

# Spring MVC 4 RESTful Web Services CRUD Example+RestTemplate

Created on: August 11, 2015 | Last updated on: November 14, 2016 [websystiqueadmin](#)

In this post we will write a CRUD Restful WebService using Spring MVC 4, and write a REST client with RestTemplate to consume those services. We will also test those services using external clients.

## Short & Quick introduction to REST

**REST** stands for Representational State Transfer. It's an architectural style which can be used to design web services, that can be consumed from a variety of clients. The core idea is that, rather than using complex mechanisms such as CORBA, RPC or SOAP to connect between machines, simple HTTP is used to make calls among them.

In Rest based design, resources are being manipulated using a common set of verbs.

- To Create a resource : HTTP POST should be used
- To Retrieve a resource : HTTP GET should be used
- To Update a resource : HTTP PUT should be used
- To Delete a resource : HTTP DELETE should be used

That means, you as a REST service developer or Client, should comply to above criteria, in order to be REST compliant.

Often Rest based Web services return JSON or XML as response, although it is not limited to these types only. Clients can specify (using HTTP **Accept header**) the resource type they are interested in, and server may return the resource , specifying **Content-Type** of the resource it is serving.

### Other interesting posts you may like

- [Spring 4 MVC+JPA2+Hibernate Many-to-many Example](#)
  - [Secure Spring REST API using OAuth2](#)
  - [AngularJS+Spring Security using Basic Authentication](#)
  - [Secure Spring REST API using Basic Authentication](#)
- [ecaching-tutorial/">ecaching-tutorial/](#)>Spring 4 Caching Annotations Tutorial

- [Spring 4 Email Template Library Example](#)
- [Spring 4 Cache Tutorial with EhCache](#)
- [Spring 4 Email With Attachment Tutorial](#)
- [Spring 4 Email Integration Tutorial](#)
- [Spring MVC 4+JMS+ActiveMQ Integration Example](#)
- [Spring 4+JMS+ActiveMQ @JmsListener @EnableJms Example](#)

- [Spring 4+JMS+ActiveMQ Integration Example](#)
- [Spring MVC 4+Apache Tiles 3 Integration Example](#)
- [Spring MVC 4+Spring Security 4 + Hibernate Integration Example](#)
- [Spring MVC 4+AngularJS Example](#)
- [Spring MVC 4+AngularJS Routing with ngRoute Example](#)
- [Spring MVC 4+AngularJS Routing with UI-Router Example](#)
- [Spring MVC 4+Hibernate 4 Many-to-many JSP Example](#)
- [Spring MVC 4+Hibernate 4+MySQL+Maven integration example using annotations](#)
- [Spring MVC4 FileUpload-Download Hibernate+MySQL Example](#)
- [TestNG Mockito Integration Example Stubbing Void Methods](#)
- [Maven surefire plugin and TestNG Example](#)
- [Spring MVC 4 Form Validation and Resource Handling](#)

## Rest Based Controller

Following is one possible Rest based controller, implementing REST API. I said possible, means Other's may implement it in another way, still (or even more pure way) conforming to REST style.

**This is what our REST API does:**

- **GET** request to /api/user/ returns a list of users
- **GET** request to /api/user/1 returns the user with ID 1
- **POST** request to /api/user/ with a user object as JSON creates a new user
- **PUT** request to /api/user/3 with a user object as JSON updates the user with ID 3
- **DELETE** request to /api/user/4 deletes the user with ID 4
- **DELETE** request to /api/user/ deletes all the users

```
package com.websystique.springmvc.controller;
```

```
import java.util.List;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PathVariable;
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.RestController;
import org.springframework.web.util.UriComponentsBuilder;
```

```
import com.websystique.springmvc.model.User;
import com.websystique.springmvc.service.UserService;
```

```
@RestController
public class HelloWorldRestController {
```

```
    @Autowired
```

```

    UserService userService; //Service which will do all data
retrieval/manipulation work

//-----Retrieve All Users-----
-----

@RequestMapping(value = "/user/", method = RequestMethod.GET)
public ResponseEntity<List<User>> listAllUsers() {
    List<User> users = userService.findAllUsers();
    if (users.isEmpty()) {
        return new ResponseEntity<List<User>> (HttpStatus.NO_CONTENT); //You
many decide to return HttpStatus.NOT_FOUND
    }
    return new ResponseEntity<List<User>> (users, HttpStatus.OK);
}

//-----Retrieve Single User-----
-----

@RequestMapping(value = "/user/{id}", method = RequestMethod.GET,
produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<User> getUser(@PathVariable("id") long id) {
    System.out.println("Fetching User with id " + id);
    User user = userService.findById(id);
    if (user == null) {
        System.out.println("User with id " + id + " not found");
        return new ResponseEntity<User> (HttpStatus.NOT_FOUND);
    }
    return new ResponseEntity<User> (user, HttpStatus.OK);
}

//-----Create a User-----
-----

@RequestMapping(value = "/user/", method = RequestMethod.POST)
public ResponseEntity<Void> createUser(@RequestBody User
user, UriComponentsBuilder ucBuilder) {
    System.out.println("Creating User " + user.getName());

    if (userService.isUserExist(user)) {
        System.out.println("A User with name " + user.getName() + "
already exist");
        return new ResponseEntity<Void> (HttpStatus.CONFLICT);
    }

    userService.saveUser(user);

    HttpHeaders headers = new HttpHeaders();

```

```

        headers.setLocation(ucBuilder.path("/user/{id}").buildAndExpand(user.
getId()).toUri());
        return new ResponseEntity<Void>(headers, HttpStatus.CREATED);
    }

```

```

//----- Update a User -----
-----

```

```

    @RequestMapping(value = "/user/{id}", method = RequestMethod.PUT)
    public ResponseEntity<User> updateUser(@PathVariable("id") long id,
    @RequestBody User user) {
        System.out.println("Updating User " + id);

        User currentUser = userService.findById(id);

        if (currentUser==null) {
            System.out.println("User with id " + id + " not found");
            return new ResponseEntity<User>(HttpStatus.NOT_FOUND);
        }

        currentUser.setName(user.getName());
        currentUser.setAge(user.getAge());
        currentUser.setSalary(user.getSalary());

        userService.updateUser(currentUser);
        return new ResponseEntity<User>(currentUser, HttpStatus.OK);
    }

```

```

//----- Delete a User -----
-----

```

```

    @RequestMapping(value = "/user/{id}", method = RequestMethod.DELETE)
    public ResponseEntity<User> deleteUser(@PathVariable("id") long id) {
        System.out.println("Fetching & Deleting User with id " + id);

        User user = userService.findById(id);
        if (user == null) {
            System.out.println("Unable to delete. User with id " + id + " not
found");
            return new ResponseEntity<User>(HttpStatus.NOT_FOUND);
        }

        userService.deleteUserById(id);
        return new ResponseEntity<User>(HttpStatus.NO_CONTENT);
    }

```

```

//----- Delete All Users -----
-----

```

```

    @RequestMapping(value = "/user/", method = RequestMethod.DELETE)

```

```

    public ResponseEntity<User> deleteAllUsers() {
        System.out.println("Deleting All Users");

        userService.deleteAllUsers();
        return new ResponseEntity<User>(HttpStatus.NO_CONTENT);
    }
}

```

### Detailed Explanation :

**@RestController** : First of all, we are using Spring 4's new **@RestController** annotation. This annotation eliminates the need of annotating each method with **@ResponseBody**. Under the hood, **@RestController** is itself annotated with **@ResponseBody**, and can be considered as combination of **@Controller** and **@ResponseBody**.

**@RequestBody** : If a method parameter is annotated with **@RequestBody**, Spring will bind the incoming HTTP request body (for the URL mentioned in **@RequestMapping** for that method) to that parameter. While doing that, Spring will [behind the scenes] use [HTTP Message converters](#) to convert the HTTP request body into domain object [deserialize request body to domain object], based on ACCEPT or Content-Type header present in request.

**@ResponseBody** : If a method is annotated with **@ResponseBody**, Spring will bind the return value to outgoing HTTP response body. While doing that, Spring will [behind the scenes] use [HTTP Message converters](#) to convert the return value to HTTP response body [serialize the object to response body], based on Content-Type present in request HTTP header. As already mentioned, in Spring 4, you may stop using this annotation.

**ResponseEntity** is a real deal. It represents the entire HTTP response. Good thing about it is that you can control anything that goes into it. You can specify status code, headers, and body. It comes with several constructors to carry the information you want to sent in HTTP Response.

**@PathVariable** This annotation indicates that a method parameter should be bound to a URI template variable [the one in '{}'].

Basically, **@RestController**, **@RequestBody**, **ResponseEntity** & **@PathVariable** are all you need to know to implement a REST API in Spring 4. Additionally, spring provides several support classes to help you implement something customized.

**MediaType** : With **@RequestMapping** annotation, you can additionally, specify the **MediaType** to be produced or consumed (using **produces** or **consumes** attributes) by that particular controller method, to further narrow down the mapping.

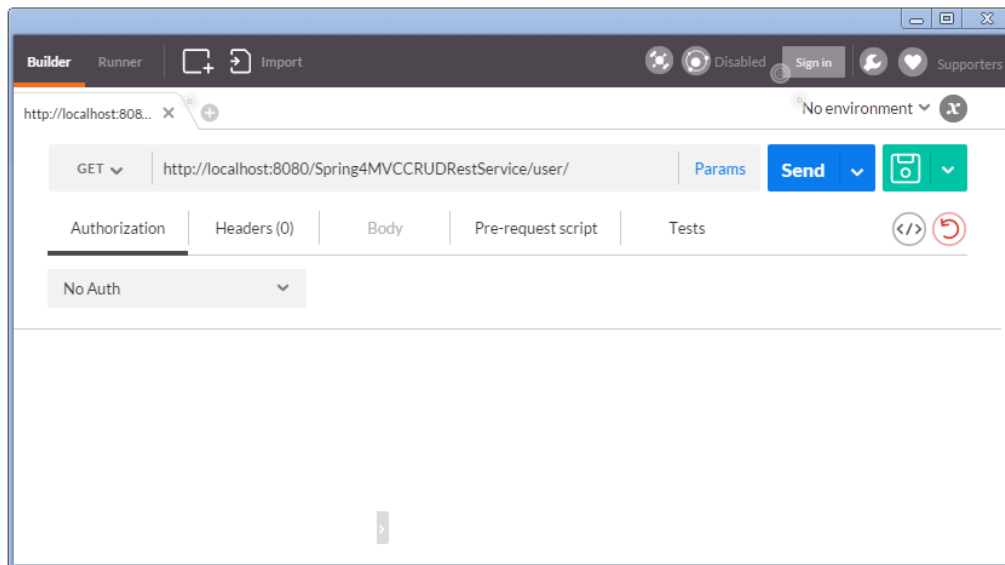
### Deploy and Test this API, let's dig deeper into how this thing works

At the at end of day, it's just a plain controller class, part of a deploy-able application. [Complete downloadable application code is shown further down in post which you can deploy straight-away in your container]. I am going to deploy it, in order to see things live and discuss each operation in detail. Deployed Application is available at <http://localhost:8080/Spring4MVCCRUDRestService>.

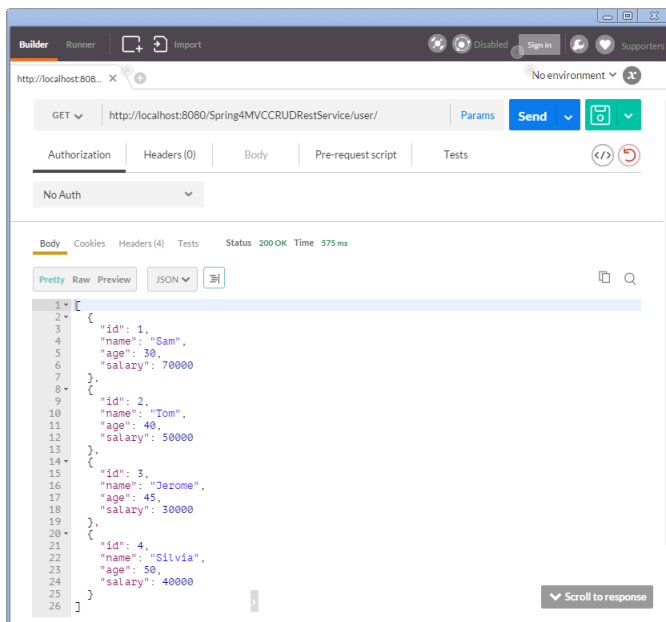
To test this API, i will use an external client POSTMAN (An extension from CHROME). We will write our own client in just few minutes.

## 1. Retrieve all users

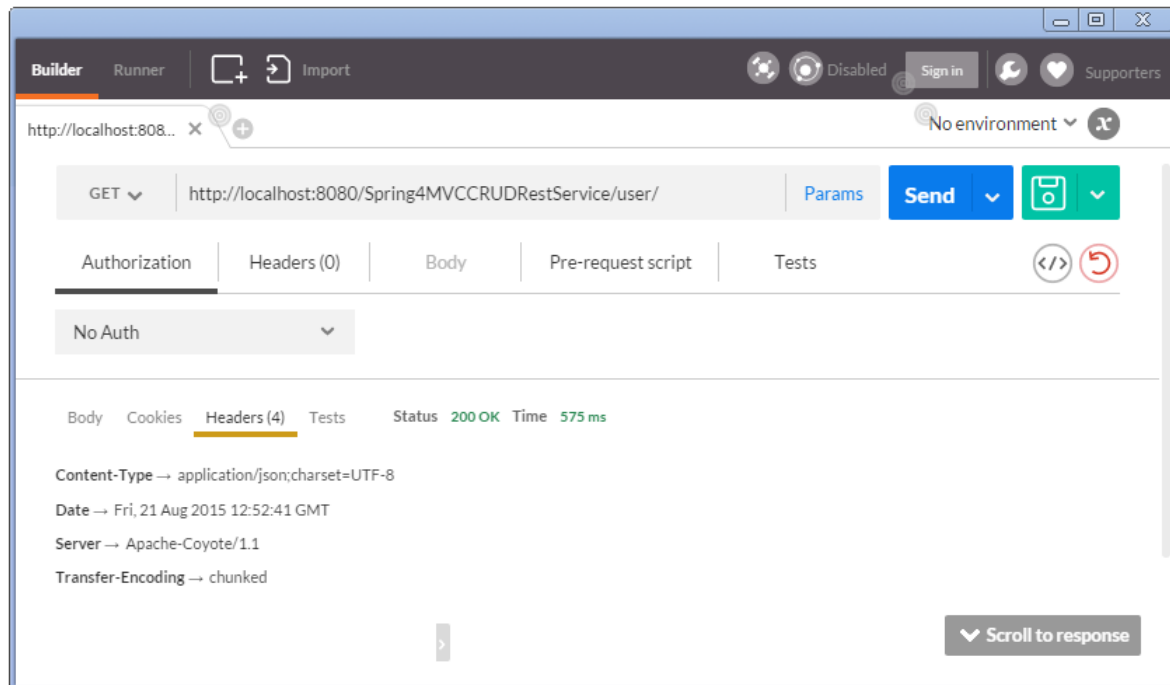
Open POSTMAN tool, select request type [GET for this usecase], specify the operation uri.



