Spring Boot Micro Services

1. **What is dispatcher servlet?**

The job of the *DispatcherServlet* is to take an incoming URI and find the right combination of handlers (generally methods on *Controller* classes) and views (generally JSPs) that combine to form the page or resource that's supposed to be found at that location.

Dispatcher Servlet follows front-controller design pattern.

**Front controller design pattern:** The front controller design pattern means that all requests that come for a resource in an application will be handled by a single handler and then dispatched to the appropriate handler for that type of request.

1. **Who is configuring dispatcher servlet?**

Spring boot starter web has the dependency on spring web mvc framework, therefore we get the dispatcher servlet class in our class path. So spring boot Auto configuration will configure the dispatcher servlet as it is available in class path.

Basically Spring boot auto configuration will find the relevant classes in classpath and configure it. (Example: dispatcher servlet, Error MVC Auto Configuration, WhitelabelErrorViewConfiguration and HttpMessageConvertersAutoConfiguration etc..)

Beans are automatically converted to JSON because of HttpMessageConverters.

JacksonAutoConfiguration: This actually convert to JSON to Beans and Beans to JSON.

1. **What does dispatcher servlet do?**

Dispatcher Servlet handles all the requests (URI). This is the root of the web application. Once it gets the request it determines which is the right controller to execute is. It looks at the URL and find the right method and execute it.

Once we get the response from the method, DispatcherServlet looks at how to send the response back?

When we declare the @RequestController Annotation, inside that we have @ResponseBody annotation. The functionality of @ResonseBody is, response from the controller would be mapped by a message converter into some other format, here message converter which is going to be used is Jackson. Then Dispatcher servlet uses Jackson and converts into JSON.

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

HttpMessageConverters and JacksonAutoConfiguration.

1. **Who is configuring the error mapping?**

Spring boot auto configuration

1. **What is the difference between @RestController and @Controller?**

The @RestController annotation in Spring MVC is nothing but a combination of @Controller and @ResponseBodyannotation. It was added into Spring 4.0 to make the development of RESTful Web Services in Spring framework easier. If you are familiar with the [REST web services](http://www.java67.com/2017/04/3-great-books-to-learn-java-web-services-soap-and-restful.html) you know that the fundamental difference between a web application and a REST API is that the response from a web application is generally view (HTML + CSS + JavaScript) while REST API just return data in form of JSON or XML. This difference is also obvious in the @Controller and @RestController annotation. The job of @Controller is to create a Map of model object and find a view but @RestControllersimply return the object and object data is directly written into HTTP response as JSON or XML.

This can also be done with traditional @Controller and use @ResponseBody annotation but since this is the default behavior of RESTful Web services, Spring introduced @RestController which combined the behavior of @Controller and @ResponseBodytogether.  
  
In short, following two code snippet are equal in Spring MVC:

@Controller

@ResponseBody

public class MVCController {

.. your logic

}

@RestController

public class RestFulController {

.... your logic

}

1. **What are the new Spring MVC Request Annotations?**

New Spring MVC Request Annotations

* @PostMapping – Handle HTTP POST Requests
* @GetMapping – Handle HTTP Get Requests
* @PutMapping – Handle HTTP Put Requests
* @DeleteMapping – Handle HTTP Delete Requests

Ex 1:

@GetMapping("/status/check")

public String status()

{

return "Working";

}

Ex 2: @GetMapping(path = "/{id}", produces = {MediaType.APPLICATION\_JSON\_VALUE, MediaType.APPLICATION\_XML\_VALUE})

1. **What is @PathVaribale annotatioon?**

In Spring MVC, the @PathVariable annotation binds a URI template variable to a method parameter in a controller.

A URI template contains one or more path variables enclosed by braces ({}) as follows.

[*http://www.example.com/user/{firstName}/{lastName}*](http://www.example.com/user/%7bfirstName%7d/%7blastName%7d)

When path variables of a URI template are substituted with actual value then it becomes a URI.

[*http://www.example.com/user/Sunil/Singh*](http://www.example.com/user/Sunil/Singh)

*Example:*

**/\*** /hello-World/path-variable/Raman -- If somebody typed Raman in URI to get mapped to {name}. **\*/**

@GetMapping(path = "/hello-World/path-variable/{name}")

public HelloWorldBean getHelloWorldPathVariable(@PathVariable String name) {

return new HelloWorldBean(String.format("Hello-World, %s",name) );

}

<http://localhost:8080/hello-World/path-variable/Raman>

1. **How to send timestamp value instead of nano seconds in JSON response?**

By specifying the below property to false.

spring.jackson.serialization.write-dates-as-timestamps=false

1. **How to return HTTP status code and How to send the URI of the created resource?**

