

## **Section 1:**

Course preview , where are the resources , Github , how to utilize the course etc.

## **Section 2:**

Theory of what a rest service is.

## **Section 3:**

**Lecture 10,11,12:** How to setup spring boot project

**Lecture 13 ?** How REST is mapped , how URI is mapped

**Lecture 14,15:** Creating Hello world, add bean to it.

**Lecture 16:** Theory of springs

What is dispatcher servlet? ? Servlet that handles http requests ,it uses FRONT controller design pattern.It knows all the mapping present .

Who is configuring dispatcher servlet ? ? SpringBootApplication

What does dispatcher servlet do ? ? handles all the requests as per the mapping

How does the HelloWorldBean gets converted to Object ? - Jackson2Mapper bean converter.

Who is configuring the Error mapping ? ? SpringBootApplication

**Lecture 17:** Path Variable example

**Lecture 18:** Create User Bean/Service ,@component

**Lecture 19:** Implementing GET method for User Resource ,@RestController

**Lecture 20:** Implementing POST method for User Resource

@RequestBody ? maps the values from an http request body to the object

Invoke POST request? POSTMAN( or any REST Client)

**Lecture 21:** \*Enhancing POST method for to return correct status /return type

\*http best Practices ? return correct status for correct operation/infformation.

```
URI location=ServletUriComponentBuilder.fromCurrentRequest()
    .path("/{id}")
    .buildAndExpand(savedUser.getId())
    .toUri();
return ResponseEntity.created(location).build();
```

**Lecture 22:** Exception handling.

\*To send a more fitting response use: @ResponseStatus(HttpStatus.NOT\_FOUND)

\* Best practices is :to return 404 if resource not found rather than 500.

i.e to return suitable status rather than any error or non 200 status.

**Lecture 23:** Exception Handling: Generic style.

? define the response structure(generally a class containing exception timestamp,message,category,severity,etc)

? ResponseEntityExceptionHandler:An abstract class that provides basic exception handling methods. Extend this class, override methods as shown below.

@ControllerAdvice ? This class is now treated as exception handler for all the controllers defined in this app.

@RestController ? Tells Springs that this class is a Controller

extends ResponseEntityExceptionHandler? to inherit the default methods & override as needed.

@ExceptionHandler(XXX.class) ? fire this annotated method when exception of the class xxx.class is thrown.

ExceptionHandler ? Org based exception structure.Throw this when exception occurs.

@ExceptionHandler(UserNotFoundException.class) ? Custom exception class that is thrown when a user is not found from the UserService.

**Lecture 24:** Implement Exception Handling for POST method.

**Lecture 25:** Implement DELETE method.

**Lecture 26:** Validation Framework .

The response is 400 ? but it doesnt tells what went wrong. Thus we need to provide a customized message.i.e override the handleMethodArgumentNotValid() from ResponseEntityExceptionHandler.class in our CustomizedResponseEntityExceptionHandler.java

Execute the request & see the message.

There is too much of details , so we can customize the message by :

? Providing message at user entity.

? remove `ex.getBindingResult().toString()` with Validation Failed

Execute the REST request again, see the message outputs & the @nnnotated message.:

Various type of validation present are listed under: `javax.constraints` package of `validation-api-1.1.0.Final.jar` & is implemented in `hibernate-validator-5.4.jar`. They are defined as spring web started web dependency, thus we get these jars when we import the project.

### **Lecture 27: HATEOAS**

Hypermedia As The Engine Of Application State (HATEOAS) is a component of the REST application architecture that distinguishes it from other network application architectures. It mandates a service should provide information on how to access the resource.

Ref: <https://spring.io/understanding/HATEOAS>

Response = Response + Link to other useful resource.

**Lecture 28:** Advanced REST using Spring ( Internationalization, Content Negotiation, Documentation, Springs Actuator,

**Lecture 29,30:** INTERNATIONALIZATION (i18n)

how to achieve this :

- Configure LocaleResolver

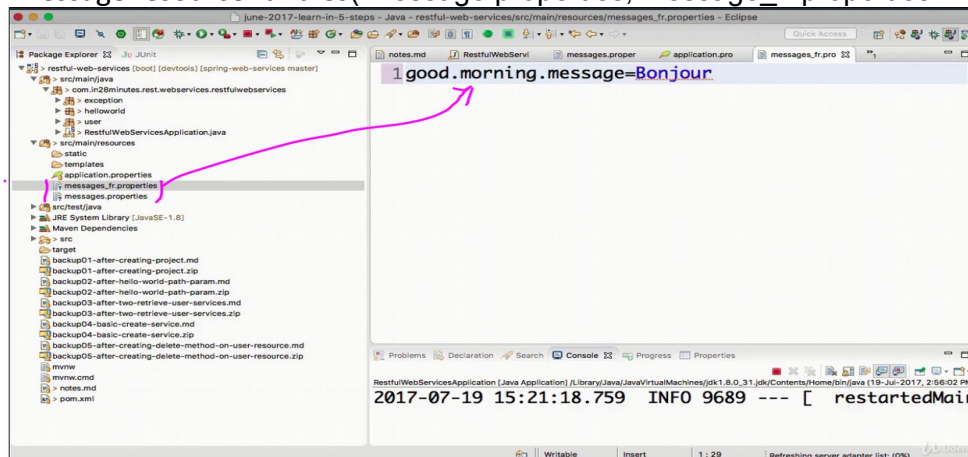
- set default Resolver

- Set ResourceBundleMessageSource

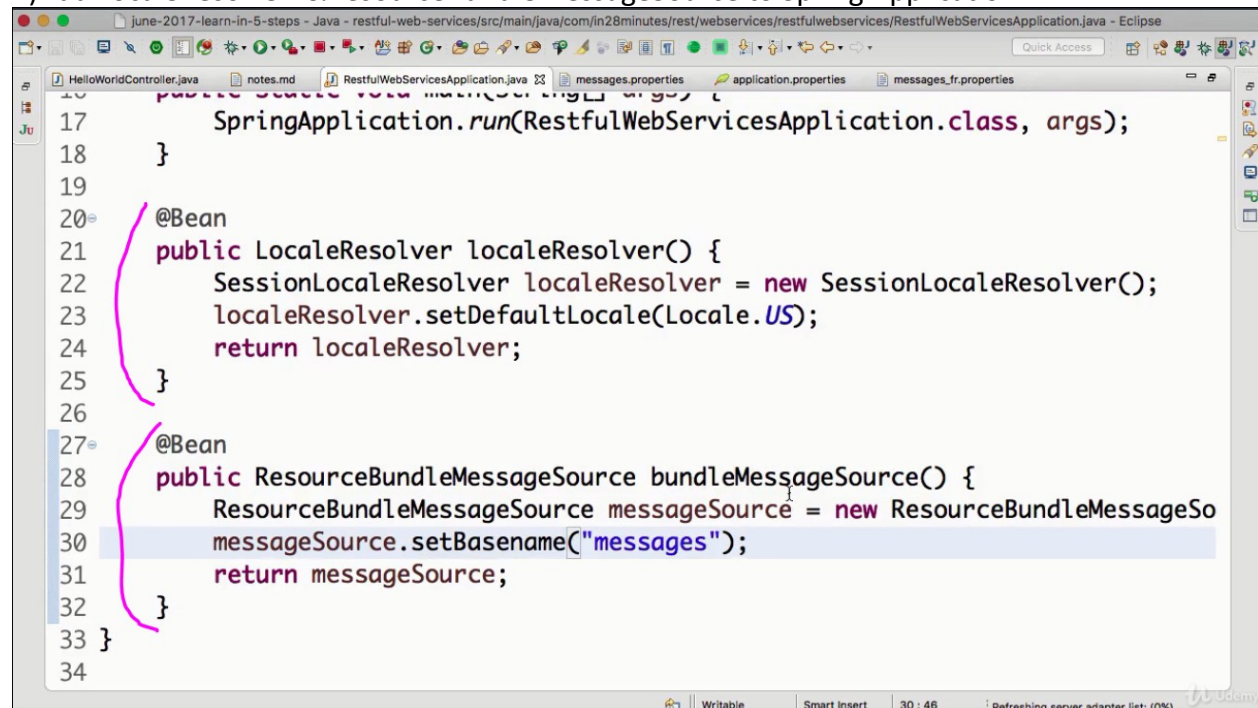
- Use Autowiring to get the MessageSource

1) Create message resources

Add MessageResourceBundles( message.properties, message\_fr.properties.



