***DispatcherServletAutoConfiguration matched means it found the class org.springframework.web.servlet.DispatcherServlet on its class path.***

*We have added the spring boot starter web and spring boot starter web having the dependency on Spring MVC FRAMEWORK.*

*Spring boot autoconfiguration says I have found org.springframework.web.servlet.DispatcherServlet on class path.*

**DispatcherServletAutoConfiguration matched:**

- @ConditionalOnClass found required class 'org.springframework.web.servlet.DispatcherServlet'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)

- @ConditionalOnWebApplication (required) found StandardServletEnvironment (OnWebApplicationCondition)

DispatcherServletAutoConfiguration.DispatcherServletConfiguration matched:

- @ConditionalOnClass found required class 'javax.servlet.ServletRegistration'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)

- Default DispatcherServlet did not find dispatcher servlet beans (DispatcherServletAutoConfiguration.DefaultDispatcherServletCondition)

DispatcherServletAutoConfiguration.DispatcherServletRegistrationConfiguration matched:

- @ConditionalOnClass found required class 'javax.servlet.ServletRegistration'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)

- DispatcherServlet Registration did not find servlet registration bean (DispatcherServletAutoConfiguration.DispatcherServletRegistrationCondition)

DispatcherServletAutoConfiguration.DispatcherServletRegistrationConfiguration#dispatcherServletRegistration matched:

- @ConditionalOnBean (names: dispatcherServlet; types: org.springframework.web.servlet.DispatcherServlet; SearchStrategy: all) found beans 'dispatcherServlet', 'dispatcherServlet' (OnBeanCondition)

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

*All these configurations are getting fired due to spring boot auto configuration.*

*Spring boot looks for all the classes and jar at its class path, based on whatever on its class path, it tried to autoconfigured accordingly.*

ErrorMvcAutoConfiguration matched:

- @ConditionalOnClass found required classes 'javax.servlet.Servlet', 'org.springframework.web.servlet.DispatcherServlet'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)

- @ConditionalOnWebApplication (required) found StandardServletEnvironment (OnWebApplicationCondition)

ErrorMvcAutoConfiguration#basicErrorController matched:

- @ConditionalOnMissingBean (types: org.springframework.boot.autoconfigure.web.ErrorController; SearchStrategy: current) did not find any beans (OnBeanCondition)

ErrorMvcAutoConfiguration#errorAttributes matched:

- @ConditionalOnMissingBean (types: org.springframework.boot.autoconfigure.web.ErrorAttributes; SearchStrategy: current) did not find any beans (OnBeanCondition)

ErrorMvcAutoConfiguration.DefaultErrorViewResolverConfiguration#conventionErrorViewResolver matched:

- @ConditionalOnBean (types: org.springframework.web.servlet.DispatcherServlet; SearchStrategy: all) found bean 'dispatcherServlet'; @ConditionalOnMissingBean (types: org.springframework.boot.autoconfigure.web.DefaultErrorViewResolver; SearchStrategy: all) did not find any beans (OnBeanCondition)

ErrorMvcAutoConfiguration.WhitelabelErrorViewConfiguration matched:

- @ConditionalOnProperty (server.error.whitelabel.enabled) matched (OnPropertyCondition)

- ErrorTemplate Missing did not find error template view (ErrorMvcAutoConfiguration.ErrorTemplateMissingCondition)

ErrorMvcAutoConfiguration.WhitelabelErrorViewConfiguration#beanNameViewResolver matched:

- @ConditionalOnMissingBean (types: org.springframework.web.servlet.view.BeanNameViewResolver; SearchStrategy: all) did not find any beans (OnBeanCondition)

ErrorMvcAutoConfiguration.WhitelabelErrorViewConfiguration#defaultErrorView matched:

- @ConditionalOnMissingBean (names: error; SearchStrategy: all) did not find any beans (OnBeanCondition)

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

*We have bean object is getting automatically converted into JSON so below converter are responsible for doing that.*

HttpMessageConvertersAutoConfiguration matched:

- @ConditionalOnClass found required class 'org.springframework.http.converter.HttpMessageConverter'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)

HttpMessageConvertersAutoConfiguration#messageConverters matched:

- @ConditionalOnMissingBean (types: org.springframework.boot.autoconfigure.web.HttpMessageConverters; SearchStrategy: all) did not find any beans (OnBeanCondition)

HttpMessageConvertersAutoConfiguration.StringHttpMessageConverterConfiguration matched:

- @ConditionalOnClass found required class 'org.springframework.http.converter.StringHttpMessageConverter'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)

HttpMessageConvertersAutoConfiguration.StringHttpMessageConverterConfiguration#stringHttpMessageConverter matched:

- @ConditionalOnMissingBean (types: org.springframework.http.converter.StringHttpMessageConverter; SearchStrategy: all) did not find any beans (OnBeanCondition)

***\*\*\* Below JacksonAutoConfiguration does the conversation form bean object to json and json to be an object***

JacksonAutoConfiguration matched:

- @ConditionalOnClass found required class 'com.fasterxml.jackson.databind.ObjectMapper'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)

JacksonAutoConfiguration.Jackson2ObjectMapperBuilderCustomizerConfiguration matched:

- @ConditionalOnClass found required classes 'com.fasterxml.jackson.databind.ObjectMapper', 'org.springframework.http.converter.json.Jackson2ObjectMapperBuilder'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)

JacksonAutoConfiguration.JacksonObjectMapperBuilderConfiguration matched:

- @ConditionalOnClass found required classes 'com.fasterxml.jackson.databind.ObjectMapper', 'org.springframework.http.converter.json.Jackson2ObjectMapperBuilder'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)

JacksonAutoConfiguration.JacksonObjectMapperBuilderConfiguration#jacksonObjectMapperBuilder matched:

- @ConditionalOnMissingBean (types: org.springframework.http.converter.json.Jackson2ObjectMapperBuilder; SearchStrategy: all) did not find any beans (OnBeanCondition)

JacksonAutoConfiguration.JacksonObjectMapperConfiguration matched:

- @ConditionalOnClass found required classes 'com.fasterxml.jackson.databind.ObjectMapper', 'org.springframework.http.converter.json.Jackson2ObjectMapperBuilder'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)

JacksonAutoConfiguration.JacksonObjectMapperConfiguration#jacksonObjectMapper matched:

- @ConditionalOnMissingBean (types: com.fasterxml.jackson.databind.ObjectMapper; SearchStrategy: all) did not find any beans (OnBeanCondition)

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**What is dispatcher servlet ?**

**From the debug logs**

**-- Mapping servlet : 'dispatcherServlet' to [/]**

**-- Dipatcher servlet is the a front controller**

**Who is configuring dispatcher servlet ?**

**SpringBootAutoConfiguration**

**What does dispatcher servlet do ?**

**How does the HelloWorldBean object get converted to JSON?**

**its due to SpringbootAutoConfiguration and JacksonBean object is getting initialized.**

**Who is configuring the error mapping?**

**SpringBootAutoConfiguration - its create the default error page us.**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**@ResponseBody**

**public @interface RestController {**

**}**

***@ResponseBody : Once a particular method return the response then this response is getting mapped with some of the other message converter.***

***in our case due to JacksonConver - our response is getting converted from bean to json format.***

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

***http://localhost:2019/hello-bean***

***Once we hit the above URL, hello-bean request to go to dispatcher servlet,***

***then Dispatcher servlet find the respective controller, it will call the respective method and then it will get the bean object done the message conversation and its return it as a response to the client.***

***Dispatcher servlet is playing the important roll.***

2021-09-09 20:09:02.136 DEBUG 10972 --- [ main] s.w.s.m.m.a.RequestMappingHandlerMapping : 3 request handler methods found on class com.springboot.controller.HellowWorldController: {public com.springboot.model.HelloWorld com.springboot.controller.HellowWorldController.getHelloWorldBean()={ [/hello-bean],methods=[GET]}, public java.lang.String com.springboot.controller.HellowWorldController.helloGet()={[/hello-get],methods=[GET]}, public java.lang.String com.springboot.controller.HellowWorldController.helloWorld()={[/hello-world],methods=[GET]}}

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

[**@RequestBody**](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestBody.html) : Annotation indicating a method parameter should be bound to the body of the HTTP request.

For example:

@RequestMapping(path = "/something", method = RequestMethod.PUT)

**public** **void** handle(@RequestBody String body, Writer writer) **throws** IOException {

writer.write(body);

}

[**@ResponseBody**](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/ResponseBody.html) annotation can be put on a method and indicates that the return type should be written straight to the HTTP response body.

For example:

@RequestMapping(path = "/something", method = RequestMethod.PUT)

**public** @ResponseBody String helloWorld() {

**return** "Hello World";

}

***How to set the return the status of created user into the Response?***

***How to set the URI of created resource into the Response?***

URI location = ServletUriComponentsBuilder.fromCurrentRequest().path("{id").buildAndExpand(user.getId()).toUri();

return ResponseEntity.created(location).build();

After running check the response at the header of the response - will get the location

<http://localhost:2019/createUsers/kumar>

***Exception Handling in Spring boot***

***if the user is not found what will be the response need to display on the screen.***

@GetMapping("/users/{id}")

**public** User retrievedUsers(@PathVariable **int** id){

User user = userService.findOne(id);

// Adding exception if the user not found in the list

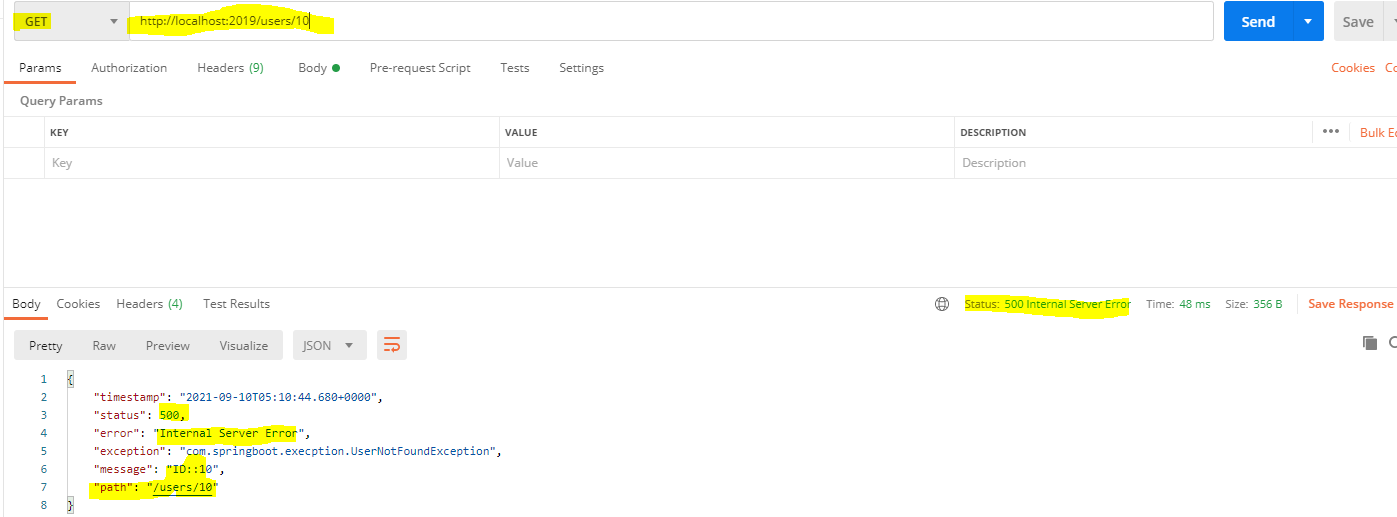
**if**(user == **null** ) {

**throw** **new** UserNotFoundException("ID::"+id);

}

**return** user;

}



*It is not a Internal Server Error.*

*We are trying to fetch the record, which is not available in the list or database, instead of returning Internal Server Error how to provide the exact reason for this failure.*

*How?*

*We can achieve by using @ResponseStatus(HttpStatus.NOT\_FOUND)*

@ResponseStatus(HttpStatus.***NOT\_FOUND***)

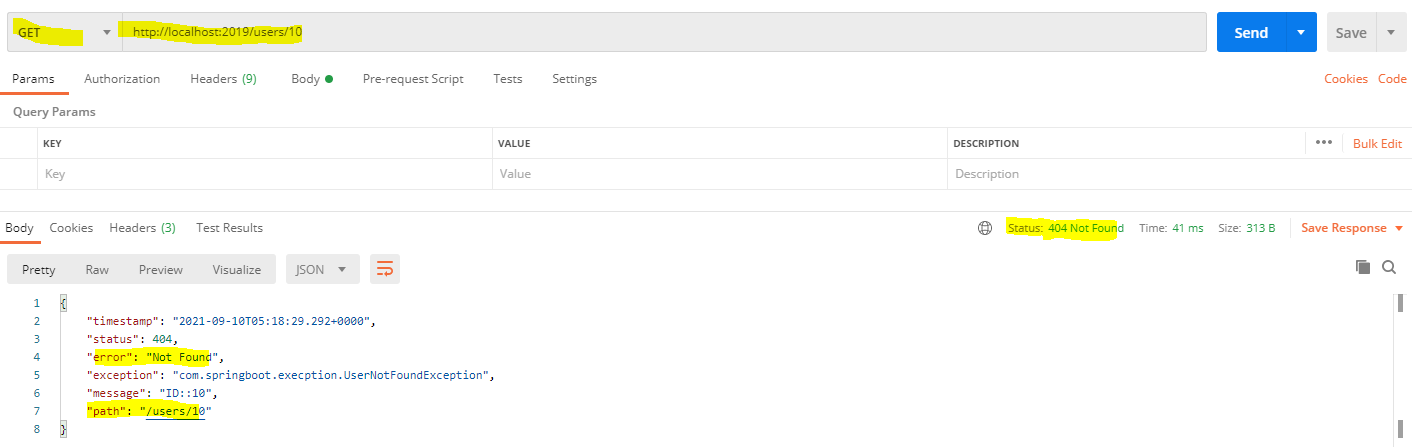
**public** **class** UserNotFoundException **extends** RuntimeException {

**public** UserNotFoundException(String message) {

**super**(message);

}

}



***How to construct/design your own exception message?***

We can modify below field details as per the company requirement.

