Creating a MOCA Web Service

Contents

Introduction

MOCA provides support for hosting web services using Spring MVC. For more information, see the Spring Web MVC framework documentation website. This section provides information on creating web services using the functionality that MOCA provides through Spring MVC and Jetty. By using Jetty, an external web hosting solution (such as Apache Tomcat) is not needed to host the web services. Instead, web archives (WARs) can be hot swapped and/or deployed with Jetty. For more information, see the Eclipse Jetty website. Please note that this functionality only work for the 2012 versions and higher.

Development Setup

> An assumption will be made that you have downloaded all the pre-requisite software called out on the Development Setup Guide

For the purposes of this guide we will be setting up a development environment for a WMS instance that will utilize a web service. You will need the following MOCA Levels:

- MOCA
- MCS
- SAL
- SEAMLES
- MTF
- WMD
- LES

This guide will also make the assumption that you do not have Eclipse set up.

Checkout

To expedite this process a `checkout.bat` file has been attached. Download this file to the directory you want to check the source out to, edit the `version` variable in the file to check out the correct tag of the product, and then run the batch file. This will automatically check out all the source code that is noted above.

Build

Registry

Now we need to build the source code that we just checked out. In the les directory that was checked out create a data folder, and download the `registry` file attached. Open the `registry` and set the <CHECKOUT DIRECTORY> for each environment variable. Update your `rptab` file (`C:\ProgramData\RedPrairie\Server\rptab`) to point to the new environment that you created.

Example

```
# INSTANCE_NAME (required);LES_DIRECTORY (required);REGISTRY_FILE
(optional)
dev2013_2_0;c:\dev\2013\2\0\les
```

Change the SERVER `URL`, `PORT`, `CLASSIC-PORT`, and `RMI-PORT` to be ports that are not currently in use. Note that the `URL` and `PORT` share the same value, but otherwise the ports must be different. Commonly, sequential ports are used (e.g. `URL` and `PORT` are 4400, `CLASSIC-PORT` is 4401, and `RMI-PORT` is 4402).

In the DATABASE section set the `username`, `password`, and `databaseName`.

Compile the source code

Open the command prompt that you will be using (tccle, 4NT, cmd.exe, whatever) and perform an `rpset` on your new environment.

Download the `build.bat` attached to this page to the same directory you checked you code out to. Run this batch file to iterate through the various levels and build the source code.

Build the database

Follow the instructions to create and build a database based on the DATABASE information from the registry using the steps listed on the Development Setup Guide.

Developing Web Services

Introduction

When developing web services there are two types that you will need to take into consideration. While development of both are developed the same, the way they are built differs.

Development setup

No matter which type of Web Service you perform the following setup must occur:

1. Create a new directory in `$LESDIR` called `ws`
2. Create a new directory in `$LESDIR/ws` called `webapp`
3. Create a new directory in `$LESDIR/ws/webapp` called `spring`
4. In your running instance run the following:

```
cd %LESDIR%/ws
cp %MOCADIR%/ws/webapp/web.xml web.xml
cp %MOCADIR%/ws/webapp/jetty-web.xml jetty-web.xml
cd spring
cp %MOCADIR%/ws/webapp/spring/webservices-config.xml
webservices-config.xml
cp %MOCADIR%/ws/webapp/spring/webservices.xml webservices.xml
```

This has now setup the base structure and standard authentication for web services. It should be noted that standard authentication performs the following for every web service call:

1. Check if the `MOCA-WS-SESSIONKEY` is set and the session it is linked to is still alive
    a. If the `MOCA-WS-SESSIONKEY` is set and the session is still alive, allow the web service to run
    b. If the `MOCA-WS-SESSIONKEY` is not set or the session is no longer alive check if the web service request is for login
        i. If the `MOCA-WS-SESSIONKEY` is not set or the session is no longer alive, and the web service request is not for a login return a 401 error (Authentication Required)
        ii. If the `MOCA-WS-SESSIONKEY` is not set or the session is no longer alive, and the web service request is for a login, attempt to login
            1. Login failure results in a 401 error (Authentication Required)
            2. Login success returns a new `MOCA-WS-SESSIONKEY` which links to a running session

Pre-existing Standard Web Services

There are Web Services that already exist that you can build on or override. At the time of writing this, only 3 web services exist at the MOCA level:

1. LoginService
2. JobService
3. TaskService

Your First Web Service

Introduction

When developing web services, you have the ability to extend an existing web service, or create your own. For the purposes of this tutorial we are going to create a new web service and leave the existing ones alone.

Simple Test Web Service

Setup

To create our test web service we first need to decide how we want the URL to look as this will drive our directory structure and our request mappings. When someone wishes to access your test web service the URL that the user will access follows this format: `http://host:port/ws/war_file_name/mapping_value`. For our structure our `war_file_name` will be `test` and our `mapping_value` will be `test_web_service`, so our ending URL will look like the following: `http://host:port/ws/test/test_web_service`.