Notice that we did not specify any HTTP header here. Click on Send, you will receive list of all users.



Also notice the HTTP 200 response. Additionally check headers.



You might be wondering how the response is sent as JSON string, and the Content-Type header in response confirms that. Glad you asked. This is due to the fact that we have included Jackson library in our project.

```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.5.3</version>
</dependency>
```

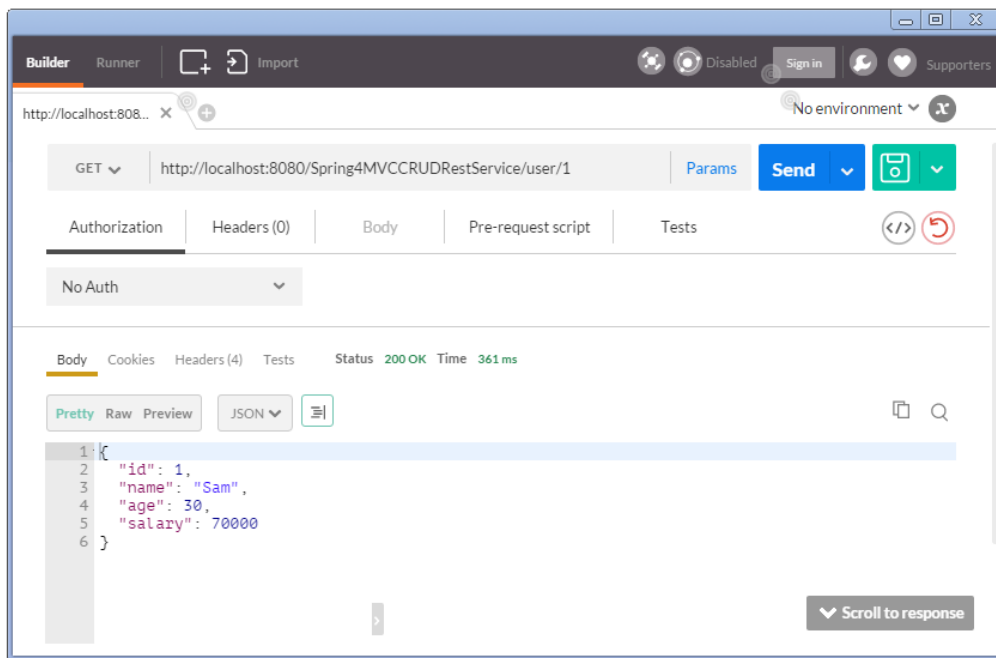
Since spring finds this library in class path, it invokes

inbuilt [MappingJackson2HttpMessageConverter](#) converter to convert the response (List of objects) into JSON.

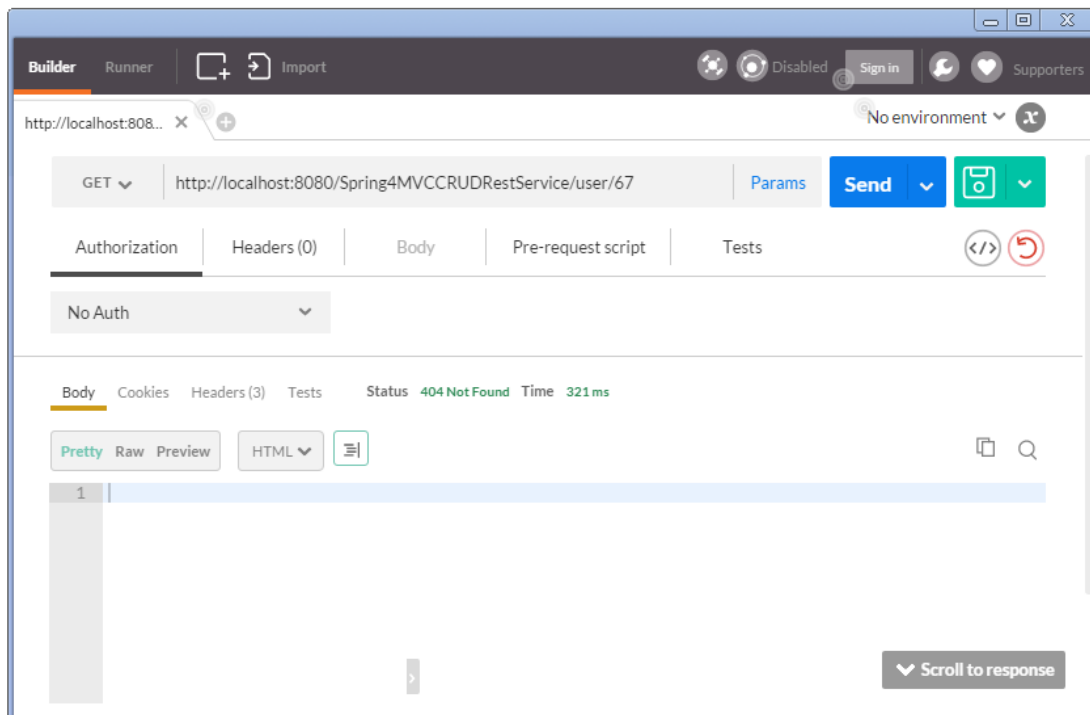
Good thing about Spring inbuilt converters are that most of the time they just need certain library in classpath in order to perform conversion. Of course sometime we do need to adapt our API/application as well. For instance, if we want to serve XML as well, we should annotate User class with proper JAXB annotations.

## 2. Retrieve Single User

Specify a GET with /user/1 , click on send.



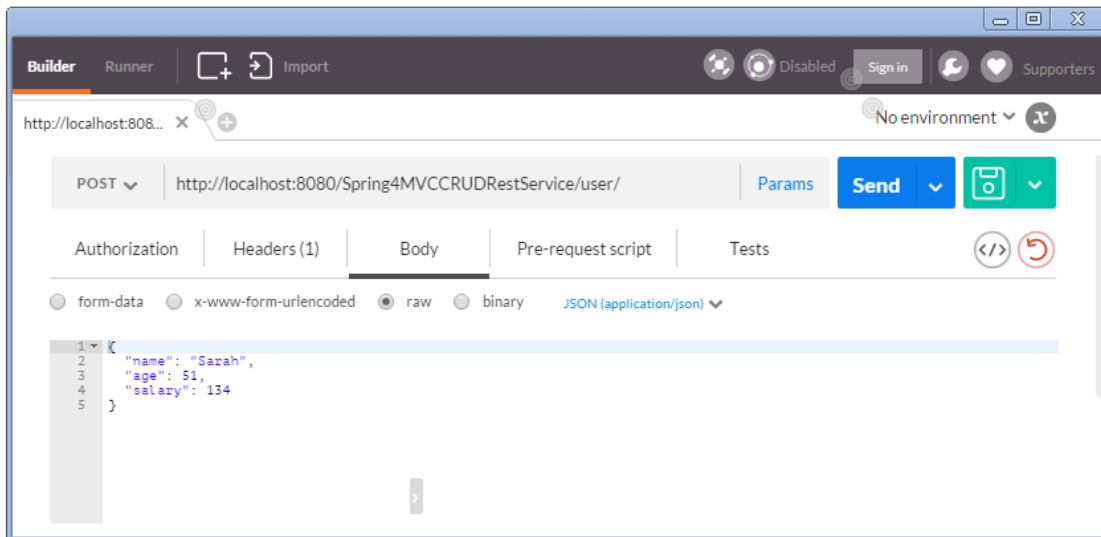
Now try to send a GET with invalid identifier, you should receive a HTTP 404.



