Design Patterns

Algorithms, java programs

Spring microservices

Spring mvc

Restful api

Spring BUS

Spring Cloud

Spring security

K8s

Deploying MSs in AWS

Logging

Junit

Ribbon is a client side LB. it is a software LB.

Add spring-cloud-starter-netflix-ribbon dependency in pom.xml.

Ribbon -> client side LB

Eureka -> Discovery server

Zuul -> API gateway

Feign -> To ease the calling complexity of other microservices from one microservice.

Sleuth -> Distributed tracing of our MS application which gives trace id and span id.

Zipkin -> to send the above tracing logs to zipkin. It is having a UI which gives, how many calls done on a MS, dependencies among each MS.

OAuth -> Security

Hysterix -> fail fast, fall back mechanism.(Resilience)

Minimum 4 requests should be sent in 4 secs and 50% of them should be failed. Then fallback circuit opens. After 60secs this should close and actual circuit should reopen.

**Microservices in action:**

**Learning objectives:**

Split a monolithic application into microservices Store configuration details on cloud using CloudConfig, Load balance requests between microservices using Ribbon, Discover services in cloud using Eureka Increase resilience through Hystrix, Use asynchronous communication to improve performance(RibbitMQ, Kafka), Create a API gateway using Zuul, Simplify REST calls through Feign Secure microservices using OAuth through Spring Cloud Security, Monitor your microservices through Turbine, Sleuth and Zipkin.

**Problems with monolithic applications:** maintenance, scalability, deployment, reliability and also making it cloud native.

* Even for a small change the entire app has to be redeployed thus increasing downtime of the app
* Larger the app, larger the deployment time and startup time
* Even if only specific parts of the application experiences a larger load, we have to deploy the entire app in multiple servers to take care of scaling. This takes up resources and increases maintenance problems
* A large code becomes very intimidating
* We are stuck with chosen technology. If later on we find a particular functionality can be better written using python or scala or C#, modification is extremely difficult.
* In monolithic even for small change, we need to test all functionalities. In microservices, only the changed MS and dependent MSs should test.

**Specifically, you will be able to:**

* Split a monolithic application into microservices.
* Store configuration details on cloud using CloudConfig
* Load balance requests between microservices using Ribbon
* Discover services in cloud using Eureka
* Increase resilience through Hystrix
* Use asynchronous communication to improve performance
* Create a API gateway using Zuul
* Simplify REST calls through Feign
* Secure microservices using OAuth through Spring Cloud Security
* Monitor your microservices through Turbine, Sleuth and Zipkin

**How to get details from one micro service to another?**

* Using RestTemplate.getObjectFor(“rest end point”, “returning object”), we can talk to the relevant end point.
* Create an end point which should send the data and use that end point in the service which need the data.

**Cloud native applications:**

Our applications have to be cloud native, in other words, they should be written in such a way that it can make use of the advantages of cloud ecosystem. Elasticity, availability, security are some of the features we get by deploying our application in a cloud environment.

Below are challenges when we go for cloud native applications.

* Since the services are deployed at random hosts and ports, how can Customer microservice know where to find the other microservices?
* Most of the configuration details of the 4 microservices are similar. How can we avoid this duplication?
* Since in cloud plan and friend-family are dynamically deployed, how will the customer microservice know where to find them?
* When we scale friend-family in cloud, how customer service can load balance the load across multiple instances of them?
* What if while fetching the profile the request friend-family service fails. How to bring in resilience?
* Since the application is now spread across as multiple microservices, how do we trace the flow of a request?

Netflix OSS components are tough to use in its raw form. This is where Spring Cloud comes into the picture. Spring Cloud is actually a suite of projects from Spring. Some of the major projects under the Spring Cloud umbrella project are:

* Spring Cloud Config
* Spring Cloud Netflix
* Spring Cloud Security
* Spring Cloud Sleuth, etc

The Spring Cloud Netflix provides spring integration for the common components from Netflix OSS with Spring Boot

**Spring cloud config:**

* Most of our configuration details of our micro services are common and we are duplicating them in all MSs. If we want to change DB host name, we need to change in all places and need to redeploy all MSs.
* To avoid this, we can place common properties file in cloud and write a spring cloud starter project and create a bootstrap properties file in all our MSs and add this cloud config server project details there to get these details on our application.
* Bootstrap will run before starting our application and get the details.

**Spring cloud config in action:**

1. Create a GIT repository
2. Create a file called application.properties in the repository
3. Place the common properties in application.properties file
4. Create separate properties file for each microservice with name of the file matching the spring.application.name of the service
5. Place the respective properties in their respective files
6. Create a spring starter project for config server with relevant dependencies(spring-cloud-dependencies and spring-cloud-config-server)
7. Add information of the git server in the properties file of config server as shown below.

**server.port=1111**

**spring.application.name=ConfigServer**

**spring.cloud.config.server.git.uri=https://github.com/PlaygroundConfiguration/SpringMicroservices.git**

1. Add **@EnableConfigServer**annotation in the application file of config server
2. Add relevant dependencies to all the microservices(spring-cloud-dependencies and spring-cloud-starter-config)
3. Create a bootstrap.properties file in each microservice with a property for the config server and the database credentials as below.

**spring.cloud.config.uri=http://localhost:1111**

**spring.datasource.username=root**

**spring.datasource.password=root**

* Remove database connection details from the application.properties of individual services .

**Note:**

* The config-server is contacted by the clients only once, during the start of the project. Therefore any changes made to the configuration after the application starts will not be reflected in the application.
* We can also configure the clients retry attempts to contact the config server using the properties like,
* Initial interval defaulter to 1000ms
* Multiplier for next interval of 1.1 times defaulter
* Maxinterval defaulted to 2000ms
* Max attempts defaulted to 6 times.
* If a port is not specified for the config server, it runs in its default port 8888. Also, if the cloud-config server is down, then the client will throw an error not during startup, but while trying to access a property at runtime. To avoid this we can have the failFast property set to true. By this the client will fail at startup time rather than at run time.
* **spring.cloud.config.failFast=true**
* Also, in order to avoid config-server to be a single point of failure, we usually deploy multiple instances of it to ensure high availability. If the cloud config server is unavailable, it will use the properties files in the individual applications as a fallback

**Dt: 06/07/2019:**

**Load balancing:**

* If we increase the number of instances, the infytel-customer has to send requests to both these instances so that the load is properly balanced.
* One way is to put a load balancer in front of the infytel-friend-family instances. This is usually a hardware load balancer. This is server-side load balancing.
* Server-side load balancing has several problems:
* If the load balancer fails, then we don’t have access to any of the instances of the microservice
* Since each microservice would have a dedicated load balancer, we have to manage, track and maintain hundreds of such load balancers.
* It increases network latency. Now it would take two hops to reach the service. One to the load balancer and another from the load balancer
* Client-side load balancing is a natural solution to this. The client is responsible for deciding to whom it will send the request to. The client-side load balancers are thus software load balancers and not traditional hardware load balancers. Of course, the downside is we are mixing our application code with load balancing code.
* Spring-Cloud Netflix has a client load balancer called Ribbon. We will use Ribbon to perform client-side load balancing in our application.

**Ribbon in action:**

* Add dependency in the infytel-customer

**<dependency>**

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

**<artifactId>spring-cloud-starter-netflix-ribbon</artifactId>**

**</dependency>**

* Create a Configuration class with the below bean

@Configuration

**public** **class** LoadBalRibbonConfig {

**@Bean @LoadBalanced**

**public RestTemplate restTemplate() {**

**return new RestTemplate();**

**}**

**}**

* Autowire the RestTemplate as @Autowired RestTemplate template; in the CustomerController
* Add **@RibbonClient(name="custribbon",** configuration=LoadBalRibbonConfig.**class) annotation on the CustomerController class**
* Add the below properties in the properties file:

**custribbon.ribbon.eureka.enabled=false**

**custribbon.ribbon.listOfServers=http://localhost:8001,http://localhost:8002**

**Note:** We should mention only server details with port number but shouldn’t mention complete end points. Ribbon can’t identify the end points.