**public** **class** ExceptionResponse {

**private** Date timestamp;

**private** String message;

**private** String details;

**public** ExceptionResponse(Date timestamp, String message, String details) {

**super**();

**this**.timestamp = timestamp;

**this**.message = message;

**this**.details = details;

}

**public** Date getTimestamp() {

**return** timestamp;

}

**public** String getMessage() {

**return** message;

}

**public** String getDetails() {

**return** details;

}

}

*Earlier we were getting like below*

**{**

**"timestamp": "2021-09-10T05:18:29.292+0000",**

**"status": 404,**

**"error": "Not Found",**

**"exception": "com.springboot.execption.UserNotFoundException",**

**"message": "ID::10",**

**"path": "/users/10"**

**}**

*Whenever exception happen, we want the exception in below format*

**{**

**"timestamp": "2021-09-10T05:18:29.292+0000",**

**"details": "com.springboot.execption.UserNotFoundException",**

**"message": "ID::10"**

**}**

*How? Below code for handling all the exception and it will provide the message as INTERNAL SERVER ERROR*

**@ControllerAdvice**

**@RestController**

**public class CustomizedResponseEntityExceptionHandlerController extends ResponseEntityExceptionHandler {**

**@ExceptionHandler(Exception.class)**

**public final ResponseEntity<Object> handleAllException(Exception ex, WebRequest request) {**

**ExceptionResponse exceptionResponse =**

**new ExceptionResponse(new Date(), ex.getMessage(), request.getDescription(false));**

**return new ResponseEntity<>(exceptionResponse,HttpStatus.*INTERNAL\_SERVER\_ERROR*);**

**}**

**}**



If we want to this class CustomizedResponseEntityExceptionHandlerController in another controller class also

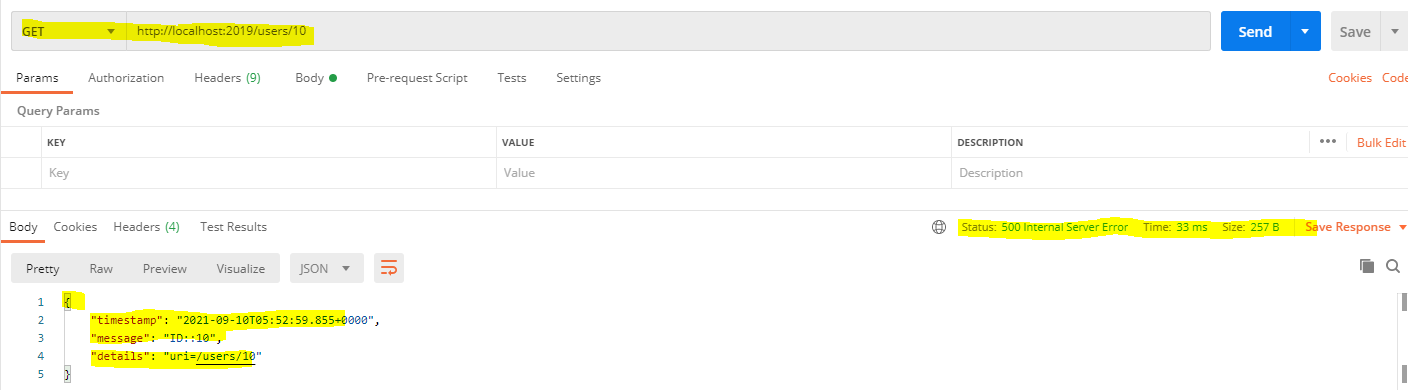
then we must use @ControllerAdvice

Specialization of {@link Component @Component} for classes that declare

{@link ExceptionHandler @ExceptionHandler}, {@link InitBinder @InitBinder}, or

{@link ModelAttribute @ModelAttribute} methods to be shared across

multiple {@code @Controller} classes.



**//Below will handle the specific exception UsernotfoundExcpetion**

**@ExceptionHandler(UserNotFoundException.class)**

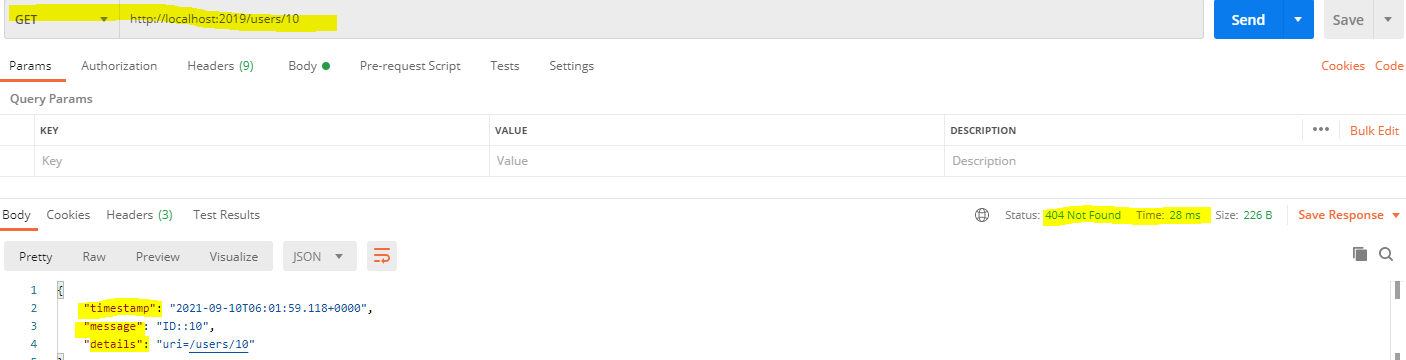
**public final ResponseEntity<Object> handleUserException(UserNotFoundException ex, WebRequest request) {**

**ExceptionResponse exceptionResponse =**

**new ExceptionResponse(new Date(), ex.getMessage(), request.getDescription(false));**

**return new ResponseEntity<>(exceptionResponse,HttpStatus.*NOT\_FOUND*);**

**}**



***How to delete the use?***

[

    {

        "id": 1,

        "name": "Amit",

        "birthDate": "2021-09-10T06:24:57.136+0000"

    },

    {

        "id": 2,

        "name": "Niraj",

        "birthDate": "2021-09-10T06:24:57.136+0000"

    },

    {

        "id": 3,

        "name": "Sandeep",

        "birthDate": "2021-09-10T06:24:57.136+0000"

    },

    {

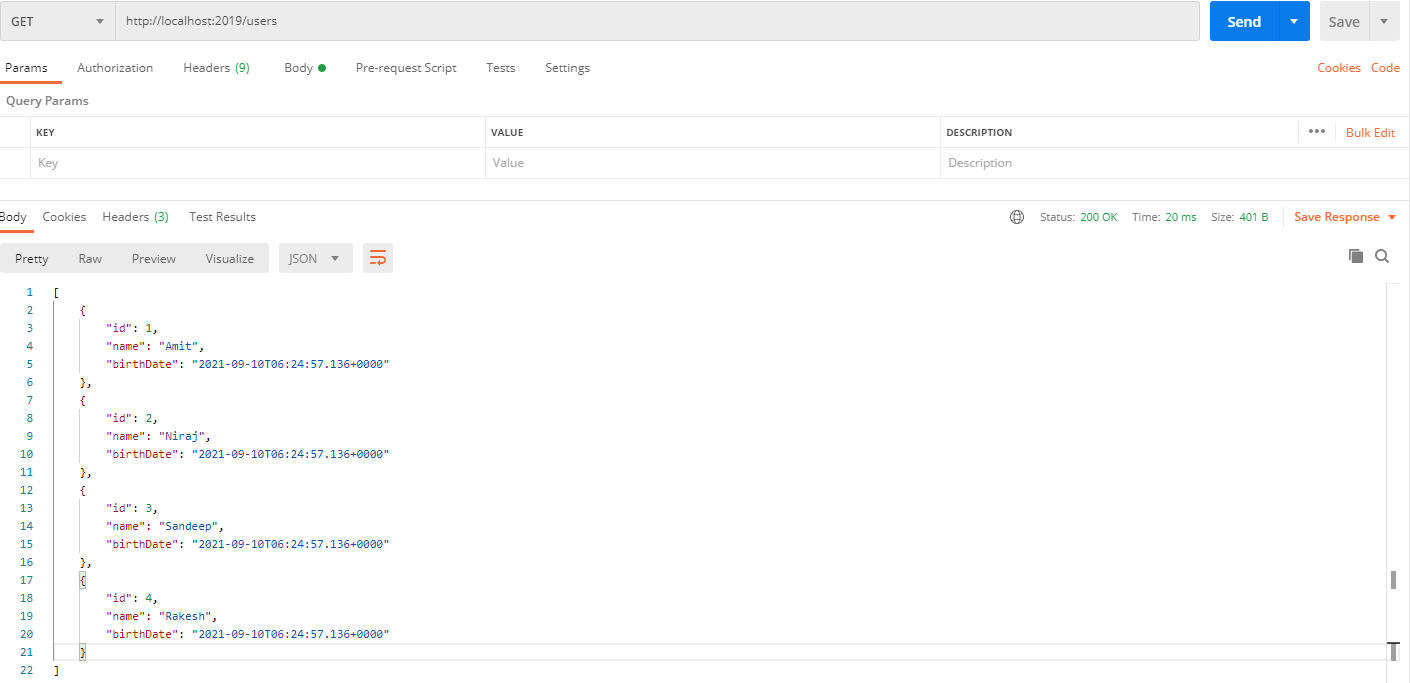
        "id": 4,

        "name": "Rakesh",

        "birthDate": "2021-09-10T06:24:57.136+0000"

    }

]



// if the user deleted success full then it will return response status 200 else it will throw UserNotFoundExcpetion

@DeleteMapping("/delete/{id}")

**public** **void** deleteUserByID(@PathVariable **int** id){

User user = userService.deleteByID(id);

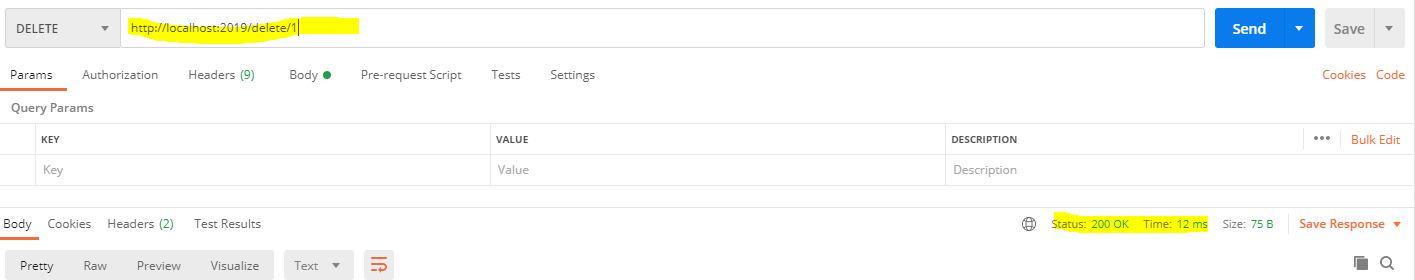
**if**(user == **null** ) {

**throw** **new** UserNotFoundException("ID::"+id);

}

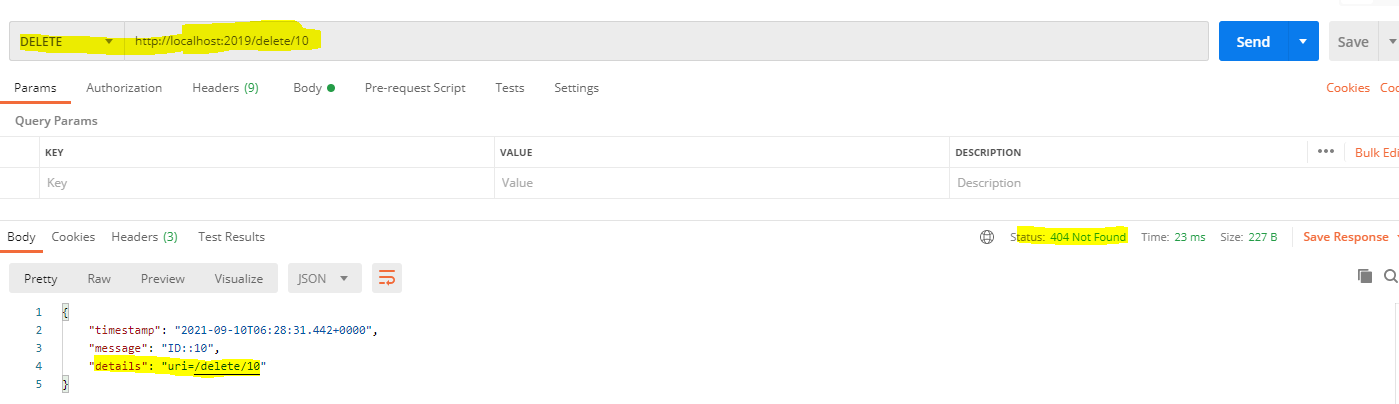
}

*Once the respective records get deleted then we will get only HttpStatus code as 200.*

