe2. For Google Appengine projects it should be located in the `war` directory:

```
Workflow {
    component = @Sculptorworkflow {
        outlet.webroot.dir = 'war'

        modelFile = "classpath:/model.btdesign"
```

Skip generation of JSP

If you don't need html representation or use another view technology than JSP (e.g. Facelet xhtml) you can omit generation of JSPs by defining the following property in sculptor-generator.properties.

```
generate.restWeb.jsp=false
```


 2 3 4 5 6 7 8 9