JAVA ACCESSMENT WEEK 05

NAME : VINAY REDDY SABBELLA

PRACTICAL QUESTIONS:

Q3:

What does one mean by Service Registration and Discovery? How is it implemented in Spring Cloud?

ANSWER:

Microservice Registration and Discovery with Spring Cloud and Netflix's Eureka

The microservice style of architecture is not so much about building individual services so much as it is making the interactions between services reliable and failure-tolerant. While the focus on these interactions is new, the need for that focus is not. We’ve long known that services don’t operate in a vacuum. Even before cloud economics, we knew that - in a practical world - clients should be designed to be immune to service outages. The cloud makes it easy to think of capacity as ephemeral, fluid. The burden is on the client to manage this intrinsic complexity.

In this post, we’ll look at how Spring Cloud helps you manage that complexity with a service registry like Eureka and Consul and client-side load-balancing.

**The Cloud’s Phone Book**

A service registry is a phone book for your microservices. Each service registers itself with the service registry and tells the registry where it lives (host, port, node name) and perhaps other service-specific metadata - things that other services can use to make informed decisions about it. Clients can ask questions about the service topology (“are there any ‘fulfillment-services’ available, and if so, where?”) and service capabilities (“can you handle X, Y, and Z?”). You probably already use a technology that has some notion of a cluster (Cassandra, Memcached, etc.), and that information is ideally stored in a service registry.

There are several popular options for service registries. Netflix built and then open-sourced their own service registry, Eureka. Another new, but increasingly popular option is Consul. We’ll look principally at some of the integration between Spring Cloud and Netflix’s Eureka service registry.

From the the Spring Cloud project page: “Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state). Coordination of distributed systems leads to boiler plate patterns, and using Spring Cloud developers can quickly stand up services and applications that implement those patterns. They will work well in any distributed environment, including the developer’s own laptop, bare metal data centres, and managed platforms such as Cloud Foundry.”

Spring Cloud already supports both Eureka and Consul, though I’ll focus on Eureka in this post because it can be bootstrapped automatically in one of Spring Cloud’s auto-configurations. Eureka is implemented on the JVM but Consul is implemented in Go.

Installing Eureka

Standing up an instance of the Eureka service registry is easy if you have org.springframework.boot:spring-cloud-starter-eureka-server on your classpath.

package registry;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication

@EnableEurekaServer

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

COPY

My nominal src/main/resources/application.yml looks like this these days.

server:

port: ${PORT:8761}

eureka:

client:

registerWithEureka: false

fetchRegistry: false

server:

waitTimeInMsWhenSyncEmpty: 0

COPY

The service’s port is defaulted to the well-known 8761 if Cloud Foundry’s VCAP\_APPLICATION\_PORT environment variable isn’t available. The rest of the configuration simply tells this instance to not register itself with the Eureka instance it finds, because that instance is.. itself. If you run it locally, you can point a browser to http://localhost:8761 and monitor the registry from there.

Deploying Eureka

Spring Cloud will startup a Eureka instance with its Spring Boot auto-configuration. There are a couple of things to consider when deploying Eureka. First, you should always use a highly-available configuration in production. The Spring Cloud Eureka sample shows how to deploy it in a highly-available configuration.

Clients need to know

where to find the Eureka instance. If you have DNS then that might be one option, if you’re not polluting too large a global namespace. If you’re running in a Platform-as-a-Service and embracing 12-Factor app style applications then backing service credentials are configuration, and live external to the application, often exposed as environment variables. You can get the effect of having a Eureka service right now, though, by using Cloud Foundry’s cf CLI to create a user-provided service.

cf cups eureka-service -p '{"uri":"http://host-of-your-eureka-setup"}'

COPY

Point host-of-your-eureka-setup to a well-known host for your highly-available Eureka setup. I suspect we’ll soon see a way to create Eureka as a backing service in the same way you might a PostgreSQL or ElasticSearch instance on Pivotal Cloud Foundry.

Now that Eureka is up and running, let’s use it to connect some services to each other!

**Speak for Yourself**

Spring Cloud-based services have a spring.application.name property. It’s used to pull down configuration from the Configuration server, to identify the service to Eureka, and is referenceable in numerous other contexts when building Spring Cloud-based applications. This value typically lives in src/main/resources/bootstrap.(yml,properties), which is picked up earlier in the initialization than the normal src/main/resources/application.(yml,properties). A service with org.springframework.cloud:spring-cloud-starter-eureka on the classpath will be registered with the Eureka registry by its spring.application.name.

The src/main/resources/boostrap.yml file for each of my services looks like this, where my-service is the service name that changes from service to service:

spring:

application:

name: my-serviceCOPY

Spring Cloud uses the information in bootstrap.yml at service startup to discover the Eureka service registry and register the service and its spring.application.name, host, port, etc. You might wonder about that first bit. Spring Cloud attempts to look for it at a well-known address (http://127.0.0.1:), but you can change that. Here’s my src/main/resources/application.yml for a nominal Spring Cloud microservice, though there’s no reason this couldn’t live in the Spring Cloud configuration server. There may be many instances identifying themselves as my-service; Eureka will append the process’ information to a list of registrations for the same ID.

eureka:

client:

serviceUrl:

defaultZone: ${vcap.services.eureka-service.credentials.uri:http://127.0.0.1:8761}/eureka/

---

spring:

profiles: cloud

eureka:

instance:

hostname: ${APPLICATION\_DOMAIN}

nonSecurePort: 80

COPY

In this configuration, the Spring Cloud Eureka client knows to connect to the Eureka instance running on localhost if Cloud Foundry’s VCAP\_SERVICES environment variable doesn’t exist or contain valid credentials.

The bit of configuration under the --- delimiter is for when the application is run under the cloud Spring profile. It’s easy to set a profile using the SPRING\_PROFILES\_ACTIVE environment variable. You can configure Cloud Foundry environment variables in your manifest.yml or, on Cloud Foundry Lattice, your Docker file.

The cloud profile specific configuration specifically tells the Eureka client how to register the service in the discovered Eureka registry. I do this because my services don’t use fixed DNS. APPLICATION\_DOMAIN is an environment variable I set in my deploy scripts that tells a service what its externally referenceable URI is.

Click refresh on the Eureka web UI after 30 seconds (as of this writing) and you’ll see your web service(s) registered.

Client-Side Load Balancing with Ribbon

Spring Cloud references other services through their spring.application.name value. Knowing this value can be handy in a lot of contexts when building Spring Cloud-based services.

The goal, you’ll recall, is to let the client decide based on contextual information (which could change from client to client) which service instance it will connect to. Netflix has a Eureka-aware client-side load-balancing client called Ribbon that Spring Cloud integrates extensively. Ribbon is a client library with built-in software load balancers. Let’s look at an example that uses Eureka directly and then uses it through the Ribbon and Spring Cloud integration.

package passport;

import org.apache.commons.lang.builder.ToStringBuilder;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.boot.builder.SpringApplicationBuilder;

import org.springframework.cloud.client.ServiceInstance;

import org.springframework.cloud.client.discovery.DiscoveryClient;

import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

import org.springframework.cloud.netflix.feign.EnableFeignClients;

import org.springframework.cloud.netflix.feign.FeignClient;

import org.springframework.core.ParameterizedTypeReference;

import org.springframework.http.HttpMethod;

import org.springframework.http.ResponseEntity;

import org.springframework.stereotype.Component;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.client.RestTemplate;

import java.util.List;

@SpringBootApplication

@EnableEurekaClient

@EnableFeignClients

public class Application {

public static void main(String[] args) {

new SpringApplicationBuilder(Application.class)

.web(false)

.run(args);

}

}

@Component

class DiscoveryClientExample implements CommandLineRunner {

@Autowired

private DiscoveryClient discoveryClient;

@Override

public void run(String... strings) throws Exception {

discoveryClient.getInstances("photo-service").forEach((ServiceInstance s) -> {

System.out.println(ToStringBuilder.reflectionToString(s));

});

discoveryClient.getInstances("bookmark-service").forEach((ServiceInstance s) -> {

System.out.println(ToStringBuilder.reflectionToString(s));

});

}

}

@Component

class RestTemplateExample implements CommandLineRunner {

@Autowired

private RestTemplate restTemplate;

@Override

public void run(String... strings) throws Exception {

// use the "smart" Eureka-aware RestTemplate

ResponseEntity<List<Bookmark>> exchange =

this.restTemplate.exchange(

"http://bookmark-service/{userId}/bookmarks",

HttpMethod.GET,

null,

new ParameterizedTypeReference<List<Bookmark>>() {

},

(Object) "mstine");

exchange.getBody().forEach(System.out::println);

}

}

@Component

class FeignExample implements CommandLineRunner {

@Autowired

private BookmarkClient bookmarkClient;

@Override

public void run(String... strings) throws Exception {

this.bookmarkClient.getBookmarks("jlong").forEach(System.out::println);

}

}

@FeignClient("bookmark-service")

interface BookmarkClient {

@RequestMapping(method = RequestMethod.GET, value = "/{userId}/bookmarks")

List<Bookmark> getBookmarks(@PathVariable("userId") String userId);

}

class Bookmark {

private Long id;

private String href, label, description, userId;

@Override

public String toString() {

return "Bookmark{" +

"id=" + id +

", href='" + href + '\'' +

", label='" + label + '\'' +

", description='" + description + '\'' +

", userId='" + userId + '\'' +

'}';

}

public Bookmark() {

}

public Long getId() {

return id;

}

public String getHref() {

return href;

}

public String getLabel() {

return label;

}

public String getDescription() {

return description;

}

public String getUserId() {

return userId;

}

}

COPY

The DiscoveryClientExample bean demonstrates using the Spring Cloud common DiscoveryClient to interrogate the services. The results contain information like the hostname and the port for each service.

The RestTemplateExample bean demonstrates the auto-configured Ribbon-aware RestTemplate instance. Note that the URI uses a service ID, not an actual hostname. The service ID from the URI is extracted and given to Ribbon which then uses a load-balancer to pick from among the registered instances in Eureka and, finally, the HTTP call is made to a real service instance.

The FeignExample bean demonstrates using the Spring Cloud Feign integration. Feign is a handy project from Netflix that lets you describe a REST API client declaratively with annotations on an interface. In this case, we want to map the HTTP results from calls to the bookmark-service to the BookmarkClient Java interface. This mapping is configured in the Application class towards the top of the code page:

@Bean

BookmarkClient bookmarkClient() {

return loadBalance(BookmarkClient.class, "http://bookmark-service");

}

COPY

The URI is a service reference, not an actual hostname. It’s passed through the same processing as the URI given to the RestTemplate in the last example.

Pretty cool, eh? You can use the more basic DiscoveryClient API and make a call, or use the Ribbon and Eureka-aware RestTemplate or Feign-integrated client.

THEORY QUESTIONS:

Q1:

What does one mean by Load Balancing ? How is it implemented in Spring Cloud?

ANSWER:

Spring Cloud Load Balancer provides a simple round robin rule for load balancing between multiple instances of a single service. Our goal here is to implement a rule, which measures each application response time and gives a weight according to that time. The longer the response time, the less weight it will get.

Q2:

In which business scenario to use Netflix Hystrix?

ANSWER:

Netflix Hystrix is a design and implementation of circuit breaker pattern to isolate failure points and stop cascading them. It uses circuit breaker and fall back mechanisms to detect failures. This can be done with annotation @HystrixCommand where classes are annotated with @Component or @Sevice.  
When we use @HystrixCommand, Hystrix monitors failures till threshold point then subsequent calls are failed and redirected to fallback method.  
Example:

Class BookService{

@HystrixCommand(fallbackMethod="reliable")\

**public** String readingList() {

        URI uri = URI.create("http://localhost:8090/recommended");

**return** **this**.restTemplate.getForObject(uri, String.**class**);

     }

**public** String reliable() {

**return** "Cloud Native Java (O'Reilly)";

     }

}

In above example, Hystrix monitors readingList() method calls for failures. Once threshold is reached, Hystrix fails calls to readingList() and call reliable() method.

Q3: What is Spring Cloud Gateway? What are its advantages over Netflix Zuul?

ANSWER: Spring Cloud Gateway is API Gateway implementation by Spring Cloud team on top of Spring reactive ecosystem. It provides a simple and effective way to route incoming requests to the appropriate destination using Gateway Handler Mapping.And Spring Cloud Gateway uses Netty server to provide non-blocking asynchronous request processing.

**Spring Cloud Gateway vs. Zuul:**

I know you all who are using Spring boot must be wondering, we are already using Zuul. And it pretty much provides the same functionality as Spring Cloud Gateway. So, why do we switch to a new framework?

The main reason to me is — Zuul 1.x is not reactive. It is blocking. Now if you are moving to Reactive pattern to get better performance from your microservices and you are using Spring boot 2 with Reactor — you can use Zuul 2.0 which does have Reactive non blocking support with netty.

But Spring ecosystem does not have in-built support like Zuul 1.x. This means, you will have to run your own separate service just with Zuul. You can’t make it integrated with other existing spring microservice. And Spring team is not planning to add support for it with Spring reactive.

This means, if you are like me and moving to Spring reactive, I would advise to use Spring Cloud Gateway as it can integrate with any microservice and make it true api gateway with adding authentication and other security features at one place.

Q4: What is Spring Cloud Bus? Need for it?

ANSWER:

Spring Cloud Bus links nodes of a distributed system with a lightweight message broker. This can then be used to broadcast state changes (e.g. configuration changes) or other management instructions. AMQP and Kafka broker implementations are included with the project. Alternatively, any [Spring Cloud Stream](https://spring.io/projects/spring-cloud-stream) binder found on the classpath will work out of the box as a transport.

Q5: What is Spring Cloud Data Flow? Need for it?

ANSWER:

Microservice based Streaming and Batch data processing for Cloud Foundry and Kubernetes.

Spring Cloud Data Flow provides tools to create complex topologies for streaming and batch data pipelines. The data pipelines consist of [Spring Boot](https://projects.spring.io/spring-boot/) apps, built using the [Spring Cloud Stream](https://cloud.spring.io/spring-cloud-stream) or [Spring Cloud Task](https://cloud.spring.io/spring-cloud-task/) microservice frameworks.

Spring Cloud Data Flow supports a range of data processing use cases, from ETL to import/export, event streaming, and predictive analytics.

## **Features**

The Spring Cloud Data Flow server uses [Spring Cloud Deployer](https://github.com/spring-cloud/spring-cloud-deployer/), to deploy data pipelines made of Spring Cloud Stream or Spring Cloud Task applications onto modern platforms such as Cloud Foundry and Kubernetes.

Q6: What is Docker? How to deploy Spring Boot Microservices to Docker?

ANSWER:

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker’s methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.

**Deploying Spring Boot Microservice to Docker**

1. Create a Spring Boot Web Application.
2. Create image for starting this application.
3. Run the above image as container to start the jar.

Q7: How to deploy multiple microservices to docker?

ANSWER:

Microservices to Docker Container using docker networking. But previously we had to do the following steps manually-

* Create custom docker network named consumer-producer network
* Start Container named producer using image employee-producer and the custom network consumer-producer
* Start Container named consumer using image employee-consumer and the custom network consumer-producer

The above steps can be automated using docker compose.  
Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.  
Using docker-compose we will be creating the custom network named consumer-producer and then starting the containers employee-producer and employee consumer.

Q8: How to implement feign client using spring-boot?

ANSWER:

**Configure Feign Client in Spring Boot**

1. Overview.
2. Project Setup. Maven Project. Gradle Project. Feign Client Implementation Setup.
3. Enable Feign Client.
4. Create Feign Client.
5. Feign Client Configuration. From Property file. From Configuration Class file. Request Interceptor. ...
6. Consume the Feign Client.
7. Unit Test for Feign Client.

Q9: How to implement security for microservices?

ANSWER:

**Here are eight steps your teams can take to protect the integrity of your microservices architecture.**

1. Make your microservices architecture secure by design. ...
2. Scan for dependencies. ...
3. Use HTTPS everywhere. ...
4. Use access and identity tokens. ...
5. Encrypt and protect secrets. ...
6. Slow down attackers. ...
7. Know your cloud and cluster security.
8. Cover Your Security Bases.

Q10. How to implement distributed logging for microservices?

ANSWER:

**Logging in microservices architecture**

1. Use a correlation ID. A correlation ID is a unique identifier that developers use to segregate sets of operations and track individual requests. ...
2. Structure logs appropriately. ...
3. Provide informative application logs. ...
4. Visualize log data. ...
5. Use centralized log storage. ...
6. Query logs. ...
7. Handle failures.
8. Log Useful and Meaningful data to avoid Regret.