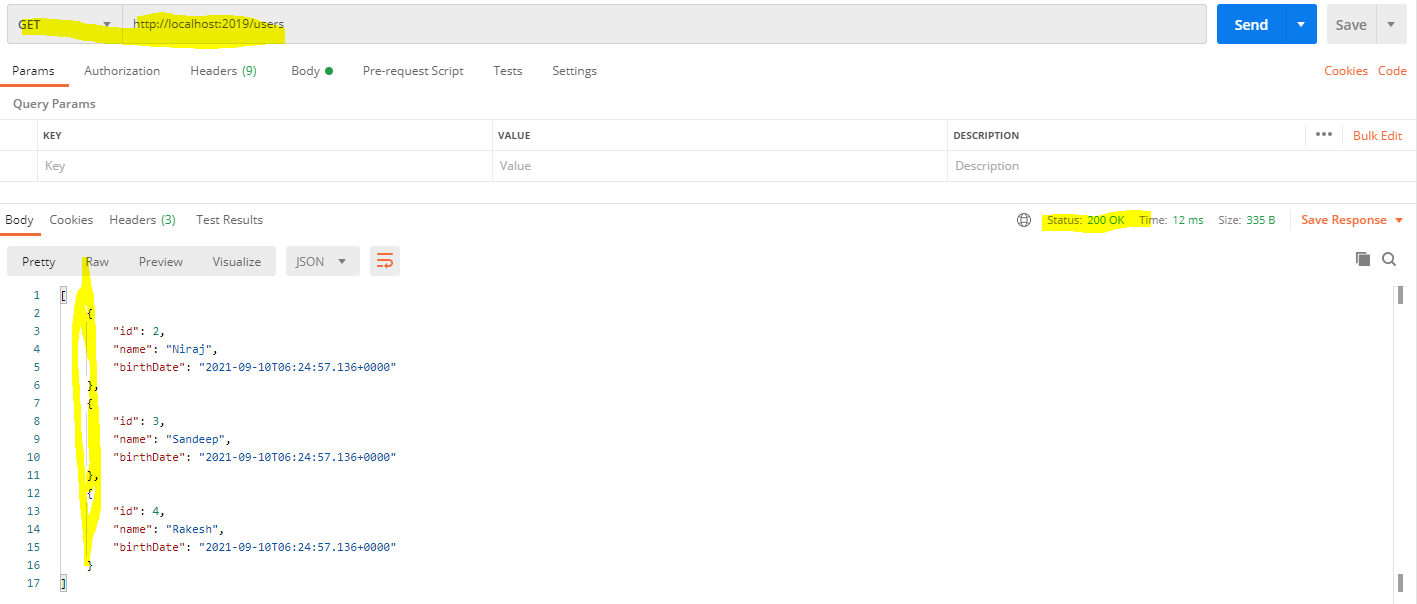


*In case of record not found*

*We must show the meaning full message to the end users.*



*Check the remaining user.*



Record with id =1 is not present.

***Adding dependency for validation***

***Step 15***

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>**spring-boot-starter-validation**</artifactId>

</dependency>

We will build a simple Spring MVC application that takes user input and checks the input by using standard validation annotations.

Data validation is a basic requirement for any application. This is more significant for web applications that accept data as input.

[Bean Validation](https://beanvalidation.org/) or commonly known as [JSR-380](https://beanvalidation.org/2.0-jsr380/) is a Java standard that is used to perform validation in Java applications.

To perform validation, data Items are applied constraints. As long as the data satisfies these constraints, it will be considered valid.

We can apply the built-in constraints as annotation in the fields of a Java class. The constraints can be a numerical value range, being null or not null, having a specific number of items or length, for instance.

In this post, I will explain how to use Bean Validation in a Spring Boot application.

*Some of the most common validation annotations are:*

* **@NotNull:** to say that a field must not be null.
* **@NotEmpty:** to say that a list field must not empty.
* **@NotBlank:** to say that a string field must not be the empty string (i.e. it must have at least one character).
* **@Min and @Max:** to say that a numerical field is only valid when it’s value is above or below a certain value.
* **@Pattern:** to say that a string field is only valid when it matches a certain regular expression.
* **@Email:** to say that a string field must be a valid email address.

**Step 15 Validation Implementation**

**public** **class** User {

**private** Integer id;

@Size(min = 2, message = "Name should have atleast 2 characters!!")

**private** String name;

// Birth date can not be future it should be in past

@Past

**private** Date birthDate;

**public** User() {

// **TODO** Auto-generated constructor stub

}

**public** User(Integer id, String name, Date birthDate) {

**super**();

**this**.id = id;

**this**.name = name;

**this**.birthDate = birthDate;

}

//Develop getter method

}

**@PostMapping("/createUsers")**

**public ResponseEntity<Object> createUser(@Valid @RequestBody User user) {**

**userService.save(user);**

**//ResponseEntity<T>.created(location)**

**URI location = ServletUriComponentsBuilder.*fromCurrentRequest*().path("/{name}").buildAndExpand(user.getName()).toUri();**

**return ResponseEntity.*created*(location).build();**

**@RestController**

**public class CustomizedResponseEntityExceptionHandlerController extends ResponseEntityExceptionHandler {**

**@Override**

**protected ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException ex,**

**HttpHeaders headers, HttpStatus status, WebRequest request) {**

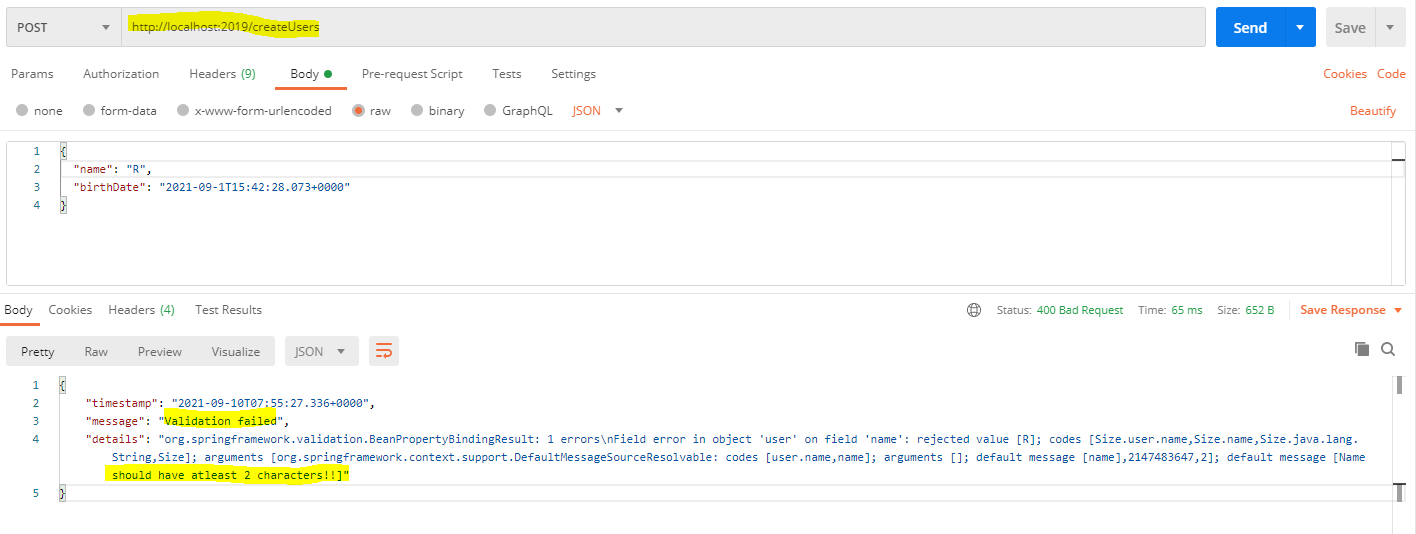
**ExceptionResponse exceptionResponse =**

**new ExceptionResponse(new Date(), "Validation failed", ex.getBindingResult().toString());**

**return new ResponseEntity<>(exceptionResponse,HttpStatus.*BAD\_REQUEST*);**

**}**

**}**



***Step 16***

***Implementation of HATEOAS for RESTful services.***



*In addition to getting the data, we are also seeing some actions on for it the above journaldev github branch it having many more features likes fork, MIT License, showing the updated date and stars*

*So, if we want such additional features for our webpage then must use HATEOAS.*

***HATEOAS – Hypermedia as the Engine of Application State.***

*What Is HATEOAS?*

What do we see when we visit a web page?

The data that we would want to see. Is that all? We would also see links and buttons to see related data.

For example, if we go to a student page, we will see

* Student profile
* Links to Edit and Delete Student details
* Links to see details of other students
* Link to see details of the courses and grades of the student

HATEOAS brings the same concepts to RESTful Web Services.

When some details of a resource are requested, we will provide the resource details as well as details of related resources and the possible actions you can perform on the resource.

For example, when requesting information about a Facebook user, a REST service can return the following

* User details
* Links to get his recent posts
* Links to get his recent comments
* Links to retrieve his friend’s list.

Need to add the below dependency

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-hateoas</artifactId>

</dependency>

@GetMapping("/users/{id}")

**public** EntityModel<User> retrieveUsers(@PathVariable **int** id){

User user = userService.findOne(id);

**if**(user == **null** ) {

**throw** **new** UserNotFoundException("ID::"+id);

}

//Creating object for EntityModel

EntityModel<User> model = EntityModel.*of*(user);

//Build the link

WebMvcLinkBuilder linkToUsers = *linkTo*(*methodOn*(**this**.getClass()).retrievedAllUsers());

//Adding linkToUsers to model

model.add(linkToUsers.withRel("all-users"));

**return** model;

}

**import** org.springframework.hateoas.EntityModel;

**import** org.springframework.hateoas.server.mvc.WebMvcLinkBuilder;

//importing all the static method from WebMvcLinkBuilder

**import** **static** org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.\*;

*Requested URL*

[***http://localhost:2019/users/2***](http://localhost:2019/users/2)

*Response*

{

   "id": 2,

    "name": "Niraj",

    "birthDate": "2021-09-10T12:18:39.163+00:00",

**"\_links": {**

**"all-users": {**

**"href": "http://localhost:2019/users"**

**}**

**}**

}

**Here we are seeing additional links.**

***Step-17 (its just about the overview of Advanced RESTful webservice)***

***Step-18***

***Advanced RESTful webservice features (Internationalization-i18)***

**Internationalization – i18**

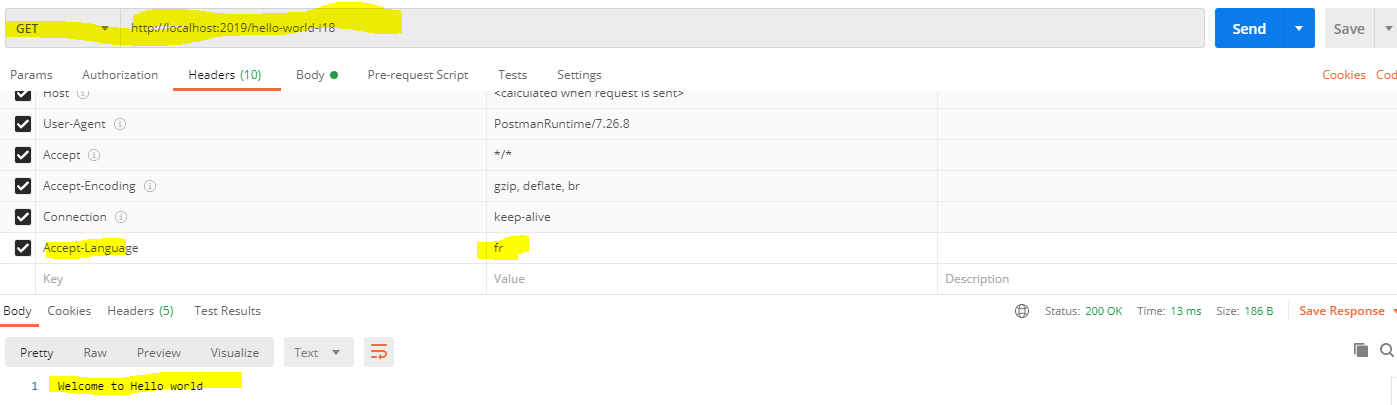
**Customizing our web services with different peoples around the world so that they can easily use it.**

@GetMapping( value="/hello-world-i18")

**public** String helloWorldInternationalization(){

**return** "Welcome to Hello world";

}



**Even Accept-language is fr – France – message still showing in English its because message is not internalization zed now.**

**How to make our API to pick up above header and action it accordingly.**

**@RestController**

**public class HelloWorldInternationalization {**

***@Autowired***

***private MessageSource messageSource;***

**@GetMapping(value = "/hello-world-i18")**

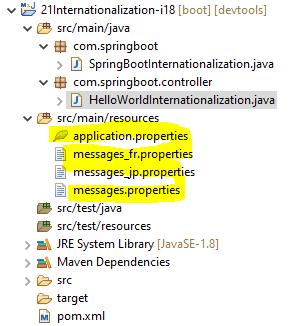
**public String helloWorldInternationalization(**

***@RequestHeader(name = "Accept-Language", required = false) Locale locale)* {**

**return messageSource.getMessage("welcome.message", null, "Default Message", locale);**

**}**

**}**



**From**

**message.properties**

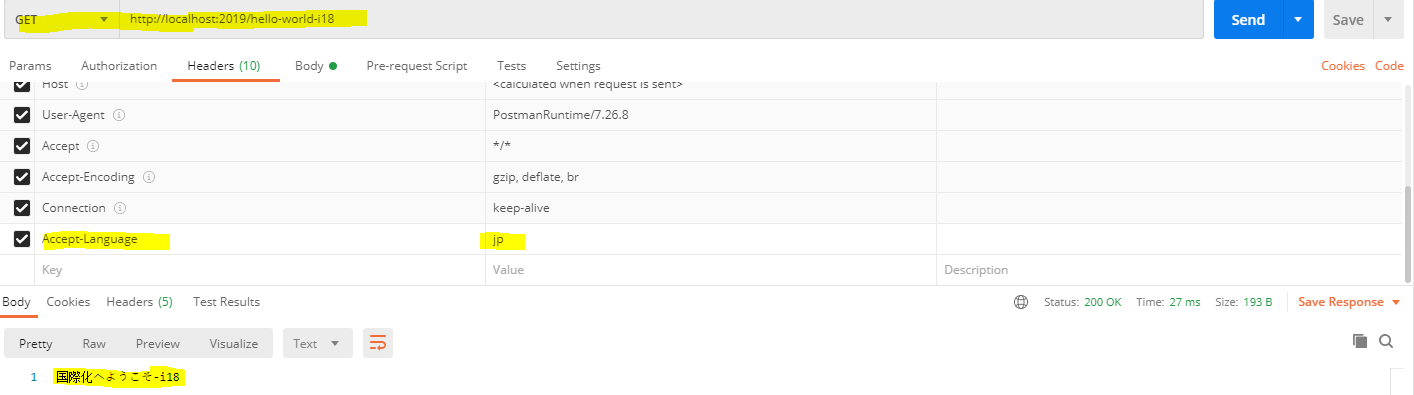
welcome.message=Welcome to Internationalization - i18

messages\_fr.properties

welcome.message=Bienvenue dans l'internationalisation - i18

messages\_jp.properties

welcome.message=\u56FD\u969B\u5316\u3078\u3088\u3046\u3053\u305D-i18



/\*

Suppose we have to implements Locale in n -numbers of method – then

in this approach we have to take n – numbers of Local in method parameter.

