**REST Services:**

**REST API Validations:**

For field level validations such as Empty Name, Future Date check. We can use ***spring-boot-starter-validation*** for field-level validations in Spring Boot.

Let’s work with a simple example:

**Validation to apply:**

We have UserController, where we have the createUser method to create a new user. We have two fields in the user object for which we want to add validations for the incoming request of user creation:

1. **Name:** user name should not be less than 4 characters.
2. **Date Of Birth:** The user’s date of birth should not be a future date.

**How to apply with Spring boot validations:**

1. We already added the dependency ***spring-boot-starter-validation*** in***pom.xml****.*
2. At the code level, we need to do below things:
   1. Add @Valid annotation at createUser method of UserContoller.

@PostMapping("/createUser")

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

* 1. Add validations on fields of the User entity.

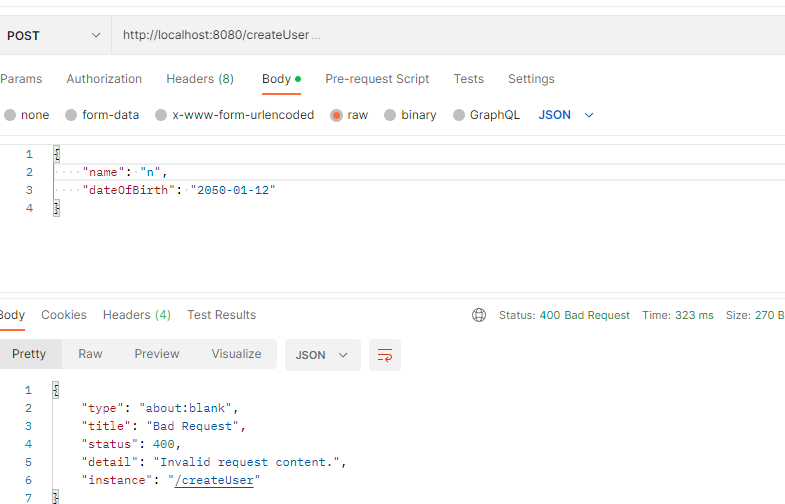
@Size(min = 4)

**private** String name;

@Past

**private** LocalDate dateOfBirth;

**Let’s run the project and see the output:** Here I have passed 1 character in name and future date in the date of birth field. You can see the response status 400 (Bad request), which is correct as per the applied validations. But here user will not understand the reason for the status code: 400, so next, we will see how to send correct validation error messages.

****

* 1. Return the readable response to the consumer of the API.

We need to add a validation message on each field for returning proper validation messages. Next, we need to override the handleMethodArgumentNotValid method of ResponseEntityExceptionHandler class of the spring framework. Below is an example:

**ErrorDetail Entity to represent the error message:**

**public** **class** ErrorDetailDto {

**private** String message;

**private** String details;

**private** LocalDateTime dateTime;

**private** List<AppFieldError> errors;

**public** ErrorDetailDto() {

}

**public** ErrorDetailDto(String message, String details, LocalDateTime dateTime) {

**this**.message = message;

**this**.details = details;

**this**.dateTime = dateTime;

}

//getters and setters

}

**Custom messages at User Entity for information to the consumer:**

// Name of the User Entity

@Size(min = 4, max = 50, message = "Name field should have minimum 4 characters and maximum 50 characters")

**private** String name;

// Birthday of the user.

@Past(message = "date of Birth should not be future date")

**private** LocalDate dateOfBirth;

**Example to override the handleMethodArgumentNotValid method of ResponseEntityExceptionHandler class of spring framework:**

@ControllerAdvice

**public** **class** CustomExceptionHandler **extends** ResponseEntityExceptionHandler{

@Nullable

@Override

**protected** ResponseEntity<Object> handleMethodArgumentNotValid(

MethodArgumentNotValidException ex, HttpHeaders headers, HttpStatusCode status, WebRequest request) {

ErrorDetailDto error = **new** ErrorDetailDto("Validation errors", request.getDescription(**false**), LocalDateTime.*now*());

ResponseEntity<Object> response = **new** ResponseEntity<>(error, HttpStatus.***BAD\_REQUEST***);

error.setErrors(fieldErrors(ex.getFieldErrors()));

**return** handleExceptionInternal(ex, response, headers, status, request);

}

**protected** List<AppFieldError> fieldErrors(List<FieldError> errors){

**return** errors.stream().map(error

-> **new** AppFieldError(error.getObjectName(), error.getField(), error.getDefaultMessage())).collect(Collectors.*toList*());

}

}

**Now when you run the application and you will get the proper validation messages in simple JSON format:**

**Request:**

{

    "name": "n",

    "dateOfBirth": "2050-01-12"

}

**Response:**

{

    "headers": {},

    "body": {

        "message": "Validation errors",

        "details": "uri=/createUser",

        "dateTime": "2023-01-26T20:24:50.9309323",

        "errors": [

            {

                "object": "user",

                "name": "dateOfBirth",

                "message": "date of Birth should not be the future date"

            },

            {

                "object": "user",

                "name": "name",

                "message": "Name field should have minimum 4 characters and maximum 50 characters"

            }

        ]

    },

    "statusCode": "BAD\_REQUEST",

    "statusCodeValue": 400

}

Following the above steps, you can modify the response format according to your requirement in the application.

**Advanced REST API Features with Spring-boot:**

Here we will cover below topics:

1. Documentation
2. Content Negotiation
3. Internationalization i18n
4. Versioning
5. HATEOS
6. Static filtering
7. Dynamic filtering
8. Monitoring
9. **Documentation:**
   * REST API consumers need to understand the below things about API:
     1. Resources
     2. Actions
     3. Structure of the Request and the Response along with constraints.
   * Challenges in documenting the REST APIs:
     1. Accuracy: To ensure that the documentation is up to date and correct.
     2. Consistency: All the APIs should be documented in a consistent format.
   * Options for documentation:
     1. Manually maintain the documentation.
     2. Generate the document from the code.

**Generate the documentation from code: Using swagger and Open API**

Spring can help us to generate documentation from our source code. To do so, we need to add spring boot starter springdoc-openapi-starter-webmvc-ui This will add swagger and open API support in the application. Here, we are using version 3 of spring-boot. The compatible version of springdoc-openapi-starter-webmvc-ui is 2.0.x. Below is the spring boot starter dependency tag which we are using in our project:

<dependency>

<groupId>org.springdoc</groupId>

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

<version>2.0.2</version>

</dependency>

The version of dependency may keep on changing. You can refer to the Spring doc official website (https://springdoc.org) and the GitHub page (https://github.com/springdoc/springdoc-openapi) to choose the compatible dependency with your Spring-boot application.