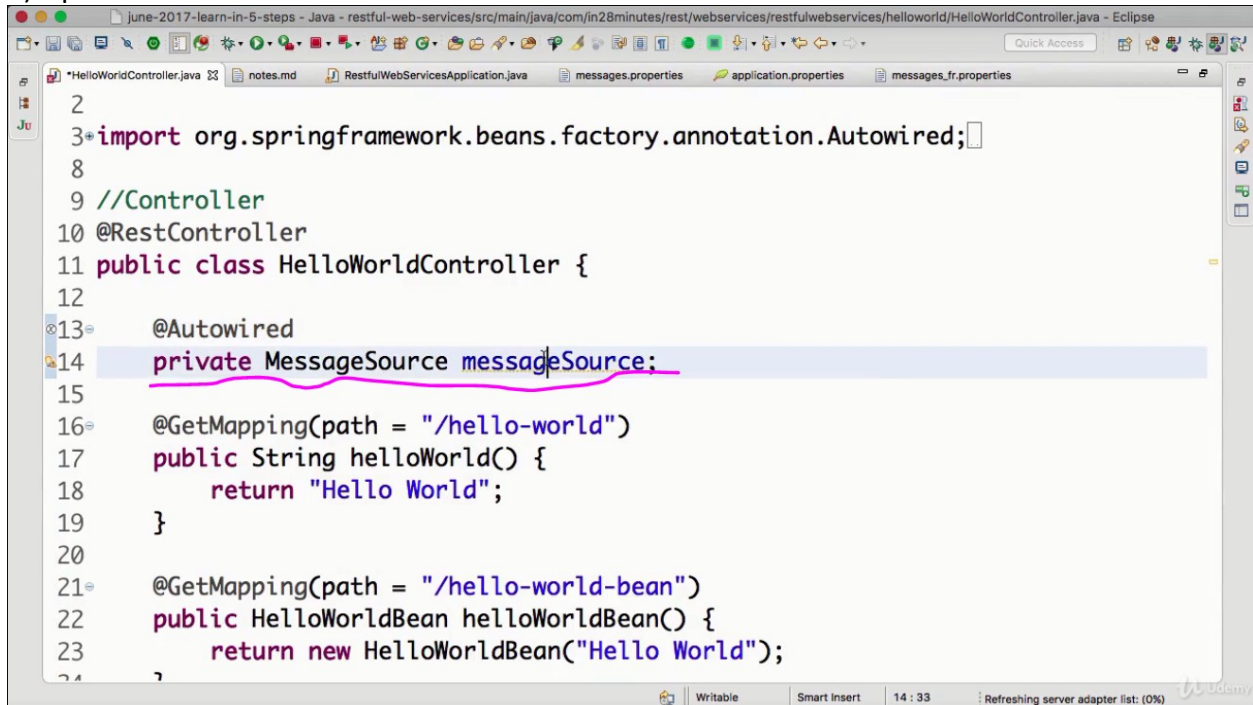
2)Add LocaleResolver & resourceBundleMessageSource to Spring Application



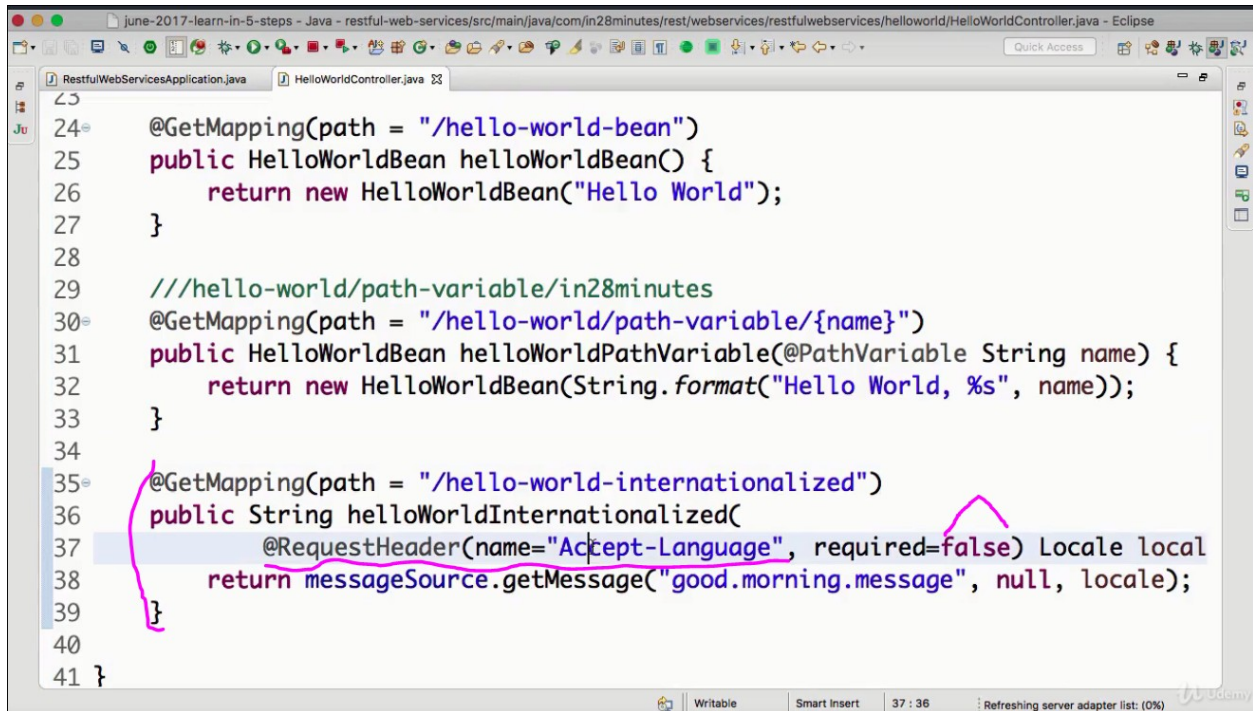
. Note

messageSource.setBaseName("message") ==> our messages are stored in message\* files.  
I.e message.properties, message\_fr.properties,message\_in.properties etc .

3) Update Controller to use the resource bundle



```
2
3 import org.springframework.beans.factory.annotation.Autowired;
8
9 //Controller
10 @RestController
11 public class HelloWorldController {
12
13     @Autowired
14     private MessageSource messageSource;
15
16     @GetMapping(path = "/hello-world")
17     public String helloWorld() {
18         return "Hello World";
19     }
20
21     @GetMapping(path = "/hello-world-bean")
22     public HelloWorldBean helloWorldBean() {
23         return new HelloWorldBean("Hello World");
24     }
25 }
```

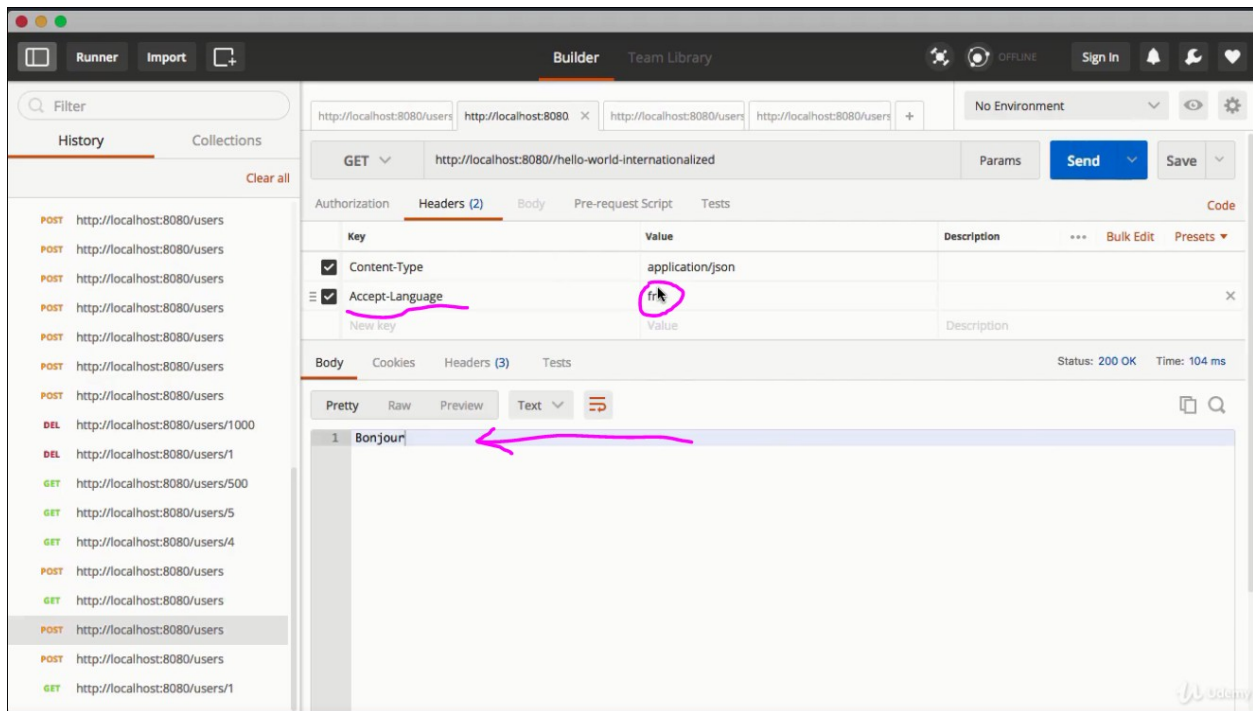


```
24 @GetMapping(path = "/hello-world-bean")
25 public HelloWorldBean helloWorldBean() {
26     return new HelloWorldBean("Hello World");
27 }
28
29 ///hello-world/path-variable/in28minutes
30 @GetMapping(path = "/hello-world/path-variable/{name}")
31 public HelloWorldBean helloWorldPathVariable(@PathVariable String name) {
32     return new HelloWorldBean(String.format("Hello World, %s", name));
33 }
34
35 @GetMapping(path = "/hello-world-internationalized")
36 public String helloWorldInternationalized(
37     @RequestHeader(name="Accept-Language", required=false) Locale locale
38 ) {
39     return messageSource.getMessage("good.morning.message", null, locale);
40 }
41 }
```