Instead of each method get the local we can re-write the logic in better way.

\*/

@GetMapping(value = "/hello-world2-i18")

**public** String helloWorldInternationalizationBetterImp() {

**return** messageSource.getMessage("welcome.message", **null**, "Default Message", **LocaleContextHolder.*getLocale*()**);

}



***Step-19 Content Negotiation – Implementing Support for XML***

[http://localhost:2019/**users**](http://localhost:2019/users)

Here users is the resource and we getting the output in json format.

[

    {

        "id": 1,

        "name": "Amit",

        "birthDate": "2021-09-10T06:24:57.136+0000"

    },

    {

        "id": 2,

        "name": "Niraj",

        "birthDate": "2021-09-10T06:24:57.136+0000"

    },

    {

        "id": 3,

        "name": "Sandeep",

        "birthDate": "2021-09-10T06:24:57.136+0000"

    },

    {

        "id": 4,

        "name": "Rakesh",

        "birthDate": "2021-09-10T06:24:57.136+0000"

    }

]

**Json is widely being used.**

**How will get the xml representation.**

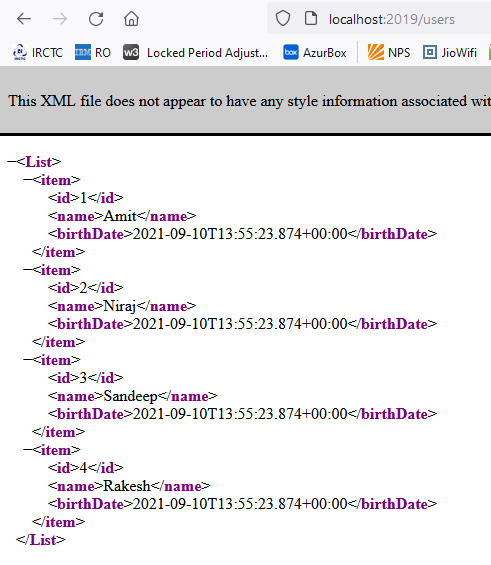
**We have to add the dependency for it into the POM.xml**<dependency>

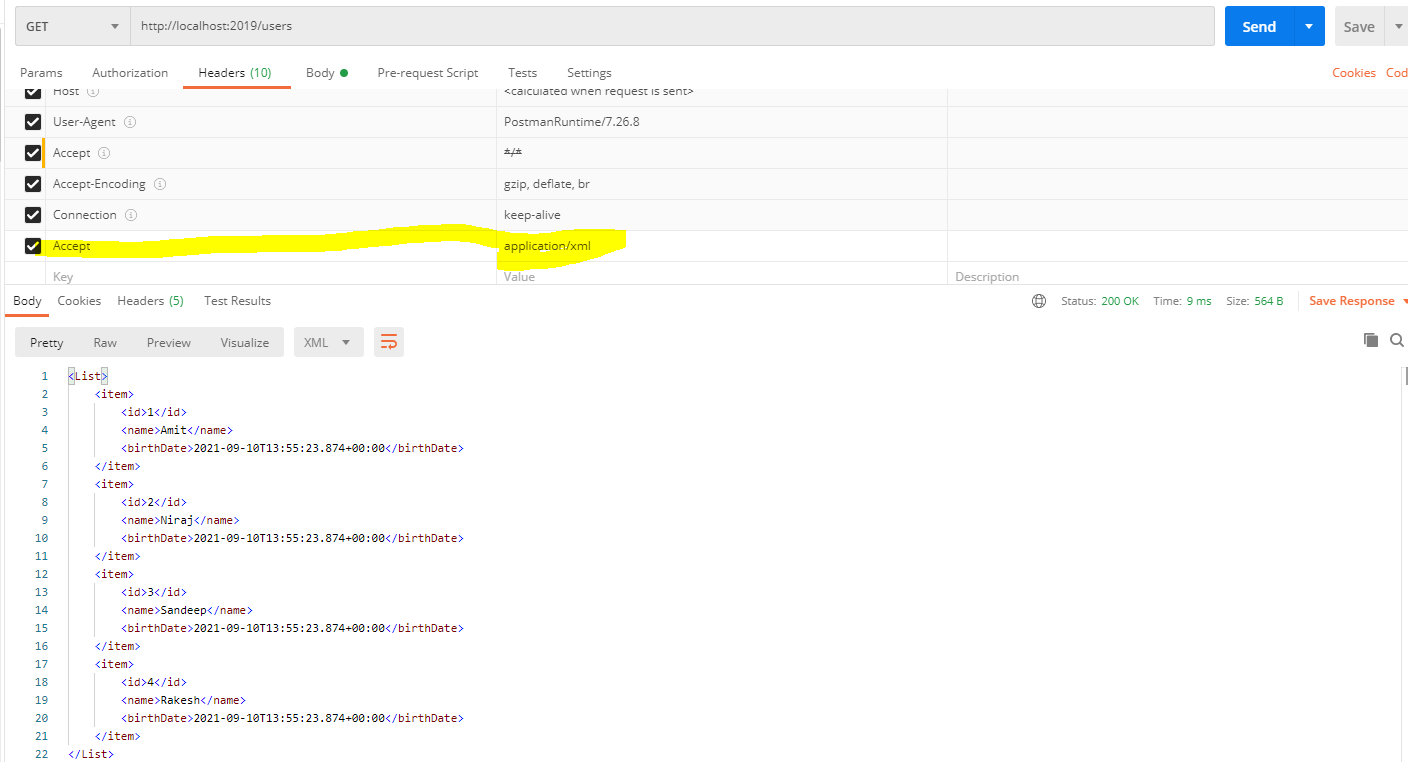
<groupId>com.fasterxml.jackson.dataformat</groupId>

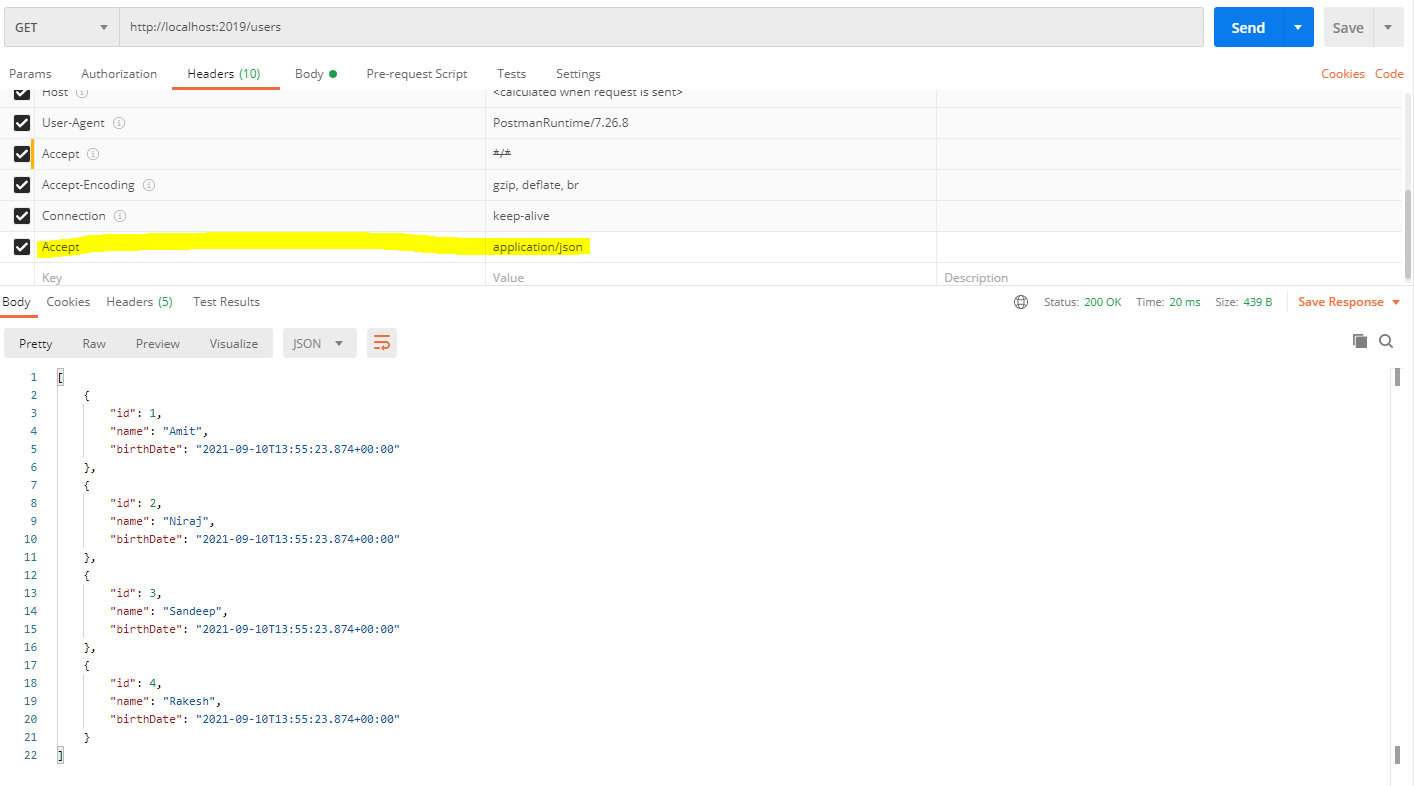
<artifactId>jackson-dataformat-xml</artifactId>

</dependency>

Getting the response in xml format







***Why Content Negotiation?***

Because there is a negotiation which is happening between the consumer and the provider of the service.

Here, The Consumer is saying.

I would want to accept application XML and you would send us an application XML back.

The Consumer might say, I want to accept application Json, what does the service provider do.

He provides the Json responds back, so there is negotiation which is happening between the consumer and the producer in terms of what representation to use, and that is why this is called as content negotiation.

***Step-20***

***Configuring Auto Generation of Swagger Documentation***

Let's say I would want my consumers to use the restfully Web services that we are creating, the typical

questions that we would expect from the consumer.

Hey, what is it we are told to use?

What is the format of the request?

What is the format of the response?

Are there any validations?

Should we adhere to any constraints?

So, the consumers would want to know how to use the apps or the services that we are creating.

Now, how can we provide that information to your consumers?

That's where you need to document your API.

One option we have is to manually document your first API.

We can have a word document or HTML page where we keep the documentation some the location.

We can say that you are can take this documentation from given location.

And in the response, we can get its name and birth date.

Name can only have two characters, but it should always be in the past.

So, we can have a documentation explaining that.

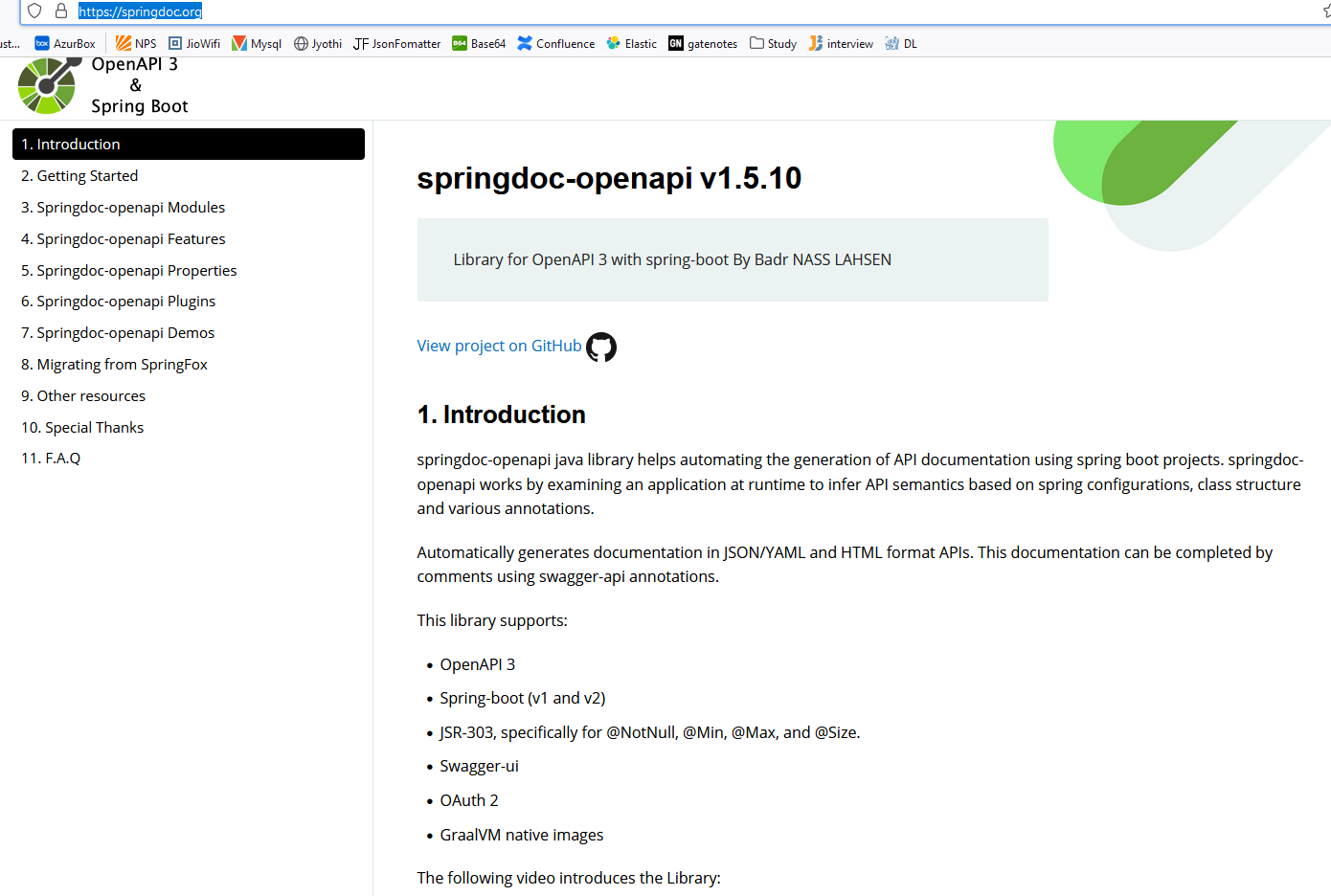
But wouldn't it be great if we can automatically generate the documentation from your code?