Once you add the starter dependency for springdoc-openapi, you can restart the server/application. It will generate the code documentation automatically. You can see the documentation by browsing the below URLs:

* <http://server:port/swagger-ui.html> (For Swagger UI)
* <http://server:port/v3/api-docs> (For open API -JSON format documentation)

**Content Negotiation:**

For REST APIs, the Same resource i.e., the same URI may have different representations. For example, a resource may have a response in different content-types (JSON/ XML) or may have a response in different languages. Content-Negotiation is the mechanism through which consumers tell REST API what representation they want.

The consumer uses Accept-Headers for content negotiation. Examples:

* Accept-Language-Header (en, fr)
* Accept-Header (MIME Type – JSON, XML)

Let’s work with an example of different content-types in response to a REST API. In this example, we will enable multiple content-types for a response. In spring-boot it is very easy, we need to simply add the dependency *jackson-dataformat-xml* in POM.xml

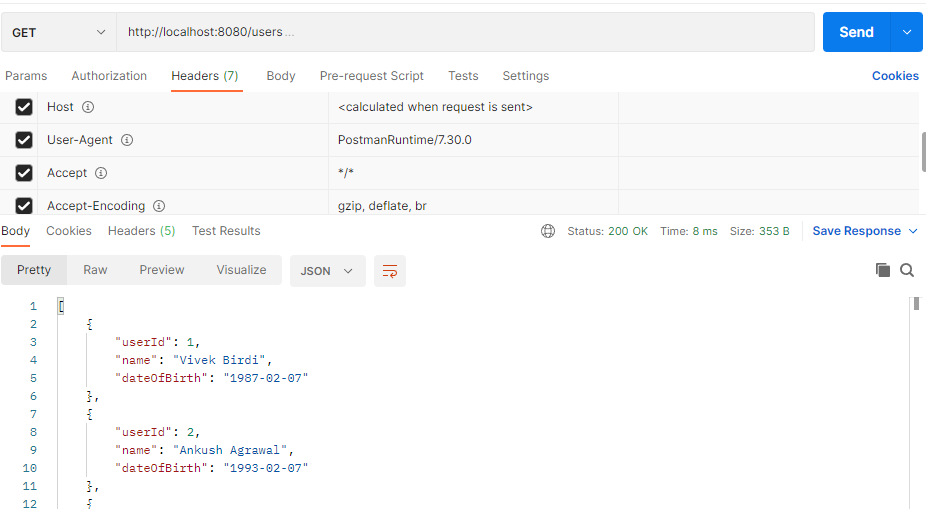
<dependency>

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

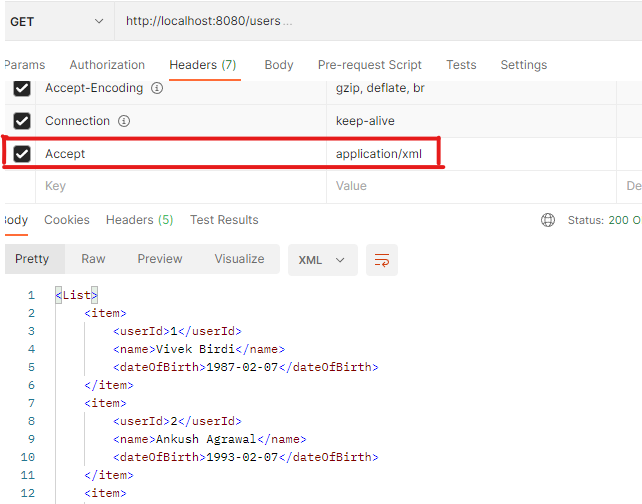
<artifactId>jackson-dataformat-xml</artifactId>

</dependency>

This will enable XML content-type for REST APIs (JSON is enabled by default). Let’s work with /users Resource of our project. When we send a request to /users resource, it returns a JSON response by default.



To get XML representation, we need to add *Accept* Header (*Accept: application/xml*) in the request. Below you can see the user is getting a response in XML :



Now let us see how can we enable multiple languages for REST APIs.

**Internalization: i18n**

REST API may have consumers of multiple languages. To respond in multiple languages internalization: i18n is used. The consumer uses *Accept-Language* Header (*Accept-Language: en*) to indicate the language preference (en- Stands for English).

Let’s work on server-side implementation for i18n. Spring-boot provides the inbuilt support for i18n. Consider that we need to write a REST API, which will return new year greetings in different languages.

* En- Happy New Year (English)
* Fr- bonne année (French)
* nl- Frohes Neues Jahr (Dutch)

We will create a resource, which will return the i18n message. For that let’s create a method *newYearGreeting* with *GetMapping* in HelloController. In this method, we require a *MessageSource* bean. *MessageSource* is the bean provided by Spring for i18n. We will use constructor injection to get the object of this bean in our HelloController.

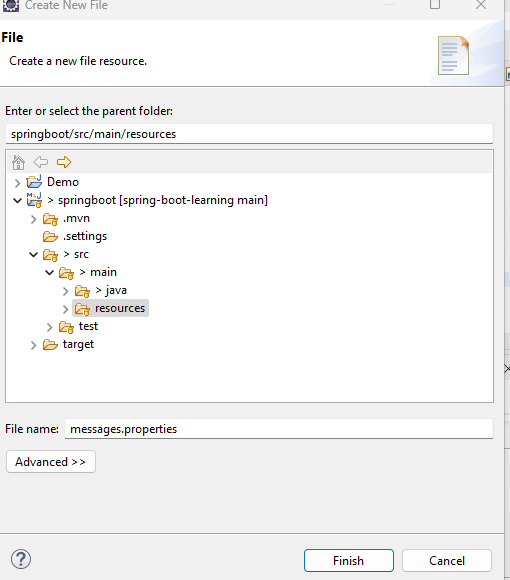
**private** MessageSource messageSource;

**public** HelloController(MessageSource messageSource) {

**this**.messageSource = messageSource;

}

Next, we need to define messages in different languages. Spring reads messages\_{locale}.properties for i18n messages (For English messages.properties) from the resources folder. Note that messages.properties should be placed in parallel to application.properties.



In this file, you can write your localized message.

