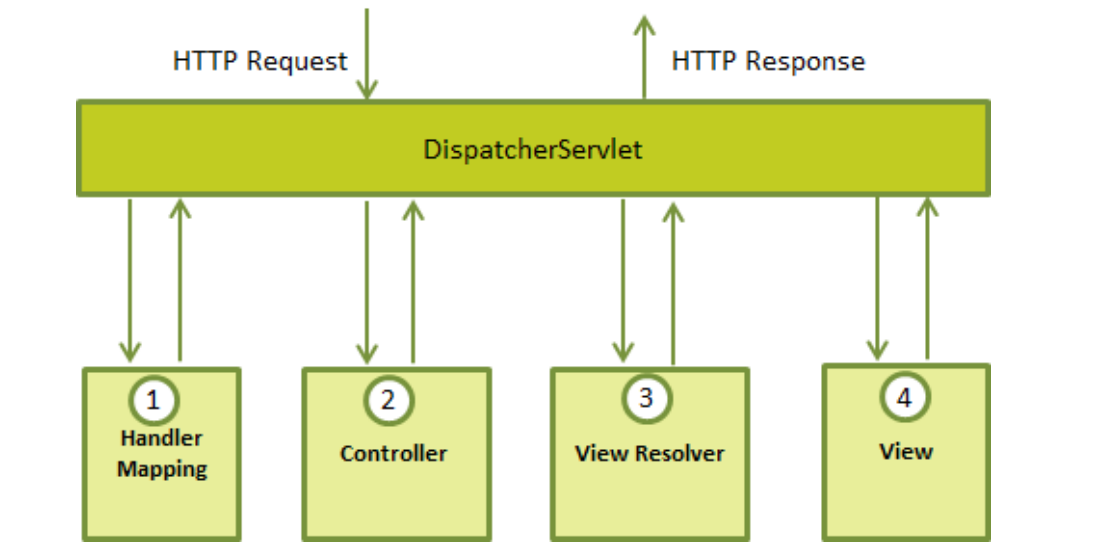
SpringMVC

The Spring web MVC framework provides model-view-controller architecture and ready components that can be used to develop flexible and loosely coupled web applications. The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

* The **Model** encapsulates the application data and in general they will consist of POJO.
* The **View** is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.
* The **Controller** is responsible for processing user requests and building appropriate model and passes it to the view for rendering.

# The DispatcherServlet

The Spring Web model-view-controller (MVC) framework is designed around a DispatcherServlet that handles all the HTTP requests and responses. The request processing workflow of the Spring Web MVC DispatcherServlet is illustrated in the following diagram:



Following is the sequence of events corresponding to an incoming HTTP request to DispatcherServlet:

1. After recieving an HTTP request, DispatcherServlet consults the HandlerMapping to call the appropriate Controller.

2. The Controller takes the request and calls the appropriate service methods based on used GET or POST method. The service method will set model data based on defined business logic and returns view name to the DispatcherServlet.

3. The DispatcherServlet will take help from ViewResolver to pick up the defined view for the request.

4. Once view is finalized, The DispatcherServlet passes the model data to the view which is finally rendered on the browser.

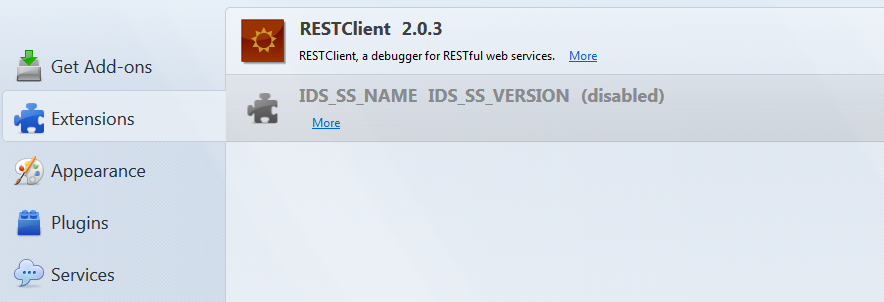
All the above mentioned components, ie. HandlerMapping, Controller and ViewResolver are parts of WebApplicationContext which is an extension of the plain ApplicationContext with some extra features necessary for web applications.