That's the option that we would be exploring right now.

If there is a standard format to define your rest API, this standard is called open API specification.

The framework that we would be using to generate the documentation is something called Spring Dock Open.

[**https://springdoc.org/**](https://springdoc.org/)



Add the below dependency in the pom.xml

<dependency>

<groupId>org.springdoc</groupId>

<artifactId>springdoc-openapi-ui</artifactId>

<version>1.5.10</version>

</dependency>

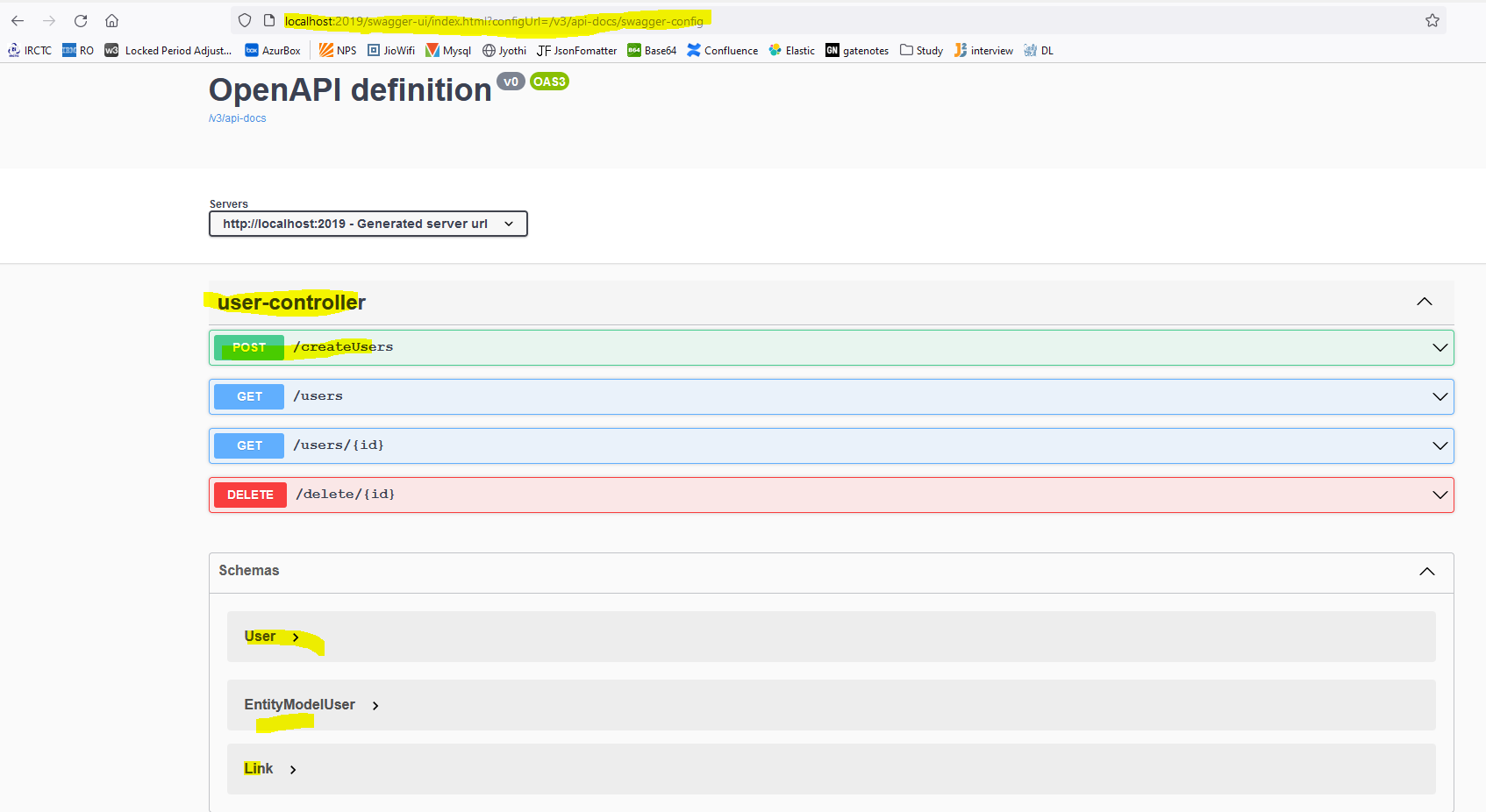
Save it and stop the server and start it again.

Use below url to access the documentation

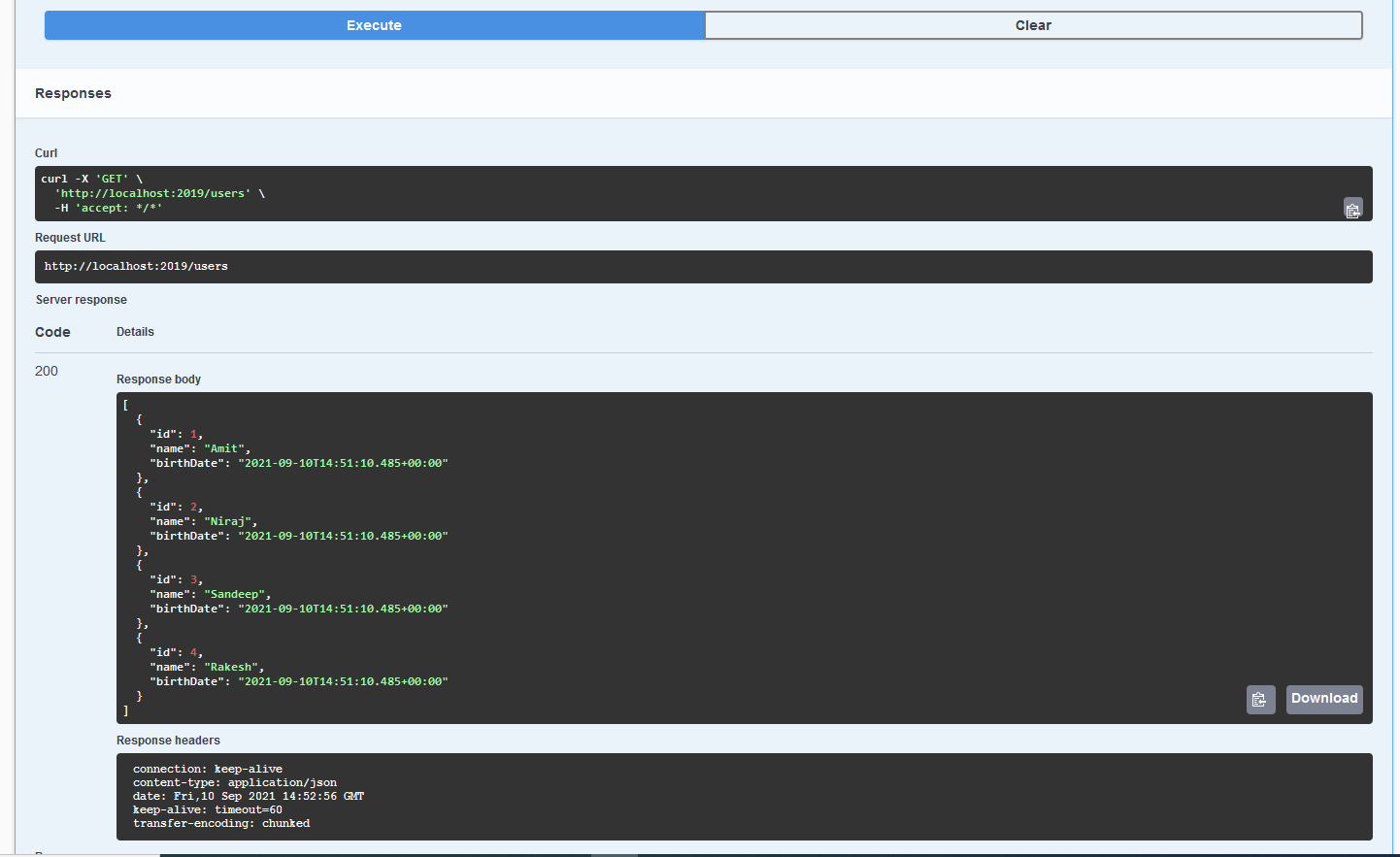
**springdoc.swagger-ui.path=/swagger-ui.html**

[**http://localhost:2019/swagger-ui.html**](http://localhost:2019/swagger-ui.html)

**After hitting the above URL, it will give the below screen**



We can execute the respective API from the above documentation page



It will also show the schema of the classes



***Step-21 Introduction of Swagger Documentation format***

what is open API, what is the difference between them?

This specification was earlier called swagger Specification.

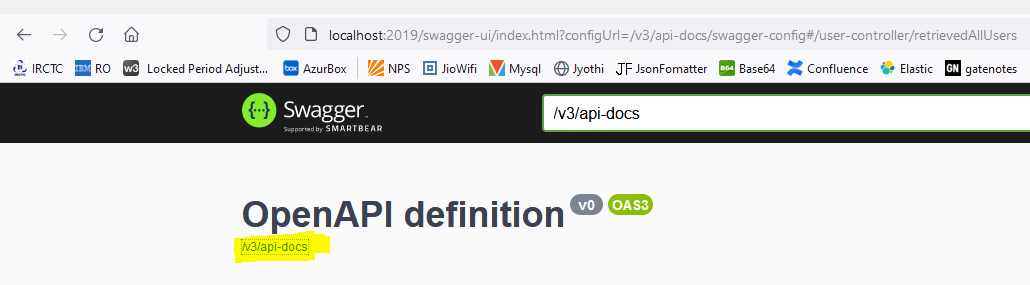
Swagger was initially created in 2011, and when Swagger was initially created, swagger provided

the definition, which is swagger specification, and it also provided you with tools to play with the

specification so you can use to evaluate and explore the API documentation, which is created.

In the next few years after 2011, swagger became very, very popular and in 2016, swagger specification

was open sourced and it was renamed as open API specification.



***Once we click on***

***/w3/api-docs will get the more details***



***To get the more details click on arrowhead showing above.***

***Step-22 Monitoring API with Spring boot Actuator***

Spring Boot includes several additional features to help you monitor and manage your application when you push it to production.

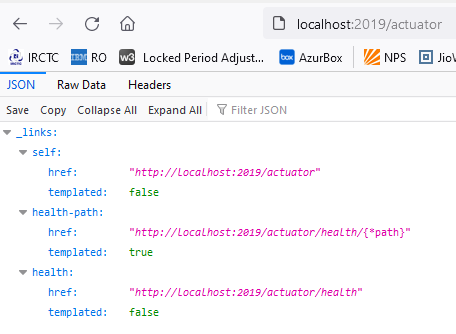
Whenever we put our application in production, monitoring the application becomes very, very important.

We want to look at the different metrics which are behind the application.

We would want to monitor the health of the application that what Actuator provides you with by default.

We would want to be able to see the health of the application that we're actuator comes into picture,

When we launch actuator, we see that only three wells are being exposed.

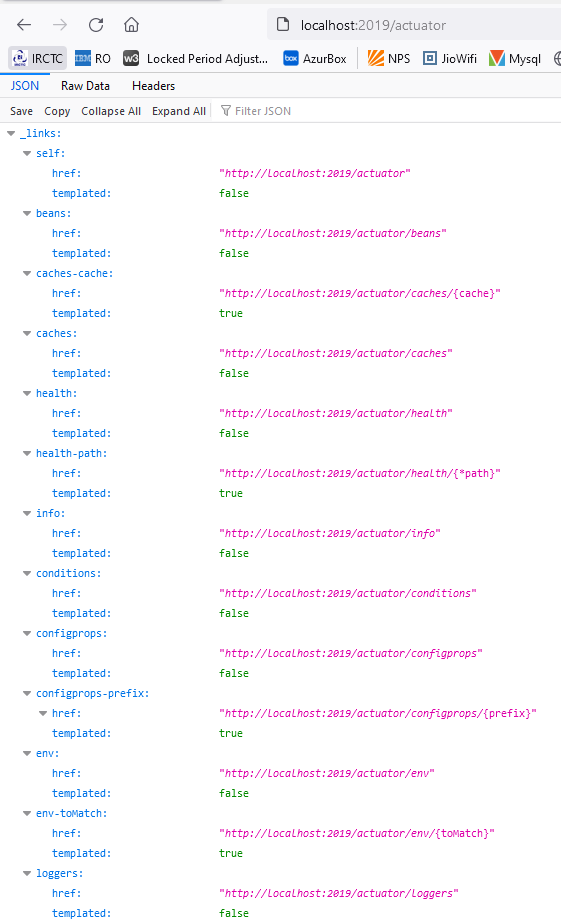


**Get more details for Actuator, click below URL**

[**https://docs.spring.io/spring-boot/docs/current/reference/html/actuator.html**](https://docs.spring.io/spring-boot/docs/current/reference/html/actuator.html)

management.endpoints.web.exposure.include=\*

Add above in the application.property file for enable everything.



