(..)

Requirements

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

What you will learn/review

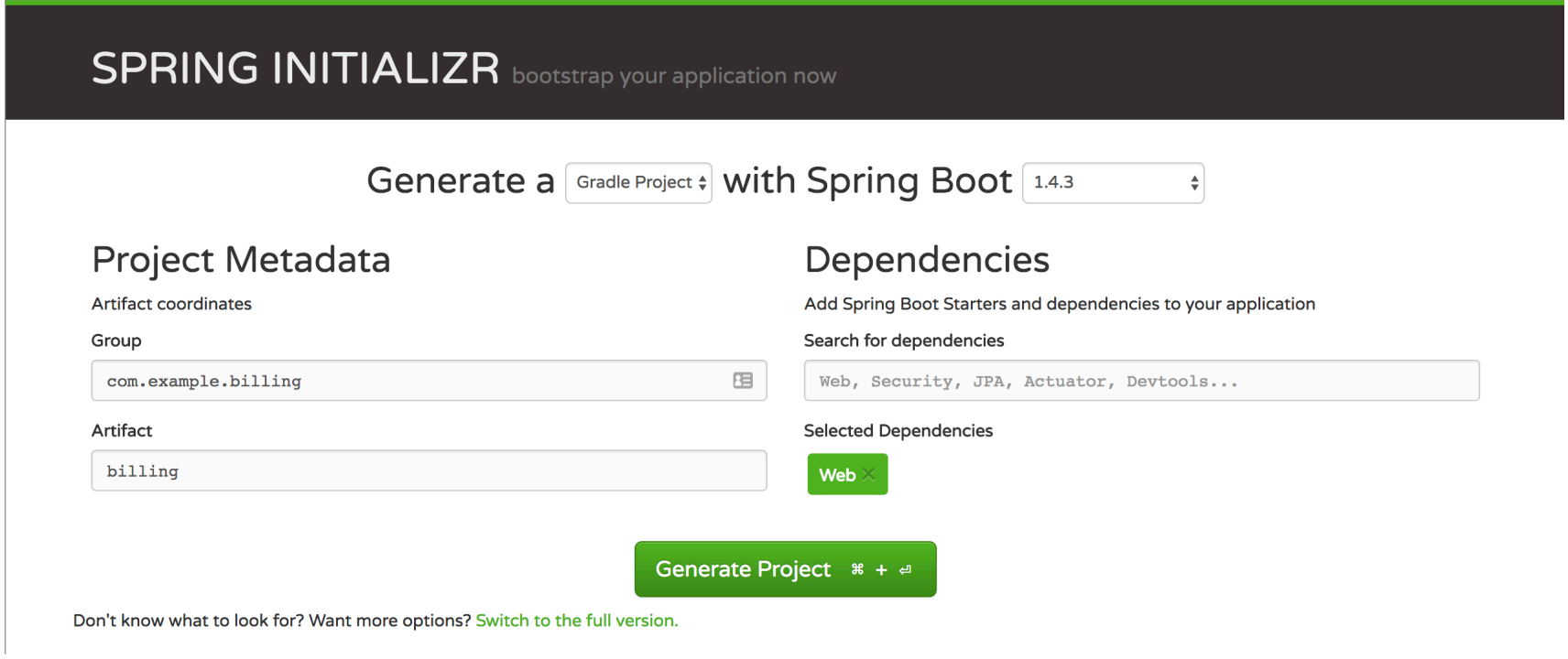
- How to test service endpoints using `curl`
- How to extract a microservice from a well structured monolith
- Deploy microservice applications to PCF
- Implement the Service Discovery via Spring Cloud Services

Extract the first service from the well structured monolith

Creating a new Spring Boot application for your service.

1. Create a new Spring Boot web application from the Spring Initializer (http://start.spring.io/) called `billing`. Put it in the `applications` directory.

   NOTE: For consistency with our code examples, we created a "Gradle" project with `com.example.billing` as the "Group", `billing` as the "Artifact", and incuded the "Web" dependency.



2. Remove the `applications/billing/gradle` directory and the `applications/billing/gradlew`, `applications/billing/gradlew.bat`, and `.gitignore` files. We will include our new application in the top level Gradle build.
3. Add the `applications/billing` application to the top level `settings.gradle` file.
4. Confirm that your new application successfully builds using `./gradlew applications/billing:build`.

Extracting payment logic into the new service.

You will be moving all logic related to payments into the new billing service.

1. Implement a controller in `applications/billing` to bill users. This controller should have one endpoint, 'reoccurringPayment' that responds to a POST request. The requirements for this controller are:
    - Inject the payment gateway
    - Accepts one integer request body parameter
    - Returns JSON in the format of `{errors: []}`
    - Returns a `ResponseEntity` with a status of `CREATED` if a call to the `createReoccurringPayment` method of a payment gateway returns true. This code already exists in the billing component, you just need to move it in to the billing application.
    - Returns a `ResponseEntity` with a status of `BAD_REQUEST` and a descriptive error message otherwise.
2. Introduce a `Client` class into `components/billing`. This class will communicate with the new service. It has the following requirements:
    - a constructor that takes a service endpoint string for the new billing service.
    - one method called `billUser` that takes a `userId` String and an Integer that is the

amount to bill the user. Inside this method, use an instance of Spring's `RestTemplate` class to post the amount to the `/reoccurringPayment` endpoint on the new billing service.

3. Modify the `CreateSubscription` class in `components/subscriptions` to use an injected `BillingClient` class instead of the `ChargeUser` class.
4. Modify the `com.example.ums.subscriptions.Controller` to inject the `BillingClient` into the `CreateSubscription` instance. The `BillingClient` will be `@Autowired` into the controller and the billing endpoint will be configured as a configuration property in `applications/ums`.
5. Build `applications/ums` to make sure that everything is correctly wired together.
6. Start both applications using `./gradlew applications/[applicationName]:bootRun`.
7. Test your system by creating a subscription using `curl`.

If you would like to check your work, you can see a diff of this step here (https://github.com/platform-acceleration-lab/apps-cloud-native-evolution-code/compare/v1...v2).

Questions to answer:

1. How does the UMS application know where the Billing application is located?
2. How would you tell the UMS application about a second instance of the Billing application that was added because of load?
3. What happens if the Billing application is not running when a subscription is created in the UMS application?

Deploy both microservices to PCF

1. Build and deploy the Billing application (applications/billing/build/libs/billing-0.0.1-SNAPSHOT.jar) to PCF using a random route.
2. Build and deploy the UMS application to get the new changes.
3. Test your system by creating a subscription using `curl`.

   Did that work? If not, why not?

   Hint: What `cf` command can help you figure out what went wrong? Look for a log entry that looks like this:

```
2016-12-22T13:42:37.17-0700 [APP/0]      OUT 2016-12-22 20:42:37.177 ERROR 20 --- [nio-8080-exec-5] o.a.c.c.C.[.[.[/].[dispatcherServlet]    : Servlet.service() for servlet [dispatcherServlet] in context with path [] threw exception [Request processing failed; nested exception is org.springframework.web.client.ResourceAccessException: I/O error on POST request for "http://localhost:8081/reoccurringPayment":Connection refused (Connection refused); nested exception is java.net.ConnectException: Connection refused (Connection refused)] with root cause
```

Implement service discovery via Spring Cloud Services

At this point, we have a "microservice" architecture, but things are still not ideal since we still need to orchestrate the consumer app and the backing service.

We can use service discovery to hook up our microservices so that when they scale or move we do not have to re-deploy their clients.

See the Spring Cloud Services Service Registry introduction (https://docs.pivotal.io/spring-cloud-services/service-registry/) for more background.

See the Spring Cloud Service docs (https://docs.pivotal.io/spring-cloud-services/service-registry/writing-client-applications.html) for the full details on implementing a service discover client.

1. To ease our local development, install the spring CLI (http://docs.spring.io/spring-boot/docs/current/reference/html/getting-started-installing-spring-boot.html#getting-started-homebrew-cli-installation) as well as the spring cloud extension (https://cloud.spring.io/spring-cloud-cli/) with `spring install org.springframework.cloud:spring-cloud-cli:1.2.1.RELEASE` You can now run `spring cloud` and get several of the components of Spring Cloud Services running locally- but for more fine grained control, start just a Eureka server with `spring cloud eureka`.

2. Name both the UMS and Billing applications in their respective configuration files. This is necessary so Eureka can identify their instances in the dashboard.

3. Register both the UMS and billing application with Eureka via `@EnableDiscoveryClient`. You will need to add the proper dependencies for each application to get the annotation.

> Be careful here! There are a few packages that provide a Spring friendly version of Eureka - make sure you pull in the one that will configure itself with Spring Cloud Services. If your unclear about which one to use, refer back to the Spring Cloud Service docs (https://docs.pivotal.io/spring-cloud-services/service-registry/writing-client-applications.html) you read above.

4. Run both applications locally via the `bootRun` task and confirm that they both register with the service discovery service via the Eureka UI (http://localhost:8761).
5. Define a `RestTemplate` bean annotated with the `@LoadBalanced` annotation, which is what makes it aware of Ribbon and Eureka.
6. Modify the billing client to get an injected `RestTemplate`. Use the name the billing service has registered with Eureka as the base URL instead of the passed in value.

> If using the service name in the RestTemplate is not working correctly, check your dependencies. You may be getting one that is not aware of eureka.

7. Restart the UMS application and test your services using `curl`.

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

1. Create a `p-service-registry` service instance and bind it to both the UMS and Billing applications.
2. Re-build and re-deploy both applications.
3. Confirm that both applications register with the service discovery service via the Eureka UI in PCF Apps Manager.
4. Test your services using `curl`.

Questions:

1. What benefits do you get from using Eureka instead of the hard coded version from above?

Issues

1. If you encounter problems with the `ums` application reporting that there are no available instances of `billing`, try booting the `billing` application first, and waiting for it to come up before booting `ums`.
2. If your applications have trouble communicating with the `p-service-registry` service you created on PCF, make sure you follow the instructions around adding the self-signed ssl certificate to the JVM Truststore (https://docs.pivotal.io/spring-cloud-services/service-registry/writing-client-applications.html#self-signed-ssl-certificate).

Assignment

Once you are done with this section and the application is deployed and working on PCF, you can submit the assignment using the `submitServiceDiscovery` gradle task.

It requires the `umsUrl` project property, for example:

```
cd ~/workspace/assignment-submission
./gradlew submitServiceDiscovery -PumsUrl=http://my-ums.cfapps.io
```