* messages.properties
  + new.year.greeting= Happy New Year
* messages\_fr.properties
  + new.year.greeting= bonne année
* messages\_nl.properties
  + new.year.greeting= Frohes Neues Jahr

Next, we need to get the localized message from MessageSources and return it in response from the *newYearGreeting* method. Here we will use [MessageSource](eclipse-javadoc:%E2%98%82=springboot/D:%5C/Vivek%5C/mavenrepo%5C/org%5C/springframework%5C/spring-context%5C/6.0.3%5C/spring-context-6.0.3.jar=/maven.pomderived=/true=/=/maven.pomderived=/true=/=/maven.groupId=/org.springframework=/=/maven.artifactId=/spring-context=/=/maven.version=/6.0.3=/=/maven.scope=/compile=/%3Corg.springframework.context(MessageSource.class%E2%98%83MessageSource).getMessage([String](eclipse-javadoc:%E2%98%82=springboot/D:%5C/Vivek%5C/mavenrepo%5C/org%5C/springframework%5C/spring-context%5C/6.0.3%5C/spring-context-6.0.3.jar=/maven.pomderived=/true=/=/maven.pomderived=/true=/=/maven.groupId=/org.springframework=/=/maven.artifactId=/spring-context=/=/maven.version=/6.0.3=/=/maven.scope=/compile=/%3Corg.springframework.context(MessageSource.class%E2%98%83MessageSource~getMessage~Ljava.lang.String;~%5C%E2%98%83Ljava.lang.Object;~Ljava.lang.String;~Ljava.util.Locale;%E2%98%82java.lang.String) code, **@**[**Nullable**](eclipse-javadoc:%E2%98%82=springboot/D:%5C/Vivek%5C/mavenrepo%5C/org%5C/springframework%5C/spring-context%5C/6.0.3%5C/spring-context-6.0.3.jar=/maven.pomderived=/true=/=/maven.pomderived=/true=/=/maven.groupId=/org.springframework=/=/maven.artifactId=/spring-context=/=/maven.version=/6.0.3=/=/maven.scope=/compile=/%3Corg.springframework.context(MessageSource.class%E2%98%83MessageSource~getMessage~Ljava.lang.String;~%5C%E2%98%83Ljava.lang.Object;~Ljava.lang.String;~Ljava.util.Locale;@arg1!0!-1!0!-1!%5C%E2%98%83Ljava.lang.Object;!-1!true%7Dorg.springframework.lang.Nullable%E2%98%82org.springframework.lang.Nullable)[Object](eclipse-javadoc:%E2%98%82=springboot/D:%5C/Vivek%5C/mavenrepo%5C/org%5C/springframework%5C/spring-context%5C/6.0.3%5C/spring-context-6.0.3.jar=/maven.pomderived=/true=/=/maven.pomderived=/true=/=/maven.groupId=/org.springframework=/=/maven.artifactId=/spring-context=/=/maven.version=/6.0.3=/=/maven.scope=/compile=/%3Corg.springframework.context(MessageSource.class%E2%98%83MessageSource~getMessage~Ljava.lang.String;~%5C%E2%98%83Ljava.lang.Object;~Ljava.lang.String;~Ljava.util.Locale;%E2%98%82java.lang.Object)[] args, **@**[**Nullable**](eclipse-javadoc:%E2%98%82=springboot/D:%5C/Vivek%5C/mavenrepo%5C/org%5C/springframework%5C/spring-context%5C/6.0.3%5C/spring-context-6.0.3.jar=/maven.pomderived=/true=/=/maven.pomderived=/true=/=/maven.groupId=/org.springframework=/=/maven.artifactId=/spring-context=/=/maven.version=/6.0.3=/=/maven.scope=/compile=/%3Corg.springframework.context(MessageSource.class%E2%98%83MessageSource~getMessage~Ljava.lang.String;~%5C%E2%98%83Ljava.lang.Object;~Ljava.lang.String;~Ljava.util.Locale;@arg2!0!-1!0!-1!Ljava.lang.String;!-1!true%7Dorg.springframework.lang.Nullable%E2%98%82org.springframework.lang.Nullable)[String](eclipse-javadoc:%E2%98%82=springboot/D:%5C/Vivek%5C/mavenrepo%5C/org%5C/springframework%5C/spring-context%5C/6.0.3%5C/spring-context-6.0.3.jar=/maven.pomderived=/true=/=/maven.pomderived=/true=/=/maven.groupId=/org.springframework=/=/maven.artifactId=/spring-context=/=/maven.version=/6.0.3=/=/maven.scope=/compile=/%3Corg.springframework.context(MessageSource.class%E2%98%83MessageSource~getMessage~Ljava.lang.String;~%5C%E2%98%83Ljava.lang.Object;~Ljava.lang.String;~Ljava.util.Locale;%E2%98%82java.lang.String) defaultMessage, [Locale](eclipse-javadoc:%E2%98%82=springboot/D:%5C/Vivek%5C/mavenrepo%5C/org%5C/springframework%5C/spring-context%5C/6.0.3%5C/spring-context-6.0.3.jar=/maven.pomderived=/true=/=/maven.pomderived=/true=/=/maven.groupId=/org.springframework=/=/maven.artifactId=/spring-context=/=/maven.version=/6.0.3=/=/maven.scope=/compile=/%3Corg.springframework.context(MessageSource.class%E2%98%83MessageSource~getMessage~Ljava.lang.String;~%5C%E2%98%83Ljava.lang.Object;~Ljava.lang.String;~Ljava.util.Locale;%E2%98%82java.util.Locale) locale) method which requires:

1. Message code – the message for which we want i18n. It is the key we defined in messages\_{locale}.properties.
2. Object[] args – the argument list to replace params in a localized message. We are passing a null value as we do not have any parameters in the message.
3. Default Message – this can be a hardcoded message if there is no key in messages\_{locale}.properties and messages.properties.
4. Locale object which consumer has passed as a header – it can be obtained from LocaleContextHolder.getLocale().

@GetMapping("/greetNewYear")

**public** String newYearGreeting() {

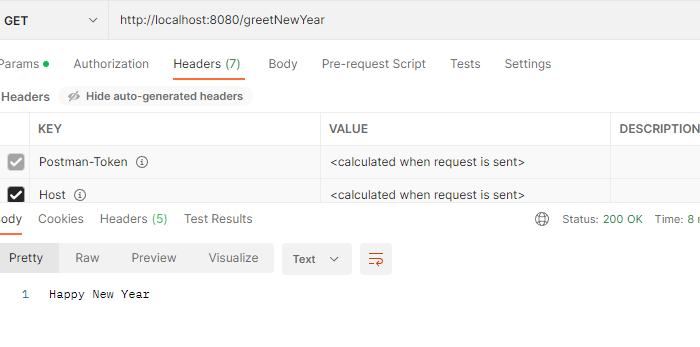
Locale locale = LocaleContextHolder.*getLocale*();

**return** messageSource.getMessage("new.year.greeting", **null**, "Happy New Year All", locale);

}

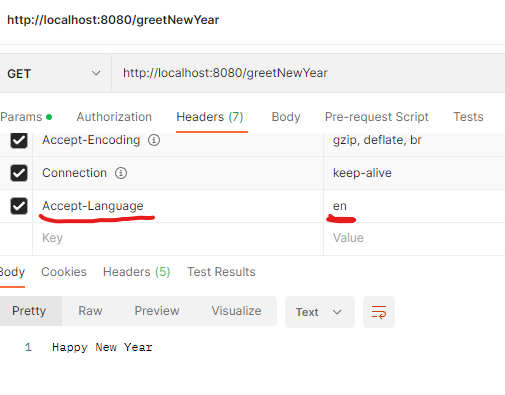
**TEST –** For Testing the i18n, let’s open postman. Below are the results in different scenarios:

1. Default message with no *Accept-Language* Header.



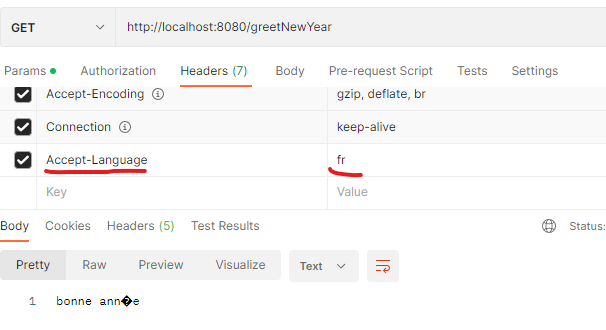
It returns a message in English from messages.properties.

