Section 56: Spring REST - Overview

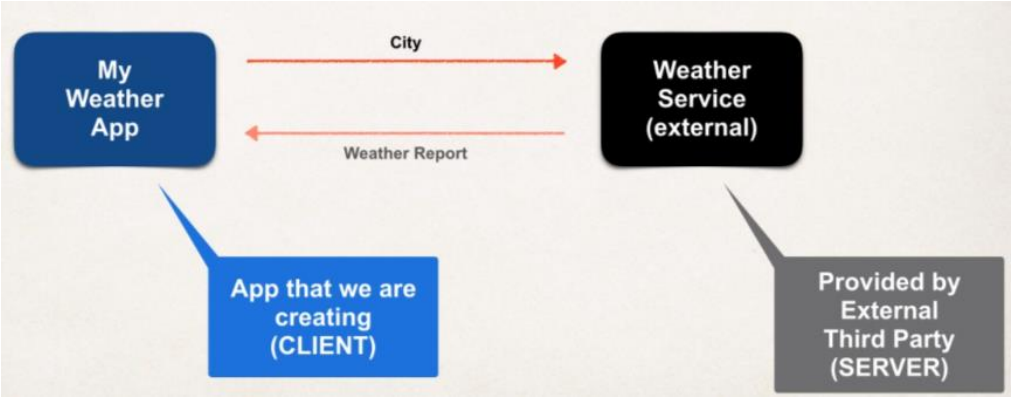
0 / 2 | 10min

➢ **<u>What are REST Webservices:</u>**

Business Problem:

Suppose we are building a client app that provides the weather report for a city

Need to get weather data from an external service.



How will we connect to the weather service?

- We can make REST API calls over HTTP.
- REST: **RE**presentational **S**tate **T**ransfer.
- Light weight approach for communication between applications.

What programming language do we use?

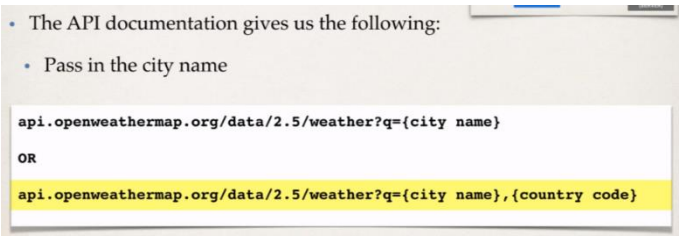
- REST is language independent.
- The client application can use **ANY** programming language.
- The server application can use **ANY** programming language.

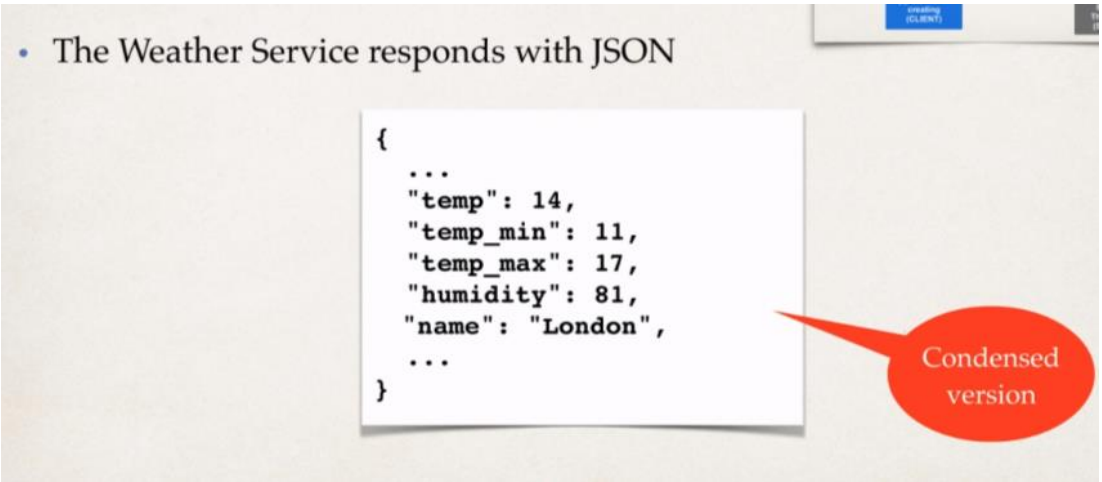
What is the data format?

- REST Applications can use any data format.
- Commonly see XML and JSON
- JSON is most popular and modern
- **J**ava**S**cript **O**bject **N**otation

Possible Solution:

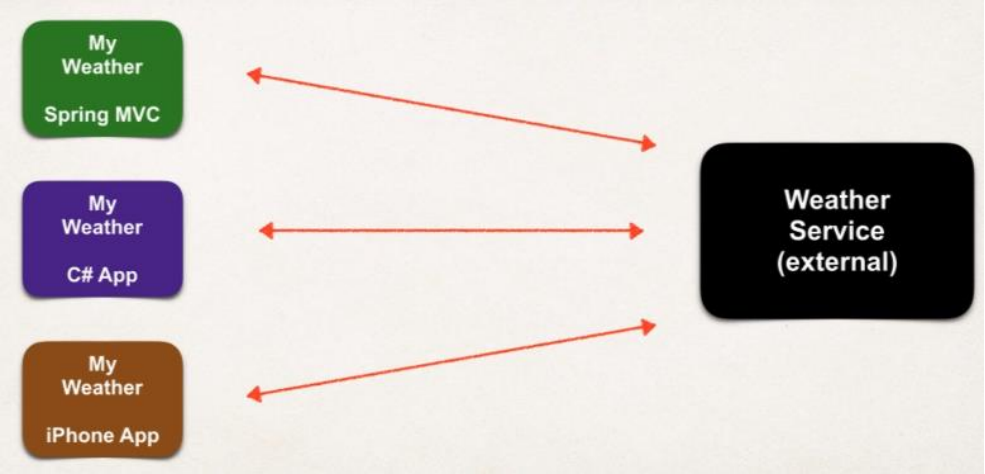
Use free Weather service provided by: openweathermap.org





Nice thing about this approach of REST API REST Webservice:

There can be multiple type of clients that can access Weather service developed by the group weathermap.



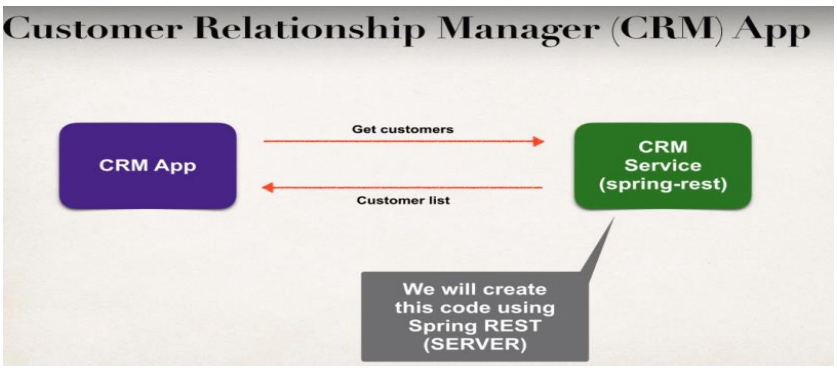
Remember:

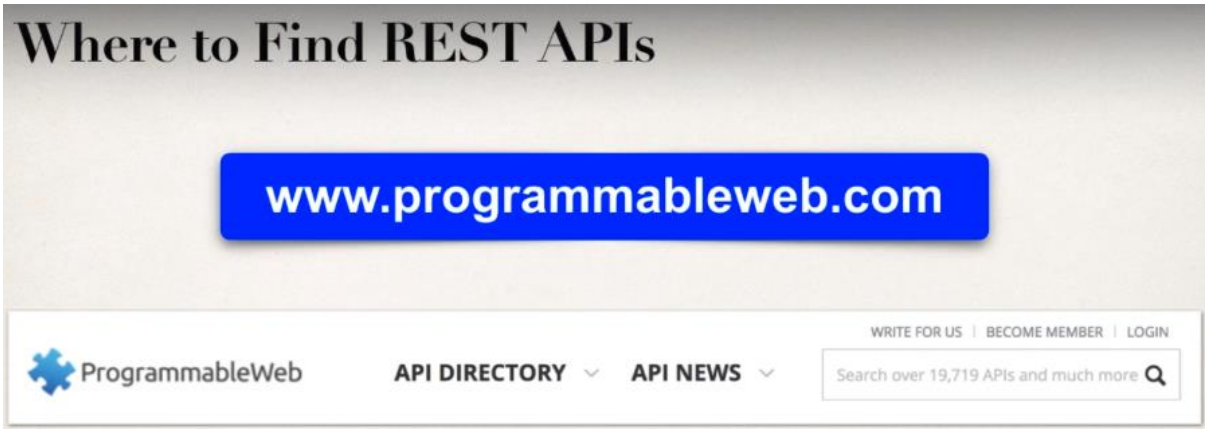
- REST calls can be made over HTTP.
- REST is language independent.

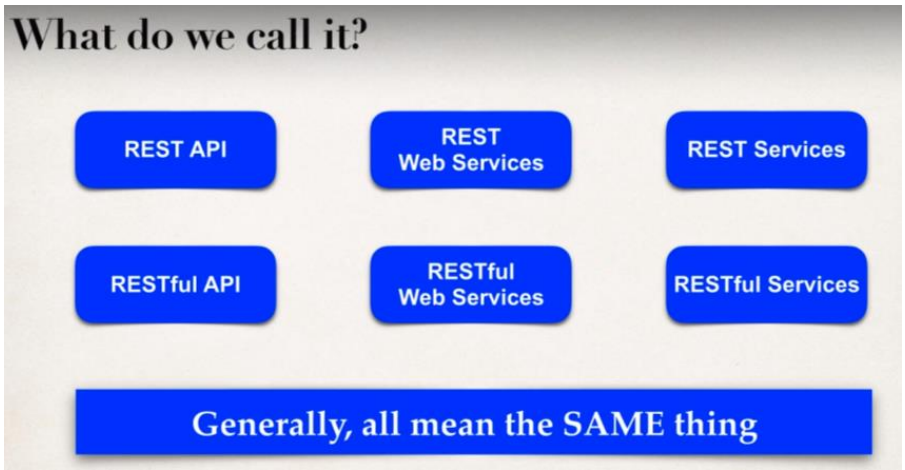
So, it gives us flexibility on actual implementation language.

<u>Another example</u>

In this course, we are going to create CRM Service than can pass customer data as JSON.







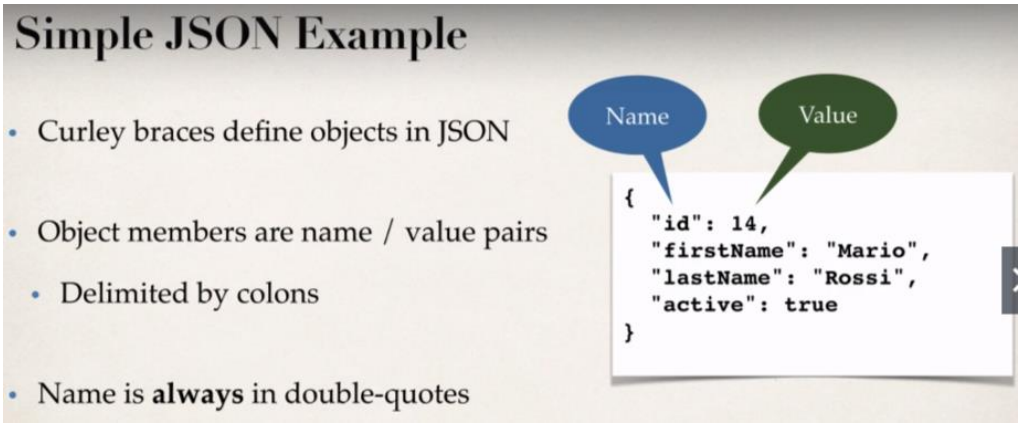
Section 57: Spring REST - JSON Data Binding

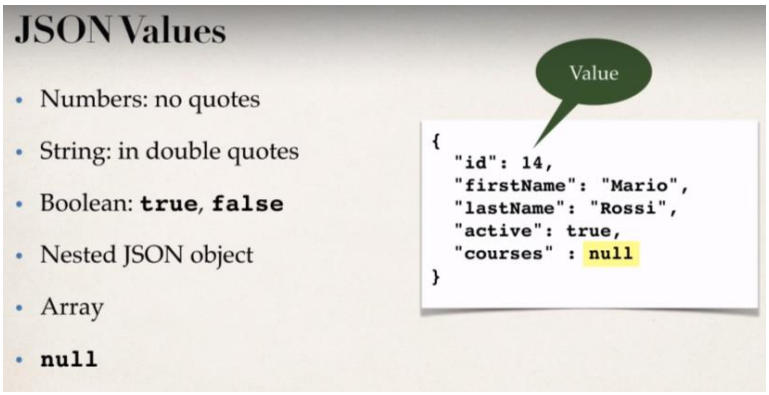
0 / 8 | 40min

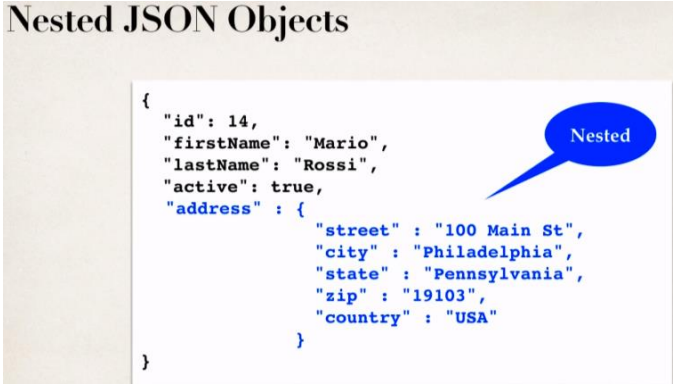
What is JSON?