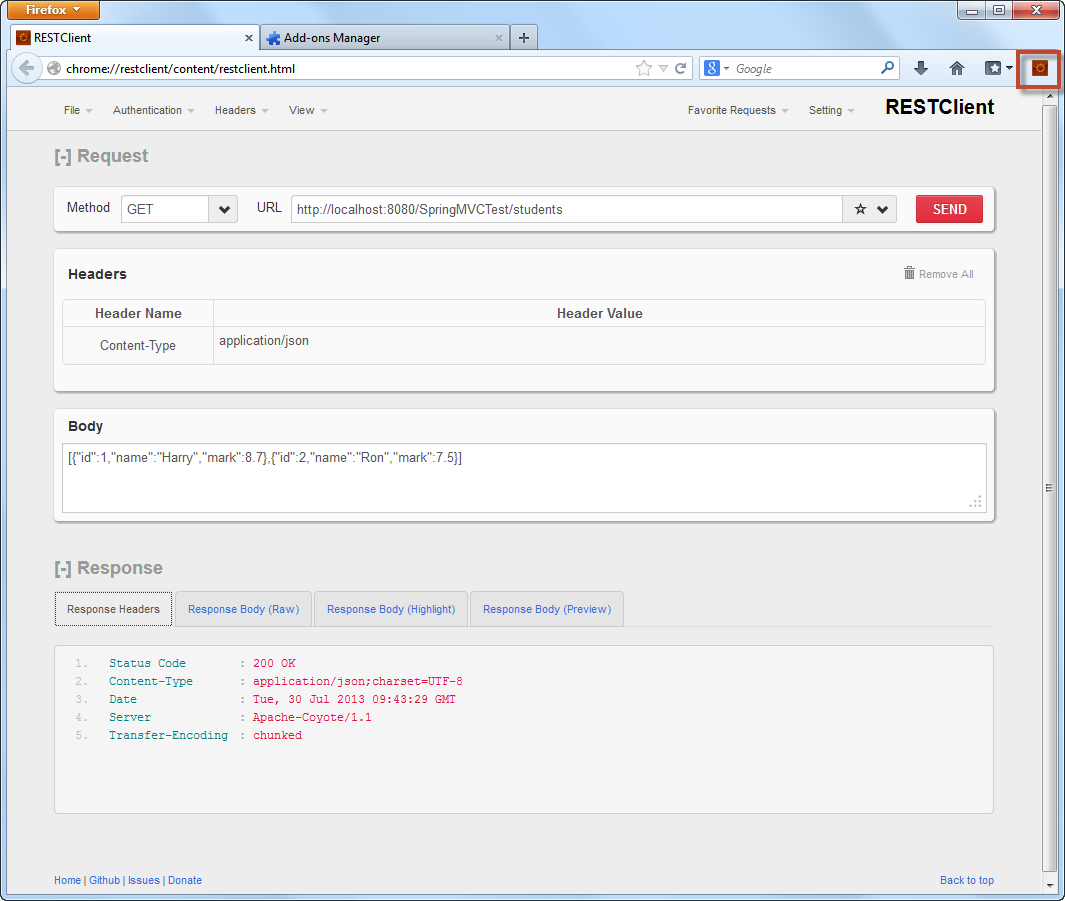
In this document, we will know how to make RESTful Web services with SpringMVC then you don’t need to know all about SpringMVC.

# Tools

Before moving to next steps, you should prepare these tools:

* **IDE:** Eclipse
* **Test tool:** Mozilla with add-ons RESTClient





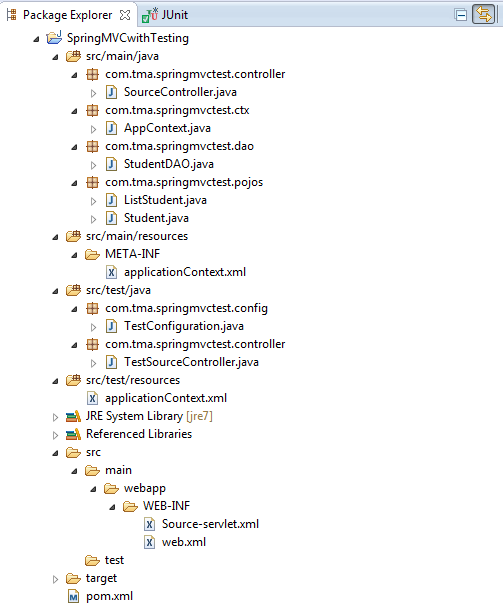
# Build RESTful Web services with SpringMVC

In this session, you learn how to make a RESTful Web services with SpringMVC.

This project provides four RESTful Web services to get all students, register a new student, get a student by id and delete a list of students. To make this project simpler, we will hard code the data access object (DAO).

