Enterprise Java with Spring
Spring REST API
Lab 2

1 Lab setup

Make sure you have the following items installed

- Latest LTS JDK version (at this point: JDK 21)
- Spring Tool Suite (STS) or IntelliJ IDEA
- Latest version of Maven (at this point: Maven 3.9.9)
- A free account at Postman and installed the Postman app
- A suitable text editor (Notepad ++)
- A utility to extract zip files (7-zip)

In each of the main lab folders, there are two subfolders: `changes` and `final`. The `changes` subfolder just holds the source code files for the lab, while the `final` subfolder holds the complete Eclipse project starting from its project root folder. We will use the code from the `changes` subfolder to build up our applications from scratch and you can always fall back on the complete Eclipse project if you encounter any errors while building up the application.

2 REST HTTP methods and response types

The source code for this lab is found in `Rest-Methods/changes` folder.

Start up STS. Ensure you are in the Java EE perspective.

Go to File -> New -> Other -> Spring Boot -> Spring Starter Project. Complete it with the following details:

Name: `RestMethodsResponse`
Group: `com.workshop.rest`
Artifact: `RestMethodsResponse`
Version: `0.0.1-SNAPSHOT`
Description: `Demo Spring Rest methods and response types`
Package: `com.workshop.rest`

Click Next.

Add the following dependencies:
`Web -> Spring Web`
`Developer Tools -> Spring Boot Dev Tools`

Add this new dependency to the new project `pom.xml`

```
<!-- Required for processing and returning XML content-->
<dependency>
	<groupId>com.fasterxml.jackson.dataformat</groupId>
	<artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

Right click on the project, select Maven -> Update Project, and then click on the project name and then refresh.

This dependency (in addition to the standard Spring Boot starters) is required for the app to be able to return XML content in addition to standard JSON content. Note that this dependency already exists in the dependency hierarchy, so you don't have to specify the version number.

In the package `com.workshop.rest` in `src/main/java`, place the files

`Employee`
`Resume`
`EmployeeController`

`RestMethodsResponseApplication`

EmployeeController uses @Autowired to initialize its two fields `myEmployee` and `myResume`; the initialization code for this is available in RestMethodsResponseApplication via the @Bean factory methods. This is possible because the @SpringBootApplication annotation incorporates 3 other annotations implicitly (@Configuration, @EnableAutoConfiguration and @ComponentScan).

Notice that we have path mappings for all common REST HTTP methods (Get, Post, Put, Delete) with the exception of Patch (which we will look at in a later lab session).

The MIME (or media) type for the response can be specified along with the path mapping as a string (`"application/json", "application/xml",` etc). If none is specified, the default is JSON (`"application/json"`). This can be specified explicitly if desired using the `produces` attribute and the corresponding String value. Alternatively, we can use the MediaType constants provided by Spring framework:

Similarly, we can also specify the type of data in the HTTP Request using the `consumes` attribute.

It is possible to specify more than one type (usually it will be JSON and XML) for the return or request type in the body. Different types can be mapped to the same endpoints but handled by different mapping methods. Alternatively, you can have a single method being mapped to different types for the same endpoint.

2.1 Testing GET, POST, PUT and DELETE

We will test all the REST methods that are mapped in EmployeeController using Postman.

Start Postman and create a new collection to save your requests: `REST lab requests`

Start up the app in the usual manner.

Right click on the project entry in the Project explorer pane and then select Run As -> Spring Boot App, or right click on the project entry in Spring Boot dashboard (Window -> Show View -> Other -> look in the Other folder -> Boot Dashboard) and select Start / Restart.

You can also use the Eclipse TCP/IP monitor to help out in debugging any issues that arise.

Create new GET requests to:

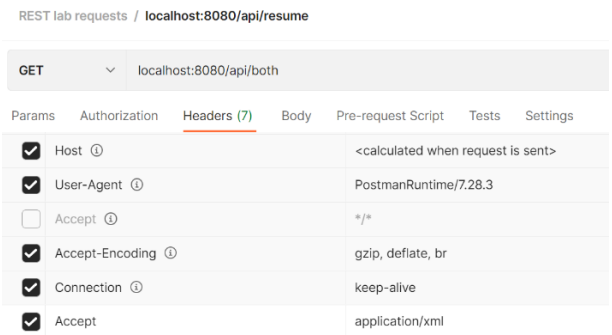
```
localhost:8080/api/resume
localhost:8080/api/employee
```

Verify the JSON content is returned and the appropriate log messages corresponding to the mapped methods appear in the console view.

Create a new GET request to:

```
localhost:8080/api/both
```

Set up the Accept header with a new value of `application/xml` and unselect the preset header with the value of */*



Send the request and verify the content returned is XML and the appropriate log messages corresponding to the mapped methods appear in the console view.

For the same request, modify the Accept header now to `application/json`.
Send the request and verify the content returned is JSON and the appropriate log messages corresponding to the mapped methods appear in the console view.

Create a new POST request to:

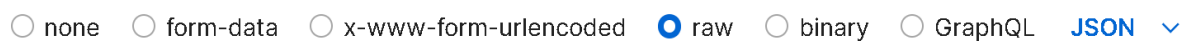
`localhost:8080/api/resume`

Set the body to `raw` with XML format and enter this as its content:


```
<Resume>
    <university>Oxford</university>
    <spokenLanguages>
        <spokenLanguages>German</spokenLanguages>
        <spokenLanguages>Malay</spokenLanguages>
    </spokenLanguages>
    <skillSets>Fantastic project manager</skillSets>
    <yearsExperience>3</yearsExperience>
    <cgpa>3.25</cgpa>
</Resume>
```

Send the request and verify that the log messages corresponding to the mapped methods appear in the console view showing the content received and deserialized correctly.

Set the body to `raw` with JSON format and enter this as its new content:



```
{
    "university": " IIT",
    "spokenLanguages": [
        "Hindi",
        "Tamil"
    ],
    "skillSets": "Cool CEO",
    "yearsExperience": 15,
    "cgpa": 3.99
}
```

Send the request and verify that the log messages corresponding to the mapped methods appear in the console view showing the content received and deserialized correctly.

The @RequestBody annotation on the `res` parameter in `mapSingleEmployee` binds the HTTP request body data to this parameter, which tells Spring to deserialize the incoming HTTP request body to the specified Java object type (in this case a Resume object). This deserialization is typically handled to JSON / XML is handled by Jackson, the dependency that we included earlier in our POM.

Create a new POST request to:

`localhost:8080/api/employee`

Set the body to `raw` with JSON format and enter this as its content:

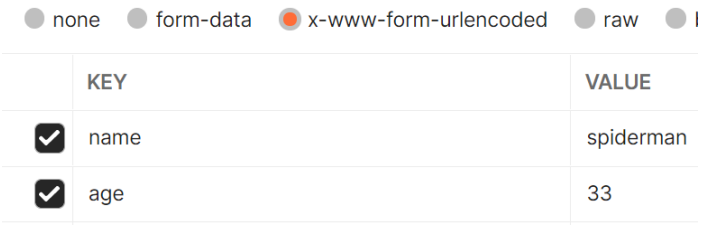
```
{
    "name": "Spiderman",
    "age": 22,
    "resume": {
        "university": "Marvel Universe",
        "spokenLanguages": [
            "English",
            "Chinese",
            "Russian"
        ],
        "skillSets": "Great webslinger",
        "yearsExperience": 2,
        "cgpa": 4.0
    },
    "married": false
}
```

Send the request and verify that the log messages corresponding to the mapped methods appear in the console view showing the content received and deserialized correctly.

Create a new POST request to:

`localhost:8080/api/vals`

Set the body to `x-www-form-urlencoded` and enter some random key-value pairs:



Send the request and verify that the log messages corresponding to the mapped methods appear in the console view showing the content received and deserialized correctly.

The type `x-www-form-urlencoded` is used for HTML form data submitted via a POST request. Notice that we now need to use @RequestParam and a Map structure to extract the data, rather than deserializing into a specific class structure using @RequestBody.

Create a new PUT request to:

`localhost:8080/api/resume`

Set the body to `raw` with JSON format and enter this as its content:

```
{
    "university": " IIT",
    "spokenLanguages": [
        "Hindi",
        "Tamil"
    ],
    "skillSets": "Cool CEO",
    "yearsExperience": 15,
    "cgpa": 3.99
}
```

Send the request and verify that the log messages corresponding to the mapped methods appear in the console view showing the content received and deserialized correctly.

Create a new DELETE request to:

`localhost:8080/api/resume`

Select `None` for the body content.

Send the request and verify that the log messages corresponding to the mapped methods appear in the console view.

Notice that for all cases so far, successful invocation of the mapped method in the @RestController class returns a status 200 OK by default.

2.2 Common client-side error messages

Lets examine some of the messages that are returned by default by the Spring framework due to client-side errors. The message typically includes a timestamp, a HTTP status code and error message and the path portion of the URL.

The most common client-side HTTP error messages are:

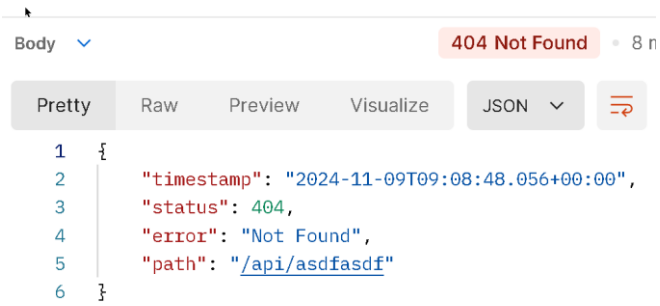
- 404 Not Found
- 405 Method Not Allowed
- 400 Bad Request
- 415 Unsupported Media Type

There are Spring MVC exceptions that correspond to these errors

Try to send a GET, POST, PUT or DELETE request to a URL with no matching mapping, for e.g:

```
localhost:8080/api/asdfasdf
```

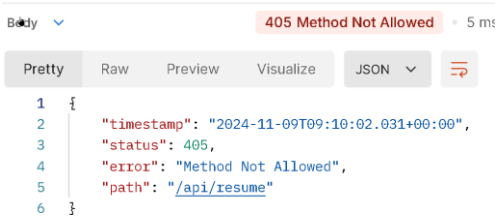
Verify a 404 Not Found error message is received in return. Note that the log output on the service does not show anything as this error is intercepted and handled by the embedded Tomcat server itself.



Try to send a PATCH request to this URL:

```
localhost:8080/api/resume
```

Verify a 405 Method not allowed error message is received in return. Check the log output on the service to see the warning message displayed.



Try to send a POST request to:

```
localhost:8080/api/resume
```

with the raw JSON content of

```
{
    "city": "Metropolis",
    "population": 10000
}
```

Although a 200 OK Status is received, notice that the log messages on the server side show that deserialization results in an object with null fields.

This is because the structure of the JSON content does not match the structure of the Java object that is bound to exactly, which results in the failure of the deserialization process.

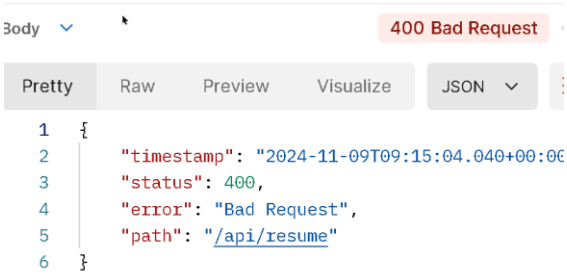
Resend the POST request to:

`localhost:8080/api/resume`

with the malformed raw JSON content of