Directory Structure

Now that we know our URL structure we can build our directory structure. In `$LESDIR/ws` create a new directory called `test`, this will contain all the code related to our web service. Inside of `$LESDIR/ws/test` create two more directories, `spring` and `src`. The `spring` directory will contain a single XML file that will tell MOCA which class controls the web service. This is done by utilizing standard spring beans. Copy the following into a file named `webservices.xml`:

webservices.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">
    <bean class="com.redprairie.les.test.VarTestService"/>
</beans>
```

Finally In the `src` folder create the following directory structure `java/com/redprairie/les/test`. This is where the JAVA files will be located to control the web service.

At this point you will now have the following directory structures:

```
$LESDIR/ws/test
$LESDIR/ws/test/spring
$LESDIR/ws/test/spring/webservice.xml
$LESDIR/ws/test/src
$LESDIR/ws/test/src/java
$LESDIR/ws/test/src/java/com
$LESDIR/ws/test/src/java/com/redprairie
$LESDIR/ws/test/src/java/com/redprairie/les
$LESDIR/ws/test/src/java/com/redprairie/les/test
```

Web Service Controller

Web services tend to follow the MVC methodology (Model View Controller). The `Controller` tells the system what `Model` to create and then returns the `View` to the end user. So now we are now going to create the `Controller` for our web service. In `$LESDIR/ws/test/src/java/com/redprairie/les/test` create a new file called `VarTestWebService.java`.

Start editing the file and create the class structure:

Example

```
package com.redprairie.test.test;

public class VarTestWebService
{

}
```

To define this class as the `Controller` we need to add the `org.springframework.stereotype.Controller` tag to the class.

Example

```
package com.redprairie.test.test;

import org.springframework.stereotype.Controller;

@Controller
public class VarTestWebService
{

}
```

We now have a web service `Controller`, however it is not controlling anything since there are no request mappings defined. To create a request mapping first we need a method in our class, and then add the `org.springframework.web.bind.annotation.RequestMapping` tag. This tag requires two additional pieces of information, the `value` and the `method`. The value is the `request_mapping` that when specified in the URL will perform this action, and the `method` is an HTTP request method (`GET, PUT, POST, or DELETE`). Note that multiple methods can utilize the same `RequestMapping value` as long as the `RequestMapping method` is different. While each HTTP request method has a request and response, each one tends to refer to a specific type of process:

- GET - Retrieve Information
- PUT - Update if exists, error if does not exist
- POST - Create if does not exist, error if exists
- DELETE - Delete Information

For our example we are going to perform both a `POST` and a `GET`. `GET` methods can pull arguments from the URL and process them, `POST` methods process information in the body of the request. The `POST` information can be in whatever format that the coder wishes, for this example we are going to be processing a `JSON` message.

So first we need to add two new methods to our class, both pointing to the same value:

Example

```
package com.redprairie.test.test;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
public class VarTestWebService
{
    @RequestMapping(value="test_web_service", method=RequestMethod.GET)
    public void getTest(HttpServletRequest request,
                        HttpServletResponse response)
    {
    }

    @RequestMapping(value="test_web_service", method=RequestMethod.POST)
    public void postTest(HttpServletRequest request,
                         HttpServletResponse response)
    {
    }
}
```

Second we need to define what parameters (if any) we are processing. As noted above `GET` methods pull from the URL arguments. These are noted in the method declaration with a different `org.springframework.web.bind.annotation.RequestParam` for each parameter required. Please note that if a parameter is listed in the method declaration the web service call will error if it is not there. Each `RequestParam` n eeds a `value` defined for it. This links it to the URL argument that it needs to process. This is followed by a variable type (String, int, etc.) and the variable name. So we will add a single simple `RequestParam` to the `GET` method:

Example

```
package com.redprairie.test.test;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class VarTestWebService
{
    @RequestMapping(value="test_web_service", method=RequestMethod.GET)
    public void getTest(HttpServletRequest request,
                        HttpServletResponse response,
                        @RequestParam(value="param1")String param1)
    {
    }

    @RequestMapping(value="test_web_service", method=RequestMethod.POST)
    public void postTest(HttpServletRequest request,
                         HttpServletResponse response)
    {
    }
}
```

For the `POST` we need to get it access to the request body by utilizing a `org.springframework.web.bind.annotation.RequestBody` variable to the method declaration. Unlike `GET`, the `POST` does not require a `value` to be defined as it is processing the entire `body` of the request. Like the `GET`, after the `RequestBody` declaration it is followed by a variable type (String, int, etc.) and the variable name:

Example

```
package com.redprairie.test.test;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class VarTestWebService
{
    @RequestMapping(value="test_web_service", method=RequestMethod.GET)
    public void getTest(HttpServletRequest request,
                        HttpServletResponse response,
                        @RequestParam(value="param1")String param1)
    {
    }

    @RequestMapping(value="test_web_service", method=RequestMethod.POST)
    public void postTest(HttpServletRequest request,
                         HttpServletResponse response,
                         @RequestBody String argMap)
    {
    }
}
```

Web Service Model

We now have the shell of our class created so that we can accept and process `GET` and `POST` requests. We can perform processing directly inside these methods, or we can send the processing off to the `Model` which will create our `View` to return to the user. The view that we return to the user can be whatever we choose, for our example we are going to return a `JSON` message. For simplicity our `Model` is going to be a private Inner Class of our `VarTestWebService` called `ProcessTest`, and it will have methods for `get` and `post`. For now we will implement them as void, and change them to the correct type when we create the `View`.

Example