1. Message with *Accept-Language: en* Header.



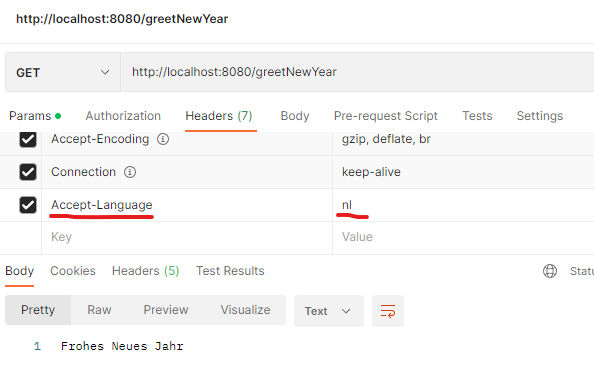
It also returns a message in English from messages.properties.

1. Message with *Accept-Language: fr* Header.



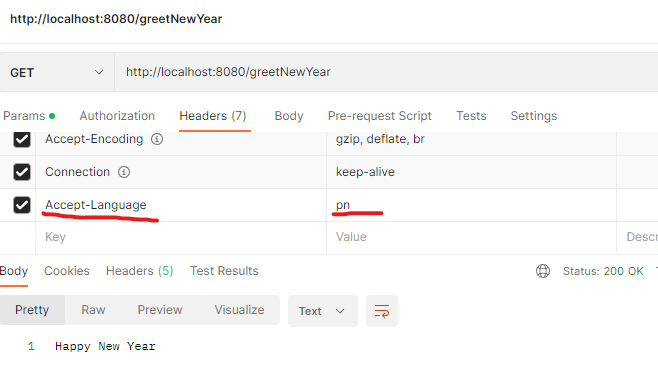
It returns a message in French from messages\_fr.properties.

1. Message with *Accept-Language: nl* Header.



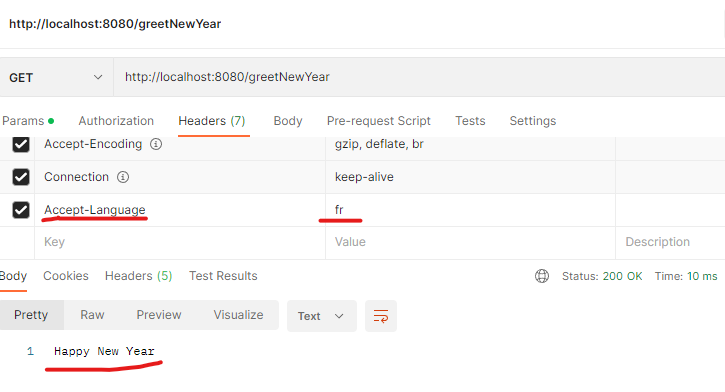
It returns a message in Dutch from messages\_nl.properties

1. Message with *Accept-Language: pn* Header (For which corresponding messages\_pn.properties is not present).



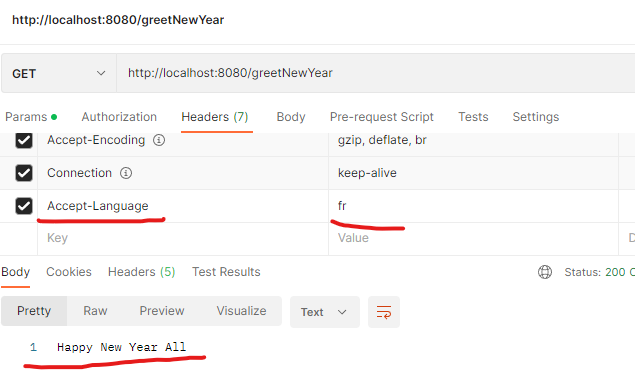
It returns a message in English from messages.properties

1. Message with *Accept-Language: fr* Header, but let’s change the key ( new.year.greeting -> new.year.greeting.msg) in messages\_fr.properties, so that the key in the newYearGreeting method does not match. This API should return a message from messages.properties.



It returns a message from the messages.properties file.

1. Message with *Accept-Language: fr* Header, but let’s change the key ( new.year.greeting -> new.year.greeting.msg) in messages\_fr.properties and messages.properties file, so that the key in the newYearGreeting method does not match. In this case, API should return the default message.



Here key in property files does not match, so API returns a default message.

**Versioning of REST API:**

Consider the case, your REST API is live in production and there is more than 1 consumer (2/10/100/1000 .. n). You need to modify the input/output parameters for API for certain requirements. In this case, you cannot directly modify the REST API in production, as all the consumers will not be able to do the modification at the same time. You cannot wait for all the consumers to complete the modifications to make the modifications in production. In this scenario, we use versioning for REST API. We can use one of the below methods for versioning:

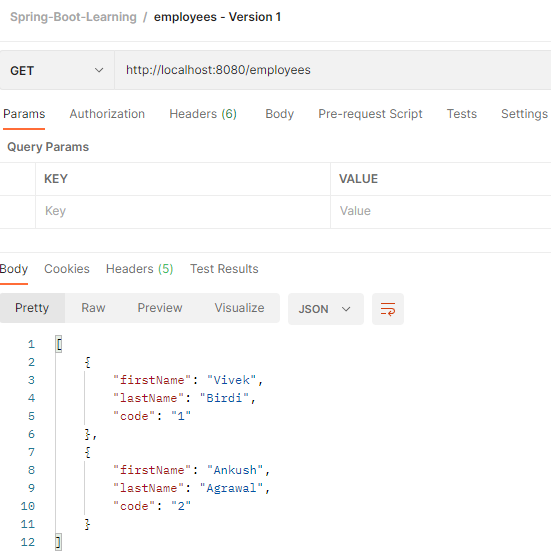
* URL based.
* Request Parameter Based.
* Header Based
* Media Type Based.

1. **URL Based:**

Using URL-based versioning, we can publish a new version of the URL with modified input/output and leave the old URL intact.

Let’s take the example of the Employee List:

**Employee List Version 1:** In version 1, we have a URL that returns a list of employees. Employee entity has three properties: firstName, lastName, and code. We have published the URL <http://server:port/employees> for version 1.



**Employee List Version 2:** Now there is a requirement that along with the employee entity, the address of the employee should also be returned. For example:

[

    {

        "firstName": "Vivek",

        "lastName": "Birdi",

        "code": "1",

        "communicationAddress": {

            "line1": "B-57",

            "line2": "DB Woods",

            "line3": "Mumbai"

        }

    },

    {

        "firstName": "Ankush",

        "lastName": "Agrawal",

        "code": "2",

        "communicationAddress": {

            "line1": "B-56",

            "line2": "DB Woods",

            "line3": "Mumbai"

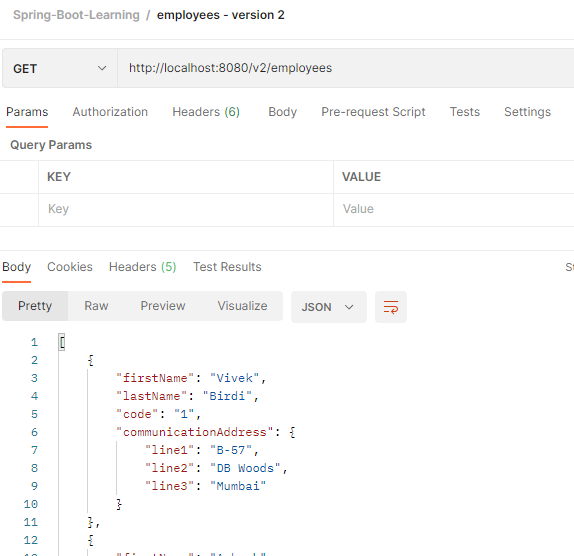
        }

    }

]

For the above modifications, if we directly make modifications in the output of <http://server:port/employees> REST API, it may cause a problem for the consumers of this API. Also, we cannot wait for all consumers to make modifications and test the same to make this amendment live in production. To solve this problem, we can publish a new version of REST API with a versioned URL (e.g <http://server:port/v2/employees>) which will return a modified response.

**(Please refer EmployeeController in the source code and dependent classes for REST API versions)**



1. **Request Parameter Based:** In this method, we use request parameters to differentiate the different versions of the REST API. URI remains the same for all the versions. We will take the same example as in URL-based versioning.
   1. <http://server:port/employees/rp?version=1> It will return the list of employees without their addresses. We need to add mapping and a new method in the Employee controller.

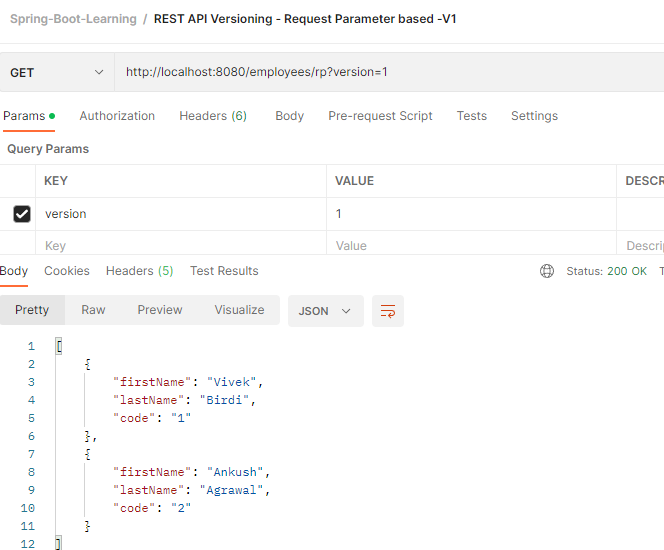
@GetMapping(path = "/employees/rp", params = "version=1")

**public** List<Employee> getEmployeeListRPV1(){

**return** employeeDao.findAll();

}

On the client side, you need to pass *version=1* in the URL.



* 1. <http://server:port/employees/rp?version=2> It will return the list of employees with their addresses. We need to add mapping and a new method in the Employee controller.

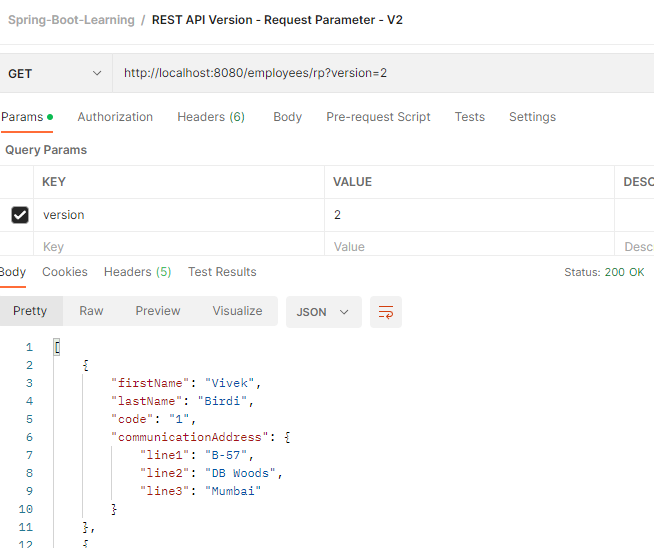
@GetMapping(path = "/employees/rp", params = "version=2")

**public** List<EmployeeV2> getEmployeeListRPV2(){

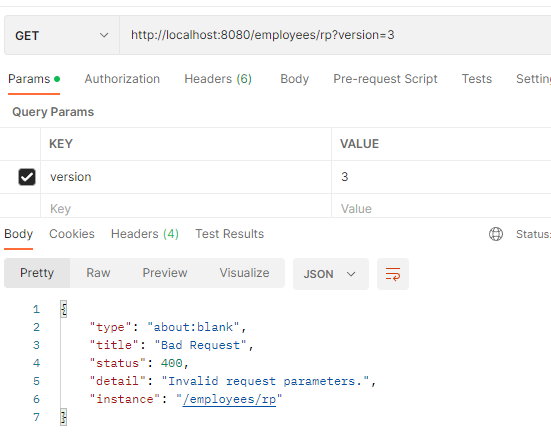
**return** employeeV2Dao.findAll();

}

On the client side, you need to pass *version=2* in the URL.

****

**Note: We will get a bad request error if we pass an incorrect version in the request URL. For example, we do not have mapping for version=3. So if we try with the URL** [**http://server:port/employees/rp?version=2**](http://server:port/employees/rp?version=2)**, we will get a Bad Request error.**

****

1. **Custom Header based:** In this method, we will use custom headers with the same URI. Let’s work with the same example. To enable Header-based REST API Versioning, we need to add mappings in the EmployeeController.
   1. [http://server:port/employees/header](http://server:port/employees/rp) – (Header: *api-version=1*) It will return the list of employees without their addresses. We need to add mapping and a new method in the Employee controller.

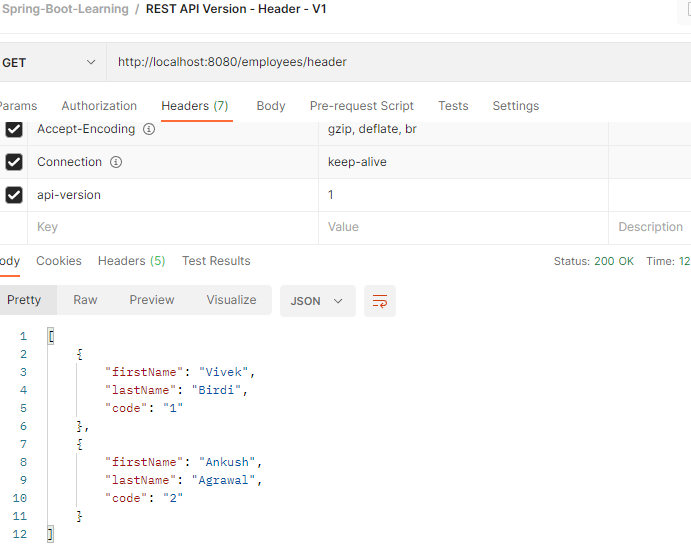
@GetMapping(path = "/employees/header", headers = "api-version=1")

**public** List<Employee> getEmployeeListHeaderV1(){

**return** employeeDao.findAll();

}

On the client side, we need to header *API-version=1* in the request:



* 1. <http://server:port/employees/header> – (Header: *api-version=2*) It will return the list of employees with their addresses. We need to add mapping and a new method in the Employee controller.

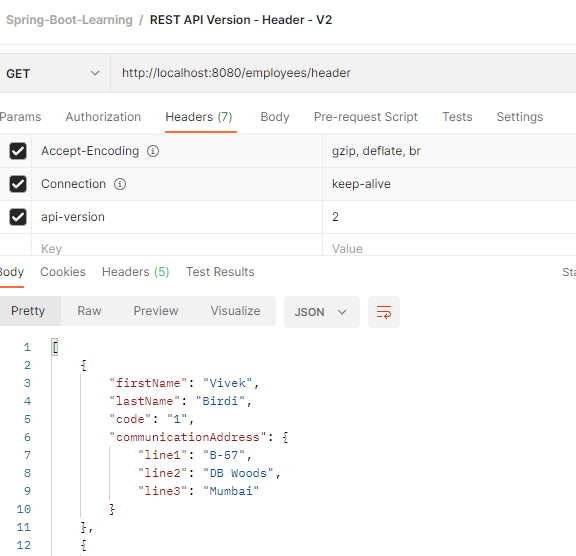
@GetMapping(path = "/employees/header", headers = "api-version=2")

**public** List<EmployeeV2> getEmployeeListHeaderV2(){

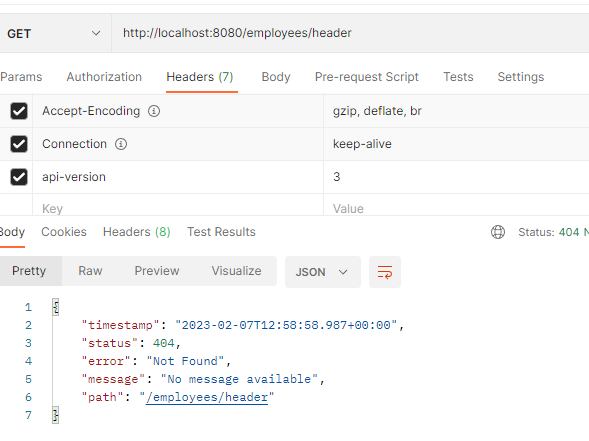
**return** employeeV2Dao.findAll();

}

On the client side, we need to header *API-version=2* in the request:



**Note: We will get a 404 error if we pass an incorrect value in the request header (api-version). For example, we do not have mapping for *api-version=3*. So if we send *api-version=3* as the header value, we will get a 404 error:**



1. **Media Type based:** In this method, we will use content-type negotiation (Accept header) with the same URI. Let’s work with the same example. To enable Header-based REST API Versioning, we need to add mappings in the EmployeeController.
   1. <http://server:port/employees/accept> – (Accept: *application/api-version-1+json*) It will return the list of employees without their addresses. We need to add mapping and a new method in the Employee controller.

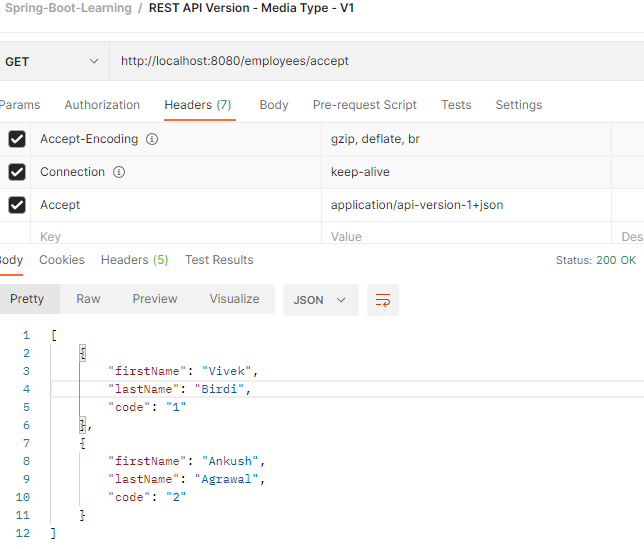
@GetMapping(path = "/employees/accept", produces = "application/api-version-1+json")

**public** List<Employee> getEmployeeListAcceptV1(){

**return** employeeDao.findAll();

}

On the client side, we need to add the header Accept: *application/api-version-1+json* in the request:



* 1. <http://server:port/employees/accept> – (Accept: *application/api-version-2+json*) It will return the list of employees with their addresses. We need to add mapping and a new method in the Employee controller.