* Autowire the rest template and update the infytel-friend-family invocation in CustomerController as

List<Long> friends = template.getForObject("http://custribbon/customers/" + phoneNo+"/friends", List.class);

* Run the application, with the two instances of FriendFamilyMS running in two different ports.
* You can create multiple instances by right click on micro service in STS and click on Duplicate Config option.
* Then right click on the newly created config and click on open config. Then edit the configurations like server port and name.
* Do the same for the first MS as well.

**Note:** in the above configuration, if one MS config is down, Ribbon can’t identify it and try to send the requests to down instance as well in round robin, so 50% of the requests will fail.

By default ribbon uses the **NoOpPing**strategy for checking if the services are up. However, the NoOpPing is a dummy strategy. It assumes that all services are up. Thus it will keep pinging the services even if they are down. We can configure the Ping strategy so that we stop sending requests to services which are down.

Also, Ribbon by default uses Round Robin load balancing strategy.

These things can be modified by adding a configuration file.

* Ribbon can be configured as:

**custribbon.default.NFLoadBalancerRuleClassName=com.netflix.loadbalancer.RandomRule**

**Service discovery – Eureka:**

* In our CustomerMS, we had got the FriendMS URI from the cloud-config server. In a cloud situation, a instance may be provisioned and deprovisioned randomly. At one time the service may run in one port and after sometime, the cloud provider may shift it to another.
* How can infytel-customer find out the current port and host where infytel-friend-family is running?
* Even if we mention the port and path in a cloud-config server, it will not help our case as the values change dynamically. The service cannot go and modify the properties which are in GIT every time it changes and refresh all the relevant microservices. That will be very cumbersome.
* Service Registry could be the solution for this. We need to register our service in SR.
* In this pattern, a service registers itself with a central server called the Service Registry. Now once it registers itself with the Service Registry two things happen:
* Its details like name, port, host, etc are stored in the service registry
* A list of other registered services become available to it
* Thus even if one if the services were to get redeployed at a different host and port, the other services need not worry about it. When the service redeploys, it would simply update its information in the service registry again. The other services would discover about its updated details through the service registry.
* There are many service registry solutions like Netflix Eureka, etcd, ZooKeeper, consul, etc. In this course we will learn about Netflix Eureka.

**Eureka in action:**

* Create a Spring Starter project with name infytel-eureka
* Add the below dependencies:

**<dependencyManagement>**

**<dependencies>**

**<dependency>**

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

**<artifactId>spring-cloud-dependencies</artifactId>**

**<version>Greenwich.RELEASE</version>**

**<type>pom</type>**

**<scope>import</scope>**

**</dependency>**

**</dependencies>**

**</dependencyManagement>**

**<dependency>**

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

**<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>**

**</dependency>**

* Add the below properties in the application.properties file of infytel-eureka

**spring.application.name=Eureka1**

**server.port=5555**

**eureka.client.fetch-registry=false**

**eureka.client.register-with-eureka=false**

**eureka.client.service-url.defaultZone=http://localhost:5555/eureka**

* Add @EnableDiscoveryServer/@EnableEurekaServer annotation in the application file of infytel-eureka
* Add the below dependencies in the microservices:

**<dependency>**

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

**<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>**

**</dependency>**

* Add @EnableDiscoveryClient in all microservices application file
* Add the below property in the application.properties file in git

**eureka.client.service-url.defaultZone=http://localhost:5555/eureka**

* Autowire Discovery client in CustomerController class as @Autowired DiscoveryClient client;
* Remove the String friendUri; from CustomerController and update the code in accessing the friend-family-service as:

**List<ServiceInstance> instances=client.getInstances("FriendFamilyMS");**

**ServiceInstance instance=instances.get(0);**

**URI friendUri = instance.getUri();**

* **Note**: The service discovery happens through the spring.application.name value of the services. Hence they should not change.
* By hitting localhost:5555, you can see the Eureka dashboard which shows what are all the services registered with it.
* By hitting localhost:5555/eureka/apps/, you can see all the host, port and other details of all registered services.

**Ribbon with Eureka:**

* Remove Autowire of DiscoverClient and Autowire load balanced RestTemplate
* Remove code for getting URI from discovery client
* Access the PlanMS and FriendMS through rest template object and use the names of the service instead of the URI. Since Eureka is used, the URI will be picked automatically based on the service name.

**PlanDTO planDTO=template.getForObject("http://PLANMS"+"/plans/"+custDTO.getCurrentPlan().getPlanId(), PlanDTO.class);**

**List<Long> friends=template.getForObject("http://FRIENDFAMILYMS"+"/customers/"+phoneNo+"/friends", List.class);**

* If we run this application, eureka will always route to only one instance of friend and family because, InstanceId in Eureka dashboard would be unique and the 2nd instance of F&F will overriding the first one. So only one instance will be available at Eureka.
* To avoid this, in GIT we need to add below property for F&F application.properties.

**Eureka.instance.instance-id = ${spring.cloud.client.hostname}:{spring.application.name}:{spring.application.instance-id:${random.value}}**

**Eureka Cluster:**

* Eureka is rarely run as a single instance, as it would become a single point of failure. Typically we run multiple instances of Eureka forming a cluster. In a cluster, each Eureka server replicates the information in the other servers.
* Open the hosts file in C:\Windows\System32\drivers\etc
* Add the below hostnames:

**127.0.0.1 Eur1**

**127.0.0.1 Eur2**

**127.0.0.1 Eur3**

* Use the below yml file in the infytel-eureka server

**spring:**

**profiles: Eureka1**

**application:**

**name: Eureka**

**server:**

**port: 2222**

**eureka:**

**instance:**

**hostname: Eur1**

**client:**

**registerWithEureka: true**

**fetchRegistry: true**

**serviceUrl:**

**defaultZone: http://Eur2:2223/eureka/,http://Eur3:2224/eureka/**

**---**

**spring:**

**profiles: Eureka2**

**application:**

**name: Eureka**

**server:**

**port: 2223**

**eureka:**

**instance:**

**hostname: Eur2**

**client:**

**registerWithEureka: true**

**fetchRegistry: true**

**serviceUrl:**

**defaultZone: http://Eur1:2222/eureka/,http://Eur3:2224/eureka/**

**---**

**spring:**

**profiles: Eureka3**

**application:**

**name: Eureka**

**server:**

**port: 2224**

**eureka:**

**instance:**

**hostname: Eur3**

**client:**

**registerWithEureka: true**

**fetchRegistry: true**

**serviceUrl:**

**defaultZone:** <http://Eur1:2222/eureka/,http://Eur2:2223/eureka/>

* Comma separated values in the defaultZone indicate peer awareness. Eureka1, Eureka2 and Eureka3 are peers of each other and hence will replicate the details across each other.
* Update the application.properties file in GIT for the below property:

**eureka.client.service-url.defaultZone=http://Eur1:2222/eureka,http://Eur3:2223/eureka,http://Eur3:2224/eureka**

* Run all the three profiles of the Eureka server and restart all the microservices
* You will get three dashboards in three different Eureka ports. Since we have a cluster, each dashboard will have the same details of microservices as the other two Eureka Servers in the cluster.
* Bring down a microservice. You will find that since each Eureka server in a cluster replicates itself, all Eureka servers in the cluster will now have the same updated information.

**Resilience:**

* The better approach would be that if a particular service is taking more time than usual, then don’t send anymore requests to that service. Stop requests to that service. This prevents an increase in slowing down of other services. This pattern of not sending request after a certain time period in fault tolerance is called as the Circuit breaker pattern.
* When does a fuse trip in a house? When the current flow exceeds a threshold. Similarly, in microservices communication, when the number of errors in a given time frame is beyond an acceptable limit, the circuit opens, thereby preventing further flow and protecting other parts of the application.
* Using Netflix Hystrix, we can apply a circuit breaker pattern in our application. Hystrix will open the circuit when the numbers of failures in a given time frame are more. Hystrix uses the Fail Fast approach. It is better to fail fast than to fail big time later.
* After opening the Circuit, Hystrix will attempt to close the circuit again. Just like you coming and checking the bike in the evening, Hystrix checks if there is any change in the status quo by sending a single request again. If that fails, it opens the circuit again and waits again.
* The error threshold, waiting time, retry attempts, etc are all configurable in Hystrix.
* Hystrix allows you to mention any alternate piece of code that you wish to run if a service is down. Obviously you don’t get the same result as you wish you had. But, providing some form of data instead of an error is better.

