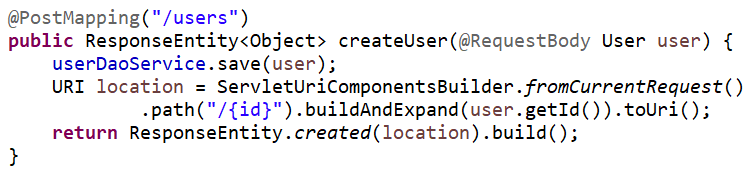
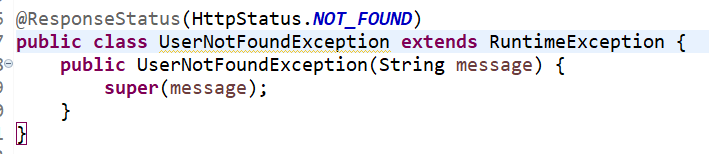
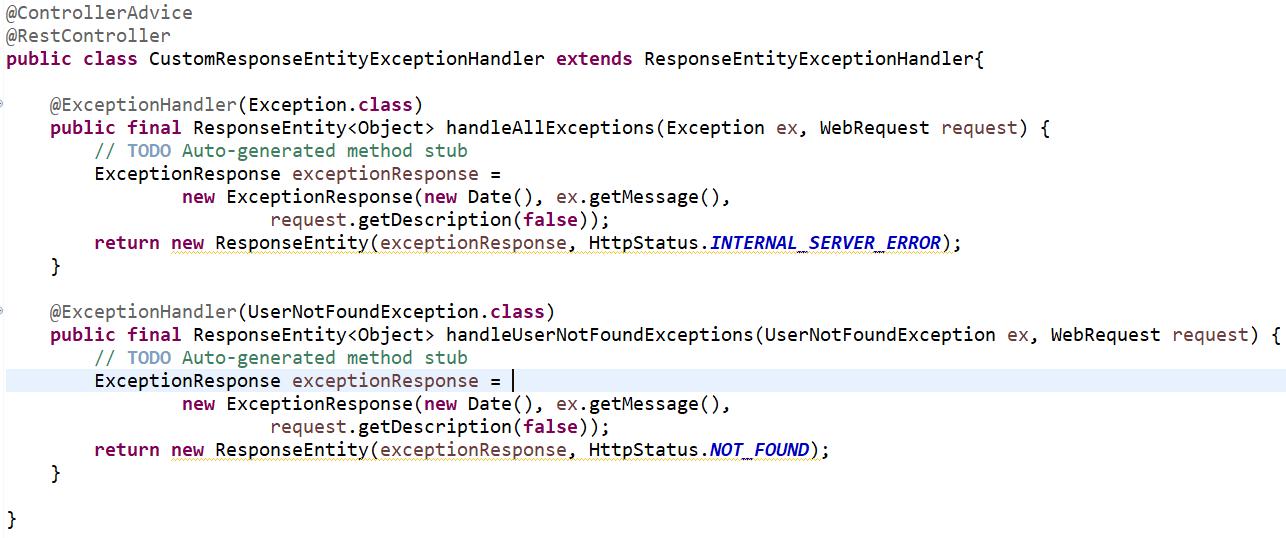
1. Spring Microservices
   1. Introduction to web services
      1. Key points about webservices
         1. Designed for application-to-application interaction
         2. Should be interoperable – Not platform dependent
         3. Should allow communication over a network
      2. Webservices take request as input, then perform some business logic and the process response as output.
      3. To make webservices as platform independent we need to use request/response as format which acceptable by every technology. Two popular data exchanges format is xml and json.
      4. Service definition
         1. Request/response format
         2. Request structure
         3. Response structure
         4. Endpoint
   2. SOAP Web services
      1. Simple Object Access Protocol
      2. It used XML as data exchange format
      3. SOAP uses specific xml structure which called SOAP Envelope (SOAP-ENV Header and SOAP-ENV Body)
      4. Service Definition
         1. It is done using WSDL(Web service definition language)
   3. RESTful Web services
      1. REpresentational State Transfer
      2. Make best use of HTTP
      3. HTTP Methods (GET, POST, PUT, DELETE)
      4. HTTP Status Codes (200 - OK,201 – Created, 204- No Content, 400- Bad Request, 401 – Unauthorized, 404 -Not Found, 500 Server Error etc)
      5. Resource
         1. A resource has an URI (Uniform Resource Identifier)
         2. A resource can have different representations
            1. XML
            2. JSON
      6. Data exchange format
         1. No restrictions. But JSON is popular
      7. Service Definition
         * 1. No Standard Service Definition
2. Restful Web services with Spring Boot
   1. @RestController
      1. The @Controller is a annotation to mark class as Controller Class in Spring While @RestController is used in REST Web services and similar to @Controller and @ResponseBody.
      2. The @Controller annotation indicates that the class is controller like web Controller while @RestController annotation indicates that the class is controller where @RequestMapping Method assume @ResponseBody by Default(i.e REST APIs).
      3. The key difference is that you do not need to use @ResponseBody on each and every handler method once you annotate the class with @RestController.
      4. @Controller create a Map of Model Object and find a view while @RestController simply return object and object data directly written into http response as JSON orXML.
      5. @RequestMapping(method=RequestMethod.GET, path=”url”)
      6. @GetMapping(path=”url”) or @GetMapping(“url”)
      7. If we return Model then it automatically converted into string and return string (model should convert get method for fields)
   2. Spring Boot Auto Configuration & Dispatcher Servlet
      1. Add line in application.properties (logging.level.org.springframework = debug) to enable debug mode for checking all auto configuration details.
      2. ErrorMvcAutoConfiguration.WhitelabelErrorViewConfiguration is for white label error page.
      3. HttpMessageConvertersAutoConfiguration is for converting bean into json while returning.
      4. @RestController = @Controller + @ResponseBody (convert response into json)
      5. @ConditionalOnClass, @ConditionalOnProperty, @ConditionalOnBean and @ConditionalOnMissingBean
   3. (@PathVariable String name) is used to get variable from url path (path="/hello-world-bean/path-variable/{name}")
   4. @Component and @Service annotation is used to inject class in controller using @AutoWired annotation.
   5. Return uri in header to access post request data



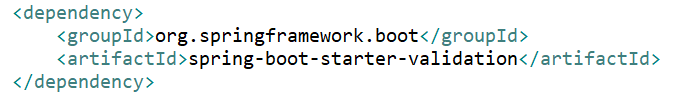
* 1. Implementing Exception Handling – 404 Resource not Found
     1. Add @ResponseStatus (HttpStatus.NOT\_FOUND) to set any error class status code. So if we throw this custom exception then its will display with same Http Error Code.



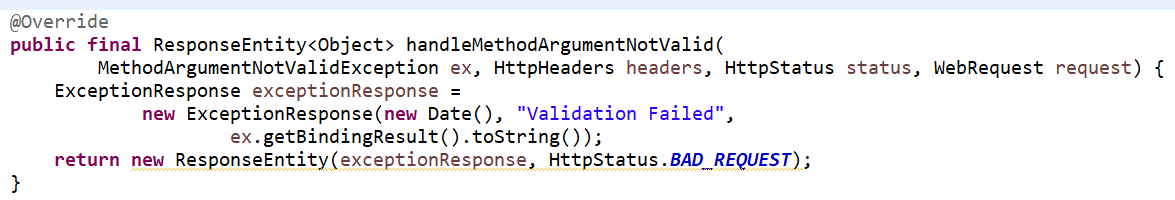
* 1. Generic Exception Handling for all controller



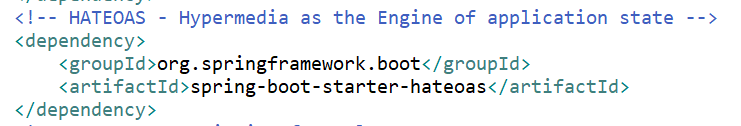
* 1. Starter Validation
     1. In previous versions it was add by default but in new versions we need to add dependency in pom.xml

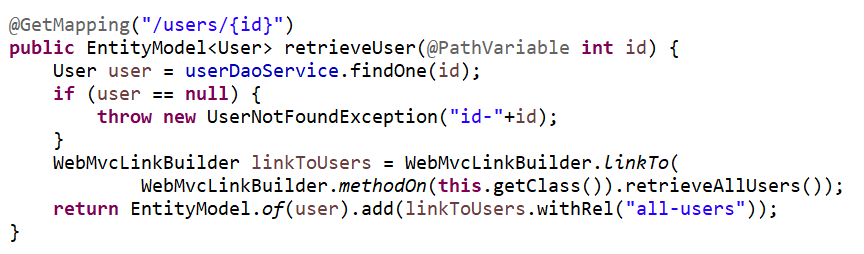


* + 1. This is used to add validation in request data and if request data is invalid then it will return Bad Request Http Status Code(400).
    2. Implementing for RESTful services
       1. Add @Valid before @RequestBody in parameter
       2. @Size(min=2, message=”Name should have atleast 2 characters”), @Past
       3. By Default if request data is invalid then it will not return any message that what is invalid. So to implement custom message need to override handleMethodArgumentNotValid method
       4. Exception.getBindingResult() will give all why request is invalid.
       5. Package javax.validation.constraints contains all valid constraints in Jakarta.validation-api

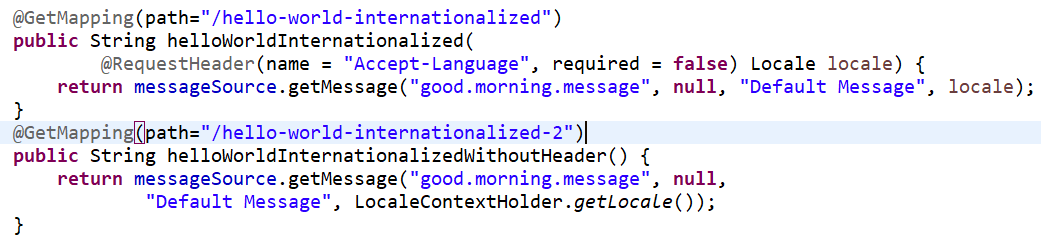


* 1. HATEOAS For RESTful services
     1. Hypermedia as the Engine of application state
     2. It is used to return data with links. Links are to provide action detail which will done by user.

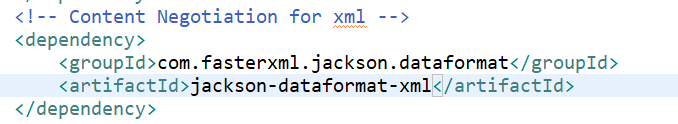




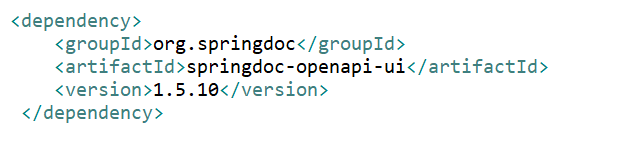
* 1. Internationalization(i18n) for RESTful services
     1. RESTful services which provide data in different languages.
     2. Pass “**Accept-Language**” = “en” or “fr” or “nl” in Request Header
     3. Create **messages.properties** file and add message (for other lang **messages\_fr.properties**, **messages\_nl.properties**)
     4. @Autowired **MessageSource** to access message from property file.



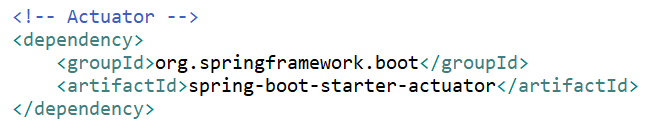
* 1. Content Negotiation Implementing Support for XML
     1. Add dependency into pom.xml
     2. Pass “Accept” = “application/xml” in Request Header then it will convert response into xml format.



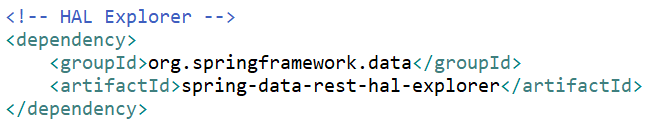
* 1. Configuring Auto Generation of Swagger Documentation (Spring Open API)
     1. <https://springdoc.org/>
     2. Add dependency in pom.xml



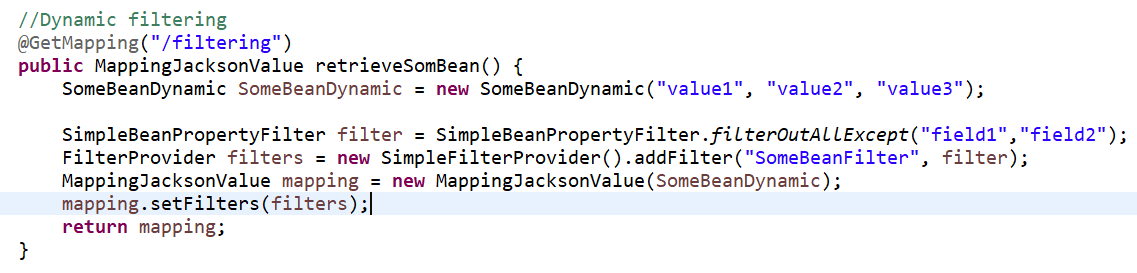
* + 1. Access swagger ui or open api using <http://localhost:8080/swagger-ui/index.html>
  1. Monitoring APIs with Spring Boot Actuator
     1. It helps to monitor and manage application when we push it into production
     2. Add dependency in pom.xml



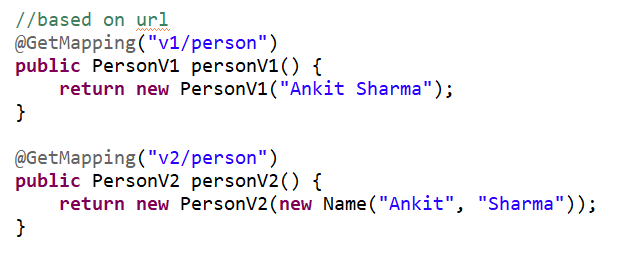
* + 1. Access actuator details using <http://localhost:8080/actuator>
    2. To enable to endpoints related to actuator add management.endpoints.web.exposure.include=\* in applications.properties
  1. Visualizing APIs with HAL Explorer
     1. It provides easy way to explore all the apis like postman.
     2. Add dependency in pom.xml



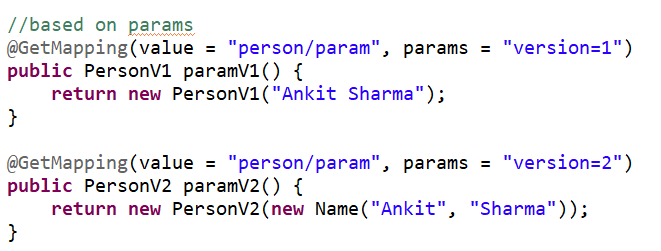
* + 1. Access HAL Explorer using <http://localhost:8080/>
  1. Implementing static filtering for RESTful services
     1. If we don’t want to return some fields in response, then we can use filtering.
     2. Use **@JsonIgnore** at fields level and **@JsonIgnoreProperties**(value={“field1”, “field2”}) at class level
  2. Implementing Dynamic filtering for RESTful services
     1. For some requests send 2 fields and for some requests send all fields that can be handle by dynamic filtering
     2. Add @JsonFilter("SomeBeanFilter") at class in bean class
     3. Then use MappingJacksonValue for filtering and return same.



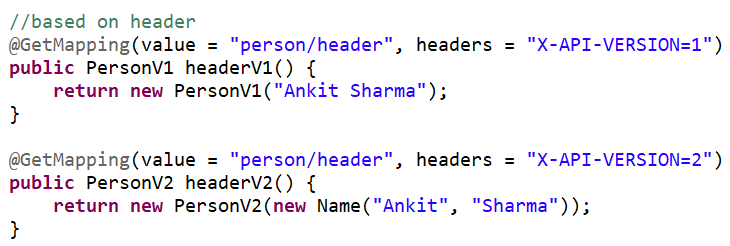
* 1. Versioning RESTful services
     1. We can handle versioning in multiple ways
        1. Basic approach with URIs



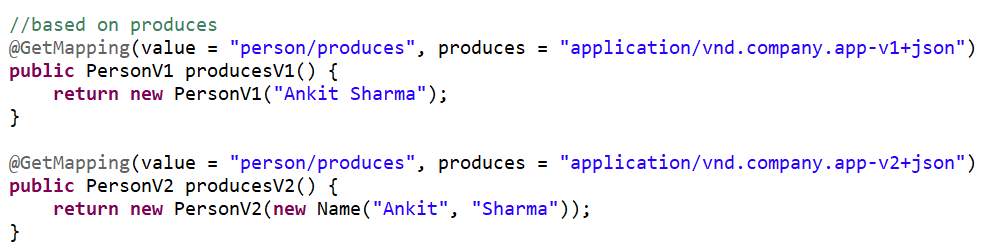
* + - 1. Sending Param with version=1 or 2



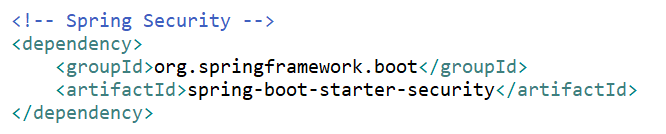
* + - 1. Sending Header with X-API-VERSION=1 or 2



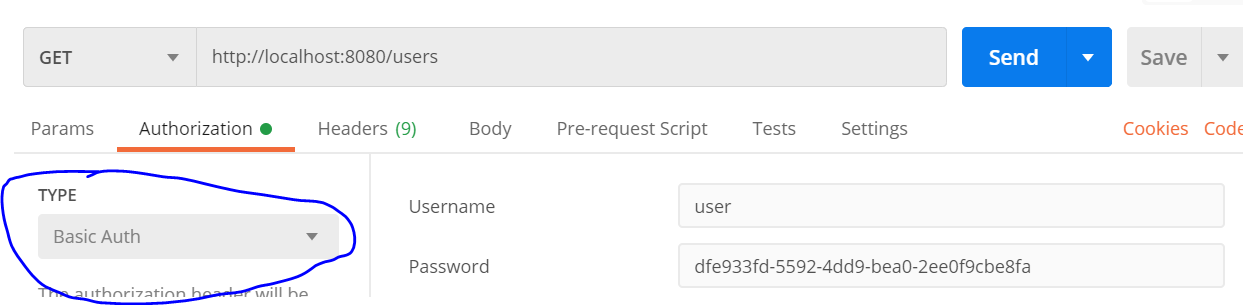
* + - 1. Media Type versioning i.e. Content Negotiation or accept header (mention produces in mapping, send produces value in “Accept” header)



* + 1. Factors to choose which versioning should use in applications are
       1. URI Pollution
       2. Misuse of HTTP Headers
       3. Caching
       4. Can we execute the request in the browser?
       5. API Documentation
  1. Implementing Basic Authentication with Spring Security
     1. Adding dependency in pom.xml



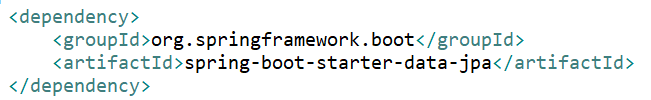
* + 1. Check console log for security password after restarting the server (Using generated security password: dfe933fd-5592-4dd9-bea0-2ee0f9cbe8fa)
    2. After restarting server every time password got changed.
    3. After Basic Authentication with username: user and password: generated security password in each request.



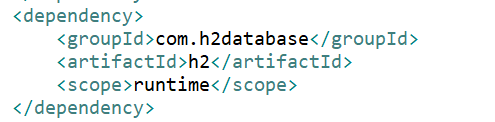
* + 1. By default, it will setup the security and also add login page for providing username/password to access each api. So if we don’t pass username/password then it will give unauthorized error in response.
    2. We can override default username and password using application.properties file



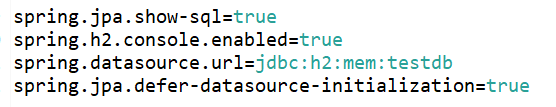
* 1. Connecting RESTful service to JPA
     1. Add dependency in pom.xml



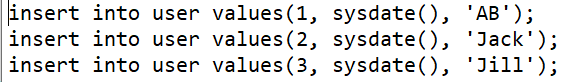
* + 1. If want to use h2 in memory database then add dependency



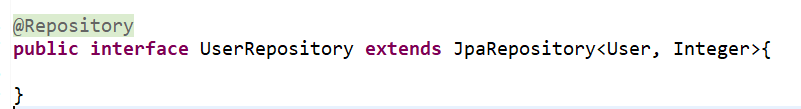
* + 1. Set property in application.properties



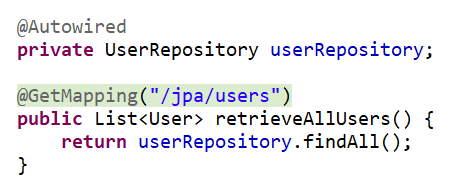
* + 1. Add @Entity at class level to create table for that class and add @Id and @GeneratedValue at field level for making field as id for table.
    2. We can insert some default data for table using data.sql file in resources folder



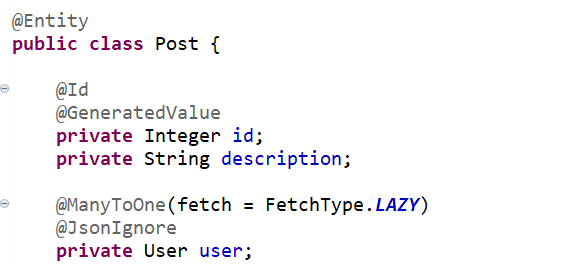
* + 1. We can access h2 console by <http://localhost:8080/h2-console> and mention url used in application.properties file
    2. Create Repository to access database



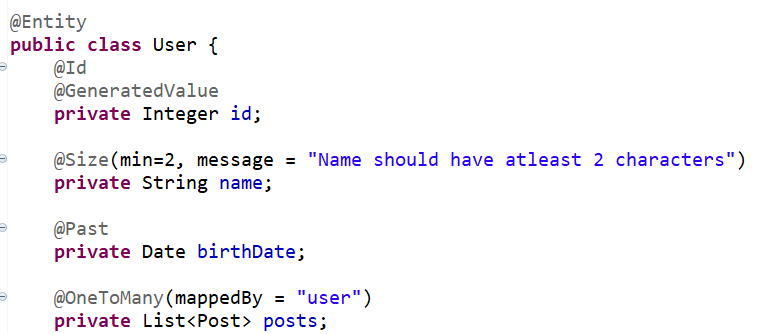
* + 1. @Autowired repository in controller and perform all operation in database



* + 1. Creating relationship between two model, add @ManytoOne or @OnetoOne or @OnetoMany annotation at fields level. Also @JsonIgnore for avoiding looping.

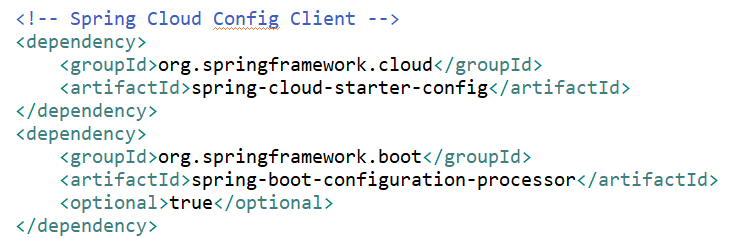


* + 1. Use mappedBy to mention reference field name



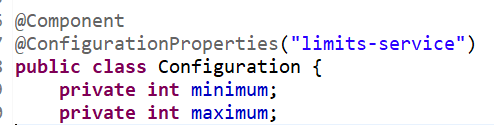
* 1. Richardson Maturity Model
     1. Level 0: Expose SOAP web services in rest style
     2. Level 1: Expose resources(controller) with proper URI
     3. Level 2: Level 1 + HTTP methods
     4. Level 3: Level 2 + HATEOAS (Data + Next possible actions)

1. Microservices with Spring Cloud
   1. Introduction to Microservices
      1. Rest webservices
      2. Small well-chosen deployable units
      3. Cloud enabled (we can manage multiple instances of each microservices)
   2. Introduction to Spring Cloud
      1. Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems.
      2. Common patterns are configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state.
   3. Configuration Management in spring cloud
      1. Spring Cloud Config Client
         1. Add dependency in pom.xml

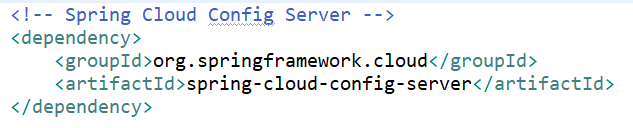


* + - 1. Set **spring.config.import** = **optional:configserver:http://localhost:8888** in application.properties for config server. If we add optional infront then it check with we are getting properties from config server or not. If we receive file then it will use that in configuration class or use local application.properties file.
      2. Config server has high priority than local files.
      3. Create Configuration Class to Fetch properties and mention start name of property.





* + - 1. Then @Autowired configuration file in controller to access properties.
      2. Set application as property file name in application.properties file.
    1. Spring Cloud Config Server
       1. Add dependency in pom.xml



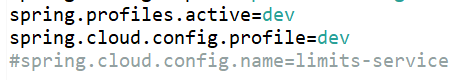
* + - 1. Set application name and port



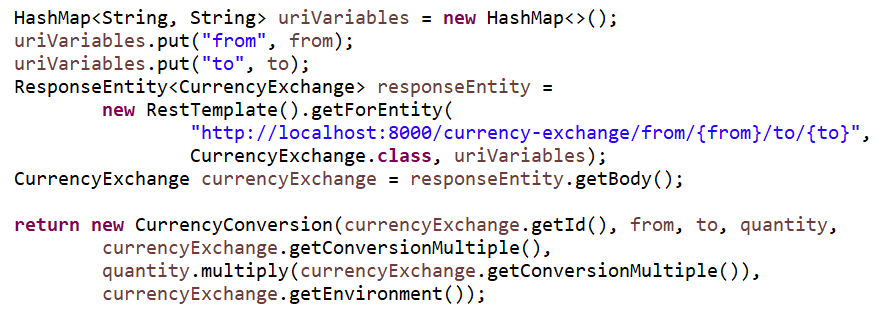
* + - 1. Create **limits-service.properties** file and property into it.
      2. Set git local or Remote uri in application.properties file



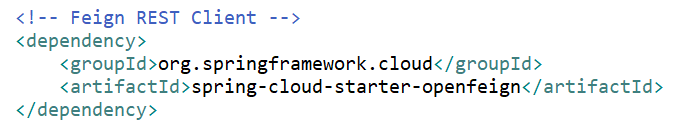
* + - 1. Add @**EnableConfigServer** annotation at class level in Application.java file.
      2. Access properties file of default profile using <http://localhost:8888/limits-service/default>
    1. Configuring Multiple Profiles
       1. Create file limits-service-dev.properties and limits-service-qa.properties.
       2. Access properties file of dev profile using <http://localhost:8888/limits-service/dev>
       3. It returns dev and default profile property, but dev has high priority.
       4. To make active profile in config client and Also name of file we can set below property



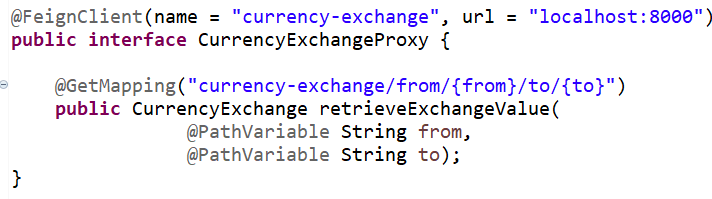
* 1. Create Multiple Instance
     1. Go to run configuration and rename application instance
     2. Make duplicate and rename application instance and add **-Dserver.port=8001**
  2. Invoking One Microservices from another
     1. Use RestTemplate to call microservices



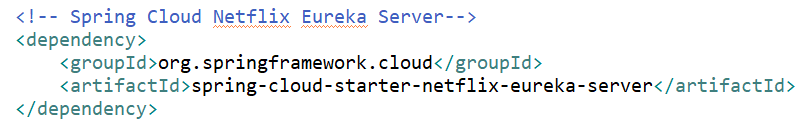
* 1. Using Open Feign REST Client for service Invocation
     1. This is used when we are calling lot of microservices from other microservices.
     2. Add dependency in pom.xml



* + 1. Add **@EnableFeignClients** at class level in Application.java file
    2. Create New Interface For **@FeignClient** and pass application name which we need to call and url as well.



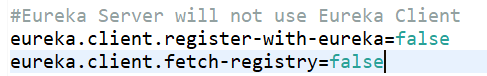
* 1. Service Discovery
     1. Eureka instances can be registered, and clients can discover the instances using Spring-managed beans.
     2. Setting up Eureka Naming Server
        1. Add dependency in pom.xml file



* + - 1. Add @EnableEurekaServer annotation at class level in Application.java
      2. Setting app name and port in application.properties



* + - 1. Mention that Eureka server won’t use Eureka Client



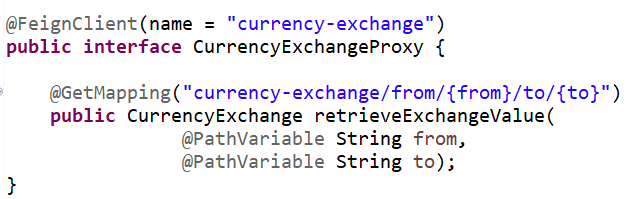
* + - 1. Access Eureka Server using <http://localhost:8761/>
    1. Register microservices to Eureka Server
       1. Add dependency in pom.xml file



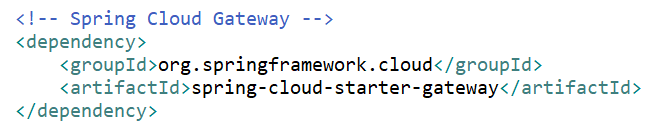
* + - 1. If we run the application it will automatically register in eureka server without any property changes because the default setup for **eureka.client.serviceUrl.defaultZone = http//:localhost:8761/eureka.**
      2. If we used different port for eureka server then we need mention Eureka server url in application.properties file



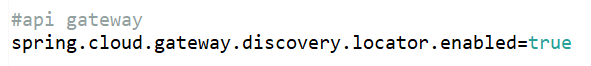
* + 1. Spring Cloud Load Balancer (Load Balancing)
       1. @FeignClient Automatically Finding the webservice instance in Eureka Server and manage the load balancing at client side between multiple instances.
       2. To pick instance url we don’t have to pass url in @FeignClient annotation.



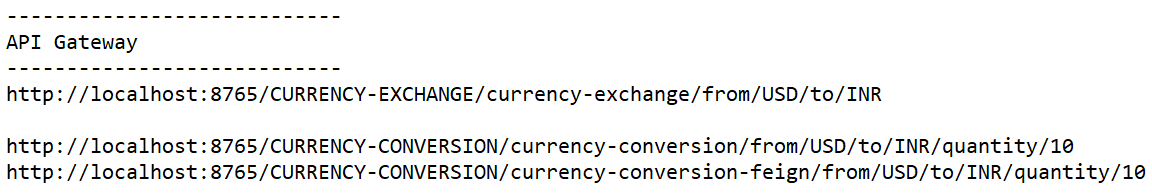
* + 1. For load balancing Eureka client adds one more dependency which is spring cloud load balancer used by Open Feign Client. In previous version of spring cloud we used Ribbon as load balancer.
  1. Spring Cloud Gateway (API Gateway)
     1. Spring Cloud Gateway aims to provide a simple, yet effective way to route to APIs and provide cross cutting concerns to them such as: security, monitoring/metrics, and resiliency.
     2. Built on top of Spring WebFlux (Reactive Approach)
     3. In previous version of spring cloud we used zuul. Now we use Spring Cloud Gateway for API Gateway.
     4. Add dependency in pom.xml file



* + 1. Enable gateway discovery locator in application.properties file



* + 1. Now we can access all microservice using API gateway by using api gateway url and application name mentioned in eureka server



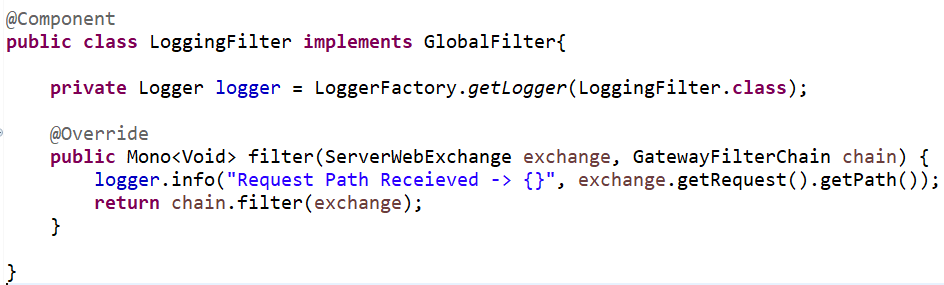
* + 1. We can change application id to lower case by adding property in application.properties file



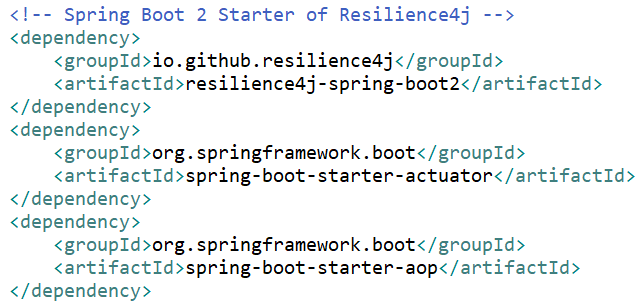
* 1. Exploring Routes with Spring Cloud Gateway
     1. Features of Spring Cloud Gateway:
        1. Routes redirecting based on path
        2. Able to match routes on any request attribute.
        3. Add new parameter and header to routes
        4. Define Predicates and filters are specific to routes.
        5. Path Rewriting
        6. Integrate with Spring Cloud Discovery Client (Load Balancing)



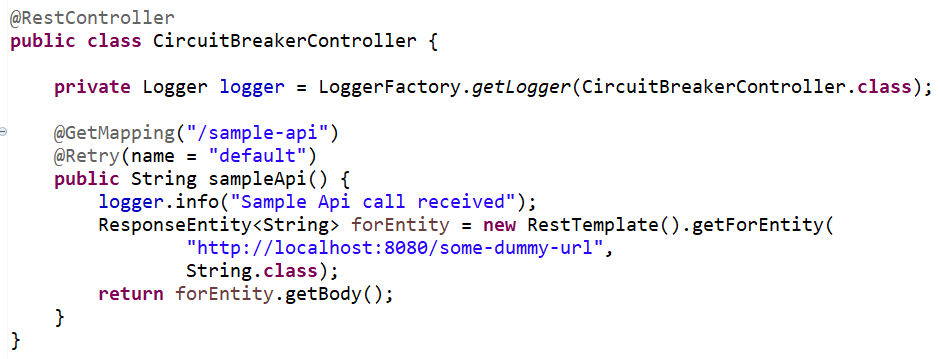
* + 1. Spring Cloud Gateway Logging Filter
    2. Create Class and implements GlobalFilter Interface which provide filter method for implementing cross-cutting like logging.



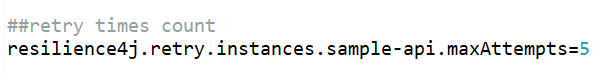
* 1. Circuit Breaker Framework – Resilience4j
     1. The circuit Breaker pattern framework used to fix Microservices call chain (MS1 -> MS2 -> MS3 -> MS4 -> MS5). If any of the MS(Microservice) is down or slow then it will impact whole chain.
     2. In previous version spring cloud for circuit breaker pattern we used Netflix Hystrix.
     3. Features:
        1. We can return a fallback response if a service is down
        2. We can implement a circuit breaker pattern to reduce load
        3. We can retry request in case of temporary failures
        4. We can implement rate limiting
     4. Retry, Circuit Breaker, Rate Limiter and Bulk head we can use at method level (create providing name and same name use in application properties file for instances) and also can use default name (or instance in application properties file).
     5. Setup Resilience4j with spring boot
        1. Add 3 dependency which can be used for implementing Circuit Breaker pattern



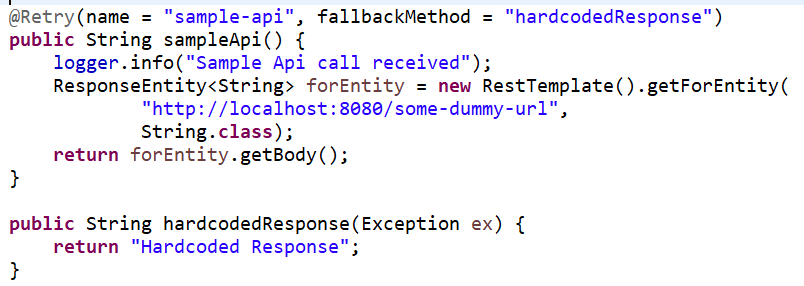
* + 1. Implementing Retry and Fallback methods (Fallback pattern):
       1. We can retry request in case of temporary failure
       2. Add **@Retry(name = "default")** for default implementation of retry (mean it will call 3 times and then return the error if api is down) at method level.



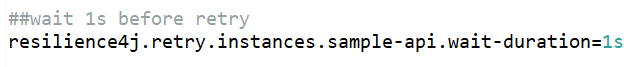
* + - 1. We can change how many times it should retry by adding property in application.properties file



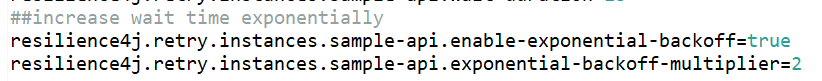
* + - 1. We can add fallback method which will be called and return hardcoded response instead of throwing error.



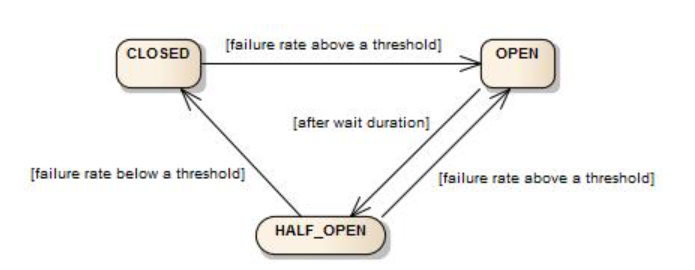
* + - 1. We can create different methods for different type of exception by method overloading like RuntimeException, Exception.
      2. We can configure after how much time it should retry



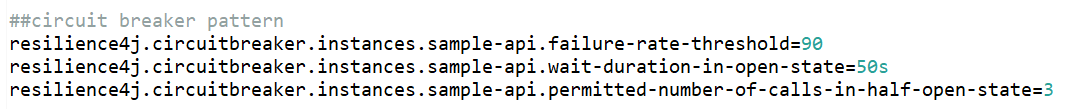
* + - 1. We can increase wait exponentially be setting enable exponential backoff to true and also can mention multiple for exponential.



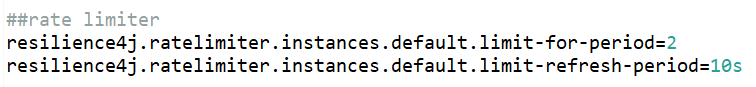
* + 1. Implement Circuit Breaker Features of Resilience4j
       1. The CircuitBreaker is implemented via a finite state machine with three normal states: CLOSED, OPEN and HALF\_OPEN and two special states DISABLED and FORCED\_OPEN.



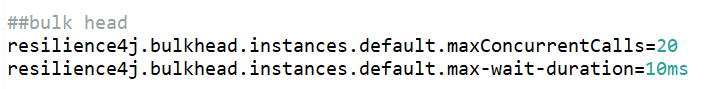
* + - 1. In CLOSED state
         1. Circuit Breaker Start with CLOSED state and stay in CLOSED state until failure rate is below a threshold.
         2. It will call microservice and return the response.
      2. In OPEN state
         1. If microservice goes it down and the failure rate goes above the threshold then it switch to OPEN State.
         2. It always returns the fallback method response and never call the microservice.
      3. In HALF\_OPEN state
         1. If it is in OPEN state and after waiting some duration it switches to HALF\_OPEN state.
         2. Call the microservice and if it fails then return fallback method response.
         3. If failure rate goes above a threshold then it switches to OPEN state again.
         4. If failure rate goes below a threshold then it switches to CLOSED state
      4. We can configure the Threshold and Wait duration and no of call can make in HALF\_OPEN state in application.properties



* + 1. Exploring Rate Limiting and BulkHead Features of Resilience4j
       1. Rate limiting is used to allow some specific api calls during some time (in 10s make only 1000 api calls)
          1. Add @RateLimiter(name = "default", fallbackMethod = "hardcodedResponse") at method level for rate limiter.
          2. Add properties in application.properties file



* + - 1. Bulkhead pattern that can be used to limit the number of concurrent execution.
         1. Add @Bulkhead(name = "default", fallbackMethod = "hardcodedResponse") at method level for bulk head.
         2. Add properties in application.properties file



* 1. Sprint Handler Interceptor
     1. Spring interceptor is a class that either extends the HandlerInterceptorAdapter class or implments the HandlerInterceptor interface.
     2. It has three main methods
        1. preHandle() – called before the execution of the actual handler
        2. postHandle() – called after the handler is executed
        3. afterCompletion() – called after the complete request is finished and the view is generated.
     3. They help to avoid repetitive handler code such as logging and authorization checks.
  2. Difference between @Configuration and @Component
     1. @Configuration is a type-level annotation that indicates that a class declares one or more @Bean methods and may be processed by the Spring container to generate bean definitions and service requests for those beans at runtime. Classes annotated with @Configuration are usually used as sources of bean definitions.
     2. @Component is a class-level annotation that indicates that an annotated class is a "component". Such classes are considered as candidates for auto-detection when using annotation-based configuration and classpath scanning. It is a generic stereotype for any Spring-managed component or bean.