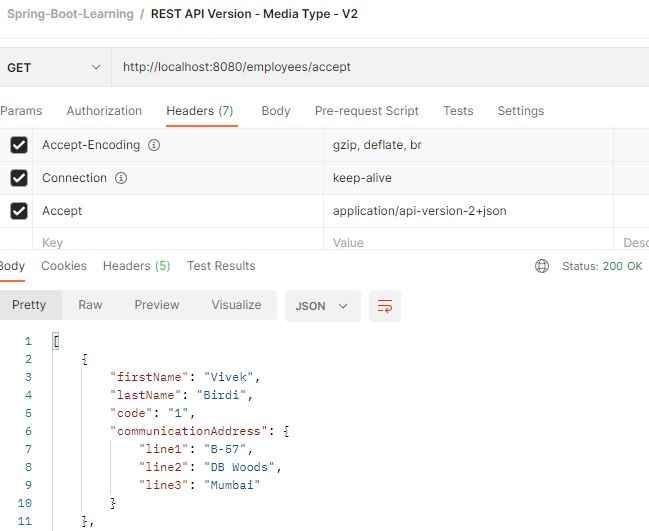
@GetMapping(path = "/employees/accept", , produces = "application/api-version-1+json")

**public** List<EmployeeV2> getEmployeeListAcceptV2(){

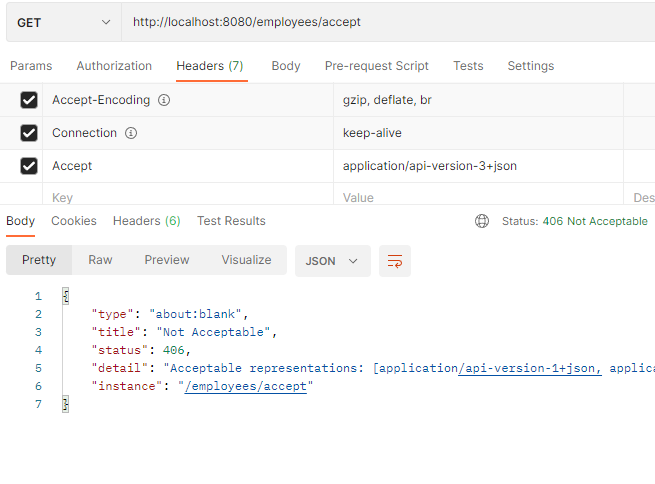
**return** employeeV2Dao.findAll();

}

On the client side, we need to add the header Accept: *application/api-version-2+json* in the request:



**Note: We will get a 406 error (Not Acceptable) if we pass an incorrect value in the request header (Accept: ). For example, we do not have mapping for Media Type** *application/api-version-3+json***. So if we send** Accept: *application/api-version-3+json* **as the header value, we will get a 406 error:**



**Evaluating the approaches to use for versioning:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | URL Versioning | Request Param Versioning | Custom Header Versioning | Versioning |
| **URI Pollution** | Yes | Yes | No | No |
| **Miss use of HTTP Headers** | No | No | Yes | Yes |
| **Caching** | Simple | Simple |  |  |
| **Request From Browser** | Yes (GET Req) | Yes (GET Req) | No | No |
| **API Documentation using Tools** | Simple | Simple | The tool | The tool Might not support |
|  |  |  |  |  |

**HATEOAS:** Hyper Media As Engine Of Application State.

When we require to inform the consumer about subsequent actions in response to a REST API, we can do it in two ways: Manually or Using HATEOAS. Let’s take the case of /users/{userId} mapping. It returns the response below:

{

    "userId": 1,

    "name": "Vivek Birdi",

    "dateOfBirth": "1987-02-08"

}

Below is the current implementation for getUser in UserContoller:

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

**public** User getUser(@PathVariable ("userId") **int** userId) {

User user = userDao.find(userId);

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

**throw** **new** UserNotFoundException("Requested user does not exist");

}

**return** user;

}

Now, we want to add a link to the subsequent action as below, which consumers can perform in the response of REST API.

{

    "userId": 1,

    "name": "Vivek Birdi",

    "dateOfBirth": "1987-02-08",

    "\_links": {

        "all\_users": {

            "href": "http://localhost:8080/users"

        }

    }

}

This can be achieved either manually by modifying the *User* Entity, or SpringBoprovidingdes the support for the HATEOAS. To enable HATEOAS support SpringBoot, we need to add *spring-boot-starter-hateoas* starter dependency in pom.xml

<dependency>

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

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

</dependency>

After adding the required dependency, we need to make changes in the code. We need to use *EntityModel* and *WebMvcLinkBuilder* Classes. I will create a new version of the getUser method in UserContoller for this.

@GetMapping("/v2/users/{userId}")

**public** EntityModel<User> getUserV2(@PathVariable ("userId") **int** userId) {

User user = userDao.find(userId);

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

**throw** **new** UserNotFoundException("Requested user does not exist");

}

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

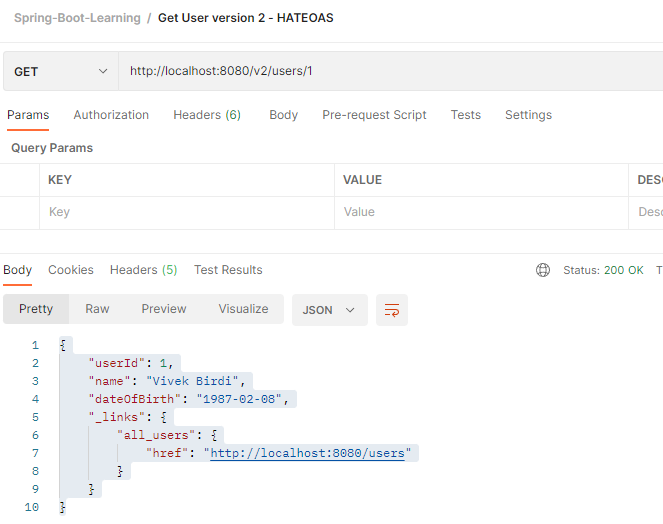
WebMvcLinkBuilder link = WebMvcLinkBuilder.*linkTo*(WebMvcLinkBuilder.*methodOn*(getClass()).getUsers());

userModel.add(link.withRel("all\_users"));

**return** userModel;

}

Now test it using Postman.



You can see, the link to subsequent action in the output. Please note that EntityModel cannot process the collection object, so we need to design the response object accordingly.

**Customize REST API Response:**

In some cases, we require to customize the output from REST API. We may require to modify the field names in response or we may require to filter the fields from the response.

Spring converts POJOs to JSON/XML format (Serialization) using Jackson data formmater. Let’s take the example of an Account Entity:

**public** **class** Account {

**private** String accountNo;

**private** String name;

**private** String pin;

**private** String password;

**private** String bankName;

**private** String bankCode;

**private** String balance;

// Getters and Setter

}

When returning the object of this entity Spring will serialize it into JSON/XML using jackson formmater:

{

    "accountNo": "1",

    "name": "Vivek",

    "pin": "1000",

    "password": "password",

    "bankName": "Swiss Bank",

    "bankCode": "SB",

    "balance": "100"

}

Or when we return collection in response:

[

    {

        "accountNo": "1",

        "name": "Vivek",

        "pin": "1000",

        "password": "password",

        "bankName": "Swiss Bank",

        "bankCode": "SB",

        "balance": "100"

    },

    {

        "accountNo": "2",

        "name": "Ankush",

        "pin": "1001",

        "password": "password",

        "bankName": "Cooperative Bank",

        "bankCode": "COPB",

        "balance": "10000000000"

    }

]

We may need to customize the above response, such as modifying the field names or filtering out sensitive fields from the response.