@ResourceHeader(name=Accept-Language,required=false) Locale locale

If the header attribute Accept-Language is present take the locale preference from here ,also required=false=> if not present take default locale.

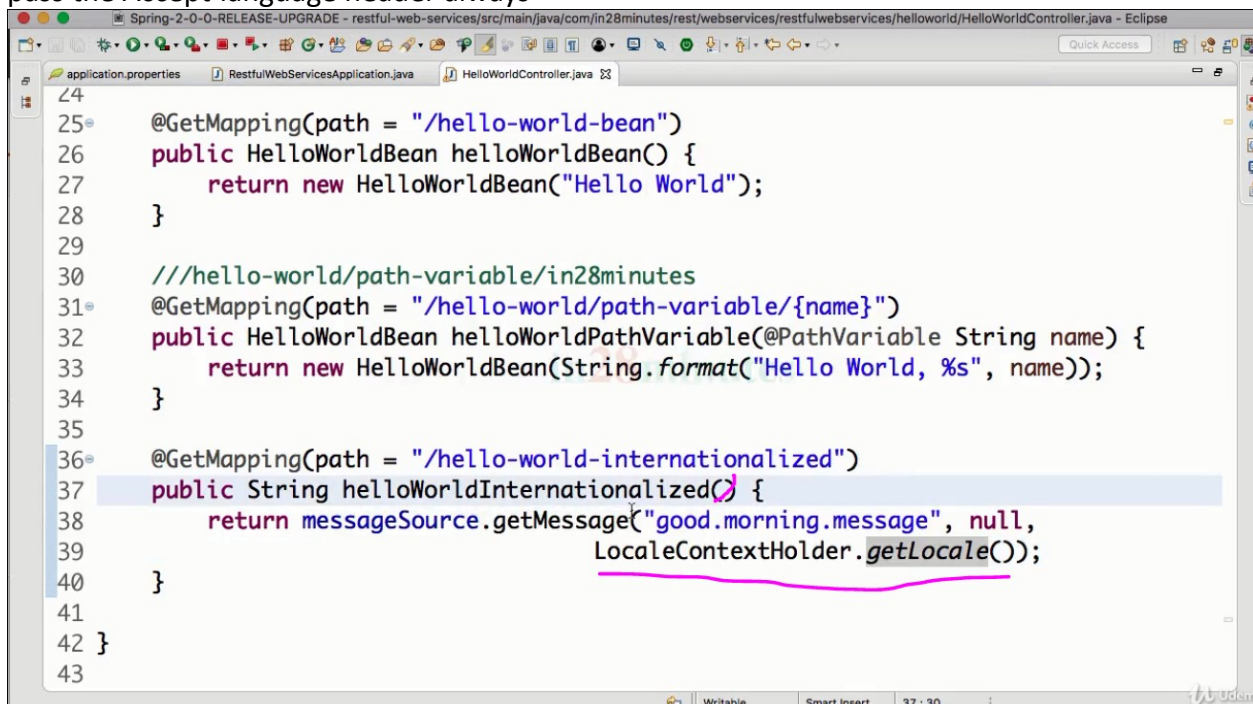
Test from postman:





FIX-1 (needs FIX-2 as well)

Now we have added to the method, it's a pain to add it to every method. So instead of picking it from the Accept-Language header let's pick it from LocaleContextHolder. That way we need not pass the Accept-language header always

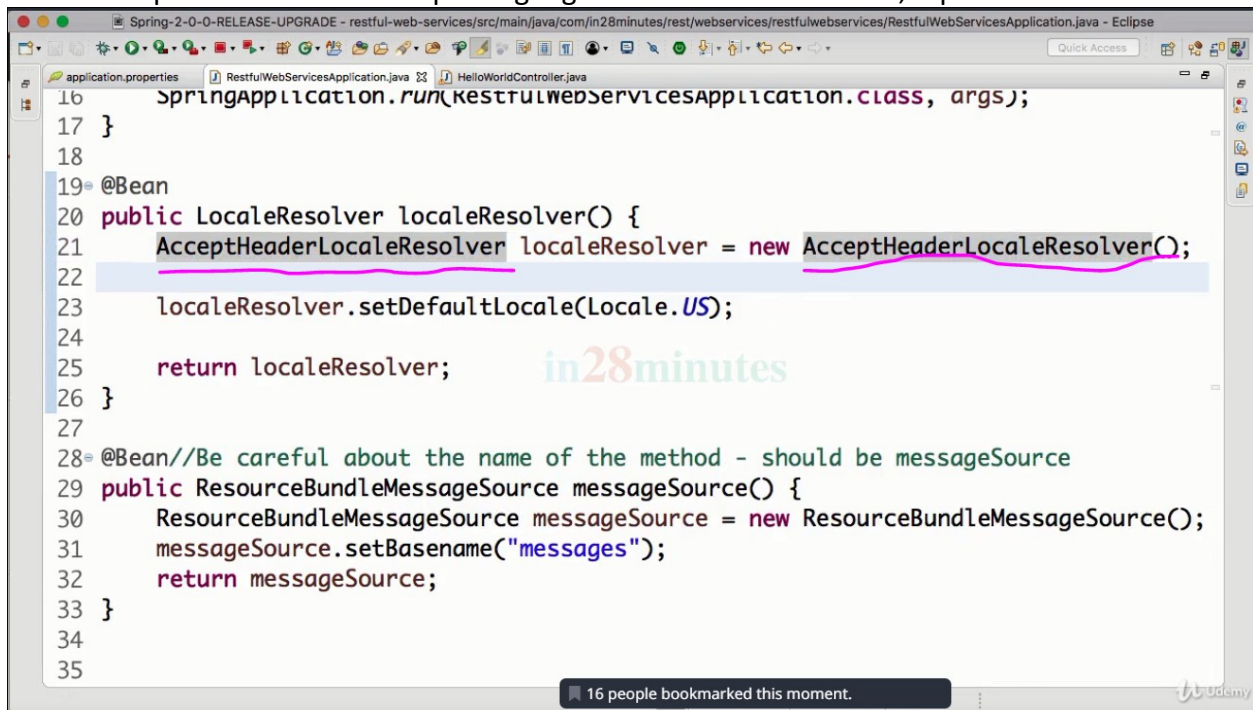


```
24
25 @GetMapping(path = "/hello-world-bean")
26 public HelloWorldBean helloWorldBean() {
27     return new HelloWorldBean("Hello World");
28 }
29
30 //hello-world/path-variable/in28minutes
31 @GetMapping(path = "/hello-world/path-variable/{name}")
32 public HelloWorldBean helloWorldPathVariable(@PathVariable String name) {
33     return new HelloWorldBean(String.format("Hello World, %s", name));
34 }
35
36 @GetMapping(path = "/hello-world-internationalized")
37 public String helloWorldInternationalized() {
38     return messageSource.getMessage("good.morning.message", null,
39                                     LocaleContextHolder.getLocale());
40 }
41
42 }
43
```



