Pivotal

Workshop Hands On – Spring Boot to PCF

The goal of this workshop is to guide you towards building your first Spring Boot microservice application, complete with its own backing database. You will also go through the paces of deploying this app to Pivotal Application Service and experience first hand its ease and benefits. You will also learn how you can easily monitor you microservice application simply by adding a dependency.

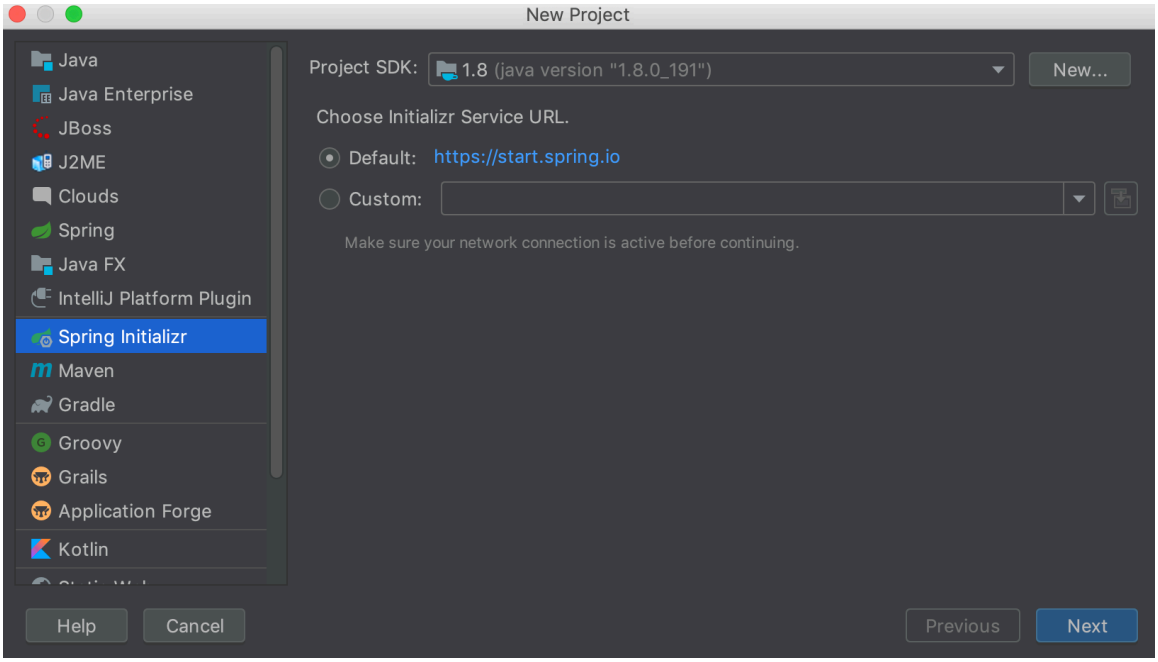
Note: The full source code for this workshop is in GitHub. You can access it publicly at https://github.com/derrick81/springboot-to-pcf.

Contents

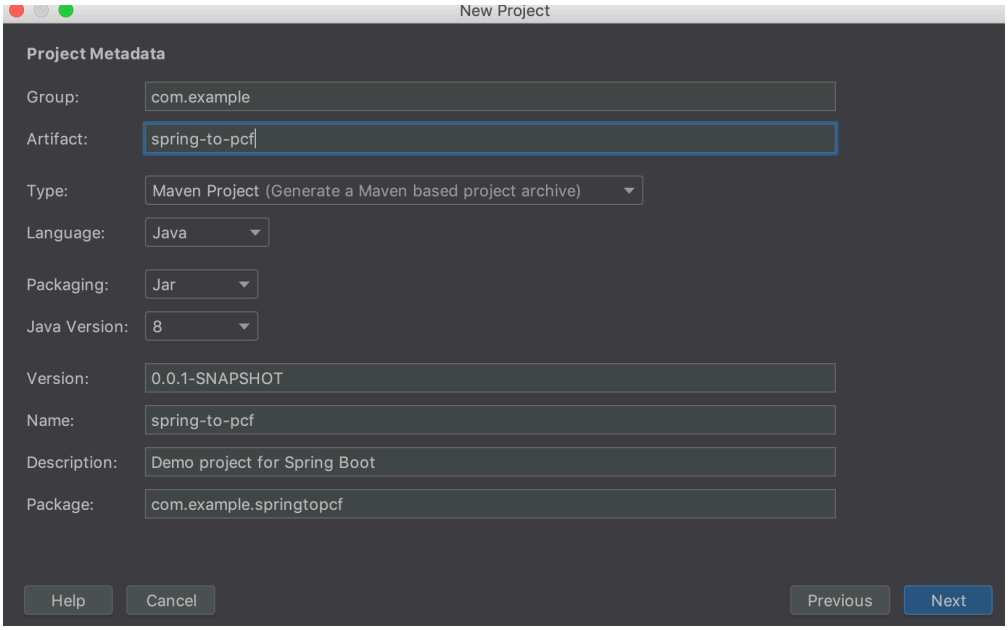
Pivotal

1. Your first Spring Boot microservice

1. Start up the IntelliJ IDE and select the “Create New Project” option. Select “Spring Initializr” on the left hand panel and click “Next”.
Note: Alternatively you can also create the project using your browser by going to http://start.spring.io. However, you will have to import the project into your IDE if you do so.



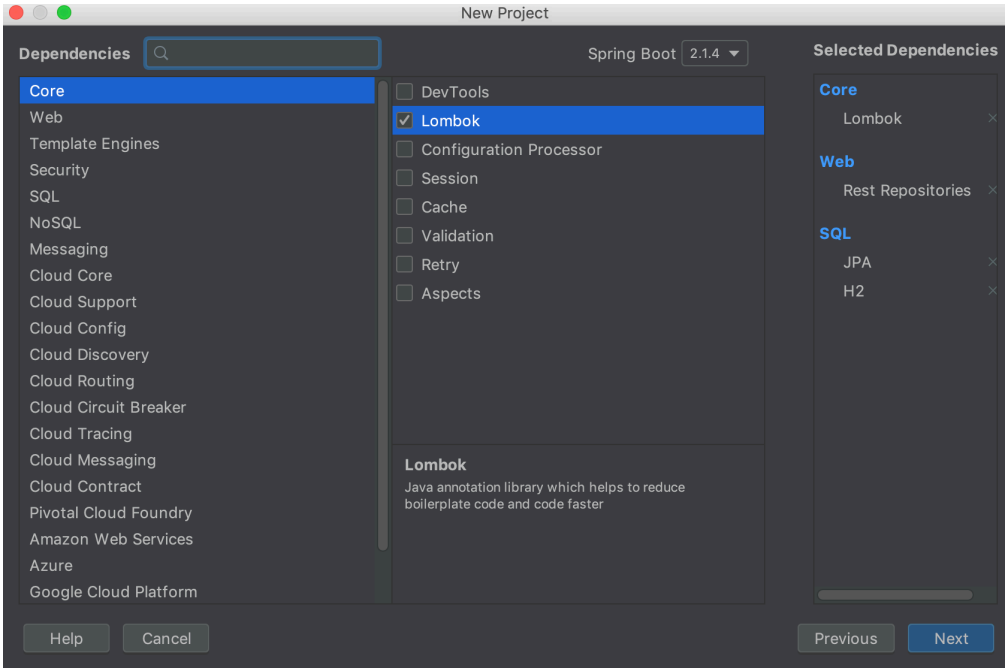
2. Enter “spring-to-pcf” as the artifact name, leaving the rest of the fields to their default values.



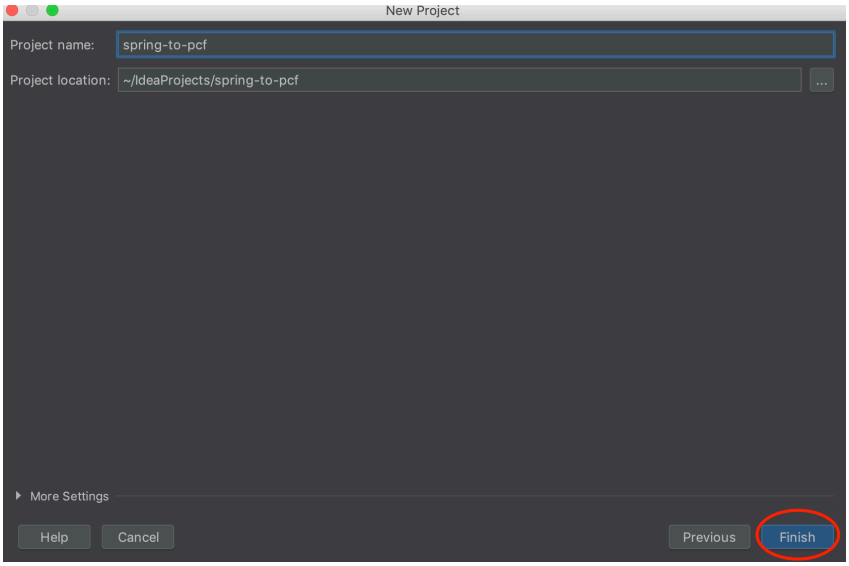
Pivotal

3. Select the following dependencies:
   - H2
   - JPA
   - Rest Repositories
   - Lombok

   You can use the search function to quickly identify these dependencies. Click “Next” to continue.

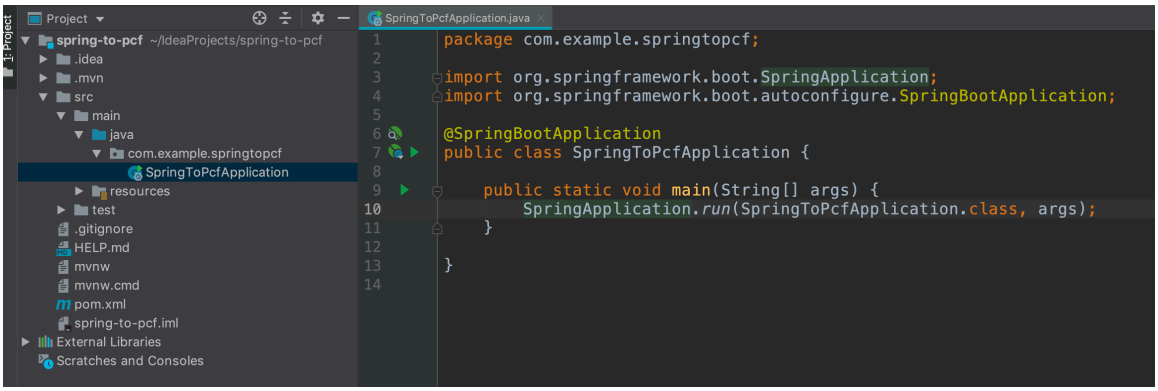


4. Click “Finish” and wait for Maven to complete downloading the required JAR files. Note: If a notification appears saying “Maven projects need to be imported”, please select “Enable Auto Import”. This can take a few minutes.



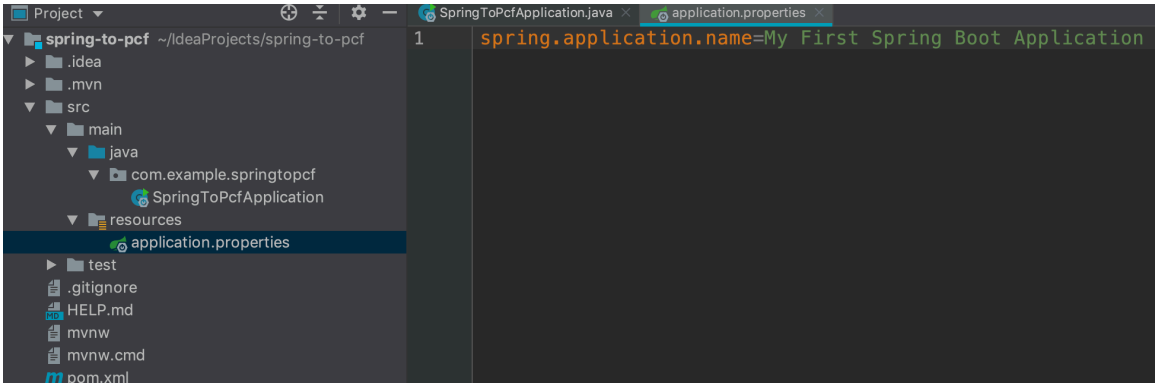
Pivotal

5. Open the SpringToPcfApplication file. Take note of this annotation:
@SpringBootApplication
You can find out more about this annotation at https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-using-springbootapplication-annotation.html
In this application, Spring Boot will scan the class path and given H2 JAR files exist will auto configure a Data Source for us out of the box. Settings can be overridden using the "application.properties" or "application.yml" file.



6. Set the microservice application name by adding the following property to application.properties:

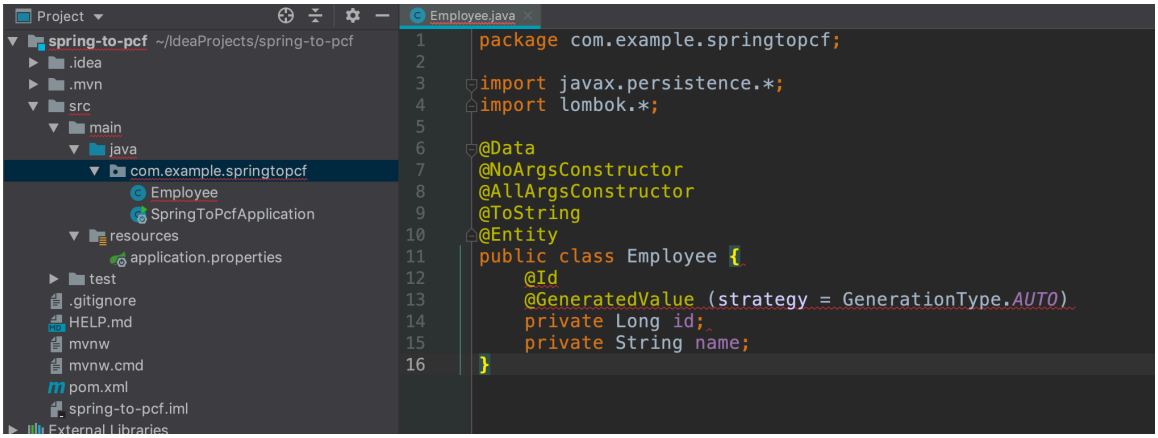
```
spring.application.name=My First Spring Boot Application
```



Pivotal

7. Add a new Java class named “Employee” and make it a JPA @Entity. We will also make use of Lombok annotations (see https://projectlombok.org/features/all) to save us from typing boiler plate codes. Here is the code for the class:

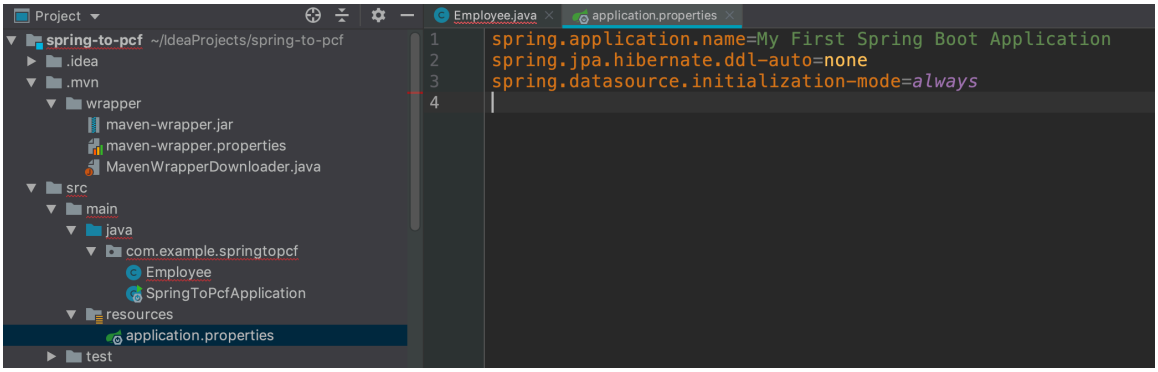
```
package com.example.springtopcf;

import javax.persistence.*;
import lombok.*;

@Data
@NoArgsConstructor
@AllArgsConstructor
@ToString
@Entity
public class Employee {
   @Id
   @GeneratedValue (strategy = GenerationType.AUTO)
   private Long id;
   private String name;
}
```



8. By virtue of the Spring Boot starter, Hibernate libraries have been added in as the JPA provider. Now we will specify some of the Hibernate properties. Add the following two lines to application.properties:

```
spring.jpa.hibernate.ddl-auto=none
spring.datasource.initialization-mode=always
```

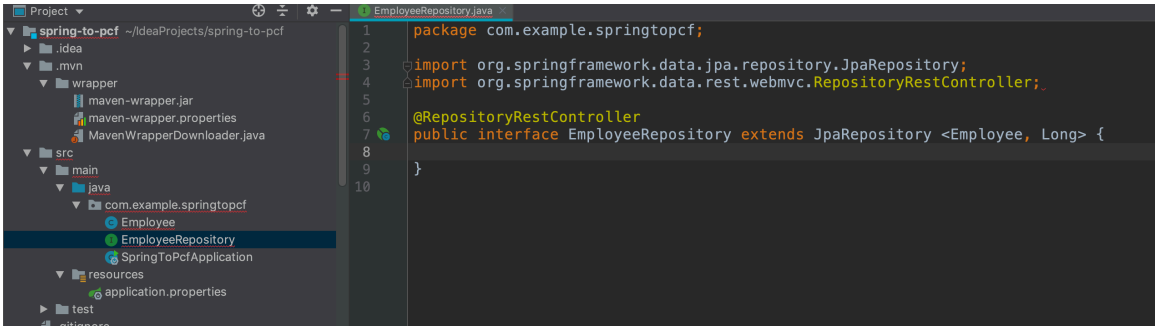
The resulting application.properties should look like this:



Pivotal

9. Next we will create REST endpoints for Employee repository. We will do so using REST Repositories. With the "@RepositoryRestResource" annotation, Spring Data REST exposes a collection resource named after the uncapitalized, pluralized version of the domain class in this case "employee". This is all done using Spring HATEOAS which it's goal is all about giving the client as much information as possible to be able to navigate the API. Here is the code for the interface:

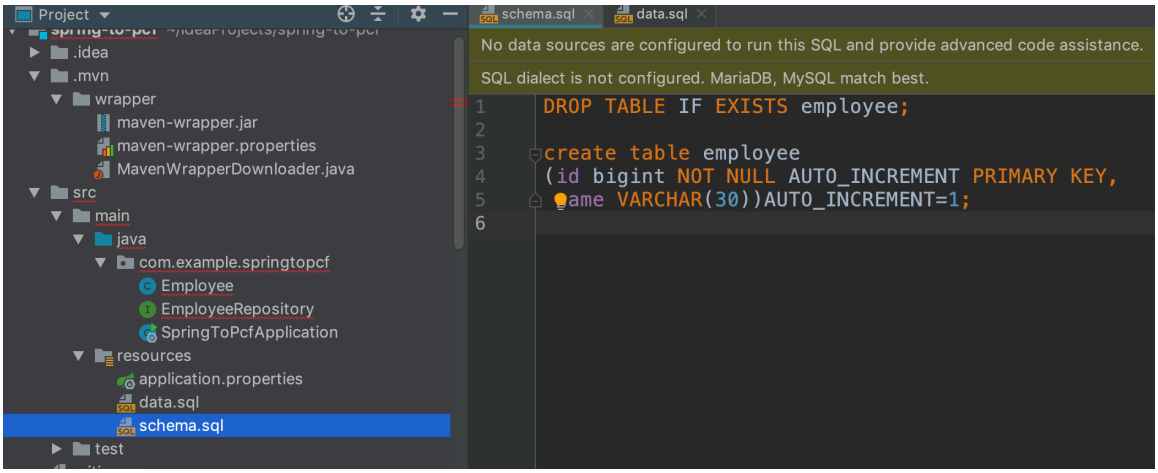
```
package com.example.springtopcf;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.rest.webmvc.RepositoryRestController;

@RepositoryRestController
public interface EmployeeRepository extends JpaRepository <Employee, Long> {

}
```



10. Now we will add some scripts that will provide for both the schema and the records.

    For the schema, please add schema.sql to the resources folder with the following script:

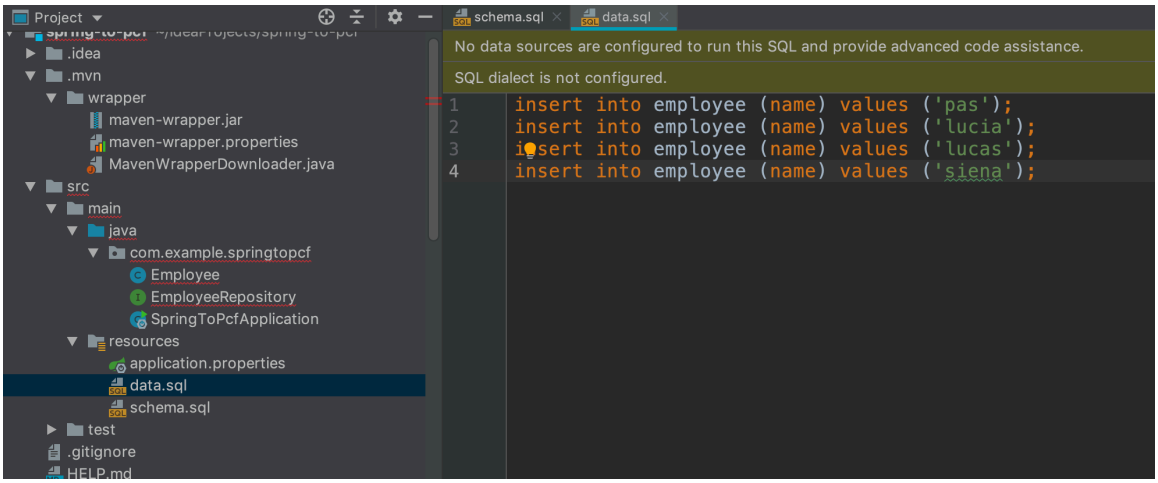
```
DROP TABLE IF EXISTS employee;

create table employee
(id bigint NOT NULL AUTO_INCREMENT PRIMARY KEY,
 name VARCHAR(30))AUTO_INCREMENT=1;
```



Pivotal

For the records , please add data.sql to the resources folder with the following script:

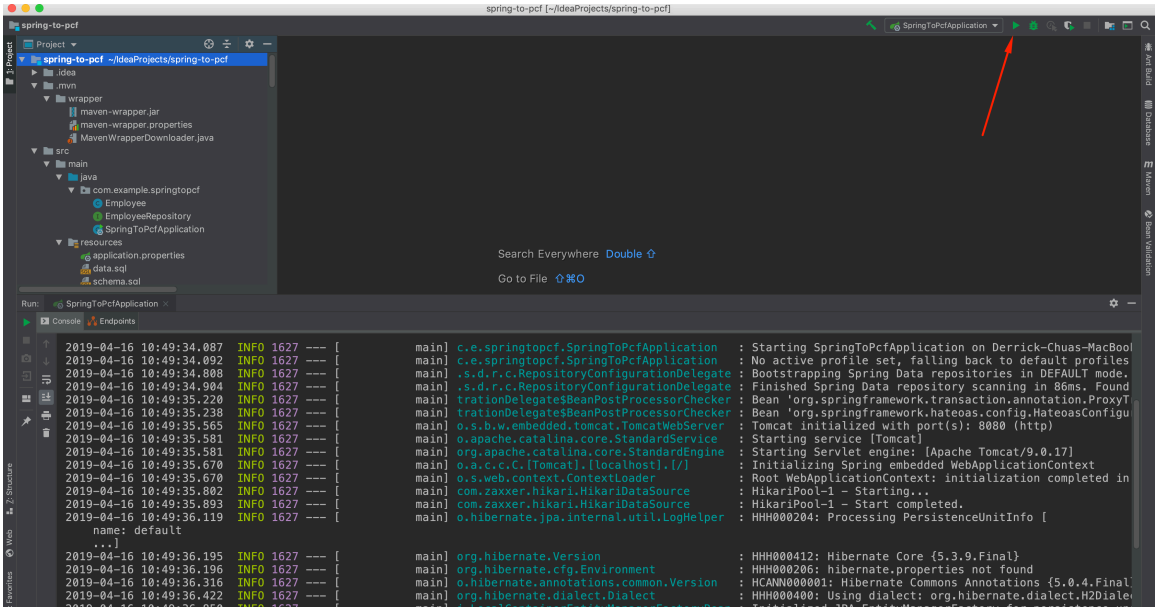
```
insert into employee (name) values ('pas');
insert into employee (name) values ('lucia');
insert into employee (name) values ('lucas');
insert into employee (name) values ('siena');
```



11. Now lets run our Spring Boot application by clicking on the green “Run” button as illustrated.

Few things to Note:

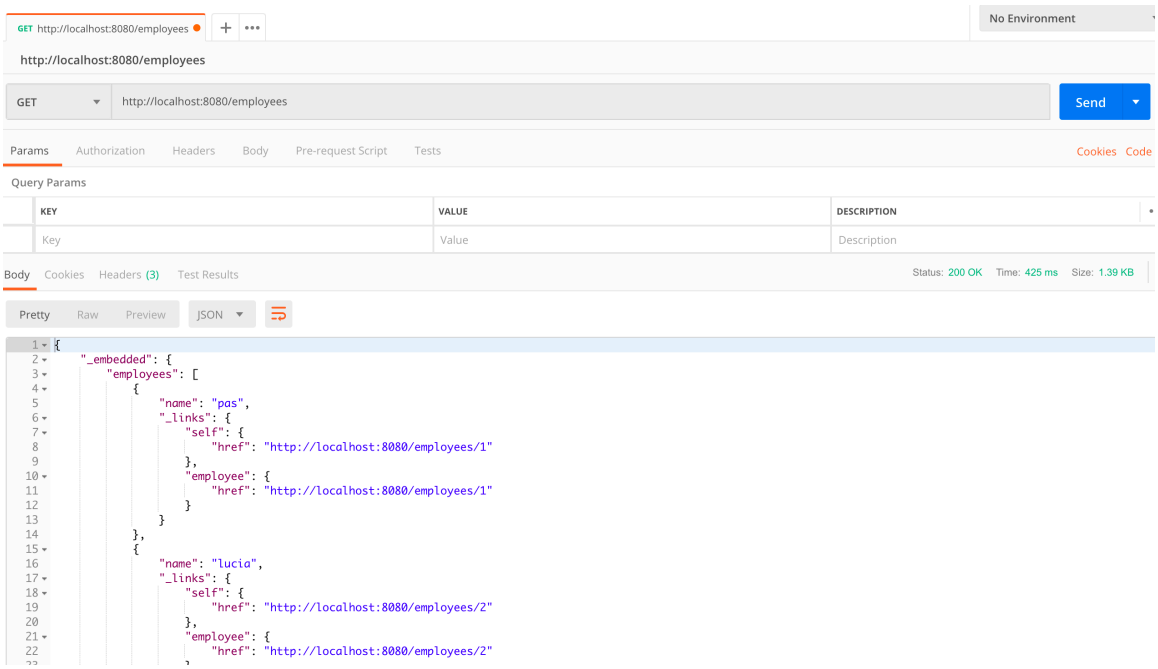
- Tomcat is bundled and used to run our application as per the spring starter pom.xml entries
- data.sql will be run once JPA has created the underlying table’s in the database
- Various Spring Boot Auto Configurations have been run



Pivotal

12. Your first Spring Boot microservice is now running. You can test this using Postman by executing a GET request on

    http://localhost:8080/employees



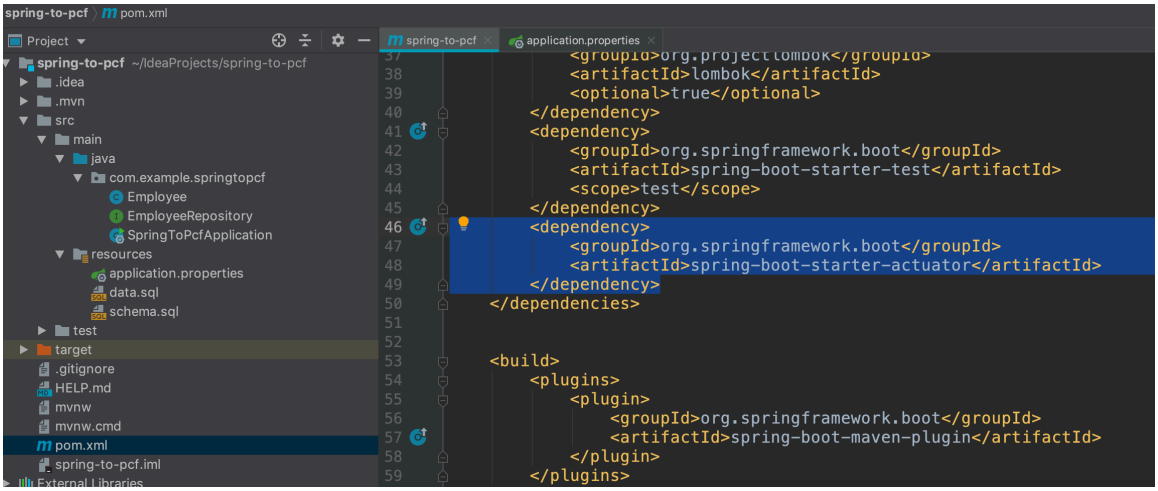
Pivotal

2. Add monitoring to your microservice

1. Spring Boot Actuator is a sub-project of Spring Boot . It adds several Spring Boot's production-ready features to your application with little effort on your part.
   You can find out more at
   https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready

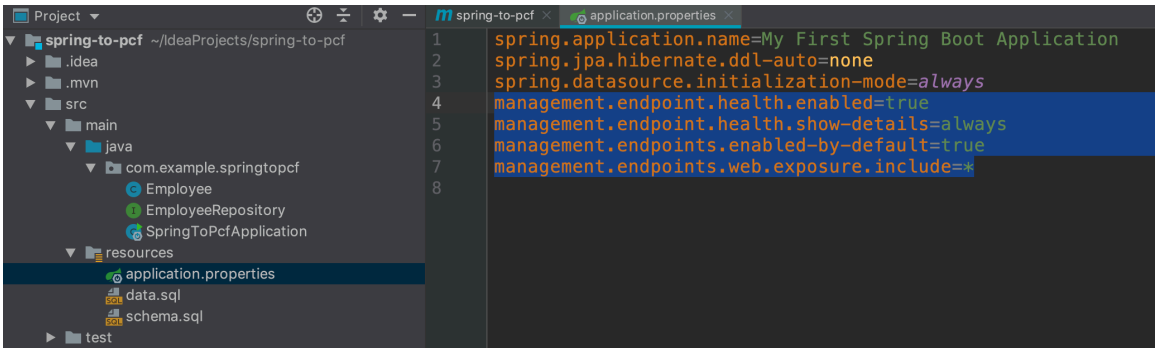
   Now lets add the following dependency to the pom.xml of your microservice application.

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```



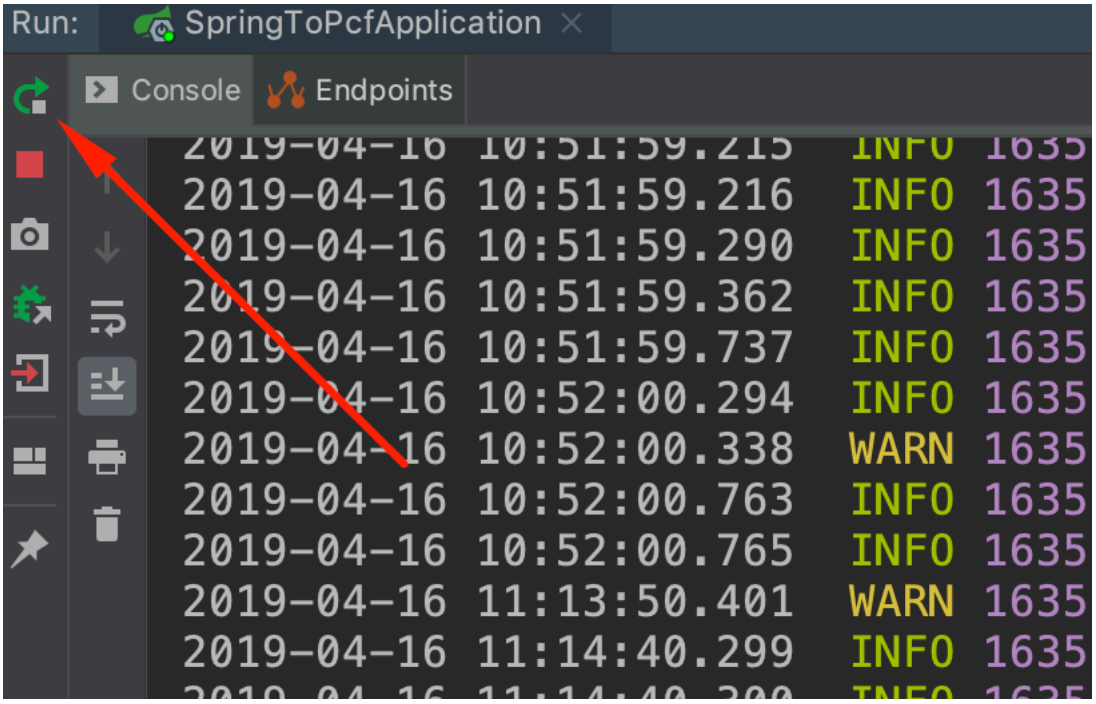
2. We will also add some properties to enable the features. Please add the following to the application.properties:

```
management.endpoint.health.enabled=true
management.endpoint.health.show-details=always
management.endpoints.enabled-by-default=true
management.endpoints.web.exposure.include=*
```



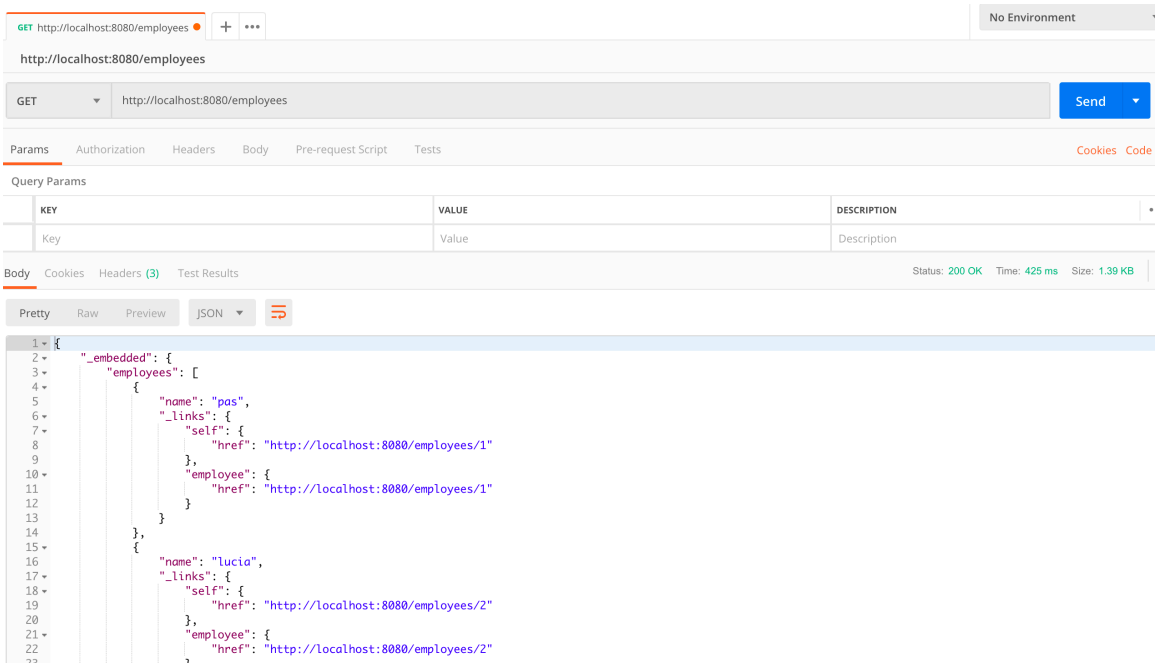
Pivotal

3. Make sure the changes are saved and proceed to restart the application.



4. Ensure your employee microservice is still running. Execute the following HTTP request. You should see the same output.

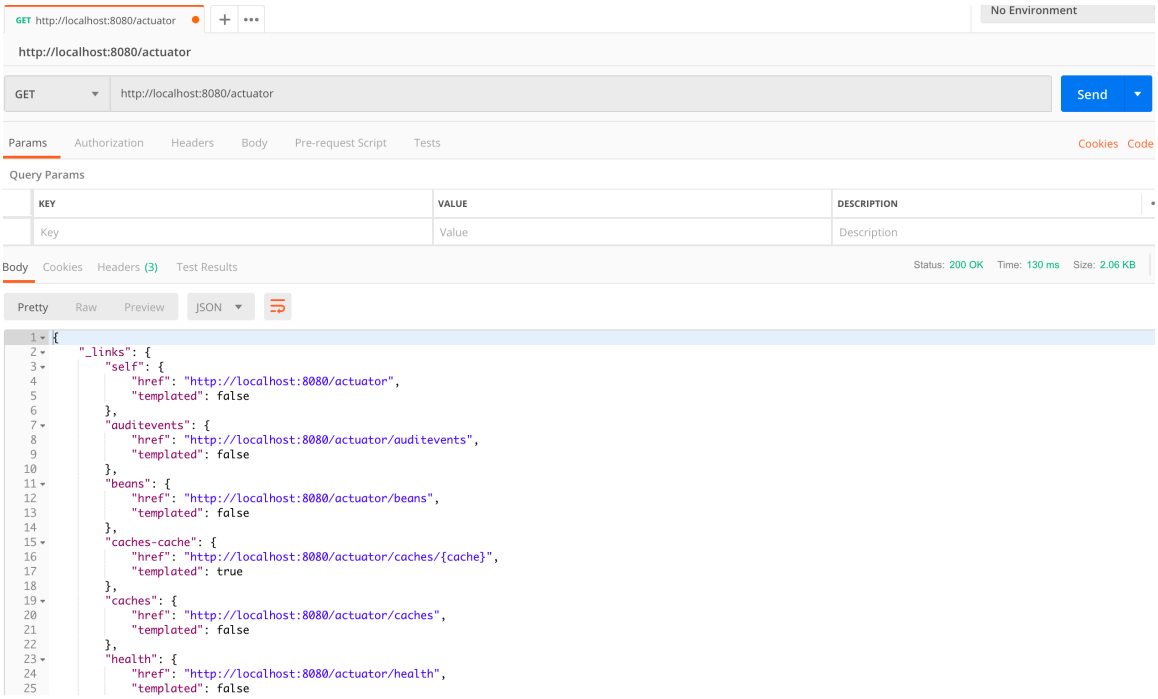
GET http://localhost:8080/employees



Pivotal

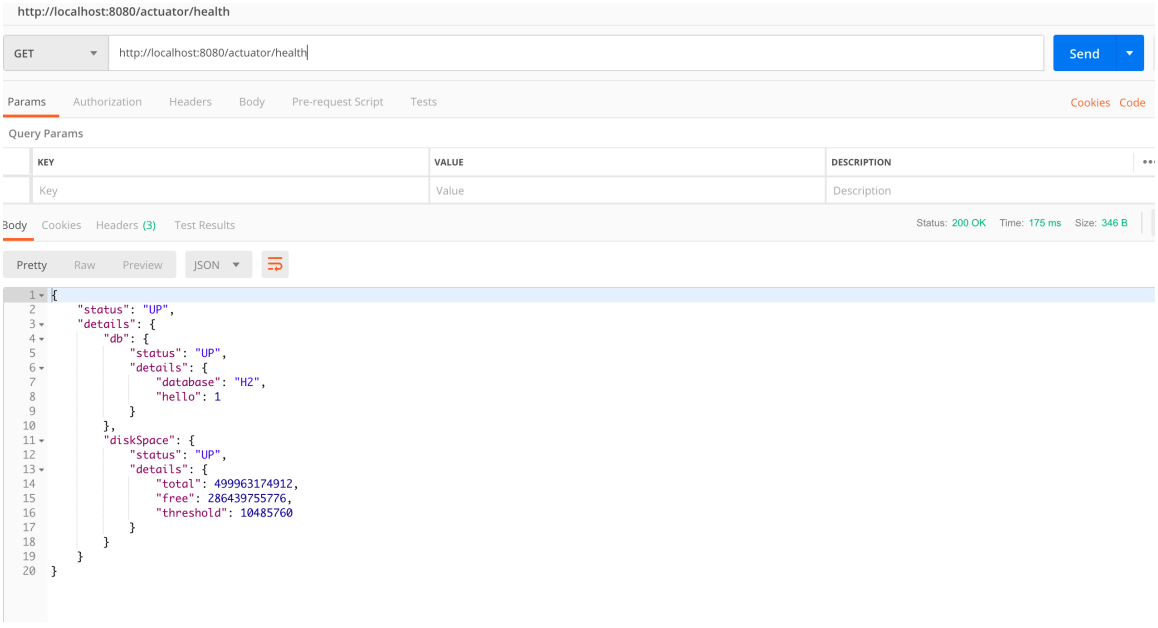
5. Next execute the following HTTP request. You should get a list of actuator endpoints.

GET http://localhost:8080/actuator



6. Try visiting the different health endpoint. You will see that the h2 database is in use.

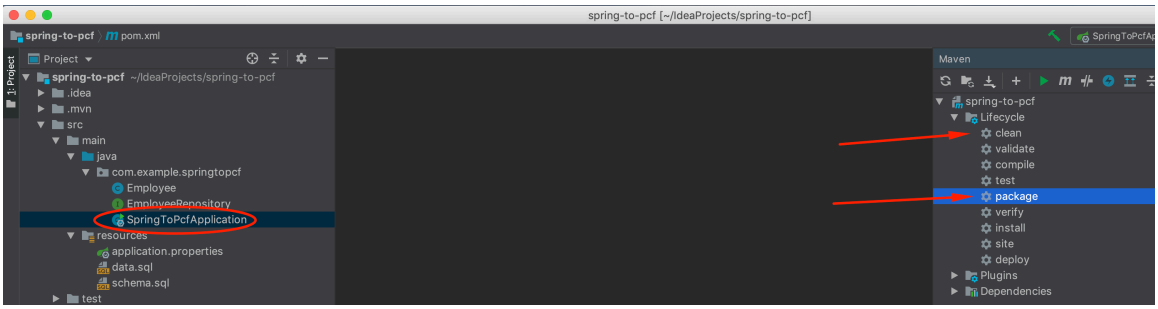
GET http://localhost:8080/actuator/health



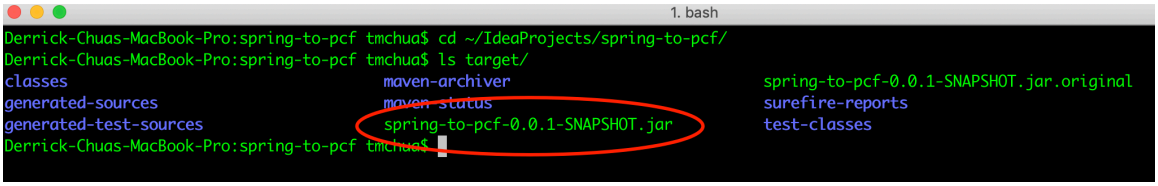
Pivotal

3. Deploying and running on Pivotal Application Service

1. Perform a maven clean and package of your microservice from the IDE. You can do so by double-clicking on each of the maven stages separately.

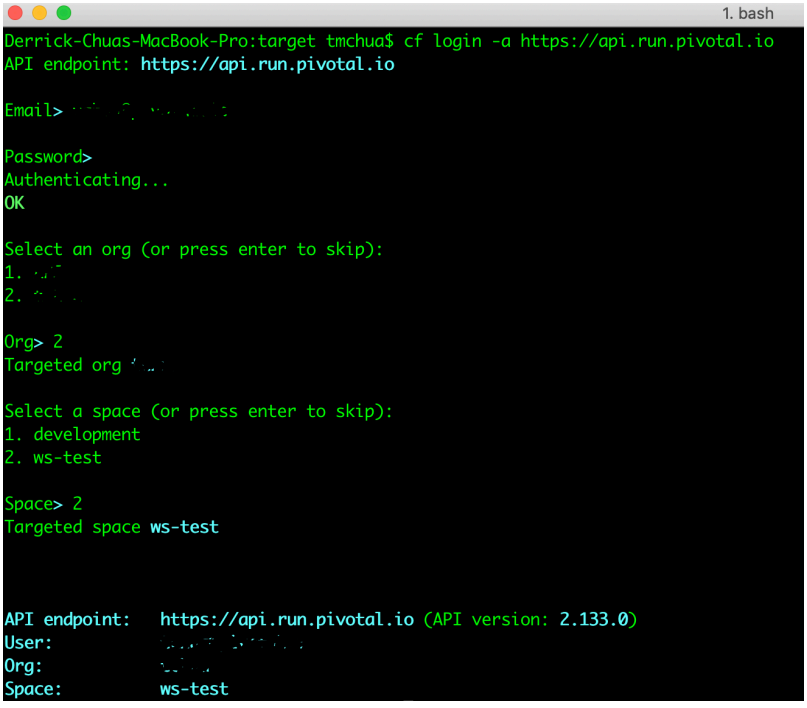


2. Open up a terminal and 'cd' to the project directory. Perform a 'ls' on the target directory and you should see the jar file.



3. Using the cf cli, log in to your PWS account. The command to do so is

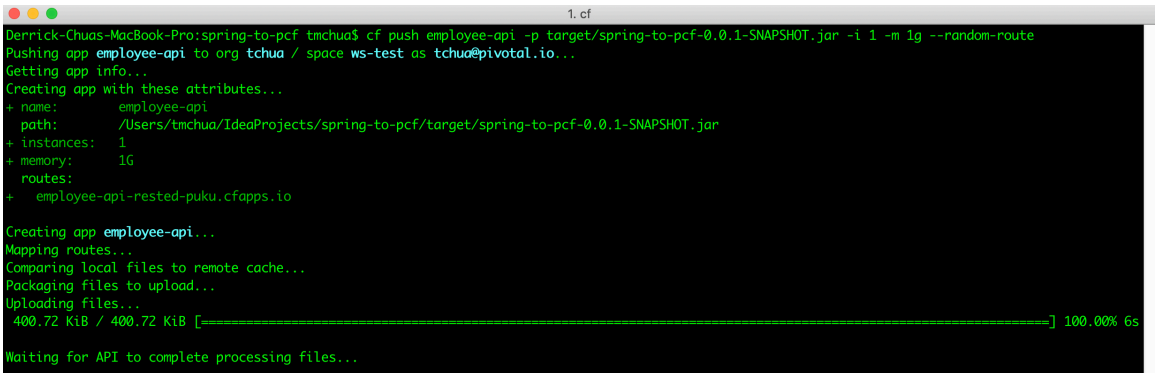
cf login -a https://api.run.pivotal.io



Pivotal

4. Deploy and run the app on PWS using the following command:

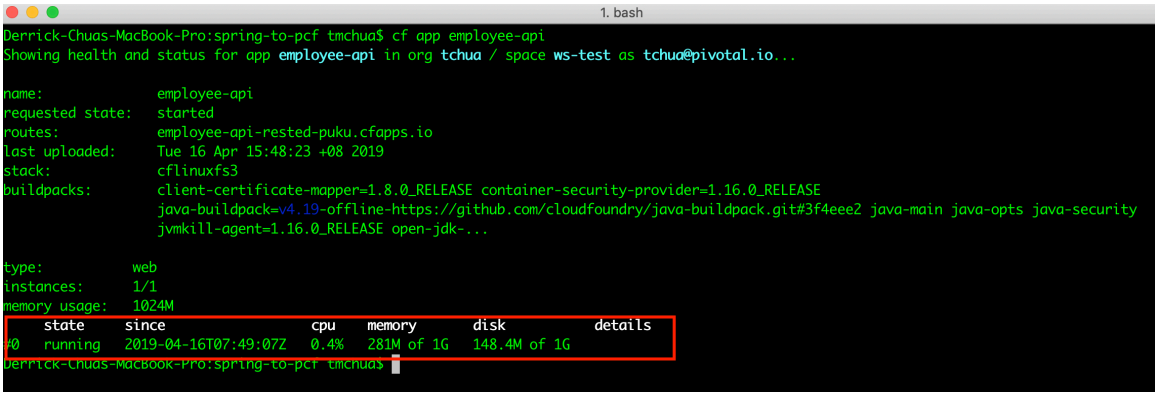
cf push employee-api -p target/spring-to-pcf-0.0.1-SNAPSHOT.jar -i 1 -m 1g --random-route



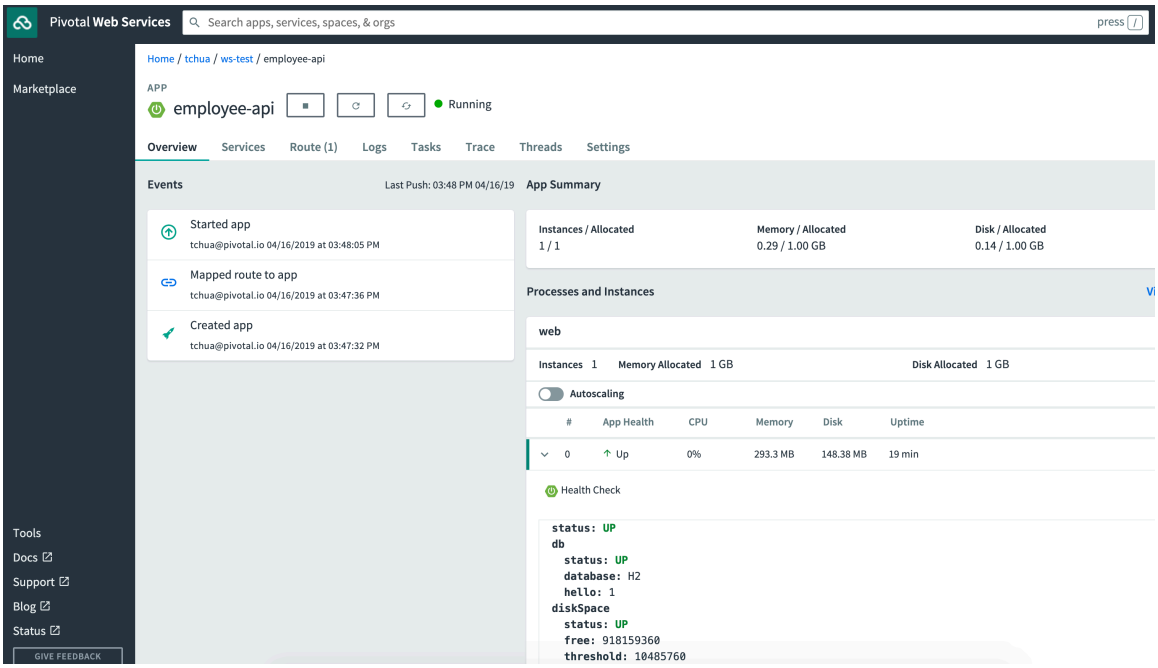
5. How do you now know the status of your app on the platform? You can do so through either the cf cli or through the PWS web console.

On the terminal, execute the following cf command:

cf app employee-api



On the browser, navigate to https://console.run.pivotal.io and log in if you have to. Click into your org and space, and then select the app. You should see the following. You can also observe that the actuator health endpoint has been integrated into the console.

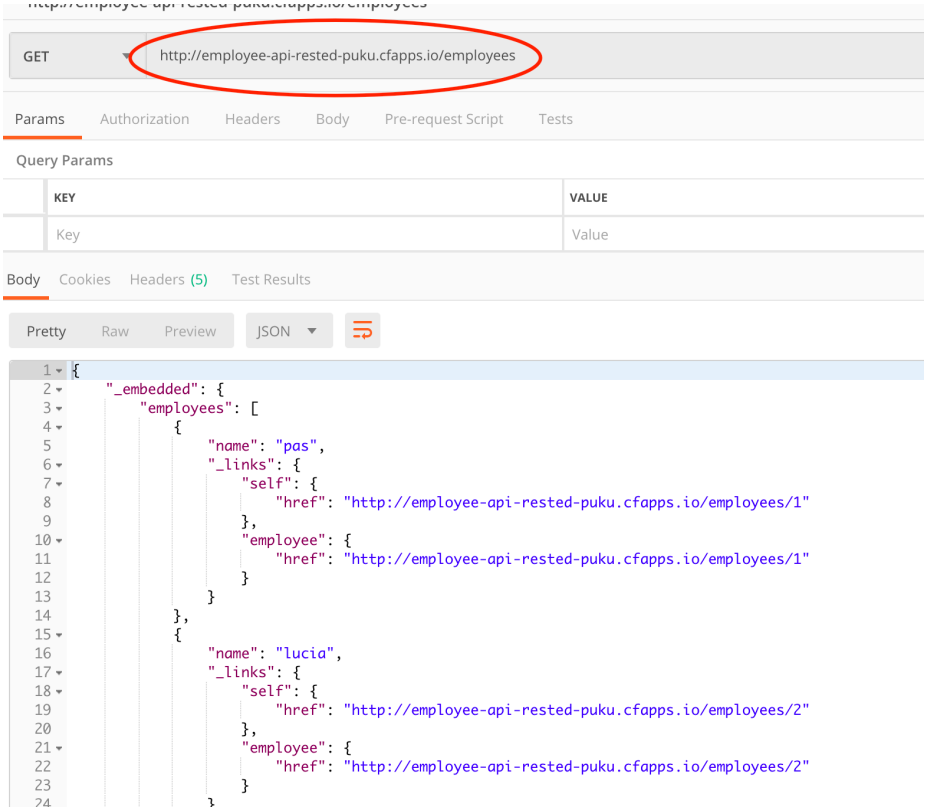


6. From the status of ‘cf app’ command, you can see the route. This route forms the domain name part of your employee API URL. Now you can query your employee API on

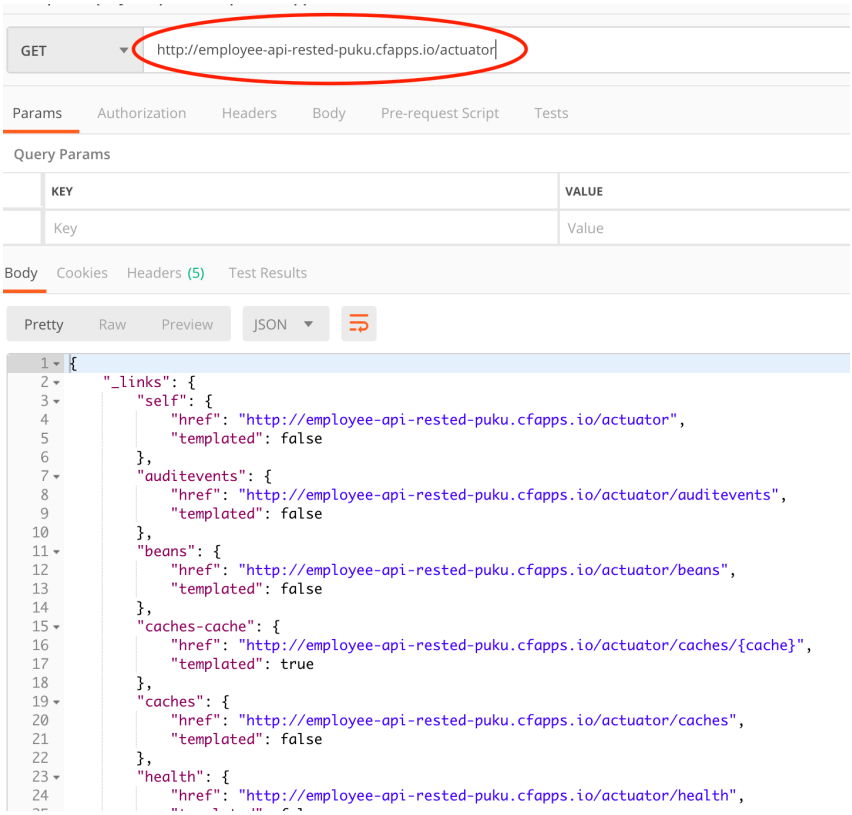
   http://{route}/employees

   You can try this on Postman.

```
Derrick-Chuas-MacBook-Pro:spring-to-pcf tmchua$ cf app employee-api
Showing health and status for app employee-api in org tchua / space ws-test as tchua@pivotal.io...

name:              employee-api
requested state:   started
routes:            employee-api-rested-puku.cfapps.io
last uploaded:     Tue 16 Apr 15:48:23 +08 2019
stack:             cflinuxfs3
buildpacks:        client-certificate-mapper=1.8.0_RELEASE container-security-provider=1.16.0_RELEASE
                   java-buildpack=v4.19-offline-https://github.com/cloudfoundry/java-buildpack.git#3f4eee2 java-main java-opts java-security
                   jvmkill-agent=1.16.0_RELEASE open-jdk-...

type:           web
instances:      1/1
memory usage:   1024M
     state     since                  cpu    memory       disk          details
#0   running   2019-04-16T07:49:07Z   0.4%   281M of 1G   148.4M of 1G
Derrick-Chuas-MacBook-Pro:spring-to-pcf tmchua$
```



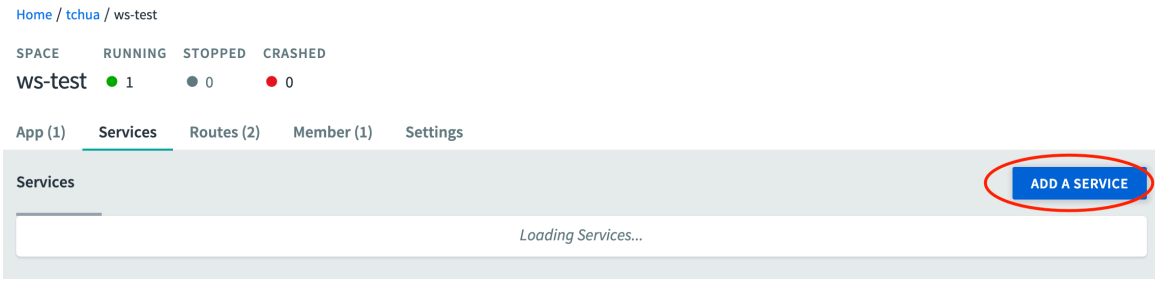
7. You can also access the actuator endpoints the same way.



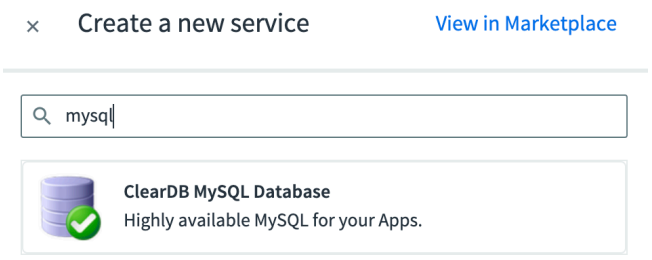
Pivotal

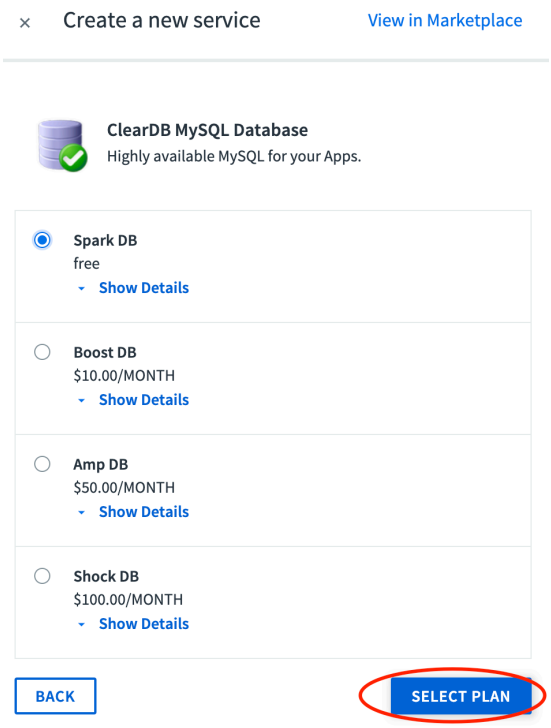
4. Creating and binding a service on demand

1. In your PWS console, go to the “Services” tab. Click on “Add A Service” button.

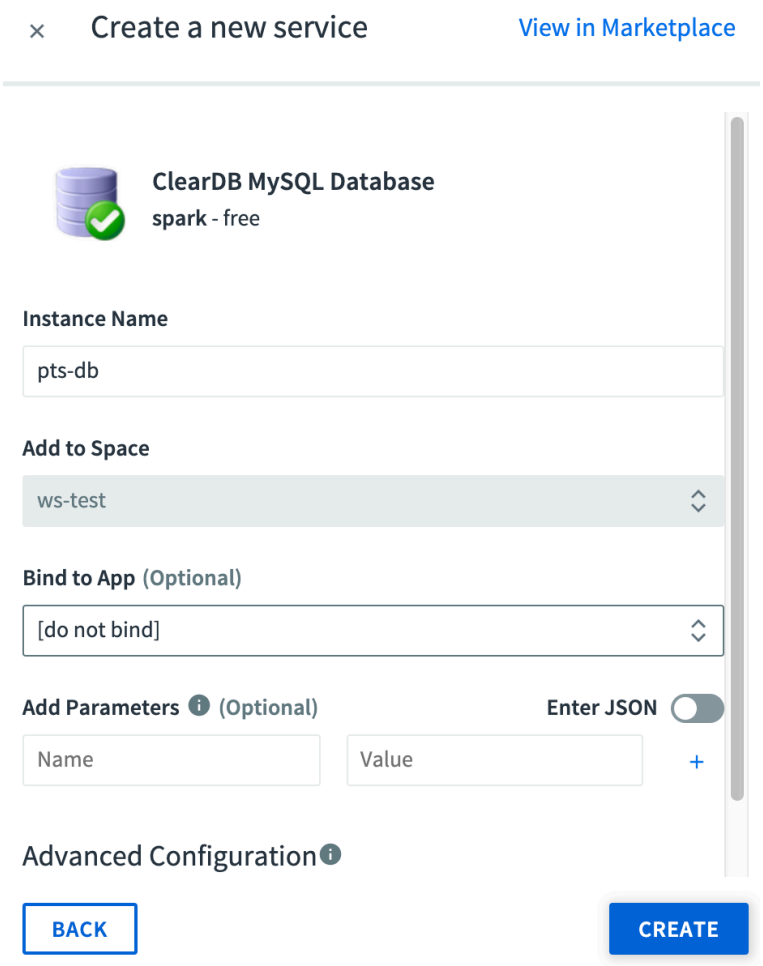


2. In the panel that appears, search for ‘mysql’ and ClearDB should appear. Click on ClearDB, select the free Spark DB plan and then click “Select Plan”.



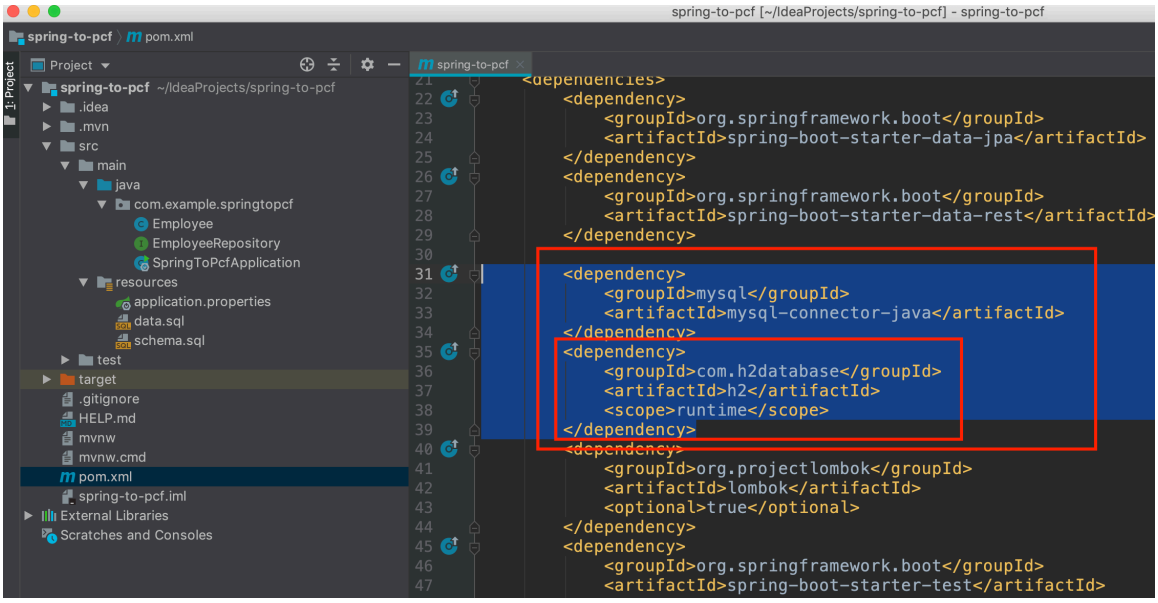


3. Enter the value “pts-db” as the Instance Name. Ensure “Bind to App” field is set to “[do not bind]”. Proceed to click “Create”. Wait for a few minutes for the service to be created.



4. With the MySQL instance created, now we need to may changes to the Employee API microservice to make use of MySQL, instead of h2. This will involve just a change in the pom.xml.
Add the following XML snippet to pom.xml, making sure that it is before the h2 dependency.

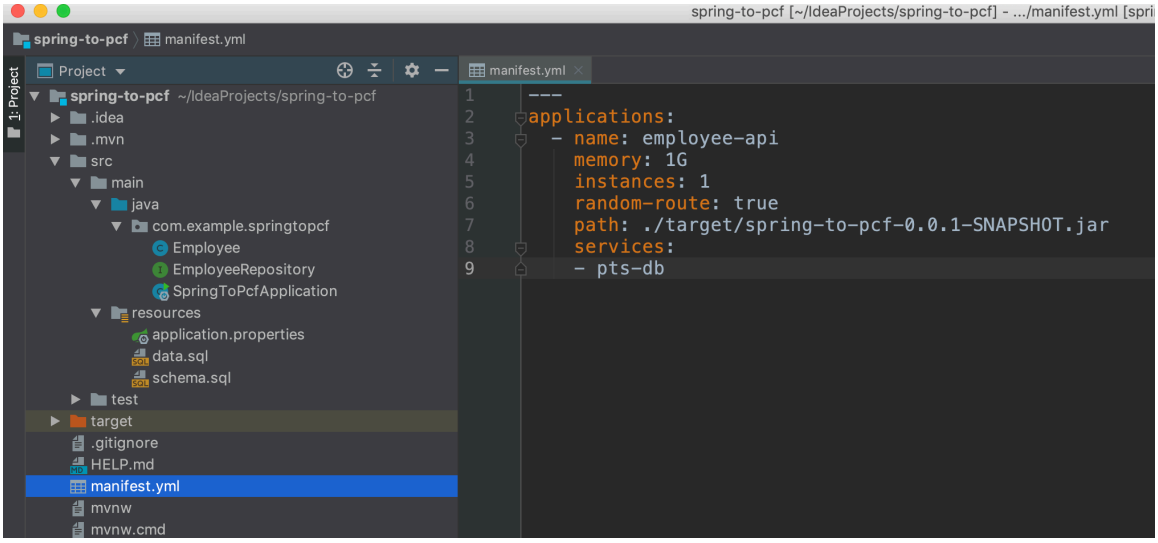
```
<dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
</dependency>
```



5. Add a new file named “manifest.yml” to the project root. Include the following content in the file.

```
---
applications:
 - name: employee-api
   memory: 1G
   instances: 1
   random-route: true
   path: ./target/spring-to-pcf-0.0.1-SNAPSHOT.jar
   services:
   - pts-db
```

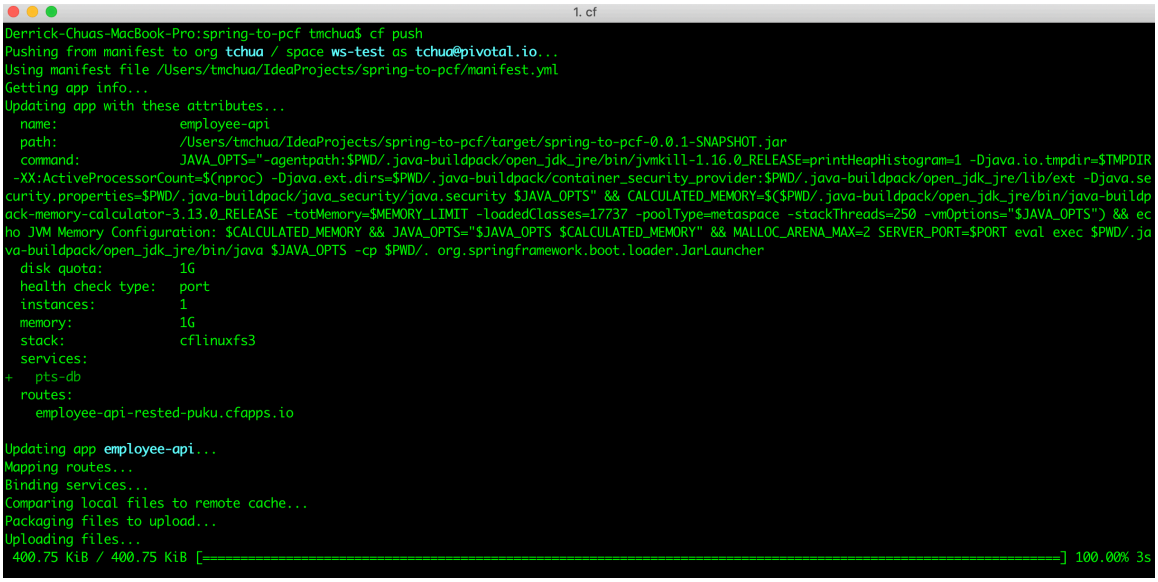
Instead of specifying arguments to the ‘cf push’ command, running ‘cf push’ without any argument will cause it to look for manifest.yml in the current working directory for parameters. Using a manifest will allow application specific parameters to be source controlled together with the application in git. It also does away with the need to configure and update CI/CD pipelines.



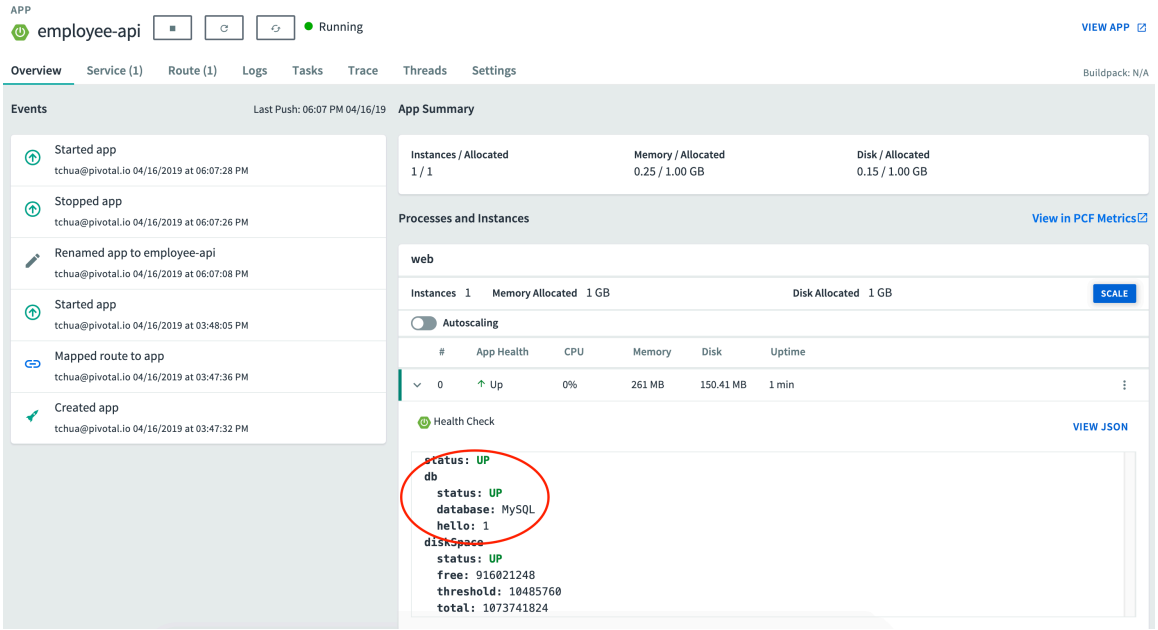
6. In the maven panel, double click on clean and package to create a new clean copy of the jar file. Switch over to the terminal and execute the command

   cf push

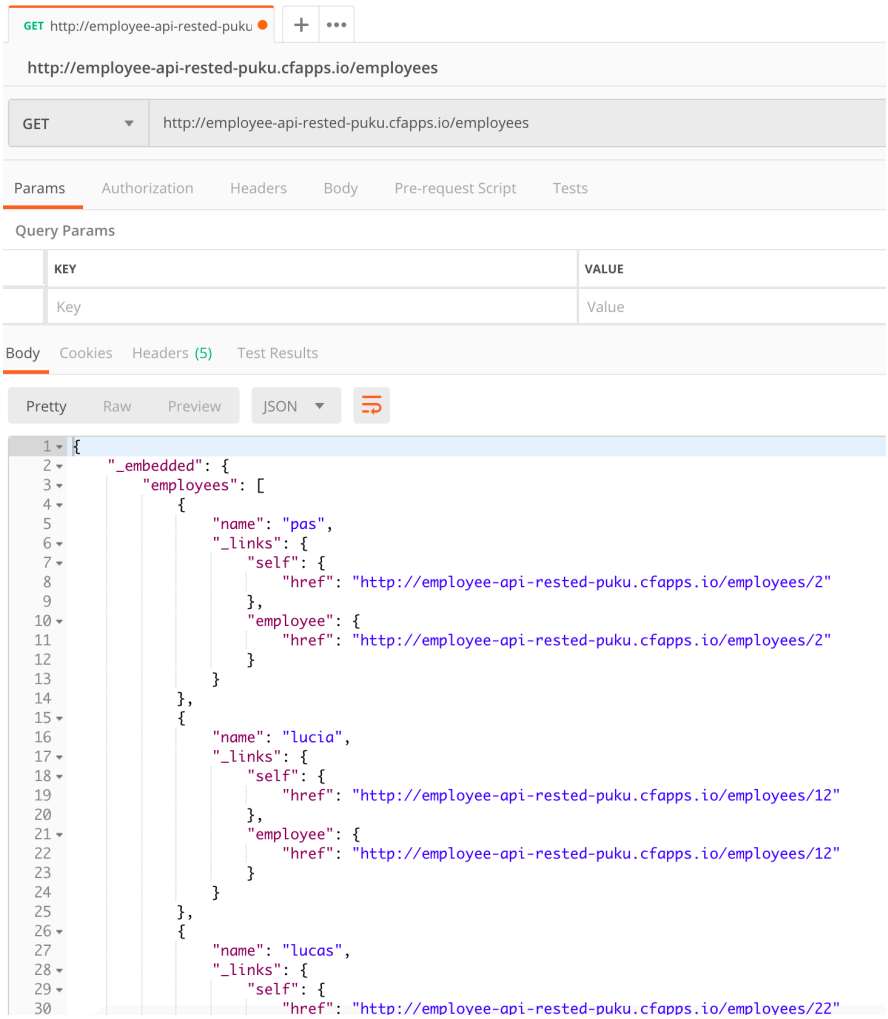
   You will see in the console log that the manifest file is being used to read in the app attributes



7. Switch over to the console to look at the actuator based health status of the app, and you will see that it is now using MySQL. You might also want to verify that the Employee API is working correctly using Postman.



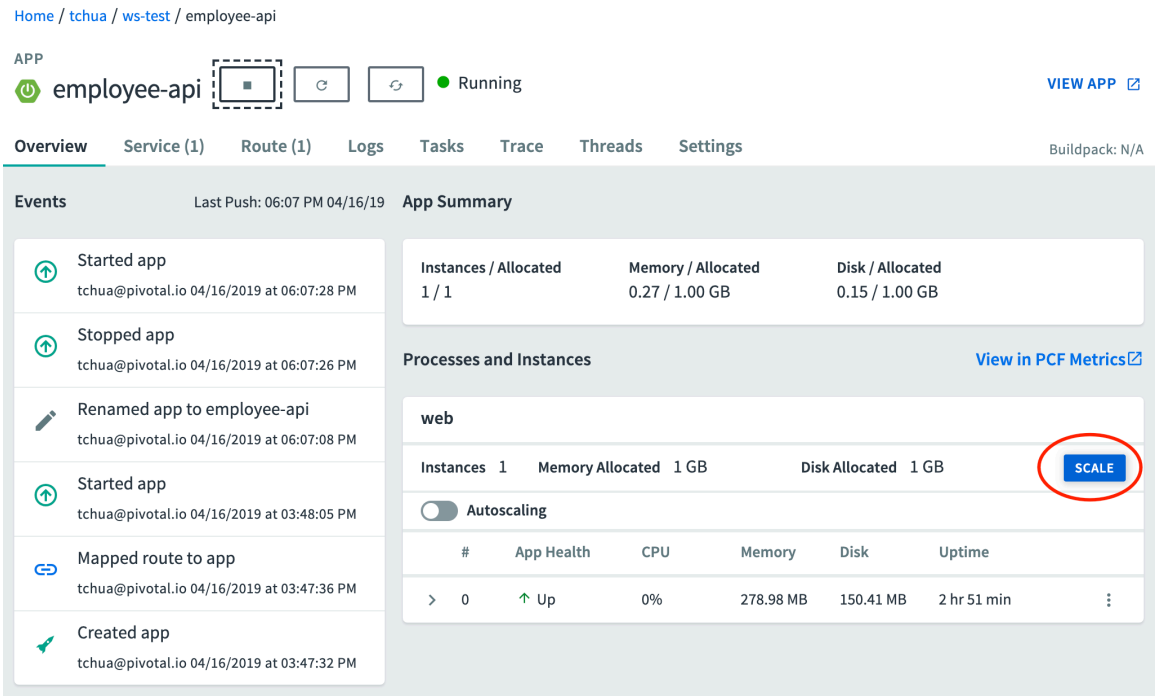
Pivotal


GET http://employee-api-rested-puku
http://employee-api-rested-puku.cfapps.io/employees
GET
http://employee-api-rested-puku.cfapps.io/employees
Params
Authorization
Headers
Body
Pre-request Script
Tests
Query Params
KEY
VALUE
Key
Value
Body
Cookies
Headers (5)
Test Results
Pretty
Raw
Preview
JSON
{
"_embedded": {
"employees": [
{
"name": "pas",
"_links": {
"self": {
"href": "http://employee-api-rested-puku.cfapps.io/employees/2"
},
"employee": {
"href": "http://employee-api-rested-puku.cfapps.io/employees/2"
}
}
},
{
"name": "lucia",
"_links": {
"self": {
"href": "http://employee-api-rested-puku.cfapps.io/employees/12"
},
"employee": {
"href": "http://employee-api-rested-puku.cfapps.io/employees/12"
}
}
},
{
"name": "lucas",
"_links": {
"self": {
"href": "http://employee-api-rested-puku.cfapps.io/employees/22"

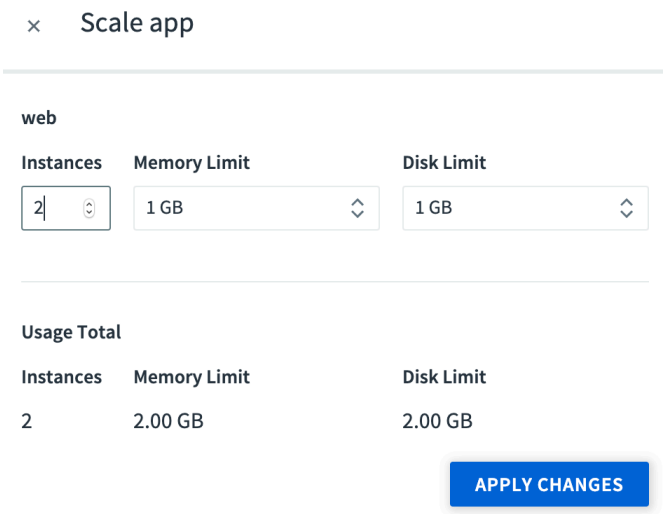
Pivotal

5. Scaling your app

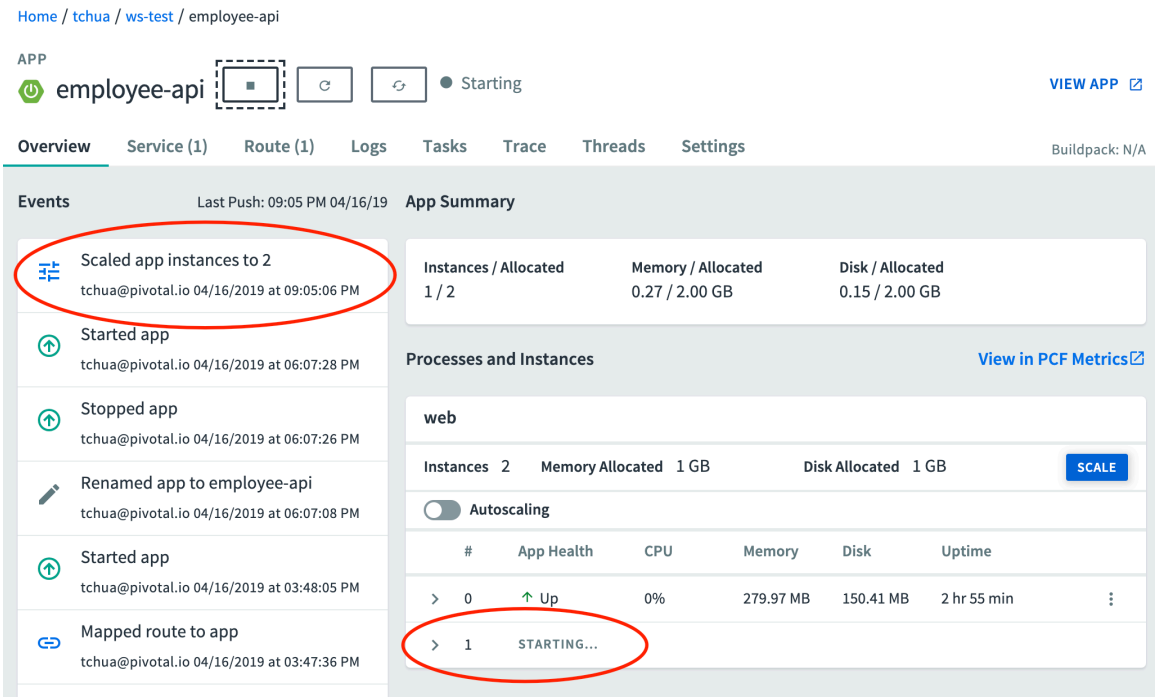
1. Navigate to the PWS console and go to your employee-api app. Click on the “Scale” button as shown.



2. Increase the number of instances to 2 and click on “Apply Changes”



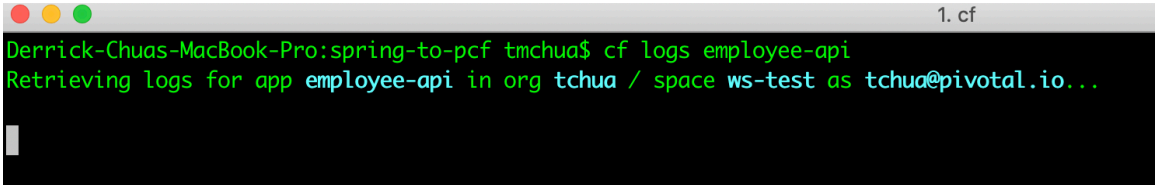
3. Observe that action has been taken to scale the app instances to 2.