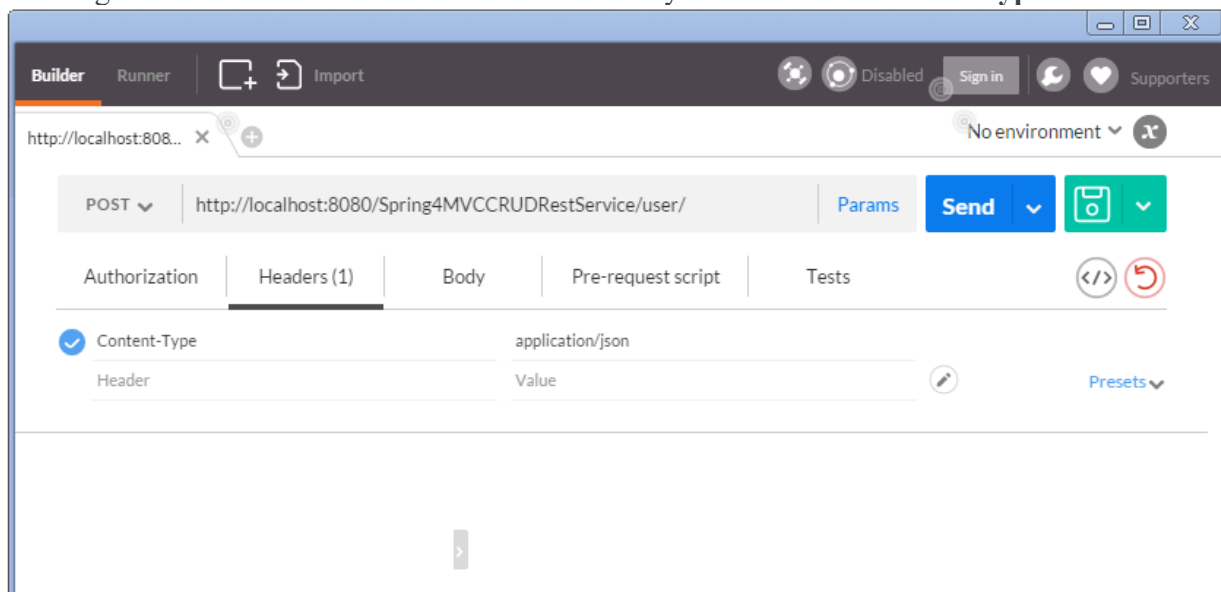


### 3. Create a User

Select the method as POST, specify uri as /user/, specify body in POSTMAN body tab, select the type [application/json].



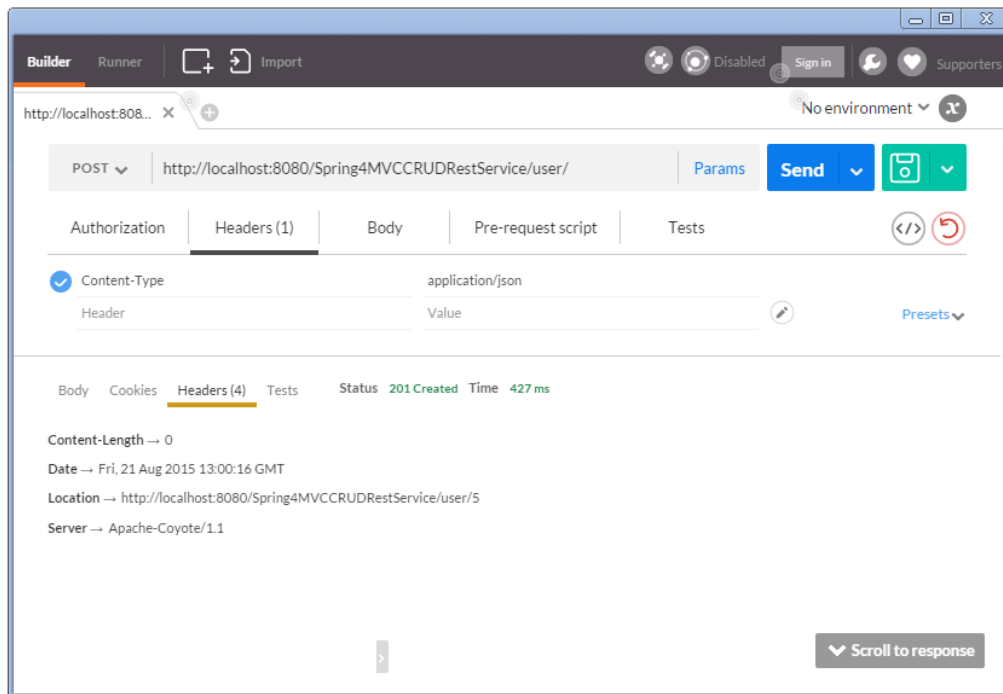
You might have noticed that POSTMAN automatically adds a header **Content-Type**.



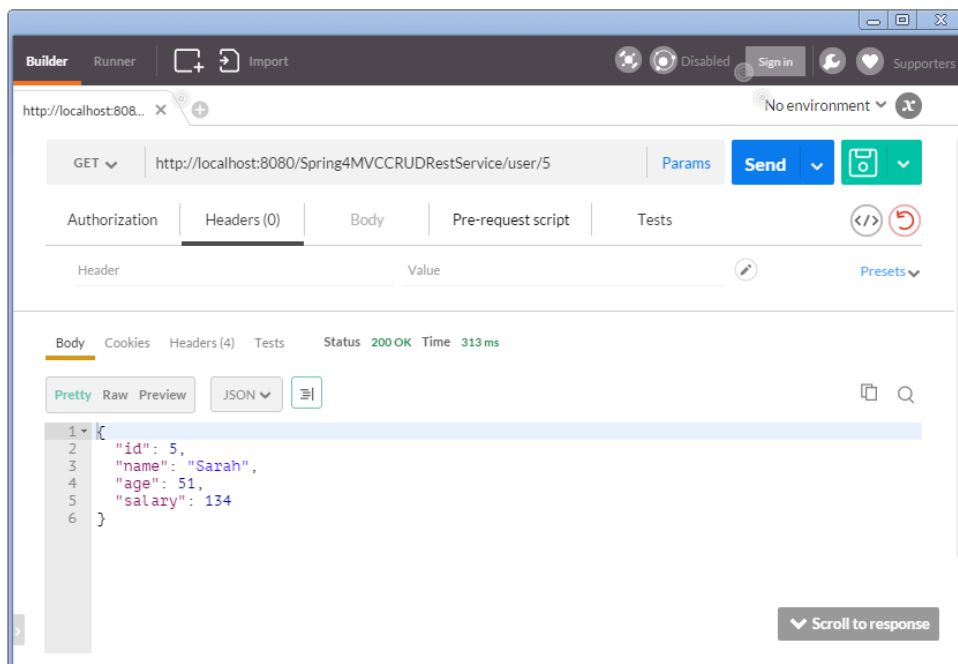
Along with POST and PUT request, clients send the data to the server and they should specify the actual content type of the data being sent.