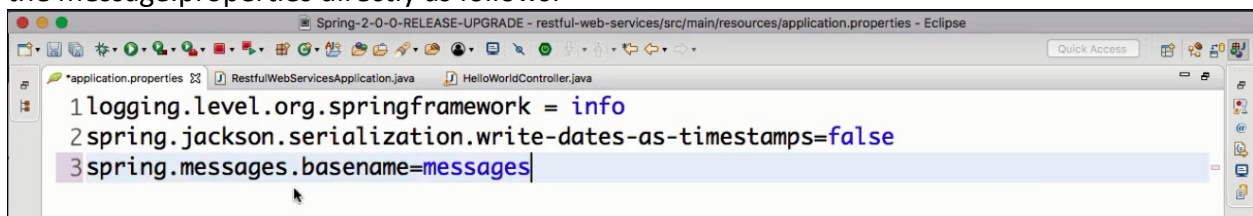
## FIX-2

Instead of SessionLocaleResolver we can use AcceptLocaleResolver, this enables SPRINGS to lead locale preference from Accept-Language header. If it is not here, it picks the default.



```
16 SpringApplication.run(RestfulWebServicesApplication.class, args);
17 }
18
19 @Bean
20 public LocaleResolver localeResolver() {
21     AcceptHeaderLocaleResolver localeResolver = new AcceptHeaderLocaleResolver();
22     localeResolver.setDefaultLocale(Locale.US);
23
24     return localeResolver;
25 }
26
27
28 @Bean // Be careful about the name of the method - should be messageSource
29 public ResourceBundleMessageSource messageSource() {
30     ResourceBundleMessageSource messageSource = new ResourceBundleMessageSource();
31     messageSource.setBasename("messages");
32     return messageSource;
33 }
34
35
```

FIX-3: The messageSource() can be removed from RestfulWebServiceApplication.java & done in the message.properties directly as follows:



```
1 logging.level.org.springframework = info
2 spring.jackson.serialization.write-dates-as-timestamps=false
3 spring.messages.basename=messages
```

restart  
the

application & test in POSTMAN client.

## Lecture 31: Content Negotiation

For GET Request

1. Add the below dependency to the pom

```
<dependency>
  <groupId>com.fasterxml.jackson.dataformat</groupId>
  <artifactId>jackson-dataformat-xml</artifactId>
  <version>2.8.5</version>
</dependency>
```

2. send in header Content-type: application/xml in the POSTMAN client

http://localhost:8080/ x + No Environment

GET http://localhost:8080/users Params Send Save

Authorization Headers (1) Body Pre-request Script Tests Code

Key	Value	Description	...	Bulk Edit	Presets
<input checked="" type="checkbox"/> Accept	application/xml				
New key	Value	Description			

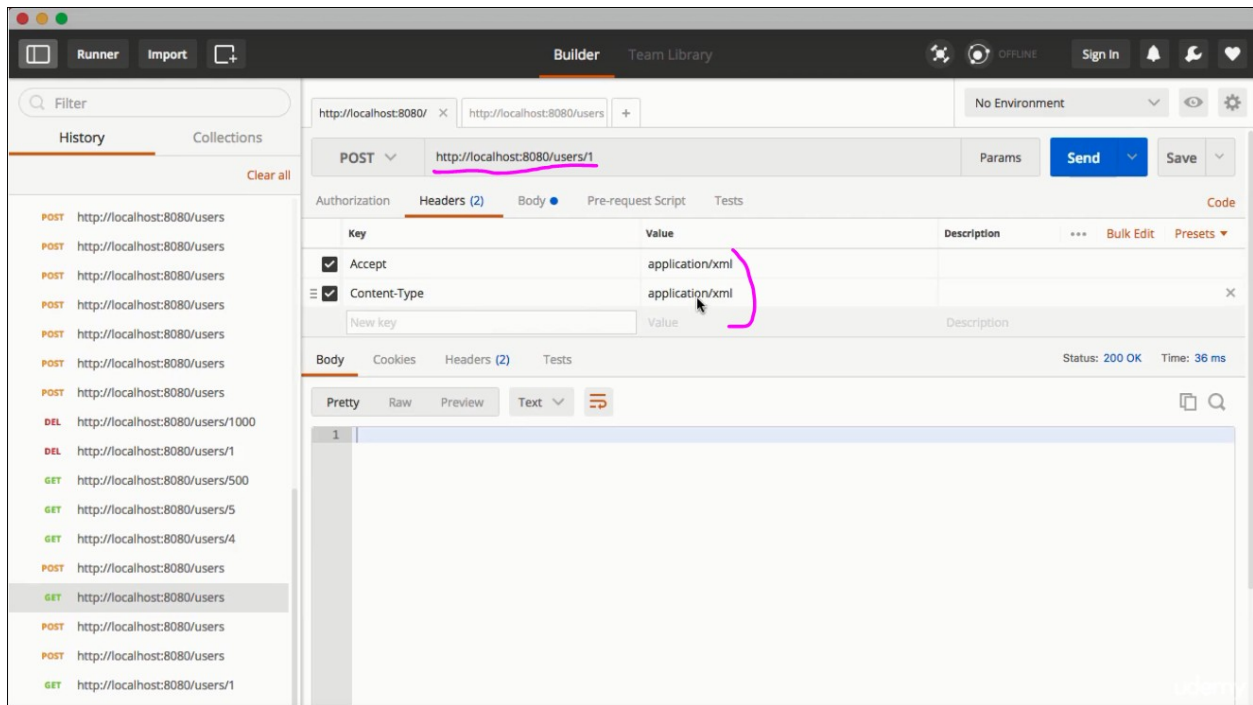
Body Cookies Headers (3) Tests Status: 200 OK Time: 893 ms

Pretty Raw Preview XML

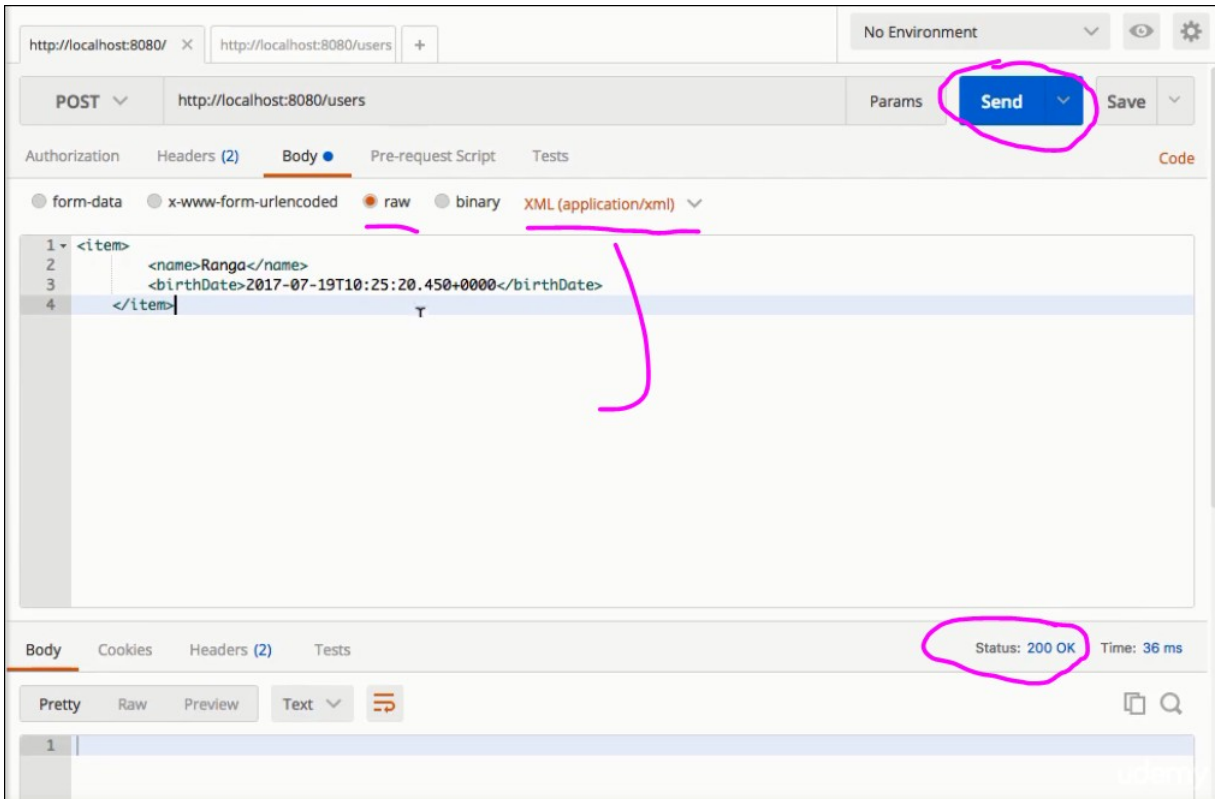
```
1 <list>
2   <item>
3     <id>1</id>
4     <name>Adam</name>
5     <birthDate>2017-07-19T10:25:20.450+0000</birthDate>
6   </item>
7   <item>
8     <id>2</id>
9     <name>Eve</name>
10    <birthDate>2017-07-19T10:25:20.450+0000</birthDate>
11  </item>
12  <item>
13    <id>3</id>
14    <name>Jack</name>
15    <birthDate>2017-07-19T10:25:20.450+0000</birthDate>
16  </item>
17 </list>
```

### For POST request:

1. In Header put the below :

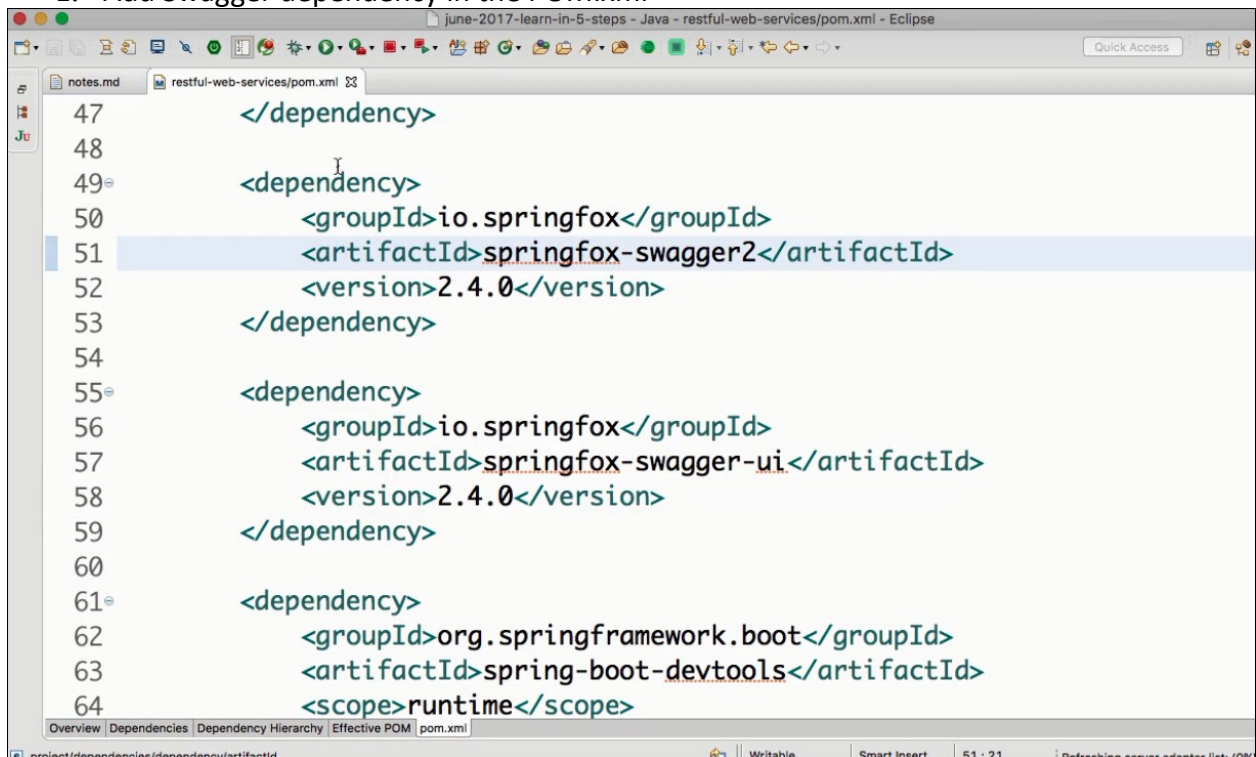


2. In Body put the below & test:

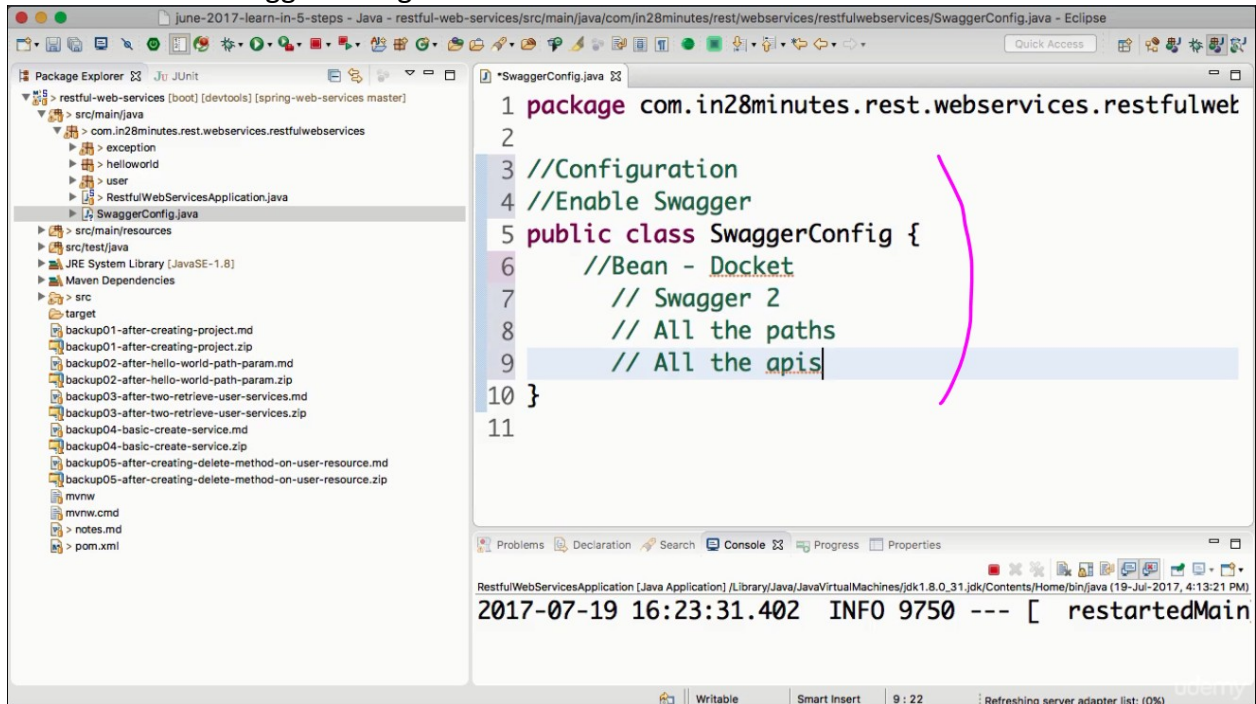


## Lecture 32,33,34: Swagger Documentation

### 1. Add Swagger dependency in the POM.xml



## 2. Add the Swagger Configuration



The screenshot shows the Eclipse IDE with the Package Explorer on the left and the Editor on the right. The Package Explorer shows the project structure: `restful-web-services` (boot) [devtools] [spring-web-services master] with sub-packages `src/main/java` and `src/main/resources`. The `src/main/java` package contains `com.in28minutes.rest.webservices.restfulwebservices`, which includes `SwaggerConfig.java`. The Editor shows the content of `SwaggerConfig.java`:

```
1 package com.in28minutes.rest.webservices.restfulweb
2
3 //Configuration
4 //Enable Swagger
5 public class SwaggerConfig {
6     //Bean - Docket
7     // Swagger 2
8     // All the paths
9     // All the apis
10 }
11
```

The console at the bottom shows the output of the application:

```
RestfulWebServicesApplication [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_31.jdk/Contents/Home/bin/java (19-Jul-2017, 4:13:21 PM)
2017-07-19 16:23:31.402 INFO 9750 --- [ restartedMain
```



The screenshot shows the Eclipse IDE with the Package Explorer on the left and the Editor on the right. The Package Explorer shows the project structure: `restful-web-services` (boot) [devtools] [spring-web-services master] with sub-packages `src/main/java` and `src/main/resources`. The `src/main/java` package contains `com.in28minutes.rest.webservices.restfulwebservices`, which includes `SwaggerConfig.java`. The Editor shows the content of `SwaggerConfig.java`:

```
20
21 // Bean - Docket
22 @Bean
23 public Docket api() {
24     HashSet<String> consumesAndProduces = new HashSet<String>(Arrays.asList("application/json", "application/xml"));
25     // Swagger 2- doc type
26     return new Docket(DocumentationType.SWAGGER_2)
27         .apiInfo(metadata())
28         .consumes(consumesAndProduces)
29         .produces(consumesAndProduces)
30         .pathMapping("/");
31 }
32
33 private ApiInfo metadata() {
34     return new ApiInfoBuilder()
35         .title("Spring Boot Proj API")
36         .description("Spring Boot API Description")
37         .version("1.0")
38         .contact(new Contact("Ranga", "http://www.in28minutes.com",
39             "in28minutes@gmail.com"))
40         .license("Apache 2.0")
41         .licenseUrl("http://www.apache.org/licenses/LICENSE-2.0")
42         .build();
43 }
44
45
46 // All the paths
47 // All the apis
48 }
```

An orange arrow points from the `Info` label to the `metadata()` method.

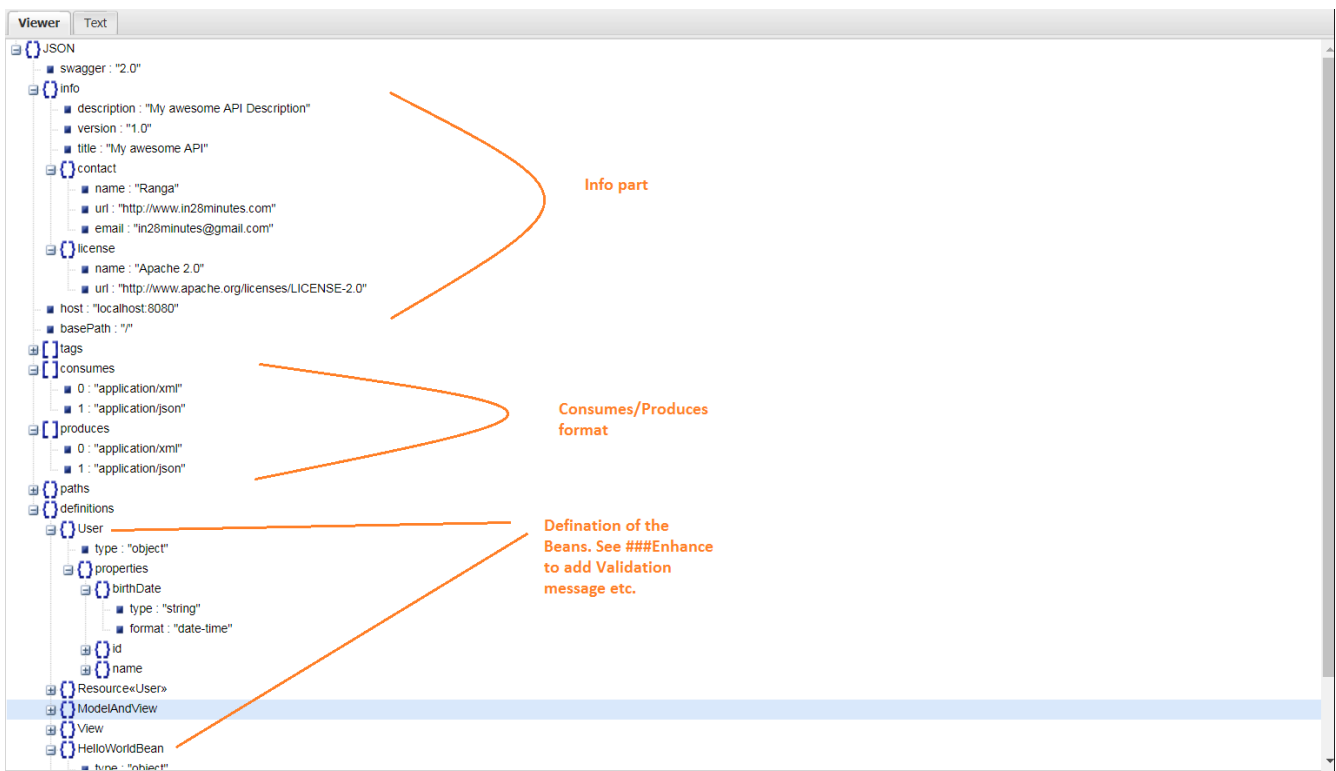
## 3. Check the output at the below URL:

<http://localhost:8080/v2/api-docs> → JSON view ( analyze using a JSON viewer)

<http://localhost:8080/swagger-ui.html> → web UI View

## 4. Now See the section of definitions. Bean Definitions are provided .





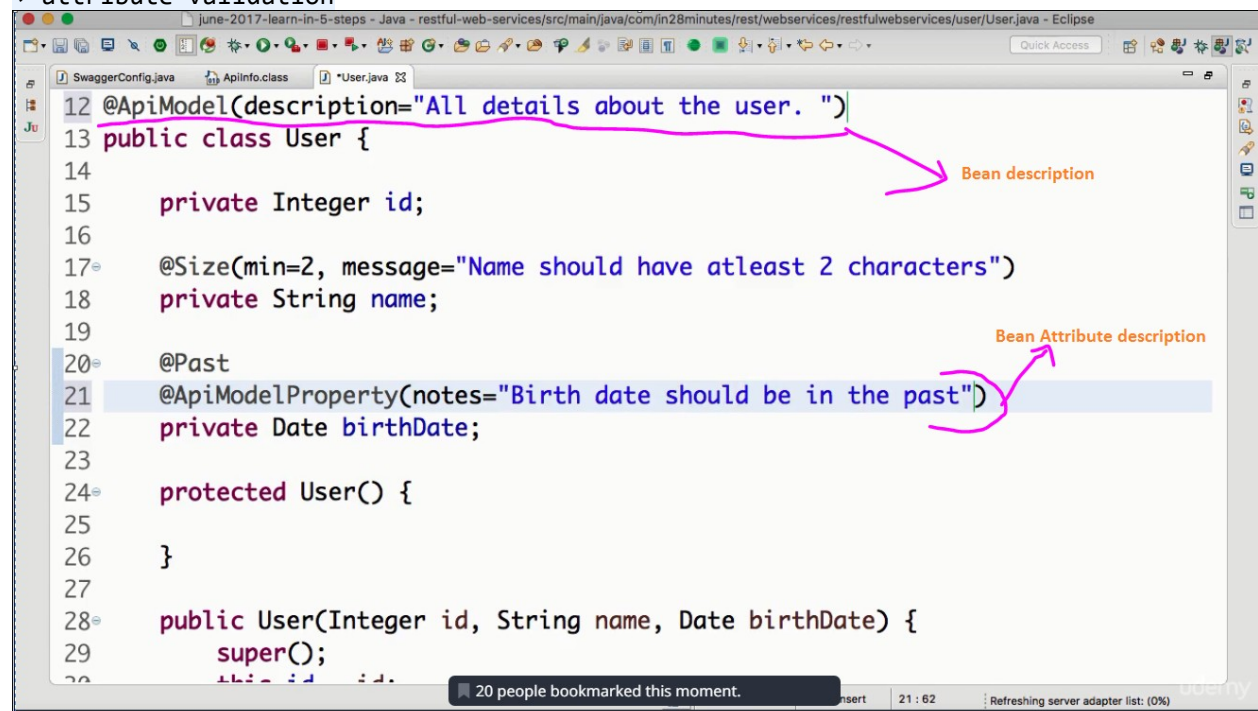
### ###Enhance:

Use the below annotations on the bean:

`@ApiModelProperty(description="All Details about the user")` → Bean description

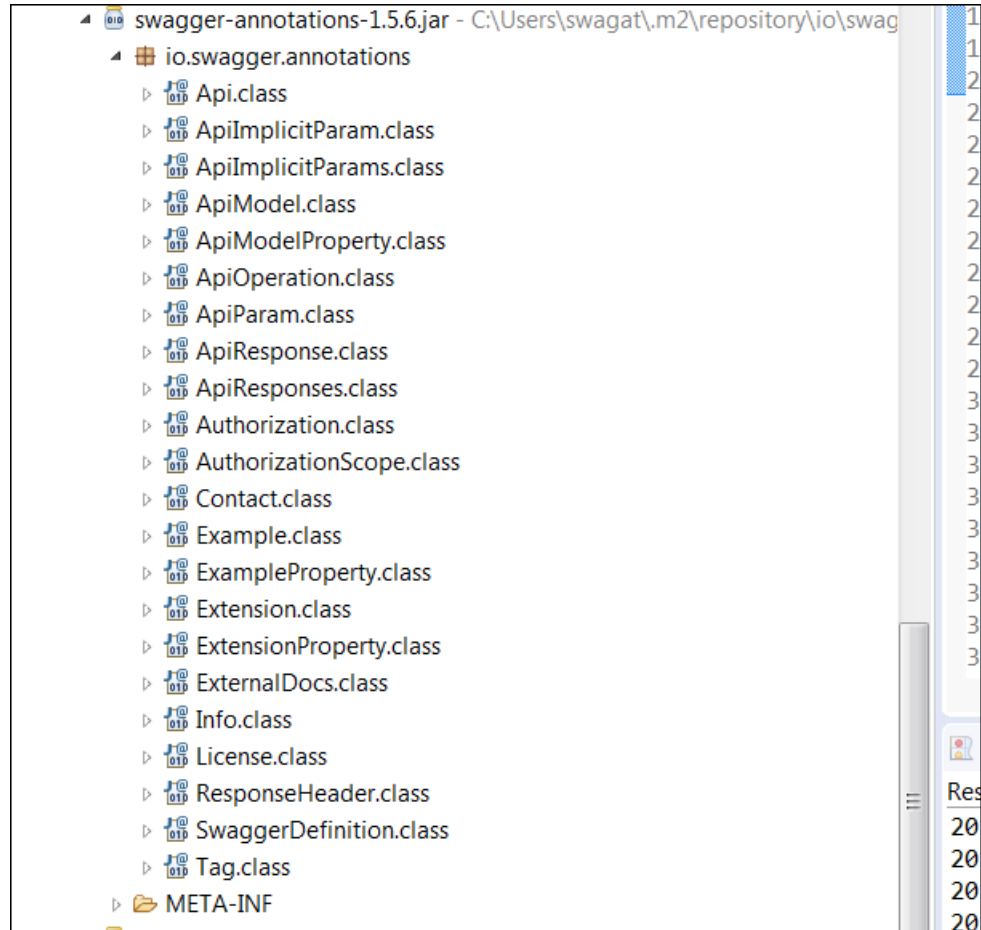
`@ApiModelPropertyProperty(notes="Name should at least be 2 character & max of 50 character")`

→ attribute validation



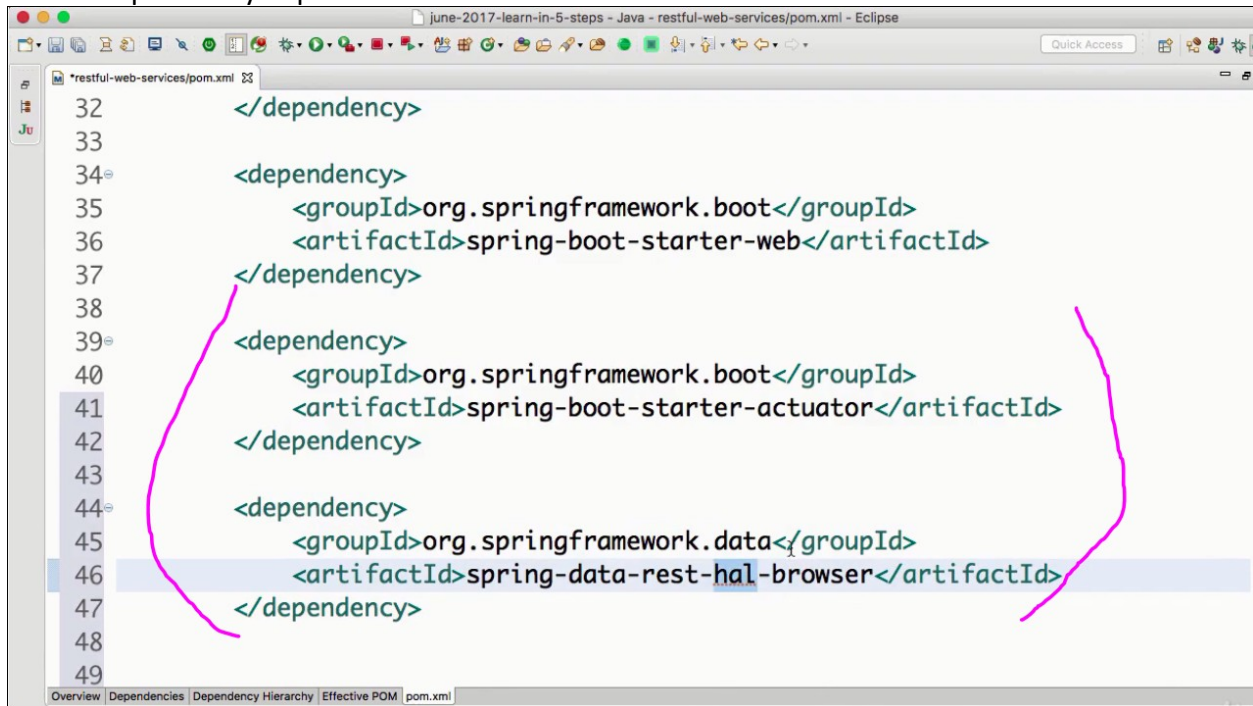


Check out the swagger annotations below:



## Lecture 35: Monitoring API with --- Spring Boot Actuator

### 1. Add dependency in pom.xml



```
32     </dependency>
33
34     <dependency>
35         <groupId>org.springframework.boot</groupId>
36         <artifactId>spring-boot-starter-web</artifactId>
37     </dependency>
38
39     <dependency>
40         <groupId>org.springframework.boot</groupId>
41         <artifactId>spring-boot-starter-actuator</artifactId>
42     </dependency>
43
44     <dependency>
45         <groupId>org.springframework.data</groupId>
46         <artifactId>spring-data-rest-hal-browser</artifactId>
47     </dependency>
48
49
```

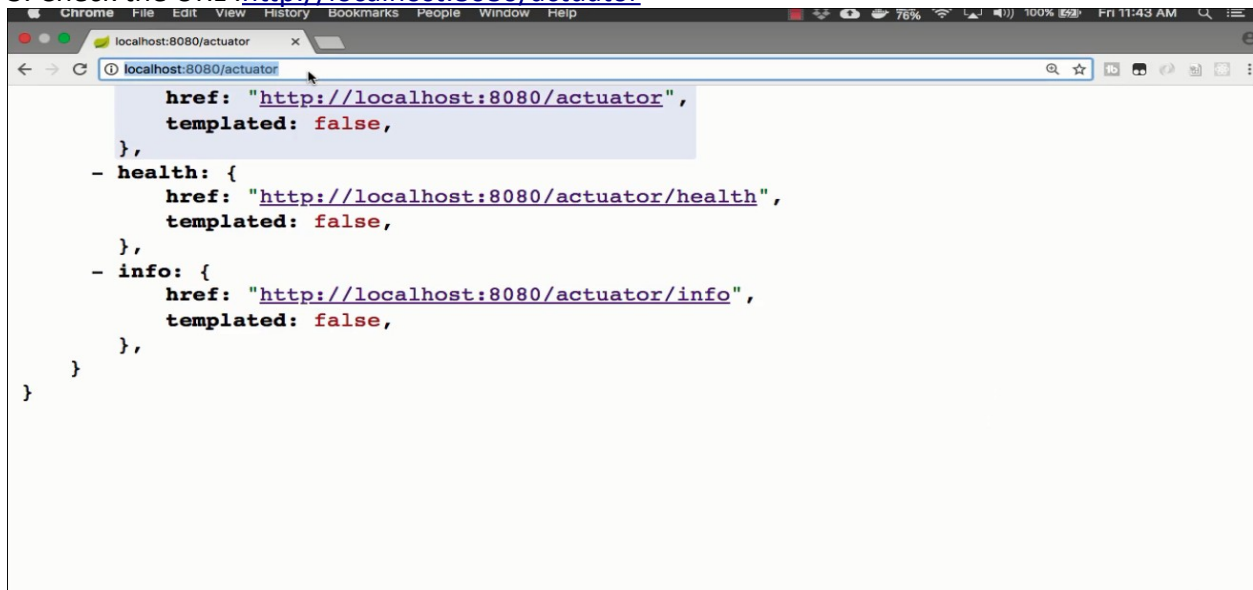
starter-actuator is for → monitoring health/matrices

hal-browser → for interpreting the actuator info. HAL => Hypertext Application Language

Ref: [http://stateless.co/hal\\_specification.html](http://stateless.co/hal_specification.html)