**Resilience in action with the help of Hystrix:**

* Add dependency in the infytel-customer service

**<dependency>**

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

**<artifactId>spring-cloud-starter-netflix-hystrix</artifactId>**

**</dependency>**

* Add the @**EnableCircuitBreaker** annotation in the application class of infytel-customer. This tells Spring Cloud that the application uses the Circuit Breaker pattern
* Create a service class called CustomerCircuit with @**service** annotation
* Add the below method in the service class with @**HystrixCommand** annotation. This annotation makes the method call fault tolerant. This annotation will work only if the class is component or service

**@HystrixCommand(fallbackmethod="getFriendsFallback")**

**public List<Long> getFriends(Long phoneNo){**

**List<Long> numbers= template.getForObject("http://FriendFamilyMS/friends/"+phoneNo, List.class);**

**return numbers;**

**}**

* Create the below method in the same class:

**public List<Long> getFriendsFallback(Long phoneNo){**

**return new ArrayList<Long>();**

**}**

* Add the below properties in the CustomerMS.properties

**hystrix.command.default.circuitBreaker.requestVolumeThreshold=10**

**hystrix.metrics.rollingStats.timeInMilliseconds=10000**

**hystrix.command.default.circuitBreaker.errorThresholdPercentage=50**

**hystrix.command.default.circuitBreaker.sleepWindowInMilliseconds=10000**

* Add @Autowired CustomerCircuit circuit; in the CustomerController
* Update the call from CustomerController as shown below:

**List<Long> friends=circuit.getFriends(phoneNo);**

* Once the specified number of requests (threshold volume) are sent within the specified time (rolling stats time window) and if the specified percentage of requests end up as errors (error percentage), the hystrix opens the circuit. Once the circuit is open, no more requests will be sent to the infytel-friend-family service. After a specified time interval ( sleep window ), hystrix will again close the circuit and pass one request. If that request fails, then again circuit is automatically closed for the specified time again. This repeats in a cycle.
* The customer microservice talks to both the friend-family service and the plan service. These communications are happening synchronously. That means, only after friend-family service request completes, we can send the request to the plan service. In situations where one service can be completed independent of the other, we can use asynchronous communication. Though Asynchronous communications reduce overall time, we cannot use it indiscriminately. For example, it cannot be used if one service depends on the data from another service call.

**Asynchronous calls in action:**

* Create the below method in the hystrix service class:

**public Future<PlanDTO> getSpecificPlans(int planId) {**

**return new AsyncResult<PlanDTO>() {**

**@Override**

**public PlanDTO invoke() {**

**return template.getForObject("http://PLANMS"+"/plans/"+planId, PlanDTO.class);**

**}**

**};**

**}**

* The method returns a Future. Future is a datatype which indicates that the result will be available in future.
* To return a future value, the method returns a AsyncResult. AsyncResult is an abstract class. Hence we are returning an anonymous implementation of it, while overriding its invoke method. The invoke method does the actual code execution.
* In the CustomerController, we process the result of the Future as :

**Future<PlanDTO> planDTOFuture=hystService.getSpecificPlans(custDTO.getCurrentPlan().getPlanId());**

**custDTO.setCurrentPlan(planDTOFuture.get());**

**API Gateway – Zuul:**

* In our application, the UI application has to send requests to the different microservices directly. But since the microservices are in cloud, the host and port are changing frequently. Hence it is not sustainable for the UI application to talk to the microservices directly.
* The better approach is the UI application sends its request to a proxy server which then forwards the request to appropriate microservices. This type of proxy is also called as reverse proxy.
* There are several problems with this approach:
* The client is now responsible for both gathering, aggregating and formatting the content
* The client must know the port and host of the services. If they change constantly then the client code also has to change
* If we decide to refractor the services later on by merging two services together, it would break the client
* The services may not always respond back in HTTP protocols. Thus the protocols may be incompatible with a browser
* The number of requests fired by the client is more
* Different clients need different data.
* The API gateway pattern is a solution to the problems of a client invoking microservices. In a API gateway pattern you have a API gateway server which comes in between the client and the services.
* The API gateway performs complex tasks including:
* Intelligent routing. When a request comes from the client it intelligently routes it to the appropriate services
* Client Specific API. A mobile client and a web client need different treatment and an API gateway provides specific API’s for each client
* It also does:
* Request Aggregation. Based on a single request from the client, it invokes multiple services, aggregates the result and sends it back
* Protocol translation. It will be responsible for taking data from a service through let’s say AMQP and sends the data to the client over HTTP
* Security
* Load balancing

**Zuul in action:**

* Create a new spring boot starter application called infytel-zuul
* Add the below dependencies apart from the dependency management, eureka and cloud config dependencies

**<dependency>**

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

**<artifactId>spring-cloud-starter-netflix-zuul</artifactId>**

**</dependency>**

* Add the below annotations in the application file of infytel-zuul

**@EnableZuulProxy**

**@EnableDiscoveryClient**

* Routes can be added using the zuul.routes.<routeName>.path=/<URI> and zuul.routes.<routeName>.service-id=<ServiceName> . by using strip-prefix=false we can avoid repetition in the path.
* /<URI>/\*\* will match anything after the given URI. Sometimes a single \* can be used as a wildcard  character : /URI/\*/hello.
* Add the below routes to the properties file

**spring.application.name=ZuulServer**

**server.port=8001**

**zuul.routes.customer\_create.path=/customers**

**zuul.routes.customer\_create.service-id=CustomerMS**

**zuul.routes.customer\_login.path=/customers/\*/login/\***

**zuul.routes.customer\_login.strip-prefix=false**

**zuul.routes.customer\_login.service-id=CustomerMS**

**zuul.routes.customer\_profile.path=/customers/\***

**zuul.routes.customer\_profile.strip-prefix=false**

**zuul.routes.customer\_profile.service-id=CustomerMS**

**zuul.routes.friends\_customer.path=/customers/\*/friends**

**zuul.routes.friends\_customer.strip-prefix=false**

**zuul.routes.friends\_customer.service-id=FriendMS**

**zuul.routes.calldetails\_customer.path=/customers/\*/calldetails**

**zuul.routes.calldetails\_customer.service-id=CallDetailsMS**

**zuul.routes.plan.path=/plan/\*\***

**zuul.routes.plan.service-id=PlanMS**

**zuul.ignored-patterns.customer=/customerms/\*\***

**zuul.ignored-patterns.plans=/planms/\*\***

**eureka.client.service-url.defaultZone=http://localhost:5555/eureka**

* Update the UI application code to use the host of Zuul
* Run the application

**Zuul with Ribbon and Hystrix:**

* Zuul automatically uses Ribbon. Thus all requests through Zuul are load balanced. Zuul also automatically uses Hystrix for resilience. We can add fallbacks for Zuul routes.
* The below is one of the common set of configurations you can have on Zuul:
* Adding a prefix:

**zuul.prefix=/infytel**

* Through this configuration, all requests must be prefixed with infytel. Ex:http://localhost:3333**/infytel/**customers/9009009001
* By default the route name in the configuration is the service name registered with Eureka server. Hence in the below given configuration, any request with path **/customers/\*** will be routed to **CustomerMS**.

**zuul.routes.CustomerMS.path=/customers/\***

**zuul.routes.CustomerMS.strip-prefix=false**

* Ex: http://localhost:3333/customers/9009009001
* **Modifying the route name:**
* In the below given example, the configured route name is **"customer"** and also the **service-id** is explicitly configured to handle the requests routed through this route. Hence all the requests with the url pattern **/customers/\*** is redirected to **CustomerMS**based on the configured **service-id**.

**zuul.routes.customer.path=/customers/\***

**zuul.routes.customer.strip-prefix=false**

**zuul.routes.customer.service-id=CustomerMS**

* Ex: http://localhost:3333/customers/9009009001

**Zuul with hystrix:**

* Create the below configuration class.