URI location = ServletUriComponentsBuilder.*fromCurrentRequest*().

path("{id}").

buildAndExpand(savedUser.getId()).toUri();

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

**Note:** ResponseEntity has different methods to return respected responses

*a)created* (location): HttpStatus.CREATED - 201 Created.

b) noContent(): [HttpStatus](eclipse-javadoc:%E2%98%82=Restful-Web-Services/C:%5C/Users%5C/ramanjaneyulu_t%5C/.m2%5C/repository%5C/org%5C/springframework%5C/spring-web%5C/5.0.8.RELEASE%5C/spring-web-5.0.8.RELEASE.jar%3Corg.springframework.http(HttpStatus.class%E2%98%83HttpStatus).NO\_CONTENT – 204

c) notFound():HttpStatus.NOT\_FOUND - 404 Not Found.

d) ok(): HttpStatus.OK - 200 OK.

e) Accepted() - ACCEPTED - 202 Accepted.

1. **How to implement Exception Handling using Spring Boot – 404 Not found?**

Create a user defined exception and throw it whenever the resource not found.

Sample Code:

**Controller Logic:**

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

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

User user = userDaoService.findOne(id);

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

**throw** **new** UserNotFoundException("id-"+ id);

}

**return** user;

}

**User Defined Exception:**

**package** com.test.rest.webservices.RestfulWebServices.user;

**import** org.springframework.http.HttpStatus;

**import** org.springframework.web.bind.annotation.ResponseStatus;

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

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

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**public** UserNotFoundException(String message) {

**super**(message);

}

}

Note: @ResponseStatus(HttpStatus.NOT\_FOUND) will send the status code 404 in response.

1. **How to Implement Generic Exception handling for all Resources?**

**Step 1:** Create a bean with the properties that you want to send when you get exception.

**Ex:**

**package** com.test.rest.webservices.RestfulWebServices.Exception;

**import** java.util.Date;

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

**private** Date timeStamp;

**private** String message;

**private** String details;

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

**super**();

**this**.timeStamp = timeStamp;

**this**.message = message;

**this**.details = details;

}

**public** Date getTimeStamp() {

**return** timeStamp;

}

**public** String getMessage() {

**return** message;

}

**public** String getDetails() {

**return** details;

}

}

**Step 2:** Create a class which handles all the exceptions and if you want to customize for specific exception, you can do it using the following logic.

**package** com.test.rest.webservices.RestfulWebServices.Exception;

**import** java.util.Date;

**import** org.springframework.http.HttpStatus;

**import** org.springframework.http.ResponseEntity;

**import** org.springframework.web.bind.annotation.ControllerAdvice;

**import** org.springframework.web.bind.annotation.ExceptionHandler;

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

**import** org.springframework.web.context.request.WebRequest;

**import** org.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionHandler;

**import** com.test.rest.webservices.RestfulWebServices.user.UserNotFoundException;

@ControllerAdvice

@RestController

**public** **class** CustomizedResponseEntityExceptionHandler **extends** ResponseEntityExceptionHandler {

@ExceptionHandler(Exception.**class**)

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

ExceptionResponse exceptionResponse = **new** ExceptionResponse(**new** Date(), ex.getMessage(),

request.getDescription(**false**));

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

}

@ExceptionHandler(UserNotFoundException.**class**)

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

ExceptionResponse exceptionResponse = **new** ExceptionResponse(**new** Date(), ex.getMessage(),

request.getDescription(**false**));

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

}

}

**Note:** Important classes/Annotations to remember.

**@ControllerAdvice:** @ControllerAdvice is an annotation provided by Spring allowing you to write global code that can be applied to a wide range of controllers — varying from all controllers to a chosen package or even a specific annotation.

Using @ControllerAdvice Classes

A controller advice allows you to use exactly the same exception handling techniques but apply them across the whole application, not just to an individual controller. You can think of them as an annotation driven interceptor.

Any class annotated with @ControllerAdvice becomes a controller-advice and three types of method are supported:

1. Exception handling methods annotated with @ExceptionHandler. A controller with @ExceptionHandler methods to handle its own exceptions.
2. Model enhancement methods (for adding additional data to the model) annotated with

@ModelAttribute. Note that these attributes are not available to the exception handling views.

1. Binder initialization methods (used for configuring form-handling) annotated with

@InitBinder.

ResponseEntityExceptionHandler:

A convenient base class for [@ControllerAdvice](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/ControllerAdvice.html) classes that wish to provide centralized exception handling across all @RequestMapping methods through @ExceptionHandler methods.

If you want to handle a custom exception that doesn’t have a default implementation in the base class, we need to override a **base method out of the***ResponseEntityExceptionHandler***and providing our own custom implementation.**

1. **How do you implement validations for Restful web services?**

**Step 1:** Per the JSR 380 specification, the validation-api dependency contains the standard validation APIs:

<dependency>

<groupId>javax.validation</groupId>

<artifactId>validation-api</artifactId>

<version>2.0.0.Final</version>

</dependency>

**Step 2:** We’ll use a User bean as the main example here and work on adding some simple validation to it

public class User {

@NotNull(message = "Name cannot be null")

private String name;

@AssertTrue

private boolean working;

@Size(min = 10, max = 200, message

= "About Me must be between 10 and 200 characters")

private String aboutMe;

@Min(value = 18, message = "Age should not be less than 18")

@Max(value = 150, message = "Age should not be greater than 150")

private int age;

@Email(message = "Email should be valid")

private String email;

// standard setters and getters

}

All of the annotations used in the example are standard JSR annotations:

**@NotNull** – validates that the annotated property value is not null

**@AssertTrue** – validates that the annotated property value is true

**@Size** – validates that the annotated property value has a size between the attributes min and max; can be applied to String, Collection, Map, and array properties

**@Min** – validates that the annotated property has a value no smaller than the value attribute

**@Max** – validates that the annotated property has a value no larger than the value attribute

**@Email** – validates that the annotated property is a valid email address

Some annotations accept additional attributes, but the message attribute is common to all of them. This is the message that will usually be rendered when the value of the respective property fails validation.

Some additional annotations that can be found in the JSR are:

**@NotEmpty** – validates that the property is not null or empty; can be applied to String, Collection, Map or Array values

**@NotBlank** – can be applied only to text values and validated that the property is not null or whitespace

**@Positive and @PositiveOrZero** – apply to numeric values and validate that they are strictly positive, or positive including 0

**@Negative and @NegativeOrZero** – apply to numeric values and validate that they are strictly negative, or negative including 0

**@Past and @PastOrPresent** – validate that a date value is in the past or the past including the present; can be applied to date types including those added in Java 8

**@Future and @FutureOrPresent** – validates that a date value is in the future, or in the future including the present

The validation annotations can also be applied to elements of a collection:

List**<@NotBlank** String> preferences;

**Step 3:**

Add @Valid validation in controller.

Ex:

@PostMapping("/users")

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

**Step 4:**

If you simply follow the above steps, client will get 400 status code, then how does consumer will know where it goes wrong. We have to return a specific message to provide more details.

We will add it in CustomizedResponseEntityExceptionHandler.java class

If any validation failure happens, then handleMethodArgumentNotValid() method will get executed when binding to the specific method fails (which is available in ResponseEntityExceptionHandler.java class)

@Override

protected ResponseEntity<Object> handleMethodArgumentNotValid(

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

ExceptionResponse exceptionResponse = new ExceptionResponse(new Date(), “validation failed”,ex.getBindingResult().toString());

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

}

Note: ex.getBindingResult().toString() – give the consumer what went wrong.

1. **What is HATEOS for Restful services?**

The Spring HATEOAS project is a library of APIs that we can use to easily create REST representations that follow the principle of HATEOAS (Hypertext as the Engine of Application State).

HATEOAS is an extra level upon REST and is used to present information about a REST API to a client, allowing for a better understanding of the API without the need to bring up the specification or documentation. This is done by *including links in a returned response and using only these links to further communicate with the server*.

If there are some static endpoints that the client can make use of and further calls are done via the links included in the response, the client’s code should not break (though I am not claiming it is 100% safe). This makes the assumption that the links returned with the response have already implemented the standard REST verbs. Please find the below steps for implementation.

Step 1: Add dependency in pom.xml

<dependency>

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

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

</dependency>

Step 2: Implement logic in controller, similar to the below code.

@GetMapping("/users")

public List<User> findAll(){

return userDaoService.findAll();

}

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

public Resource<User> findOne(@PathVariable int id){

User user = userDaoService.findOne(id);

if(user==null) {

throw new UserNotFoundException("id-"+ id);

}

// HATEOAS

// link to all - users, server path + "/users"

Resource<User> resource = new Resource<User>(user);

ControllerLinkBuilder linkTo = ControllerLinkBuilder.linkTo(methodOn(this.getClass()).findAll());

resource.add(linkTo.withRel("all-users"));

return resource;

}

*Output:*

{

"id": 2,

"name": "Ram",

"birthDate": "2018-09-03T11:33:54.885+0000",

*"\_links": {*

*"all-users": {*

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

*}*

*}*

}