1. **To modify field names in response:** Consider the case where we need to modify the *name* property in the output as *accountName*. This can be achieved by using annotation @JSONProperty.

@JsonProperty("accountName")

**private** String name;

As a result, the *name* field will be returned as *accountName* in the response.

{

    "accountNo": "1",

    "pin": "1000",

    "password": "password",

    "bankName": "Swiss Bank",

    "bankCode": "SB",

    "balance": "100",

    "accountName": "Vivek"

}

1. **Filtering:** Entity may contain sensitive fields, which we do not want to return in response such as *pin* or *password*. We need to filter the response. Filtering is of two types:
   1. **Static Filtering:** In static filtering, we use @JsonIgnore at the field level or @JsonIgnoreProperties at the class level.

@JsonIgnoreProperties("password")

**public** **class** Account {

………………………….

@JsonIgnore

**private** String pin;

It will filter out the *pin* and *password* fields from the response.

For Single Object:

{

    "accountNo": "1",

    "bankName": "Swiss Bank",

    "bankCode": "SB",

    "balance": "100",

    "accountName": "Vivek"

}

For Collection:

[

    {

        "accountNo": "1",

        "bankName": "Swiss Bank",

        "bankCode": "SB",

        "balance": "100",

        "accountName": "Vivek"

    },

    {

        "accountNo": "2",

        "bankName": "Cooperative Bank",

        "bankCode": "COPB",

        "balance": "10000000000",

        "accountName": "Ankush"

    }

]

Static filtering is used when we want to remove the fields from the response at the application level. It will remove the fields in response across all the REST APIs. Also note that for serialization, it is recommended to use @JsonIgnore at the field level, as @JsonIgoreProperties requires the field names as arguments. If in the future we change the name of the field in the entity, we need to modifications in @JsonIgnoreProperties too.

* 1. **Dynamic Filtering:** We may need to send different attributes of an entity from different REST APIs. In such a scenario, we use dynamic filtering. We need to make use of *MappingJacksonValue* and *@JsonFilter(“{BeanFilterName}”)* from rom the Jackson data formatter. Consider the case of the Account entity.
     1. From one API, say /v2/accounts we need to filter out the balance filed from the response.

Below is the source code to apply the dynamic filter:

@GetMapping("/v2/accounts")

**public** MappingJacksonValue getAccountsV2(){

List<Account> accounts = accountDao.getAccounts();

MappingJacksonValue filteredResponse = **new** MappingJacksonValue (accounts);

// Property Filter Object

SimpleBeanPropertyFilter filter = SimpleBeanPropertyFilter.*filterOutAllExcept*("accountNo","accountName","bankName", "bankCode","pin","password"); // The list of fields required in response, note that we need to use the field name passed in @JsonProperty annotation in case we have used this annotation. E.g. accountName.

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

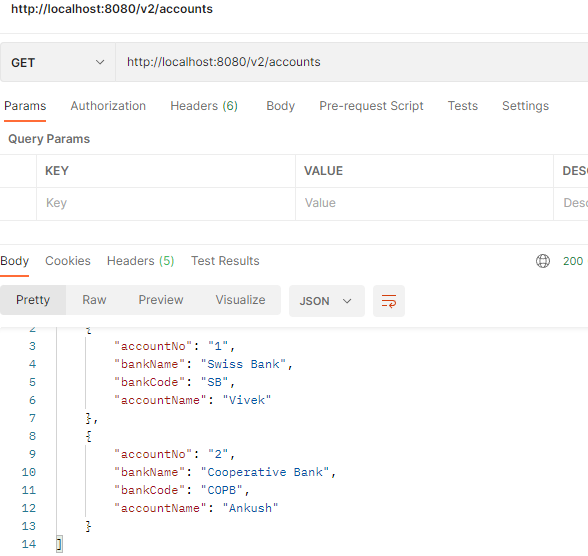
filteredResponse.setFilters(filterProvider);

**return** filteredResponse;

}

Also, we need to apply *@JsonFilter(“AccountBeanFilter”)* on the Account entity.

**Output: The balance field is not in the response.**

****

* + 1. From other API, say /v3/accounts we need to filter our bankName and bankCode fields from the response.

@GetMapping("/v3/accounts")

**public** MappingJacksonValue getAccountsV3(){

List<Account> accounts = accountDao.getAccounts();

MappingJacksonValue filteredResponse = **new** MappingJacksonValue (accounts);

/\* Property Filter Object

\* The list of fields required in response.

\* Note that we need to use the field name passed in the @JsonProperty annotation in case we have used this annotation.

\* E.g. accountName. \*/

SimpleBeanPropertyFilter filter = SimpleBeanPropertyFilter.*filterOutAllExcept*("accountNo","accountName","balance","pin","password");

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

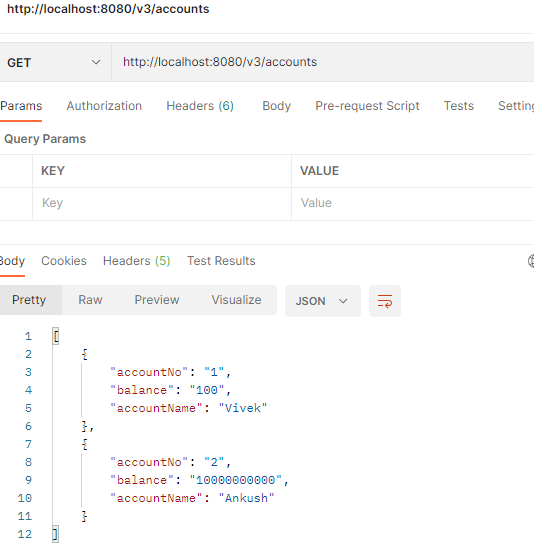
filteredResponse.setFilters(filterProvider);

**return** filteredResponse;

}

Also, we need to apply *@JsonFilter(“AccountBeanFilter”)* on the Account entity.

**Output: bankName and bankCode fields are not in the response.**

****

**Note: Statically filtered fields will not appear in response, as you can see pin and password fields from the Account entity is not there is response.**

**Monitoring:**

Spring-boot provides the support for the application monitoring. We need to add *spring-boot-starter-actuator* in pom.xml and add below line in application.properties

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

Spring Now you can start server and browse the URL <http://server:port/actuator>. You can see many URLs in response which can browse for monitoring such as Head dump, Thread Dump, List of Spring managed beans, Cache, Health etc.

