Desenvolvimento de Aplicações com Arquitetura Baseada em Microservices

Prof. Vinicius Cardoso Garcia
vcg@cin.ufpe.br :: @vinicius3w :: assertlab.com

[IF1007] - Tópicos Avançados em SI 4
https://github.com/vinicius3w/if1007-Microservices

AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 anos

Licença do material

Este Trabalho foi licenciado com uma Licença

Creative Commons - Atribuição-NãoComercial-CompartilhaIgual 3.0 Não Adaptada

Mais informações visite

http://creativecommons.org/licenses/by-nc-sa/3.0/deed.pt

AssertLab
Advanced Software and Systems Engineering Research Technologies

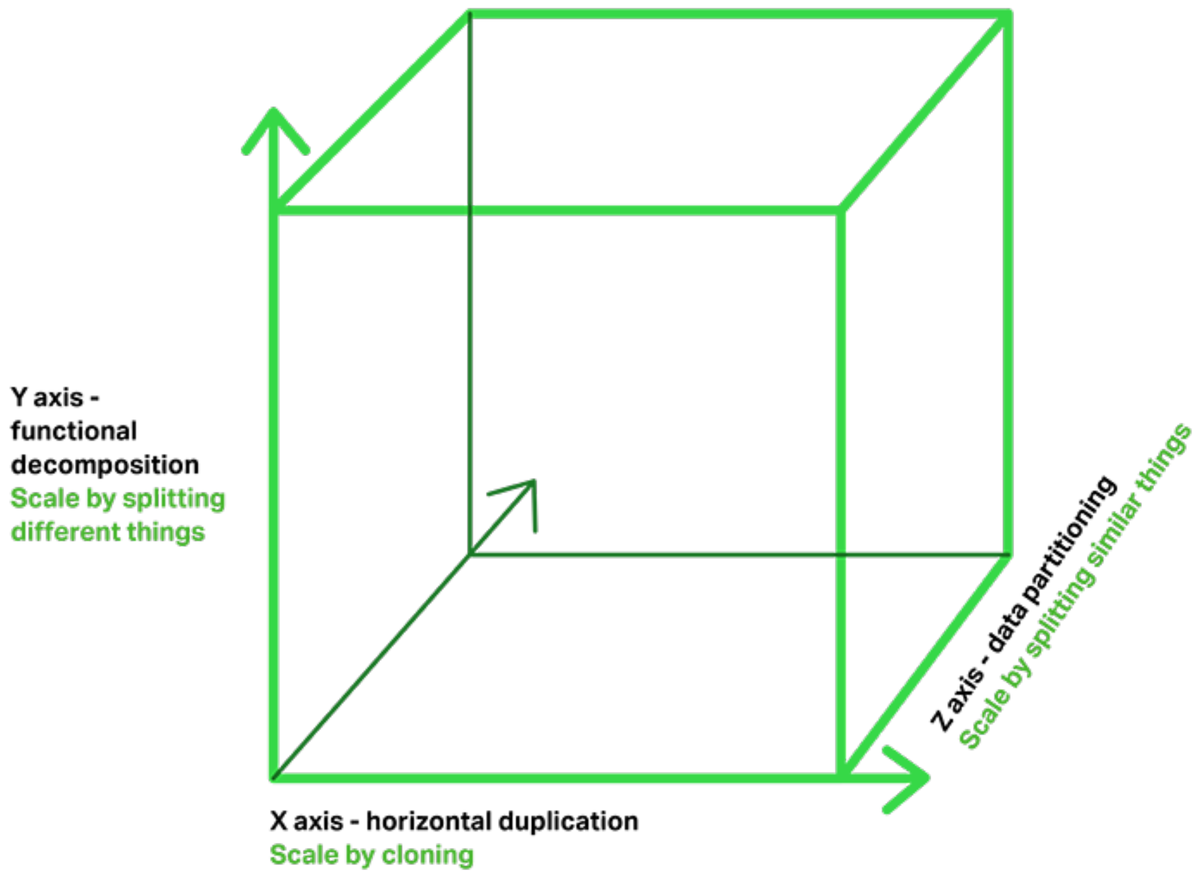
Centro de Informática

40 anos

Resources

- There is no textbook required. However, the following are some books that may be recommended:
  - Building Microservices: Designing Fine-Grained Systems
  - Spring Microservices
  - Spring Boot: Acelere o desenvolvimento de microsserviços
  - Microservices for Java Developers A Hands-on Introduction to Frameworks and Containers
  - Migrating to Cloud-Native Application Architectures
  - Continuous Integration
  - Getting started guides from spring.io

AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años



Scaling Microservices with Spring Cloud

Context

- Spring Cloud project has a suite of purpose-built components to achieve these additional capabilities effortlessly
- This lecture will provide a deep insight into the various components of the Spring Cloud project such as Eureka, Zuul, Ribbon, and Spring Config by positioning them against the microservices capability model
- How the Spring Cloud components help to scale the BrownField Airline's PSS microservices system

AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

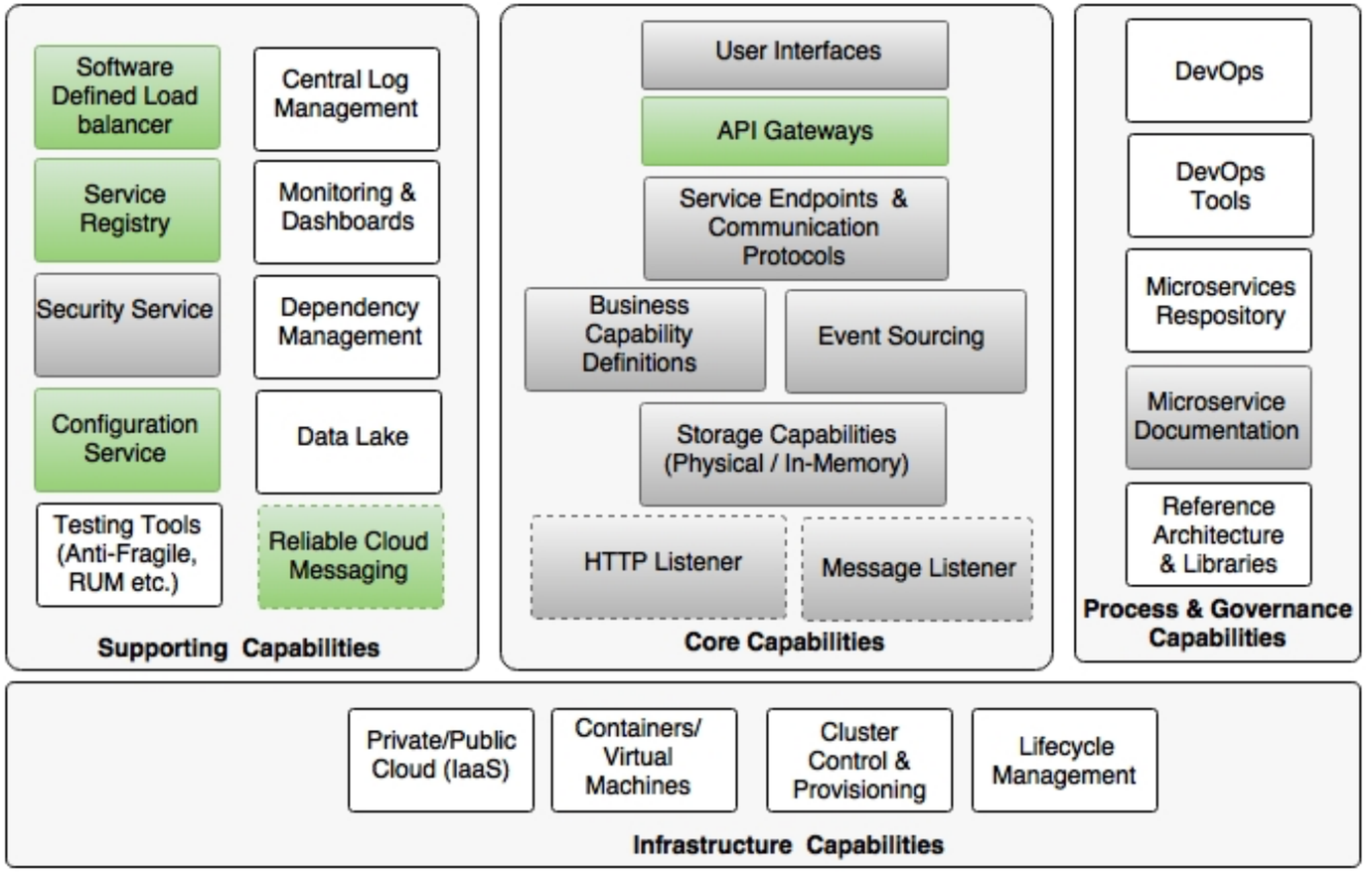
You will learn

- The Spring Config server for externalizing configuration
- The Eureka server for service registration and discovery
- The relevance of Zuul as a service proxy and gateway
- The implementation of automatic microservice registration and service discovery
- Spring Cloud messaging for asynchronous microservice composition

AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

Reviewing microservices capabilities

- The examples explore the following microservices capabilities from the microservices capability model
  - Software Defined Load Balancer
  - Service Registry
  - Configuration Service
  - Reliable Cloud Messaging
  - API Gateways



Reviewing BrownField's PSS implementation

- We have accomplished the following items in our microservice implementation so far
  - Each microservice exposes a set of REST/JSON endpoints for accessing business capabilities
  - Each microservice implements certain business functions using the Spring framework.
  - Each microservice stores its own persistent data using H2, an in-memory database
  - Microservices are built with Spring Boot, which has an embedded Tomcat server as the HTTP listener
  - RabbitMQ is used as an external messaging service. Search, Booking, and Check-in interact with each other through asynchronous messaging
  - Swagger is integrated with all microservices for documenting the REST APIs.
  - An OAuth2-based security mechanism is developed to protect the microservices

AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática 40 años

What is Spring Cloud?

- Is an umbrella project from the Spring team that implements a set of common patterns required by distributed systems, as a set of easy-to-use Java Spring libraries
- Are agnostic to the deployment environment
  - The cloud-ready solutions that are developed using Spring Cloud are also agnostic and portable across many cloud providers

AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

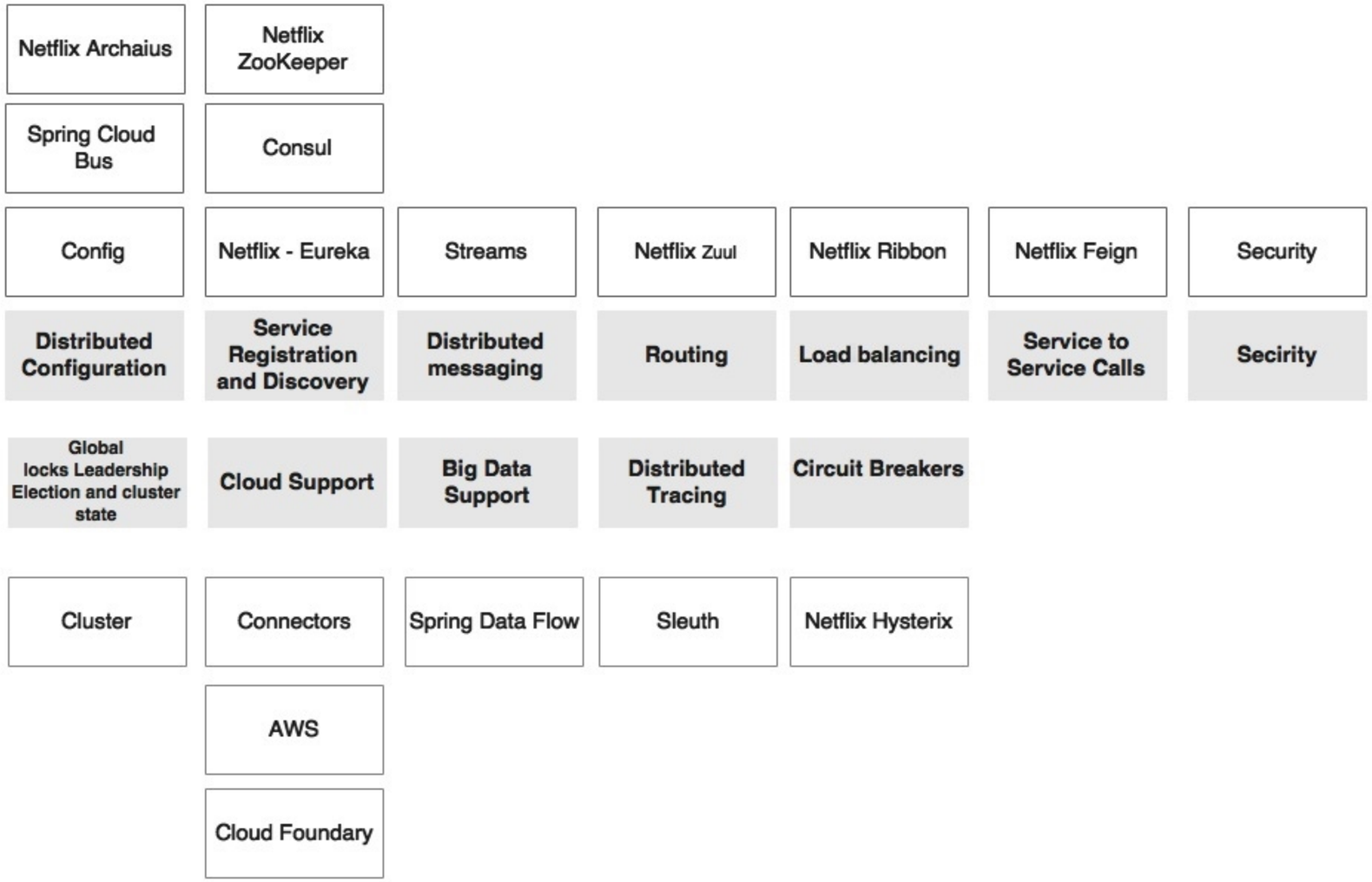
Spring Cloud releases

- The Spring Cloud project is an overarching Spring project that includes a combination of different components. The versions of these components are defined in the `spring-cloud-starter-parent` BOM
  - We are relying on the `Brixton.RELEASE` version of the Spring Cloud

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-dependencies</artifactId>
    <version>Brixton.RELEASE</version>
    <type>pom</type>
    <scope>import</scope>
 </dependency>
```


AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática 40 años

Components of Spring Cloud



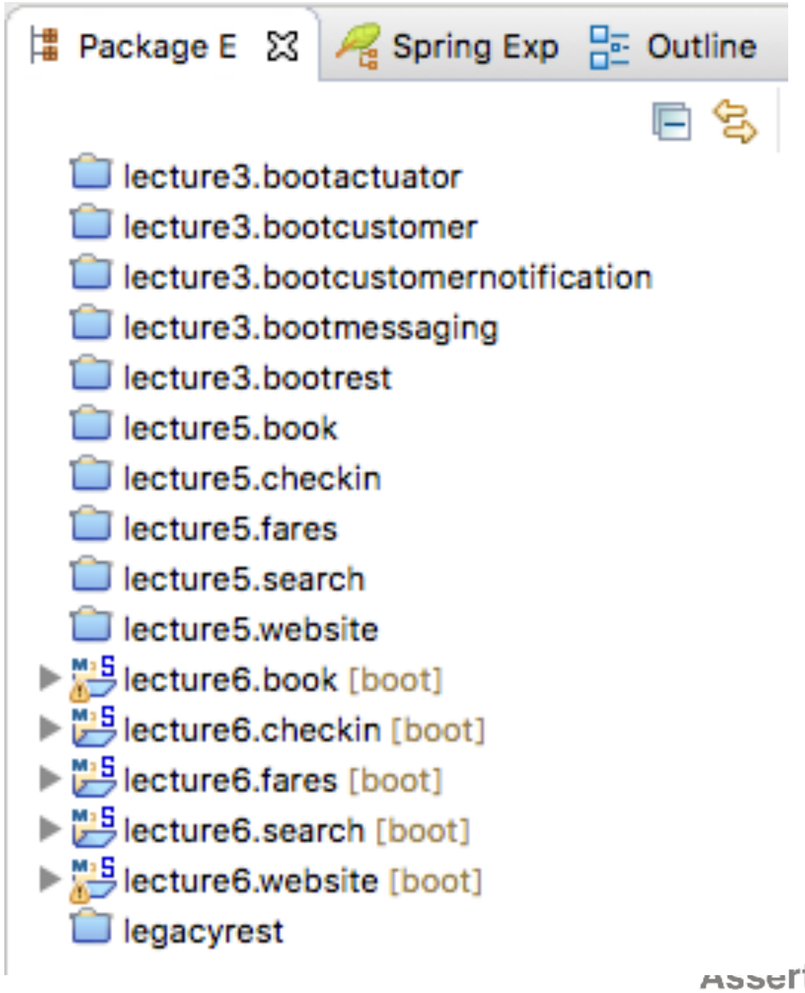
Spring Cloud and Netflix OSS

- Many of the Spring Cloud components which are critical for microservices' deployment came from the Netflix Open Source Software (Netflix OSS) center
- These components are extensively used in production systems, and are battle-tested with large scale microservice deployments at Netflix

AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

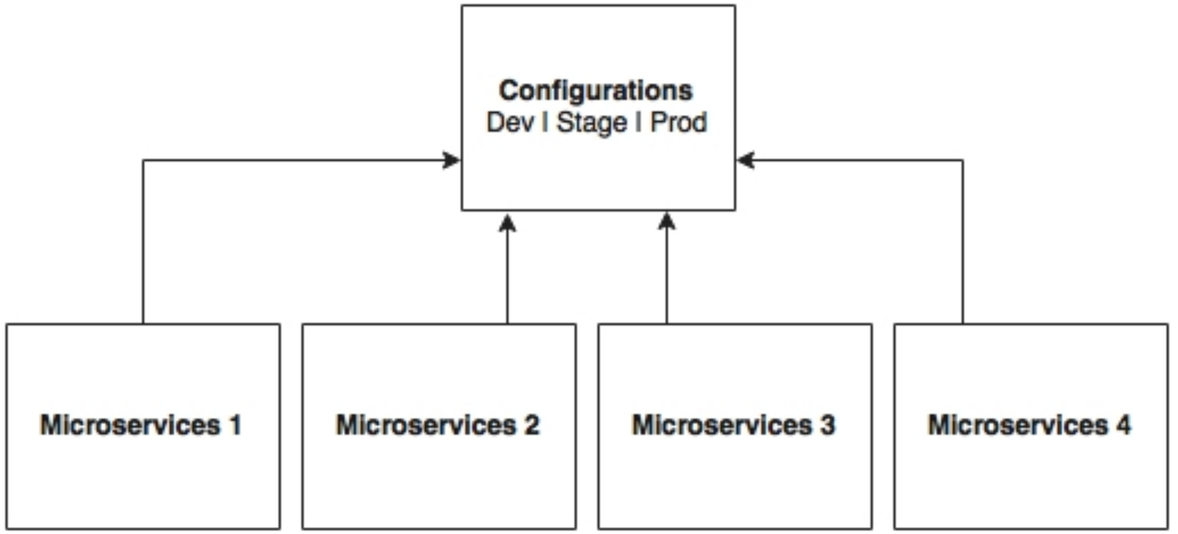
Setting up the environment for BrownField PSS

- We will amend the BrownField PSS microservices developed earlier
  - How to make these services enterprise grade using Spring Cloud components?
- In order to prepare the environment for this lecture, import and rename (`lecture5.*` to `lecture6.*`) projects into a new STS workspace



Spring Cloud Config

- The Spring Cloud Config server is an externalized configuration server in which applications and services can deposit, access, and manage all runtime configuration properties
  - `application.properties` or `application.yaml`
    - When microservices are moved from one environment to another?
- For large scale deployments, a simple yet powerful centralized configuration management solution is required

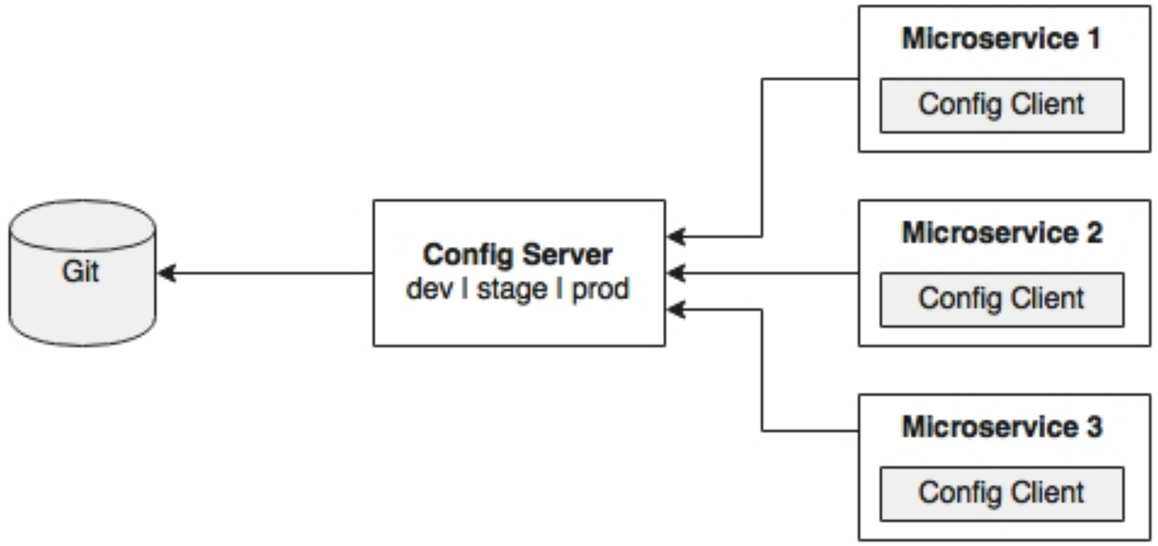


Spring Cloud Config

- There are multiple options available under the Spring Cloud project for building the configuration server

Cloud Config

- ☐ Config Client
  spring-cloud-config Client
- ☐ Config Server
  Central management for configuration via a git or svn backend
- ☐ Zookeeper Configuration
  Configuration management with Zookeeper and spring-cloud-zookeeper-config
- ☐ Consul Configuration
  Configuration management with Hashicorp Consul



- Spring Config server stores properties in a version-controlled repository such as Git or SVN

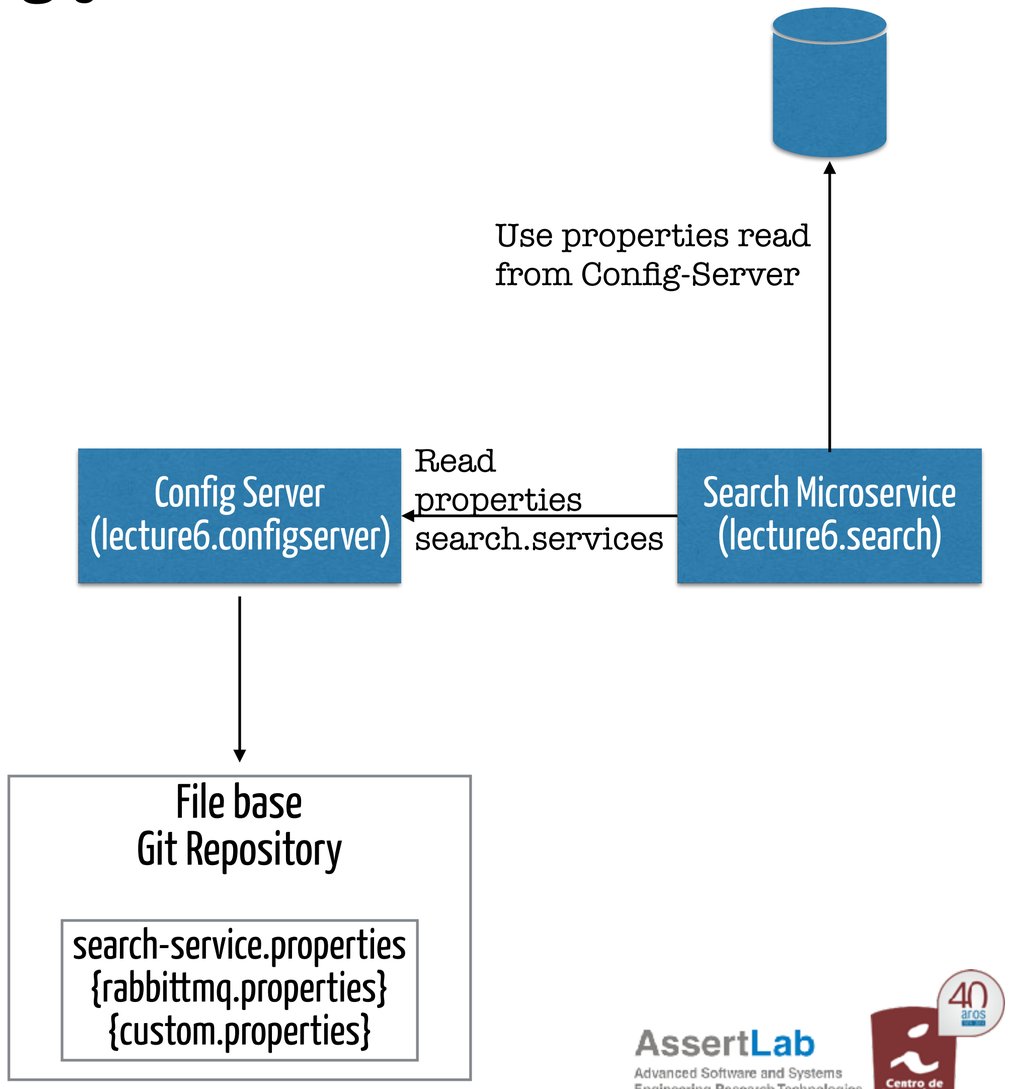
Spring Cloud Config

- Unlike Spring Boot, Spring Cloud uses a bootstrap context, which is a parent context of the main application
  - `bootstrap.yaml` or `bootstrap.properties` for loading initial configuration properties
- To make this work in a Spring Boot application, rename the `application.*` file to `bootstrap.*`

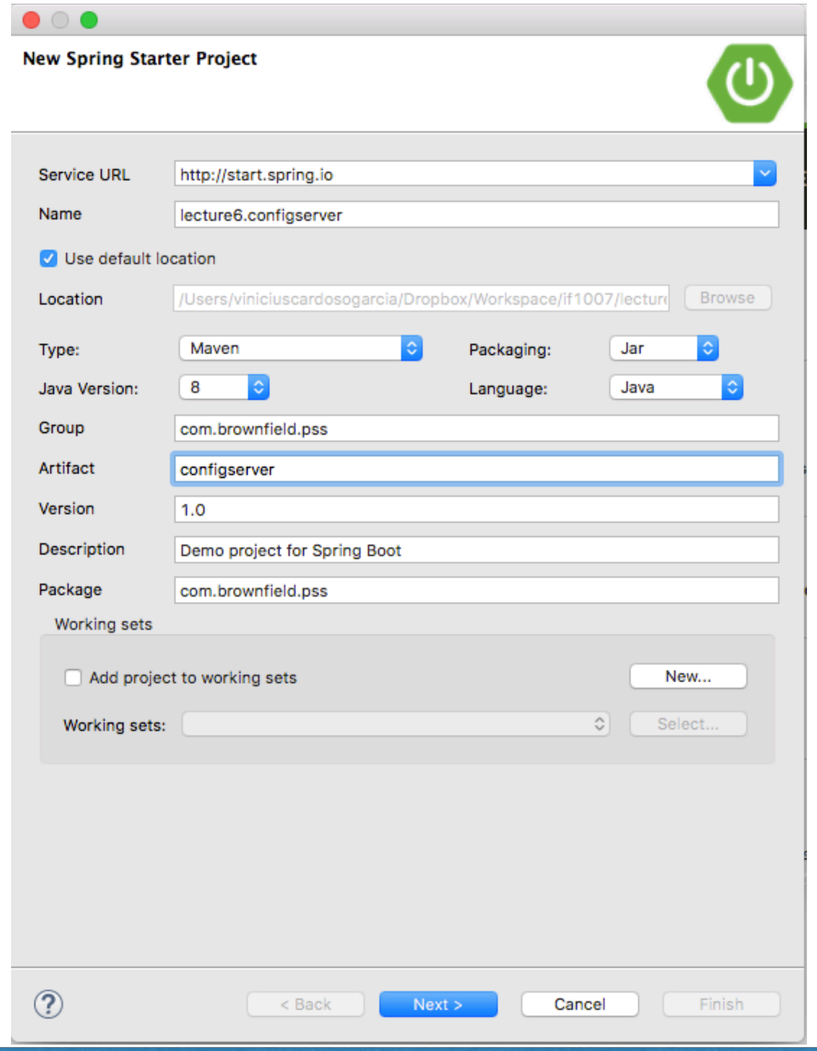
AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

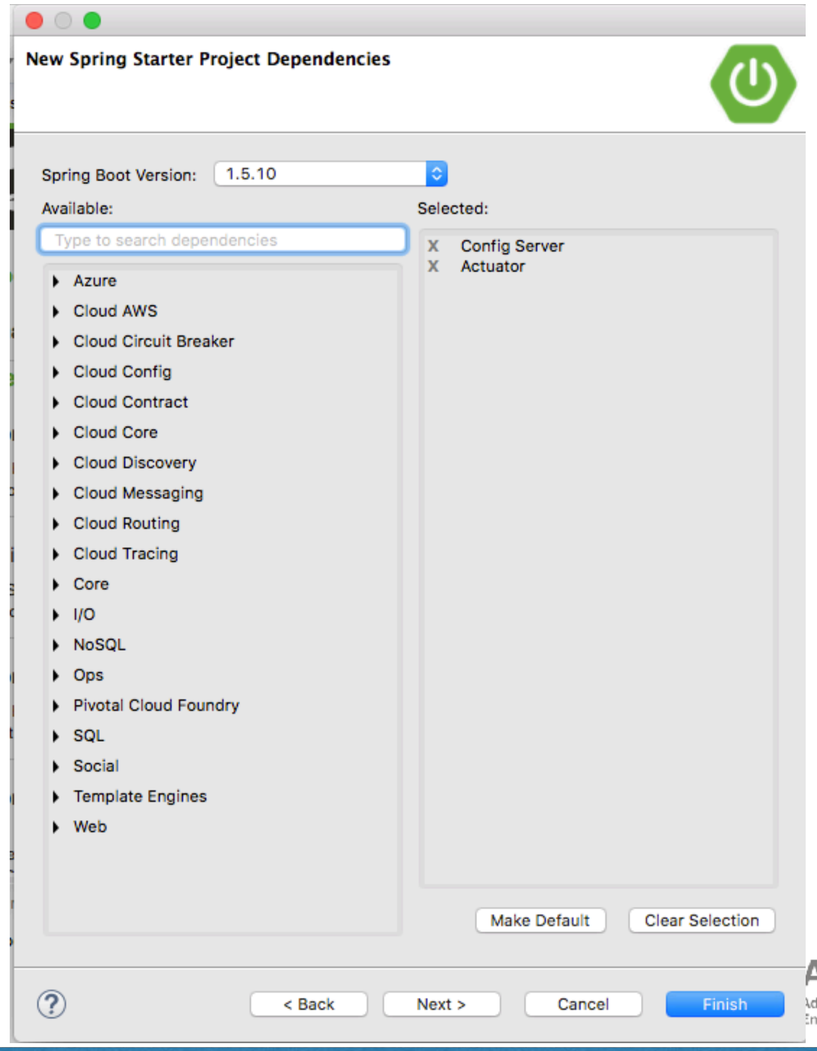
What's next?

- How to use the Config server in a real-world scenario?
- In order to do this, we will modify our search microservice (`lecture6.search`) to use the Config server



Setting up the Config server





AssertLab
Advanced Software and Systems Engineering Research Technologies
40 años
Centro de Informática

Setting up the Config server

- Set up a Git repository. This can be done by pointing to a remote Git configuration repository like the one at https://github.com/spring-cloud-samples/config-repo
- Alternately, a local filesystem-based Git repository can be used. In a real production scenario, an external Git is recommended