## Step 1 – Build a project

Build a project with this structure with Maven, this structure includes these packages:



## Step 2 – Dependencies and Configuration

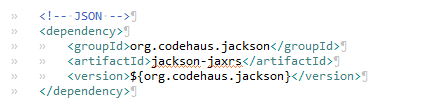
### Dependencies

Dependencies you need to declare in pom.xml:

* **Spring:**

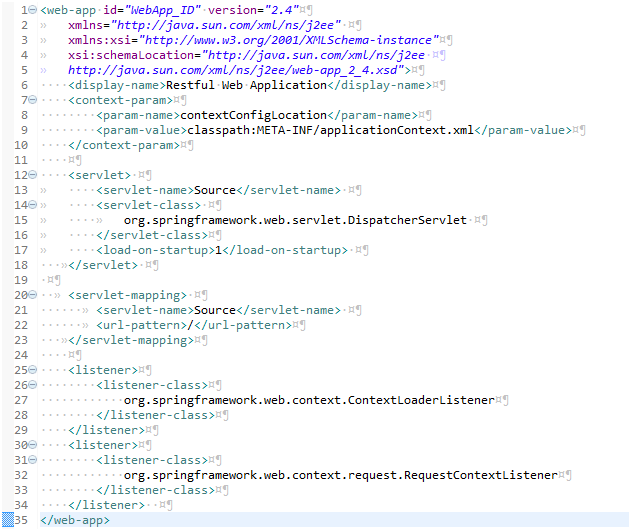


* **JSON:**



### Configuration

#### web.xml

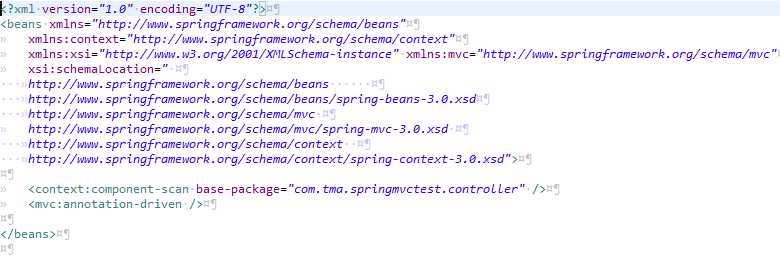


In web.xml, you have tag **<servlet>** to define the servlet using in this project. This tag has **<servlet-name>** which defines the name of the Servlet and by default, project will load configuration of this servlet in **<servlet name>-servlet.xml**. In this case, it loads file Source-servlet.xml

The **<url-pattern>** defines the root of api to contact with this servlet.

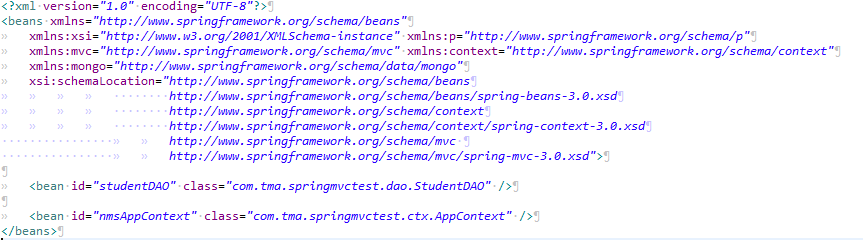
Tag **<context-param>** references the configuration of this project in META-INF/applicationContext.xml.

#### <servlet name>-servlet.xml



In this file, you define two tags: <context> and <mvc: annotation-driven />. Tag <context> defines the location of package where you store the controllers; projects will scan this package to get the api. Tag **<mvc:annotation-driven />** defines this controller will be use annotation of springmvc.

#### applicationContext.xml



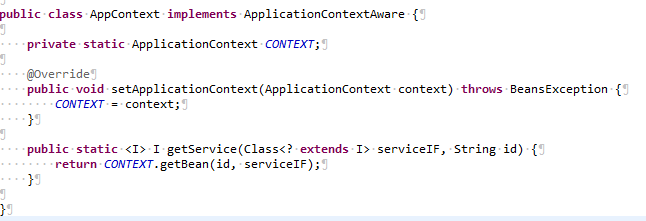
This file defines beans used in the project. In this case, you define studentDAO and nmsAppContext.

## Step 3: Create Controller

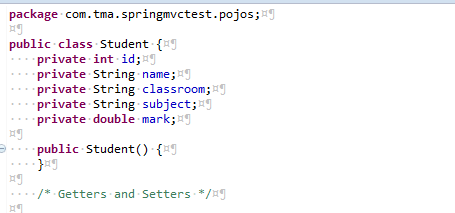
#### POJOS and DAO

Before create controller, you have to create pojos and DAO as following code:

##### AppContext.java



##### Student.java



##### StudentDAO.java



#### Controller

When you define a servlet with SpringMVC, all things you need to control are its annotation, you could see those through the example and there are list of them:

**@Controller**

**@RequestMapping**

**@RequestParam**

**@RequestBody**

**@ResponseBody**

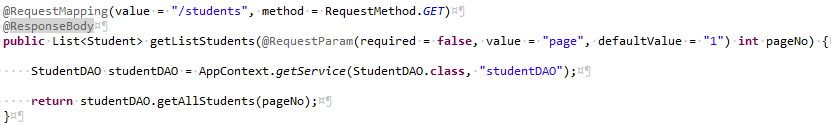
##### Get list of students

First, you create a class named SourceController with annotation @Controller



**@Controller:** let the project knows this class is controller.

Next, you create an api for users to get all the students.



