**Spring Microservices**

**Section 1**

Two key sections:

* Restful web services with spring boot
* Microservices with spring cloud

**Section 2**

Web service: A software system designed to support interoperable machine to machine interaction over a network.

A web service will be designed for machine to machine or application or application interaction. It should be interoperable, i.e., be able to work to any operating system. It should also allow communication over a network.

A web service communicates by taking requests and sending responses to a machine. A web service can be made platform independent by making the request and response platform independent. One way is to use XML, another is JSON. What request to send, where to send it and the format of response is defined by the service definition.

Request

Response

Service provider

Service consumer

Service definition

Transport – defines how a service is called – over the internet (HTTP) - or over queue (MQ)

Web service groups: SOAP – based & REST – based

REST and SOAP aren’t really comparable. REST defines an architectural approach whereas SOAP imposes restrictions on the format of request and response.

SOAP defines a specific way of building web services. In SOAP, XML is used as the request exchange format. It defines a specific XML request and response structure however transport may be http or MQ. In SOAP service definition is done by using WSDL (Web service definition language).

REST (REpresentational state transfer) makes best use of HTTP. HTTP has request method types such as GET, POST, PUT etc. Method type depends on the type of the request. HTTP also has status codes returned by responses. RESTful services are services that implement REST. The things to be exposed are called resources and a URI (Uniform resource identifier) is used to get a resource. A resource can have different representations : XML, HTML, JSON. Transport is always HTTP whereas there is not standard service definition (WADL/Swagger).

**Section 3**

**RESTful Services**

We can create a web service by creating a controller in spring mvc. Use @RestController to tell spring that it will handle rest requests. @RequestMapping is used to create a mapping for a method. Within request mapping we can write @RequestMapping(method = RequestMethod.GET, path=”/hello-world”) to specify method type and path. Now if we try localhost-8080/hello-world, it will return whatever the method specifies. Instead of @RequestMapping we can use @GetMapping to mark the method as a GET method by default.

If we are trying to return an object, the object parameters must have getters, otherwise the automatic conversion from object to JSON won’t work.

We can add path variables to a web service method by typing “/hello-world/{name}”, here name will be a path variable passed during the call.

ServletUriComponentsBuilder.fromCurrentRequest().path(“/{pathVariable}”).buildAndExpand(getPathVariable()).toURI will return the URI of the current request. We can return We can set the response statement of created by writing ResponseEntity<T>.created(location).build() which will return the status code of 201 created. In the headers, we can see the location which is the URI of the current request.

In case of errors, we need to throw the proper exception and status code. Normally, an exception would throw 500 – internal server error. We can write @ResponseStatus(HttpStatus.NOT\_FOUND) above an exception class to return 404 not found.

We can give a specific format to our exceptions. We can create a bean for an exception and then create an exception handling class which extends ResponseEntityExceptionHandler. The exception handling class has to have the annotation @RestController as it provides a response in case of exception. It has to be applicable across all other controllers – it can be done by using @ControllerAdvice. Controller Advice is used to share methods across multiple controller classes.

@ExceptionHandler(Exception.class) – exception handler can be used above a method in this class to tell it which exceptions to handle. The exceptions to handle can be specified within the parenthesis. Inside this method we can create a new instance of our bean with the details and return it using a ResponseEntity.

In order to run a delete http method, we can use the annotation @DeleteMapping.

We can add validations in methods by using @Valid before a parameter in method parameters. Inside the object class before which we used @Valid, we can use annotations @Size for string, @Past for time etc. So if the validation fails, the request will return 400 bad request. We can give custom messages by overriding method handleMethodArgumentNotValid from ResponseEntityExceptionHandler and return a message (exception.getBiningResult() will have the reason for failing) with exceptionResponse. We can also write a customized message along with @Size or @Past annotations for parameter validations in objects. All the validations are defined in validation-api.jar.

**HATEOS**

HATEOS (Hypermedia as the engine of application state ) is used to provide links to navigate a REST interface. It does so by dynamically including hypermedia links with JSON or XML responses from an API. We can use HATEOS by including the dependency spring-boot-starter-hateos in pom.xml.

In HATEOS, there is a concept called Resource<T> where T is a class name. We can create resource and then add links to that resource. ControllerLinkBuilder will enable us to create links from methods. ControllerLinkBuilder can be used to run a method called linkTo() and the argument can be passed as methodOn() which takes the method as argument. The link created using ControllerLinkBuilder can be added to resource to create a link.

**Internationalization**

Internationalization can be done for REST services to return different messages according to the language.

We need to create a properties file for all the messages. Different properties files must be created for the different languages. Then we need to add a bean localeResolver in the main application. We can set the default locale here and set it to what we want (eg: Locale.US). The file is messages.properties for us by default and messages\_fr.properties for French and so on.

We need something to read the properties and configure the messages according to the locale. For that we need to create resourceBundleMessageSource in the main application and call messageSource(). Inside this method we can set messageSource.setBasename(“propertiesFileName”).

In order to make our service to make use of internationalization, we need to create a MessageSource in the controller and auto wire it in. Then at the places where the message is returned, we can use messageSource.getMessage(“propertyName”, locale). Locale can be set as a header by the client to identify the language. In order to tell spring that locale is a header, we can use the annotation @RequestHeader (name=”nameOfTheHeader”, required=false).

We don’t need to pass locale as a parameter, instead we can run LocaleContextHolder.getLocale() to get the locale automatically. We can use an AcceptHeaderLocaleResolver instead of a sessionlocaleresolver to get the locale from the header (Header value: Accept-Language). We can configure the base name of the properties file which holds the messages in application.properties – spring.messages.basename=messages

**Content Negotiation**

We can specify the content we want to use for our requests by specifying application/xml or application/json. All the binding form json to objects and vice versa happens via Jackson. In order to get the same for xml, we can add a simple jar – Jackson-dataformat-xml. With this, we can get our requests and responses in either json or xml.

**Swagger Documentation**

Swagger is used to document restful services so that the end users can get details on how to use these services. We need to add springfox-swagger2 and springfox-swagger-ui in pom.xml to get swagger dependency. @EnableSwagger2 annotation is used to enable swagger for a class. A class docket is used to return docket(DocumentationType.SWAGGER\_2). Once this is done we can go localhost:8080/v2/api-docs we can see the documentation that is being generation. Or we can go to localhost:8080/swagger-ui.html to see the services offered in a more refined UI.

We can add custom documentation by creating constants of the type ApiInfo. Refer the constants in ApiInfo to check how documentation is being generated there. We can specify more documentation by using @ApiModel above classes and class parameters which are part of the service. Swagger annotations jar holds

**Monitoring APIs with Spring Boot Actuator**

The dependencies for monitoring are spring-boot-starter-actuator and spring-data-rest-hal browser. The hal browser looks at the hal services exposed by the actuator and display it in an easy to read manner. We can enter management.endpoints.web.exposure.include=\* in properties to enable tracking for everything. However, it is performance intensive and therefore only those endpoints which are required must be enabled.

**Filtering**

Filtering is the process of filtering out fields from the output bean. @JsonIgnore annotation can be used on top of a field to tell spring to exclude a field from the json output. @JsonIgnoreProperties(value={“fieldname”}), can be used on top of a class to specify which fields to ignore. However, this filtering is static as it does not depend on any conditions.

Dynamic filtering is the process where filtering is done according to some condition. Dynamic condition cannot be done directly on the bean. To do that, we have to use MappingJacksonValue class and pass the bean we need to filter in the constructor. Then we can filter the fields by using FilterProvider – FilterProvider filters = new SimpleFilterProvider().addFIlter(“FilterName”,”field1, field2”);

Then we add the FilterProvider to MappingJacksonValue with the method setFilters(filters). The MappingJacksonValue is returned instead of the bean. The list of valid filters have to be defined on top of the bean with @JsonFilter(“filterName”).

**Versioning RESTful services**

Versioning is the process of updating a web service to another version while the older version still remains. Therefore multiple versions of a service must be maintained.

\*The simplest version of mapping different versions is to use a different URI for each version. (URI versioning)

\*Another way is to use a request parameter. This parameter can be used to differentiate between the versions. (Request parameter versioning)

Eg: @GetMapping(value=”/person/param”, params=”version=1”)

@GetMapping(value=”/person/param”, params=”version=2”)

Thus we can sent the URI person/param=?version=1 – the version number will direct to the corresponding version.

\*Header versioning can be used to differentiate according to a header parameter. (Header versioning)

Eg: @GetMapping(value=”/person/param”, headers=”HEADERNAME=1”)

We can specify the header name and its value in the request header. And depending on this value we will get the appropriate version.

