



## CST-339 Activity 3 Guide

### Contents

Part 1: Creating Spring Bean Services using Spring Core.....	1
Part 2: Spring Bean Lifecycle and Scopes .....	3
Part 3: Creating REST Services using Spring REST Controllers.....	6
Appendix A: Information on Postman.....	9

### Part 1: Creating Spring Bean Services using Spring Core

#### Overview

##### Goal and Directions:

In this activity, you will code a number of Business Services using Spring Beans, Spring's Inversion of Control (IoC) Container, and Dependency Injection (DI).

#### Execution

Execute this assignment according to the following guidelines:

1. Make a copy of the *topic2-2* project by right-clicking on the project, selecting the Copy menu option, then right-clicking in the Workspace and selecting the Paste menu option. Name the project *topic3-1*. Rename the Spring application file from *Topic22Application.java* to *Topic31Application.java*. Fix up the *artifactId* and *name* tags in the POM file (pom.xml) so the value is set to *topic3-1*. Run the project to ensure that everything is working properly.
2. Create a new Java Interface named *OrdersBusinessInterface* by selecting the New > Interface menu options in the *com.gcu.business* package. Add a *public void test()* method to the interface.
3. Create a Spring Bean class by right-clicking on the project and selecting the New > Class menu options. Create the Spring Bean in the *com.gcu.business* package with a class name of *OrdersBusinessService* that implements the *OrdersBusinessServiceInterface* interface. Implement the *test()* method that simply prints *Hello from the OrdersBusinessService* to the console.
4. Create a Spring Bean class by right-clicking on the project and selecting the New > Class menu options. Create the Spring Bean in the *com.gcu.business* package with a class name of *AnotherOrdersBusinessService* that implements the *OrdersBusinessServiceInterface* interface. Implement the *test()* method that simply prints *Hello from the AnotherOrdersBusinessService* to the console.
5. Create a Spring Configuration class by right-clicking on the project and selecting the New > Class menu options. Create the class in the *com.gcu* package with a class name of

*SpringConfig*. The class should be marked with the *@Configuration* annotation at the class level. The annotation should be imported from the *org.springframework.context.annotation* package.

```
@Configuration
public class SpringConfig
```

6. Add a Spring Bean definition to the *SpringConfig* class by adding a public method named *getOrdersBusiness()* that returns an *OrdersBusinessServiceInterface* and within the implementation returns an instance of a new *OrdersBusinessService* class. The method should be marked with the *@Bean* annotation with a name parameter set to *ordersBusinessService*.

```
@Bean(name="ordersBusinessService")
public OrdersBusinessServiceInterface getOrdersBusiness()
{
    return new OrdersBusinessService();
}
```

7. Inject and autowire the Orders Business Service into the Login Controller by declaring a private class member variable named *service* of type *OrdersBusinessServiceInterface*. Mark the class member variable with the *@Autowired* annotation. Make a call to the *test()* method in the *doLogin()* method of the Login Controller.
8. Run the application. Open a browser and go to *localhost:8080/login/*. Submit the form to display the Orders View. Verify that the Orders Business Service printed a test message to the Console window. Take a screenshot of the Console window.
9. Update the *SpringConfig* class and the Bean definition for the *getOrdersBusiness()* method to return a new instance of the *AnotherOrdersBusinessService()* class.

```
@Bean(name="ordersBusinessService")
public OrdersBusinessServiceInterface getOrdersBusiness()
{
    return new AnotherOrdersBusinessService();
}
```