**@Configuration**

**public class ZuulConfigClass implements FallbackProvider {**

**public String getRoute() {**

**return null;**

**}**

**public ClientHttpResponse fallbackResponse(String route, final Throwable cause) {**

**if (cause instanceof HystrixTimeoutException) {**

**return response(HttpStatus.GATEWAY\_TIMEOUT);**

**} else {**

**return response(HttpStatus.INTERNAL\_SERVER\_ERROR);**

**}**

**}**

**public ClientHttpResponse response(final HttpStatus status) {**

**return new ClientHttpResponse() {**

**public HttpHeaders getHeaders() {**

**HttpHeaders headers = new HttpHeaders();**

**headers.setContentType(MediaType.TEXT\_PLAIN);**

**return headers;**

**}**

**public InputStream getBody() throws IOException {**

**return new ByteArrayInputStream("Sorry. Something went wrong".getBytes());**

**}**

**public String getStatusText() throws IOException {**

**return "OK";**

**}**

**public HttpStatus getStatusCode() throws IOException {**

**return HttpStatus.OK;**

**}**

**public int getRawStatusCode() throws IOException {**

**return 200;**

**}**

**public void close() {**

**}**

**};**

**}**

**}**

* The class should implement the org.springframework.cloud.netflix.zuul.filters.route.FallbackProvider interface. This bean should provide the body for two abstract methods getRoute() and  fallbackResponse(). getRoute() method should return "\*" or null if we wish to apply this for all routes. Else we should mention the specific route name alone.
* The other method should be fallbackResponse() which return a ClientHttpResponse() object.

**Declarative Client – Feign – Replacement of RestTempate to call other MS:**

* We have been using RestTemplate to talk to other microservices. But there are several problems with that approach:
* You have to be aware of the various methods of the Rest Template API to use it.
* You need a separate bean for Load balancing
* You need a separate service for Circuit breaker
* The header details of a request from Zuul are not forwarded to the other microservices using RestTemplate
* Hence we need a form of contacting other microservices which makes it easier by avoiding the above mentioned problems
* Feign is a declarative client from Netflix. It is declarative because we as developers declare the api's for contacting other microservices. We define the rules in the form of our own interfaces. At runtime, Feign will create implementation for our interfaces automatically. Thus with minimal code and self made interfaces, we can have greater control on how one microservice communicates with the other.
* Feign automatically uses Ribbon. Thus all calls are automatically load balanced. Feign also works well with Hystrix. With appropriate dependency and configuration, Feign will automatically use circuit breaker and fallback for all calls without the need for a separate service class.

**Feign in action:**

* Add Feign dependency in Customer service
* Add @EnableFeignClients annotation to Customer main class.
* Create an interface

**@FeignClient("PlanMS")**

**public interface CustPlanFeign {**

**@RequestMapping(value="/plans/{planId}")**

**PlanDTO getSpecificPlan(@PathVariable("planId") int planId);**

**}**

* The interface must be annotated with @FeignClient and must mention the service name for which we are writing the Feign Client.
* We need not provide an implementation for this interface. The framework will provide a dynamic implementation at run time.
* Autowire the feign client as:

**@Autowired**

**CustPlanFeign planFeign;**

* Invoke the plan microservice as

**planFeign.getSpecificPlan(planId);**

**Distributed Tracing using Sleuth and Zipkin:**

* Our request travels around in our application, going from one microservice to another. For example, a request to customers profile goes to infytel-customer, infytel-friend-family and infytel-plan. In a large application, we may have hundreds of microservices and a request may go though quite a bunch of them. We would need to trace the flow of request so that we can find which functionalities are most requested and where our request spends the most time in.
* Spring Cloud Sleuth is one of the projects under Spring Cloud which allows us to trace a request. Sleuth adds two things: a **traceId** and a **spanId** in our logs. TraceId is a unique ID generated for every request. Span ID is a unique ID generated for span or path of a single microservice. So if a request flows through two microservices, it will have one TraceId and two SpanId's. One span is created for the network hop and the other is for the application execution.
* The below image shows a sample log statement. The First Parameter is the Service name, the second is the traceId, third is the span Id. The fourth parameter is false as we have not yet linked it with Zipkin. We will look at Zipkin later.



* The only thing we need to add is the below dependency in all the microservices

**<dependency>**

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

**<artifactId>spring-cloud-starter-sleuth</artifactId>**

**</dependency>**

* Sleuth generates the traceId and spanId. But it is difficult to make use of that data in its raw form. This is where Zipkin is used. Zipkin is a distributed tracing system which allows us to gather information on the trace of a request. By adding appropriate dependency, we make sleuth send all its details to a Zipkin server. The Zipkin server has an in memory DB where the details are stored. Zipkin has a powerful UI application, which allows to analyse the logs and take appropriate action.

**Zipkin in action:**

* Create a Spring Started project with name infytel-zipkin
* Apart from the relevant dependency management, add the below dependencies:

**<dependency>**

**<groupId>io.zipkin.java</groupId>**

**<artifactId>zipkin-autoconfigure-ui</artifactId>**

**<scope>runtime</scope>**

**</dependency>**

**<dependency>**

**<groupId>io.zipkin.java</groupId>**

**<artifactId>zipkin-server</artifactId>**

**</dependency>**

* Add the @EnableZipkinServer annotation to the infytel-zipkin application file
* Configure zipkin to run in port 9411
* Add the below dependencies in all the microservices:

**<dependency>**

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

**<artifactId>spring-cloud-starter-zipkin</artifactId>**

**</dependency>**

* Add below two cofigurations in application.properties in GIT.

**Spring.sleuth.sampler.percentage = 1.0**

**Spring.zipkin.baseUrl =** <http://localhost:4444>

**SECURIRTY:**

* Securing a monolith is very different from securing a Microservice application. In a monolith the client access the application providing the credentials and this is used in a session and checked with a database. As long as the user data is in session, he is authorized to access the functionalities exposed by the app.
* In a microservice, the client contacts only the API gateway server. Though securing the API gateway is good, it is not good enough against internal threats. The services cannot know the source of the request. They cannot determine if the request is coming from a user or another service or some malicious code.

**OAuth security in action:**

* Register the zuul api gateway with infy GIT security provider.
* Add the below dependency in the infytel-zuul server

**<dependency>**

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

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

**</dependency>**

**<dependency>**

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

**<artifactId>spring-cloud-starter-oauth2</artifactId>**

**</dependency>**

* Add the below annotations to the infytel-zuul application

**@EnableOAuth2Sso**

**@EnableZuulProxy**

* Add the below to the application.yml file

**client:**

**clientId: bd1c0a783ccdd1c9b9e4**

**clientSecret: 1a9030fbca47a5b2c28e92f19050bb77824b5ad1**

**accessTokenUri: https://github.com/login/oauth/access\_token**

**userAuthorizationUri: https://github.com/login/oauth/authorize**

**clientAuthenticationScheme: form**

**resource:**

**userInfoUri: https://api.github.com/user**

* Run the application

**Securing individual micro services:**

* Add the below dependencies to all microservices

**<dependency>**

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

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

**</dependency>**

**<dependency>**

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

**<artifactId>spring-cloud-starter-oauth2</artifactId>**

**</dependency>**

* Add the @EnableResourceServer annotation to all microservices application file
* Add the below configuration in the application.yml file

**~~client:~~**

**~~accessTokenUri: https://github.com/login/oauth/access\_token~~**

**~~userAuthorizationUri: https://github.com/login/oauth/authorize~~**

**~~resource:~~**

**~~userInfoUri:~~** <https://api.github.com/user>

**security:**

**oauth2:**

**resource:**

**UserInforUri:** <http://infygit.ad.infosys.com/api/v3/user>

* We are not providing the other details because, we don’t want user to directly hit individual micro service. Everyone should go through our API gateway Zuul.
* if anyone tries to access individual MS, it won’t navigate the user but give error message saying “Full authentication is required to access this resource”.
* When we secure individual microservices, we face the issue of secure communication between these services. The authentication header which flows from the Zuul server to the microservice does not get automatically forwarded when one microservice talks to another. We have to programaticaly extract the headers received from Zuul and add it when we make a request through RestTemplate.
* This means when we are trying access a MS which is internally uses another MS which is individually secured, we will get 500 error saying “Whitelabel Error Page”.
* For this we need to do 3 changes in customer controller.
* In the method where we are connecting to another MS, we need to add one extra argument “@RequestHeader HttpHeaders headers” as a first argument.
* Add the below line to get entity object.

**HttpEntity<String> entity = new HttpEntity<String>(headers);**

* Instead of using template.getForObject, use template.exchange which takes HttpMethod.GET, entity as 2nd and 3rd arguments along with existing arguments and return ResponseEntity object.

**ResponseEntity<PlanDTO> planDTO = template.exchange(“**<http://planMS/plans/>**”+custDTO.getCurrentPlan().getPlanID(), HttpMethod.GET, entity, PlanDTO.class);**

**Securing spring cloud config:**

* While we have secured the individual services, we also need to secure the configuration server so that one does not directly access the property files by using the endpoints of the config server. This can be done by using basic authentication and configuring username and password for it.
* Add below dependency to spring cloud project.

**<dependency>**

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

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

**</dependency>**

* In application.properties, add below properties.

**Security.user.name=username**

**Security.user.password=password**

* It asks for the credentials when user tries to access config server directly.
* But with this, all other MSs which are using this service won’t start as we have secured it.
* To avoid this, in all MSs, go to bootstrap.properties where we have mentioned config server uri, provide the username and pwd as below.

**Spring.cloud.config.uri=http://<username>:<password>@localhost:1111**

**===========================================================================**

* We are storing sensitive information, including DB credentials, in a public repository. This is not very secure. The solution is to encrypt the values and place them in Git. Once encrypted we can either make the config server decrypt the values for us or it can be left to the individual services to decrypt it themselves.
* In config server, create bootstrap.properties and add below entry and start the config server.

**Encrypt.key=infy123**

* Now open the postman and send an endpoint /encrypt (<http://localhost:1111/encrypt>). Under Auth tab, select basic authentication and provide the uname/pwd we have created before.
* This is a POST request and under Body tab, take raw data and provide the value to be encrypted. Here it is our DB password(root).
* Once we hit the send request, we will get the encrypted value of it.
* Go to GIT application.properties, replace the root with this encrypted value by giving {cipher}. This is because, we are telling our config server to decrypt this with the key given in bootstrap.properties.
* So, now config server decrypts it and use.
* We also have /decrypt end point to decrypt this data but, without basic authentication details, no one can decrypt/encrypt the data.

**The order of services to be started:**

Database -> spring cloud config server-> Eureka -> Zipkin -> Zuul -> remaining MSs.

**Flow of requests:**

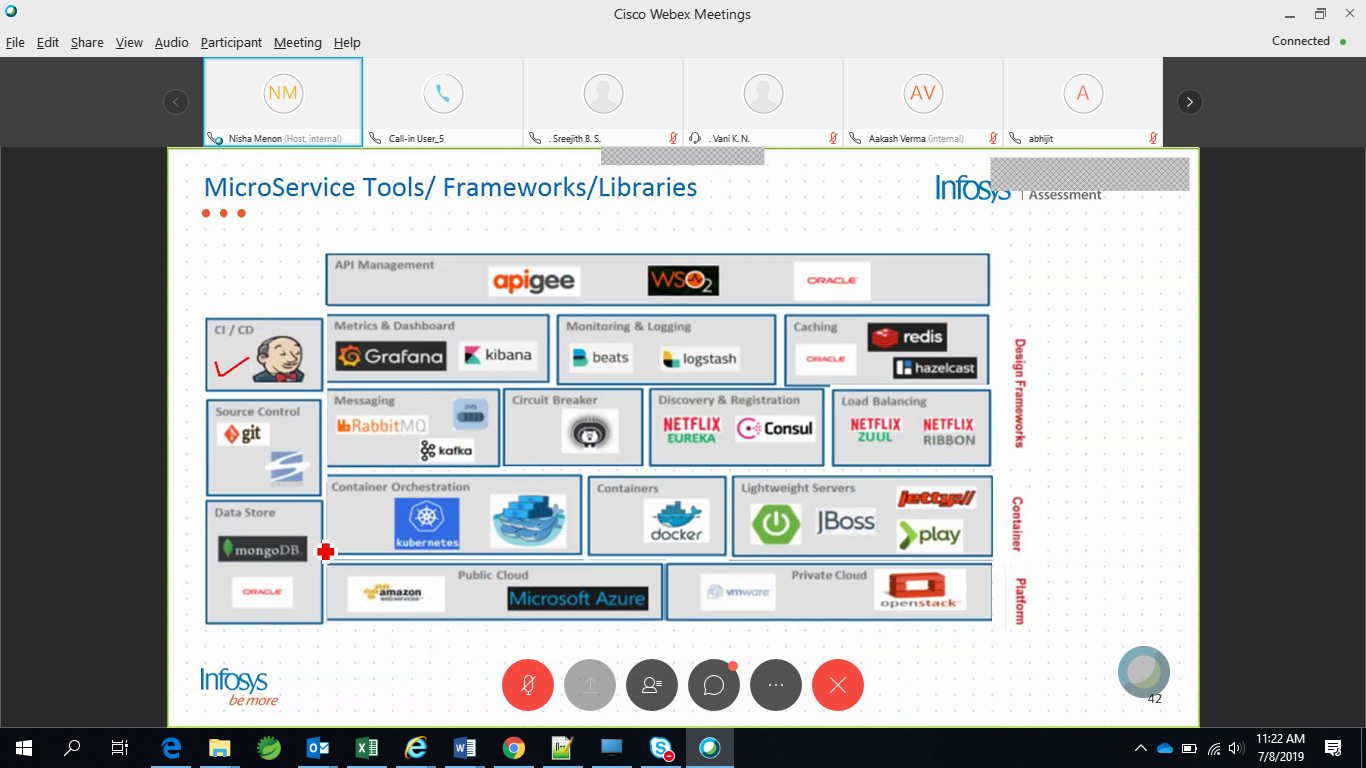
UI sends request to Zuul API -> Zuul calls Customer service as Zuul knows the Eureka details as Eureka started before Zuul -> Customer calls Friend and Plan and send the traces to Zipkin.

**Challenges with MSs:**

Creating, deploying and maintaining a microservice application is not easy. It has many technical and practical considerations. Some of them can be solved using patterns as below:

* As the number of microservices increases, managing them and scaling them becomes difficult -> **use LB like Ribbon**
* Failure points are more -> **user Circuit breaker pattern, Fallback pattern(Hystrix)**
* Since the attack area increases, security challenges also increase -> **use spring cloud security, spring oauth2**
* Since the number of microservices are high, keeping track of what is happening where becomes difficult -> **use Service Discovery(Eureka Dashboard), Circuit Breaker ( Hystrix, Turbine ), Distributed Tracing Pattern (Zipkin)**
* We need to monitor the different services and find out which are up, which are down, which need scaling, etc -> **use Service Discovery(Eureka Dashboard), Circuit Breaker ( Hystrix, Turbine ), Distributed Tracing Pattern (Zipkin)**
* Log data will be immense and we need to find out meaningful information from them -> **use Sleuth and ELK stack**
* A single faulty service can bring down all the other services -> **use Hystrix**
* Coordinating with external services is a challenge -> **API gateway for back end and front end separately**
* Managing databases becomes increasingly difficult -> **use** **Single DB, DB Per Service, API composition(**If you need data from both tables make a database call to fetch data owned by own service and make REST call to fetch data from the other. This pattern is called API Composition)**, Saga, CQRS**
* Since each service may use different technologystack, deploying them also becomes a challenge -> **Multiple service instances per host, Service instance per host , Service instance per Container**
* Testing microservices is a challenge as one microservice may depend on data from other microservices

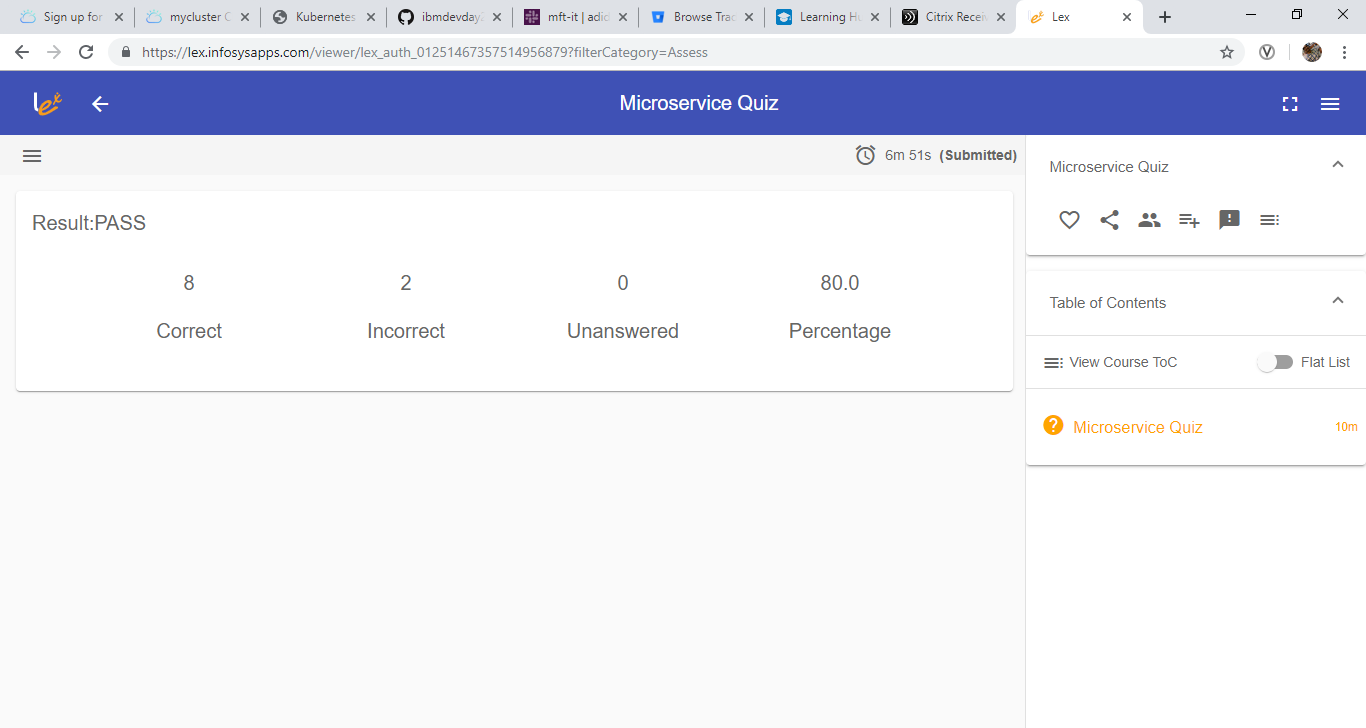
**Micro services need most of the below things:**