**Click on each of the URL and get the more details about the respective end points.**

***Step-23 Visualizing API with HAL explorer***

HAL Explorer provides a visualization of all that information.

It shows all the links in a visual format.

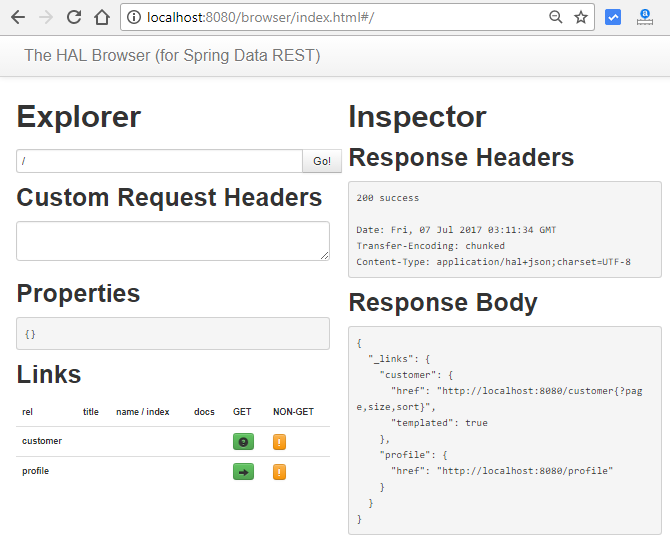
And we can also see the response status and the response headers when we execute that API.

So, HAL Explorer helps us to explore the space in an easy way.

**HAL Browser** is a web application which we can use by adding a dependency: spring-data-rest-hal-browser.

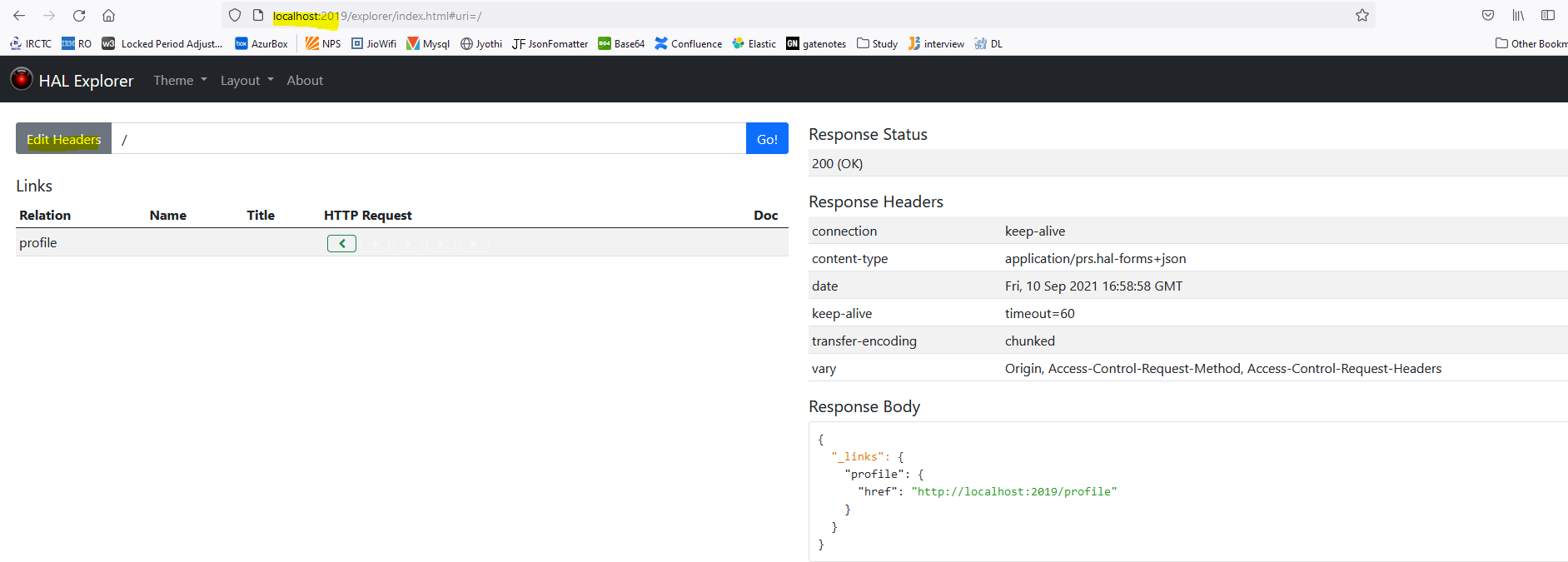
The dependency will autoconfigure the **HAL Browser** to be served up. So when we visit the uri http://localhost:8080 , we got a first screen shot:

<https://grokonez.com/spring-framework/spring-data/use-hal-browser-spring-data-rest-springboot-mysql>

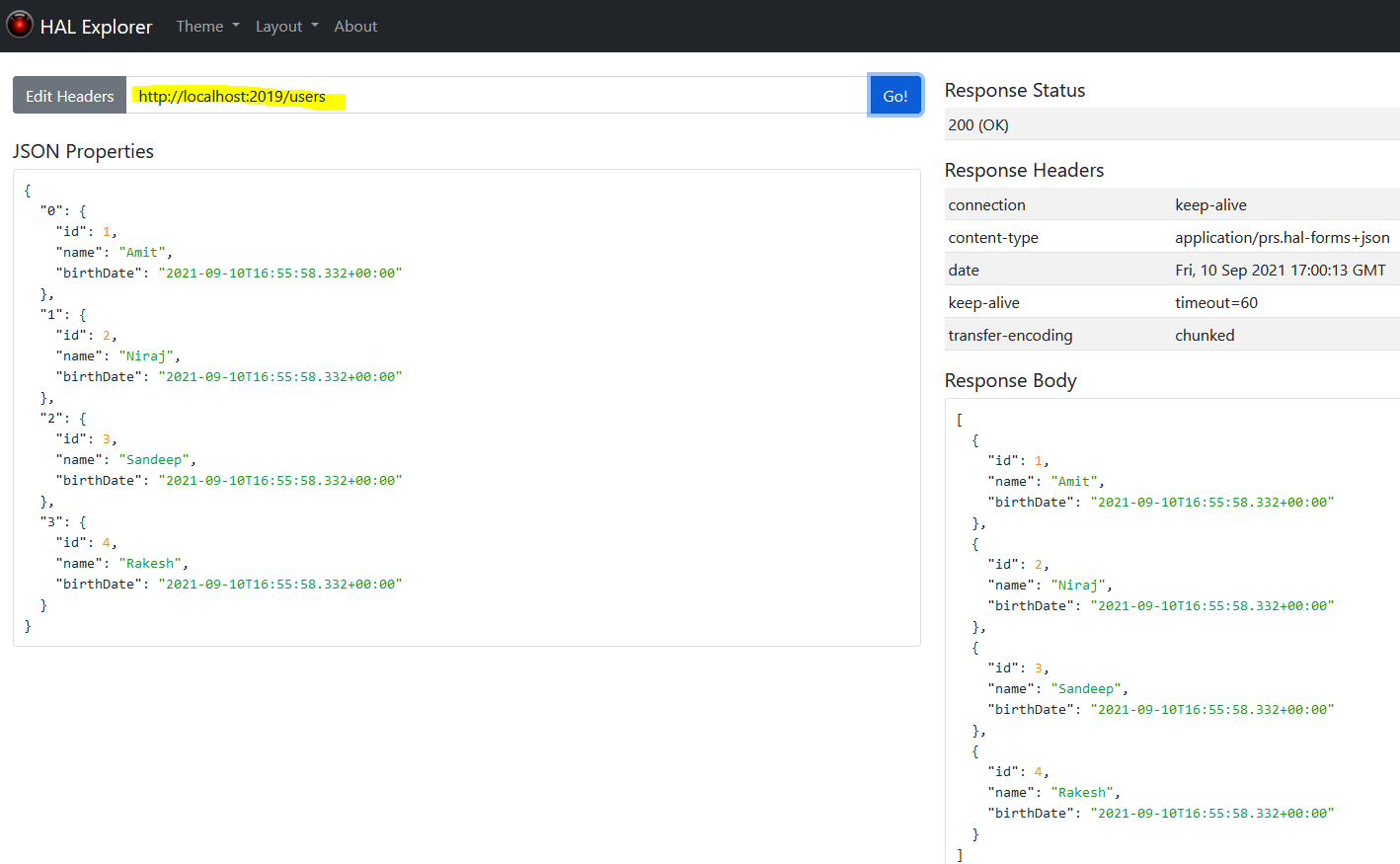


[**http://localhost:2019**](http://localhost:2019)

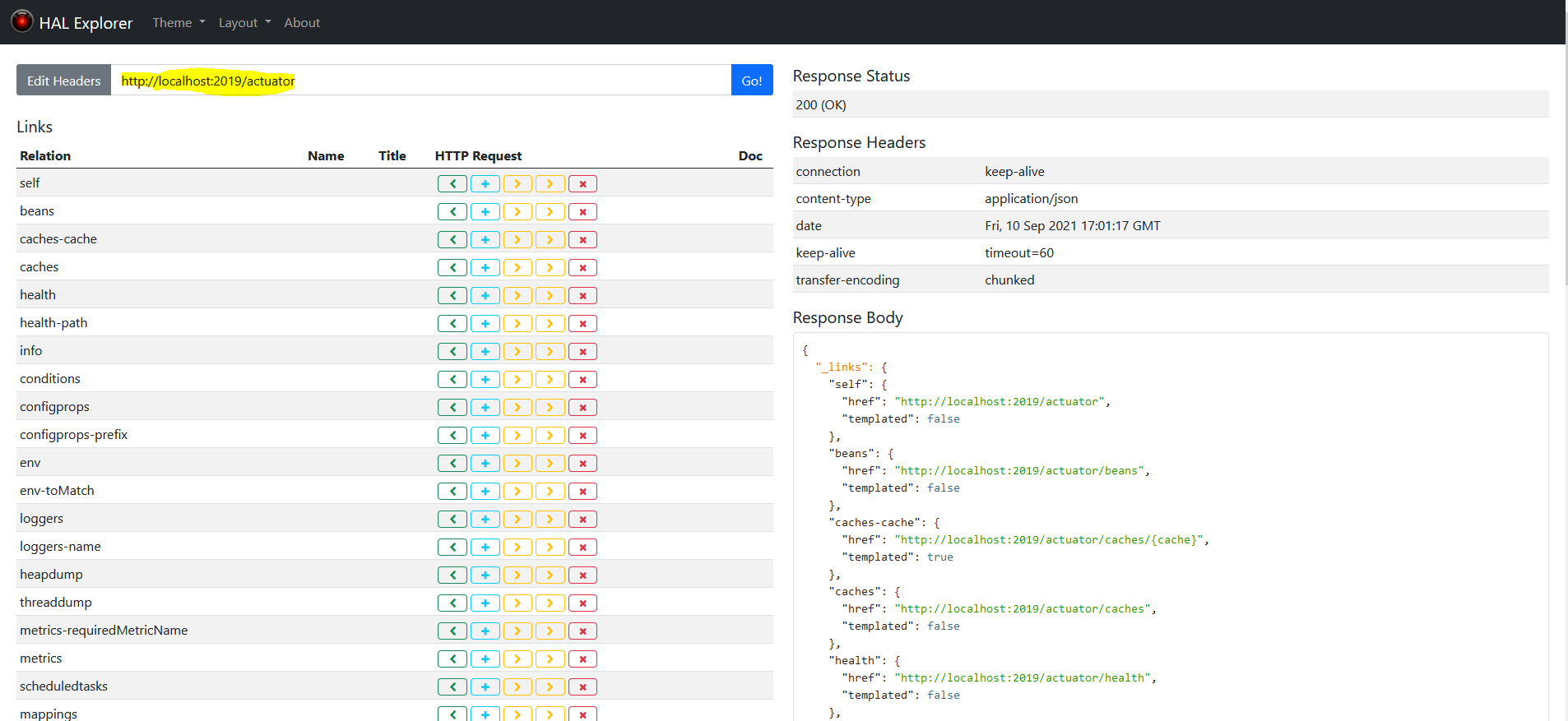
**Once we hit the above URL will get the below screen**



[**http://localhost:2019/users**](http://localhost:2019/users)



[**http://localhost:2019/actuator**](http://localhost:2019/actuator)