```
{
    "city : "Metropolis",
    "population": 10000
}
```

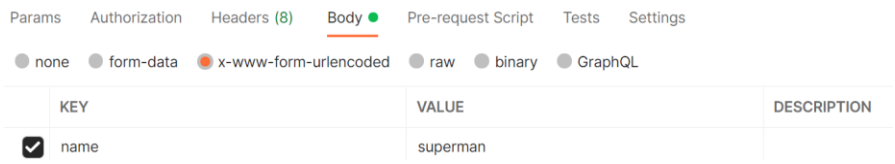
Verify a 400 Bad Request error message is received in return. Check the log output on the service to see the warning message displayed.



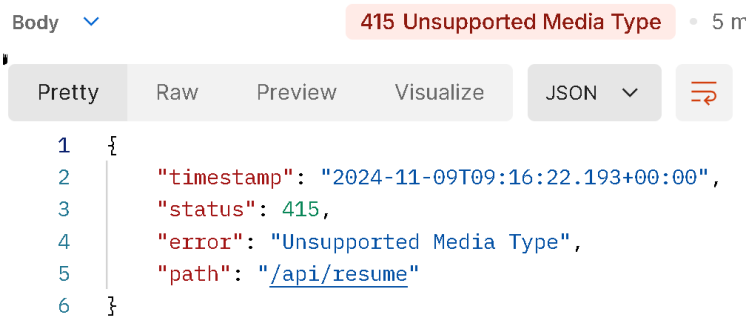
Resend the POST request to:

`localhost:8080/api/resume`

in the `x-www-form-urlencoded` format with a dummy key and value pair:



Verify a 415 Unsupported Media Type error message is received in return. Check the log output on the service to see the warning message displayed.



The reason for this is that the @PostMapping handler method for that API endpoint (`/resume`) is not specified to consume this kind of incoming media type (`x-www-form-urlencoded`), which results in error code 415

3 Accessing Path and Query parameters

In the package `com.workshop.rest` in `src/main/java`, place the file

`ParamsController`

Restart the app

Notice that we can have multiple @RestController classes with the same top level request mapping (in this case both ParamsController and EmployeeController have the top level @RequestMapping of `/api`)as long as

- they provide unique request mappings for their individual hander methods and
- they are in same package (or subpackage) that the @SpringBootApplication class is in.

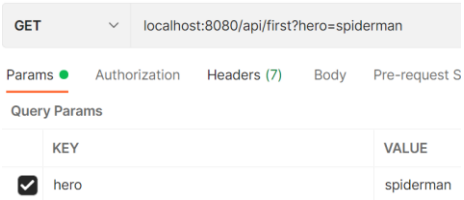
In this session, we will demonstrate the variety of ways to access path and query parameters. This is demonstrated using only @GetMapping methods, but they are equally applicable to all standard REST HTTP method mappings.

3.1 Accessing query parameters via @QueryParam

Make a GET request to:

`localhost:8080/api/first?hero=spiderman`

You can either type the query string directly or set it up via the Params tab in Postman.



Verify that the value is displayed correctly in the log output on the server side

Make a GET request to:

`localhost:8080/api/second?person=superman&age=33`

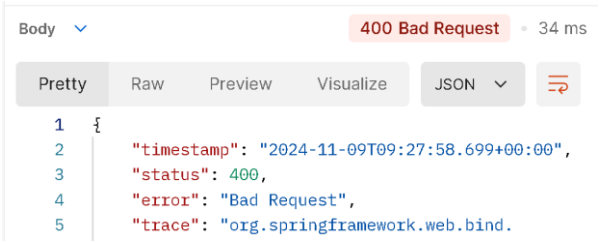
Verify that the values are displayed correctly in the log output on the server side

Make a GET request to:

`localhost:8080/api/first`

without any query string parameters.

Notice that a 400 Bad Request is returned as the mapping method expects a compulsory query parameter with the key `hero`.



Make a GET request to:

`localhost:8080/api/third?hero=spiderman`

Verify that the value is displayed correctly in the log output on the server side

Repeat the GET request without the parameter:

`localhost:8080/api/third`

Verify that the correct statement appears in the log output on the server side

This time we don't have a 400 Bad Request returned as the mapping method specifies that the query parameter is optional.

Make a GET request to:

`localhost:8080/api/fourth?hero=spiderman`

Verify that the value is displayed correctly in the log output on the server side

Make a GET request to:

`localhost:8080/api/fourth`

Verify that the default parameter value is displayed correctly in the log output on the server side

Make a GET request to:

```
localhost:8080/api/fifth?person=superman&age=33&job=journalist
```

Verify that the 3 key-value pairs are displayed correctly in the log output on the server side

Make a GET request to:

```
localhost:8080/api/sixth?heroes=ironman,black widow,thor
```

Verify that the 3 values for `heroes` are displayed correctly in the log output on the server side

3.2 Accessing path parameters via @PathVariable

Make GET requests to:

```
localhost:8080/api/seventh/ironman
localhost:8080/api/seventh/superman
```

Verify that the values are displayed properly at the server side

Make GET requests to:

```
localhost:8080/api/eighth/ironman/22
localhost:8080/api/eighth/black-widow/35
```

Verify that the values are displayed properly at the server side

Make a GET request to:

```
localhost:8080/api/ninth/Jane/manager/female
```

Verify that the values are displayed properly at the server side

Make a GET request to:

```
localhost:8080/api/ninth/Jane/manager
```

Notice a 404 Not Found error is returned. This is because the API endpoint mapping as specified in the code expects 3 path portions after the /ninth (`/ninth/{name}/{job}/{gender}`) but only 2 path portions were supplied.

Make a GET request to:

```
localhost:8080/api/tenth/Jane
```

Verify that the value is displayed properly at the server side

Make a GET request to:

```
localhost:8080/api/tenth
```

Verify that the correct statement appears in the log output on the server side

There are no default values for @PathVariable, unlike @RequestParam. Default values will have to be explicitly included in the logic of the path processing code.

4 Setting HTTP response status codes

In the package `com.workshop.rest` in `src/main/java`, place the file

`ResponseController`

4.1 Using @ResponseStatus and ResponseEntity

So far, all of the successfully executed request mapping methods return an implicit 200 OK status. We can also choose to provide other HTTP status codes in the response. There are two ways to do this (using @ResponseStatus and the ResponseEntity object) to be used in conjunction with the available status codes in Spring.

Note that this is only a subset of the full range of HTTP status codes officially available:

Make a GET request to:

`localhost:8080/api/status-one`

and verify a 208 Already Accepted status code is returned. Experiment with changing to other status codes and repeat the GET request.

Make a GET request to:

`localhost:8080/api/status-two`

and verify a 301 Moved Permanently status code is returned. Experiment with changing to other status codes and repeat the GET request.

5 Accessing HTTP Headers in request and response

In the `com.workshop.rest` package in `src/main/java`, place the file

`HeaderController`

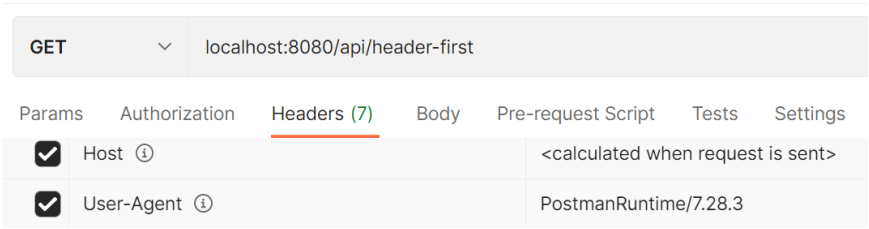
5.1 Retrieving request headers with @RequestHeader

We demonstrate setting request headers here using only @GetMapping methods, but it works exactly the same for any of the other standard REST HTTP methods.

Make a GET request to:

`localhost:8080/api/header-first`

and make sure that the Host and User-Agent header values in the Headers are checked (this is usually the default) before sending the request.



Verify that these header values appear in the log output on the server side.

Make a GET request to:

`localhost:8080/api/header-second`

Verify that the appropriate header values appear in the log output on the server side.

We can access any of the [HTTP header values we want] using the appropriate get method.

Experiment by adding in extra code to extract other header values that might be present in the list of headers shown in Postman.

Make a GET request to:

`localhost:8080/api/header-third`

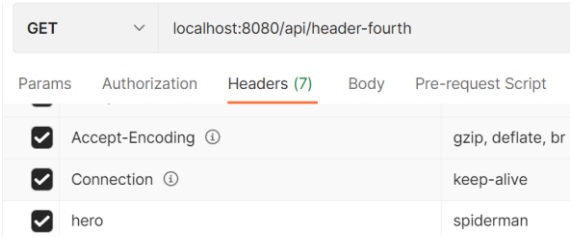
Verify that all header values appear in the log output on the server side.

Make a GET request to:

`localhost:8080/api/header-fourth`

Verify that the appropriate statement appears in the log output on the server side

Add a header for `hero` with an appropriate value in the Headers view and repeat the request.



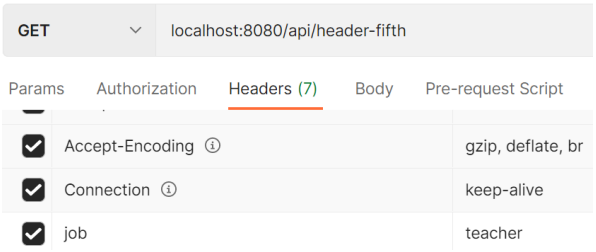
Verify that the header value appears in the log output on the server side

Make a GET request to:

`localhost:8080/api/header-fifth`

Verify that the default value for the `job` header appears in the log output on the server side

Add a custom value for the `job` header and repeat the request.



Verify that the set value for the `job` header now appears in the log output on the server side

5.2 Adding response headers with HttpServletResponse and ResponseEntity

In the `com.workshop.rest` package in `src/main/java`, place the file

`AddResponseHeaderFilter`

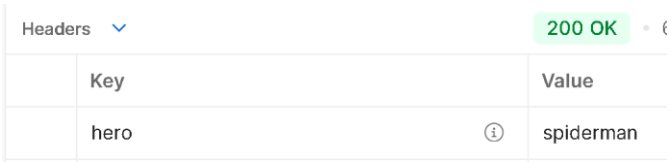
and make the change:

`RestMethodsResponseApplication-v2`

Make a GET request to:

`localhost:8080/api/header-sixth`

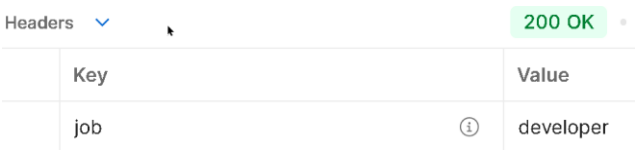
Check in the response view in Postman for the custom `hero` header that was added



Make a GET request to:

`localhost:8080/api/header-seventh`

Check in the response view in Postman for the `job` header.



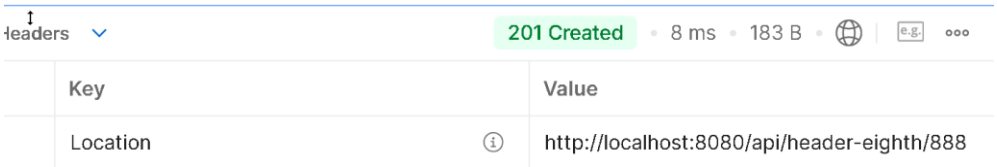
A @PostMapping method will typically persist/store the JSON content sent to it from the client. To help the client fetch this stored content in the future, it will also typically return a URL which can be used with a future GET request for this purpose. This URL is typically provided as the value of the `Location` header in the response from the @PostMapping method and the response is returned with a status code of 201 Created.

Here, we pretend that a new resource has been created which is accessible at the current path URL with the resource ID appended to its end and return this in the `Location` header

Make a POST request to:

```
localhost:8080/api/header-eighth
```

Check in the response view in Postman for the `Location` header.



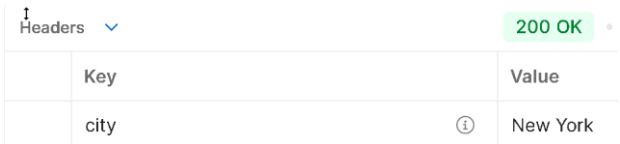
We can add filters to process incoming requests or responses, so that certain actions are performed to specific requests before they are processed by the server or to specific responses before they are returned to the client. This helps us to avoid repeating code to add response headers to all our relevant handler methods in a @RestController class.

This can be done by implementing a Filter (AddResponseHeaderFilter) which is then registered as a FilterRegistrationBean in the @SpringBootApplication class.

Make GET requests to:

```
localhost:8080/api/filter-header/first
localhost:8080/api/filter-header/second
```

Check in the response view in Postman for the `city` header.



6 Handling exceptions in REST

In the `com.workshop.rest` package in `src/main/java`, place the files

```
AnotherCustomException
ResourceNotFoundException
SampleController
```

In `src/main/resources`, add this additional entry to `application.properties`

```
server.error.include-message=always
```

So far, we have seen a variety of client side errors resulting in a variety of error messages returned back from the server, such as:

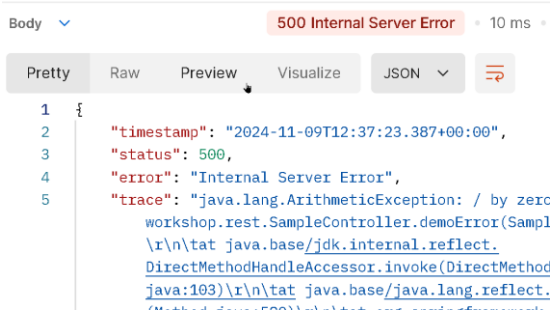
- 404 Not Found
- 405 Method Not Allowed
- 400 Bad Request
- 415 Unsupported Media Type

Runtime errors which can potentially cause the Spring REST app to crash will result in a variety of exceptions in any @RestController methods. If these exceptions are not caught and handled appropriately, a generic 500 Internal Server Error is returned.

Make a GET request to:

`localhost:8080/api/demo-error`

Check the body of the response for the 500 Internal Server Error and the customized error message and verify the occurrence of the Divide by zero ArithmeticException on the server side.



6.1 Creating a custom exception and message with @ResponseStatus

We can provide a custom RuntimeException class of our own (ResourceNotFoundException) with a specific HTTP status code (404 Not Found) and error message (via the `reason` attribute of @ResponseStatus).

The typical use case for this is when the API consumer sends a HTTP GET request is in valid format which can be processed correctly by the Spring framework. However, the request specifies a resource that does not actually exist on the server, so the app must provide a response indicating this situation (which is considered an error) to the API consumer.

The handler method processing this specific HTTP GET request will therefore throw this custom RuntimeException when the code logic in it detects this issue with the incoming HTTP Request that we just mentioned above.

Make a GET request to:

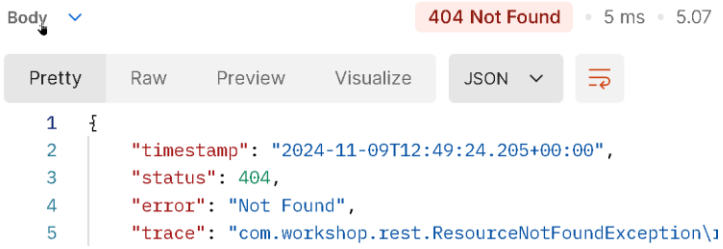
`localhost:8080/api/firstdemo/10`

Note that a valid Resume object is returned. This is simulating the situation where the last path portion specifies the ID of the resource to be returned, and this particular resource does actually exist on the server.

Make a GET request to:

`localhost:8080/api/firstdemo/200`

This is simulating the situation where the last path portion specifies the ID of the resource to be returned, and this particular resource does NOT EXIST on the server. Here, we return a standard 404 Not found error message with corresponding status code as a result of our custom user-defined exception (ResourceNotFoundException) being thrown, but also provide our own custom error message at the end that would not come with normal 404 messages.



6.2 Using ResponseStatusException

A more dynamic way of implementing a custom Exception is through the ResponseStatusException class.

Make a GET request to:

`localhost:8080/api/seconddemo/10`

Note that a valid Resume object is returned.

Make a GET request to:

`localhost:8080/api/seconddemo/200`

Note that 404 Not Found error is returned with a custom error message. The advance of using the second approach with ResponseStatusException is that it allows the error message returned to be dynamically constructed instead of being hardcoded as in the case of @ResponseStatus. This allows it to incorporate extra information, such as the resource id being requested for, that resulted in the error as well as the ability to also fine tune the HttpStatus code (if something other than the standard 404 is required).

6.3 Creating @ControllerAdvice class with @ExceptionHandler methods

In the `com.workshop.rest` package in `src/main/java`, place the files

