(..)

Requirements

- A cloned version of the code repository (https://github.com/platform-acceleration-lab/apps-cloud-native-evolution-code) and finished steps 1 and 2 in the cloud native evolution.

What you will learn/review

- Implement Circuit Breaker patterns via Spring Cloud Services
- Leverage GitHub based configuration for a Spring Cloud Services configuration server

Implement Circuit breakers with Spring Cloud Services

At this point, we have solved the discovery problem for microservices, but things are still not perfect. If the Billing application goes down, the UMS application will also fail.

We can use the circuit breaker pattern to prevent cascading failures from bringing down the whole system.

See the Spring Cloud Services Circuit Breaker introduction (https://docs.pivotal.io/spring-cloud-services/circuit-breaker/) for more background.

See Writing Client Applications (https://docs.pivotal.io/spring-cloud-services/circuit-breaker/writing-client-applications.html) for client implementation details.

1. Confirm that the `spring cloud eureka` command is still running in a terminal window. If not, re-start it.
2. Enable circuit breakers in both the UMS and billing application via `@EnableCircuitBreaker`. You will need to add the proper dependencies for each application to get the annotation.

3. Annotate the method in `BillingClient` that could fail as a result of a service that is down with the `@HystrixCommand` annotation and implement a fallback method that logs the failure, with the user id and amount, at the `info` level. In real life, you might send an email or some other fallback. You will need to determine the minimum dependency that you need to include to get the annotation in the billing component. **Hint**: Do a Google search for `@HystrixCommand` and see if you can find the Gradle dependency entry.

4. Start both applications locally using the `bootRun` task and confirm via `curl` that they still work as expected.

5. Stop the Billing application and test again via `curl`. The UMS application still works.

6. Use the spring cloud cli to view a local hystrix dashboard with the command `spring cloud eureka hystrixdashboard` and view the circuit breaker dashboard.

> The dashboard won't show anything until the method with the Hystrix annotation is actually called.

Now that we have our microservices running locally, deploy them to PCF and verify they work in the cloud.

1. Create a `p-circuit-breaker-dashboard` service instance and bind it to both the UMS and Billing applications.

2. Re-build and re-deploy both applications.

3. Confirm that the applications deployed to PCF still work as expected via `curl`.

4. Open up the circuit breaker dashboard in the PCF Apps Manager and confirm that the `billUser` circuit is displayed.

5. Stop the Billing application and test again via `curl`. The UMS application still works.

6. Notice that the circuit breaker dashboard reports the circuit as open.

7. Restart Billing and `curl` again a couple of times. The circuit will report as closed. This visibility into failing services is very important for the proper monitoring of a microservices based architecture.

Did that all work? If not, see if you can use `cf logs` to figure out the cause of any errors.

Questions:

1. Why is service resiliency important in a microservices based architecture?

Issues

1. If you have trouble viewing the `hystrix.stream` for your applications locally via the Hystrix Dashboard, try it out on PCF before worrying about it. We sometimes saw buggy behavior locally.

Implement distributed configuration via Spring Cloud Services

As our microservice based system gets larger, we need to easily manage configuration across applications and environments. We may also need to keep a record of configuration changes for auditing purposes.

We can use the configuration server to make all of this happen.

See the Spring Cloud Services Config Server introduction (https://docs.pivotal.io/spring-cloud-services/config-server/) for more background.

See Writing Client Applications (https://docs.pivotal.io/spring-cloud-services/config-server/writing-client-applications.html) for implementation details.

1. Stop the `spring cloud` command if it is still running.
2. Create a new GitHub repository to hold your configuration by forking the class repository (https://github.com/platform-acceleration-lab/apps-platform-acceleration-spring-configuration-code) and cloning `YOUR FORK` to your local machine.
3. Configure the config server within the Spring Cloud CLI to point to this GitHub repository. See the documentation (http://cloud.spring.io/spring-cloud-static/spring-cloud-cli/1.2.1.RELEASE/#_running_spring_cloud_services_in_development) for details. You place the `configserver.yml` file in the root of the code directory. The URI is a file based URI pointing to your local clone.
4. Add the `security.basic.enabled=false` property to the `application.yml` file in your configuration repository.
5. Re-start the `spring cloud` command and confirm that both the Config Server and the Eureka server start.

   ```
   curl http://localhost:8888/foo/development
   ```

   The output should look similar to the following:

```
{"name":"foo","profiles":["development"],"label":"master","version
":"2ecec0bc9d25efd91580afa68e0768f43ccb8053","state":null,"propert
ySources":[{"name":"file:///Users/mikegehard/workspace/edu/platfor
m-acceleration-spring-configuration/application.yml","source":{"eu
reka.client.serviceUrl.defaultZone":"http://localhost:8761/eureka/
","eureka.client.instance-info-replication-interval-seconds":5,"eu
reka.client.initial-instance-info-replication-interval-seconds":6}
}]}
```

To test the Eureka server, navigate a browser window to `http://localhost:8761/`.

6. Modify both applications to use the configuration server. See Spring Cloud Services docs for details.
7. Start both applications with the `bootRun` Gradle task.
8. Test the UMS application with `curl` to confirm that everything is working correctly. Assure yourself that the fallback method to the billing service is not being called. You may have trouble communicating with the billing service until it has a chance to register itself with Eureka.
9. Commit and push your changes to the configuration GitHub repo.

Now that we have our microservices running locally, deploy them to PCF and verify they work in the cloud.

1. Create a `p-config-server` service instance and bind it to both the UMS and Billing applications.
2. Update the config server instance to use your repository as it's configuration source via the CLI `update-service` command.
3. Re-build and re-deploy both applications to PCF.
4. Test your services using the `curl` commands above. The responses should stay the same.

Issues

1. You get a 401 when you curl the UMS server. Did you configure the config server correctly? Did the log files for the applications show any errors?
2. If you have added the `spring-boot-starter-actuator` to your project, and disabled basic auth for it `management.security.enabled=false`, make sure you put that configuration in an application specific file, ie. `ums.yml`. If it is in `application.yml` it might break your local Eureka instance brought up by the `spring cloud` command.

3. If your PCF Service Registry stops working after moving config to the server, check to make sure you don't have the following in your `application.yml`:

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
    instance-info-replication-interval-seconds: 5
    initial-instance-info-replication-interval-seconds: 6
```

   This configuration may be overriding the auto configuration of the PCF Service Registry. Either remove it, or move it to a "development" specific profile.

Assignment

Once you are done with this section and the application is deployed and working on PCF, you can submit the assignment using the `submitDistributedConfig` gradle task. It requires you to provide the `umsUrl`, `username` and `password` project properties. The default user is `user`. Your password can be found in your `ums` logs. For example:

```
cd ~/workspace/assignment-submission
./gradlew submitDistributedConfig -PumsUrl=http://my-ums.cfapps.io -Pu
sername=user -Ppassword=secret
```