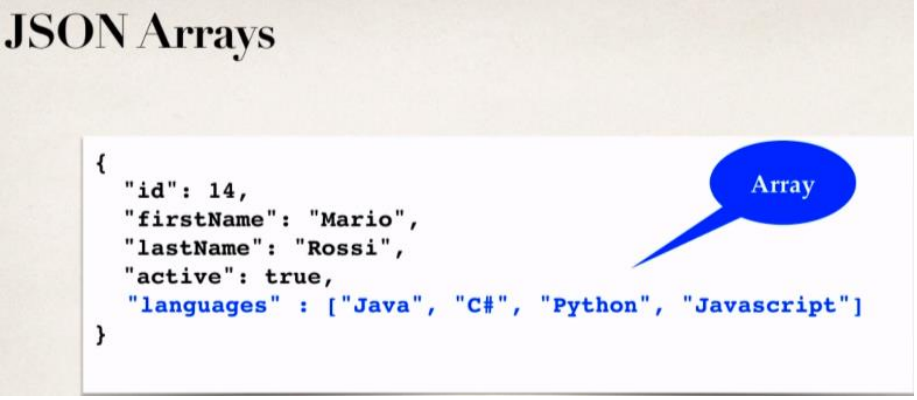
- JavaScript Object Notation
- Lightweight data format for storing and exchanging data … plain text
- Language independent … not just for **JavaScript**
- Can use with any programming language: **Java, C#, Python** etc …

JSON is just plain text data





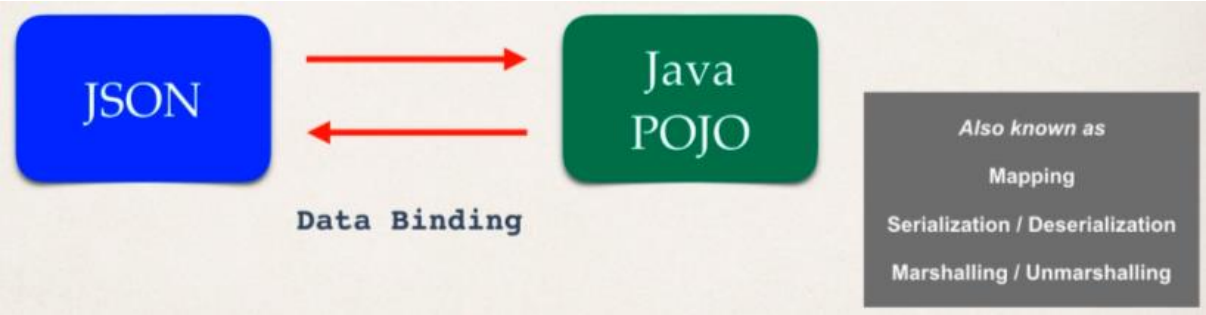




➢ <u>Java JSON Data Binding</u>

- Data binding is a process of converting JSON data to a Java POJO.

So, it will read the JSON file or data and populate the Java object with that given data or we can start with Java POJO and then send it to a JSON file or JSON Data.



JSON Data Binding with Jackson

- Spring uses the **Jackson Project** behind the scenes
- Jackson Handles data binding between JSON and Java POJO.

(Jackson is a separation project for Data Binding and they have support for doing data binding in XML and in JSON)

Details in Jackson Project:

htttps://github.com/FasterXML/Jackson-databind

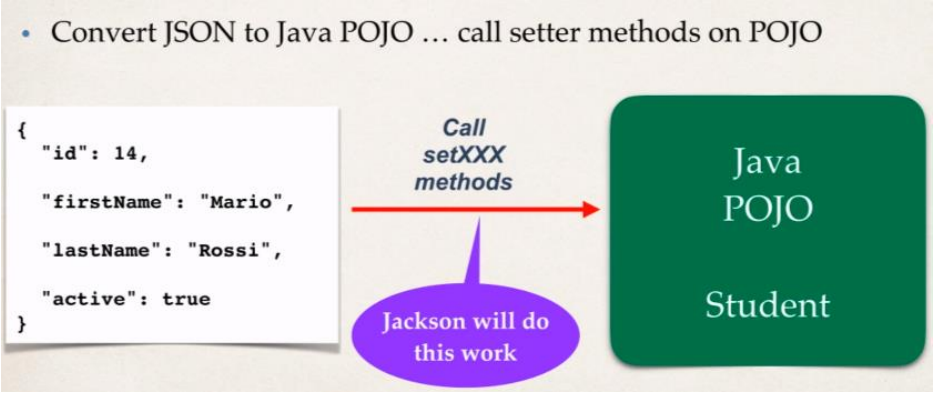
Jackson Data Binding API:

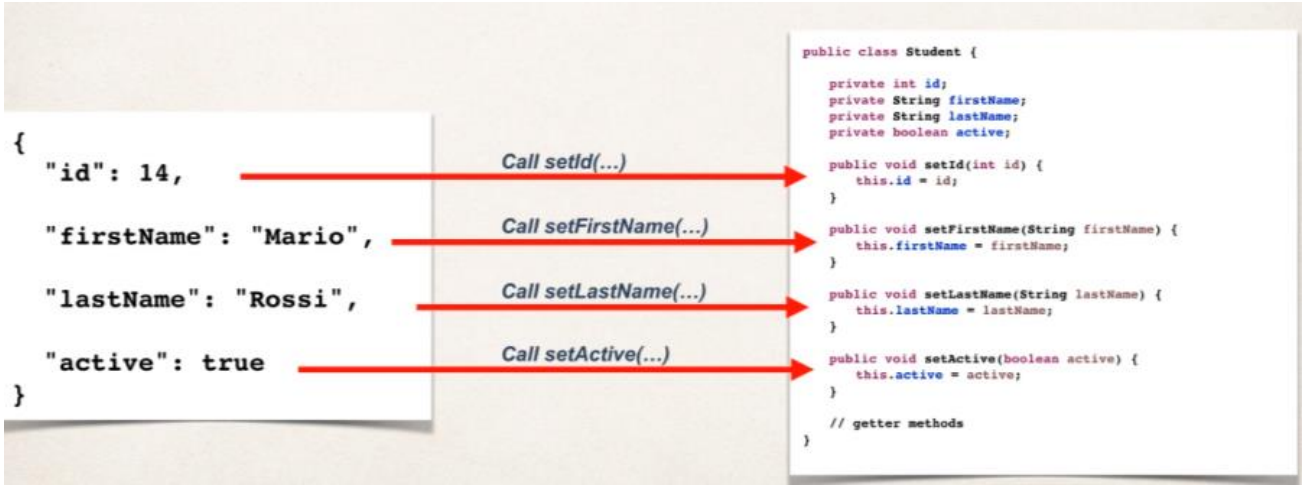
Package: **com.fasterxml.jackson.databind**

Maven Dependency:

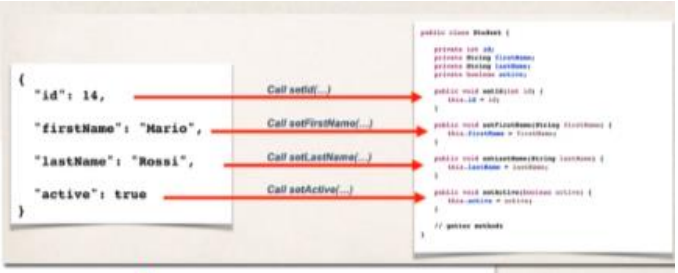
```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.9.0</version>
</dependency>
```

By Default, Jackson will call appropriate getter/setter methods when it handles the conversion i.e., when it is converting from JSON to POJO then it calls setter methods and vice versa.





JSON to Java POJO




```java
import java.io.File;

import com.fasterxml.jackson.databind.ObjectMapper;

public class Driver {

  public static void main(String[] args) throws Exception {

      // create object mapper
      ObjectMapper mapper = new ObjectMapper();

      // read JSON from file and map/convert to Java POJO
      Student myStudent = mapper.readValue(new File("data/sample.json"), Student.class);
```

Jackson does all of the work for you!

1. Read data from this file

2. Create an instance of this class and populate it

Java POJO to JSON

```java
// create object mapper
ObjectMapper mapper = new ObjectMapper();

// read JSON from file and map/convert to Java POJO
Student myStudent = mapper.readValue(new File("data/sample
...

// now write JSON to output file
mapper.enable(SerializationFeature.INDENT_OUTPUT);
```

Indent the JSON output for "pretty printing"

Java POJO to JSON

```java
// create object mapper
ObjectMapper mapper = new ObjectMapper();

// read JSON from file and map/convert to Java POJO
Student myStudent = mapper.readValue(new File("data/sample.json"), Student.class);
...

// now write JSON to output file
mapper.enable(SerializationFeature.INDENT_OUTPUT);
mapper.writeValue(new File("data/output.json"), myStudent);
```

Jackson calls the getter methods on Student POJO to create JSON output file

File: data/output.json

```
{
  "id": 14,
  "firstName": "Mario",
  "lastName": "Rossi",
  "active": true
}
```

Spring and Jackson Support:

When building Spring REST applications;

- Spring will automatically handle Jackson Integration
- JSON data being passed to REST controller is converted to POJO
- Java object being returned from REST controller is converted to JSON automatically.

JSON Jackson Demo – Set up Maven Project: Java Project Link

Here is link for starter project, import it into eclipse and give location to the folder where we have pom.xml file.

Then go to pom.xml file and add the dependency:

```
<dependency>
	<groupId>com.fasterxml.jackson.core</groupId>
	<artifactId>jackson-databind</artifactId>
	<version>2.12.3</version>
</dependency>
```

Create the Student class: Student.java

```
public class Student {

	private int id;
	private String firstName;
	private String lastName;
	private boolean active;

// default constructor and getters/setters

}
```

Create the Driver class which will contain the main() method: Driver.java

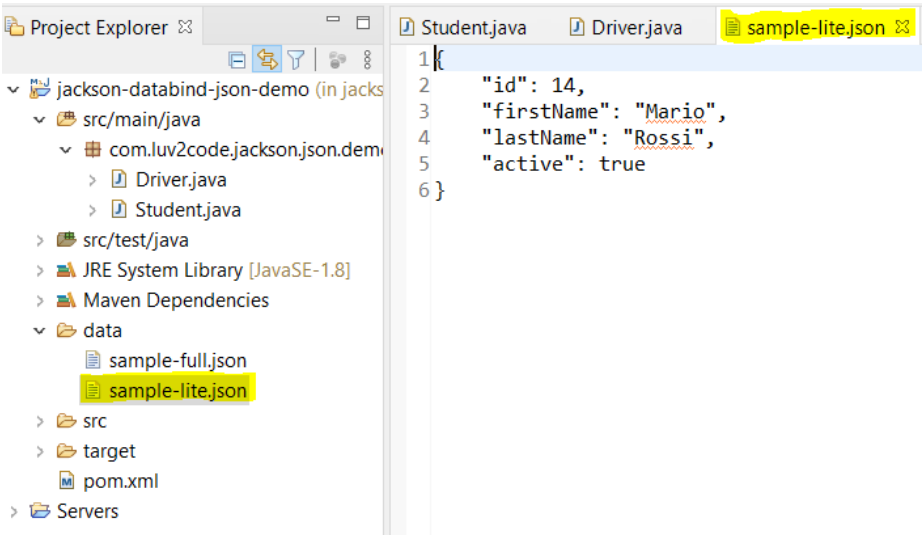
```
package com.luv2code.jackson.json.demo;
import java.io.File;
import com.fasterxml.jackson.databind.ObjectMapper;

public class Driver {

	public static void main(String[] args) {

		try {
		//create object mapper
		ObjectMapper mapper = new ObjectMapper();

		//read JOSN file and map/convert to Java POJO
		//data/sample-lite.json
		Student theStudent = mapper.readValue(
						new File("data/sample-lite.json"),Student.class);

		//print first name and last name
		System.out.println("First Name: "+theStudent.getFirstName());
		System.out.println("First Name: "+theStudent.getLastName());

		}catch (Exception e) {
```

```
                e.printStackTrace();
                }
            }
        }
```

Output:

```
Markers  Properties  Servers  Data Source Explorer  Snippets  Console  Progress
<terminated> Driver [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw.exe  (May 14, 2021, 9:31:18 PM – 9:31:18 PM
First Name: Mario
First Name: Rossi
```

And here is JSON file inside the **data** folder:



➢ **<u>JSON Jackson Demo -Nested Objects and Arrays:</u>**

File": sample-full.json

Now we need to create a new Student class where we should have extra setter methods for fields like languages and address otherwise Jackson will throw error.

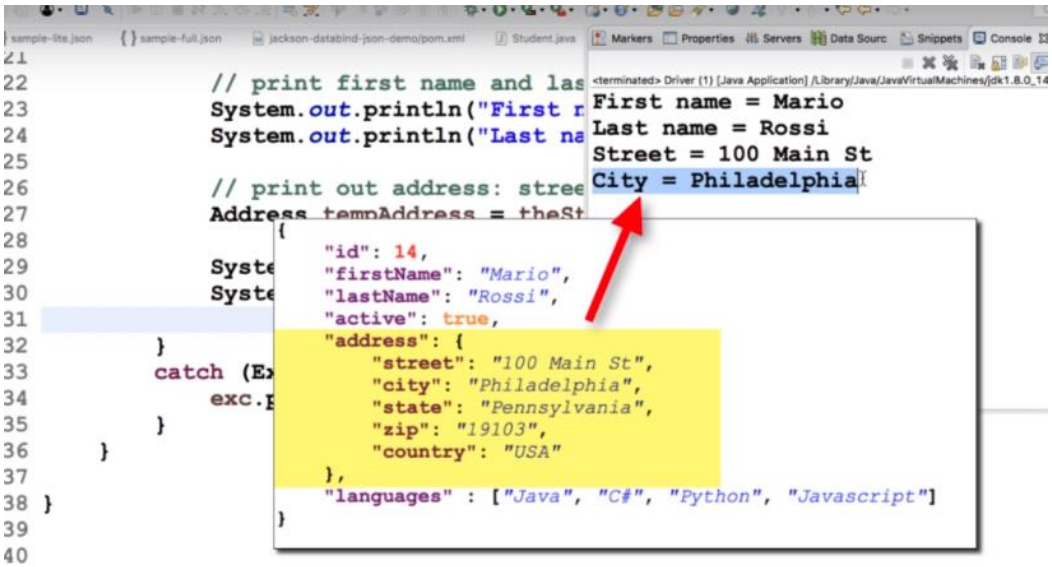
File: Address.java

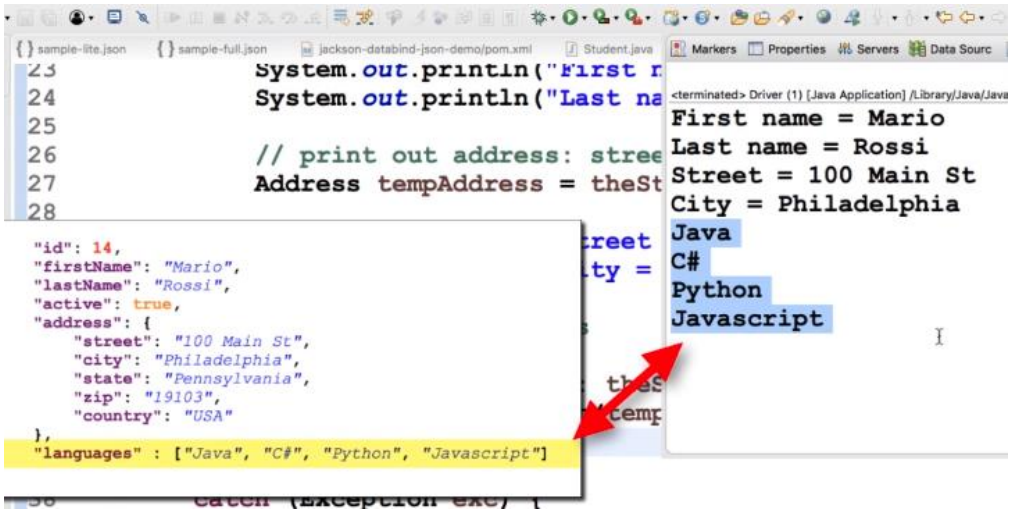
```
package com.luv2code.jackson.json.demo;

public class Address {

	private String street;
	private String city;
	private String state;
	private String zip;
	private String country;

	//getters/setters

	@Override
	public String toString() {
		return "Address [street=" + street + ", city=" + city + ", state=" + state +
	", zip=" + zip + ", country="+ country + "]";
	}

}
```

File: DriverFullJSON.java

```
//create object mapper
ObjectMapper mapper = new ObjectMapper();

//read JOSN file and map/convert to Java POJO
//data/sample-lite.json

Student theStudent =
		mapper.readValue(
				new File("data/sample-full.json"),
				Student.class);

//print first name and last name
System.out.println("First Name: "+theStudent.getFirstName());
System.out.println("First Name: "+theStudent.getLastName());

//print out address: street and city
Address tempAddress = theStudent.getAddress();
System.out.println("Street: "+tempAddress.getState());
System.out.println("City: "+tempAddress.getCity());

//print out languages
for(String tempLang : theStudent.getLanguages()) {
	System.out.println(tempLang);
```

Output:





If JSON has property that we don't care about….

Wouldn't it be great to ignore it?

Use Case

A new property is added to JSON our code is not aware of it…it can cause exception like we saw in previous section.

In this case, we will modify our code to "**ignore**" unknown properties.

Now, this new property under normal circumstances will cause our application to fail, so we just want to work around with that property in our application.

Error that we will get:

```
Markers  Properties  Servers  Data Source Explorer  Snippets  Console  Progress
<terminated> Driver (1) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_144.jdk/Contents/Home/bin/java (Apr 21, 2018, 8:47:05 AM)
edPropertyException: Unrecognized field "company" (class com.luv2code.jackson.json.
hrough reference chain: com.luv2code.jackson.json.demo.Student["company"])
.UnrecognizedPropertyException.from(UnrecognizedPropertyException.java:61)
erializationContext.handleUnknownProperty(DeserializationContext.java:822)
er.std.StdDeserializer.handleUnknownProperty(StdDeserializer.java:1152)
er.BeanDeserializerBase.handleUnknownProperty(BeanDeserializerBase.java:1582)
er.BeanDeserializerBase.handleUnknownVanilla(BeanDeserializerBase.java:1560)
er.BeanDeserializer.vanillaDeserialize(BeanDeserializer.java:294)
er.BeanDeserializer.deserialize(BeanDeserializer.java:151)
ectMapper._readMapAndClose(ObjectMapper.java:4001)
ectMapper.readValue(ObjectMapper.java:2890)
ver.main(Driver.java:19)
```

So, we will make use of a special annotation ***@JsonIgnoreProperties(ignoreUnknown=true)***

```
sample-lite.json  sample-full.json  jackson-databind-json-demo/pom.xml  Student.java  Driver.java  Address.java
1  package com.luv2code.jackson.json.demo;
2
3  import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
4
5  @JsonIgnoreProperties(ignoreUnknown=true)
6  public class Student {
7
8      private int id;
9      private String firstName;
10     private String lastName;
11     private boolean active;
12
13     private Address address;
14
15     private String[] languages;
```

Let's ignore unknown properties

```
sample-lite.json  sample-full.json  jackson-databind-json-demo/pom.xml  Student.java  Driver.java  Address.java
1  {
2      "id": 14,
3      "firstName": "Mario",
4      "lastName": "Rossi",
5      "active": true,
6      "address": {
7          "street": "100 Main St",
8          "city": "Philadelphia",
9          "state": "Pennsylvania",
10         "zip": "19103",
11         "country": "USA"
12     },
13     "languages" : ["Java", "C#", "Python", "Javascript"],
14     "company" : "Acme Inc"
15 }
```

We ignored unknown properties

So based on that annotation we are ignoring the **`company`** property, since this property is not present in our **`Student`** class.

Section 58: Spring REST - Create a Spring REST Controller

0 / 8 | 42min

➢ **REST HTTP Basics**

- Most common use of REST is over HTTP.
- Leverage HTTP methods for CRUD operations.

HTTP Method	CRUD Operation
POST	Create a new entity
GET	Read a list of entities or single entity
PUT	Update an existing entity
DELETE	Delete an existing entity

HTTP Messages:

In our example of CRM REST Services;

We have our CRM Client which will send over REST Request to a server or CRM REST Service.

Let's break down what's inside HTTP Request message and in HTTP response message.

HTTP Request Message:

The Actual request message has three main parts:

1. Request Line
2. Header Variables
3. Message Body

Request line: it has the actual HTTP command or method (like GET, POST, DELETE method)

Header variable: it has the request Metadata, so additional information about this request.

Message Body: It has the actual contents of the message or the payload.

HTTP Response Message:

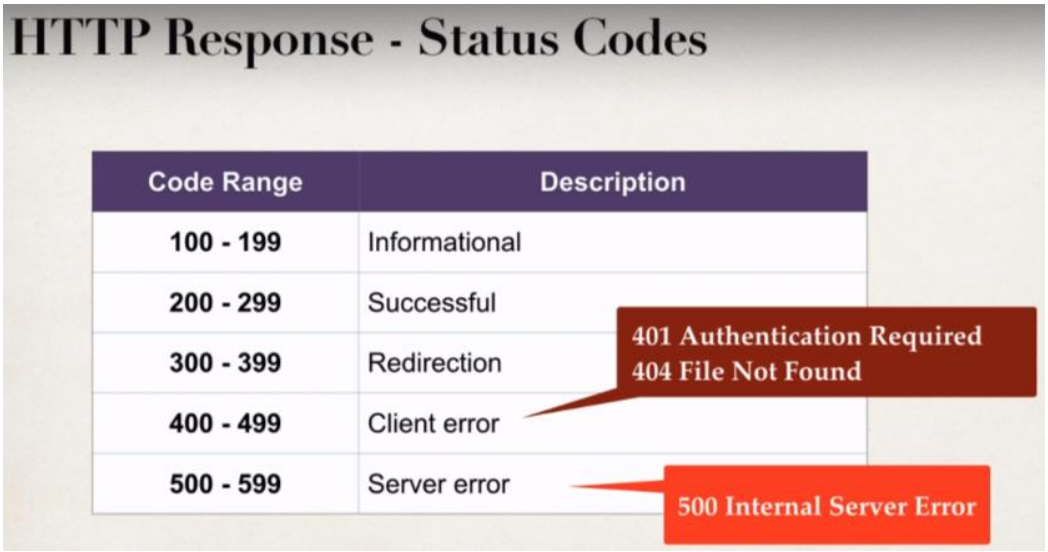
It also has three main parts:

1. Response Line
2. Header Variables
3. Message Body

Response Line: It has the actual server protocol and the status code (like 200, 404, 500).

Header Variables: it has response metadata; it has the actual information about the data (like content type of the data i.e., XML or JSON and the size and length of the data)

Message Body: It contains the contents of the message, like if we say give me the list of customers then that customer list will actually come into the Message Body either in XML or JSON.



MIME Content Type:

- This is basically the message format for the actual payload.
- MIME stands for **M**ultipurpose **I**nternet **M**ail-**E**xtension.
- Basic syntax: `type/sub-type`
- Examples: `text/html, text/plain`

So, this is the information turn back to the client and then client can render it accordingly. In the examples above,

If we return back ***`text/html`*** to a web browser, then the web browser will render that based on the HTML tags.

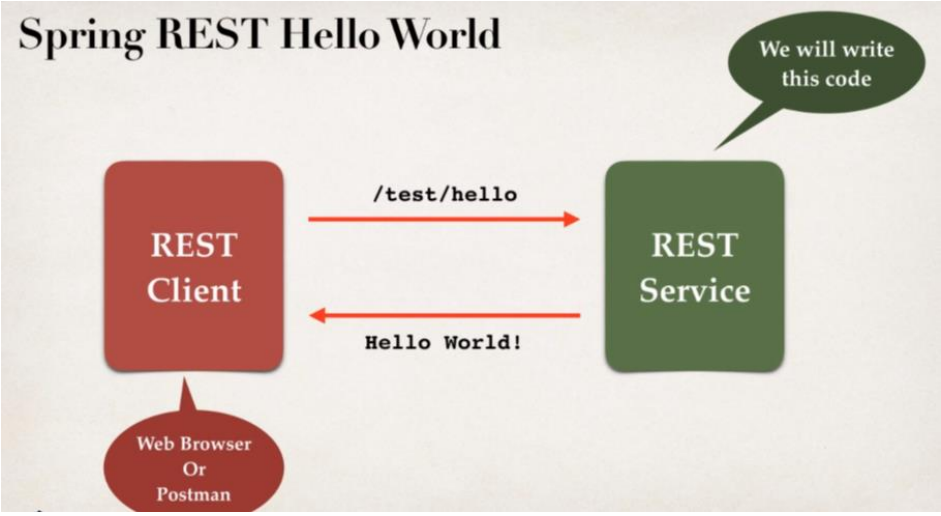
If we pass back ***`text/plain`***, the web browser will simply juts show you the plain text in the browser.

In particular, for RESTful clients, you can pass back ***`application/json`***, so we can tell the client that we are returning JSON data for you or we can have ***`application/xml`***, saying this content coming back is XML.

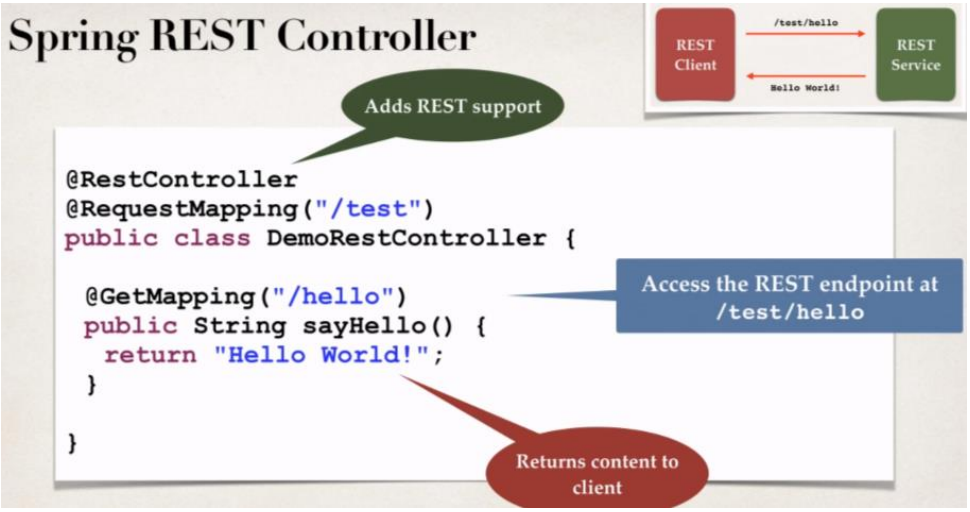
Spring REST controller:

- Spring Web MVC provides support for Spring REST.
- New annotation **@RestController**
  - Extension of @Controller that we use in regular spring MVC development.
  - But @RestController have the support for REST requests and responses.
    - So, Spring REST will also automatically convert Java POJOs to JSON
    - i.e., As long as Jackson project is on the classpath or pom.xml, spring REST will handle this conversion for us automatically.

Spring REST Hello world Example: Java Project Link



Spring REST Controller:



Spring REST Controller Development Process:

1. Add Maven Dependency for Spring MVC and Jackson project.
   - Add support for **`spring-webmvc`** as it also has support for **`REST`**.
   - Also, we need to add the Jackson support for POJO to JOSN conversion.
   - And we need to add Servlet support because when we do all Java configuration then spring dispatcher servlet initializer depends on the servlet API.

File: pom.xml

```
<!-- Add Spring MVC and REST support -->
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-webmvc</artifactId>
   <version>...</version>
</dependency>

<!-- Add Jackson for JSON converters -->
<dependency>
   <groupId>com.fasterxml.jackson.core</groupId>
   <artifactId>jackson-databind</artifactId>
   <version>...</version>
</dependency>

<!-- Add Servlet support for
    Spring's AbstractAnnotationConfigDispatcherServletInitializer -->
<dependency>
   <groupId>javax.servlet</groupId>
   <artifactId>javax.servlet-api</artifactId>
   <version>...</version>
</dependency>
```

Add

2. Add code for All Java Config: @Configuration
   - We need to add **`@Configuration, @ComponentScan(BasePackages=""), @EnableWebMvc`** annotation.

File: DemoAppConfig.java

```
@Configuration
@EnableWebMvc
@ComponentScan(basePackages="com.luv2code.springdemo")
public class DemoAppConfig {

}
```

3. Add code for All java config: Servlet Initializer.
   - Spring MVC provides support for web app initialization.
   - Makes sure your code is automatically detected.
   - Your code is used to initialize the servlet container.
   - And we will make use of the class **`AbstractAnnotationConfigDispatcherServletInitializer`**.

AbstractAnnotationConfigDispatcherServletInitializer

- Your TO DO list
  - Extend this abstract base class
  - Override required methods
  - Specify servlet mapping and location of your app config

File:MySpringMvcDispatcherServletInitializer.java

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

public class MySpringMvcDispatcherServletInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

   @Override
   protected Class<?>[] getRootConfigClasses() {
      // TODO Auto-generated method stub
      return null;
   }

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

}
```

Our config class from Step 2

File:MySpringMvcDispatcherServletInitializer.java

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

public class MySpringMvcDispatcherServletInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

   @Override
   protected Class<?>[] getRootConfigClasses() {
      // TODO Auto-generated method stub
      return null;
   }

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

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

}
```

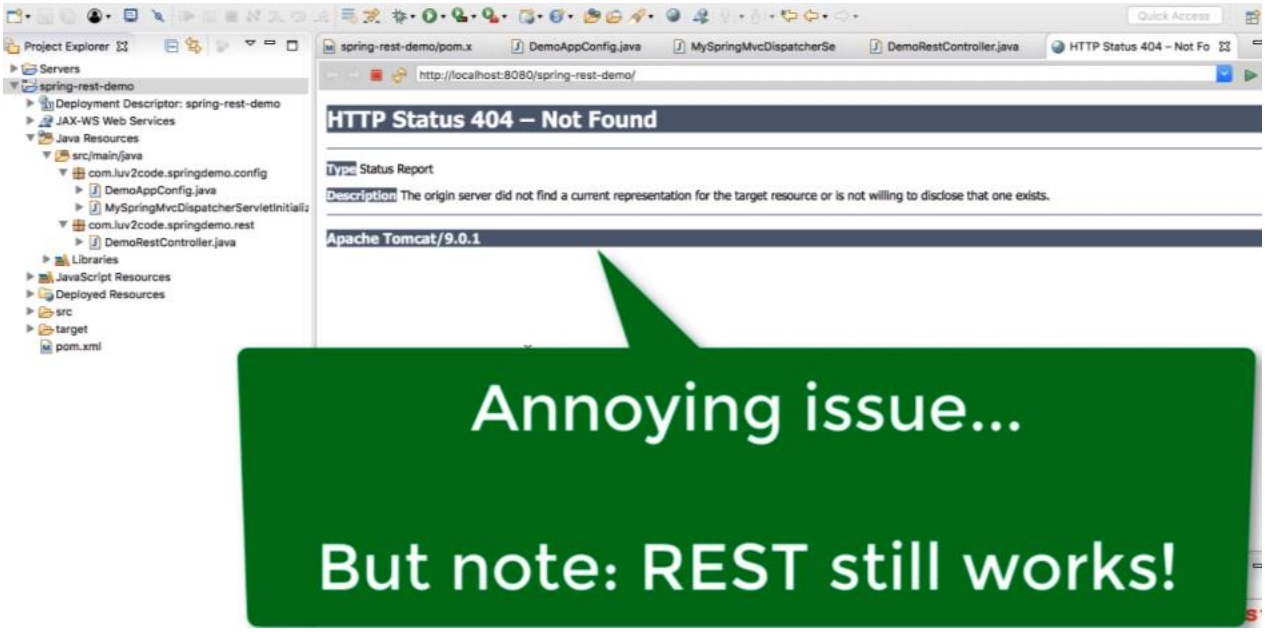
4. Create Spring REST Service using @RestController.

```java
@RestController
@RequestMapping("/test")
public class DemoRestController {

        @GetMapping("/hello")
        public String sayHellow() {
                return "Hello-world";
        }
```

On starting the REST application, we get the 404 error page but we can directly go to the link;

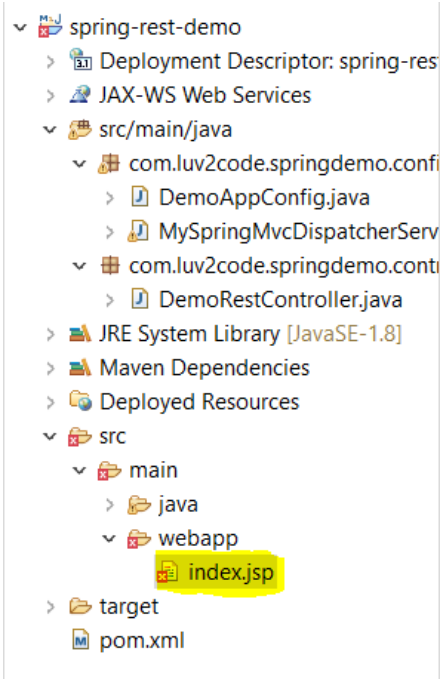
http://localhost:8080/spring-rest-demo/test/hello

But we can resolve this error:



Add an index.jsp page:

We will get to see some error, so clear those error we have to add the servlet jsp maven dependency.



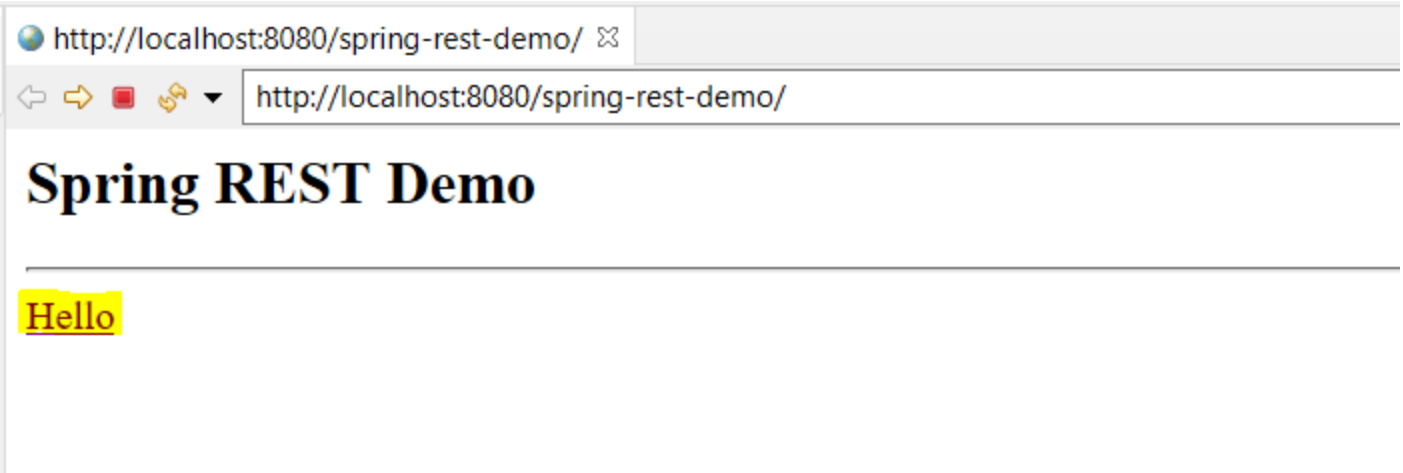

```
<!-- Add support for jsp ... to get rid of Eclipse error -->
<dependency>
        <groupId>javax.servlet.jsp</groupId>
        <artifactId>javax.servlet.jsp-api</artifactId>
        <version>2.3.1</version>
</dependency>
```

And then the error will be gone!

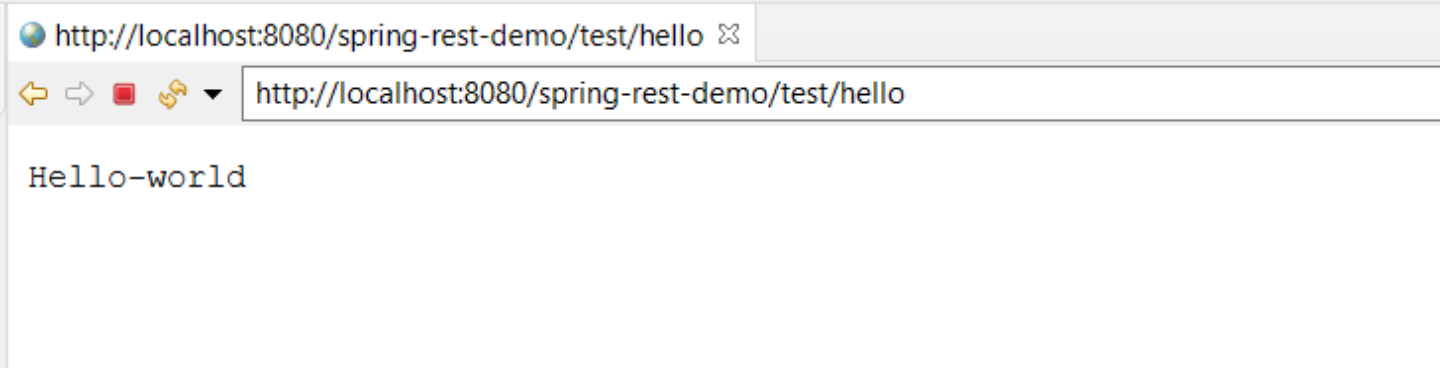
File: Index.jsp

```
<html>
<body>
<h2>Spring REST Demo </h2>
<hr>

<a href="${pageContext.request.contextPath }/test/hello">Hello</a>
</body>
</html>
```

If we don't want to give **"${pageContext.request.contextPath}"** the inside `href` provide the link without forward slash i.e., **"test/hello".**

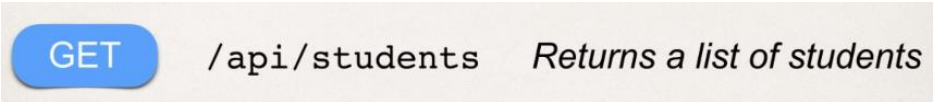


Click on Hello link:



Section 59: Spring REST - Retrieve POJOs as JSON
0 / 4 | 18min

➢ **Retrieve POJOs as JSON:**



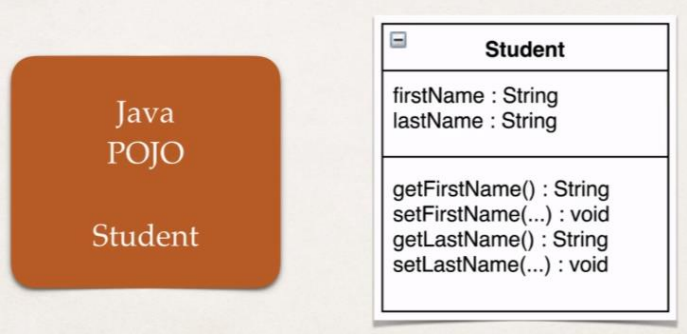
We are going to create REST Service for students.

Convert Java POJO to JSON:

- Our REST Service will return List<Student>
- Need to Convert List<Student> to JSON
- Jackson can help us out with this.

Spring and Jackson support

- Spring will automatically handle Jackson Integration
- As long as the Jackson project is on the classpath or pom.xml
- Then the JSON data being passed to the REST Controller is converted to Java POJO.
- Java POJO being returned from REST controller is converted to JSON.



Jackson will call the appropriate getter / setter methods.

Development Process:

1. Create a Java POJO class for student. **File:** Student.class
2. Create @RestController.

File: StudentRestController.java

```
@RestController
@RequestMapping("/api")
public class StudentRestController {

  // define endpoint for "/students" - return list of students

  @GetMapping("/students")
  public List<Student> getStudents() {

    List<Student> theStudents = new ArrayList<>();

    theStudents.add(new Student("Poornima", "Patel"));
    theStudents.add(new Student("Mario", "Rossi"));
    theStudents.add(new Student("Mary", "Smith"));

    return theStudents;
  }

}
```

REST Client

Jackson will convert List<Student> to JSON array

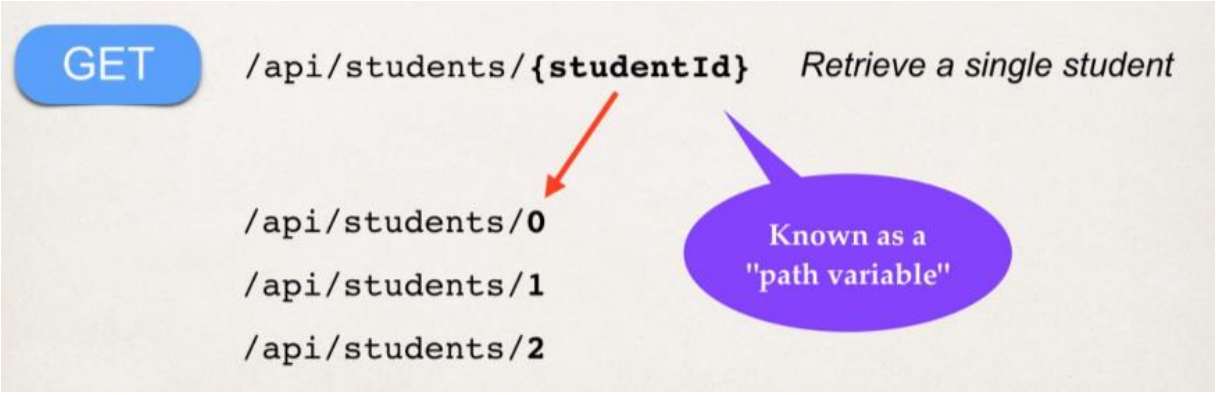
Section 60: Spring REST - Using @PathVariable for REST Endpoints

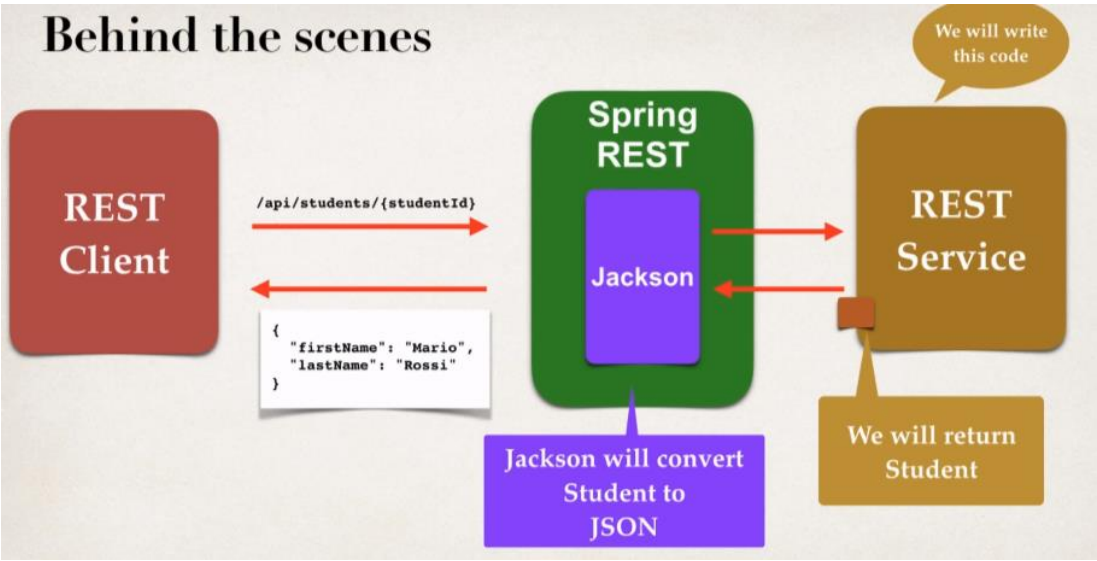
0 / 3 | 13min

Spring REST with Path Variables:

Path variables:

Retrieve a single student by id:





Development Process:

1. Add Request mapping to Spring REST service.
2. Bind path variable to method parameter using **`@PathVariable.`**

File: StudentRestController.java

```
@RestController
@RequestMapping("/api")
public class StudentRestController {

  // define endpoint for "/students/{studentId}" - return student at index

  @GetMapping("/students/{studentId}")
  public Student getStudent(@PathVariable int studentId) {
```

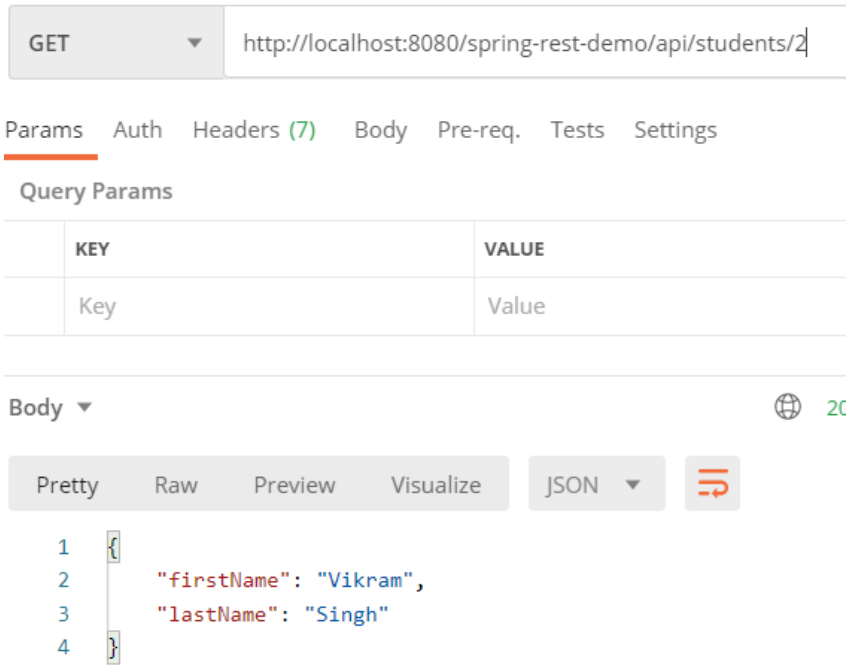
Bind the path variable (by default, must match)

File: StudentRestController.java

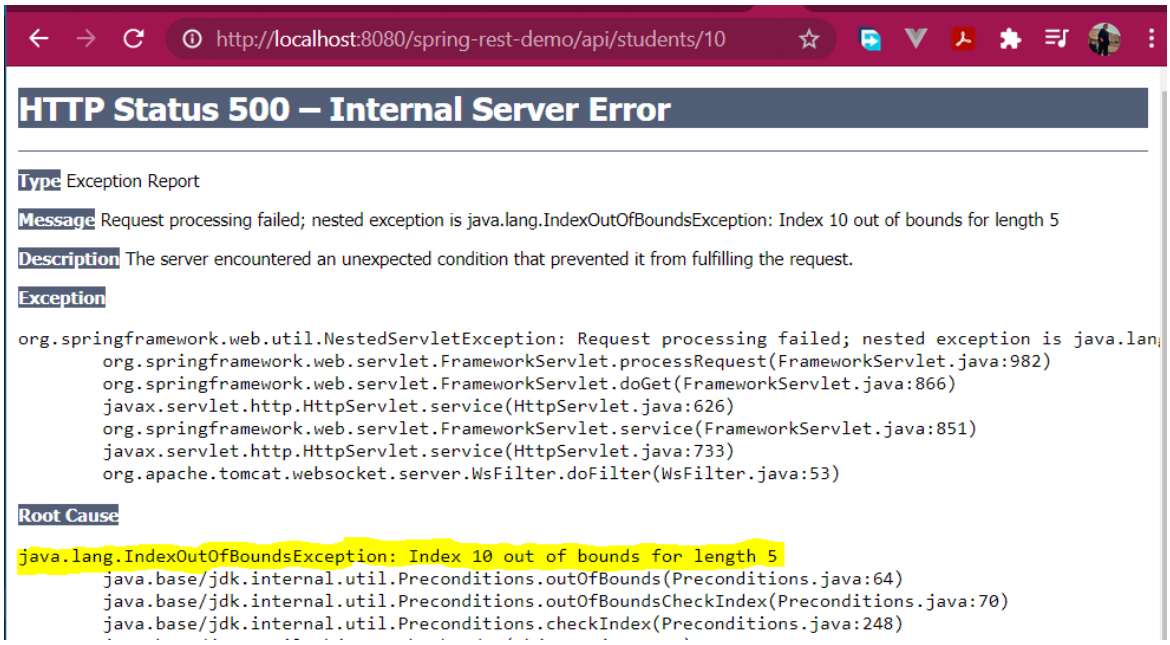
```
@RestController
@RequestMapping("/api")
public class StudentRestController {

  // define endpoint for "/students/{studentId}" - return student at index

  @GetMapping("/students/{studentId}")
  public Student getStudent(@PathVariable int studentId) {

    List<Student> theStudents = new ArrayList<>();

    // populate theStudents
    ...

    return theStudents.get(studentId);
  }

}
```

Jackson will convert Student to JSON

Using the index, we will fetch the students:



But incase we send some out of bound index to the **`pathVariable`** then it will throw an exception:

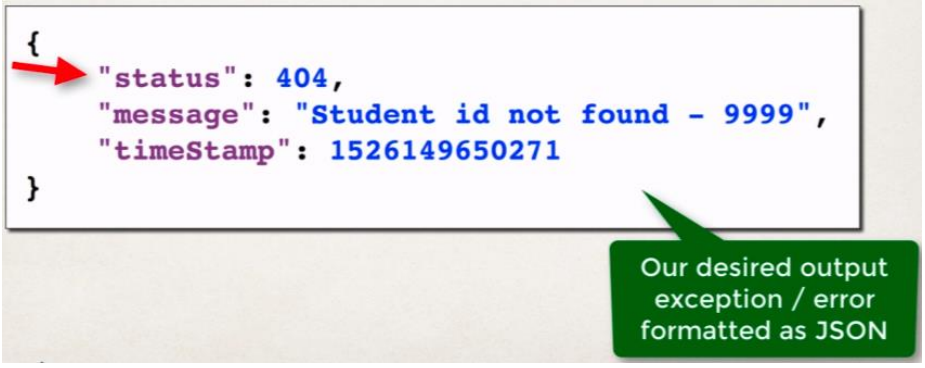


So, we will the **exceptional handling** and the **status code**.

Section 61: Spring REST - Exception Handling

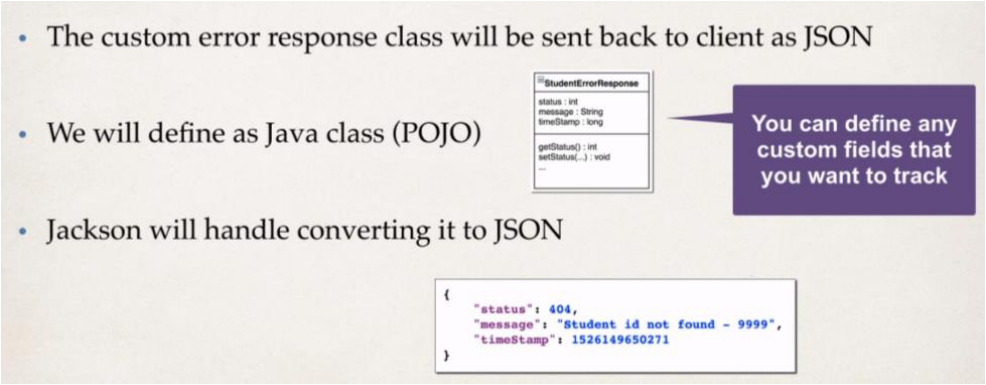
0 / 9 | 39min

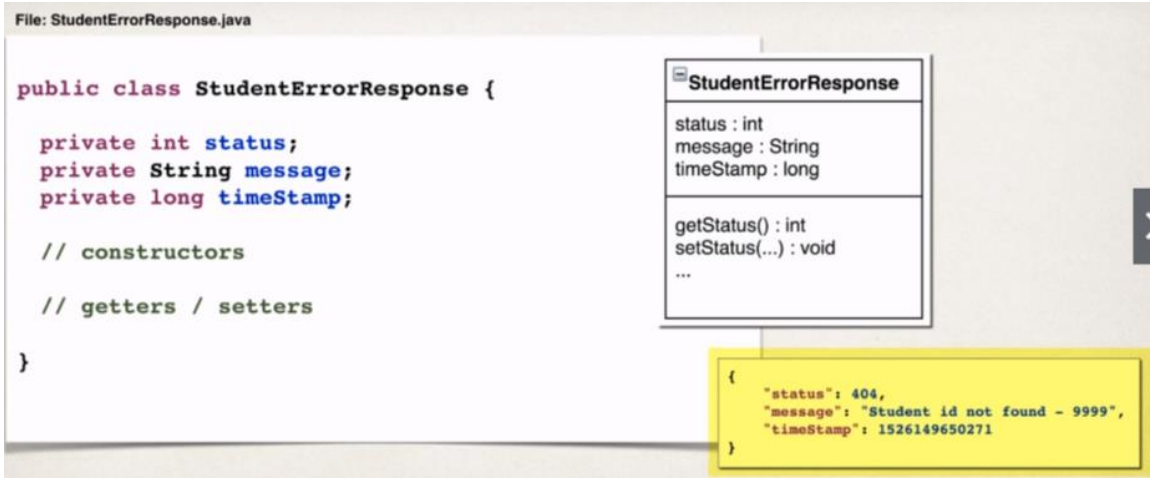
So, we will handle the exception and return the JSON.



Development Process:

1. <u>Create a custom error response class.</u>





- The Customer Student exception will used by our REST service.
- In our code, if we can't find student, then we will throw an exception.
- Need to define a custom student exception class (**`StudentNotFoundException`**)

2. <u>Create a custom exception class.</u>

File: StudentNotFoundException

```
public class StudentNotFoundException extends RuntimeException {

  public StudentNotFoundException(String message) {
    super(message);
  }

}
```

Call super class constructor

3. Update REST service to throw exception if student not found

File: StudentRestController.java

```
@RestController
@RequestMapping("/api")
public class StudentRestController {

  @GetMapping("/students/{studentId}")
  public Student getStudent(@PathVariable int studentId) {

    // check the studentId against list size

    if ( (studentId >= theStudents.size()) || (studentId < 0) ) {
      throw new StudentNotFoundException("Student id not found - " + studentId);
    }

    return theStudents.get(studentId);

  }
  ...
}
```

Throw exception

Happy path

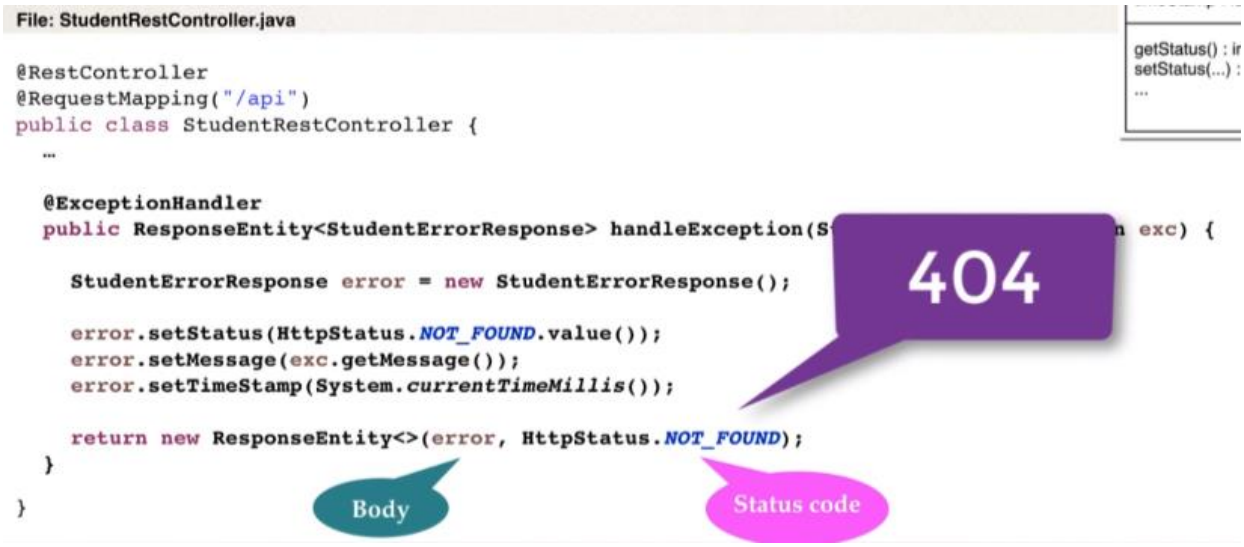
Now who is going to handle the exception and how do they give the appropriate exception back to the client.

4. Add an exception handler method using **`@ExceptionHandler.`**

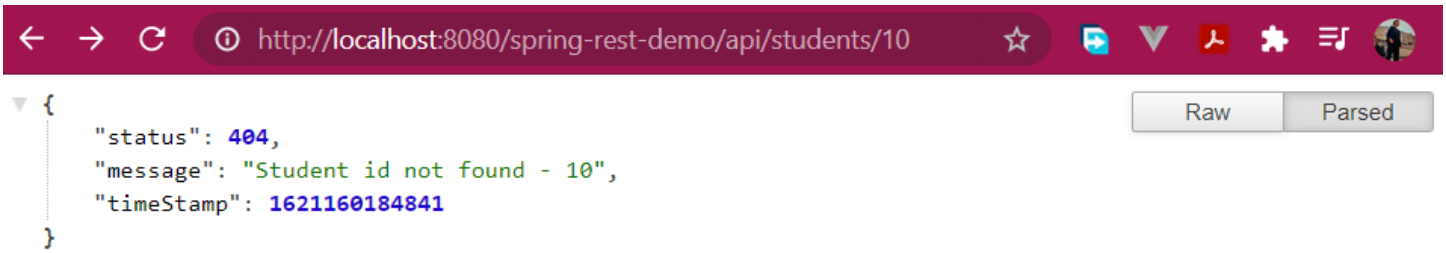
- Define the exception handler method(s) with **`@ExceptionHandler`** annotation.
- Exception handler will return a **`ResponseEntity`**.
- **`ResponseEntity`** is a wrapper for the HTTP response object.
- **`ResponseEntity`** provides fine-grained control to specify:

- **`HTTP`** `status code`, `HTTP headers` and `Response Body`.

