Application A 🡪 can be a .net applicationc

Application B🡪can be a php c ------------------------------🡪 Webservice

Application C 🡪can be a Java application

Application 🡪 Request -🡪 webservice

<---Response<---

So the request and response should be Platform Independent

Popular Formats:

1. XML
2. JSON (Javascript object Notation) – popular data transfer

How does the application A know the format of webservice?

Solution is the **Service Definition** -🡪Response/Request format (json/xml /..)

🡪 Request Structure (format of the request)

🡪 Response Structure (structure of response returned by the service

-🡪 End point (end point url)

1. How can we make web services platform independent?

Like formats like MXL / JSON

2. How does the Application A know the format of Request and Response?

By providing the Service Definition

Service provider is the one which hosts the webservice

Application A is the consumer of the webservice

Transport -🡪 defines how a service is called either HTTP or MQ

REST 🡪 Restful Web Services

Representation State Transfer

HTTP Request methods(Get,PUT,POST)

REST

* Data Exchange Format
  + No Restriction. JSON is popular
* Transport
  + Only HTTP
* Service Definition
  + No Standard. WADL/Swagger/..

What is happening in the background?

Enable debug logging

1. How are our requests handled?

* Mapping servlets: dispatcherServlet urls=[/] 🡪 all the requests goes to the dispatcher servlet
* Auto Configuration(DispatchServletAutoConfiguration) Spring boot automatically configures the dispatchservlet

1. How does the HelloWorldBean object get converted to JSON
   1. @ResponseBody + JacksonHttpMessageConverters default configuration
      1. Auto Configuration(JacksonHttpMessageConvertersConfiguration) This default configuration is automatically configured for REST API for Spring Boot
2. Who is configuring the error mapping?
   1. Auto Configuration(ErrorMvcAutoConfiguration)
3. How are all jars available(Spring, Spring MVC, Jackson, Tomcat)?
   1. Starter Projects – Spring Boot Starter Web (spring-webmvc, spring-web, spring-boot-starter-tomcat, spring-boot-starter-json)

All these are autoconfigures because of two important projects of spring boot ..Starter Projects and Auto Configurations

Request methods for REST API

* GET – Retrieve details of a resource
* POST – Create a new resource
* PUT – Update an existing resource
* PATCH – Update part of a resource
* DELETE – Delete a resource

Response status for REST API

Return the correct response status

* Resource is not found => 404
* Server Exception => 500
* Validation error => 400

Important Response statuses

* 200 -success
* 201 – created
* 204 – No Content
* 401 – Unauthorized (when authorization fails)
* 404 – Resource Not Found
* 500 – Server Error

POST Mapping:

When we create post methods for creating a user, it should return the status code of 201 and returning the location (uri) of the user created would be good for the consumer to verify the inputs

@PostMapping(“/users”)

Public ResponseEntity<User> createUser(@RequestBody User user){

User saveUser = service.save(user);

URI location = ServiceUriComponentsBuilder.fromCurrentRequest()

.path(“/{id}”)

.buildAndExpand(saveUser.getId())

.toUri();

Return ResponseEntity.created(location).build();

}

So when we post a create user request, the response output contain the response status code of 201 and also the location(uri) – <http://localhost:8080/users/4>

**Exception Handling:**

When we retrieve a user, if we enter a non-existing user, we will get a Whitelabel Error Page and also returning Exception traceback.

Public User findOne(int id){

Predicate<? Super User> predicate = user 🡪 user.getId().equals(id);

Return users.stream().filter(predicate).findFirst().get();

}

If no value present then .get() method will throw an NoSuchElementException

Public User findOne(int id){

Predicate<? Super User> predicate = user 🡪 user.getId().equals(id);

Return users.stream().filter(predicate).findFirst().orElse(null);

}

@GetMapping(“/users/{id}”)

Public User retrieveUser(@PathVariable int id){

User user = service.findOne(id);

If (user == null)

threw new UserNotFoundException(“id:”+id);

return user;

public class UserNotFoundException extends RuntimeException{

public UserNotFoundException(message){

super(message);

}

Now also getting WhileLabel Error with UserNotFoundException and id value also

@ResponseStatus(code.HttpStatus.Not\_Found)

public class UserNotFoundException extends RuntimeException{

public UserNotFoundException(message){

super(message);

}

Execute the Request, we will get (type=Not Found, status=404) with WhiteLabel Error Page

To remove this lot of exception tracing ..pom.xml 🡪 comment the spring boot dev tools dependency..so we will get clear exception page. In PROD, we don’t need to do that since we are running a jar file. when you run the packaged applications , dev tools will be automatically disabled.

**ResponseEntityExceptionHandler** – This method handles all Spring MVC raised exceptions

HttpStatus.INTERNAL\_SERVER\_ERROR

HttpStatus.NOT\_FOUND

**Validation: spring-boot-starter-validation**

**@valid**

**REST API DOCUMENTATION**

* **Your REST AI consumers need to understand your REST API**
  + **Resources**
  + **Actions**
  + **Request/Response Structure(Constraints/Validations)**

**Challenges:**

* **Accuracy: How do you ensure that your documentation is upto date and correct?**
* **Consistency: you might have 100s of REST API in an enterprise.**
  + **How do you ensure consistency**

**Options:**

1. **Manually maintain documentation – Additional effort to keep it in sync with code**
2. **Generate from code?**

* **Swagger –**
* **Open API – Standard, language-agnostic interface**
  + **Discover and understand REST API**

**Swagger UI – Visualize and interact with REST API**

**Content Negotiation:**

* **Same Resource – Same URI**
  + **However Different Representations are possible**
    - **Eg: Different Content Type – XML or JSON or**
    - **Eg: Different language – English or Dutch or French**
* **How can a consumer tell the REST API provide what they want?**
  + **Content Negotiation**
* **Eg: Accept header(MIME types application/xml, application/json…)**
* **Eg: Accept-Language header(en, nl,fr..)**

**<dependency>**

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

**<artifactId>Jackson-dataformat-xml</artifactId>**

**</dependency>**

**By default postman/ will return json results But When the user change the HEADER as Accept 🡪 application/xml, then it will return the xml results**

**Internationalization -i18n:**

* **Your REST API might have consumers from around the world**
* **How do you customize it to users around the world?**
  + **Internalization – i18n**
* **Typically HTTP Request Header – Accept-Language is used**
  + **Accept-Language – indicates natural language and locale that the consumer prefers**
  + **Eg: en – English (Good Morning)**
  + **Eg: n1 – Dutch (Goedemorgen)**
  + **Eg: fr – French (Bonjour)**

Versioning REST API

* You have built an amazing REST API
  + You have 100s of consumers
  + You need to implement a breaking changes
    - Eg: Split name into FirstName and LastName
* **SOLUTION: Versioning REST API**
  + URL
  + Request Parameter
  + Header
  + Media Type
* **URI Versioning – Twitter**
  + <http://localhost:8080/v1/person> - first version gives just name
  + <http://localhost:8080/v2/person> - second version give firstname & lastname
* **Parameter Versioning**
  + <http://localhost:8080/person?version=1>
  + <http://localhost:8080/person?version=2>
* **(Custom) headers Versioning – Microsoft**
  + SAME-URL headers=[X-API-VERSION=1]
  + SAME-URL headers=[X-API-VERSION=2]
* Media type versioning(a.k.a “Content negotiation”, or “accept header”) -Github
  + SAME-URL produces=application/vnd.company.app-v1+json
  + SAME-URL produces=application/vnd.company.app-v2+json

@GetMapping(“/v1/person”)

public personv1 getFirstVersonOfPerson(){

return new personV1(new Name(“Bob Charlie”)

}

@GetMapping(“/v2/person”)

public personv2 getSecondVersonOfPerson(){

return new personV2(new Name(“Bob”, “Charlie”)

}

@GetMapping(path=”/person”, params = “version=1”)

public personv1 getFirstVersonOfParamRequest(){

return new personV1(new Name(“Bob Charlie”)

}

@GetMapping(path=”/person”, params = “version=2”)

public personv2 getSecondVersonOfParamRequest (){

return new personV2(new Name(“Bob”, “Charlie”)

}

@GetMapping(path=”/person/header”, header = “X-API-VERSION=1”)

public personv1 getFirstVersonOfRequestHeader(){

return new personV1(new Name(“Bob Charlie”)

}

@GetMapping(path=”/person/header”, header = “X-API-VERSION=2”)

public personv2 getSecondVersonOfRequestHeader(){

return new personV2(new Name(“Bob”, “Charlie”)

}

@GetMapping(path=”/person/accept”, produces = “X-API-VERSION=1”)

public personv1 getFirstVersonOfPersonAcceptHeader(){

return new personV1(new Name(“Bob Charlie”)

}

@GetMapping(path=”/person/accept”, produces = “application/vnd.company.app-v1+json”)

public personv2 getSecondVersonOfRequestHeader(){

return new personV2(new Name(“Bob”, “Charlie”)

}

http headers are never meant to be used for versioning

FACTORS to consider

* URI POLLUTION – There is a lot of urls , but when it comes to header/media, there will be lesser no of urls
* Misuse of HTTP Headers - http headers are never meant to be used for versioning
* Caching – Typically caching is done based on Urls so when it comes to media/header versioning, different version however both versions using the same url. So we cannot do caching based on url , need to look at the headers too.
* Can we execute the request on browser? url and request param can easily execute in browser whereas media/header needs command line utitlity or api
* Api Documentation is easy for the first 2

Summary: There is no perfect solution when it comes to versioning

**HATEOS:**

**Hypermedia as the Engine of Application State(HATEOS)**

**Customizing REST API Responses – Filtering and more**

* **Serialization – convert object to stream(eg:JSON)**
  + **Most popular JSON serialization in Java:Jackson**
* **How about customizing the REST API response returned by Jackson framework?**
* **1. Customize field names in response**
  + **@JSONProperty**
* **2. Return only selected fields**
  + **Filtering**
  + **Eg: Filter out passwords**

**Two Types:**

**Static Filtering : Same filtering for a bean across different REST API**

**@JsonIgnoreProperties, @JsonIgnore**

**Dynamic Filtering:**

**Customize filtering for a bean for SPECIFIC REST API**

**@JsonFilter with FilterProvider**

**Spring Boot Actuator:**

* **Provides Spring boot’s Production-ready features**
  + **Monitor and manage application in your production**
* **Spring Boot Started Actuator:**
  + **Starter to add Spring Boot Actuator to your application**
    - **Spring-boot-starter-actuator**
* **Provides a number of endpoints:**
  + **Beans: Complete list of spring beans in your app**
  + **Health: Application health information**
  + **Metrics: Application metrics**
  + **Mappings: Details around Request Mapping**

**By default** [**http://localhost:8080/actuator**](http://localhost:8080/actuator) **exposes the health of the application**

**To expose more info of the application, application.properties 🡪**

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

**HAL (JSON Hypertext Application Language)**

**Spring-data-rest-hal-explorer**

**Connecting H2-Console:**

* /src/main/resources/application.properties

*spring.h2.console.enabled = true*

*spring.datasource.url=jdbc:h2:mem:testdb*

*spring.jpa.defer-datasource-initialization=true – data.sql is getting executed before our tables are created in H2 database. So we want to delay that with this code,*

**Creating Posts for Users:**

* **Posts REST API**
  + **Retrieve all posts for a user** 
    - **GET /users/{id}/posts**
  + **Create a post for a user**
    - **POST /users/{id}/posts**

***@OneToMany(mapped = “user”) -- >Jakarta.persistence.OneToMany***

***@JsonIgnore***

***private List<Post> posts;***

***@ManyToOne(fetch = FetchType.LAZY) 🡪 Jakarta.persistence.ManyToOne***

***@JsonIgnore***

***private User user***

***application.properties -> spring.jpa.show-sql=true***

***some annotations -@valid @size***

**Connecting mysql database using Docker:**

install docker

docker run –detach

--env MYSQL\_ROOT\_PASSWORD=dummypassword

--env MYSQL\_USER=social-media-user

--env MYSQL\_PASSWORD=dummypassword

--env MYSQL\_DATABASE=social-media-database

--name mysql

--publish 3306:3306

mysql:8-oracle -------🡪 image of the mysql

applications.properties 🡪 spring.datasource.url =jdbc:mysql://localhost:3306/social-media-database

spring.datasource.username= social-media-user

spring.datasource.password= dummypassword

spring.jpa.hibernate.ddl-auto=update -🡪at the start of the application, based on the entities present, the schema will be updated

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect

**Pom.xml:**

*<dependency>*

*<groupId>mysql</groupId>*

*<artifactId>mysql-connector-java</artifactId>*

*</dependency*

**Basic Authentication with Spring Security**

*<dependency>*

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

*<artifactId>spring-boot-starter-security</artifactId>*

*</dependency>*

*once run, we can get a password in the console, we can use that password to login other wise add the following in the application.properties 🡪*

*spring.security.user.name=username*

*spring.security.user.password=password*

when we send a request, spring security will intercepts the request and execute a series of filters. These series of filters are called **Filter Chains**

1. All requests should be authenticated
2. if a request is not authenticated, a web page will shown with username and password
3. one of the thing as part of filter chain is CSRF -> this will impact out POST and PUT request

etc…

we need to customize the filter chain, instead of showing login page, we should show the popup of http basic authentication

We have to override the filter chain by defininig bean and filter chain in configuration file

*SpringSecurityConfiguration.java*

*import static org.springframework.security.config.Customizer.withDefaults;*

*@Configuration*

*public class SpringSecurityConfiguration{*

*@Bean*

*public SecurityFilterChain filterChain(HttpSecurity http) throws Exception*

*# 1. all requests should be authenticated*

*http.authorizeHttpRequests(*

*auth ->auth.anyRequest().authenticate()*

*);*

*# 2. if a request is not authenticated, a web page is shown*

*http.httpBasic(withDefaults()); -🡪 now a pop up will appear when you hit the url for username and password*

*#3. CSRF 🡪 POST, PUT*

*http.csrf().disable()*

**MICROSERVICES:**

* **REST**
* **& small well chosen deployable units**
* **& Cloud Enabled – we can deploy multiple instances of these microservices (By cloud enabled, if there is more load, we can easily bring another instance) -Dynamically scale up and scale down**

**Spring Cloud Netflix:**

* **Integration with various Netflix OSS components (Eureka, Hystrix, Zuul, Archaius, etc..)**

[**https://spring.io/projects/spring-cloud**](https://spring.io/projects/spring-cloud)

**Challenges:**

1. **Configuration Management:**  When there is multiple microservices and multiple environments for each of these microservices and multiple instances in many of those environments. This would mean that there is lot of configuration for these microservices that the operation team needs to manage.
2. **Soluion:** *SpringCloudConfigServer* provides an approach where you can store all the configuration for all the different environments of all the microservices in a Git repository. so you can store all the configurations of different environments of all the microservices in just one place in a centralized location and Spring Cloud Config Server can be used to expose that configuration to all the microservices.
3. **Dynamic Scale Up and Scale Down:**
   1. **Naming Server(Eureka)**
   2. **Ribbon (Client Side Load Balancing)**
   3. **Feign (Easier REST Clients)**
4. All the instances of all the microservices would be registered with the Eureka naming server – it has two important features
   * 1. Service Registration
     2. Service Discovery – it will show the current available instance of the registered service

we can use ***Ribbon OR Spring Cloud Load Balance for the latest version*** for Load-Balancing

1. **VISIBILITY and MONITORING:**
   1. Zipking Distributed Tracing
   2. Netflix API Gateway(Zuul) OR **Spring Cloud Gateway** for the latest version
2. **Fault Tolerance:**
   1. Hystrix – if a service is down, hystrix helps us to configure a default response.
   2. **Resilience4j for the latest version as a Circuit Breaker**

**New Version with Docker: Containerize Microservices**

* Run microservices with Docker and Docker Compose

**Kubernetes:** Orchestrate all microservices with Kubernetes.

**Ports**

| **Application** | **Port** |
| --- | --- |
| Limits Service | 8080, 8081, ... |
| Spring Cloud Config Server | 8888 |
|  |  |
| Currency Exchange Service | 8000, 8001, 8002, .. |
| Currency Conversion Service | 8100, 8101, 8102, ... |
| Netflix Eureka Naming Server | 8761 |
| Netflix Zuul API Gateway Server | 8765 |
| Zipkin Distributed Tracing Server | 9411 |

**For Spring Cloud Config Server:**

**Dependency to add is :**

1. **Spring Boot DevTools**
2. **Config Server**
3. @EnableConfigServer

**in the main application**

**Connect Limits Service (microservice) to Spring Cloud Config Server:**

1. Add the dependency Config Client using Spring Initializer
2. In the pom.xml , we can see **spring-cloud-starter-config** dependency
3. Configure the url of the cloud config server in application.properties -🡪

spring.config.import=optional:configserver:http://localhost:8888

1. What is the name of the application?spring.application.name=limits-service

spring.application.service=limits-service

**if you have multiple configs in the git for DEV, QA , PROD**

**limits-service-dev.properties**

**limits-service-qa.properties**

**limits-service.properties**

**then add the actives in the application.properties**

spring.profiles.active=dev  
spring.cloud.config.profile=dev

[**http://localhost:8888/limits-service/default**](http://localhost:8888/limits-service/default)

[**http://localhost:8888/limits-service/qa**](http://localhost:8888/limits-service/qa)

**Using Feign Rest Client for Service Invocation**

We need to write a tedious/lot of code to talk from one service to another service using RESTTemplate.getEntity and getResponseBody

Spring Cloud provides you with a framework called **Feign, easy to call other microservices**

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-openfeign</artifactId>  
</dependency>

For this we need to create a Proxy

@SpringBootApplication  
@EnableFeignClients  
public class CurrencyConversionServiceApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(CurrencyConversionServiceApplication.class, args);  
 }  
  
}

**Naming Server or Service Registry(Developer Tools)**

**Dependencies**

Spring Boot DevTools(OPS)

Spring Boot Actuator

Eureka Server(SPRING Cloud Discovery)

@EnableEurekaServer

In the naming server micorservices

eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka

in the client microservices

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>  
</dependency>

In client microservices’s pom.xml

**Setting Up Spring Cloud API Gateway**

**Dependencies**

**Eureka Discovery Client(Spring Cloud Discovery)**

**Gateway(Spring Cloud Routing)**

spring.application.name=api-gateway  
server.port=8765  
eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka  
  
spring.cloud.gateway.discovery.locator.enabled=true

**if we want to implement things like authentication, implement via API gateway , you can only allow those things which are authenticated**

spring.cloud.gateway.discovery.locator.lowerCaseServiceId=true

[**http://localhost:8765/currency-conversion/currency-conversion-feign/from/USD/to/SGD/quantity/10**](http://localhost:8765/currency-conversion/currency-conversion-feign/from/USD/to/SGD/quantity/10)

[**http://localhost:8765/currency-exchange/currency-exchange/from/USD/to/SGD**](http://localhost:8765/currency-exchange/currency-exchange/from/USD/to/SGD)

**Exploring Routes through Spring Cloud Gateway**

**To build custom routes, use a configuration file**

@Configuration  
public class ApiGatewayConfiguration {  
 @Bean  
  
 public RouteLocator gatewayRouter(RouteLocatorBuilder builder){  
 return builder.routes()  
 .route(p -> p.path("/get")  
 .filters(f -> f  
 .addRequestHeader("MyHeader", "MyURI")  
 .addRequestParameter("Param", "MyValue"))  
 .uri("http://httpbin.org:80"))  
 .route(p -> p.path("/currency-exchange/\*\*")  
 .uri("lb://currency-exchange"))  
 .route(p -> p.path("/currency-conversion/\*\*")  
 .uri("lb://currency-conversion"))  
 .route(p -> p.path("/currency-conversion-feign/\*\*")  
 .uri("lb://currency-conversion"))  
 .route(p -> p.path("/currency-conversion-new/\*\*")  
 .filters(f -> f.rewritePath(  
 "/currency-conversion-new/(?<segment>.\*)",  
 "/currency-conversion-feign/${segment}"  
 ))  
 .uri("lb://currency-conversion"))  
 .build();  
 }  
}

**Implementing Spring Cloud Gateway Logging Filter:**

@Component  
public class LoggingFilter implements GlobalFilter {  
 private Logger logger = LoggerFactory.*getLogger*(LoggingFilter.class);  
 @Override  
 public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {  
 logger.info("Path of the request received -> {}", exchange.getRequest().getPath());  
 return chain.filter(exchange);  
 }  
}

**Spring Cloud Gateway**

* **Simple yet effective way to route to API**
* **Provide cross cutting concerns:**
  + **Security**
  + **Monitoring/metrics**
* **Built on top of Spring WebFlux(Reactive Approach)**
* **Features:**
  + **Match routes on any request attribute**
  + **Define Predicates and Filters**
  + **Integrates with Spring Cloud Discovery Client (Load Balancing)**
  + **Path Rewriting**

**Need to know about**

**Gateway Client**

**Gateway Handler Mapping**

**Gateway Web Handler**

**Filters**

**…**

**..**

**..**

**Proxied Service -- >** [**https://docs/spring.io**](https://docs/spring.io)

**Getting Started with Circuit Breaker – Resilience4j**

**Circuit Breaker:**

**Microservice1 🡪 Microservice2 🡪 Microservice3 🡪 Microservice4 🡪 Microservice5**

* What if one of the services is down or slow?
  + Impacts the entire chain!
* Questions:
  + Can we return a fallback response if a service is down?
  + Can we implement a Circuit Breaker pattern to reduce load?
  + Can we retry requests in case of temporary failures?
  + Can we implement rate limiting?
* Solutions: Circuit Breaker Framework – Resilience4j

Resilience4j is a light weight, easy-to-use fault tolerance library inspired by Netflix Hystrix

**Retry and Ciruit Breaker features of Resilience4j**