***Step-24 Implementing static filtering for RESTful services***

**I'm sending a GET request to our users resource and I'm getting this response back.**

**Id, name, birthdate. Id, name, birthdate.**

**Let's say, I don't want birthday to come in my response. How do I do that?**

**That concept is called filtering.**

**@RestController**

**public class FilteringController {**

**@GetMapping("/filtering")**

**public SomeBean retrieveSome() {**

**return new SomeBean("value1","value2","value3");**

**}**

**}**

**public** **class** SomeBean {

**private** String field1;

**private** String field2;

**private** String field3;

**public** SomeBean(String field1, String field2, String field3) {

**super**();

**this**.field1 = field1;

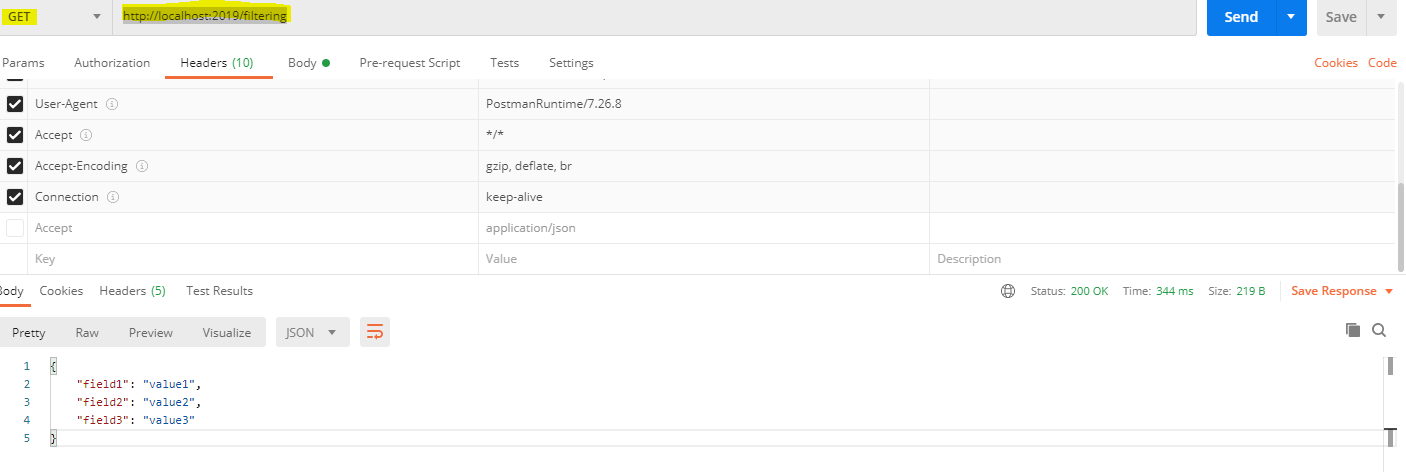
**this**.field2 = field2;

**this**.field3 = field3;

}// develop the setter and getter method

}

<http://localhost:2019/filtering>



**If we want to remove field3 from the response, then how we can achieve it?**

**public** **class** SomeBean {

**private** String field1;

**private** String field2;

@JsonIgnore

**private** String field3;

**public** SomeBean(String field1, String field2, String field3) {

**super**();

**this**.field1 = field1;

**this**.field2 = field2;

**this**.field3 = field3;

}

// develop the setter and getter method

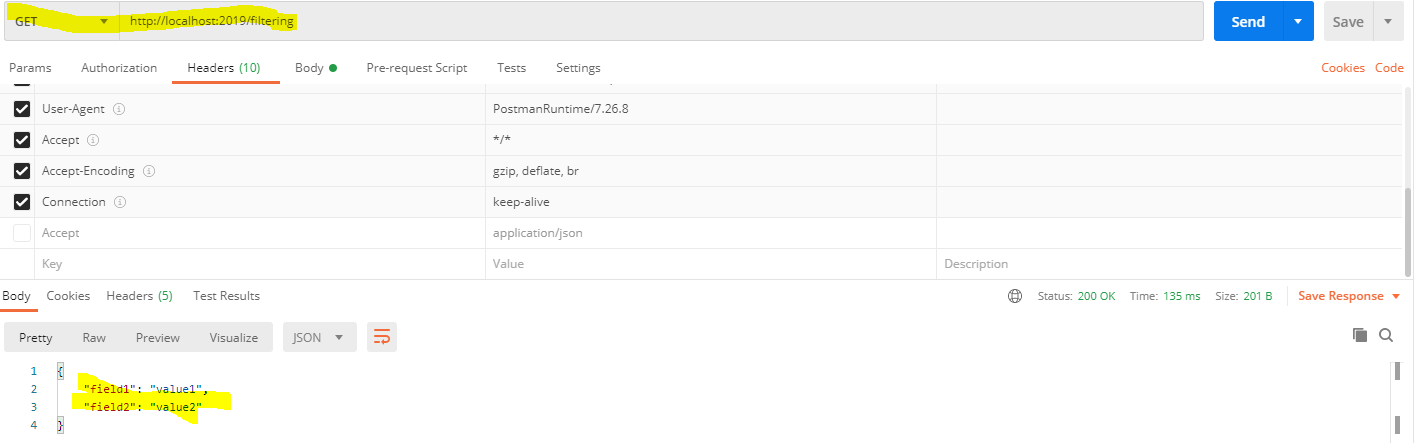
}

// field3 is removed from the response

So static filtering I mean a field I don't want it to ever come in my response at all.

So, I can use static filtering annotations like JSON ignore or I can define the JSON ignore

properties on some bean, I’d be able to make sure that this field never comes in my response.



This is the best practice to filter some field from the response.

Whatever sensitive data are there we can filter it out – means we don’t want to send those data as response.

We can achieve it in another way, at the class level

@JsonIgnoreProperties(value = {"field2", "field3"})

**public** **class** StaticFilter {

**private** String field1;

//@JsonIgnore

**private** String field2;

//@JsonIgnore

**private** String field3;

**}**

It will filter from the list also.

@GetMapping("/filtering-list")

**public** List<StaticFilter> retrieveSomeList() {

List<StaticFilter> someBeanList = **new** ArrayList<StaticFilter>();

someBeanList.add(**new** StaticFilter("value1","value2","value3"));

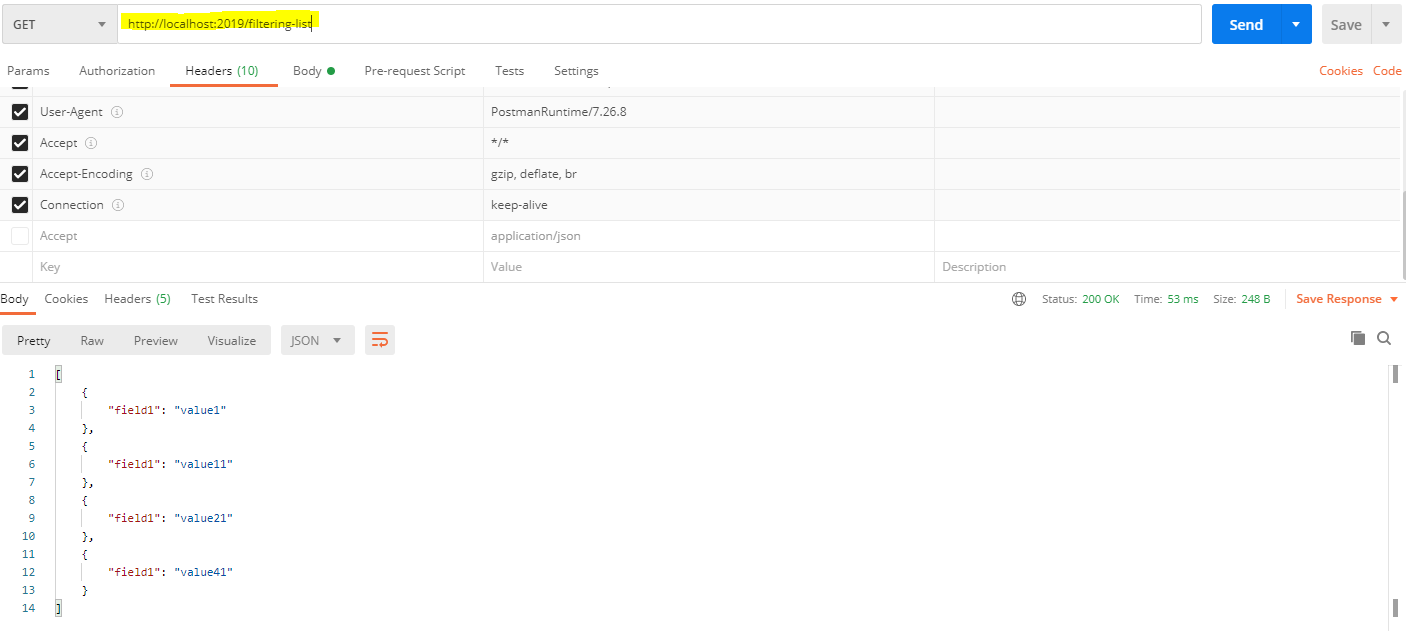
someBeanList.add(**new** StaticFilter("value11","value22","value3"));

someBeanList.add(**new** StaticFilter("value21","value2","value33"));

someBeanList.add(**new** StaticFilter("value41","value2","value3"));

**return** someBeanList;

**}**



***Step-25 Dynamic Filtering for RESTful Service***

***11/9/2021***

**Dynamic filtering.**

Sometimes for some requests I would want to send these two fields alone and for

other requests I would want to send these two fields.

So, in specific requests I don’t want to send one. In specific request I don’t want to send field three.

**How do we be able to do that?**

That's basically where the concept of dynamic filtering comes. With dynamic filtering

we cannot configure filtering directly on the bean.

we need to start configuring the filtering where they’re retrieving the values.

**@JsonFilter("Somebean")**

**public class DynamicFilterBean {**

**private String field1;**

**private String field2;**

**private String field3;**

**public DynamicFilterBean(String field1, String field2, String field3) {**

**super();**

**this.field1 = field1;**

**this.field2 = field2;**

**this.field3 = field3;**

**}**

**}**

**@RestController**

**public class DynamicFilteringController {**

**@GetMapping("/dyna-filtering")**

**public MappingJacksonValue retrieveSomeBean() {**

***DynamicFilterBean dynamicFilterBean = new DynamicFilterBean("value1", "value2", "value3");***

***SimpleBeanPropertyFilter filter = SimpleBeanPropertyFilter.filterOutAllExcept("field1","field2");***

***FilterProvider filterProvider = new SimpleFilterProvider().addFilter("Somebean", filter);***

***MappingJacksonValue mapping = new MappingJacksonValue(dynamicFilterBean);***

***mapping.setFilters(filterProvider);***

**return mapping;**

**}**

**@GetMapping("/dyna-filtering-list")**

**public MappingJacksonValue retrieveSomeList() {**

**List<DynamicFilterBean> dynamicBeanList = new ArrayList<DynamicFilterBean>();**

**dynamicBeanList.add(new DynamicFilterBean("value1", "value2", "value3"));**

**dynamicBeanList.add(new DynamicFilterBean("value11", "value22", "value3"));**

**dynamicBeanList.add(new DynamicFilterBean("value21", "value2", "value33"));**

**dynamicBeanList.add(new DynamicFilterBean("value41", "value2", "value3"));**

**SimpleBeanPropertyFilter filter = SimpleBeanPropertyFilter.*filterOutAllExcept*("field1");**

**FilterProvider filterProvider = new SimpleFilterProvider().addFilter("Somebean", filter);**

**MappingJacksonValue mapping = new MappingJacksonValue(dynamicBeanList);**

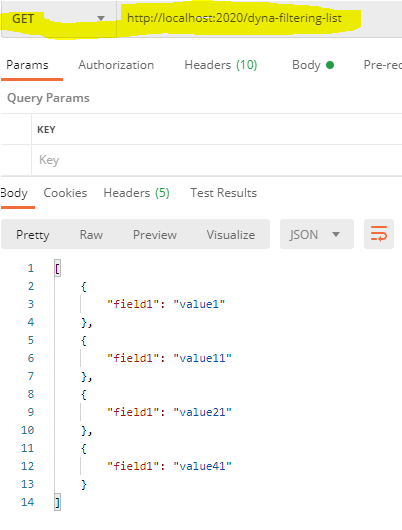
**mapping.setFilters(filterProvider);**

**return mapping;**

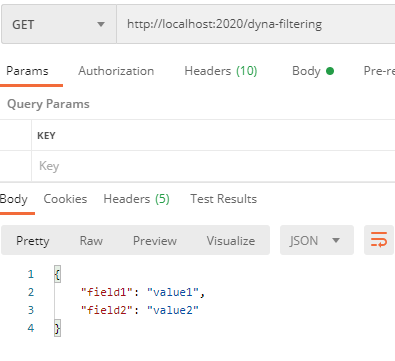
**}**

**}**

[**http://localhost:2020/dyna-filtering-list**](http://localhost:2020/dyna-filtering-list)



[**http://localhost:2020/dyna-filtering**](http://localhost:2020/dyna-filtering)



***Step-26 Versioning RESTful service – Basic Approach with URIs***

Here we understand why versioning is needed and how we can version our REST API.

Versioning is the most important and difficult part of the API as it takes backward API compatible.

Versioning helps us to iterate faster when the changes are identified. We should always version our Web API.

Consider a scenario in which we have a Web API that is up (status) and running.

The users are consuming that API. Now we want to add more functionality in the Web API but want to keep the existing functionality unchanged.

There may be few users who still want to use the old API while the other users want a new version of API with new or extended features.

It is the scenario where Web API versioning comes into existence.

**When we required versioning:**

When we made a **breaking change** in Web API, we should up versioned the API. Breaking changes includes:

* A change in the format of the response data for one or more calls.
* Change in the response type.
* Remove any part of the API.

Breaking changes should always result in a change to the major version number for an API or content response type.

Non-breaking changes (adding new points or new response parameters) do not require a change to the major version number. However, it can be helpful to track the minor version of the APIs.