\*Another method is to use produces. (Accept header versioning or mime type versioning or media type versioning)

Eg: @GetMapping(value’”/person/produces”, produces = “application/vnd.company.app-v1+json”)

We can send this as part of our request by using ‘Accept’ header name with the produces value.

**Basic authentication with spring security**

Basic authentication can be implemented by using spring-boot-starter-security in pom.xml which will download the spring security jar. When the server starts, we can see default security password in the console. Now, with every API request, we will need authorization. We will have to send requests using a basic authorization now. Username is user and password is the one shown in console. Password and username can be configured in application.properties – security.user.name=username and security.user.password=password

**Connecting RESTful service to JPA**

We can make a user an entity by using the annotation @Entity. We can set up H2 and input some test data in it. We can now access data from the embedded database by using a repository.

In case we have a two objects with a many to one relationship. We can use @ManyToOne(fetch=FetchType.LAZY) on top of the parameter which is the many to signify this. Lazy fetch type will ensure that it is not fetched till it is called.

@OneToMany(mappedBy=”Name of the field in ‘many’”) can be used on top of the parameter which is the one.

**Richardson maturity model**

Richardson maturity model helps us evaluate our RESTful services. It has three different levels:

* LEVEL 0 – expose SOAP web services in rest style : That is, we have to take our services and expose them in rest style such as <http://server/getPosts>
* LEVEL 1 - Expose resources with proper URI: In level 1, we think of resources and make proper URI to use them
* LEVEL 2 – level 1 + proper HTTP methods: level to uses level 1 but with the proper HTTP methods of get, post etcetera for getting information, creating posts etc.
* LEVEL 3 – Level 2 + HATEOAS, data + next possible actions: level 3 deals not only with returning the data, but also information on what to do next or the possible actions

**Best practices**

* Consumer first – think from the perspective of consumers, make Uri understandable and have great documentation
* Make best use of HTTP – use right request methods, ensure proper response status is returned
* No secure info in URI – ensure that nothing secure goes in the URI
* Use plurals – always use plurals for URIs
* Use nouns for resources – when creating resources, use a noun such as user instead of add user or delete user
* Define a consistent approach for all exceptions

**Section 4**

**Microservices with spring cloud**

A Microservices is a small service which is independently deployable. Microservices work together to make an application work. Microservices are services with are deployed by REST, are small well-chosen deployable units and are cloud enabled.

Cloud enabled Microservices can have multiple instances. If there is more load on a particular microservice, then it should create another instance without affecting the other services.

A few challenges faced in designing microservices are: bound context, scaling up and down, configuration management, visibility and pack of cards (if one microservice goes down, the entire service might go down – must have fault tolerance).

Spring cloud

Spring cloud provides tools for developers to quickly build some of the patterns in distributed systems.

Spring cloud config server provides an approach where we can store all the configuration in a git repository and thus manage it easily.

It also helps to do load balancing to scale up and down the instances. For this we use a naming server (Eureka) – which takes care of service registration and service discovery. Ribbon is used for client side load balancing. Feign is used as a mechanism to write simple RESTful services.

The visibility and monitoring can be done using zipkin distributed tracing (spring cloud sloat assigns IDs to request across multiple components and this can be traced using zipkin) and Netflix API gateway (API gateways provide common features like logging security analytics etc.).

Fault tolerance can be implemented using hysteric which helps with providing a default response when a service is down.

**Centralized configuration**

We need to configure an application to get the configuration from spring cloud config server. We need to have spring-cloud-config-server as a dependency in pom.xml to use spring cloud config server (add config server as a dependency at spring initializer).

Create a local git repo and right click on the project in eclipse and click on build path and source link – then point it to the local folder we created. Then we need to create a new text file to store configuration in this folder.

Now we need to connect the spring cloud config server to the local git repository. First find the folder location, and then go to application.properties of spring cloud config server and add spring.cloud.config.server.git.uri=”the location of the local git repository” (If it’s a local file – file:///location of the configuration file). We need to add @EnableConfigServer on top of the application class to enable config server.

In order to have multiple configurations for different environments, we can have multiple files in the git repo. Then to each file, we can append –dev and –qa for dev and qa. The default file would always be used unless dev or qa is required. Now we can write localhost:8080/cloudName/default (or qa or dev) to get the corresponding config values.