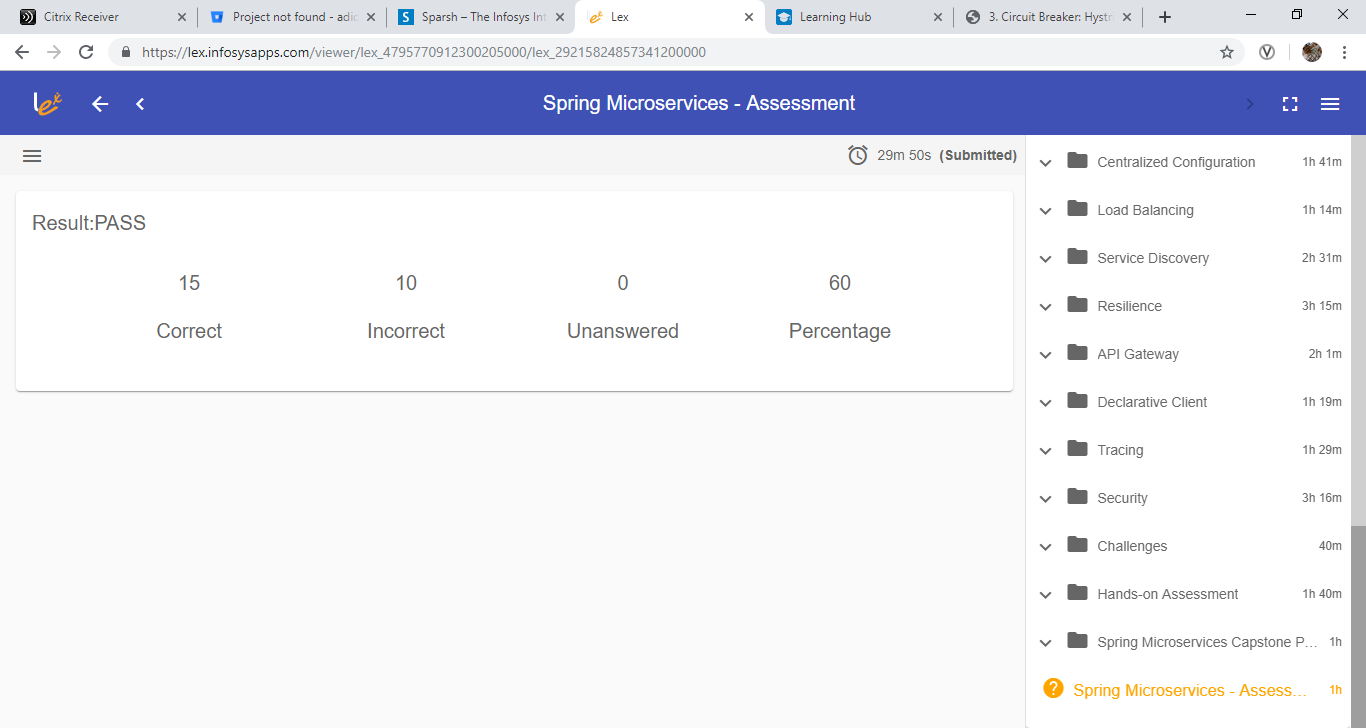


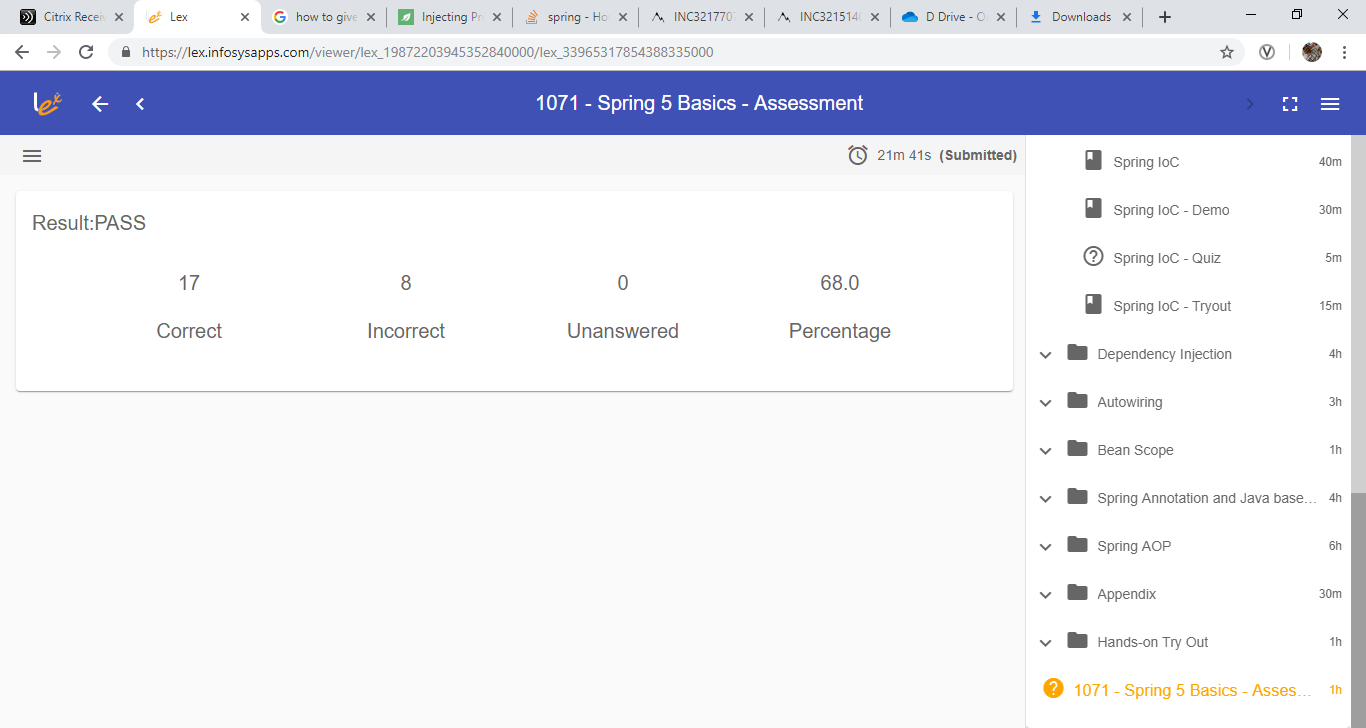
**Design patterns for MSs:**

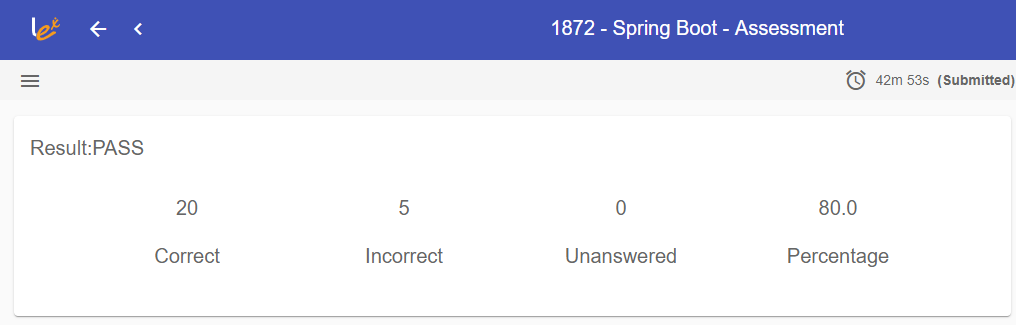
<https://dzone.com/articles/design-patterns-for-microservices>

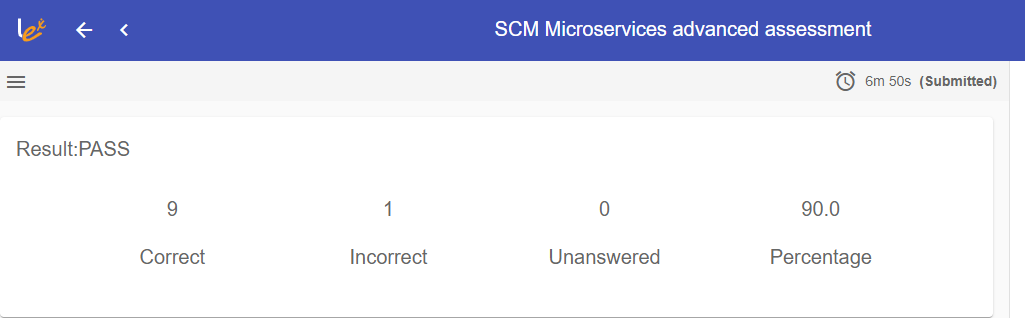
Assessment:

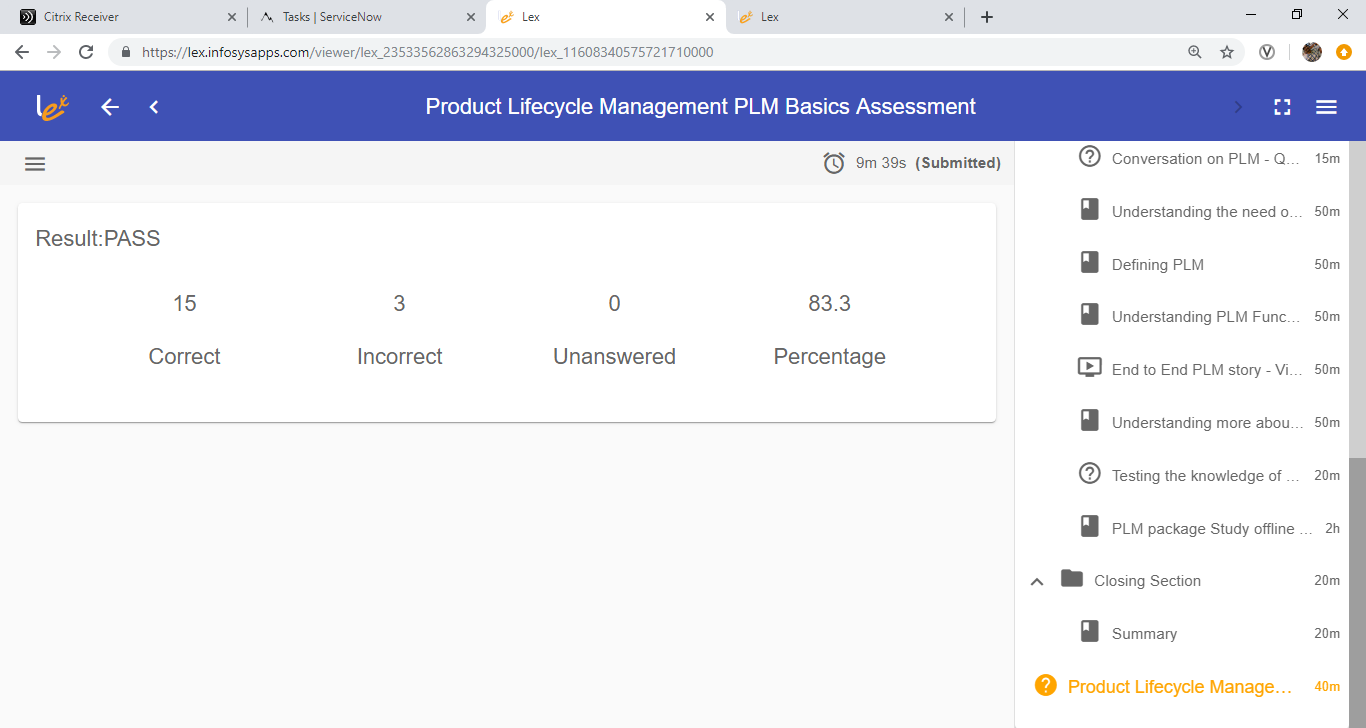












**SPRING 5 BASICS:**

* When u go for setter injection and when u go for constructor injection?
* Ans: when data is immutable and all the fields in the object are mandatory then we go for constructor based. Because if you miss one attribute it throw error but setter based will create object with default values.
* For setter injection, default constructor and setter methods are mandatory. For constructor injection, constructor with specified fields is mandatory.

**Autowiring:**

* byName, byType, constructor type.
* byName will inject the object by mapping the ref obj name in the class and id name in the config file. If name is not matched, bean remain unwired and assigned as null. Uses setter based injection.
* byType will inject the object by mapping the type of the obj in the class with the same type bean inside the config file. If more than one bean has been defined for the same type of class inside config file, it throws exception(NoUniqueBeanDependencyException). If type is not matched, bean remain unwired and assigned as null. Uses setter based injection.
* Using constructor type, it uses both byType(if only one bean is there with that type), byName(if more than one bean is available) and constructor. Uses constructor based injection.

**Annotation based configuration:**

<context-annotation:config/> -> add this in config file to achieve this.

**@Autowired** -> at property level, setter level and constructor level as well.

**@Qualifier(“point2”)**

Point p1;

* In case if the obj ref name in java class is not available in the config file with the same id name, we can give Qualifier annotation with an argument matching with the id name in config file.

**@Value(“15”)**

Private int radius;

* To provide the value for a field in case the value won’t change.