```
$ cd $HOME
$ mkdir config-repo
$ cd config-repo
$ git init .
$ echo message : helloworld > application.properties
$ git add -A .
$ git commit -m "Added sample application.properties”
```

- This code snippet creates a new Git repository on the local filesystem
- The next step is to change the configuration in the Config server to use the Git repository created in the previous step ~> rename the file `application.properties` to `bootstrap.properties`

AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

Setting up the Config server

- Edit the contents of the new bootstrap.properties file to match the following

```
server.port=8888
spring.cloud.config.server.git.uri:
file://${user.home}/config-repo
```

- Optionally, rename the default package of the auto-generated `Application.java` from `com.example` to `com.brownfield.configserver`. Add `@EnableConfigServer` in `Application.java`

AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

Setting up the Config server

- Run the Config server by right-clicking on the project, and running it as a Spring Boot app.
- Visit `http://localhost:8888/env` to see whether the server is running. If everything is fine, this will list all environment configurations. Note that `/env` is an actuator endpoint
- Check `http://localhost:8888/application/default/master` to see the properties specific to `application.properties`, which were added in the earlier step

```
{“name":"application","profiles":
["default"],"label":"master","version":"6046fd2ff4fa09d38437676
60d963866ffcc7d28","propertySources":[{"name":"file:///Users/
rvlabs /config-repo /application.properties","source":
{"message":"helloworld"}}]}
```


AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

Understanding the Config server URL

- `http://localhost:8888/application/default/master`
  - The first element in the URL is the application name, a logical name given to the application, using the `spring.application.name` property in `bootstrap.properties` of the Spring Boot application
  - The second part of the URL represents the profile. The two common scenarios are segregating different environments or segregating server configurations
  - The last part of the URL is the label, and is named master by default. The label is an optional Git label that can be used, if required.
  - `http://localhost:8888/{name}/{profile}/{label}`

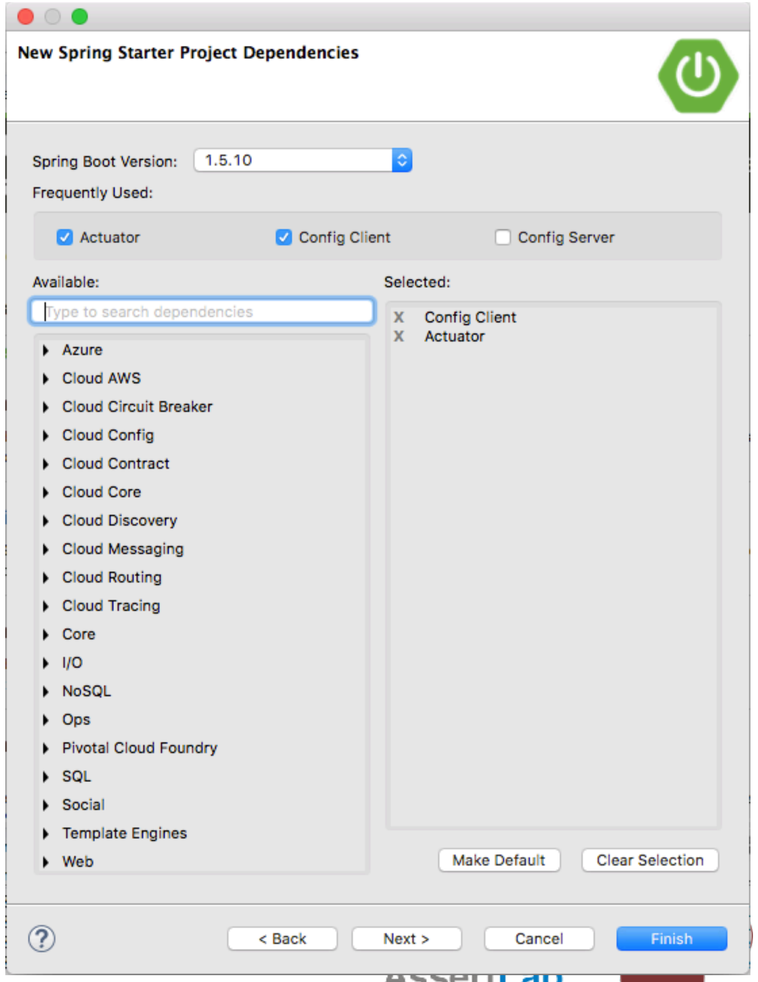
AssertLab
Advanced Software and Systems Engineering Research Technologies
40 años
Centro de Informática

Accessing the Config Server from clients

- Add the Spring Cloud Config dependency and the actuator (if the actuator is not already in place) to the pom.xml file

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

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```



Accessing the Config Server from clients

- Rename `application.properties` to `bootstrap.properties`, and add an application name and a configuration server URL

```
spring.application.name=search-service
spring.cloud.config.uri=http://localhost:8888


server.port=8090


spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
```


AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

Accessing the Config Server from clients

- Create a new configuration file for `search-service`. Create a new `search-service.properties` under the `config-repo` folder where the Git repository is created
- Move service-specific properties from `bootstrap.properties` to the new `search-service.properties` file
- In order to demonstrate the centralized configuration of properties and propagation of changes, add a new application-specific property `originairports.shutdown` to temporarily take out an airport from the search, to the property file

search-service.properties

```
1  spring.application.name=search-service
2  spring.rabbitmq.host=localhost
3  spring.rabbitmq.port=5672
4  spring.rabbitmq.username=guest
5  spring.rabbitmq.password=guest
6  orginairports.shutdown:SEA
```

Commit this new file into the Git repository by executing the following commands:

```
git add -A .

git commit -m "adding new configuration"
```


AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

Accessing the Config Server from clients

- Modify the Search microservice code to use the configured parameter, `originairports.shutdown`
- A `RefreshScope` annotation has to be added at the class level to allow properties to be refreshed when there is a change
- Add the following instance variable as a place holder for the new property that is just added in the Config server
- Change the application code to use this property

```
@RefreshScope
@CrossOrigin
@RestController
@RequestMapping("/search")
class SearchRestController {

    private static final Logger logger = LoggerFactory.getLogger(SearchComponent.class);

    private SearchComponent searchComponent;

    @Value("${originairports.shutdown}")
    private String originAirportShutdownList;

    @Autowired
    public SearchRestController(SearchComponent searchComponent){
        this.searchComponent = searchComponent;
    }

    @RequestMapping(value="/get", method = RequestMethod.POST)
    List<Flight> search(@RequestBody SearchQuery query){
        logger.info("Input : "+ query);
        if(Arrays.asList(originAirportShutdownList.split(",")).contains(query.getOrigin())){
            logger.info("The origin airport is in shutdown state");
            return new ArrayList<Flight>();
        }
        //System.out.println("Input : "+ query);
        return searchComponent.search(query);
    }

}
```

Accessing the Config Server from clients

- Start the Config server. Then start the Search microservice. Make sure that the RabbitMQ server is running ~> `rabbitmq-server`
- Modify the `lecture6.website` project to match the `bootstrap.properties` content as follows to utilize the Config server:

```
spring.application.name=test-client
server.port=8001
spring.cloud.config.uri=http://localhost:8888
```

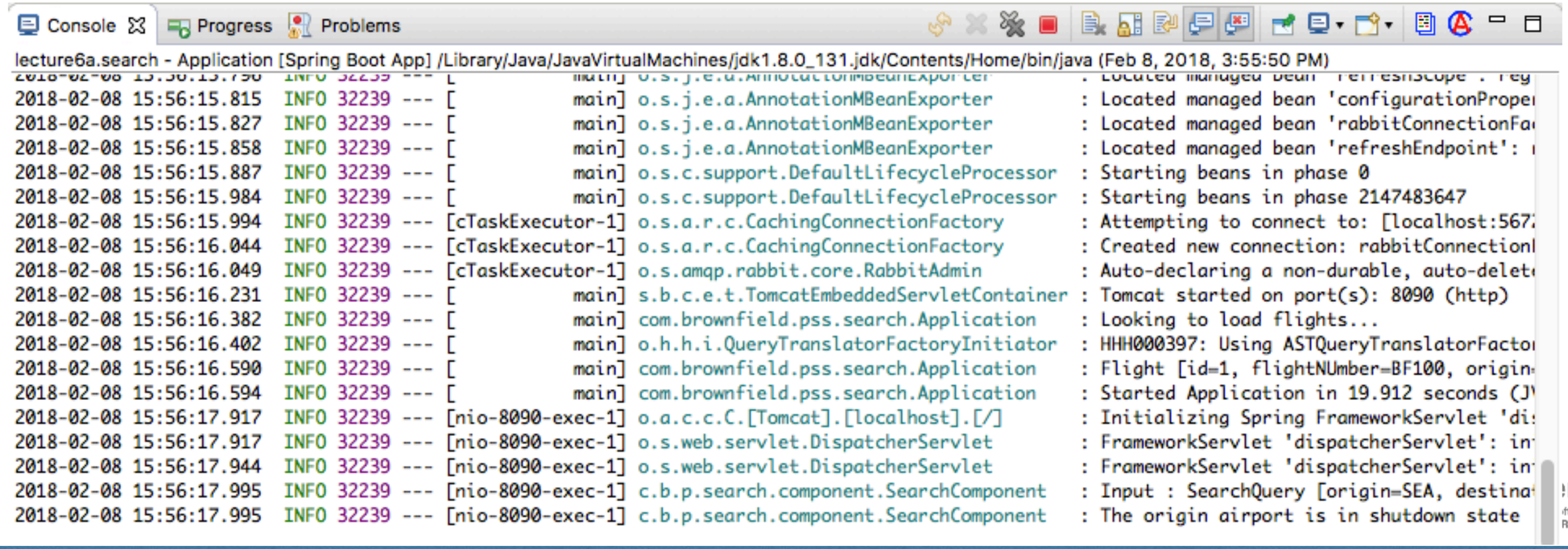
- Change the run method of `CommandLineRunner` in `Application.java` to query SEA as the origin airport:

```
SearchQuery = new SearchQuery("SEA","SFO","22-JAN-16");
```


AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

Accessing the Config Server from clients

- Run the `lecture6.website` project. The `CommandLineRunner` will now return an empty flight list. The following message will be printed in the server:

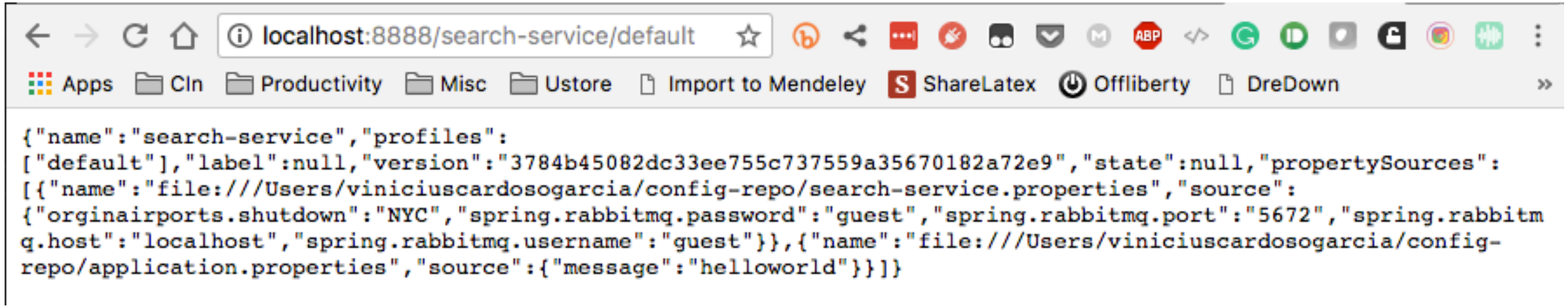


Handling configuration changes

- How to propagate configuration properties when there is a change?
- Change the property in the search-service.properties file to the following: `originairports.shutdown:NYC`
- Commit the change in the Git repository. Refresh the Config server URL (http://localhost:8888/search-service/default) and see whether the property change is reflected

AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

Handling configuration changes



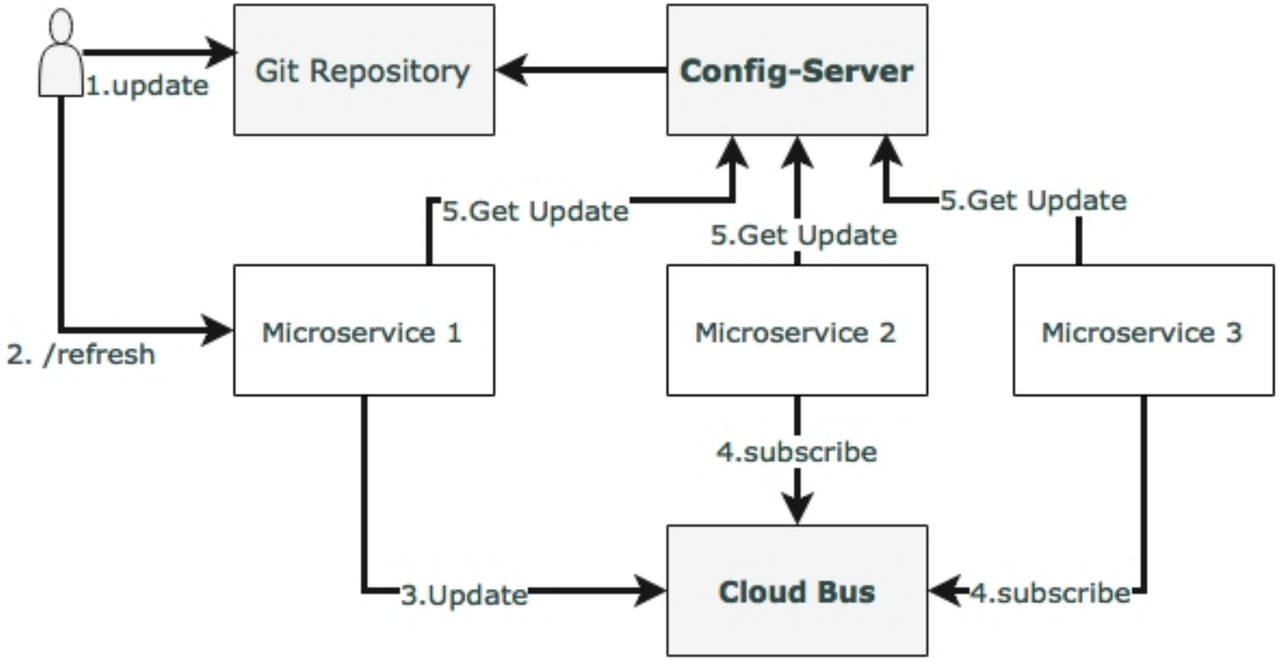
- Rerun the website project again, and observe the CommandLineRunner execution
- The service returns an empty flight list as earlier, and still complains as follows: `The origin airport is in shutdown state`
- In order to force reloading of the configuration properties, call the `/refresh` endpoint of the Search microservice

```
curl -d {} localhost:8090/refresh
```

- Rerun the website project, this should return the list of flights that we have requested from SEA.

Spring Cloud Bus for propagating configuration changes

- Spring Cloud Bus provides a mechanism to refresh configurations across multiple instances without knowing how many instances there are, or their locations
- This is done by connecting all service instances through a single message broker



Using RabbitMQ as the AMQP message broker

- Add a new dependency in the `leacture6.search` project's `pom.xml` file to introduce the Cloud Bus dependency:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

- The Search microservice also needs connectivity to the RabbitMQ, but this is already provided in `search-service.properties`

Using RabbitMQ as the AMQP message broker

- Rebuild and restart the Search microservice. In this case, we will run two instances of the Search microservice from a command line

```
java -jar -Dserver.port=8090   search-1.0.jar
java -jar -Dserver.port=8091   search-1.0.jar
```

- Rerun the website project

```
NUmber=BF101, origin=NYC, destination=SFO, flightDate=22-JAN-16, fares=Fares [id=2, fare=101, currency=USD], inventory=I
nventory [id=2, count=100]]
2018-03-15 11:21:23.655  INFO 8447 --- [           main] com.brownfield.pss.search.Application    : Flight [id=3, flight
NUmber=BF105, origin=NYC, destination=SFO, flightDate=22-JAN-16, fares=Fares [id=3, fare=105, currency=USD], inventory=I
nventory [id=3, count=100]]
2018-03-15 11:21:23.655  INFO 8447 --- [           main] com.brownfield.pss.search.Application    : Flight [id=4, flight
NUmber=BF106, origin=NYC, destination=SFO, flightDate=22-JAN-16, fares=Fares [id=4, fare=106, currency=USD], inventory=I
nventory [id=4, count=100]]
2018-03-15 11:21:23.660  INFO 8447 --- [           main] com.brownfield.pss.search.Application    : Started Application
in 30.778 seconds (JVM running for 32.123)
2018-03-15 11:22:32.145  INFO 8447 --- [nio-8090-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring
FrameworkServlet 'dispatcherServlet'
2018-03-15 11:22:32.147  INFO 8447 --- [nio-8090-exec-1] o.s.web.servlet.DispatcherServlet        : FrameworkServlet 'di
spatcherServlet': initialization started
2018-03-15 11:22:32.336  INFO 8447 --- [nio-8090-exec-1] o.s.web.servlet.DispatcherServlet        : FrameworkServlet 'di
spatcherServlet': initialization completed in 188 ms
Input : SearchQuery [origin=SEA, destination=SFO, flightDate=22-JAN-16]
```


AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

Using RabbitMQ as the AMQP message broker

- Now, update `search-service.properties` with the following value, and commit to Git:
  `originairports.shutdown:SEA`
- Run `curl -d {} localhost:8090/bus/refresh`
- Immediately, we will see the following message for both instances:

```
Received remote refresh request. Keys
refreshed [originairports.shutdown]
```


AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

Using RabbitMQ as the AMQP message broker

- The bus endpoint sends a message to the message broker internally, which is eventually consumed by all instances, reloading their property files
- Changes can also be applied to a specific application by specifying the application name like so:
- `originairports.shutdown:SEA`

AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

Using RabbitMQ as the AMQP message broker

- The bus endpoint sends a message to the message broker internally, which is eventually consumed by all instances, reloading their property files
- Changes can also be applied to a specific application by specifying the application name like so:

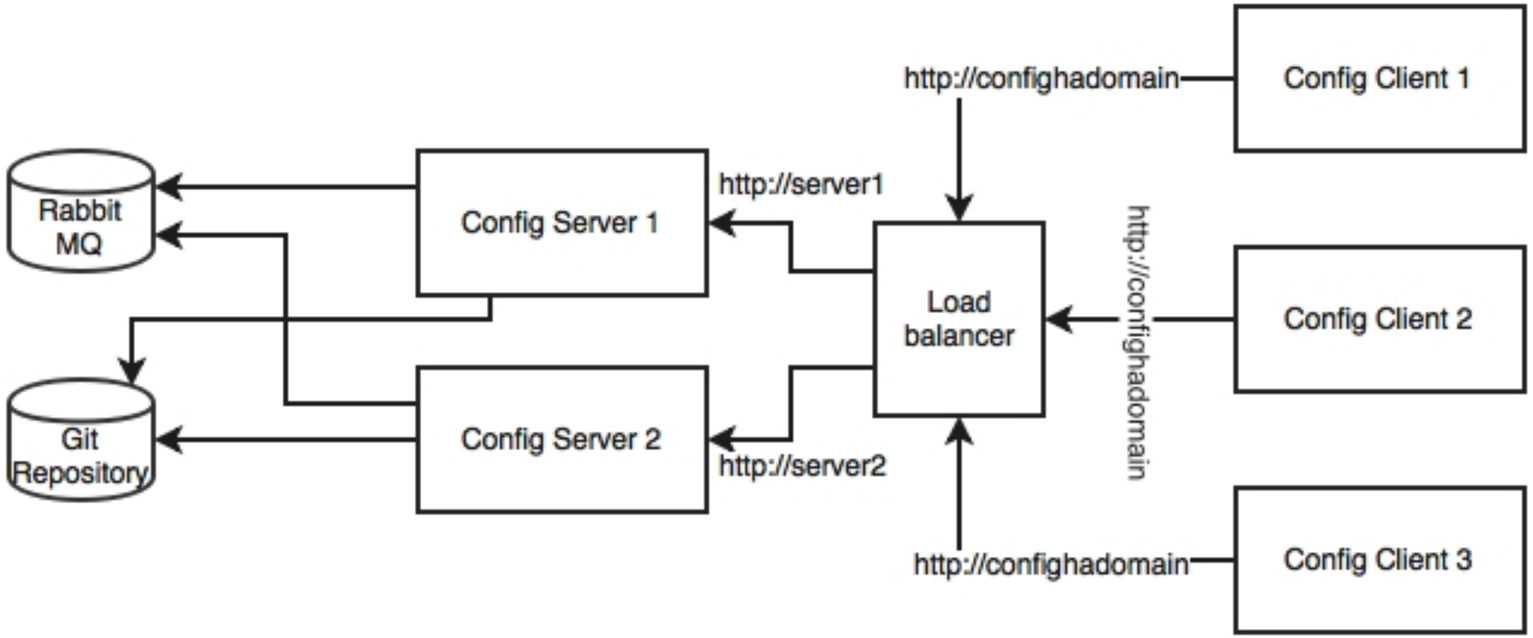
```
/bus/refresh?destination=search-service:**
```

- We can also refresh specific properties by setting the property name as a parameter

AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

Setting up high availability for the Config server

- The Config server is a single point of failure in this architecture
- The second one is the Git repository,
- and the third one is the RabbitMQ server



Setting up high availability for the Config server

- Since the Config server is a stateless HTTP service, multiple instances of configuration servers can be run in parallel
- Load balancer or DNS server URL will be configured in the microservices' `bootstrap.properties` file
- Using an external highly available Git service or a highly available internal Git service
  - The GitLab example for setting up high availability is available at https://about.gitlab.com/high-availability/

AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

Setting up high availability for the Config server

- RabbitMQ also has to be configured for high availability
  - The high availability for RabbitMQ is needed only to push configuration changes dynamically to all instances
- RabbitMQ high availability can be achieved by either using a cloud service or a locally configured highly available RabbitMQ service
  - Setting up high availability for Rabbit MQ is documented at https://www.rabbitmq.com/ha.htm

AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

Monitoring the Config server health

- The Config server is nothing but a Spring Boot application, and is, by default, configured with an actuator
- Hence, all actuator endpoints are applicable for the Config server
- The health of the server can be monitored using the following actuator URL: http://localhost:8888/health

Completing changes to use the Config server

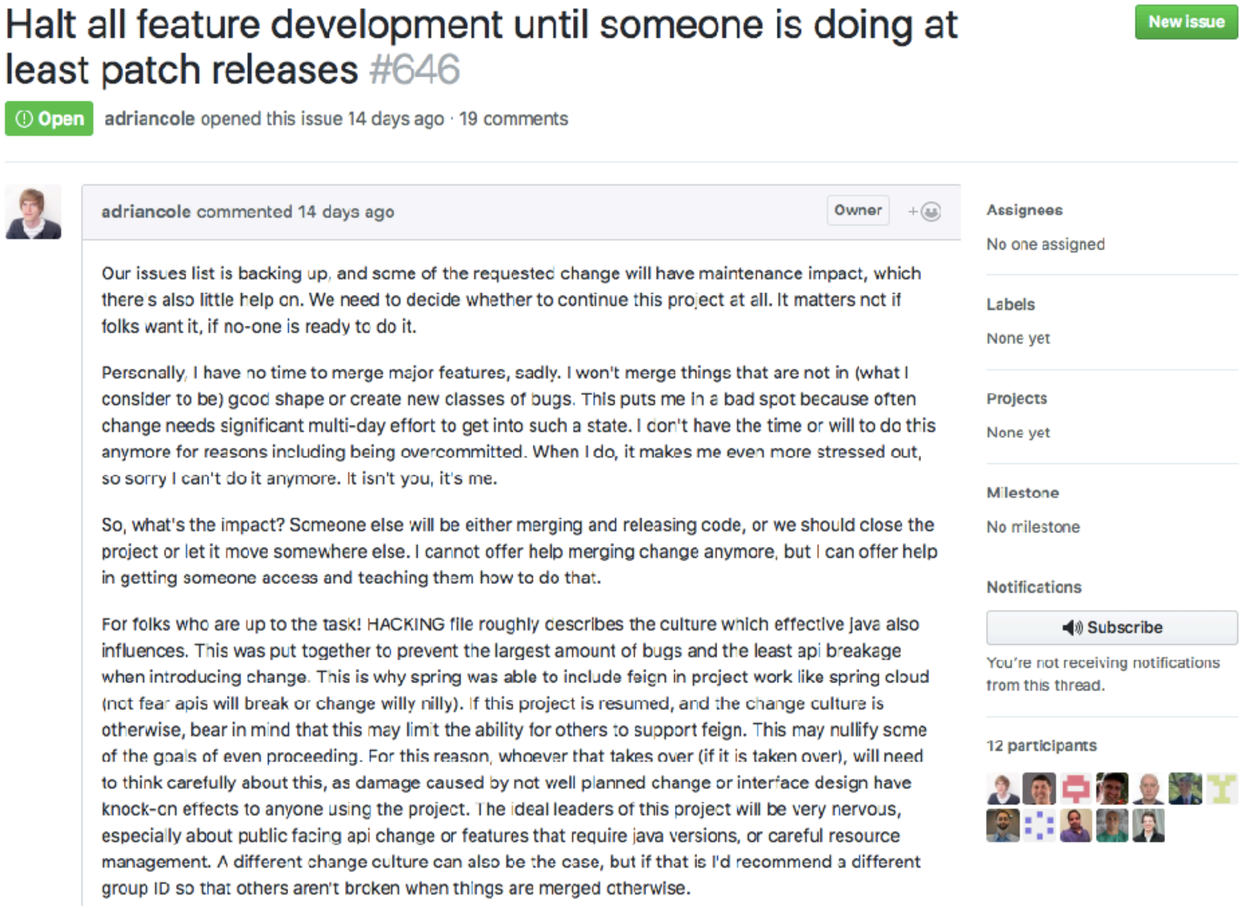
- All microservices in the examples given in `lecture6.*` need to make similar changes to look to the Config server for getting the configuration parameters
- The Fare service URL in the booking component will also be externalized:

```
private static final String FareURL = "/fares";
@Value("${fares-service.url}")
private String fareServiceUrl;
Fare = restTemplate.getForObject(fareServiceUrl+FareURL +"/
get?flightNumber="+record.getFlightNumber()
+"&flightDate="+record.getFlightDate(),Fare.class);
```

- As shown in the preceding code snippet, the Fare service URL is fetched through a new property: `fares-service.url`

Warm up

- Open Feign is looking for maintainers...
  - https://github.com/OpenFeign/feign/issues/646



Halt all feature development until someone is doing at least patch releases #646

New issue

Open adriancole opened this issue 14 days ago · 19 comments

adriancole commented 14 days ago — Owner

Our issues list is backing up, and some of the requested change will have maintenance impact, which there's also little help on. We need to decide whether to continue this project at all. It matters not if folks want it, if no-one is ready to do it.

Personally, I have no time to merge major features, sadly. I won't merge things that are not in (what I consider to be) good shape or create new classes of bugs. This puts me in a bad spot because often change needs significant multi-day effort to get into such a state. I don't have the time or will to do this anymore for reasons including being overcommitted. When I do, it makes me even more stressed out, so sorry I can't do it anymore. It isn't you, it's me.

So, what's the impact? Someone else will be either merging and releasing code, or we should close the project or let it move somewhere else. I cannot offer help merging change anymore, but I can offer help in getting someone access and teaching them how to do that.

For folks who are up to the task! HACKING file roughly describes the culture which effective java also influences. This was put together to prevent the largest amount of bugs and the least api breakage when introducing change. This is why spring was able to include feign in project work like spring cloud (not fear apis will break or change willy nilly). If this project is resumed, and the change culture is otherwise, bear in mind that this may limit the ability for others to support feign. This may nullify some of the goals of even proceeding. For this reason, whoever that takes over (if it is taken over), will need to think carefully about this, as damage caused by not well planned change or interface design have knock-on effects to anyone using the project. The ideal leaders of this project will be very nervous, especially about public facing api change or features that require java versions, or careful resource management. A different change culture can also be the case, but if that is I'd recommend a different group ID so that others aren't broken when things are merged otherwise.

Assignees: No one assigned

Labels: None yet

Projects: None yet

Milestone: No milestone

Notifications: Subscribe

You're not receiving notifications from this thread.

12 participants

AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

Feign as a declarative REST client

- In the Booking microservice, there is a synchronous call to Fare
- `RestTemplate` is used for making the synchronous call

The following code snippet is the existing code in the Booking microservice for calling the Fare service

```
Fare fare =
restTemplate.getForObject(FareURL +"/get?
flightNumber="+record.getFlightNumber()
+"&flightDate="+record.getFlightDate(),Fare.
class);
```

In order to use Feign, first we need to change the pom.xml file to include the Feign dependency

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


AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

Feign as a declarative REST client

- The next step is to create a new `FareServiceProxy` interface
- This will act as a proxy interface of the actual Fare service

```
@FeignClient(name="fares-proxy", url="localhost:8080/fares")
public interface FareServiceProxy {
  @RequestMapping(value = "/get", method=RequestMethod.GET)
  Fare getFare(@RequestParam(value="flightNumber") String
flightNumber, @RequestParam(value="flightDate") String flightDate);
}
```

- `@FeignClient` annotation tells Spring to create a REST client based on the interface provided

AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

Feign as a declarative REST client

- In the Booking microservice, we have to tell Spring that Feign clients exist in the Spring Boot application, which are to be scanned and discovered
- This will be done by adding `@EnableFeignClients` at the class level of `BookingComponent`. Optionally, we can also give the package names to scan.
- Change `BookingComponent`, and make changes to the calling part. This is as simple as calling another Java interface:

```
Fare = fareServiceProxy.getFare(record.getFlightNumber(),
record.getFlightDate());
```

- The URL of the Fare service in the FareServiceProxy interface is hardcoded:

```
url="localhost:8080/fares"
```


AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

Ribbon for load balancing

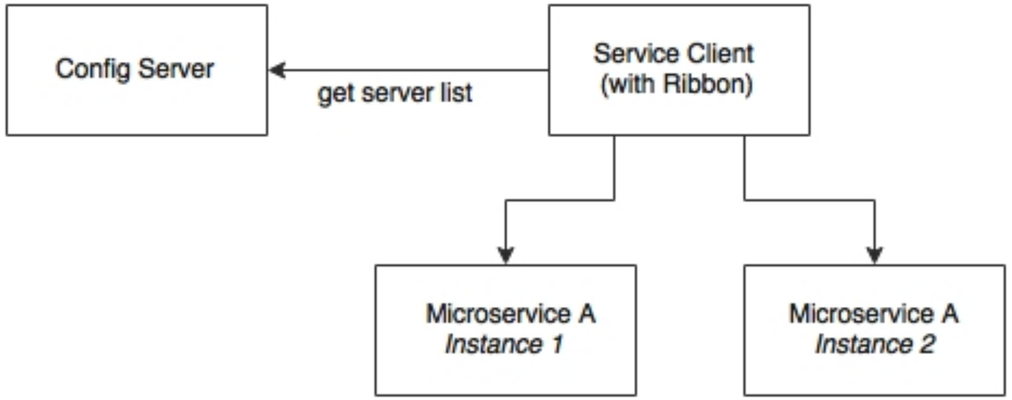
- In the previous setup, we were always running with a single instance of the microservice and the URL is hardcoded both in client as well as in the service-to-service calls
- In the real world, this is not a recommended approach, since there could be more than one service instance
  - load balancer or a local DNS server
- Ribbon is a client-side load balancer which can do round-robin load balancing across a set of servers

AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

Ribbon for load balancing

- The Ribbon client looks for the Config server to get the list of available microservice instances, and, by default, applies a round-robin load balancing algorithm

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



Ribbon for load balancing

- Update the Booking microservice configuration file, `booking-service.properties`, to include a new property to keep the list of the Fare microservices:

```
fares-proxy.ribbon.listOfServers=localhost:8080,localhost:8081
```


AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

Ribbon for load balancing

- Going back and editing the `FareServiceProxy` class created in the previous section to use the Ribbon client
- The value of the `@RequestMapping` annotations is changed from `/get` to `/fares/get` so that we can move the host name and port to the configuration easily

```
@FeignClient(name="fares-proxy")
@RibbonClient(name="fares")”
public interface FareServiceProxy {
  @RequestMapping(value = “fares/get", method=RequestMethod.GET)
```

- We can now run two instances of the Fares microservices. Start one of them on 8080, and the other one on 8081:

```
java -jar -Dserver.port=8080 fares-1.0.jar
java -jar -Dserver.port=8081 fares-1.0.jar
```


AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

Ribbon for load balancing

- Run the Booking microservice, the `CommandLineRunner` automatically inserts one booking record ~> first server
- When running the website project, it calls the Booking service. This request will go to the second server
- On the Booking service, we see the following trace, which says there are two servers enlisted:

```
DynamicServerListLoadBalancer:{NFLoadBalancer:name=fares-proxy,current
list of Servers=[localhost:8080, localhost:8081],Load balancer stats=Zone
stats: {unknown=[Zone:unknown;  Instance count:2;  Active connections
count: 0;  Circuit breaker tripped count: 0;  Active connections per
server: 0.0;]
},
```


AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

Eureka for registration and discovery

- If there is a large number of microservices, and if we want to optimize infrastructure utilization, we will have to dynamically change the number of service instances and the associated servers
- When targeting cloud deployments for highly scalable microservices, static registration and discovery is not a good solution considering the elastic nature of the cloud environment
- In the cloud deployment scenarios, IP addresses are not predictable, and will be difficult to statically configure in a file

Understanding dynamic service registration and discovery

- With dynamic registration, when a new service is started, it automatically enlists its availability in a central service registry
- Similarly, when a service goes out of service, it is automatically delisted from the service registry
- The registry always keeps up-to-date information of the services available, as well as their metadata

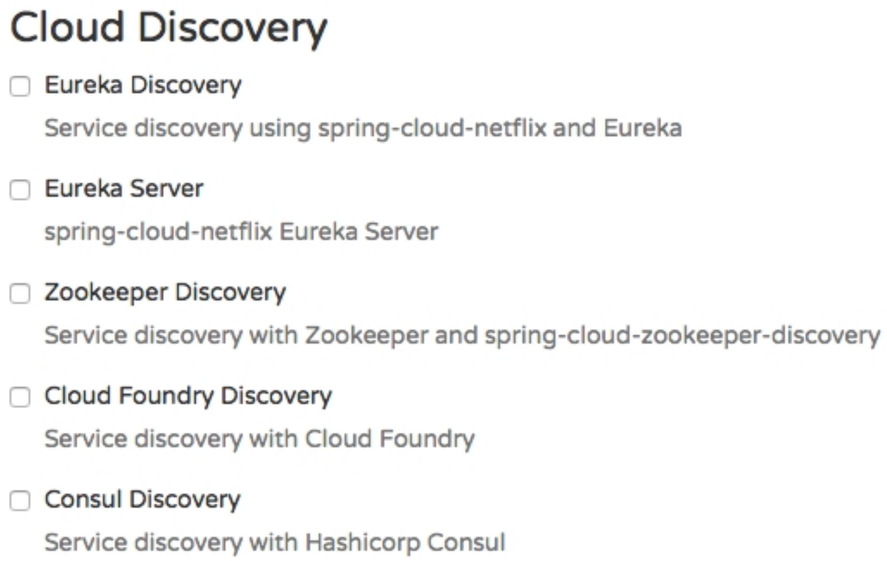
AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

Dynamic discovery

- Dynamic discovery is applicable from the service consumer's point of view
- Dynamic discovery is where clients look for the service registry to get the current state of the services topology, and then invoke the services accordingly
- In this approach, instead of statically configuring the service URLs, the URLs are picked up from the service registry.

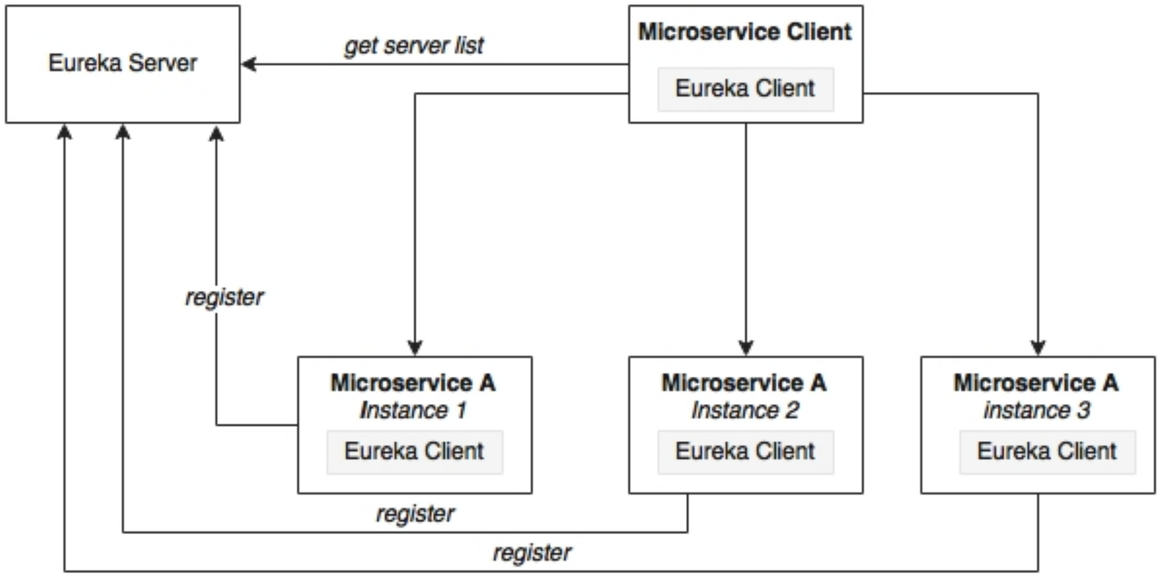
Understanding dynamic service registration and discovery

- There are a number of options available for dynamic service registration and discovery
  - Netflix Eureka, ZooKeeper, and Consul are available as part of Spring Cloud
- Etcd is another service registry available outside of Spring Cloud to achieve dynamic service registration and discovery



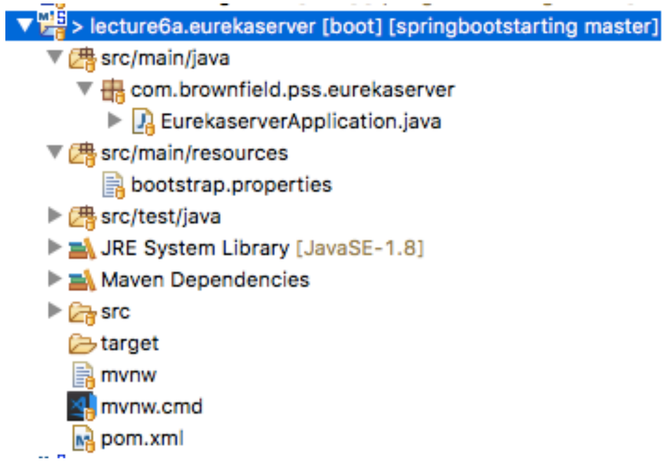
Understanding Eureka

- Eureka is primarily used for self-registration, dynamic discovery, and load balancing
  - Eureka uses Ribbon for load balancing internally



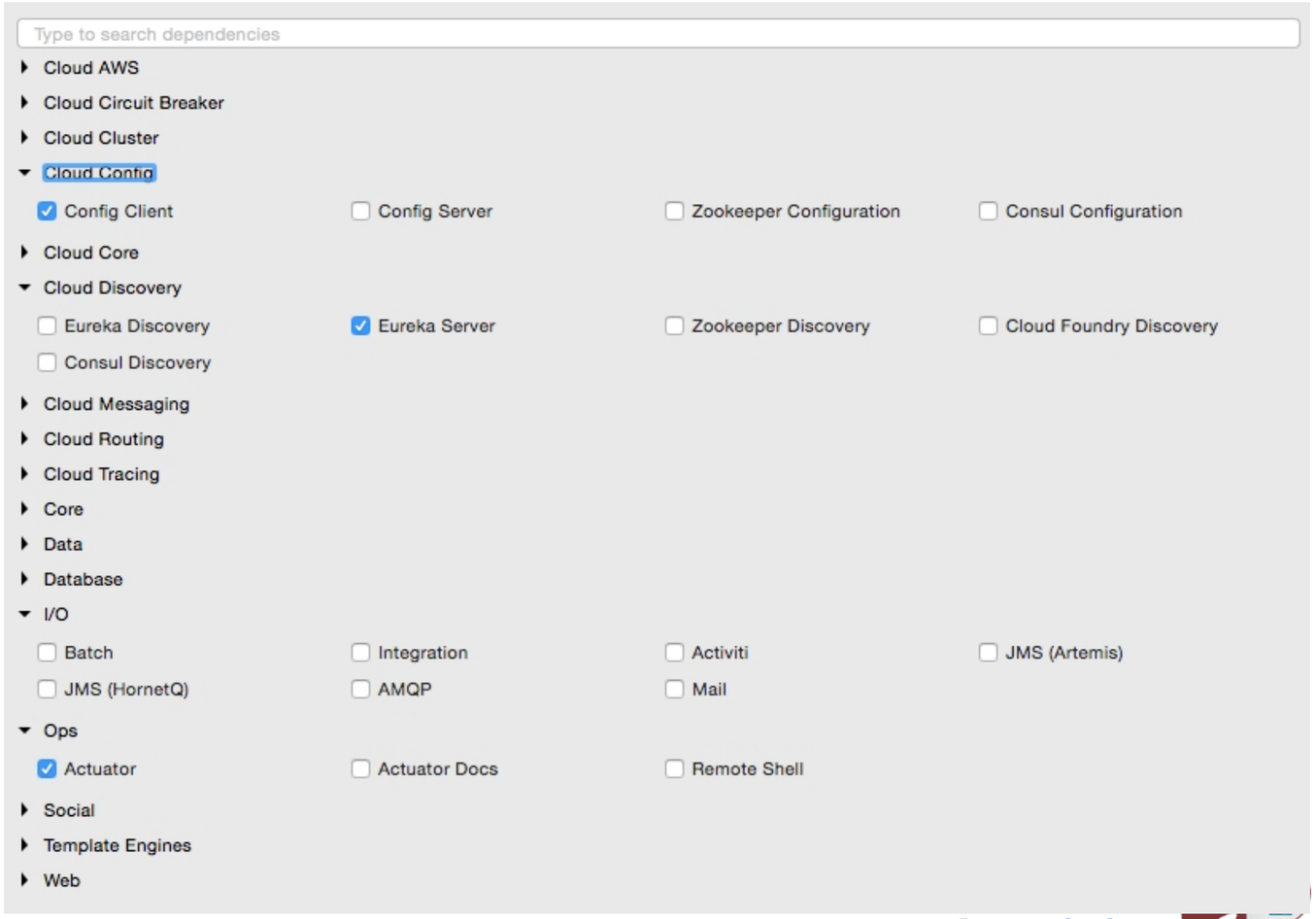
Setting up the Eureka server

- Start a new Spring Starter project, and select Config Client, Eureka Server, and Actuator



bootstrap.properties

```
1 spring.application.name=eureka-server1
2 server.port:8761
3 spring.cloud.config.uri=http://localhost:8888
```



Setting up the Eureka server

- The Eureka server can be set up in a standalone mode or in a clustered mode
- By default, the Eureka server itself is another Eureka client
  - This is particularly useful when there are multiple Eureka servers running for high availability
  - The client component is responsible for synchronizing state from the other Eureka servers
- The Eureka client is taken to its peers by configuring the `eureka.client.serviceUrl.defaultZone` property

AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

Setting up the Eureka server

- Create a `eureka-server1.properties` file, and update it in the Git repository.
- `eureka-server1` is the name of the application given in the application's `bootstrap.properties` file in the previous step
- `serviceUrl` points back to the same server

```
spring.application.name=eureka-server1
eureka.client.serviceUrl.defaultZone:http://localhost:8761/eureka/
eureka.client.registerWithEureka:false
eureka.client.fetchRegistry:false
```


AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

Setting up the Eureka server

- In EurekaserverApplication, add `@EnableEurekaServer`:

```
@EnableEurekaServer
@SpringBootApplication
public class EurekaserverApplication {
```

- Start the Eureka server, once the application is started, open `http://localhost:8761` in a browser to see the Eureka console
- In the console, note that there is no instance registered under Instances currently registered with Eureka
  - Since no services have been started with the Eureka client enabled, the list is empty at this point.

AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

Setting up the Eureka server

- Making a few changes to our microservice will enable dynamic registration and discovery using the Eureka service. To do this, first we have to add the Eureka dependencies to the `pom.xml` file

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

- The following property has to be added to all microservices in their respective configuration files under config-repo

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


AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

Setting up the Eureka server

- Add @EnableDiscoveryClient to all microservices in their respective Spring Boot main classes
- Start all servers except Booking. Since we are using the Ribbon client on the Booking service, the behavior could be different when we add the Eureka client in the class path
- Going to the Eureka URL (http://localhost:8761), you can see that all three instances are up and running

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CHECKIN-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.102:checkin-service:8070
FARES-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.102:fares-service:8080
SEARCH-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.102:search-service:8090

AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

Time to fix the issue with Booking

- We will remove our earlier Ribbon client, and use Eureka instead. Eureka internally uses Ribbon for load balancing

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

- Also remove the `@RibbonClient(name="fares")` annotation from the FareServiceProxy class
- Update `@FeignClient(name="fares-service")` to match the actual Fare microservices' service ID
  - In this case, fare-service is the service ID configured in the Fare microservices' bootstrap.properties. This is the name that the Eureka discovery client sends to the Eureka server

AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

Time to fix the issue with Booking

- Also remove the list of servers from the `booking-service.properties` file. With Eureka, we are going to dynamically discover this list from the Eureka server:

```
fares-proxy.ribbon.listOfServers=localhost:8080, localhost:8081
```

- Start the Booking service. You will see that `CommandLineRunner` successfully created a booking, which involves calling the Fare services using the Eureka discovery mechanism

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
BOOK-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.102:book-service:8060
CHECKIN-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.102:checkin-service:8070
FARES-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.102:fares-service:8080
SEARCH-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.102:search-service:8090

AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

Time to fix the issue with Booking

- Change the website project's `bootstrap.properties` file to make use of Eureka rather than connecting directly to the service instances
- We will not use the Feign client in this case, we will use the load balanced `RestTemplate`. Commit these changes to the Git repository:

```
spring.application.name=test-client

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

- Add `@EnableDiscoveryClient` to the Application class to make the client Eureka-aware

AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

Time to fix the issue with Booking

- Edit both `Application.java` as well as `BrownFieldSiteController.java`
- Add three `RestTemplate` instances. This time, we annotate them with `@Loadbalanced` to ensure that we use the load balancing features using Eureka and Ribbon.
  `RestTemplate` cannot be automatically injected

```
@Configuration
class AppConfiguration {
    @LoadBalanced
    @Bean
    RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
@Autowired
RestTemplate searchClient;
@Autowired
RestTemplate bookingClient;
@Autowired
RestTemplate checkInClient;
```


AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática 40 años

Time to fix the issue with Booking

- We use these `RestTemplate` instances to call the microservices.
- Replace the hardcoded URLs with service IDs that are registered in the Eureka server.
- In the following code, we use the service names `search-service`, `book-service`, and `checkin-service` instead of explicit host names and ports

```
Flight[] flights = searchClient.postForObject("http://search-service/search/get", searchQuery,
Flight[].class);

long bookingId = bookingClient.postForObject("http://book-service/booking/create", booking, long.class);

long checkinId = checkInClient.postFor

Object("http://checkin-service/checkin/create", checkIn, long.class);
```


AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

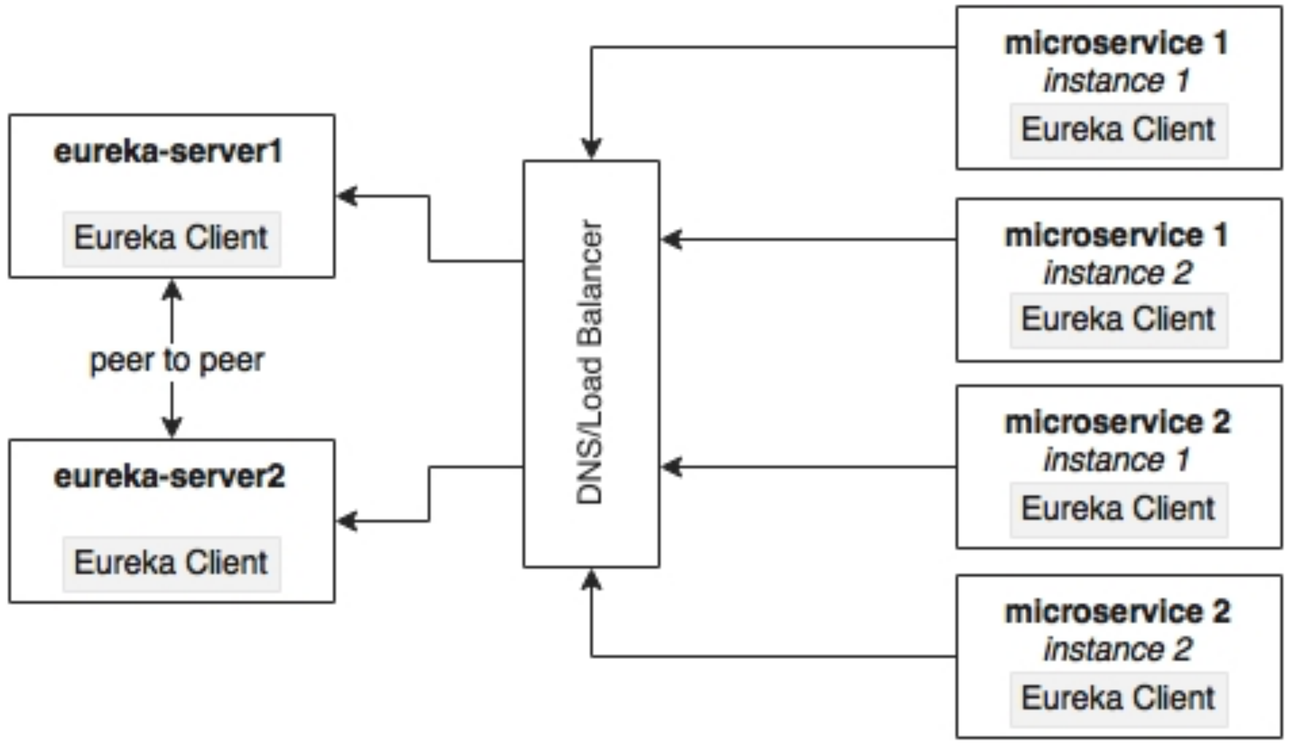
Time to fix the issue with Booking

- We are now ready to run the client
- Run the website project
  - If everything is fine, the website project's `CommandLineRunner` will successfully perform search, booking, and check-in
- The same can also be tested using the browser by pointing the browser to `http://localhost:8001`

AssertLab
Advanced Software and Systems Engineering Research Technologies
Centro de Informática
40 años

Homework 6.1

- How to get high availability with Eureka?



API Gateway

- In most microservice implementations, internal microservice endpoints are not exposed outside
- They are kept as private services. A set of public services will be exposed to the clients using an API gateway
  - Only a selected set of microservices are required by the clients.
  - If there are client-specific policies to be applied, it is easy to apply them in a single place rather than in multiple places. An example of such a scenario is the cross-origin access policy.
  - It is hard to implement client-specific transformations at the service endpoint.
  - If there is data aggregation required, especially to avoid multiple client calls in a bandwidth-restricted environment, then a gateway is required in the middle.

Homework 6.2

- How to use Zuul proxy as the API gateway?
- How to setup High availability capability of Zuul?