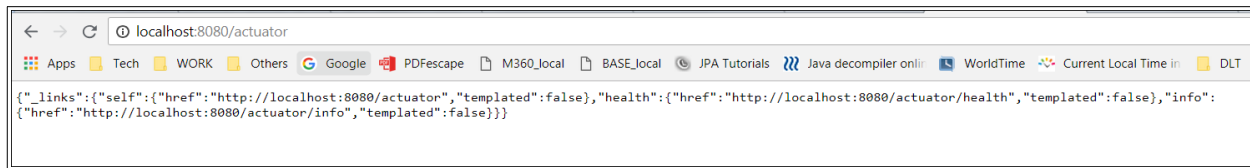
### 2. Restart the application

### 3. Check the URL :<http://localhost:8080/actuator>

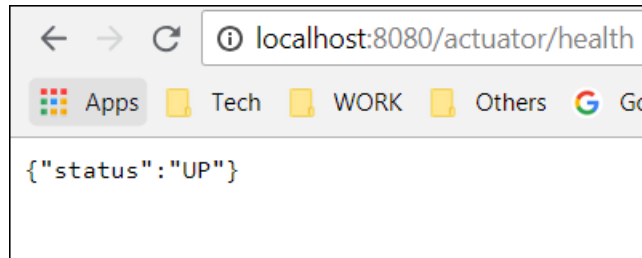


```
{
  href: "http://localhost:8080/actuator",
  templated: false,
},
- health: {
  href: "http://localhost:8080/actuator/health",
  templated: false,
},
- info: {
  href: "http://localhost:8080/actuator/info",
  templated: false,
},
}
}
```

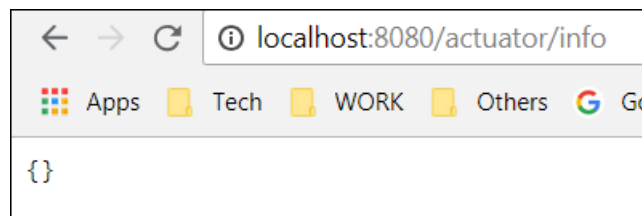
When visited these 3 links:



```
{ "_links": { "self": { "href": "http://localhost:8080/actuator", "templated": false }, "health": { "href": "http://localhost:8080/actuator/health", "templated": false }, "info": { "href": "http://localhost:8080/actuator/info", "templated": false } } }
```

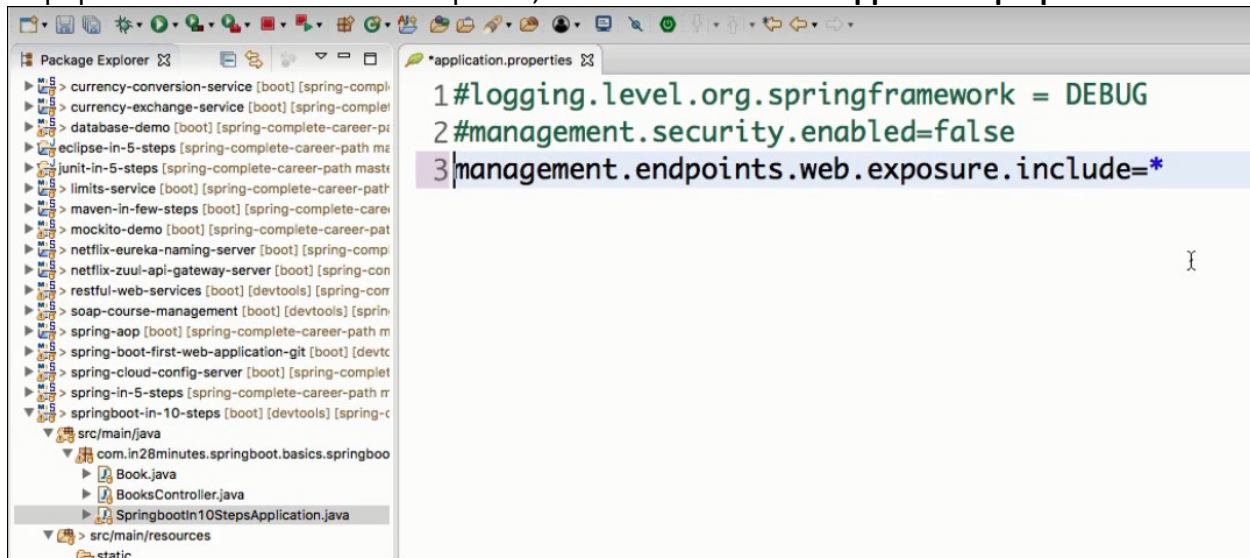


```
{ "status": "UP" }
```

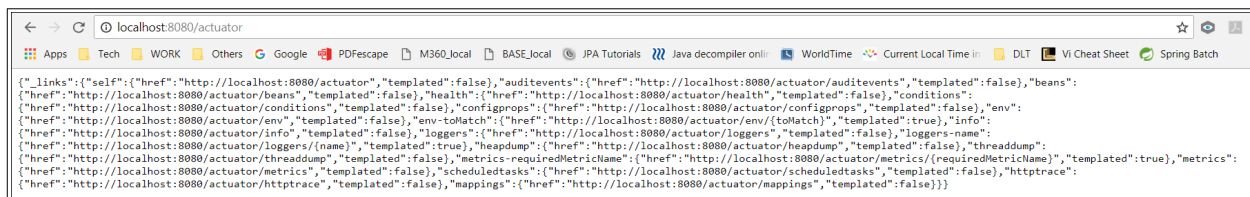


```
{ }
```

To populate the actuator with all api info, add the below line in **application.properties**.



```
1#logging.level.org.springframework = DEBUG
2#management.security.enabled=false
3management.endpoints.web.exposure.include=*
```



```
{ "_links": { "self": { "href": "http://localhost:8080/actuator", "templated": false }, "auditevents": { "href": "http://localhost:8080/actuator/auditevents", "templated": false }, "beans": { "href": "http://localhost:8080/actuator/beans", "templated": false }, "health": { "href": "http://localhost:8080/actuator/health", "templated": false }, "conditions": { "href": "http://localhost:8080/actuator/conditions", "templated": false }, "configprops": { "href": "http://localhost:8080/actuator/configprops", "templated": false }, "env": { "href": "http://localhost:8080/actuator/env", "templated": false }, "env-toMatch": { "href": "http://localhost:8080/actuator/env/{toMatch}", "templated": true }, "info": { "href": "http://localhost:8080/actuator/info", "templated": false }, "loggers": { "href": "http://localhost:8080/actuator/loggers", "templated": false }, "logger-name": { "href": "http://localhost:8080/actuator/loggers/{name}", "templated": true }, "heapdump": { "href": "http://localhost:8080/actuator/heapdump", "templated": false }, "threaddump": { "href": "http://localhost:8080/actuator/threaddump", "templated": false }, "metrics-requiredMetricName": { "href": "http://localhost:8080/actuator/metrics/{requiredMetricName}", "templated": true }, "metrics": { "href": "http://localhost:8080/actuator/metrics", "templated": false }, "scheduledtasks": { "href": "http://localhost:8080/actuator/scheduledtasks", "templated": false }, "httptrace": { "href": "http://localhost:8080/actuator/httptrace", "templated": false }, "mappings": { "href": "http://localhost:8080/actuator/mappings", "templated": false } } }
```

Now lets use the HAL-Browser to see the output:

The HAL Browser (for Spring Data REST) interface is shown. The URL bar indicates the endpoint is `localhost:8080/brower/index.html#/actuator/`. The main content area is divided into several sections:

- Explorer**: A search bar with the text `/actuator/` and a `Go!` button.
- Custom Request Headers**: A text input field for adding custom headers.
- Properties**: A text input field for adding custom properties.
- Links**: A table listing available endpoints:

rel	title	name / index	docs	GET	NON-GET
self					
auditevents					
beans					
health					
conditions					
configprops					

- Inspector**:
  - Response Headers**:

```
200 success
date: Tue, 11 Sep 2018 01:42:18 GMT
transfer-encoding: chunked
content-type: application/json;charset=UTF-8
```
  - Response Body**:

```
{
  "_links": {
    "self": {
      "href": "http://localhost:8080/actuator",
      "templated": false
    },
    "auditevents": {
      "href": "http://localhost:8080/actuator/auditevents",
      "templated": false
    },
    "beans": {
      "href": "http://localhost:8080/actuator/beans",
      "templated": false
    },
    "health": {
      "href": "http://localhost:8080/actuator/health",
      "templated": false
    },
    "conditions": {
      "href": "http://localhost:8080/actuator/conditions",
      "templated": false
    },
    "configprops": {
      "href": "http://localhost:8080/actuator/configprops",
      "templated": false
    }
  }
}
```