**How to Version**

The most commonly used approaches fall into three categories:

* URI Versioning
* Versioning using Custom Request Header
* Versioning using Accept Header

**Different Approaches to API Versioning**

There are some different ways to provide API versioning in your application. The most popular of them are:

1. Through a **URI path** – you include the version number in the URL path of the endpoint, for example, /api/v1/persons.
2. Through **query parameters** – you pass the version number as a query parameter with a specified name, for example, /api/persons?version=1.
3. Through **custom HTTP headers** – you define a new header that contains the version number in the request.
4. Through a **content negotiation** – the version number is included in the “Accept” header together with the accepted content type. The request with cURL would look like the following:

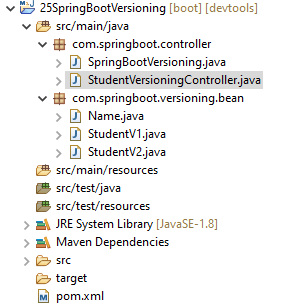
1

curl -H "Accept: application/vnd.piomin.v1+json" http://localhost:8080/api/persons

***Step-27 Versioning RESTful services- Header and Content Negotiation Approach***

Through a **URI path**

Basic approach to versioning is to create a completely different URI for the new service.



**@RestController**

**public class StudentVersioningController {**

**@GetMapping("v1/student")**

**public StudentV1 studentv1() {**

**return new StudentV1("Amit Kumar");**

**}**

**@GetMapping("v2/student")**

**public StudentV2 studentv2() {**

**return new StudentV2(new Name("Amit", "Kumar"));**

**}**

**}**

**public** **class** Name {

**private** String firstName;

**private** String lastName;

**public** Name(String firstName, String lastName) {

**super**();

**this**.firstName = firstName;

**this**.lastName = lastName;

}

}

**public** **class** StudentV1 {

**private** String name;

**public** StudentV1(String name) {

**super**();

**this**.name = name;

}

}

**public** **class** StudentV2 {

**private** Name name;

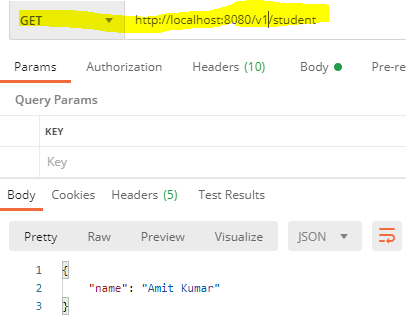
**public** StudentV2(Name name) {

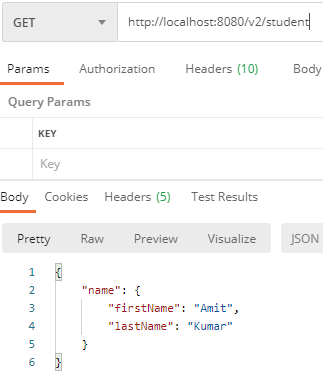
**super**();

**this**.name = name;

}

}





**Request Parameter versioning**

Next approach to versioning is to use the request parameter to differentiate versions.

Examples

* http://localhost:8080/student/param?version=1
* http://localhost:8080/student/param?version=2

Implementations are shown below:

//Request Parameter versioning start

@GetMapping(value = "/student/param", params = "version=1")

**public** StudentV1 paramV1() {

**return** **new** StudentV1("Amit Kumar");

}

@GetMapping(value = "/student/param", params = "version=2")

**public** StudentV2 paramV2() {

**return** **new** StudentV2(**new** Name("Amit", "Kumar"));

}

<http://localhost:8080/student/param?version=1>

Response

{

    "name": "Amit Kumar"

}

<http://localhost:8080/student/param?version=2>

Response

{

    "name": {

        "firstName": "Amit",

        "lastName": "Kumar"

    }

}

**(Custom) Headers versioning**

@GetMapping(value = "/student/header", headers = "X-API-VERSION=1")

**public** StudentV1 headerV1() {

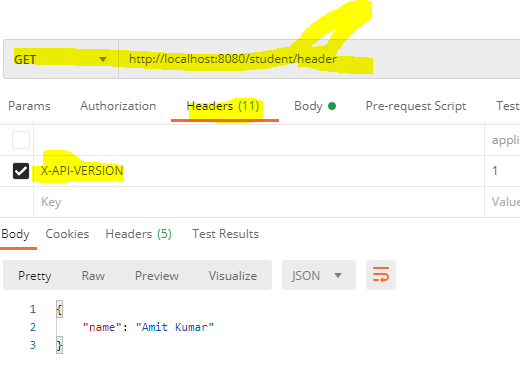
**return** **new** StudentV1("Amit Kumar");

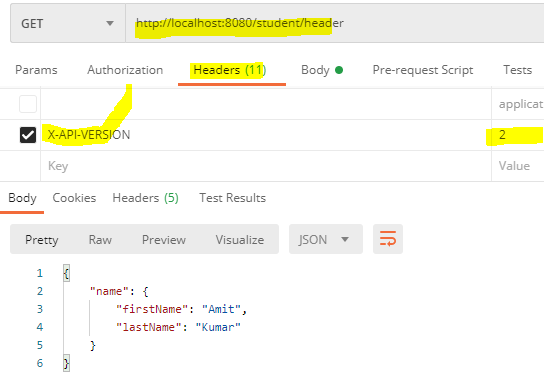
}

@GetMapping(value = "/student/header", headers = "X-API-VERSION=2")

**public** StudentV2 headerV2() {

**return** **new** StudentV2(**new** Name("Amit", "Kumar"));}





**Media type versioning (a.k.a “content negotiation” or “accept header”)**

The last versioning approach is to use the Accept Header in the request.

**@GetMapping(value = "/student/produces", produces = "application/vnd.company.app-v1+json")**

**public StudentV1 producesV1() {**

**return new StudentV1("Amit Kumar");**

**}**

**@GetMapping(value = "/student/produces", produces = "application/vnd.company.app-v2+json")**

**public StudentV2 producesV2() {**

**return new StudentV2(new Name("Amit", "Kumar"));**

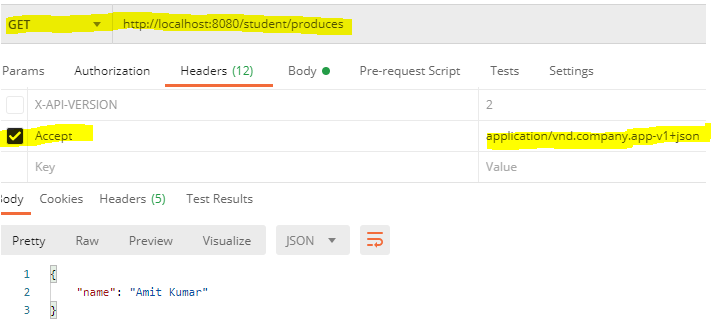
**}**

[**http://localhost:8080/student/produces**](http://localhost:8080/student/produces)

headers [Accept=application/vnd.company.app-v1+json]

[**http://localhost:8080/student/produces**](http://localhost:8080/student/produces)

headers [Accept=application/vnd.company.app-v2+json]



**The list below shows Major API providers using different versioning approaches.**

* **Media type versioning (a.k.a “content negotiation” or “accept header”)** 
  + **GitHub**
* **(Custom) Header’s versioning** 
  + **Microsoft**
* **URI Versioning** 
  + **Twitter**
* **Request Parameter versioning** 
  + **Amazon**

**Plan to avoid versioning as far as possible but evaluate and be ready with a versioning strategy before you expose your first service to your consumer. Good Luck!**

**Factors affecting Versioning Choice**

Following factors affect the choice of versioning:

* URI Pollution - URL versions and Request Param versioning pollute the URI space.
* Misuse of HTTP Headers - Accept Header is not designed to be used for versioning.
* Caching - If you use Header based versioning, we cannot cache just based on the URL. You would need take the specific header into consideration.
* Can we execute the request on the browser? - If you have non-technical consumers, then the URL based version would be easier to use as they can be executed directly on the browser.
* API Documentation - How do you get your documentation generation to understand that two different URLs are versions of the same service?

***Step-28 Implementing Basic Authentication with Spring security***

we have created a-lots of resources, but none of them is secure yet.

There is no user id and password to access the resources.

There are multiple ways to authenticate our RESTful web services.

The basic way is to use basic authentication. In the basic authentication, we send a username and password as part of our request. When we provide a username and password, it allows us to access the resource.

We have to add security dependency in pom.xml

<dependency>

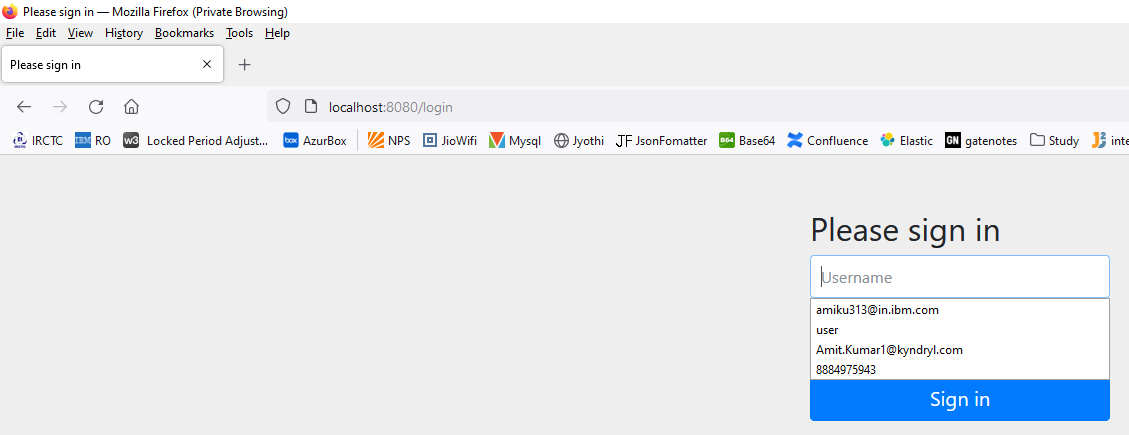
<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

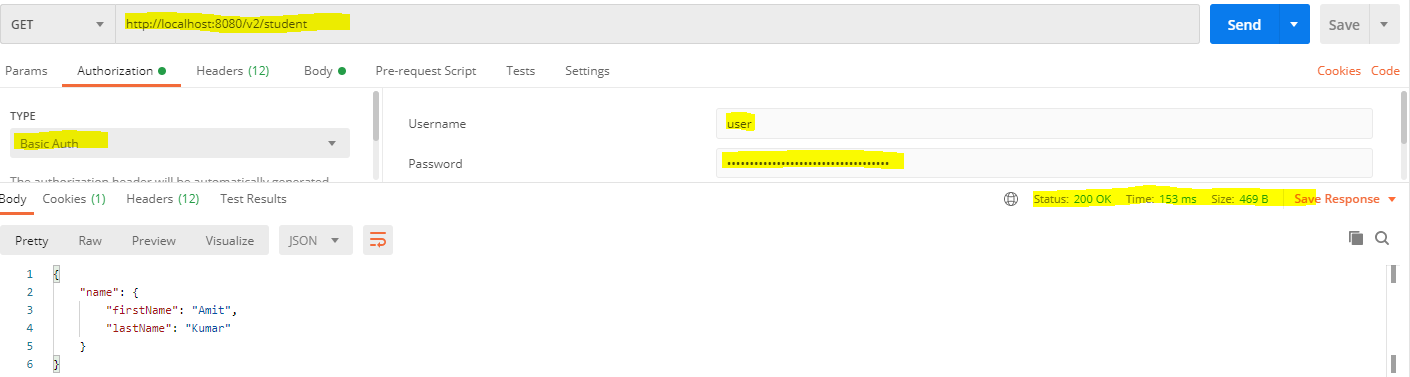
Restart the server and try to access the url

It will ask to sign in if we are tying to access it from browser

Default username: user

And password – check in the log

We can also keep our desire username and password by putting in the property file.



There is a disadvantage that when we restart the server, the password changes again and again.

The solution to this problem is that configure the username and password in the application.properties file.

spring.security.user.name=user

spring.security.user.password=password

[***https://www.techgeeknext.com/spring-boot-security/digest\_authentication\_web\_security***](https://www.techgeeknext.com/spring-boot-security/digest_authentication_web_security)

***Step-29 Overview of Connecting RESTful Service to JPA***

**Brief About the JPA.**

***H2 Database URL & Data.sql***

With the latest versions of Spring Boot

1. H2 database name is randomly generated each time we restart the server.
2. We can find the database name and URL from the console log.
3. To use data.sql, we need to add this to

application.properties -  spring.jpa.defer-datasource-initialization=true

**RECOMMENDED**:

Configure this in application.properties:

1. spring.datasource.url=jdbc:h2:mem:testdb
2. spring.jpa.defer-datasource-initialization=true

**DEBUGGING GUIDE (If you have problems)**

JPA Hibernate Debugging Guide:

<https://github.com/in28minutes/in28minutes-initiatives/blob/master/The-in28Minutes-TroubleshootingGuide-And-FAQ/jpa-and-hibernate.md>

***Two Important Things to Remember***

#### **UPDATE 1: SWAGGER**

With the recent update (2.5.0) to use springdoc-openapi-ui,

* You do not need ApiModel and ApiModelProperty annotations (needed with older Swagger frameworks)
* Do NOT worry if you see these annotations in User bean (Ignore them)

1. //IGNORE THESE ANNOTATIONS
2. @ApiModel(description="All details about the user.")
3. @ApiModelProperty(notes="Name should have atleast 2 characters")
4. @ApiModelProperty(notes="Birth date should be in the past")

***UPDATE 2: HATEOAS***

HATEOAS code with earlier versions looks something like this: Resource and ControllerLinkBuilder instead of EntityModel and WebMvcLinkBuilder.

1. //Do not worry about Resource and ControllerLinkBuilder
2. Resource<User> resource = new Resource<User>(user);
3. ControllerLinkBuilder linkTo =
4. linkTo(methodOn(this.getClass()).retrieveAllUsers());

***Step-30 Creating User Entity and some test data***

***Step-31***

***Step-32***

***Step-33***

***Step-34***

***Step-35***

***Step-36***

***Step-37***