**Remember :** Accept header says about what type client can understand. Content-Type header says what type of data actually is of.

Send. You should see HTTP 200 response with no body (as API don't send anything in body). But you should find a **Location header** specifying the location the newly created user can be found at.

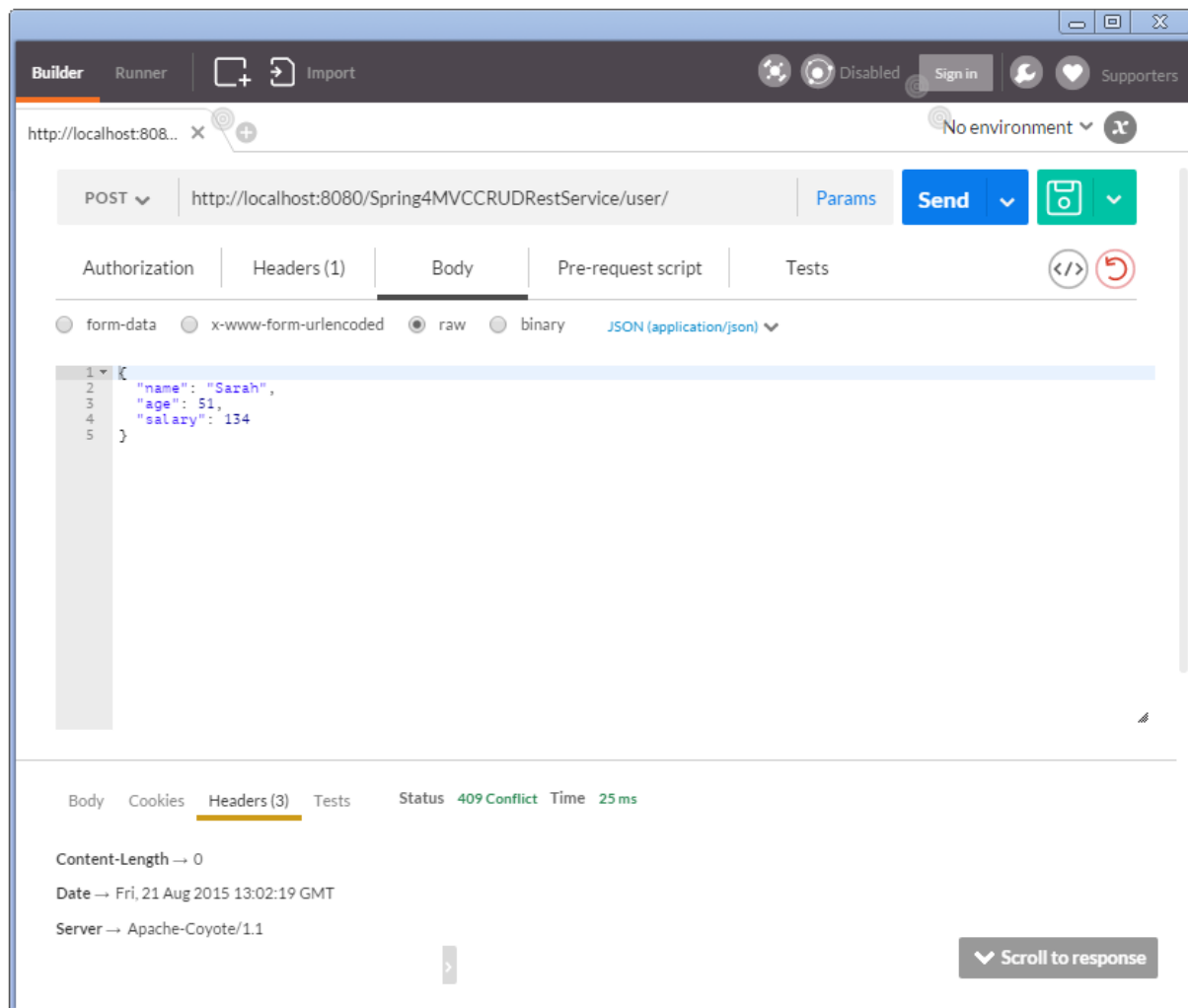


You can now fetch the newly created user.



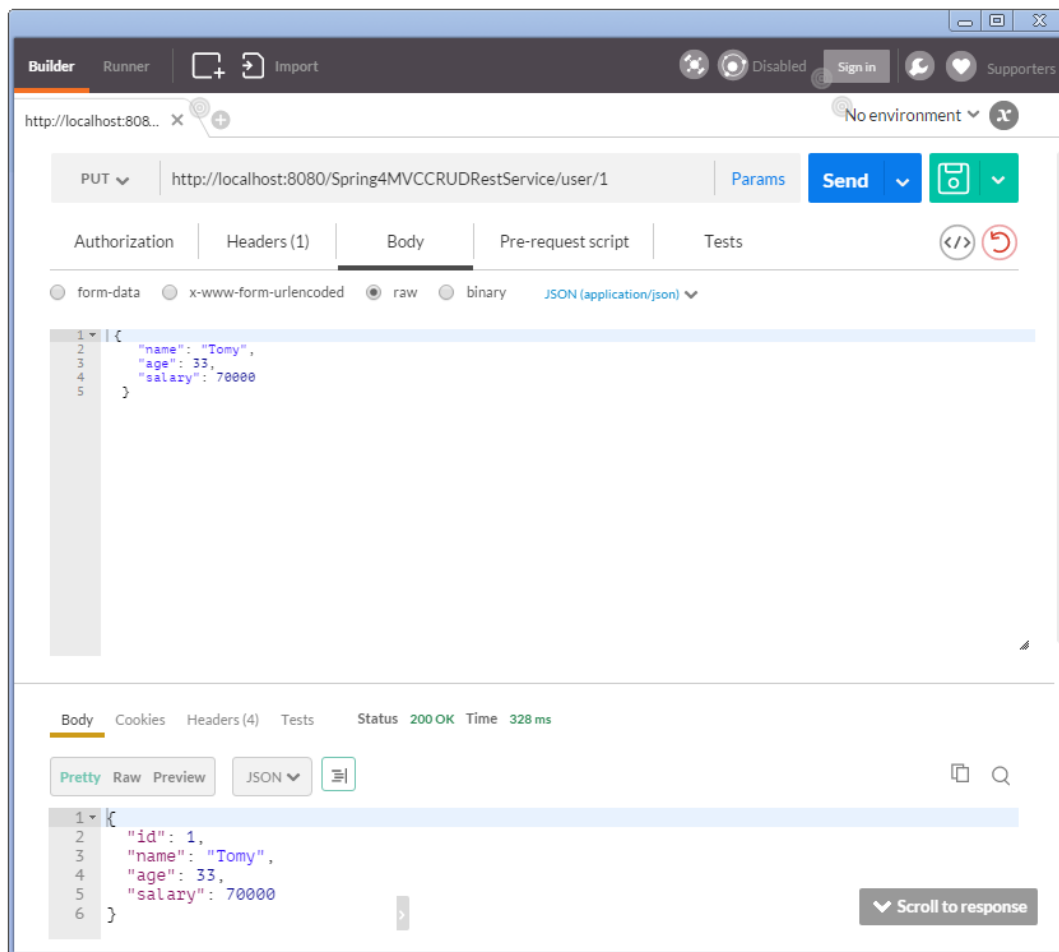
This way of implementation is common in REST. But no one stops you if you do want to send the content in Response body of a POST/PUT request. Will that still be REST compliant API? It's a debatable point.

Create the same user again. You should get HTTP Conflict response.



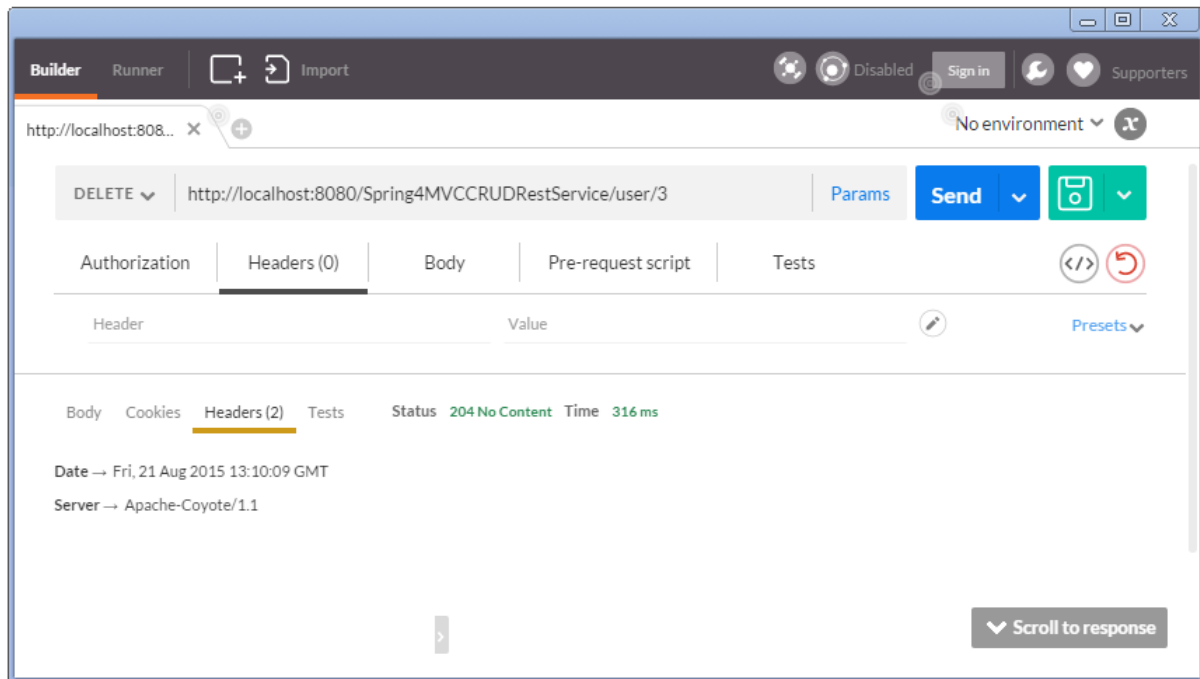
#### 4. Update a User

Send a HTTP PUT request to update a user. Send along the new user details to be put in.

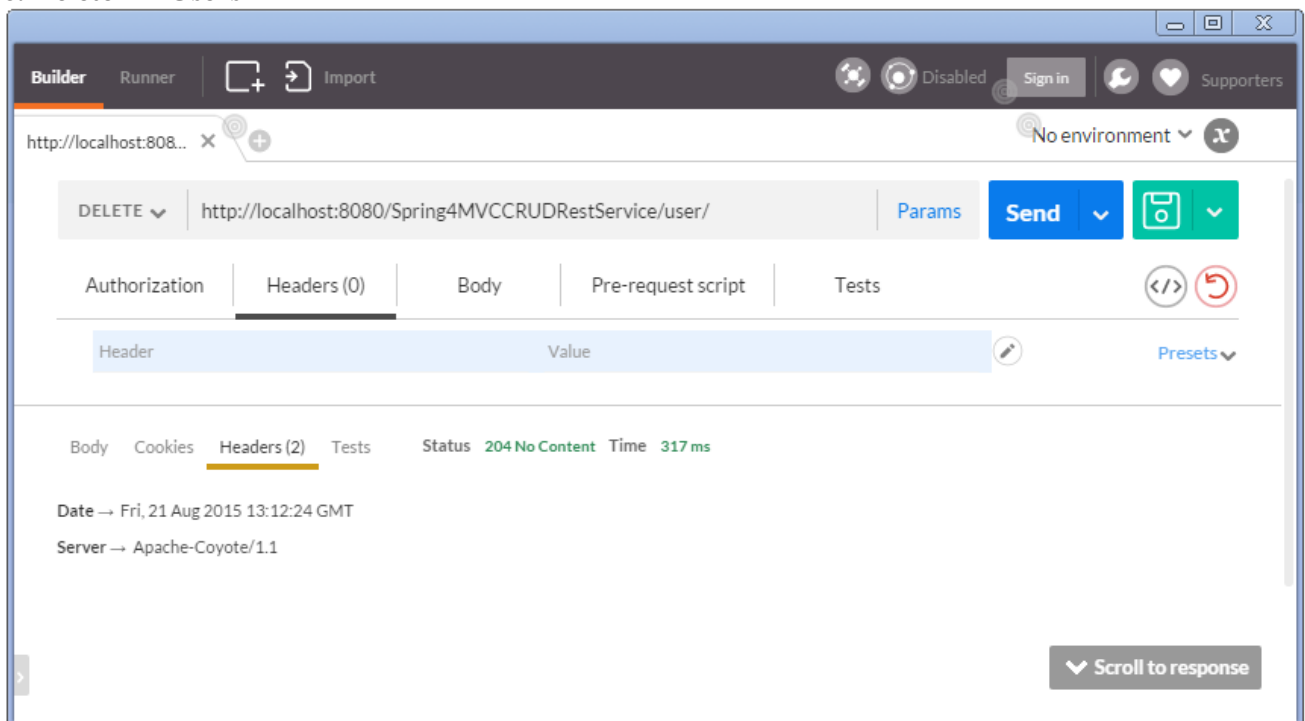


Notice that we have received response body this time. This is because the method implementation in controller is sending it. Again, one may decide not to send the updated details in response body, and just send the location header(as in create).

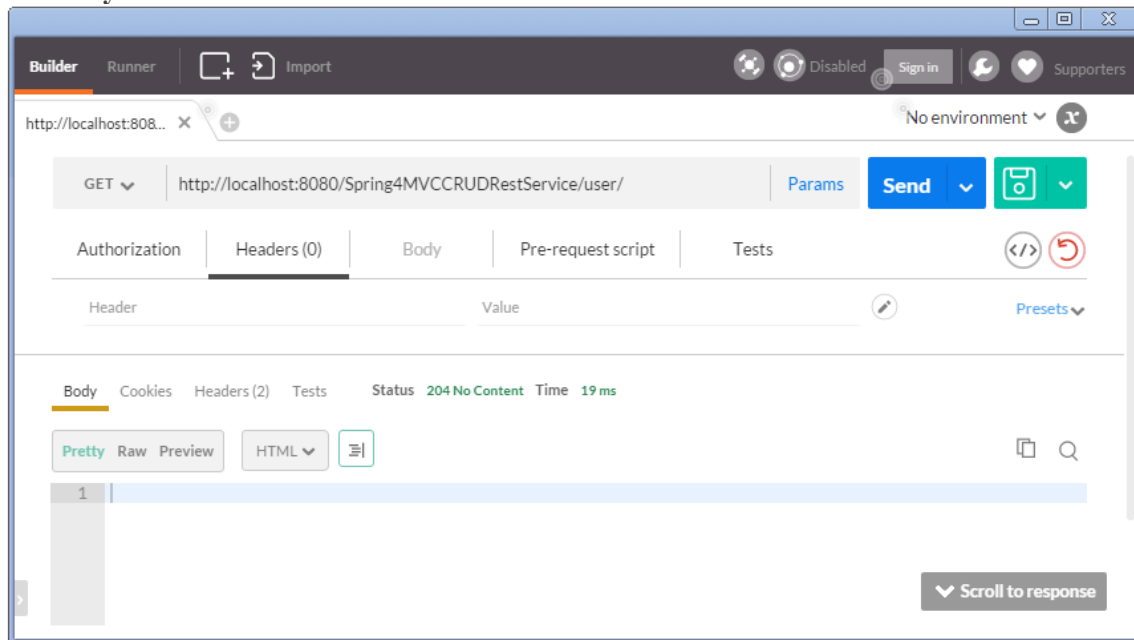
## 5. Delete a User



## 6. Delete All Users



## 7. Verify users after delete-all



## Writing REST Client using RestTemplate

Postman tool we used above is a wonderful Client to test Rest API. But if you want to consume REST based web services from your application, you would need a REST client for your application. One of the most popular HTTP client is Apache HttpComponents **HttpClient**. But the details to access REST services using this are too low level.

Spring's **RestTemplate** comes to Rescue. RestTemplate provides higher level methods that correspond to each of the six main HTTP methods that make invoking many RESTful services a one-liner and enforce REST best practices.

Below shown are HTTP methods and corresponding RestTemplate methods to handle that type of HTTP request.

### HTTP Methods and corresponding RestTemplate methods:

- HTTP GET : getObject, getForEntity
- HTTP PUT : put(String url, Object request, String...urlVariables)
- HTTP DELETE : delete
- HTTP POST : postForLocation(String url, Object request, String... urlVariables), postForObject(String url, Object request, Class responseType, String... uriVariables)
- HTTP HEAD : headForHeaders(String url, String... urlVariables)
- HTTP OPTIONS : optionsForAllow(String url, String... urlVariables)
- HTTP PATCH and others : exchange execute

### Custom Rest client , consuming the REST services created earlier.

```
package com.websystique.springmvc;

import java.net.URI;
import java.util.LinkedHashMap;
import java.util.List;

import org.springframework.web.client.RestTemplate;

import com.websystique.springmvc.model.User;

public class SpringRestTestClient {

    public static final String REST_SERVICE_URI =
"http://localhost:8080/Spring4MVCCRUDRestService";

    /* GET */
    @SuppressWarnings("unchecked")
    private static void listAllUsers() {
        System.out.println("Testing listAllUsers API-----");

        RestTemplate restTemplate = new RestTemplate();
        List<LinkedHashMap<String, Object>> usersMap =
restTemplate.getObject(REST_SERVICE_URI+"/user/", List.class);
```

```

        if(usersMap!=null){
            for(LinkedHashMap<String, Object> map : usersMap){
                System.out.println("User : id="+map.get("id")+",
Name="+map.get("name")+", Age="+map.get("age")+",
Salary="+map.get("salary"));
            }
        }else{
            System.out.println("No user exist-----");
        }
    }

    /* GET */
    private static void getUser(){
        System.out.println("Testing getUser API-----");
        RestTemplate restTemplate = new RestTemplate();
        User user = restTemplate.getForObject(REST_SERVICE_URI+"/user/1",
User.class);
        System.out.println(user);
    }

    /* POST */
    private static void createUser() {
        System.out.println("Testing create User API-----");
        RestTemplate restTemplate = new RestTemplate();
        User user = new User(0,"Sarah",51,134);
        URI uri = restTemplate.postForLocation(REST_SERVICE_URI+"/user/",
user, User.class);
        System.out.println("Location : "+uri.toASCIIString());
    }

    /* PUT */
    private static void updateUser() {
        System.out.println("Testing update User API-----");
        RestTemplate restTemplate = new RestTemplate();
        User user = new User(1,"Tomy",33, 70000);
        restTemplate.put(REST_SERVICE_URI+"/user/1", user);
        System.out.println(user);
    }

    /* DELETE */
    private static void deleteUser() {
        System.out.println("Testing delete User API-----");
        RestTemplate restTemplate = new RestTemplate();
        restTemplate.delete(REST_SERVICE_URI+"/user/3");
    }

    /* DELETE */
    private static void deleteAllUsers() {
        System.out.println("Testing all delete Users API-----");
        RestTemplate restTemplate = new RestTemplate();
        restTemplate.delete(REST_SERVICE_URI+"/user/");
    }

```



```

    public static void main(String args[]){
        listAllUsers();
        getUser();
        createUser();
        listAllUsers();
        updateUser();
        listAllUsers();
        deleteUser();
        listAllUsers();
        deleteAllUsers();
        listAllUsers();
    }
}

```

Restart server(In our example, data on server side is fixed.). Run above program.

### Output from above Client program

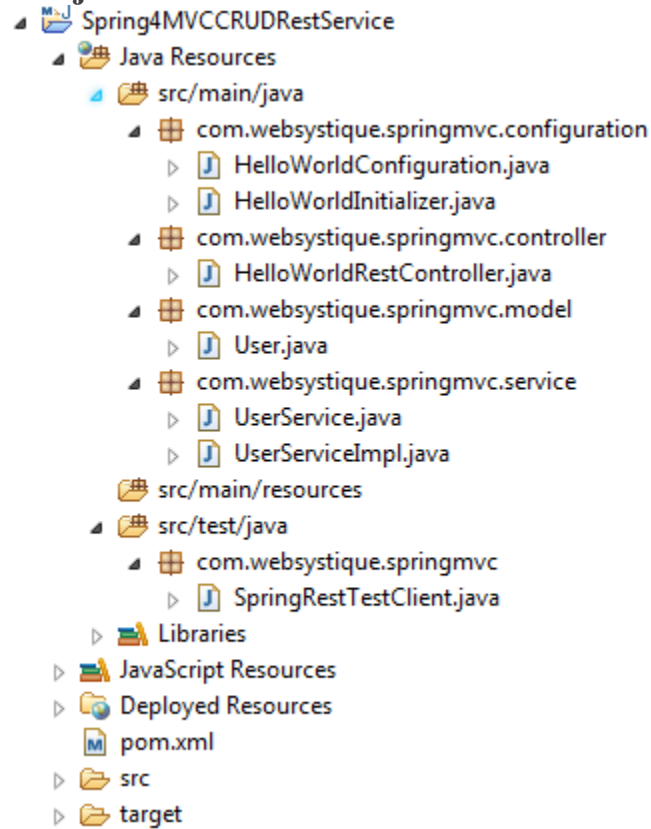
```

Testing listAllUsers API-----
User : id=1, Name=Sam, Age=30, Salary=70000.0
User : id=2, Name=Tom, Age=40, Salary=50000.0
User : id=3, Name=Jerome, Age=45, Salary=30000.0
User : id=4, Name=Silvia, Age=50, Salary=40000.0
Testing getUser API-----
User [id=1, name=Sam, age=30, salary=70000.0]
Testing create User API-----
Location : http://localhost:8080/Spring4MVCCRUDRestService/user/5
Testing listAllUsers API-----
User : id=1, Name=Sam, Age=30, Salary=70000.0
User : id=2, Name=Tom, Age=40, Salary=50000.0
User : id=3, Name=Jerome, Age=45, Salary=30000.0
User : id=4, Name=Silvia, Age=50, Salary=40000.0
User : id=5, Name=Sarah, Age=51, Salary=134.0
Testing update User API-----
User [id=1, name=Tomy, age=33, salary=70000.0]
Testing listAllUsers API-----
User : id=1, Name=Tomy, Age=33, Salary=70000.0
User : id=2, Name=Tom, Age=40, Salary=50000.0
User : id=3, Name=Jerome, Age=45, Salary=30000.0
User : id=4, Name=Silvia, Age=50, Salary=40000.0
User : id=5, Name=Sarah, Age=51, Salary=134.0
Testing delete User API-----
Testing listAllUsers API-----
User : id=1, Name=Tomy, Age=33, Salary=70000.0
User : id=2, Name=Tom, Age=40, Salary=50000.0
User : id=4, Name=Silvia, Age=50, Salary=40000.0
User : id=5, Name=Sarah, Age=51, Salary=134.0
Testing all delete Users API-----
Testing listAllUsers API-----
No user exist-----