**Autoscanning:**

* Add <context:component-scan base-package=”com.infosys.demo”> in config file.
* Add **@Component** annotation to be added at the top of our class to which bean should be create.
* **@Component(“c**”) -> here c is the bean id name.
* **@Scope(“prototype”)**

**Java based configuration:**

* Create a new java class AppConfig.java
* Annotate it with **@Configuration**, saying it is a configuration class.
* Write below code in this class

**@Bean**

**@Scope(“prototype”)**

Public Point point(){

Point p = new Point();

p.setX(10);

p.setY(15);

return p;

}

* In the main class, replace ApplicationContext getting code with the below.

ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

* To enable auto scanning, add **@ComponentScan(“com.infy”)** annotation in the config class and remove all the methods written for Beans.

**AOP**

* Aspect, Advice, join, pointcut,
* The class in which all our common code will be written is **Aspect**
* The method of the Aspect provides the common functionality is called **Advice**
* The business methods of the program before or after which the advice can be called is known as **Joinpoint.**
* **Pointcut** represents an expression that evaluates to the business method name before or after which the advice needs to be called.

**Spring MVC**

* ModelAndView (@PathVariable(“userName”) String uName) -> generally used when we have some hierarchy. User/greet
* ModelAndView sayHello(@RequestParam(“userName”) String uName) -> used for all other situation. With a key, value pair. /?userName=anand&userPwd=vijay
* If you change the method to POST in jsp page and controller the values won’t be visible in the url.
  1. DispatcherServlet web.xml
  2. Dispatcher-servlet.xml
  3. @Controller
  4. @RequestMapping method
  5. ModelAndView
  6. View
  7. @PathVariable
  8. @RequestParam
* **Spring MVC tag library**

**<%@ taglib**

It is used to map the form data directly with java bean.

For this provide spring mvc tag lib in the html page. Change <input> tag to <form:input>. Change name to path attribute.

* In RegistrationController.java, add ModelAttribute

**Spring MVC**

**Important features of Spring MVC:**

* It is designed around a Servlet and it is known as DispatcherServlet, this is responsible for handling the request.
* It accommodates numerous view technologies like JSP, Velocity, Tiles and the generation of multiple file formats like PDF, Excel Files etc.
* It provides built in and custom validation support.
* It provides customizable locale, time zone and theme (look & feel) resolution.
* It provides good amount of tag libraries to design UI forms, for data binding and themes.
* HTTP Req -> Dispatcher Servlet -> Handler Mapping -> DS -> Controller -> Service -> Repository -> DB -> DS -> View Resolver -> DS -> View -> Http Output
* **<mvc:annotation-driven />** provides required support for annotation-driven MVC controllers like @RequestMapping, @Controller, @Valid etc. The same can be done with Java style configuration class using @Configuration, @EnableWebMvc annotations
* **mvc:annotation-driven** is used for enabling the Spring MVC components with its default configurations. If you dont include mvc:annotation-driven also your MVC application would work if you have used the context:component-scan for creating the beans or defined the beans in your XML file. But, mvc:annotation-driven does some extra job on configuring the special beans that would not have been configured if you are not using this element in your XML file.
* This tag would registers the HandlerMapping and HandlerAdapter required to dispatch requests to your @Controllers. In addition, it also applies some defaults based on what is present in your classpath.
* **context:annotation-config** is used for activating annotations in beans already registered in the application context (no matter whether they were defined with XML or by package scanning). That means it will resolve @Autowired and @Qualifier annotations for the beans which are already created and stored in the spring container.
* **context:component-scan** can also do the same job, but context:component-scan will also scan the packages for registering the beans to application context. context:annotation-config will not search for the beans registration, this will only activate the already registered beans in the context.

**Controller logic:**

* It should be annotated with @Controller and the methods inside it should be annotated with @RequestMapping. Methods should return ModelAndView object.
* There will be two types of variables passed to these methods, @RequestParam and @PathVariable.

**Spring Boot**

**Spring boot starter ->** contains all spring core dependencies and transitive dependencies as well.

Web MVC/Rest

@Controller

@RequestMapping

@ResponseBody

@RestController

@RequestMapping(value = “path”, method = “”) or @GetMapping(value=”path”)

* 1. Jar: MVC, Jackson to convert the response
  2. Web.xml -> DispatcherServlet
  3. Spring-servlet.xml
  4. Server

**Spring security:**

1. Add spring starter security dependencies. Spring will give you security.
2. Write spring config class.
3. Annotate @EnableWebSecurity for a class which should extend WebSecurityConfigurerAdapter class.
4. Override configureGlobal method.
5. BcryptEncoder

**Actuator:**

* By default most of the metrics given by actuator are secured, without authentication you can’t see, that is the reason we will disable the security for testing purpose.
* You can also configure which metrics to be shown and which shouldn’t.

**Restful:**

JAX-RS is a specification and Jersey, RESTEasy, Apache CXF are the implementations of it.

Spring REST is not an implementation of JAX-RS but follows all the REST principles.

Every end point having a unique URL.

Protocol: HTTP based – GET/POST/DELETE/PUT/PATCH

Layered style – output of one end point can be input to other.

Caching

**Angular**

Difference b/n var and let. Var has global scope and let has local scope because even var is defined inside inner loop which is not even true while running the variable will be defined but not initialized. The inner declarations move to function level. See below.

Function hello(var c) {

If(false) {

Var d = 10;

}

}

Converts to:

Function hello(var c) {

Var d;

If(false) {

d= 10;

}

}

**How to convert ts to js and run:**

tsc demo.ts

node demo.js

**Data types:**

Boolean, string, number are main. There is one more called any.

Console.log(`the empname is ${empname}

The number is ${empnum}`) -> no need of use \n and + , you can simply use enter and ${}

**Enum:**

enum MobilePrice{Black=25000,Gold,White};

enum MobilePrice{Black=25000,Gold=2000+Black,White};

Enum attributes will take only numbers. Also if you don’t give value, the value will be +1 for the previous value.

If first attribute doesn’t have value, it starts from 0.

**Arrays**

**Push and pop methods**

**Functions:**

function run(arg:string, arg2:number):string {

return arg+arg2;

}

* We can use option parameters as arguments by using a question mark after the name but it should be at the end.
* We can use default arguments as well by giving the value in the arguments linst.

**Arrow functions:**

Actual functions in js can’t access variables declared in class due to scope issues but arrow functions can access.

**Rest Parameter:**

**Getters and Setters to access private vars out side the class:**

tsc demo - - target es5

npm i -g @angular/cli

ng new <app name>

* Insomnia can be used alternate for postman tool and handles more content than postman.
* We can have a lot of end points using actuator. PFB the examples.
* actuator/health, /actuator/info, /actuator/beans, /actuator/metrics
* but to see beans details, we need to enable it explicitly by adding below configurations in application.yml file.
* management:  
   endpoints:  
   web:  
   exposure:  
   include: '\*'

**Udemy spring microservices:**

**Spring MVC with lombok:**

* create a project using spring initializer and add spring web, Lombok, actuator, devtools
* File -> settings -> Annotation Processors -> enable “Enable Annotation processing”
* File -> settings -> Plugins -> install GIT and Lambok plugins.
* Install Axis TCP plugin in Intellij

**Spring developer tools:**

* This is very useful during development as it reduces the time of server restart.
* It actually load only project related class files instead of external dependencies, so it applied the changes very fast.
* We can add developer tools dependency and instead of restarting the project, simply select the build option so that changes would reloaded without restart.
* Please do remember in case if we change external dependencies, it won’t work. We need to restart the server to reflect the changes.
* These developer tools related dependencies by default won’t pushed to production environment.
* **@Slf4j** annotation is a quick way to add logging using Lombok.

**API Versioning:**

* While updating the APIs, we can have 3 different kinds of updates. MAJOR, MINOR, PATCH.
* All breaking changes(which breaks the existing API like adding attributes to model) will go via MAJOR changes. For this, we will create a new package with name v2 inside all existing packages and do the necessary changes. Also @RequestMapping has to change to v2 from v1. Annotate old Controller with @Depricated annotation.
* @RestController is the combination of @Controller and @ResponseBody