In this api, you have three annotations @RequestMapping, @ResponseBody and @RequestParam. Let’s have a look to these annotations:

**@RequestMapping:** provide api for client, **value** defines the uri for the outsiders and **method** defines method which clients use. If you define the uri before class definition, it will become root uri for all the following methods.

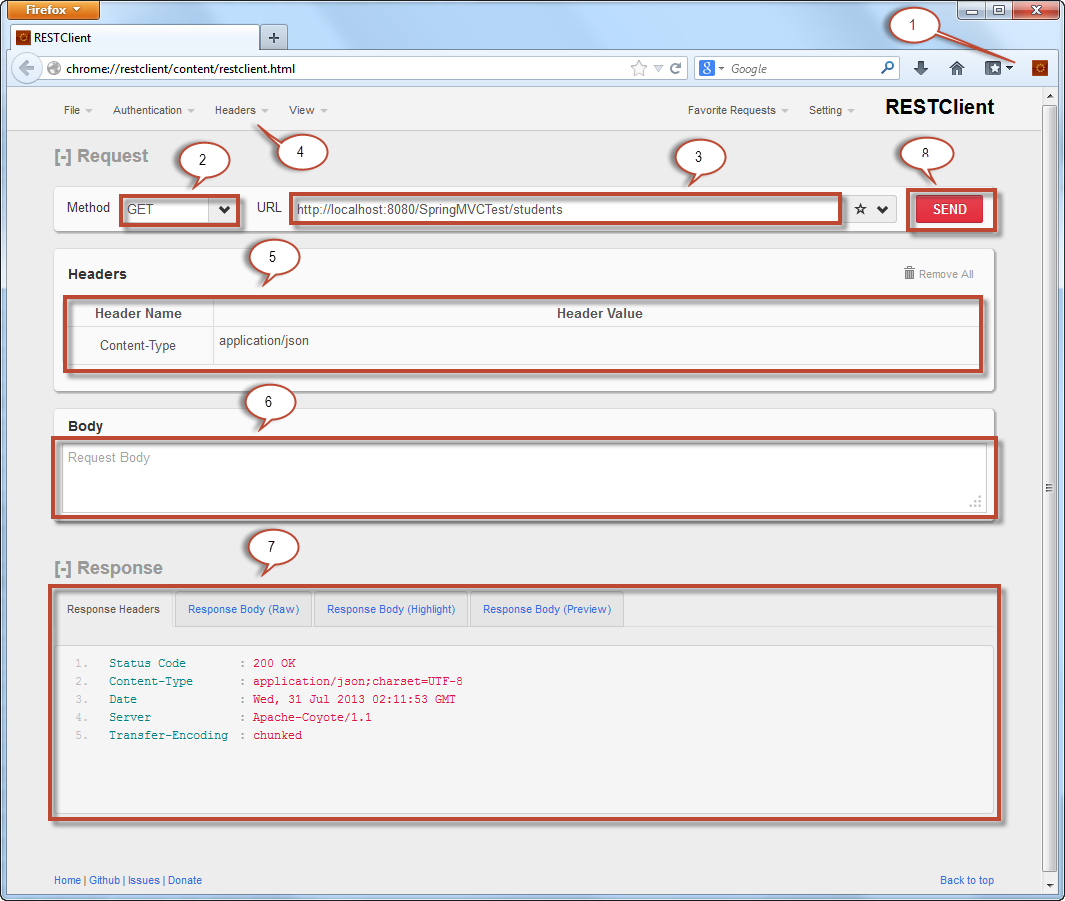
**@ResponseBody:** announce the client that this method could return the value (such as JSON, XML or CSV…). In this case, the api will return list of Student object. Because of *importing library of JSON*, then it will parse the list into array of JSON by default.

**@RequestParam:** when you need some parameters in uri, such as “www.springtest.abc/body?page=1”, you need to use this annotation. This annotation supports three parameters:

* required: accepts two values true and false. If it is true, the parameter must be present in the uri. Otherwise, an exception is thrown. By default, its value is true.
* value: defines the name of parameter on uri, if this parameter is absent, it will get name of variable by default value.
* defaultValue: only accept String value. This parameter is needed when the parameter is not required, it will get this value when parameter is not defined.

In some case, you need multiple parameters with the same name, such as ordering “body?orderby=name&orderby=type”, you need to define the variable as an array, such as “String order[]”.