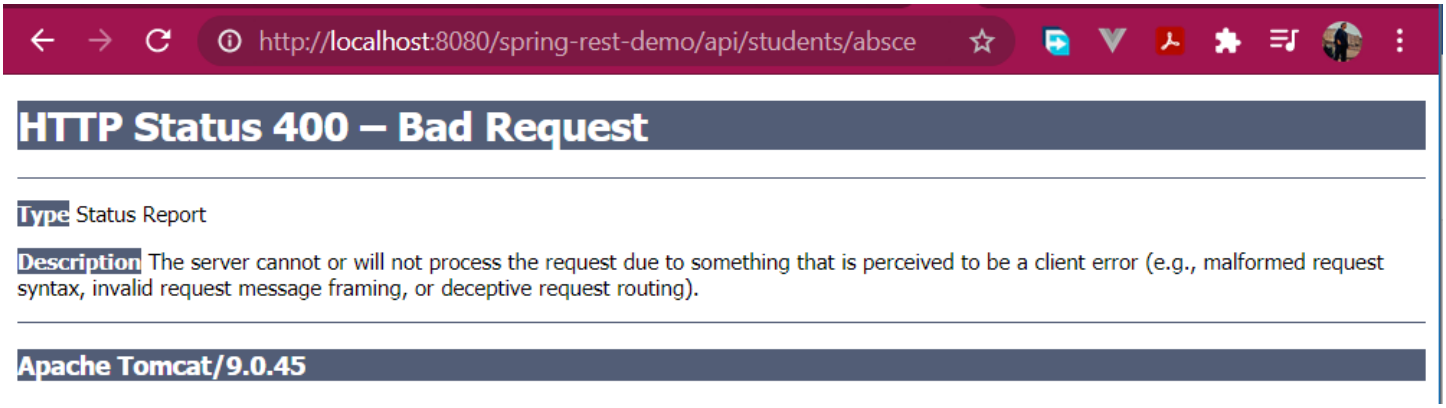
- With @ExceptionHandler we are saying hey this method that we are writing is an exception handler method.
- We also specify the type of the response body (i.e., **`StudentErrorResponse`** in our example case), so, this is the type of object that we will send back to the response body.
- Next, we are mentioning the type of exception that this Exception handler method is handling. (here in our example, it's **StudentNotFoundException**). Then any **StudentNotFound** exceptions are thorwn then the exception handler method will catch it and work on it accordingly.



Response:



But there is some edge case: what if we enter **string** instead of **int.**



Console output:

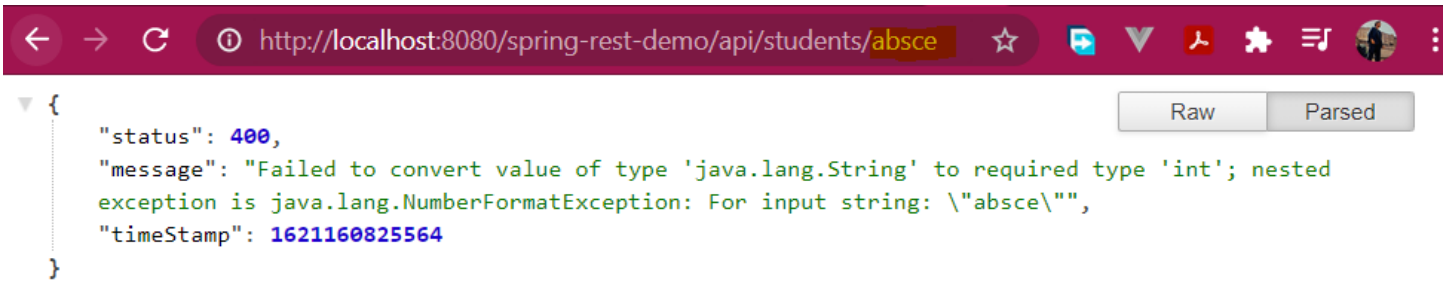
```
WARNING: Failed to bind request element:
org.springframework.web.method.annotation.MethodArgumentTypeMismatchException:
Failed to convert value of type 'java.lang.String' to required type 'int'; nested
exception is java.lang.NumberFormatException: For input string: "absce"
```

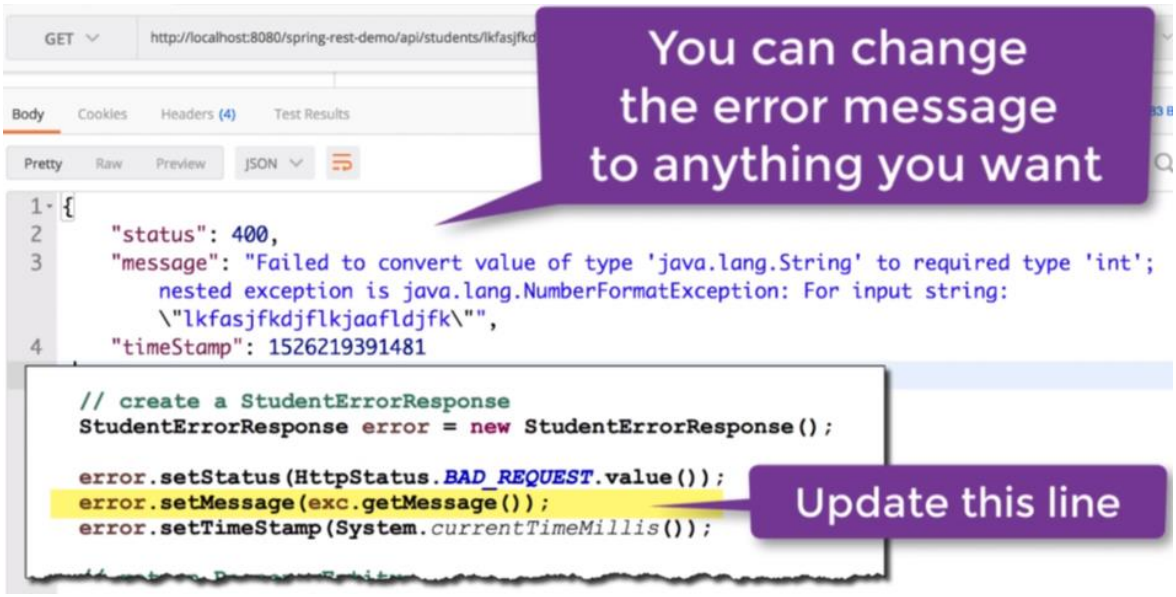
So, we need to handle such edge cases as well, by modifying our exception handler.

File: StudentRestController.java

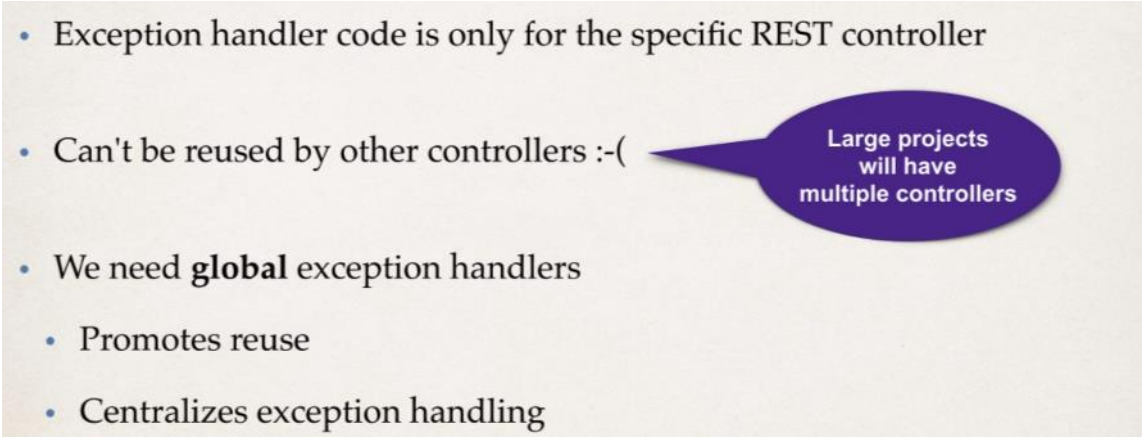
```java
//add another exception handler ...  to catch any exception
@ExceptionHandler
public ResponseEntity<StudentErrorResponse> handleException(Exception exc){

        //create a studentErrorResponse
        StudentErrorResponse error = new StudentErrorResponse();
        error.setMessage(exc.getMessage());
        error.setTimeStamp(System.currentTimeMillis());
        error.setStatus(HttpStatus.BAD_REQUEST.value());

        //return ResponseEntity
        return new ResponseEntity<StudentErrorResponse>(error,HttpStatus.BAD_REQUEST);

}
```

Response:





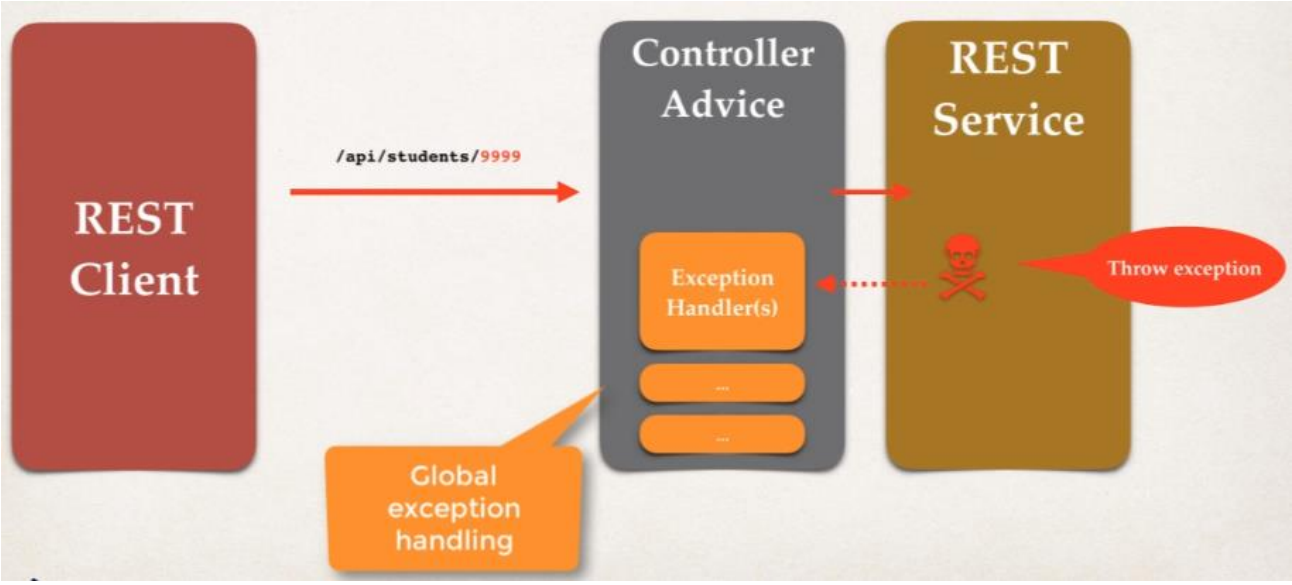
➢ Spring REST - Global Exception Handling



We will use Spring **`@ControllerAdvice`** annotation

- @ControllerAdvice is similar to an interceptor / filter.
- Pre-process requests to controllers.
- Post-process responses to handle exceptions.
- This is perfect for global exception handling.

And this is real time use of **Spring AOP**.



Now instead of exception handler being inside the REST Service, the exceptional handler is going to moved out and going to be moved out and place in the **`ControllerAdvice`**. And this will give support for Global Exception Handling.

Development Process:

- Create a new @ControllerAdvice

File: StudentRestExceptionHandler.java

New annotation

```
@ControllerAdvice
public class StudentRestExceptionHandler {

  ...

}
```

- Refactor REST service …. Remove exception handling code from the Rest Controller class.
- Add exception handling code to @ControllerAdvcie.

File: StudentRestExceptionHandler.java

Same code as before

```
@ControllerAdvice
public class StudentRestExceptionHandler {

  @ExceptionHandler
  public ResponseEntity<StudentErrorResponse> handleException(StudentNotFoundException exc) {

    StudentErrorResponse error = new StudentErrorResponse();

    error.setStatus(HttpStatus.NOT_FOUND.value());
    error.setMessage(exc.getMessage());
    error.setTimeStamp(System.currentTimeMillis());

    return new ResponseEntity<>(error, HttpStatus.NOT_FOUND);
  }

}
```

File: StudentRestExceptionHandler.java

```java
@ControllerAdvice
public class StudentRestExceptionHandler {

		//add an excption handler using @Exception Handler
		@ExceptionHandler
			public ResponseEntity<StudentErrorResponse>
	handleException(StudentNotFoundException  exc){

			//create a studentErrorResponse
			StudentErrorResponse error = new StudentErrorResponse();
			error.setMessage(exc.getMessage());
			error.setTimeStamp(System.currentTimeMillis());
			error.setStatus(HttpStatus.NOT_FOUND.value());

			//return ResponseEntity
			return new ResponseEntity<StudentErrorResponse>(error,HttpStatus.NOT_FOUND);
		}


		//add another exception handler ...  to catch any exception
		@ExceptionHandler
		public ResponseEntity<StudentErrorResponse> handleException(Exception exc){

			//create a studentErrorResponse
			StudentErrorResponse error = new StudentErrorResponse();
			error.setMessage(exc.getMessage());
			error.setTimeStamp(System.currentTimeMillis());
			error.setStatus(HttpStatus.BAD_REQUEST.value());

			//return ResponseEntity
			return new ResponseEntity<StudentErrorResponse>(error,HttpStatus.BAD_REQUEST);

		}
}
```

Response remains as it:

http://localhost:8080/spring-rest-demo/api/students/5

Raw Parsed


```
{
    "status": 404,
    "message": "Student id not found - 5",
    "timeStamp": 1621161976857
}
```


http://localhost:8080/spring-rest-demo/api/students/abs

Raw Parsed


```
{
    "status": 400,
    "message": "Failed to convert value of type 'java.lang.String' to required type 'int'; nested
    exception is java.lang.NumberFormatException: For input string: \"abs\"",
    "timeStamp": 1621162037361
}
```

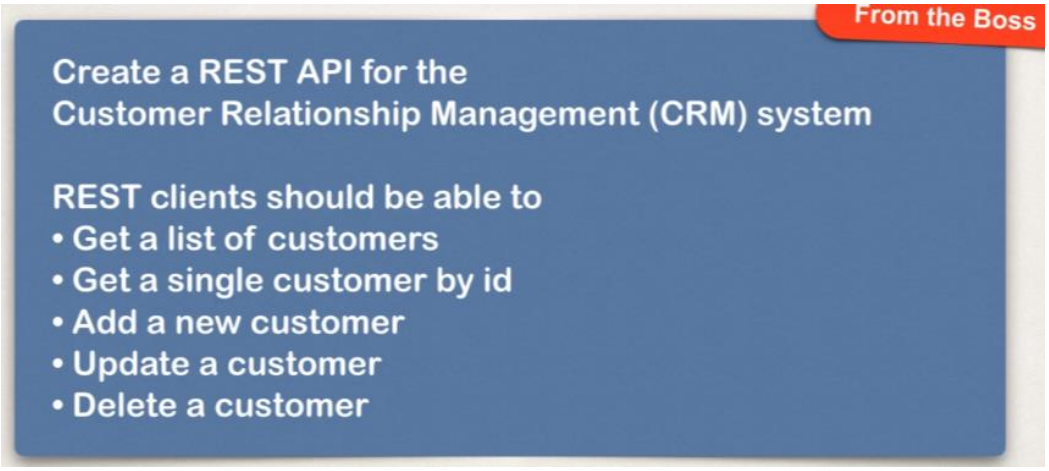
Section 62: Spring REST - API Design Best Practices

0 / 2 | 9min

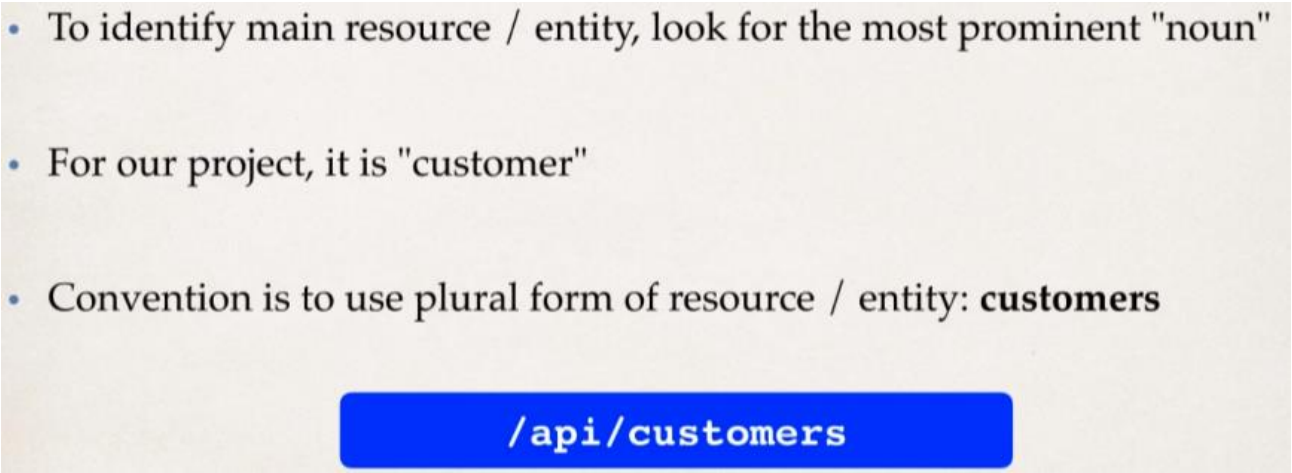
REST API Design:

API Design Process:

1. Review API requirements



2. Identify main resource / entity



3. Use HTTP methods to assign action on a given resource

HTTP Method	CRUD Action
POST	Create a new entity
GET	Read a list of entities or single entity
PUT	Update an existing entity
DELETE	Delete an existing entity

HTTP Method	Endpoint	CRUD Action
POST	/api/customers	Create a new customer
GET	/api/customers	Read a list of customers
GET	/api/customers/{customerId}	Read a single customer
PUT	/api/customers	Update an existing customer
DELETE	/api/customers/{customerId}	Delete an existing customer

For POST and PUT,
we will send customer data as JSON in request message body

Anti-Patterns

- DO NOT DO THIS … these are REST anti-patterns, bad practice

```
/api/customersList
/api/deleteCustomer
/api/addCustomer
/api/updateCustomer
```

Don't include actions in the endpoint

Instead, use HTTP methods to assign actions

<u>NOTE</u>: We should basically assign the actions based on the appropriate http methods.

- **<u>Some More examples from the real-time projects</u>**

PayPal

- PayPal Invoicing API
  - https://developer.paypal.com/docs/api/invoicing/

PayPal Developer Docs APIs Support

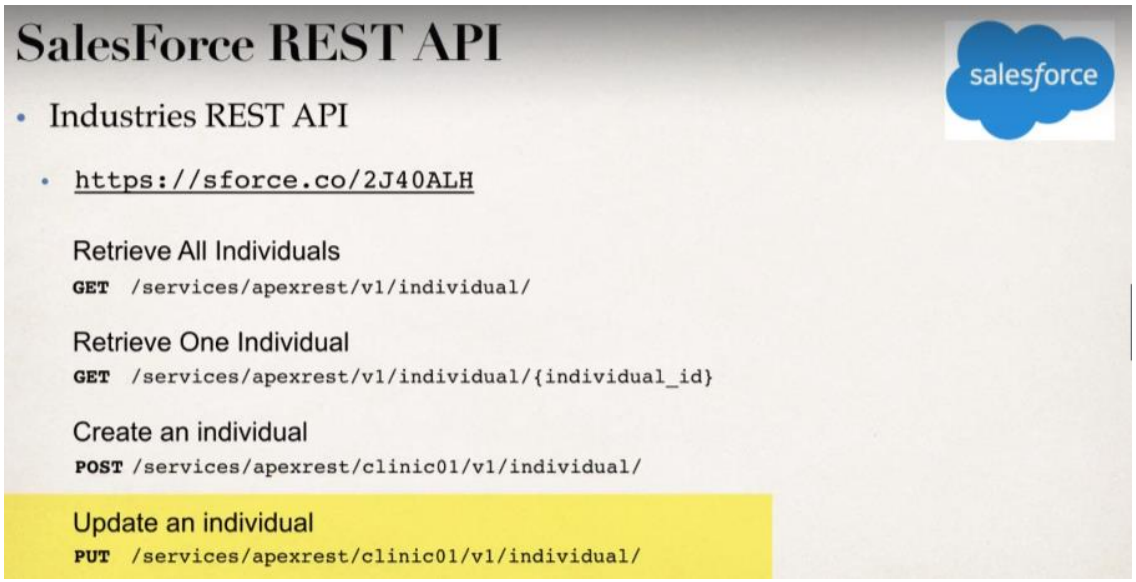
Create draft invoice
POST /v1/invoicing/invoices

List invoices
GET /v1/invoicing/invoices

Show invoice details
GET /v1/invoicing/invoices/{invoice_id}

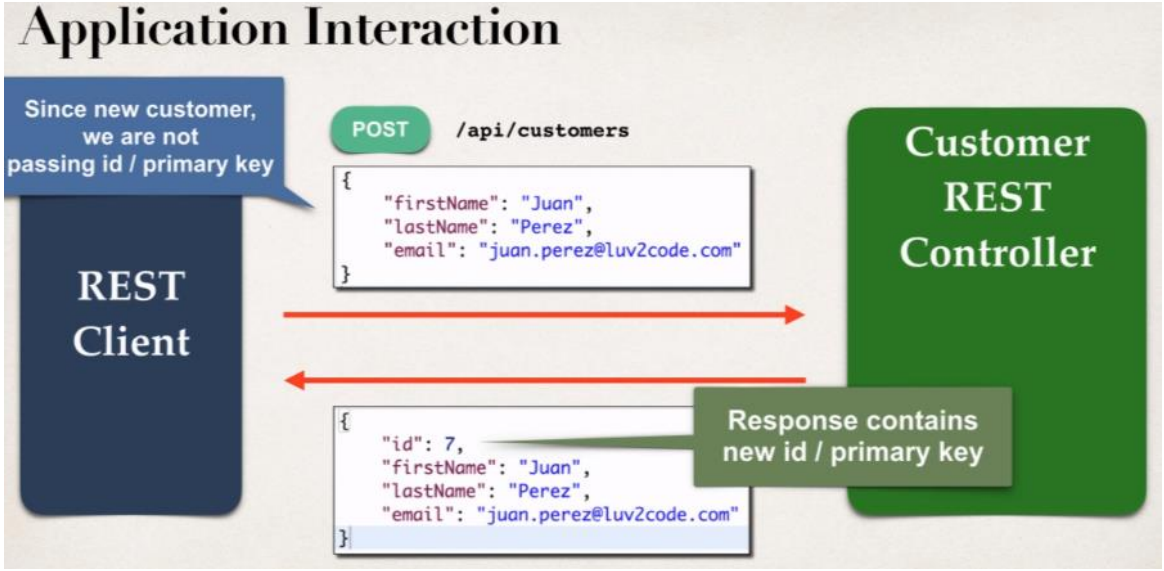
Update invoice
PUT /v1/invoicing/invoices/{invoice_id}

Delete draft invoice
DELETE /v1/invoicing/invoices/{invoice_id}



Add Customer through Spring REST:

Will cover POST method to create a customer or a new customer.



So, to access the request body, we will make use of Jackson:

- Jackson will convert the request body from JSON to POJO.
- **`@RequestBody`** annotation will bind the POJO to a method parameter.

Therefore, we use **`@RequestBody`** to access the request body as a POJO.

File: CustomerRestController.java

```
@RestController
@RequestMapping("/api")
public class CustomerRestController {
  …

  // add mapping for POST /customers  - add new customer

  @PostMapping("/customers")
  public Customer addCustomer(@RequestBody Customer theCustomer) {

    theCustomer.setId(0);

    customerService.saveCustomer(theCustomer);

    return theCustomer;
  }
}
```

What's up with customer id?

And why are we setting up the id as 0?

- In the REST controller, we explicitly set the customer id to 0
- Because our backend DAO code uses Hibernate method
  - `session.saveOrUpdate(….)`

Recall: CustomerDAOImpl

Here: "empty" means null or 0

```
36          return customers;
37      }
38
39      @Override
40      public void saveCustomer(Customer theCustomer) {
41
42          // get current hibernate session
43          Session currentSession = sessionFactory.getCurrentSession();
44
45          // save the customer ... finally LOL
46          currentSession.saveOrUpdate(theCustomer);
47
48      }
```

saveOrUpdate(...)

if (primaryKey / id) empty

then INSERT new customer

else UPDATE exiting customer

Adding customer with HTTP POST

- If REST client is sending a request to "add", using HTTP POST
- Then we ignore any id sent in the request
- We overwrite the id with 0, to effectively set it to null/empty
- Then our backend DAO code will "INSERT" new customer

Postman - Sending JSON in Request Body

- Must set HTTP request header in Postman