```
package com.redprairie.test.test;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class VarTestWebService
{
    @RequestMapping(value="test_web_service", method=RequestMethod.GET)
    public void getTest(HttpServletRequest request,
                        HttpServletResponse response,
                        @RequestParam(value="param1")String param1)
    {
    }

    @RequestMapping(value="test_web_service", method=RequestMethod.POST)
    public void postTest(HttpServletRequest request,
                         HttpServletResponse response,
                         @RequestBody String argMap)
    {
    }

    private class ProcessTest
    {
        public void get()
        {
        }

        public void post()
        {
        }
    }
}
```

Web Service View

Now that our `Controller` and `Model` have been created, we need to construct the `View` that we are going to return to the user. The `View` that we return to the user can be anything that we want it to be. Examples include `XML, HTML, and JSON`. For our examples we are going to return `JSON` messages from both our `get` and `post` methods. `MOCA` comes packaged with Jackson JSON, so we need simply import `org.json.JSONObject` to be able to create `JSON` messages. We should also import `org.json.JSONException` since we need to be able to catch JSON errors.

Example

```
package com.redprairie.test.test;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.json.JSONException;
import org.json.JSONObject;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class VarTestWebService
{
    @RequestMapping(value="test_web_service", method=RequestMethod.GET)
    public void getTest(HttpServletRequest request,
                        HttpServletResponse response,
                        @RequestParam(value="param1")String param1)
    {
    }

    @RequestMapping(value="test_web_service", method=RequestMethod.POST)
    public void postTest(HttpServletRequest request,
                         HttpServletResponse response,
                         @RequestBody String argMap)
    {
    }

    private class ProcessTest
    {
        public JSONObject get()
        {
            return new JSONObject();
        }

        public JSONObject post()
        {
            return new JSONObject();
        }
    }
}
```

GET

Now it is time to define what our `GET` method does. We already know that our `Controller` is expecting `param1` from the request URL, so we need to define `param1` as some value in the system. Since we are already connected to the WMS, we will utilize `param1` as a Warehouse ID and

return the results of **list warehouses** as our JSON message. So now we need to pass parameters into our `get` method so that we can use it to find our warehouse. We will also have our `get` method throw both a `MocaException` and a `JSONException` that we will handle later.

Example

```
...
    private class ProcessTest
    {
        public JSONObject get(String warehouseID) throws MocaException,
JSONException
        {
            // Create the return object
            JSONObject returnObject = new JSONObject();

            // Get the warehouse
            MocaResults warehouse =
MocaUtils.currentContext().executeCommand("list warehouses " +

" where wh_id = '" + warehouseID + "' ");

            // We have the warehouse, so move to the result row
            warehouse.next();

            // For each non-null column in the warehouse result, add it to
the returnObject
            for(int columnIndex = 0; columnIndex <
warehouse.getColumnCount(); columnIndex++)
            {
                if(!warehouse.isNull(columnIndex)
                {
                    returnObject.put(warehouse.getColumnName(columnIndex),
warehouse.getValue(columnIndex));
                }
            }

            return returnObject;
        }
...
```

Now we want to send our response back to the user. To do this we add logic to our `getTest` method to write the response. Since our `get` metho d can throw both a `MocaException` and a `JSONException`, we need to handle those scenarios. Please note that when creating the response you need to specify the response status. The response status needs to fall within the standard HTTP status codes.

Example

```
...
    @RequestMapping(value="test_web_service", method=RequestMethod.GET)
    public void getTest(HttpServletRequest request,
                        HttpServletResponse response,
                        @RequestParam(value="param1")String param1)
    {
```

```
        try
        {
            // Write the JSONObject from ProcessTest.get into the response.
            response.getWriter().write(ProcessTest.get(param1).toString());
            response.getWriter().flush();
            response.getWriter().close();

            // Set the content type to JSON so the receiver of the response
            // knows how to process it
            response.setContentType("application/json");

            // Set the response status to a valid response value
            response.setStatus(HttpServletResponse.SC_OK);
        }
        catch(Exception e)
        {
            // An error occured.  This could be a JSONException, a
MocaException, or
            // another exception.  We have to handle them.

            if(e instanceof MocaException)
            {
                // We have a MocaException, which we know is caused by a
bad request
                response.setStatus(HttpServletResponse.SC_BAD_REQUEST);
            }
            else if(e instanceof MocaException)
            {
                // We have a JSONException, which we know is caused by some
coding
                // error.  Since there is no HTTP Response Status for this
we
                // will simply return an internal server error.

response.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
            }
            else
            {
                // Something other than a MocaException or JSONException.
Since we do not know what
                // the error is, just tell the receiver that we have an
internal error.

response.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
            }
        }
```

```
    }
...
```

POST