Now, let’s test this function with RESTful Client add-on:

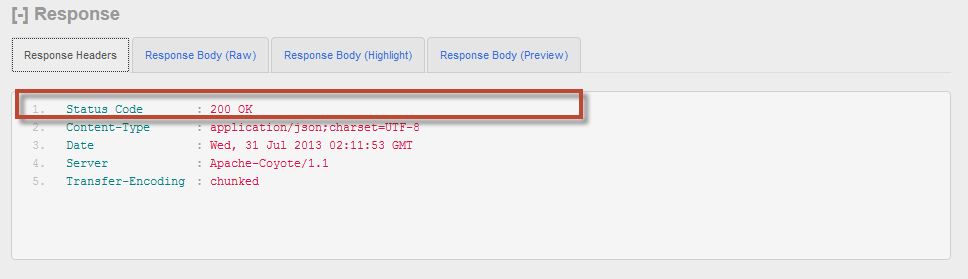


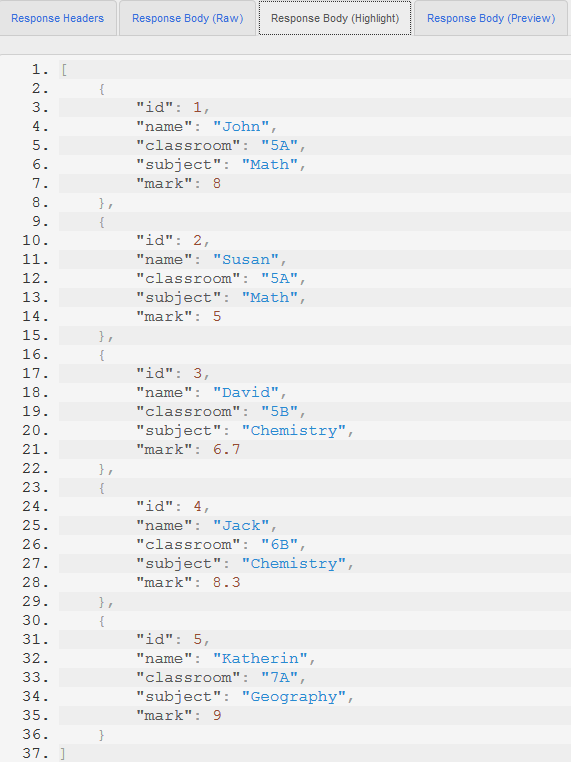
1. After installing RESTClient, click here to start.
2. Select the HTTP method to use.
3. URI of the Web service.
4. Select / create the header of the request / response.
5. Current headers are being used.
6. The body of the request.
7. The response from server with some format and filter.
8. After filling the necessary information, click here to send request.

This is result from the api that you have just created:



Result:

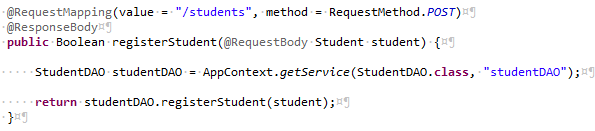




##### Register a student

Next, you create an api with method POST to create a new student, results from this api is true or false.

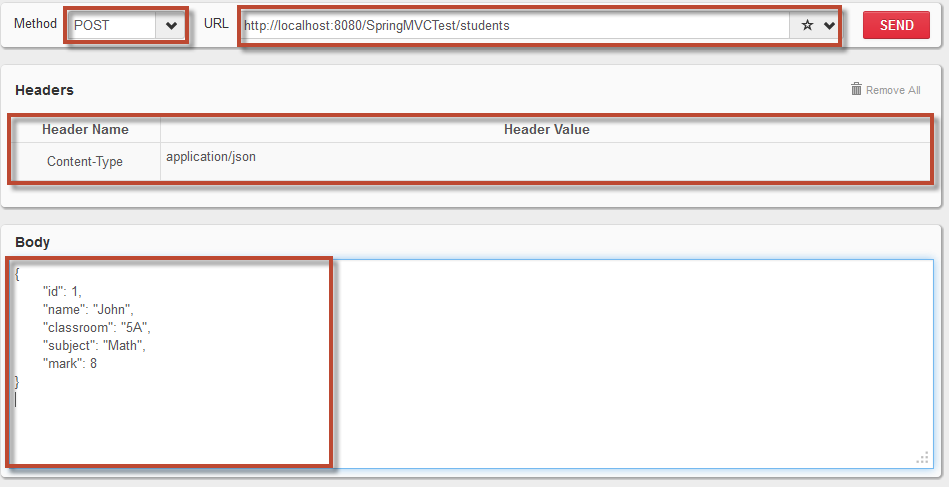
SourceController.java



In this api, you use RequestMethod.POST and annotation @RequestBody

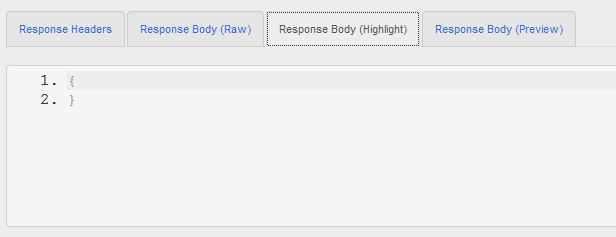
**@RequestBody:** this is the body of the request, defined in JSON format in this case. SpringMVC can parse from JSON to object automatically for you.