10. Run the application. Open a browser and go to *localhost:8080/login/*. Submit the form to display the Orders View. Verify that the Another Orders Business Service printed a test message to the Console window. Take a screenshot of the Console window.
11. Update the *SpringConfig* class and the Bean definition for the *getOrdersBusiness()* method to return a new instance of the *OrdersBusinessService()* class.
12. Create a Spring Bean class by right-clicking on the project and selecting the New > Class menu options. Create the Spring Bean in the *com.gcu.business* package with a class name of *SecurityBusinessService* that implements a *public boolean authenticate(String username, String password)* method that simply prints *Hello from the SecurityBusinessService* to the console and returns a boolean value of true. The class should be marked with the *@Service* annotation at the class level. The class should be imported from the *org.springframework.stereotype* package.

```

@Service
public class SecurityBusinessService
{
    public boolean authenticate(String username, String password)
    {
        System.out.println("Hello from the SecurityBusinessService");
        return true;
    }
}

```

13. Inject and autowire the Security Business Service into the Login Controller by declaring a private class member variable named *security* of type *SecurityBusinessService*. Mark the class member variable with the *@Autowired* annotation. Make a call to the service *authenticate()* method in the *doLogin()* method of the Login Controller.
14. Run the application. Open a browser and go to *localhost:8080/login/*. Submit the form to display the Orders View. Verify that the Security Business Service printed a test message to the Console window. Take a screenshot of the Console window.
15. Add a new public method to the *OrdersBusinessServiceInterface* named *getOrders()* that returns a *List* of *OrderModel*. Implement the method in the *OrdersBusinessService* and *AnotherOrdersBusinessService* classes by creating a hard-coded list. Remove the hard-code list from the Login Controller and replace that code with a call to the *getOrders()* method from the *service* class member variable.
16. Run the application. Open a browser and go to *localhost:8080/login/*. Submit the form to display the Orders page. Verify that the Orders are displayed correctly. Take a screenshot of the Orders page.

#### Deliverables:

The following needs to be submitted as this part of the activity:

- a. Screenshot of console output for *OrdersBusinessService*.
- b. Screenshot of console output for *AnotherOrdersBusinessService*.
- c. Screenshot of console output for *SecurityBusinessService*.
- d. Screenshot of the Orders page.

## Part 2: Spring Bean Lifecycle and Scopes

### Overview

#### Goal and Directions:

In this activity, you will explore the life cycle methods and scopes of a Spring Bean.

### Execution

Execute this assignment according to the following guidelines:

1. Make a copy of the *topic3-1* project by right-clicking on the project, selecting the Copy menu option, then right-clicking in the Workspace and selecting the Paste menu option.

Name the project *topic3-2*. Rename the Spring application file from *Topic31Application.java* to *Topic32Application.java*. Fix up the *artifactId* and *name* tags in the POM file (*pom.xml*) so the value is set to *topic3-2*. Run the project to ensure that everything is working properly.

2. Add a *public void init()* method and *public void destroy()* method to the *OrdersBusinessServiceInterface*.
3. Implement the *init()* and *destroy()* methods in the *OrdersBusinessService* and the *AnotherOrdersBusinessService* classes that writes a test message to the console with the names of the methods being called.
4. Update the Spring Bean *@Bean* annotation for the Orders Business Service in the *SpringConfig* class and add the *initMethod* attribute with a value of *init* and a *destroyMethod* attribute with a value of *destroy*. It should be noted that an alternative approach is to mark the *OrdersBusinessService init()* method with the *@PostConstruct* annotation and mark the *OrdersBusinessService destroy()* method with the *@PreDestroy* annotation.

```
@Bean(name="ordersBusinessService", initMethod="init", destroyMethod="destroy")
```

5. Run the application. Open a browser and go to *localhost:8080/login/*. Submit the form to display the Orders View. Stop the application. Verify that the Orders Business Service *init* and *destroy* test messages are displayed in the Console window. Take a screenshot of the Console window. In a *readme.txt* file, explain in 2-3 sentences how the code worked (when *init()* got called) and why the number of calls to *init()* were made.
6. Update the Spring Bean annotation for the Orders Business Service in the *SpringConfig* class by adding a *@Scope("prototype")* annotation with the *proxyMode* attribute set to the *getOrdersBusiness()* method. A new instance of the Orders Business Service will now be invoked every time the Spring Bean is requested (i.e., injected into the Login Controller).

```
@Bean(name="ordersBusinessService", initMethod="init", destroyMethod="destroy")
@Scope(value="prototype", proxyMode=ScopedProxyMode.TARGET_CLASS)
public OrdersBusinessServiceInterface getOrdersBusiness()
```

7. Run the application. Open a browser and go to *localhost:8080/login/*. Submit the form with valid data multiple times. Note when the Orders Business Service *init()* method gets called and how many times *init()* gets called by looking through the console output. Take a screenshot. In a *readme.txt* file, explain in 2-3 sentences how the code worked (when *init()* got called) and why the number of calls to *init()* were made.
8. Update the Spring Bean annotation for the Orders Business Service in the *SpringConfig* class by adding a *@RequestScope* annotation to the *getOrdersBusiness()* method. A new instance of the Orders Business Service will now be invoked every time the Login Form page is requested.

```
@Bean(name="ordersBusinessService", initMethod="init", destroyMethod="destroy")
@RequestScope
public OrdersBusinessServiceInterface getOrdersBusiness()
```

9. Run the application. Open a browser and go to *localhost:8080/login/*. Submit the form with valid data multiple times. Note when the Orders Business Service *init()* method gets called and how many times *init()* gets called by looking through the console output. Take a screenshot. In a *readme.txt* file, explain in 2-3 sentences how the code worked (when *init()* got called) and why the number of calls to *init()* were made.
10. Update the Spring Bean annotation for the Orders Business Service in the *SpringConfig* class by adding a *@SessionScope* annotation to the *getOrdersBusiness()* method. A new instance of the Orders Business Service will now be invoked for each unique browser session that is requesting the Login Form page.

```
@Bean(name="ordersBusinessService", initMethod="init", destroyMethod="destroy")
@SessionScope
public OrdersBusinessServiceInterface getOrdersBusiness()
```

11. Run the application. Open a browser and go to *localhost:8080/login/*. Submit the form with valid data multiple times. Keep this browser window open. Open an entirely new browser window and go to *localhost:8080/login/*. Note when the Orders Business Service *init()* method gets called and how many times *init()* gets called by looking through the console output. Take a screenshot for both browser sessions. In a *readme.txt* file, explain in 2-3 sentences how the code worked (when *init()* got called) and why the number of calls to *init()* were made.
12. Update the Spring Bean annotation for the Orders Business Service in the *SpringConfig* class by removing the *@SessionScope* annotation from the *getOrdersBusiness()* method. This will then be set to Singleton Scope, which is the default for Spring Beans.
13. Run the application. Open a browser and go to *localhost:8080/login/*. Submit the form with valid data multiple times. Note when the Orders Business Service *init()* method gets called and how many times *init()* gets called by looking through the console output. Take a screenshot. In a *readme.txt* file, explain in 2-3 sentences how the code worked (when *init()* got called) and why the number of calls to *init()* were made.

### Deliverables:

The following needs to be submitted as this part of the activity:

- a. Screenshot of console output for OrdersBusinessService and *readme.txt* for the Prototype Scope.
- b. Screenshot of console output for OrdersBusinessService and *readme.txt* for the Request Scope.
- c. Screenshot of console output for OrdersBusinessService and *readme.txt* for the Session Scope.
- d. Screenshot of console output for OrdersBusinessService and *readme.txt* for the Singleton Scope.

## Part 3: Creating REST Services using Spring REST Controllers

### Overview

#### Goal and Directions:

In this activity, you will design a REST Service using best practices and Swagger. You will then develop a REST Service using Spring REST Controllers and also test the REST Service using the Postman tool.

### Execution

Execute this assignment according to the following guidelines:

1. REST API Development:

- a. Add XML Binding support to the application by adding the following entry to list of dependencies in the POM file. Right-click on your project and select the Maven > Update Project menu option to update the project.

```
<dependency>
  <groupId>jakarta.xml.bind</groupId>
  <artifactId>jakarta.xml.bind-api</artifactId>
</dependency>
<dependency>
  <groupId>com.sun.xml.bind</groupId>
  <artifactId>jaxb-impl</artifactId>
  <version>2.3.3</version>
  <scope>runtime</scope>
</dependency>
```

- b. Create a new *OrderList* class in the *com.gcu.model* package. Add the *@XmlElement(name="orders")* annotation to the *OrderList* class at the class level. Create a private class member variable of type *List<OrderModel>* with a name of *orders*. Initialize the *orders* variable to an empty *ArrayList<OrderModel>*. Create getter and setter methods for the *orders* variable.

```
@XmlElement(name="orders")
public class OrderList
{
    private List<OrderModel> orders = new ArrayList<OrderModel>();

    @XmlElement(name="order")
    public List<OrderModel> getOrders()
    {
        return this.orders;
    }

    public void setOrders(List<OrderModel> orders)
    {
        this.orders = orders;
    }
}
```

- c. Create a new Spring REST Controller class by right-clicking on the project and selecting the New > Class menu options. Create the class in the *com.gcu.business* package with a class name of *OrdersRestService*.



- d. Mark the *OrdersRestService* class with the `@RestController` and `@RequestMapping("/service")` annotations at the class level. The annotations should be imported from the *org.springframework.web.bind.annotation* package.

```
@RestController
@RequestMapping("/service")
public class OrdersRestService
```

- e. Autowire an instance of the *OrdersBusinessServiceInterface* into a class scoped variable named *service*.
- f. Create a new public method in the *OrdersRestService* class named *getOrdersAsJson()* that returns a *List* of *OrderModel* and is marked with the `@GetMapping` annotation with the *path* attribute set to a value of */getjson* and the *produces* attribute set to a value of *MediaType.APPLICATION\_JSON\_VALUE*. The *MediaType* should be imported from the *org.springframework.http.MediaType* package. The method should call the *getOrders()* method implementation and return the value on the *service* variable.

```
@GetMapping(path="/getjson", produces={MediaType.APPLICATION_JSON_VALUE})
public List<OrderModel> getOrdersAsJson()
{
    return service.getOrders();
}
```

- g. Create a new public method in the *OrdersRestService* class named *getOrdersAsXml()* that returns a *OrderList* and is marked with the `@GetMapping` annotation with the *path* attribute set to a value of */getxml* and the *produces* attribute set to a value of *MediaType.APPLICATION\_XML\_VALUE*. The *MediaType* should be imported from the *org.springframework.http.MediaType* package. The method implementation should create an instance of an *OrderList*, call the *setOrders()* method with the return value of the *getOrders()* method on the *service* variable, and return the instance of the *OrderList*.

```
@GetMapping(path="/getxml", produces={MediaType.APPLICATION_XML_VALUE})
public OrderList getOrdersAsXml()
{
    OrderList list = new OrderList();
    list.setOrders(service.getOrders());
    return list;
}
```

- h. Run the application. Open a browser and go to *localhost:8080/service/getjson*. Take a screenshot of the formatted JSON displayed in the browser.
- i. Run the application. Open a browser and go to *localhost:8080/service/getxml*. Take a screenshot of the formatted XML displayed in the browser.
2. REST API Design:
- Read the "Documenting your REST API Template" PDF, located in Class Resources.
  - Review the example "REST API Design Templates 1 and 2," located in the topic Resources.

- c. Read the following best practices PDFs in Class Resources:
    - i. "REST API Documentation Best Practices"
    - ii. "10 Best Practices for Better RESTful API"
    - iii. "RESTful API Design 13 Best Practices"
    - iv. "RESTful API Designing Guidelines"
  - d. Read the "Write Beautiful REST Documentation with Swagger," located in Class Resources.
  - e. Complete a REST API design that matches the REST APIs just developed. For the REST API design, you can choose whether you want to document the REST API design using a document template (using the "REST API Design Template 1" or the "REST API Design Template 2," found in Class Resources) or by using the online Swagger Tool at <https://swagger.io> (Swagger is preferred).
3. REST API Testing:
- a. Read the "Postman Tutorial" PDF, located in Class Resources.
  - b. Review the tutorials and videos from Postman as needed; they are located in topic Resources.
  - c. Refer to Appendix A for how to download and install Postman.
  - d. Create a Collection of Tests in Postman to test the JSON based services from REST API that were implemented in the previous steps.
    - i. Set up a test in Postman to get the /getjson REST API. Take a screenshot of the response displayed in Postman. If needed, refer to Appendix A for how to use Postman.
    - ii. Set up a test in Postman to get the /getxml REST API. Take a screenshot of the response displayed in Postman. If needed, refer to Appendix A for how to use Postman.

#### Deliverables:

The following needs to be submitted as this part of the activity:

- a. Screenshot of the formatted JSON displayed in the browser.
- b. Screenshot of the formatted XML displayed in the browser.
- c. Screenshot of the formatted JSON displayed in Postman.
- d. Screenshot of the formatted XML displayed in Postman.
- e. REST API design document.

#### **Research Questions**

1. Research Questions: For traditional ground students, answer the following questions in a Microsoft Word document:
  - a. What is the difference between the @Component, @ Service, and @Bean annotations? When you would use one versus the other?



- b. Why does an Inversion of Control (IoC) Container force you to design and code to interface contracts?

### **Final Activity Submission**

1. In a Microsoft Word document, complete the following for the Activity Report:
  - a. Cover sheet with the name of this assignment, date, and your name.
  - b. Section with a title that contains all the screenshots and writeups for each part of the activity.
  - c. Section with a title that contains the answers to the Research Questions (traditional ground students only).
  - d. REST API design report.
2. Submit the Activity Report to the digital classroom.

## **Appendix A: Information on Postman**

Where do I get Postman?

Postman can be downloaded from the Class Resources.

What are some good videos to get started using Postman? The following videos are located in the topic Resources:

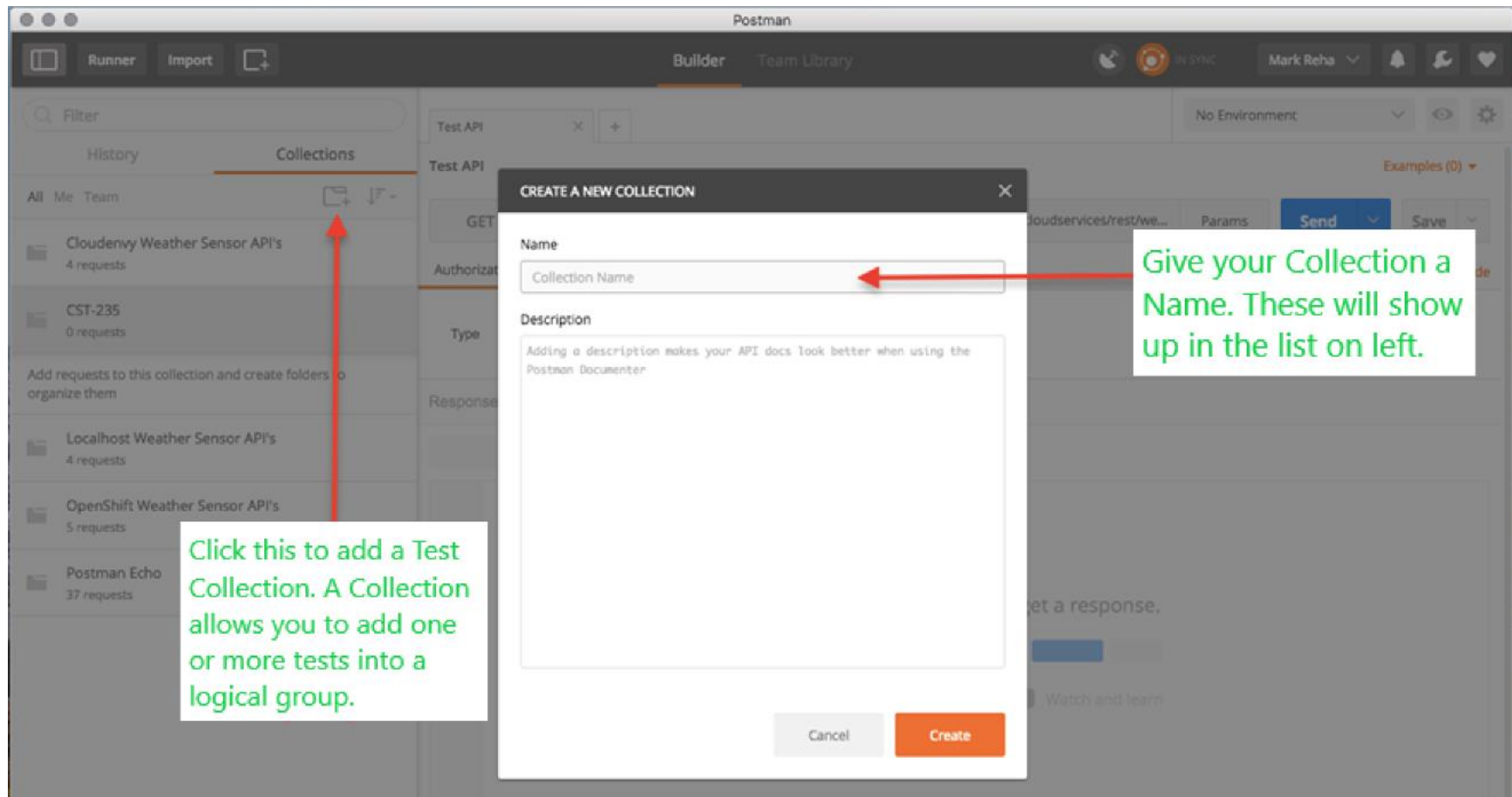
How to Use the Postman Collection Runner

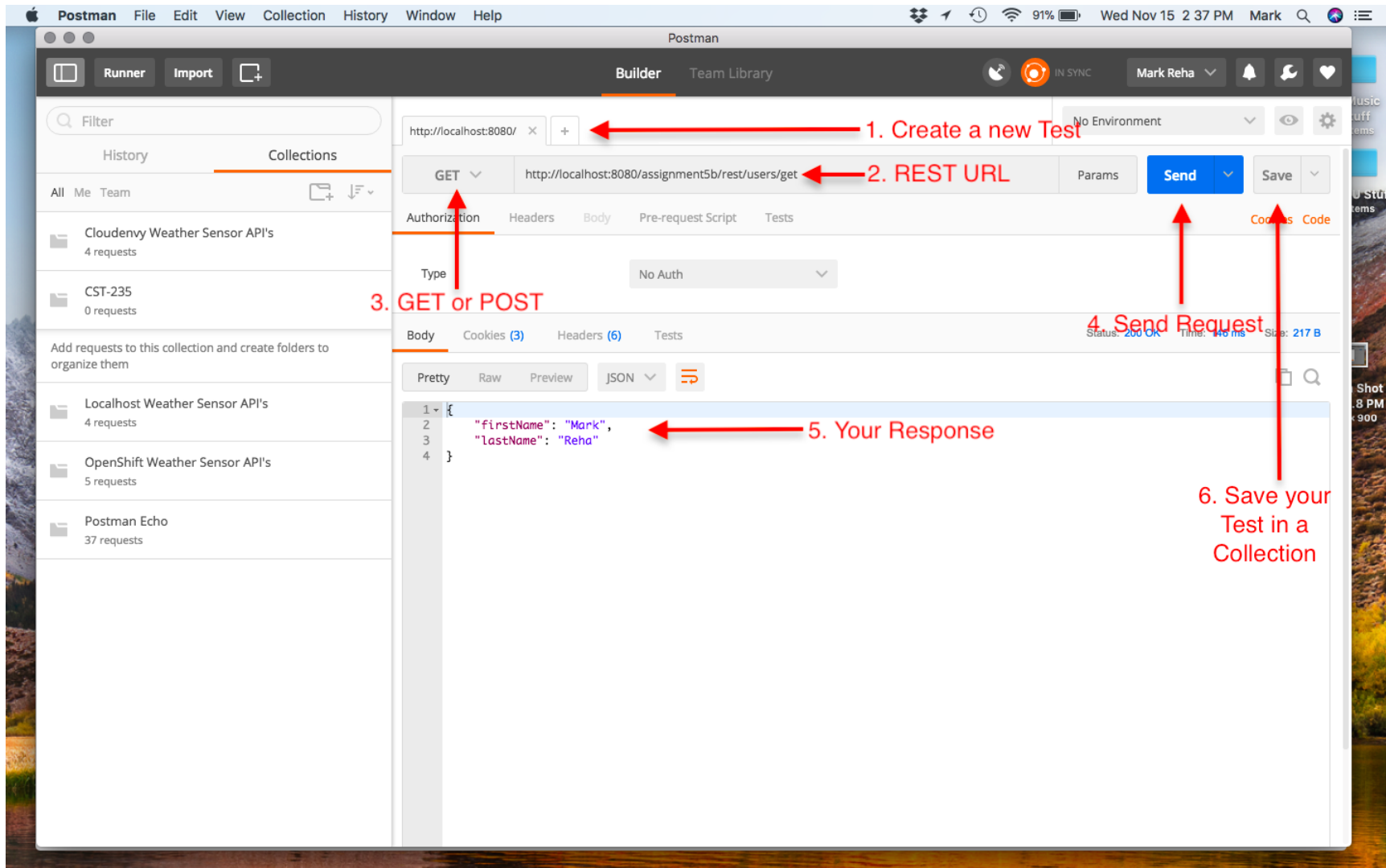
Postman Beginner Tutorial 3 – How to Create First API Request

API Testing using Postman – Advance Test Cases

How do I make a GET Request with Postman?

You can always use your browser for sending GET requests, but this is error prone and you have to write down all your URLs and test scenarios for all your APIs. Postman is a very nice desktop tool for testing your REST APIs. It allows you to set up Collections of API Tests and supports making HTTP GET and POST requests. If you sign up for an account with Postman you can even sync your Collections and Tests across development environments and team members. Postman even allows you to create unit tests and validate that you get the proper responses. This capability is available in the Tests tab of the UI. Reference the Postman home page for more information. See example below.





The screenshot shows the Postman application interface with the following annotations:

- 1. Create a new Test**: Points to the '+' button next to the URL bar.
- 2. REST URL**: Points to the URL input field containing 'http://localhost:8080/assignment5b/rest/users/get'.
- 3. GET or POST**: Points to the 'GET' dropdown menu in the request method section.
- 4. Send Request**: Points to the blue 'Send' button.
- 5. Your Response**: Points to the JSON response body showing 'firstName': 'Mark' and 'lastName': 'Reha'.
- 6. Save your Test in a Collection**: Points to the 'Save' button.

The interface also shows a left sidebar with 'Collections' and 'History' tabs, and a bottom status bar indicating 'Status: 200 OK', 'Time: 146 ms', and 'Size: 217 B'.



How do I create a Test Suite and Test Cases for my REST Services?  
Refer to the videos located in topic Resources.

How do I make a POST Request with Postman?

Postman easily supports making HTTP POST requests too. This cannot be done straight up using your browser or without writing a client page. See example below.