```

# Complete Example

## Project Structure



## Declare project dependencies

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.websystique.springmvc</groupId>
  <artifactId>Spring4MVCCRUDRestService</artifactId>
  <packaging>war</packaging>
  <version>1.0.0</version>
  <name>Spring4MVCCRUDRestService Maven Webapp</name>

  <properties>
    <springframework.version>4.2.0.RELEASE</springframework.version>
    <jackson.version>2.5.3</jackson.version>
  </properties>

  <dependencies>
    <dependency>
```

```

        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>${springframework.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-tx</artifactId>
        <version>${springframework.version}</version>
    </dependency>

    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>${jackson.version}</version>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.1.0</version>
    </dependency>

</dependencies>

<build>
    <pluginManagement>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.2</version>
                <configuration>
                    <source>1.7</source>
                    <target>1.7</target>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-war-plugin</artifactId>
                <version>2.4</version>
                <configuration>
                    <warSourceDirectory>src/main/webapp</warSourceDirecto
ry>
                    <warName>Spring4MVCCRUDRestService</warName>
                    <failOnMissingWebXml>false</failOnMissingWebXml>
                </configuration>
            </plugin>
        </plugins>
    </pluginManagement>

    <finalName>Spring4MVCCRUDRestService</finalName>
</build>
</project>

```

## User Service

```
package com.websystique.springmvc.service;

import java.util.List;

import com.websystique.springmvc.model.User;

public interface UserService {

    User findById(long id);

    User findByName(String name);

    void saveUser(User user);

    void updateUser(User user);

    void deleteUserById(long id);

    List<User> findAllUsers();

    void deleteAllUsers();

    public boolean isUserExist(User user);

}

package com.websystique.springmvc.service;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.concurrent.atomic.AtomicLong;

import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import com.websystique.springmvc.model.User;

@Service("userService")
@Transactional
public class UserServiceImpl implements UserService{

    private static final AtomicLong counter = new AtomicLong();

    private static List<User> users;

    static{
        users= populateDummyUsers();
    }
}
```

```

    }

    public List<User> findAllUsers() {
        return users;
    }

    public User findById(long id) {
        for(User user : users){
            if(user.getId() == id){
                return user;
            }
        }
        return null;
    }

    public User findByName(String name) {
        for(User user : users){
            if(user.getName().equalsIgnoreCase(name)){
                return user;
            }
        }
        return null;
    }

    public void saveUser(User user) {
        user.setId(counter.incrementAndGet());
        users.add(user);
    }

    public void updateUser(User user) {
        int index = users.indexOf(user);
        users.set(index, user);
    }

    public void deleteUserById(long id) {

        for (Iterator<User> iterator = users.iterator(); iterator.hasNext(); ) {
            User user = iterator.next();
            if (user.getId() == id) {
                iterator.remove();
            }
        }
    }

    public boolean isUserExist(User user) {
        return findByName(user.getName()) != null;
    }

    private static List<User> populateDummyUsers() {
        List<User> users = new ArrayList<User>();
        users.add(new User(counter.incrementAndGet(), "Sam", 30, 70000));
        users.add(new User(counter.incrementAndGet(), "Tom", 40, 50000));
        users.add(new User(counter.incrementAndGet(), "Jerome", 45, 30000));
    }

```

```

        users.add(new User(counter.incrementAndGet(),"Silvia",50, 40000));
        return users;
    }

    public void deleteAllUsers() {
        users.clear();
    }
}

```

## Model class

```
package com.websystique.springmvc.model;
```

```

public class User {

    private long id;

    private String name;

    private int age;

    private double salary;

    public User() {
        id=0;
    }

    public User(long id, String name, int age, double salary){
        this.id = id;
        this.name = name;
        this.age = age;
        this.salary = salary;
    }

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {

```

```

        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public double getSalary() {
        return salary;
    }

    public void setSalary(double salary) {
        this.salary = salary;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + (int) (id ^ (id >>> 32));
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        User other = (User) obj;
        if (id != other.id)
            return false;
        return true;
    }

    @Override
    public String toString() {
        return "User [id=" + id + ", name=" + name + ", age=" + age
            + ", salary=" + salary + "]";
    }

}

```

## Configuration class

```

package com.websystique.springmvc.configuration;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;

```

```

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "com.websystique.springmvc")
public class HelloWorldConfiguration {

}

```

## Initialization Class

```

package com.websystique.springmvc.configuration;

import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

public class HelloWorldInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class[] { HelloWorldConfiguration.class };
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return null;
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }

}

```

## Adding CORS support to your REST API

While accessing the REST API, you might face issues concerning [Same Origin Policy](#).

Errors like :

” No ‘Access-Control-Allow-Origin’ header is present on the requested resource. Origin

‘http://127.0.0.1:8080’ is therefore not allowed access.” OR

” XMLHttpRequest cannot load http://abc.com/bla. Origin http://localhost:12345 is not allowed by Access-Control-Allow-Origin.” are common in such case.

Solution is [Cross-Origin Resource Sharing](#). Basically, on server side, we can return additional CORS access control headers with response, which will eventually allow further inter-domain communication.

With Spring, we can write a simple filter which adds those CORS specific headers in each response.

```

package com.websystique.springmvc.configuration;

import java.io.IOException;

import javax.servlet.Filter;

```



```

import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequestResponse;

public class CORSFilter implements Filter {

    public void doFilter(ServletRequest req, ServletResponse res, FilterChain
chain) throws IOException, ServletException {
        System.out.println("Filtering
on.....");
        HttpServletResponse response = (HttpServletResponse) res;
        response.setHeader("Access-Control-Allow-Origin", "*");
        response.setHeader("Access-Control-Allow-Methods", "POST, GET, PUT,
OPTIONS, DELETE");
        response.setHeader("Access-Control-Max-Age", "3600");
        response.setHeader("Access-Control-Allow-Headers", "x-requested-
with");
        chain.doFilter(req, res);
    }

    public void init(FilterConfig filterConfig) {}

    public void destroy() {}

}

```

Then we can simply configure it in our Spring configuration like shown below:

```

package com.websystique.springmvc.configuration;

import javax.servlet.Filter;

import
org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherSer
vletInitializer;

public class HelloWorldInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class[] { HelloWorldConfiguration.class };
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return null;
    }
}

```

```
@Override
protected String[] getServletMappings() {
    return new String[] { "/" };
}

@Override
protected Filter[] getServletFilters() {
    Filter [] singleton = { new CORSFilter() };
    return singleton;
}

}
```

That's it. With these two additional steps, clients will be able to communicate with your REST API without worrying about Cross domain issues.

### **Download Source Code**

[Download Now!](#)

With CORS support:

[Download Now!](#)