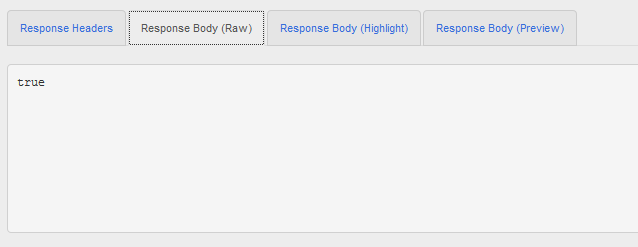
This time, you return a simple value to client instead of a complex Class. Then, the clients see value only in response body, not in JSON type.



Result:



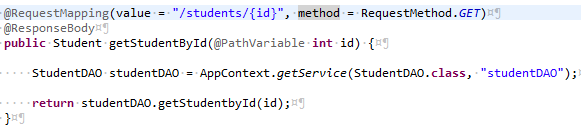




##### Get a student by id

Now you provide an ipa to get information of a specific student through id.

SourceController.java

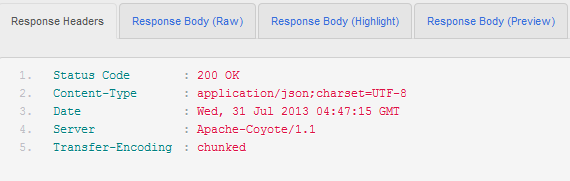


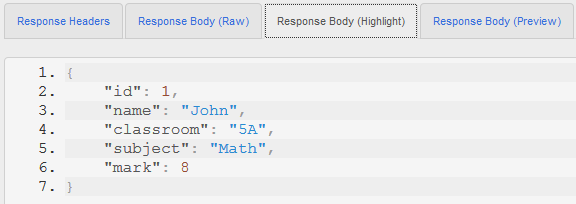
In this api, you use annotation @PathVariable. With RESTful Web services, you create a simple api and this is how it implements. In the other hand, you return to client a java class object and SpringMVC will parse it to a JSON object.

**@PathVariable**: use to recognize the variable in the uri. In the path, you specify the variable in format {<variable name>}. This annotation provide attribute name to get specific variable whether it is different from local variable.



Result:

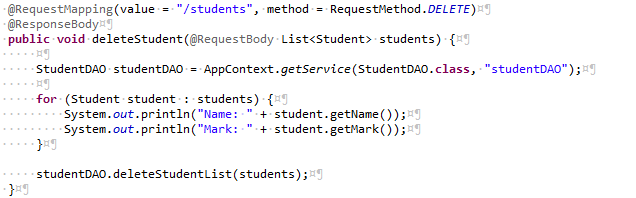




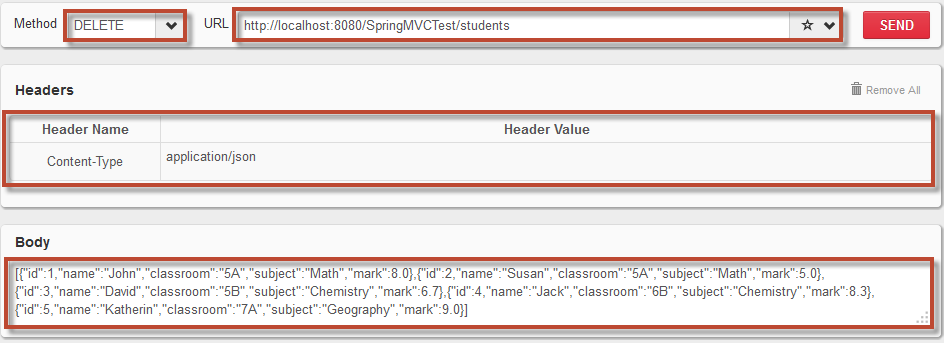
##### Delete list of students

Finally, you provide an api for client to delete a list of students.

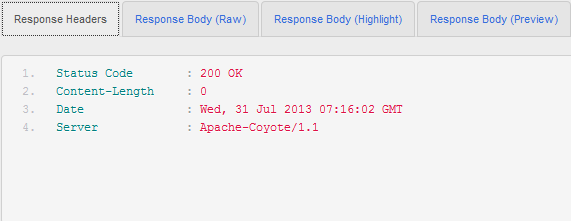
SourceController.java



This time, you use procedure instead of function to implement this api but you still use annotation @ResponseBody. If you don’t, you could get error when calling this api.

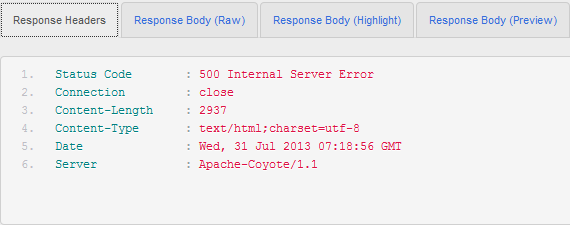


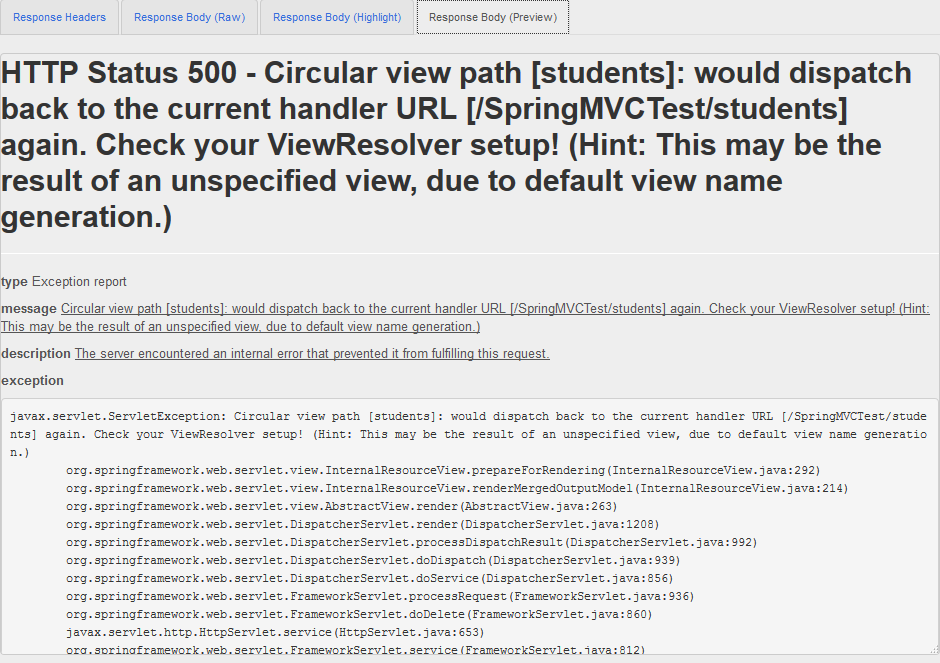
Result:



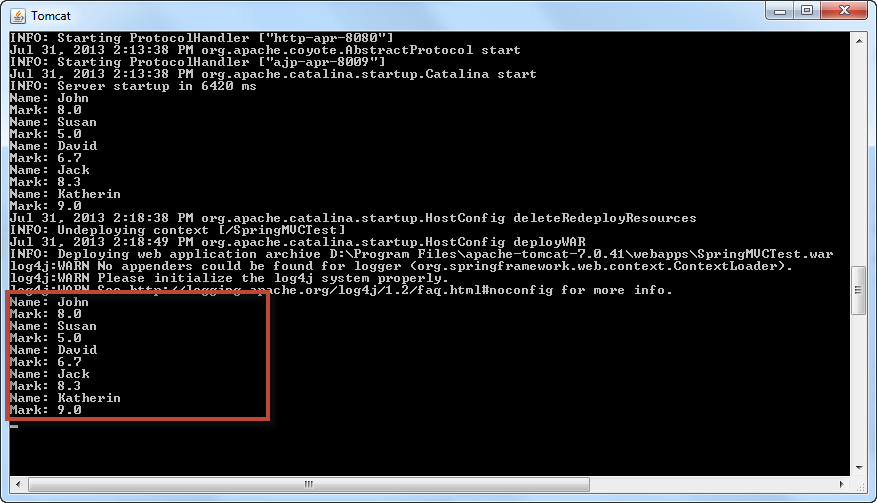
Note:

1. If you don’t have annotation @ResponseBody in the api, you might get this error:



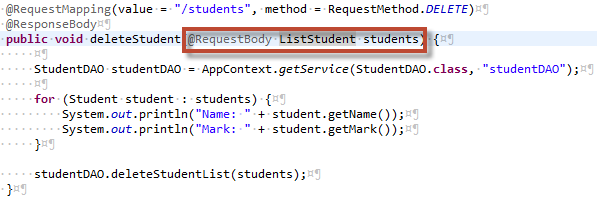


While this api is executing normally.



1. In some case, SpringMVC could not get List of object, then you need to build another class extends ArrayList to use, like:





Now you can create your RESTful Web services with SpringMVC.

# SpringMVC Testing

In this guide, you learn how to perform unit test of SpringMVC using MockMVC with Mokito.