```
SampleControllerExceptionHandler
CustomErrorMessage
```

A further extension on the concept of providing tailored and customized error responses to the API consumer is through the use of the @ControllerAdvice to annotate a class that aggregates together specialized exception handling methods that (marked with @ExceptionHandler). These handler methods that catch and handle exceptions from 3 different situations

a) Common runtime Java exceptions (such as NullPointerException, ArrayIndexOutOfBoundsException, IllegalArgumentException).
b) custom user-defined Exceptions that are thrown in the handler methods, typically due to a valid incoming HTTP request that is specifying an action that cannot be performed. A classic example is a HTTP GET request for a resource that does not exist on the app.
c) Spring Web exceptions that occur due to the inability to correctly process an incoming HTTP request because it is invalid in some way. A classic example is a POST request containing malformed JSON content.

In terms of how to handle these exceptions, the handler methods can either:

a) Continue executing code to handle and resolve these exceptions internally in the app without providing any warning to the sender of the HTTP request, who will receive a standard 200 OK response (or something similar)
b) Provide a response to the sender of the HTTP request with an appropriate response status code and message indicating an error has occurred

The first approach is suitable when the exception is due to a bug in the code implementation of the REST app (situation a)
The second approach is suitable when the exception is due to some issue in the incoming HTTP request (situation b)

6.3.1 Handling common runtime Java exceptions

Make a GET request to:

```
localhost:8080/api/demo-error
```

Notice this time that no 500 Internal Server Error response is returned; instead the appropriate exception handler logic is executed in `SampleControllerExceptionHandler`
Verify this in the console log output.
This exception handling is very useful to prevent the REST application from crashing unexpectedly due to undetected run time exception.

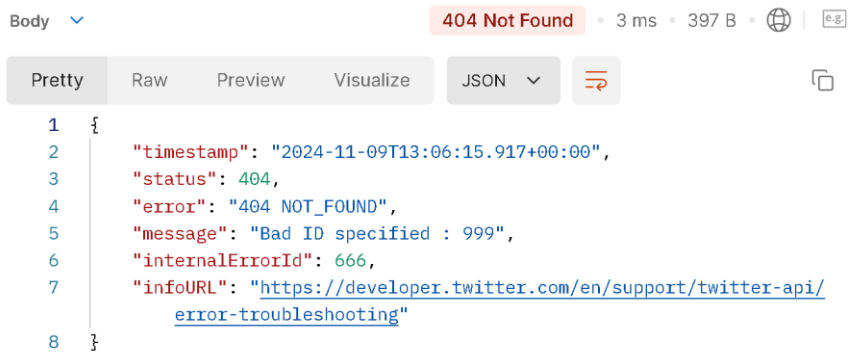
6.3.2 Handling user-defined exceptions

Make a GET request to:

```
localhost:8080/api/thirddemo/999
```

Here we are simulating a situation where the incoming HTTP GET request is in valid format and can be processed correctly by the Spring framework. However, the request is specifying a resource (based on its id of 999) that does not actually exist on the server, so the app must provide a response indicating this situation to the API consumer.

Notice that the error message returned has a customized format which includes additional fields that are not present in typical Spring framework generated messages. These fields provide much more information that help the API consumer to correct the subsequent API requests that they send.



The JSON for this error message is serialized from the CustomErrorMessage object which is constructed with custom values in the handleCustomException method in the @ControllerAdvice class, which is invoked as result of handling the custom exception AnotherCustomException thrown in `thirdemo` handler method.

6.3.3 Finetuning the amount of error information through server.error

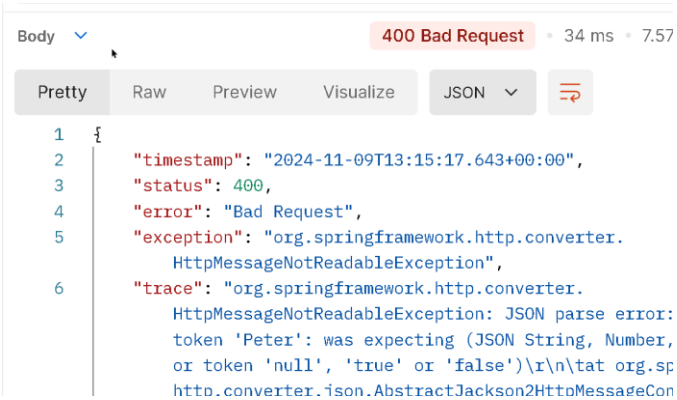
Make a POST request to:

`localhost:8080/api/fourthdemo`

with this raw JSON content that is intentionally malformed:

```
{
    "name": Peter",
}
```

As before, you will notice a great deal of detailed content regarding the exception, such as the stack trace and so on. This is the default setting for error messages in Spring Boot which help to facilitate debugging during the development process.



However, there are certain situations where all these extra information becomes problematic or even dangerous. For e.g. in a REST app that is running live on a production server, providing so much information in an error response may be confusing for users and may also leak sensitive information about the application to hackers who are constantly probing error messages for vulnerabilities in the app.

We can control the amount of information displayed in the error messages through the various server error related properties by the name of `server.error`. You can find a list of these properties at:

https://docs.spring.io/spring-boot/appendix/application-properties/index.html#appendix.application-properties.server

Remove the previous `server.error` properties in `application.properties` in `src/main/resources`
Add in these properties with these values to minimize the information in the error messages:

```
server.error.include-message=never
server.error.include-stacktrace=never
server.error.include-exception=false
server.error.include-binding-errors=never
```

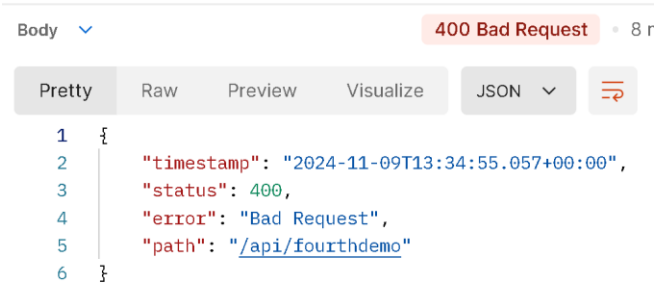
Now make another POST request again to the previous API endpoint:

`localhost:8080/api/fourthdemo`

again using raw JSON content that is intentionally malformed:

```
{
    "name": Peter",
}
```

Notice this time that the amount of information in the error message returned is minimal.



6.3.4 Handling Spring Web exceptions

In the `com.workshop.rest` package in `src/main/java`, make the following changes:

`SampleControllerExceptionHandler-v2`

Repeat the POST request to:

`localhost:8080/api/fourthdemo`

with the same malformed JSON content:

```
{
    "name": Peter",
}
```

Now we have a specific method `handleSpringFrameworkException` in the @ControllerAdvice which catches and handles the exception that occurs when the Spring framework attempts to process the malformed JSOn content in the body of the POST request. This method now returns a custom error message is returned instead

We can use this similar approach to catch and handle other similar exceptions thrown by the Spring framework if we wish. The complete list of Spring Web framework exceptions as shown at this link: