

RESTful Webservice

All codes

<https://github.com/in28minutes/spring-microservices/tree/master/02.restful-web-services>

Configure your project from link - <https://start.spring.io/>

Preparation Link- <https://github.com/in28minutes?tab=repositories>

Now let's create a simple hello world rest project

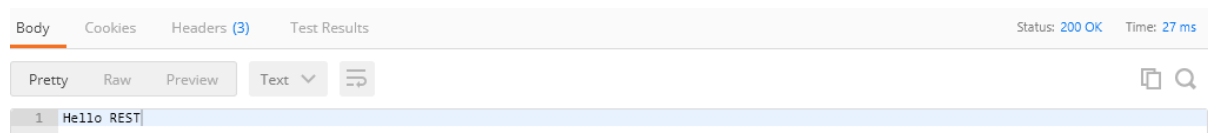
To create this we need three things – a method , type of request and an URI

```
package com.wipro.rest.restwebservice;
```

```
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.RestController;
```

```
@RestController  
public class HelloWorldController {  
  
    @GetMapping(path="/hello-rest")  
    public String helloWorld()  
    {  
        return "Hello REST";  
    }  
  
}
```

Output- Status-200 OK

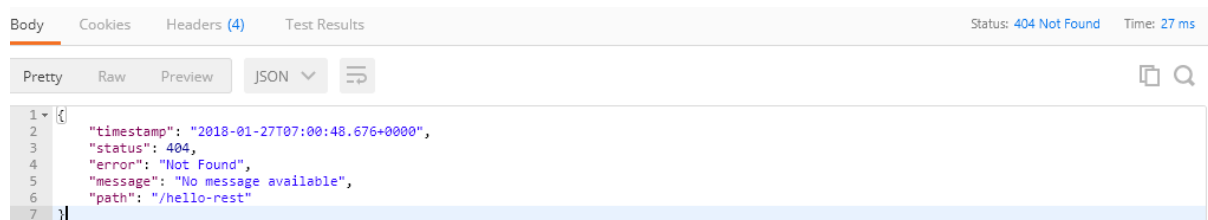


RestController- It handles REST request

Controller- It handles HTTP request

What will happen if we use Controller instead of RestController

In this case HelloWorldController class cannot handle REST request and 404 not found exception is thrown



Instead of `@GetMapping(path="/hello-rest")` we can use `@RequestMapping(method=RequestMethod.GET, path="/hello-rest")`
But it is not recommended.

Let's modify the example and create a HelloRestBean class to return the message.

HelloRestBean

```
package com.wipro.rest.restwebservice;

public class HelloRestBean {
    String message;

    public HelloRestBean(String message) {
        this.message=message;
    }

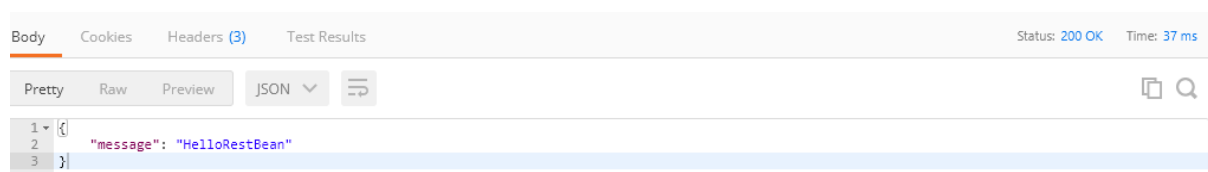
    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
    @Override
    public String toString()
    {
        return String.format("HelloRestBean [message=%s]", message);
    }
}
```

Resource-

```
@GetMapping(path="/hello-rest-bean")
public HelloRestBean helloRest()
{
    return new HelloRestBean("HelloRestBean");
}
```

Output-



Note- If we forgot to create getters then automatic conversion (Bean to JSON) will not work and you will get following error.(status 500- internal server error)



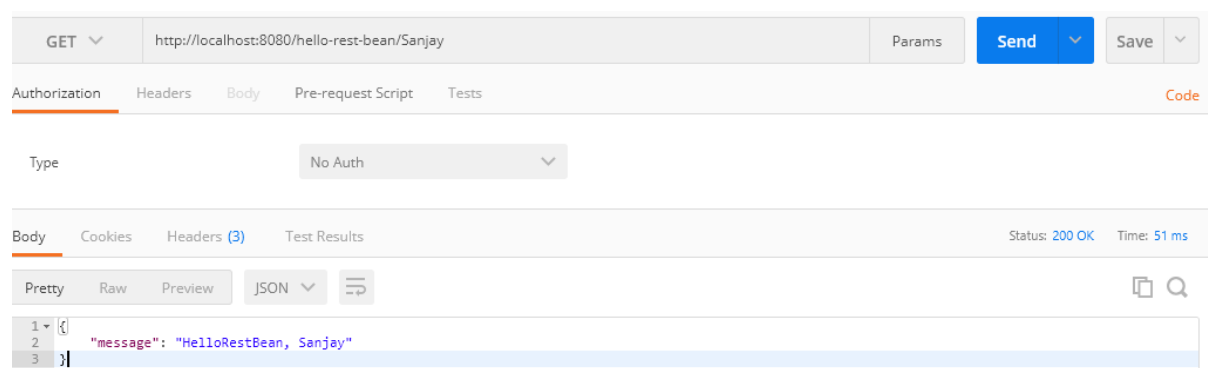
Spring boot auto configuration configures scans all classes/jars in classpath and auto configures them like Dispatcher Servlet, Default error page and http message converter Jackson.

Path Variable

```
@GetMapping(path="/hello-rest-bean/{name}")
public HelloRestBean helloRestPathVariable(@PathVariable String name)
{
    return new HelloRestBean(String.format("HelloRestBean, %s",name));
}
```

Whatever we will pass in GET request , will be appended in path variable.

Example- <http://localhost:8080/hello-rest-bean/Sanjay>



The screenshot shows a REST client interface. The top bar has a dropdown menu set to 'GET' and a text input field containing the URL 'http://localhost:8080/hello-rest-bean/Sanjay'. To the right of the URL are buttons for 'Params', 'Send', and 'Save'. Below the top bar are tabs for 'Authorization', 'Headers', 'Body', 'Pre-request Script', and 'Tests'. The 'Body' tab is selected, and it shows a 'Type' dropdown set to 'No Auth'. Below the tabs are buttons for 'Pretty', 'Raw', 'Preview', and a 'JSON' dropdown. The main area displays the response body in JSON format:

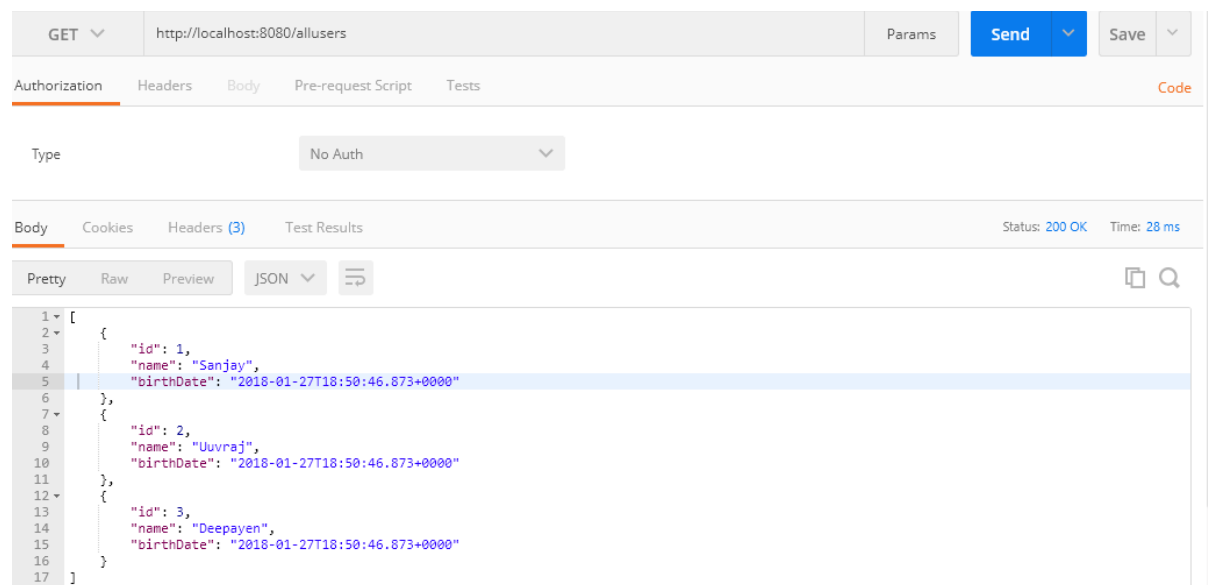
```
{
  "message": "HelloRestBean, Sanjay"
}
```

 The status bar at the bottom indicates 'Status: 200 OK' and 'Time: 51 ms'.

Now Let us create a social media application where we will be dealing with user details

To find all users

```
@GetMapping(path="/users")
public List<User> retrieveAllUsers()
{
    return service.findAll();
}
```



The screenshot shows a REST client interface. The top bar has a dropdown menu set to 'GET' and a text input field containing the URL 'http://localhost:8080/allusers'. To the right of the URL are buttons for 'Params', 'Send', and 'Save'. Below the top bar are tabs for 'Authorization', 'Headers', 'Body', 'Pre-request Script', and 'Tests'. The 'Body' tab is selected, and it shows a 'Type' dropdown set to 'No Auth'. Below the tabs are buttons for 'Pretty', 'Raw', 'Preview', and a 'JSON' dropdown. The main area displays the response body in JSON format:

```
[
  {
    "id": 1,
    "name": "Sanjay",
    "birthDate": "2018-01-27T18:50:46.873+0000"
  },
  {
    "id": 2,
    "name": "Uuvraj",
    "birthDate": "2018-01-27T18:50:46.873+0000"
  },
  {
    "id": 3,
    "name": "Deepayen",
    "birthDate": "2018-01-27T18:50:46.873+0000"
  }
]
```

 The status bar at the bottom indicates 'Status: 200 OK' and 'Time: 28 ms'.

To find a specific user

```
@GetMapping(path="/users/{id}")
public User retrieveUser(@PathVariable int id)
{
    User user=service.findOne(id);
    return user;
}
```

GET http://localhost:8080/users/2

authorization Headers Body Pre-request Script Tests

Type No Auth

body Cookies Headers (3) Test Results Status: 200 OK Time: 24 ms

Pretty Raw Preview JSON

```
1 {
2   "id": 2,
3   "name": "Uuvraj",
4   "birthDate": "2018-01-27T18:50:46.873+0000"
5 }
```

Post Mapping

```
@PostMapping("/users")
public ResponseEntity<User> createUser( @RequestBody User user)
{
    User savedUser=service.save(user);
    URI location = ServletUriComponentsBuilder.
        fromCurrentRequest().
        path("/{id}").
        buildAndExpand(savedUser.getId()).toUri();
    return ResponseEntity.created(location).build();
}
```

So now we will pass a post request

POST http://localhost:8080/users

authorization Headers (1) Body Pre-request Script Tests

form-data x-www-form-urlencoded raw binary JSON (application/json)

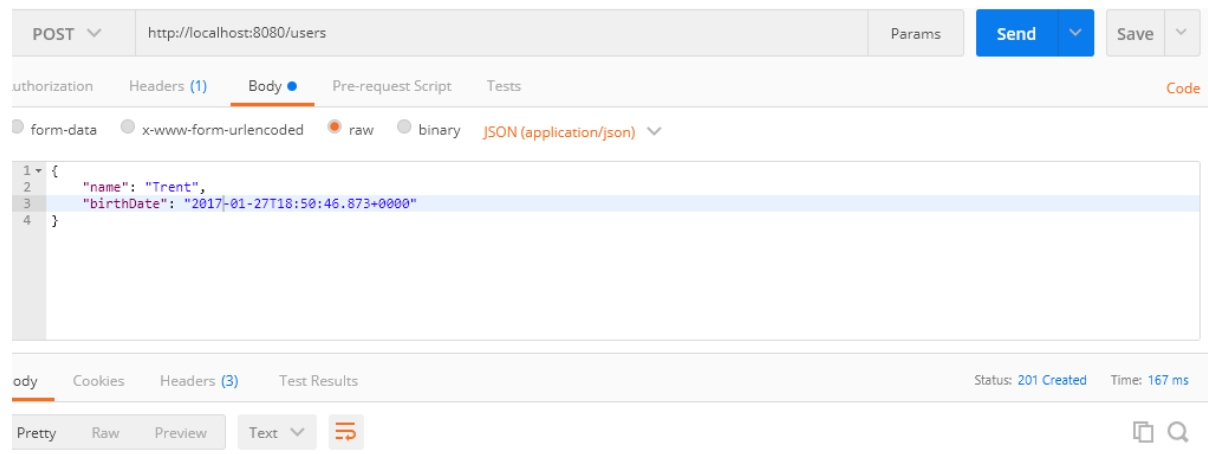
```
1 {
2   "name": "Trent",
3   "birthDate": "2017-01-27T18:50:46.873+0000"
4 }
```

body Cookies Headers (4) Test Results Status: 500 Internal Server Error Time: 303 ms

Pretty Raw Preview JSON

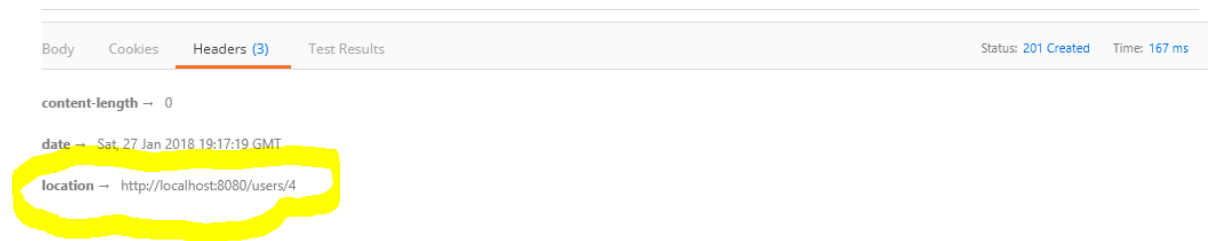
```
1 {
2   "timestamp": "2018-01-27T19:09:37.341+0000",
3   "status": 500,
4   "error": "Internal Server Error",
5   "message": "Type definition error: [simple type, class com.wipro.rest.restwebservice.users.User]; nested exception is com.fasterxml.jackson.databind.exc.InvalidDefinitionException: Cannot construct instance of `com.wipro.rest.restwebservice.users.User` (no Creators, like default construct, exist): cannot deserialize from Object value (no delegate- or property-based Creator)\n at [Source: (PushbackInputStream); line: 2, column: 5]",
6   "path": "/users"
7 }
```

We got internal server error because default constructor of user class is overridden with parametrized constructor. Let's create it and repeat the same.



This post request sends status 201 created with uri of the user created.

Check header of the response

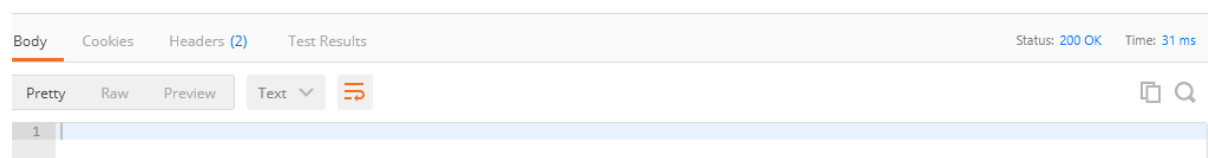


Please Note-

- Don't forgot Autowiring
- SpringBootApplication must be in parent package otherwise controllers will not be scanned and you will get 404 not found error.
- Must create default constructor in bean class.

Exception Handling

Suppose we are sending a http request for a user which does not exists. What we will get is a status 200 (OK) with empty body but it is not a good practice.



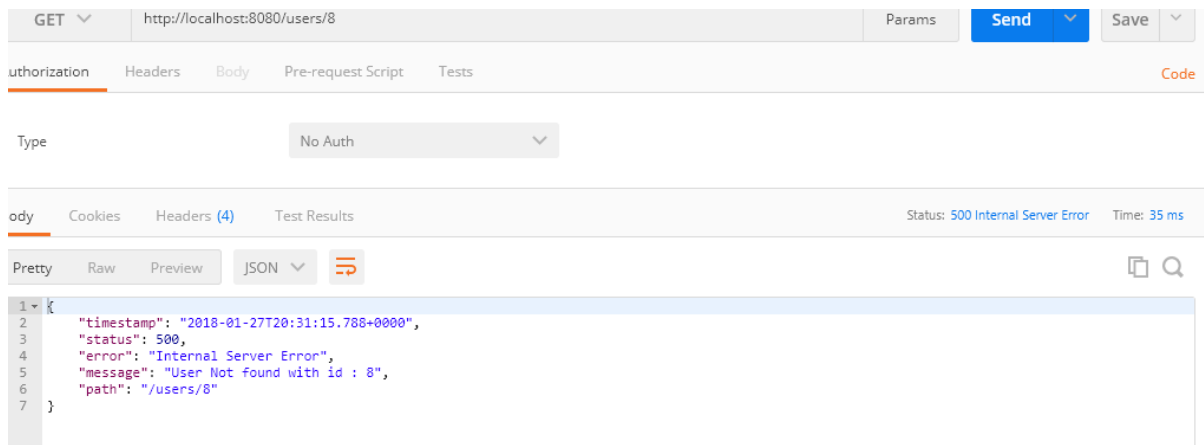
Let's fix it

```
@GetMapping(path="/users/{id}")
public User retrieveUser(@PathVariable int id)
{
    User user=service.findOne(id);
    if(user==null)
    {
        throw new UserNotFoundException("User Not found with id : "+id);
    }
}
```

```

    }
    return user;
}

```



So now it responding that it is internal server error with a meaningful message. But is it perfect? No

Because it is not really a server error. Problem is resource not found.

We need to make following changes in `UserNotFoundException` class.

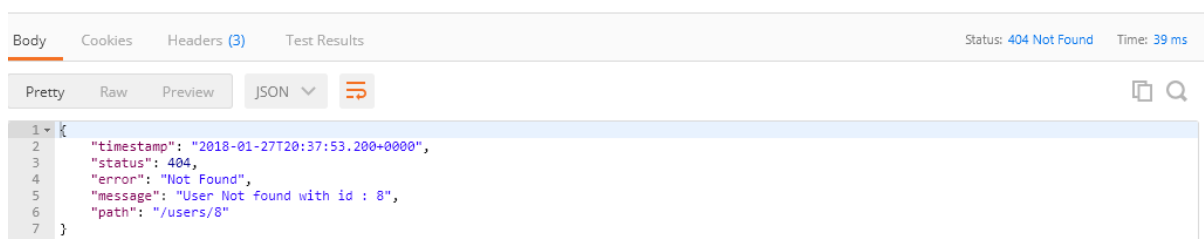
```

@ResponseStatus(HttpStatus.NOT_FOUND)//sends error code 404
public class UserNotFoundException extends RuntimeException {

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

```

And response now looks like-



It is cool now. The format in which we are getting error is by default configured by `SpringBootAutoConfiguration`. If we are working with a large organization then these error messages must be customized as per our organization. Let's do it.

Let's define a common structure for exceptions.

Create a `GeneralExceptionResponseBean` class which will provide the structure

```

package com.wipro.rest.restwebservice.exception;

```

```

import java.util.Date;

```

```

public class GeneralExceptionResponseBean {
    private Date timestamp;
    private String message;
    private String details;
}

```

```

    public GeneralExceptionResponseBean(Date timestamp, String message, String
details) {
        super();
        this.timestamp = timestamp;
        this.message = message;
        this.details = details;
    }
    public Date getTimestamp() {
        return timestamp;
    }
    public void setTimestamp(Date timestamp) {
        this.timestamp = timestamp;
    }
    public String getMessage() {
        return message;
    }
    public void setMessage(String message) {
        this.message = message;
    }
    public String getDetails() {
        return details;
    }
    public void setDetails(String details) {
        this.details = details;
    }
}

```

Now create a CustomizedResponseEntityExceptionHandler class to handle the exceptions globally.

```

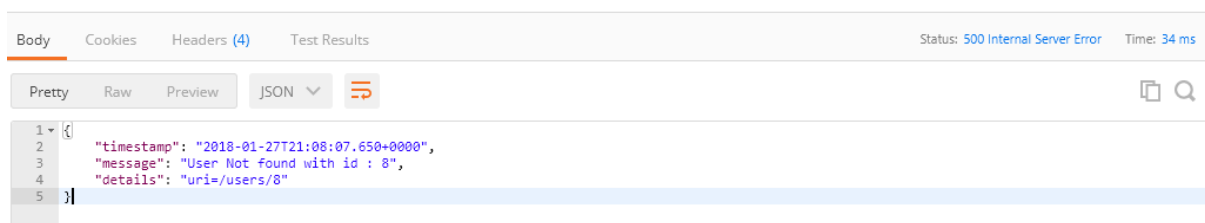
@ControllerAdvice // This means it will be available across all the controllers
@RestController
public class CustomizedResponseEntityExceptionHandler extends
ResponseEntityExceptionHandler {
    /* This is for all type of exceptions*/
    @ExceptionHandler(Exception.class)
    public final ResponseEntity<Object> handleAllException(Exception
ex, WebRequest request)
    {
        GeneralExceptionResponseBean er=new GeneralExceptionResponseBean(new
Date(),ex.getMessage(), request.getDescription(false));

        return new ResponseEntity(er,
HttpStatus.INTERNAL_SERVER_ERROR);
    }
}

```

Now fire the get request

<http://localhost:8080/users/8--User> with id 8 does not exists.



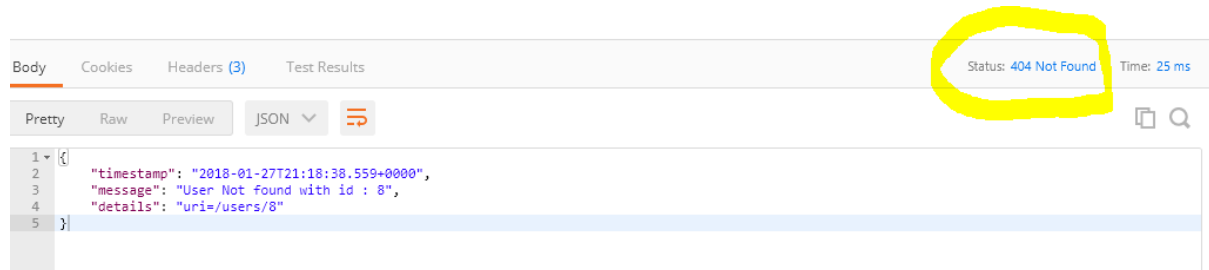
So we are getting error messages in customized format but again it is 500 internal server error as we have defined a general exception handler which is handling all the exceptions. Let's create a method which will handle `UserNotFoundException`.

Add a method in `CustomizedResponseEntityExceptionHandler` to handle this exception.

```
/*Create Specific exceptions*/
@ExceptionHandler(UserNotFoundException.class)
public final ResponseEntity<Object>
handleUserNotFoundException(UserNotFoundException ex, WebRequest request)
{
    GeneralExceptionResponseBean er = new GeneralExceptionResponseBean(new
Date(), ex.getMessage(), request.getDescription(false));
    return new ResponseEntity(er, HttpStatus.NOT_FOUND);
}
```

Now if you fire the same request- <http://localhost:8080/users/8>

We will get the response in customized format and it is 100% acceptable



Exercise-

Retrieve all posts for a User - GET `/users/{id}/posts`
Create a posts for a User - POST `/users/{id}/posts`
Retrieve details of a post - GET `/users/{id}/posts/{post_id}`

DELETE

In middle of the request we cannot delete an object use a temporary list or iterator.

```
@DeleteMapping("/users/{id}")
public ResponseEntity<String> deleteUser(@PathVariable int id)
{
    User user = service.deleteById(id);
    if (user == null)
    {
        throw new UserNotFoundException("User Not Found with id ->
"+id);
    }
    return ResponseEntity.ok("User has been deleted...");
}
```

Example of Output-

DELETE Params

Authorization Headers Body Pre-request Script Tests Code

Type

Body Cookies Headers (3) Test Results Status: 200 OK Time: 19 ms

Pretty Raw Preview Text

```
1 User has been deleted....
```

Fire the same request again and check

Body Cookies Headers (3) Test Results Status: 404 Not Found Time: 20 ms

Pretty Raw Preview JSON

```
1 {
2   "timestamp": "2018-01-27T22:09:29.194+0000",
3   "message": "User Not Found with id -> 3",
4   "details": "uri=/users/3"
5 }
```

Implementing validations

Here we will use Java validation API.

We will add `@valid` annotation in post request

```
@PostMapping("/users")
public ResponseEntity<User> createUser( @Valid @RequestBody User user)
```

in User Bean let's modify something

```
@Size(min=2)
private String name;
@Past//means should not be a future date
private Date birthDate;
```

Now fire a Post request with name "R"

POST Params

form-data x-www-form-urlencoded raw binary JSON (application/json)

```
1 {
2   "name": "R",
3   "birthDate": "2018-01-27T22:29:37.719+0000"
4 }
```

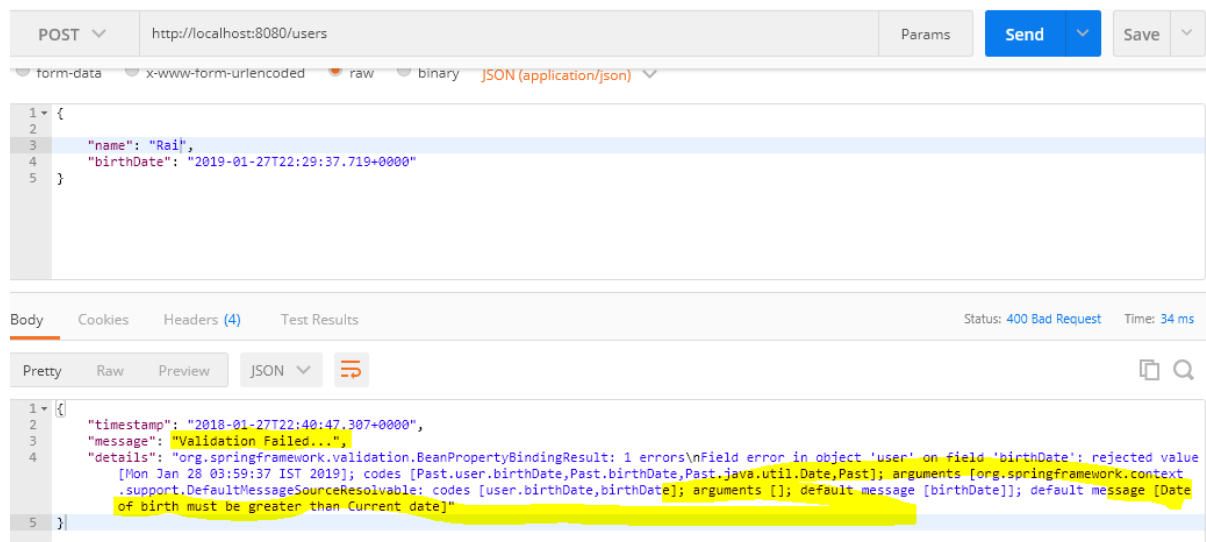
Body Cookies Headers (4) Test Results Status: 400 Bad Request Time: 25 ms

Pretty Raw Preview JSON

```
1 {
2   "timestamp": "2018-01-27T22:36:40.756+0000",
3   "message": "Validation failed for argument at index 0 in method: public org.springframework.http.ResponseEntity<com.wipro.rest.restwebservice.user.User> com.wipro.rest.restwebservice.user.UserResourceController.createUser(com.wipro.rest.restwebservice.user.User), with 1 error(s): [Field error in object 'user' on field 'name': rejected value [R]; codes [Size.user.name,Size.name,Size.java.lang.String,Size]; arguments [org.springframework.context.support.DefaultMessageSourceResolvable: codes [user.name,name]; arguments []; default message [name],2147483647,2]; default message [size must be between 2 and 2147483647]] ",
4   "details": "org.springframework.validation.BeanPropertyBindingResult: 1 errors\nField error in object 'user' on field 'name': rejected value [R]; codes [Size.user.name,Size.name,Size.java.lang.String,Size]; arguments [org.springframework.context.support.DefaultMessageSourceResolvable: codes [user.name,name]; arguments []; default message [name],2147483647,2]; default message [size must be between 2 and 2147483647]"
5 }
```

We have so much details in the message. We don't want all these to displayed to consumer. Let modify in `CustomizedResponseEntityExceptionHandler`

```
@Override
    protected ResponseEntity<Object>
    handleMethodArgumentNotValid(MethodArgumentNotValidException ex, HttpHeaders
    headers, HttpStatus status, WebRequest request)
    {
        GeneralExceptionResponseBean er=new GeneralExceptionResponseBean(new
    Date(),"Validation Failed...", ex.getBindingResult().toString());
        return new ResponseEntity(er, HttpStatus.BAD_REQUEST);
    }
```



HATEOAS (Hypermedia as the Engine of Application State)

It is used to send some extra relevant information to end user when he requests a web service.

As per example here what we will use is – A user request server to see one specific user detail by passing id, then server will response the user detail along with a link to view all the users.

How to achieve this-

Add HATEOAS dependency in pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-hateoas</artifactId>
</dependency>
```

In controller class do a static import for Controller Builder

```
import static org.springframework.hateoas.mvc.ControllerLinkBuilder.*;
```

Add some logic to `getUserById` method().

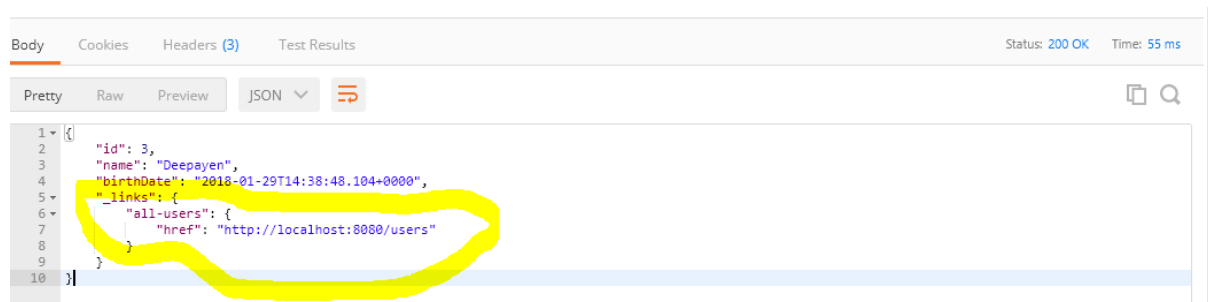
```
@GetMapping(path="/users/{id}")
    public Resource<User> retrieveUser(@PathVariable int id)
    {
        User user=service.findOne(id);
        if(user==null)
        {
```

```

        throw new UserNotFoundException("User Not found with id :
"+id);
    }
    Resource<User> resource=new Resource<User>(user);//create a resource
    ControllerLinkBuilder
    linkTo=LinkTo(methodOn(this.getClass()).retriveAllUsers());//create link
    resource.add(linkTo.withRel("all-users"));//add this link to
    resource and return
    return resource;
}

```

Sometime you may get ClassNotFoundException because we have added HATEOAS in pom.xml and that might not be found in classpath so restart the server. It will be resolved.



Internationalization in RESTful web services

Configure Locale Resolver

Customize ResourceBunndleMessageSource

Create properties file in resource folder

Now we need something to read those customized messages – will define another bean in RESTful web service header

Update HelloWorld Service to use it.

Content negotiation – Implementing Support for XML

Whatever we have done till now will take input as JSON and produces output as JSON. But what if we want XML representation of resources.

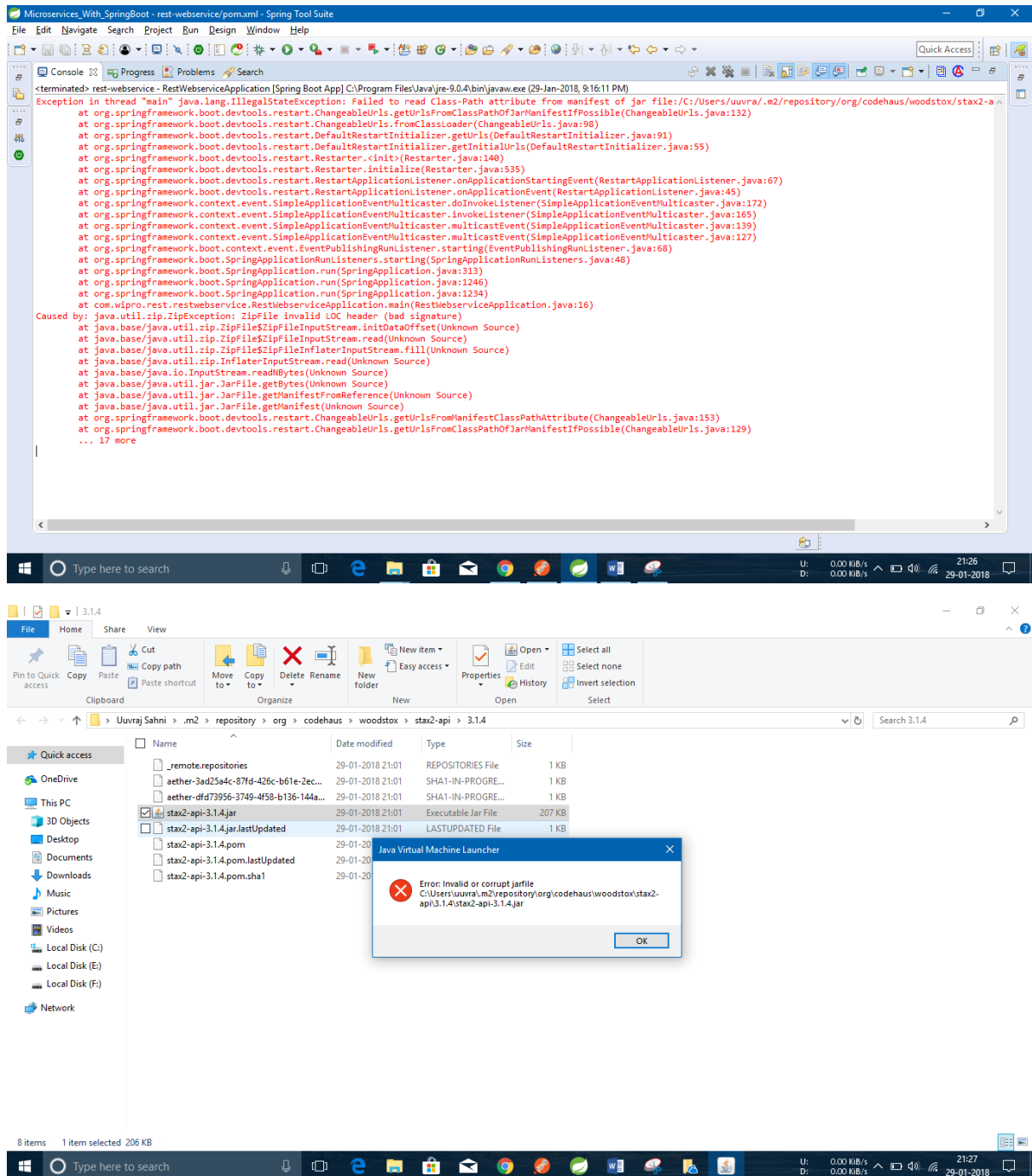
Add a new dependency

```

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

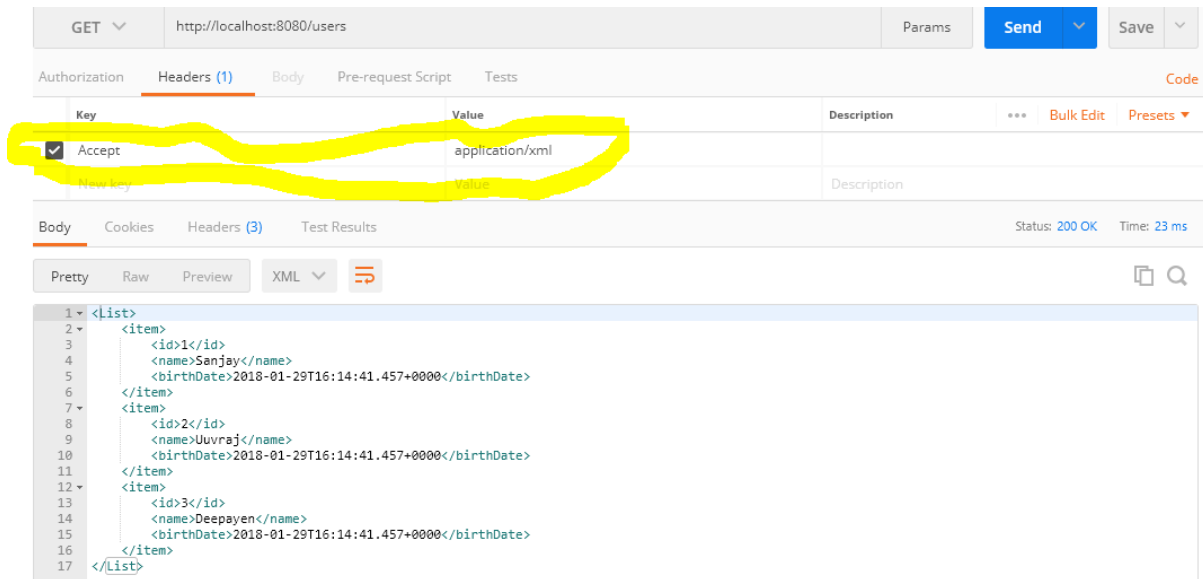
```

This dependency will do the XML conversion.



The error is due to corrupt jar file. Remove everything from from pom.xml and again rewrite and run. Clean and install

Whatever is configured in Accept value. First we need to add dependency then we can get desired response.



Till Now we created so many restful services but how to share with consumers. Consumer can ask what is contract of services. In SOAP for the same we have WSDL. For RESTful services there is no such standard. For this we will be using Swagger. Swagger is documentation format for RESTful services.

To achieve this we need to add two swagger dependencies.

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.4.0</version>
</dependency>
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.4.0</version>
</dependency>

<dependency>
```

And create a Swagger Configuration class

```
package com.wipro.rest.restwebservice.user;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

@Configuration
@EnableSwagger2
public class SwaggerConfig {

    @Bean
    public Docket api() {
```

```

        return new Docket(DocumentationType.SWAGGER_2);
    }
}

```

Take a look on api docs created-

<http://localhost:8080/v2/api-docs>

```


{
  swagger: "2.0",
+ info: {...},
  host: "localhost:8080",
  basePath: "/",
+ tags: [...],
+ paths: {...},
+ definitions: {...}
}

{
  swagger: "2.0",
- info: {
    description: "Api Documentation",
    version: "1.0",
    title: "Api Documentation",
    termsOfService: "urn:tos",
    contact: { },
    - license: {
      name: "Apache 2.0",
      url: "http://www.apache.org/licenses/LICENSE-2.0"
    }
  },
  host: "localhost:8080",
  basePath: "/",
- tags: [
  - {
    name: "hello-world-controller",
    description: "Hello World Controller"
  },
  - {
    name: "user-resource-controller",
    description: "User Resource Controller"
  },
  - {
    name: "basic-error-controller",
    description: "Basic Error Controller"
  }
]
}

```

UI of Contracts-

<http://localhost:8080/swagger-ui.html>

 **swagger**

default (/v2/api-docs) Explore

Api Documentation

Api Documentation

[Apache 2.0](#)

basic-error-controller : Basic Error Controller	Show/Hide	List Operations	Expand Operations
hello-world-controller : Hello World Controller	Show/Hide	List Operations	Expand Operations
user-resource-controller : User Resource Controller	Show/Hide	List Operations	Expand Operations

[BASE URL: / , API VERSION: 1.0]

Enhancing Swagger Documentation with Custom Annotation

If you notice previous documentation some informations are missing like length of name validation, birth date cannot be in future.

Now we will include these validations as per enhancement.

```

- info: {
  description: "description-social media API",
  version: "version-1.0.0",
  title: "title-Social Media API",
  termsOfService: "termsOfServiceUrl- No Tnc Enjoy!!!",
  - contact: {
    name: "Sanjay Rai",
    url: "http://www.sanjayrai.com",
    email: "info@sanjayrai.com"
  },
  - license: {
    name: "license here",
    url: "license url here"
  }
},
host: "localhost:8080",
basePath: "/",
- tags: [
  - {
    name: "hello-world-controller",
    description: "Hello World Controller"
  },
  - {
    name: "user-resource-controller",
    description: "User Resource Controller"
  },
  - {
    name: "basic-error-controller",
    description: "Basic Error Controller"
  }
],
- consumes: [
  "application/xml",
  "application/json"
],
- produces: [
  "application/xml",
  "application/json"
]

```

Validations-


```

- definitions: {
  - User: {
    type: "object",
    - properties: {
      - birthDate: {
        type: "string",
        format: "date-time",
        description: "Birth date must be in past"
      },
      - id: {
        type: "integer",
        format: "int32"
      },
      - name: {
        type: "string",
        description: "Name must have atleast two characters"
      }
    },
    description: "All details about users"
  },
  - Resource«User»: {
    type: "object",
    - properties: {
      - birthDate: {
        type: "string",
        format: "date-time",
        description: "Birth date must be in past"
      },
      - id: {
        type: "integer",
        format: "int32"
      },
      - links: {
        type: "array",
        - items: {
          $ref: "#/definitions/Link"
        }
      }
    }
  }
}

```

To achieve it we created one swagger config file and added some annotation in User model class

Config file

```

package com.wipro.rest.restwebservice.user;

import java.util.Arrays;

import java.util.HashSet;
import java.util.Set;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import springfox.documentation.service.ApiInfo;
import springfox.documentation.service.Contact;
import springfox.documentation.spi.DocumentationType;

```

```

import springfox.documentation.spring.web.plugins.Docket;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

@Configuration
@EnableSwagger2
public class SwaggerConfig {
    public static final Contact DEFAULT_CONTACT = new Contact(
        "Sanjay Rai", "http://www.sanjayrai.com",
        "info@sanjayrai.com");

    private static final ApiInfo DEFAULT_API_INFO = new ApiInfo("title-Social
Media API", "description-social media API",
        "version-1.0.0", "termsOfServiceUrl- No Tnc Enjoy!!!",
        DEFAULT_CONTACT, "license here", "license url here");

    private static final Set<String> DEFAULT_PRODUCES_AND_CONSUMES = new
HashSet<String>(Arrays.asList("application/json", "application/xml"));

    @Bean
    public Docket api() {

        return new
Docket(DocumentationType.SWAGGER_2).apiInfo(DEFAULT_API_INFO).produces(DEFAULT_PRODUCES_AND_CONSUMES).consumes(DEFAULT_PRODUCES_AND_CONSUMES);
    }
}

```

User Model class

```

@ApiModel(description="All details about users")
/* API model is used to provide details about model class i.e bean in
documentation*/
public class User {
    private Integer id;
    @Size(min=2, message="Name should have atleast 2 characters")/* For
validation purpose*/
    @ApiModelProperty(notes="Name must have atleast two characters") /*swagger
documentation purpose*/
    private String name;
    @Past (message="Date of birth must be greater than Current date")//means
should not be a future date-validation
    @ApiModelProperty(notes="Birth date must be in past")/*swagger
documentation purpose*/
    private Date birthDate;
}

```

Spent more time here on swagger documentation – Explore swagger-annotations jar file

Monitoring APIs with Spring Boot Actuator

- If our service is failing or down we should know ASAP
- Spring Boot provides great support to monitor these services.

Two achieve it we will add two dependencies

```

<!-- It provides a lot of monitoring facilities around our services -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<!-- Using this we can see services provided by actuator on browser in HAL
format -->
<dependency>
  <groupId>org.springframework.data</groupId>
  <artifactId>spring-data-rest-hal-browser</artifactId>
</dependency>

```

Static Filtering

While response we were providing all the attributes of bean class this is what we have learnt till now. Suppose if User class has password as attribute what we will do?

Here comes filtering in picture. In response we will return few attributes among all the attributes.

Note : Always use field level ignore.

Let's create a Customer class

```

package com.wipro.rest.restwebservice.filtering;

import com.fasterxml.jackson.annotation.JsonIgnore;
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;

/*@JsonIgnoreProperties(value= {"password","id"})
  This is not a good approach to filter the responses
  Suppose later we rename the password field as customerPassword
  and forgot to change this in JsonIgnoreProperties values
  then password will be sent as response
  so we will not use it
  * */
public class CustomerBean {
    @JsonIgnore /* If field is renamed also, will not affect response*/
    private int id;
    private String userName;
    @JsonIgnore /* If field is renamed also, will not affect response*/
    private String password;
    public CustomerBean(int id, String userName, String password) {
        super();
        this.id = id;
        this.userName = userName;
        this.password = password;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {

```

```

        this.id = id;
    }
    public String getUsername() {
        return userName;
    }
    public void setUsername(String userName) {
        this.userName = userName;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
}
}

```

Create a Controller for it

```

package com.wipro.rest.restwebservice.filtering;

import java.util.Arrays;
import java.util.List;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

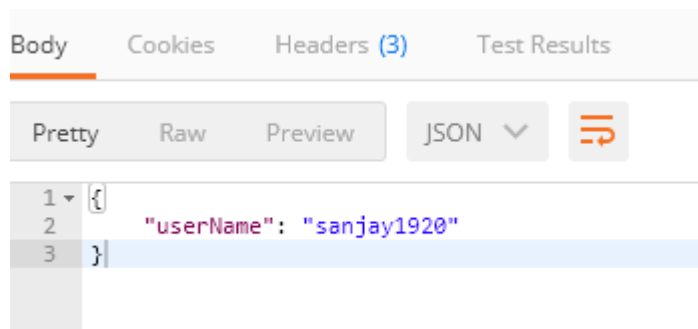
@RestController
public class CustomerStaticFilterController {
    @GetMapping(path="/customer")
    public CustomerBean getCustomer()
    {
        return new CustomerBean(100, "sanjay1920", "password");
    }

    @GetMapping(path="/customer-list")
    public List<CustomerBean> getCustomerList ()
    {
        return Arrays.asList( new CustomerBean(100, "sanjay1920",
"password"), new CustomerBean(101, "uuvraj1920", "secret"));
    }
}

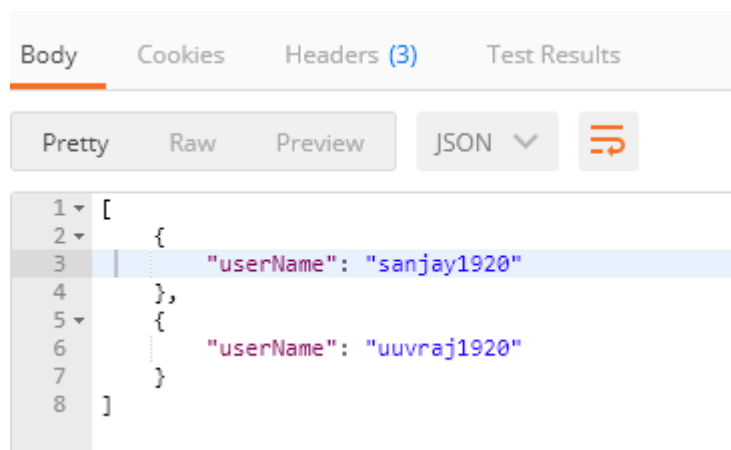
```

Output-

customer



Customer-list



Dynamic Filtering

In last example we hard coded what field to be ignored. Here we will decide it dynamically. Like in first response customer id will be returned and in second it will not.

In Dynamic filtering we cannot apply filters directly on beans/attributes.

It will be applied where we are retrieving the values i.e where our service is defined.

We will use MappingvalueJacksonValue class.

Complete Code

Bean Class

```
package com.wipro.rest.restwebservice.filtering;

import com.fasterxml.jackson.annotation.JsonFilter;
import com.fasterxml.jackson.annotation.JsonIgnore;
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;

/*@JsonIgnoreProperties(value= {"password","id"})
 This is not a good approach to filter the responses
 Suppose later we rename the password field as customerPassword
 and forgot to change this in JsonIgnoreProperties values
 then password will be sent as response
 so we will not use it
 * */
@JsonFilter("CustomerBeanOneFilter")
public class CustomerBean {
```

```

/*@JsonIgnore If field is renamed also, will not affect response*/
private int id;
private String userName;
/* @JsonIgnore If field is renamed also, will not affect response*/
private String password;
public CustomerBean(int id, String userName, String password) {
    super();
    this.id = id;
    this.userName = userName;
    this.password = password;
}
public int getId() {
    return id;
}
public void setId(int id) {
    this.id = id;
}
public String getUserName() {
    return userName;
}
public void setUserName(String userName) {
    this.userName = userName;
}
public String getPassword() {
    return password;
}
public void setPassword(String password) {
    this.password = password;
}
}
}

```

Controller Class

```

package com.wipro.rest.restwebservice.filtering;

import java.util.Arrays;
import java.util.List;

import org.springframework.http.converter.json.MappingJacksonValue;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import com.fasterxml.jackson.databind.ser.FilterProvider;
import com.fasterxml.jackson.databind.ser.impl.SimpleBeanPropertyFilter;
import com.fasterxml.jackson.databind.ser.impl.SimpleFilterProvider;

@RestController
public class CustomerStaticFilterController {
    @GetMapping(path="/customer-static")
    public CustomerBean getCustomer()
    {
        return new CustomerBean(100, "sanjay1920", "password");
    }

    @GetMapping(path="/customer-list-static")

```

```

    public List<CustomerBean> getCustomerList ()
    {
        return Arrays.asList( new CustomerBean(100, "sanjay1920",
"password"), new CustomerBean(101, "uuvraj1920", "secret"));
    }

    @GetMapping(path="/customer-dynamic")
    public MappingJacksonValue getCustomerDynamic()
    {
        CustomerBean customerBean= new CustomerBean(100, "sanjay1920",
"password");
        MappingJacksonValue mapping=new MappingJacksonValue(customerBean);
        SimpleBeanPropertyFilter
filter=SimpleBeanPropertyFilter.filterOutAllExcept("id","userName");
        /* SimpleBeanPropertyFilter is abstract class----
SimpleBeanPropertyFilter is implementation class of it*/
        FilterProvider filters =new
SimpleFilterProvider().addFilter("CustomerBeanOneFilter", filter);
        /*If we don't define "CustomerBeanOneFilter" on CustomerBean then
filtering will not work
        and all values will be returned.Most of the developers make this
mistake

        Define JsonFilter in CustomerBean class
        * */
        mapping.setFilters(filters);
        return mapping;
    }

    @GetMapping(path="/customer-list-dynamic")
    /*ALT+SHIFT+L*/
    public MappingJacksonValue getCustomerListDynamic ()
    {
        List<CustomerBean> customerList = Arrays.asList( new
CustomerBean(100, "sanjay1920", "password"), new CustomerBean(101, "uuvraj1920",
"secret"));
        MappingJacksonValue mapping=new MappingJacksonValue(customerList);
        SimpleBeanPropertyFilter
filter=SimpleBeanPropertyFilter.filterOutAllExcept("id","password");
        FilterProvider filters =new
SimpleFilterProvider().addFilter("CustomerBeanOneFilter", filter);
        mapping.setFilters(filters);
        return mapping;
    }
}

```

Output-

GET http://localhost:8080/customer-dynamic Params Send Save

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

Key	Value	Description	...	Bulk Edit	Presets
Accept	application/json				
New key	Value	Description			

body Cookies Headers (3) Test Results Status: 200 OK Time: 23 ms

Pretty Raw Preview JSON

```

1 {
2   "id": 100,
3   "userName": "sanjay1920"
4 }

```

List

GET http://localhost:8080/customer-list-dynamic Params Send Save

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

Key	Value	Description	...	Bulk Edit	Presets
Accept	application/json				
New key	Value	Description			

Body Cookies Headers (3) Test Results Status: 200 OK Time: 31 ms

Pretty Raw Preview JSON

```

1 [
2   {
3     "id": 100,
4     "password": "password"
5   },
6   {
7     "id": 101,
8     "password": "secret"
9   }
10 ]

```

Versioning Web Services

There are many ways to version web services. These are

Creating a new service and changing the URI

```

/* version-1 API*/
@GetMapping("/v1/person")
public PersonVersion1 personVersion1()
{
    return new PersonVersion1("Sanjay Rai");
}
/* version-2 API*/
@GetMapping("/v2/person")
public PersonVersion2 PersonVersion2()
{
    return new PersonVersion2(new Name("Sanjay", "Rai"));
}

```


GET ▼ http://localhost:8080/v1/person Params Send Save ▼

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

Key	Value	Description
<input checked="" type="checkbox"/> Accept	application/json	
New key	Value	Description

Body Cookies Headers (3) Test Results Status: 200 OK Time: 44 ms

Pretty Raw Preview JSON ▼ ≡

```

1 {
2   "name": "Sanjay Rai"
3 }

```

GET ▼ http://localhost:8080/v2/person Params Send Save ▼

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

Key	Value	Description
<input checked="" type="checkbox"/> Accept	application/json	
New key	Value	Description

Body Cookies Headers (3) Test Results Status: 200 OK Time: 26 ms

Pretty Raw Preview JSON ▼ ≡

```

1 {
2   "name": {
3     "firstName": "Sanjay",
4     "lastName": "Rai"
5   }
6 }

```

By Passing Request parameters

/* Versioning by passing request parameters-- Here URI is same
While consuming we will pass different parameters
* */

```

@GetMapping(value="/person/param", params="version=2")
public PersonVersion2 param2()
{
    return new PersonVersion2(new Name("Sanjay", "Rai"));
}

@GetMapping(value="/person/param", params="version=1")
public PersonVersion1 param1()
{
    return new PersonVersion1("Sanjay Rai");
}

```

GET ▼ http://localhost:8080/person/param?version=1 Params Send Save ▼

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

Key	Value	Description
<input checked="" type="checkbox"/> Accept	application/json	
New key	Value	Description

Body Cookies Headers (3) Test Results Status: 200 OK Time: 22 ms

Pretty Raw Preview JSON ▼ ≡

```

1 {
2   "name": "Sanjay Rai"
3 }

```

GET **http://localhost:8080/person/param?version=2** Params Send Save

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

Key	Value	Description
<input checked="" type="checkbox"/> Accept	application/json	
New key	Value	Description

Body Cookies Headers (3) Test Results Status: 200 OK Time: 41 ms

Pretty Raw Preview JSON

```

1 {
2   "name": {
3     "firstName": "Sanjay",
4     "lastName": "Rai"
5   }
6 }

```

By passing parameter in request Header

```

/* By Header parameters*/
@GetMapping(value="/person/header", headers="X-API-VERSION=1")
public PersonVersion1 header1()
{
    return new PersonVersion1("Sanjay Rai");
}
@GetMapping(value="/person/header", headers="X-API-VERSION=2")
public PersonVersion2 header2()
{
    return new PersonVersion2(new Name("Sanjay", "Rai"));
}

```

GET **http://localhost:8080/person/header** Params Send Save

Authorization Headers (2) Body Pre-request Script Tests Code

Key	Value	Description
<input checked="" type="checkbox"/> Accept	application/json	
<input checked="" type="checkbox"/> X-API-VERSION	1	
New key	Value	Description

Body Cookies Headers (3) Test Results Status: 200 OK Time: 26 ms

Pretty Raw Preview JSON

```

1 {
2   "name": "Sanjay Rai"
3 }

```

GET **http://localhost:8080/person/header** Params Send Save

Authorization Headers (2) Body Pre-request Script Tests Code

Key	Value	Description
<input checked="" type="checkbox"/> Accept	application/json	
<input checked="" type="checkbox"/> X-API-VERSION	2	
New key	Value	Description

Body Cookies Headers (3) Test Results Status: 200 OK Time: 33 ms

Pretty Raw Preview JSON

```

1 {
2   "name": {
3     "firstName": "Sanjay",
4     "lastName": "Rai"
5   }
6 }

```

Using produces

This is also passed in header Accept parameter.

```
/* Using produces - MIME type*/
```

```
@GetMapping(value="/person/produces",
produces="application/vnd.company.app-v1+json")
public PersonVersion1 produces1()
{
    return new PersonVersion1("Sanjay Rai");
}
@GetMapping(value="/person/produces",
produces="application/vnd.company.app-v2+json")
public PersonVersion2 produces2()
{
    return new PersonVersion2(new Name("Sanjay", "Rai"));
}
```

GET `http://localhost:8080/person/produces` Params Send Save

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

Key	Value	Description
Accept	application/vnd.company.app-v1+json	
New key	Value	Description

Body Cookies Headers (3) Test Results Status: 200 OK Time: 33 ms

Pretty Raw Preview JSON

```
1 {
2   "name": "Sanjay Rai"
3 }
```

GET `http://localhost:8080/person/produces` Params Send Save

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

Key	Value	Description
Accept	application/vnd.company.app-v2+json	
New key	Value	Description

Body Cookies Headers (3) Test Results Status: 200 OK Time: 59 ms

Pretty Raw Preview JSON

```
1 {
2   "name": {
3     "firstName": "Sanjay",
4     "lastName": "Rai"
5   }
6 }
```

Let's discuss pros and cons of these 4 ways to implementing versioning

URI versioning

- Polluting URI space. New URI is generated for every version
- Used by Twitter.
- As URI is different so it can be cached.
- Can be executed on Browser.
- End user does not require any technical knowledge to execute it as it is browser based.

Request Parameter Versioning

- Polluting URI space. New URI is generated for every version
- Used by Amazon.
- As URI is different so it can be cached.
- Can be executed on Browser.
- End user does not require any technical knowledge to execute it as it is browser based.

Media Type Versioning (Produces/Content negotiation or accept header)

- Does not pollutes URI space as new URI is not generated for every version
- Used by GitHub
- As URI is same and part of header so difficult to cache.
- Cannot be executed on Browser.
- End user require a bit of technical knowledge to execute it requires a REST Client like Postman/SOAP UI.

Header Versioning

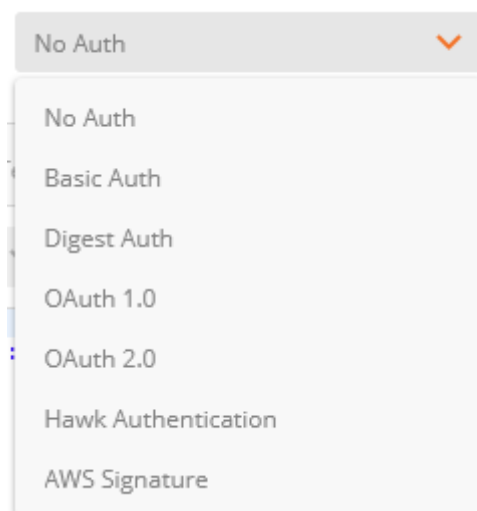
- Does not pollutes URI space as new URI is not generated for every version
- Used by Microsoft
- As URI is same and part of header so difficult to cache.
- Cannot be executed on Browser.
- End user require a bit of technical knowledge to execute it requires a REST Client like Postman/SOAP UI.

Summary : There is no perfect solution for versioning. Before we create APIs our versioning strategy should be finalized.

Implementing Basic Security

Whatever we did till now Authorization was selected as No Auth by Default

There a lot of security types



Add dependency to pom.xml

Once we restart the server , get default security password. Please make sure the time you restart the server new default security password is generated in logger.

```
2018-02-03 23:24:50.353 INFO 6128 --- [ restartedMain] a.s.s.AuthenticationManagerConfiguration :  
Using generated security password: cb4f4047-fda7-4184-ace8-e8463838eaff  
2018-02-03 23:24:50.575 INFO 6128 --- [ restartedMain] o.s.s.web.DefaultSecurityFilterChain :  
2018-02-03 23:24:50.648 INFO 6128 --- [ restartedMain] o.s.b.d.a.OptionalliveReloadServer :
```

In this case default username will be user.

If we want that password should not get change frequently, we can configure it in application.properties file

```
spring.jackson.serialization.write-dates-as-timestamps=false  
logging.level.org.org.springframework=info  
spring.security.user.name=sanjay1920  
spring.security.user.password=secret123
```

Once you implement the security try to get/post any of the API. You will get unauthorized error.

The screenshot shows a REST client interface with a GET request to `http://localhost:8080/users`. The response status is `401 Unauthorized` with a time of `26 ms`. The response body is a JSON object:

```
{  
  "timestamp": "2018-02-03T18:09:25.729+0000",  
  "status": 401,  
  "error": "Unauthorized",  
  "message": "Unauthorized",  
  "path": "/users"  
}
```

Now let's provide authentication details and see the results

GET ▼ http://localhost:8080/users/1 Params Send Save ▼

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

Type Basic Auth ▼ Clear Update Request

Username sanjay1920 The authorization header will be generated and added as a custom header

Password ***** ☐ Save helper data to request

☐ Show Password

Body Cookies Headers (9) Test Results Status: 200 OK Time: 79 ms

Pretty Raw Preview JSON ▼ ≡

```

1 {
2   "id": 1,
3   "name": "Sanjay",
4   "birthDate": "2018-02-03T18:03:38.633+0000",
5   "_links": {
6     "all-users": {
7       "href": "http://localhost:8080/users"
8     }
9   }
10 }

```

Another example

GET ▼ http://localhost:8080/person/produces Params Send Save ▼

Authorization ● Headers (2) Body Pre-request Script Tests Code

	Key	Value	Description	...	Bulk Edit	Presets
<input checked="" type="checkbox"/>	Accept	application/vnd.company.app-v2+json				
<input checked="" type="checkbox"/>	Authorization	Basic c2FuamF5MTkyMDpzZW5yZXQxMjM=				
	New key	Value	Description			

Body Cookies Headers (9) Test Results Status: 200 OK Time: 24 ms

Pretty Raw Preview JSON ▼ ≡

```

1 {
2   "name": {
3     "firstName": "Sanjay",
4     "lastName": "Rai"
5   }
6 }

```

Once we apply security in actuator also it will ask for password.

Connecting RESTful service to JPA

We already have added jpa and in memory database H2.

Important Notes-

data.sql file must be created in resource folder.

If you change file name it will not be like userData.sql values are not getting inserted.

What will happen if two resources having same URI?

Runtime exception-java.lang.IllegalStateException: Ambiguous mapping.

H2 in memory database console

Make changes in properties file

```
spring.jpa.show-sql=true  
spring.h2.console.enabled=true
```

Console of H2 database

<http://localhost:8080/h2-console>

English ▼ Preferences Tools Help

Login

Saved Settings: Generic H2 (Embedded) ▼

Setting Name: Generic H2 (Embedded) Save Remove

Driver Class: org.h2.Driver

JDBC URL: jdbc:h2:mem:testdb

User Name: sa

Password:

Connect Test Connection

Auto commit Auto Max rows: 1000 Auto complete Off Auto select On ?

jdbc:h2:mem:testdb

USER

INFORMATION_SCHEMA

Users

H2 1.4.196 (2017-06-10)

Run Run Selected Auto complete Clear SQL statement:

SELECT * FROM USER|

SELECT * FROM USER;

ID	BIRTH_DATE	NAME
1	2018-02-04 12:07:12.679	Sanjay Rai - JPA
2	2018-02-04 12:11:48.908	Uuvraj Sahani - JPA
3	2018-02-04 12:11:48.908	Deepayen Gupta - JPA

(3 rows, 3 ms)

Edit

Full Code

User Repository Interface

```
package com.wipro.rest.restwebservice.user;  
  
import org.springframework.data.jpa.repository.JpaRepository;  
import org.springframework.stereotype.Repository;  
  
@Repository
```

```

public interface UserRepository extends JpaRepository<User/*Entity to be
managed*/, Integer/*Primary key of entity*/> {

}

```

UserJpaResource.java

```

package com.wipro.rest.restwebservice.user;

import static org.springframework.hateoas.mvc.ControllerLinkBuilder.linkTo;

import static org.springframework.hateoas.mvc.ControllerLinkBuilder.methodOn;

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

import javax.validation.Valid;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.hateoas.Resource;
import org.springframework.hateoas.mvc.ControllerLinkBuilder;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;

import com.wipro.rest.restwebservice.exception.UserNotFoundException;

@RestController
public class UserJpaResource {
    @Autowired
    private UserRepository userRepository;
    @GetMapping(path="/jpa/users")
    public List<User> retrieveAllUsers()
    {
        return userRepository.findAll();
    }
    @GetMapping(path="jpa/users/{id}")
    public Resource<User> retrieveUser(@PathVariable int id)
    {
        Optional<User> user=userRepository.findById(id);
        /*Here Optional concept is new- means it will return a proper object
        whether id is found
        or not found. It will never return null.
        * */

        if(!user.isPresent())
        {
            throw new UserNotFoundException("User Not found with id :
"+id);

```



```

    }
    Resource<User> resource=new Resource<User>(user.get());//create a
resource
    ControllerLinkBuilder
linkTo=LinkTo(methodOn(this.getClass()).retriveAllUsers());//create link
    resource.add(linkTo.withRel("all-users-from-H2-database"));//add
this link to resource and return
    return resource;
}

/*
To send post request we will use Rest Client
* */
@PostMapping("/jpa/users")
public ResponseEntity<User> createUser( @Valid @RequestBody User user)
{
    User savedUser=userRepository.save(user);
URI location = ServletUriComponentsBuilder.
    fromCurrentRequest().
    path("/{id}").
    buildAndExpand(savedUser.getId()).toUri();
return ResponseEntity.created(location).build();
}

@DeleteMapping("jpa/users/{id}")
public ResponseEntity<String> deleteUser(@PathVariable int id)
{
    userRepository.deleteById(id);

    return ResponseEntity.ok("User has been deleted....");

}

}

```