## Step 1 – Dependencies and Configuration

### Dependencies



### Configuration

You need to create an applicationContext.xml in test/resources.

Note: if you use fix path to get beans in servlet, you need to create context with same name and put it in the same path in test.

#### applicationContext.xml



Now, you need to test Service layer, then you don’t care about what happened in the layer below (DAO), that’s why you use Mockito. You create these beans with the same name you used in the service and set path to you real DAO in constructor.

## Step 2 – Create Unit Test

For each Controller, you should have a class to perform unit test and put it in the same package in test. In this case, you create a class with name TestSourceController.java.

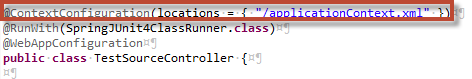
To create a unit test of SpringMVC, you should create a class with these steps:

### Import static class



### Configure class

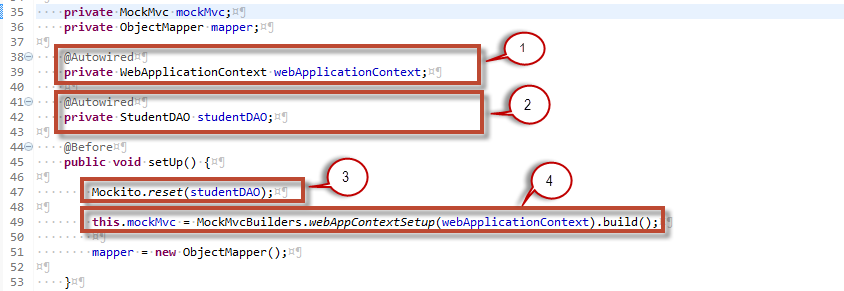
**TestSourceController.java**



You need to set the path to the context file used to test.

### Class variables and initial

**TestSourceController.java**



1. Prepare the configuration of your class as a web application with api.
2. Get your necessary DAO in this unit test.
3. Reset the Mokito.
4. Get mockMVC with the configuration.

With annotation @Before, you initiate the value each time it perform a test and rest the Mockito to avoid the memory leak or the result of the Mockito might become unpredictable.

### Unit test 1

Get started with the first unit test using mockMVC and Mockito.

In this test, you verify the api which returns a list of object in JSON type with method GET and you test another case of this api with parameter. Now, create a method with annotation @Test to build this test case.

**TestSourceController.java**



From line 58 to 68, you prepare the value which DAOs have to return. On line 67 and 68, you use method *when()* which provide by Mockito to set the result of the method.

From line 70 to 71, you prepare the results which should be returned form server when you call the api. For example, you call (“/students”), it will return a list of students in JSON format, then you prepare the result in JSON format. You can use class ObjectMapper of org.codehaus.jackson.map to help you resolve this case.

From line 74 to 85, you create the test case to test the api. The first test is from line 75 to 79, you use action *perform()* to call the uri like you use in web browser. Class MockMvcRequestBuilders which you has imported before has provide method to call all the HTTP method like get, post, put and delete, then you can use them to match with your demand. On line 76, you check the status got back when you call the uri. MockMvcResultMatchers you have imported provides method *status()* to predict this case. On line 77 you get the content in the response, you use this content to compare with the case you have prepared before.

Now you have result which you get from calling the uri and the expected value. There is one problem here; the format is in JSON type, which the exchange of variable is possible. You have to use method assert() of JSONAssert to compare these value with the strict is false.

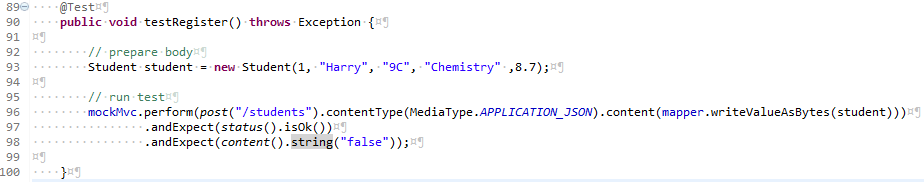
From line 81 to 85, you have a test case with parameter. You can set the parameter to the uri which format is the same as the way you set path variable. The others are the same as above.

Moving to next test, you perform the test with parameter in format the same as you use when you define path variable.

### Unit test 2

Next, you create unit test for the request with content and check the response from server which is not in JSON type.

**TestSourceController.java**



This is similar with the test you’ve done before. The addition you need to get here is the content of request and the return of response is not the same. You can set the content of the request in method *content()*  which has shown in line 96. In this test, the return is simple String, you could get it out and use normal Assert of junit to check or you could check it directly in the simpler way. MockMvcResultMatchers provides method *content()* to expect the content of response.

The others test case is similar to these test cases and you can create them by your own.

These are what you need to know about SpringMVC in RESTful Web services and how to test them. I hope those information could be helpful.