4. Switch over to your terminal and execute the following command:

cf logs employee-api

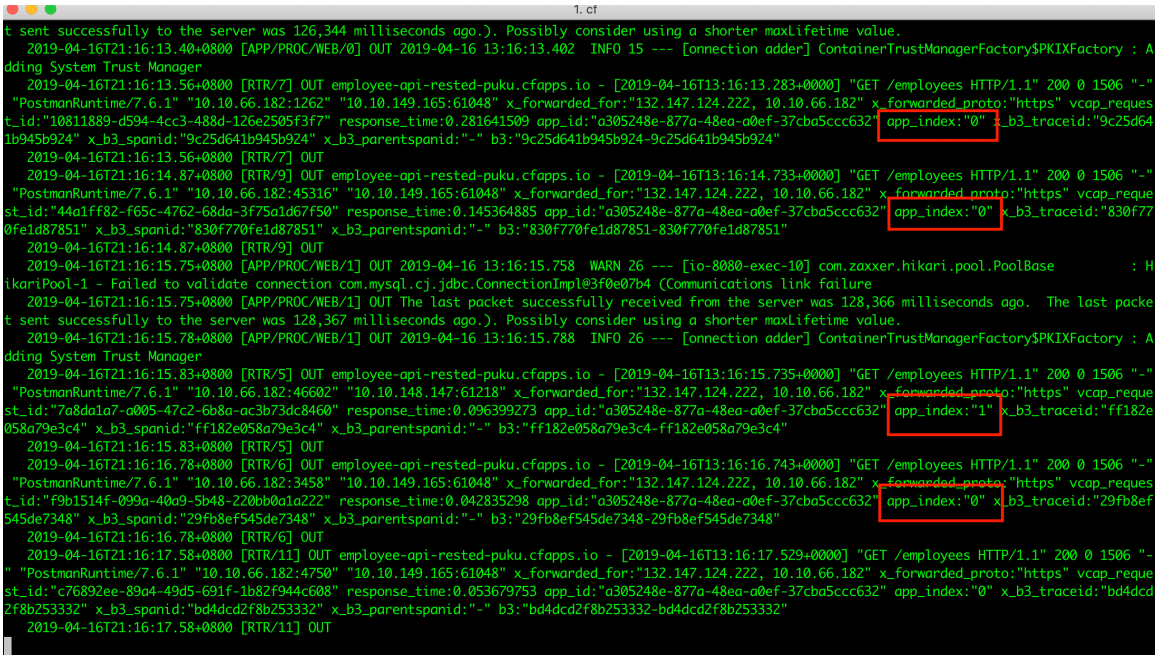
This will activate the log reading functionality in tail mode



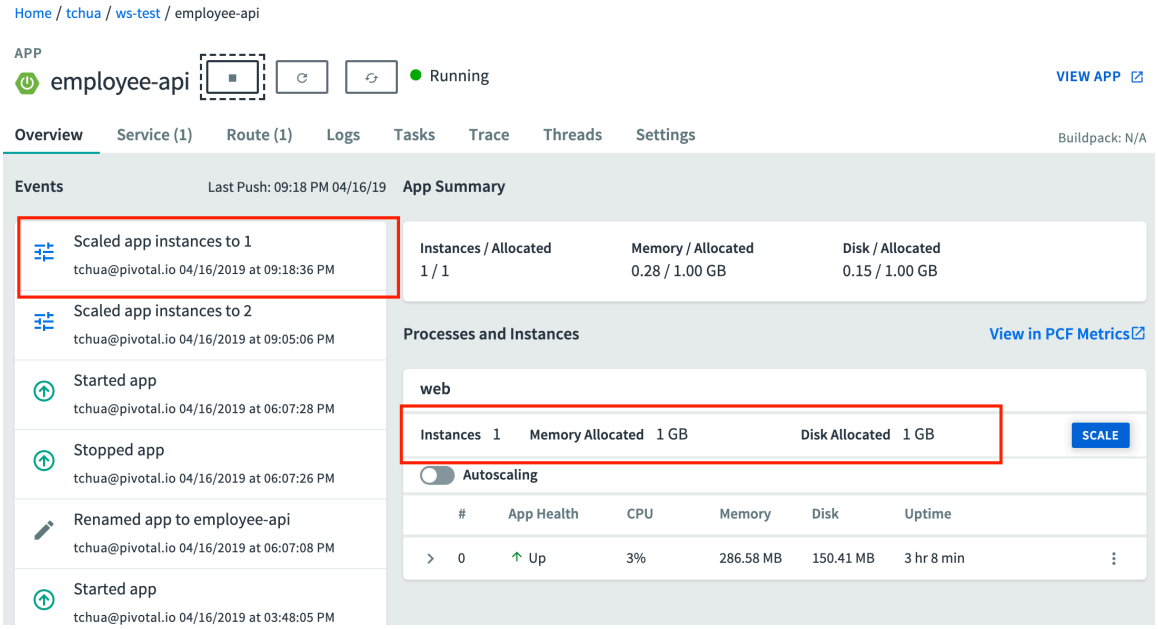
5. Using Postman, execute this HTTP request multiple times to generate some logs:

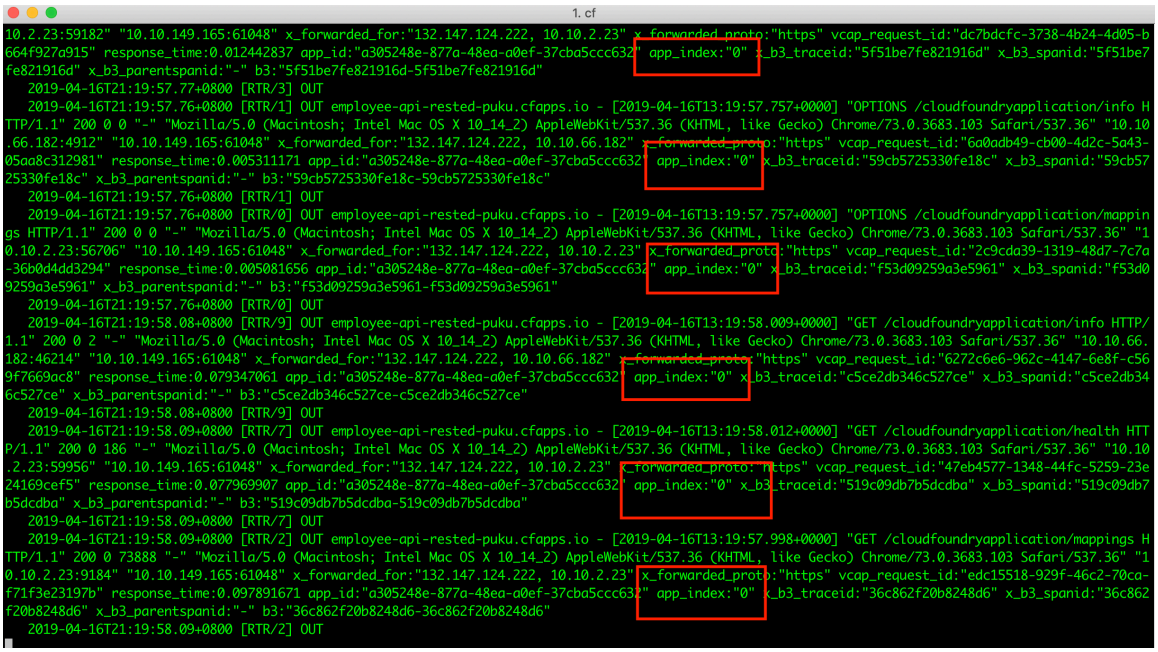
http://{route}/employees

Observe the log output at the terminal, looking specifically for a json property called “app_index”. You should values “0” and “1” interspersed.

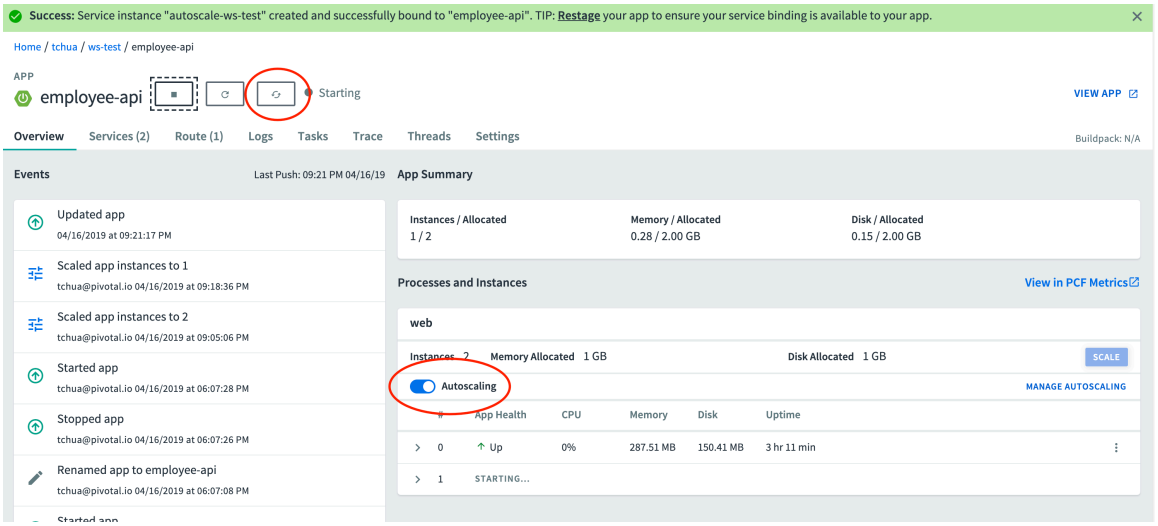


6. Reduce the number of instances back to 1, execute the GET request a few times and observe the log output. You should find that only instance 0 is handling the requests.

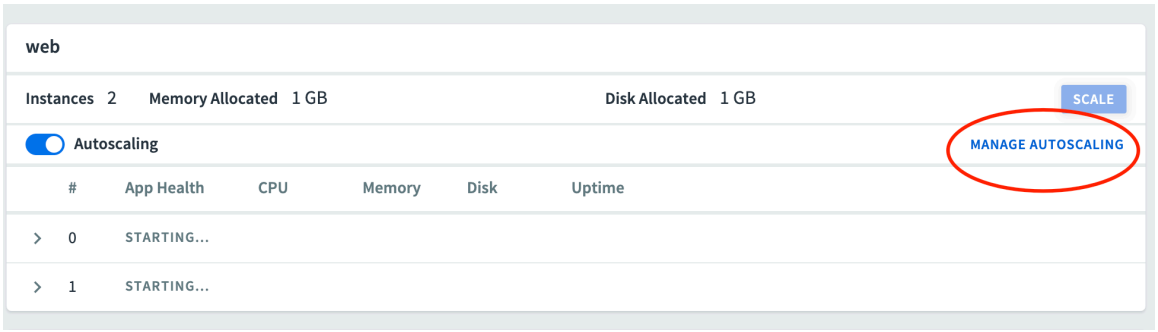




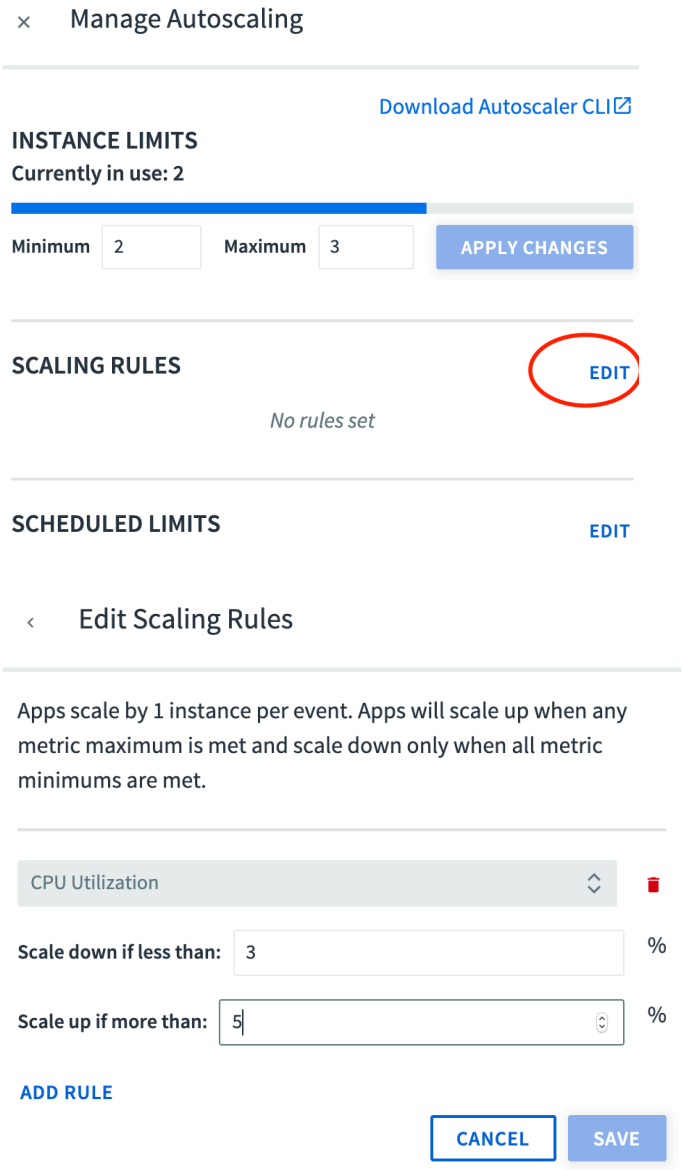
7. Enable “Autoscaling” and then restage your app using the restage button.



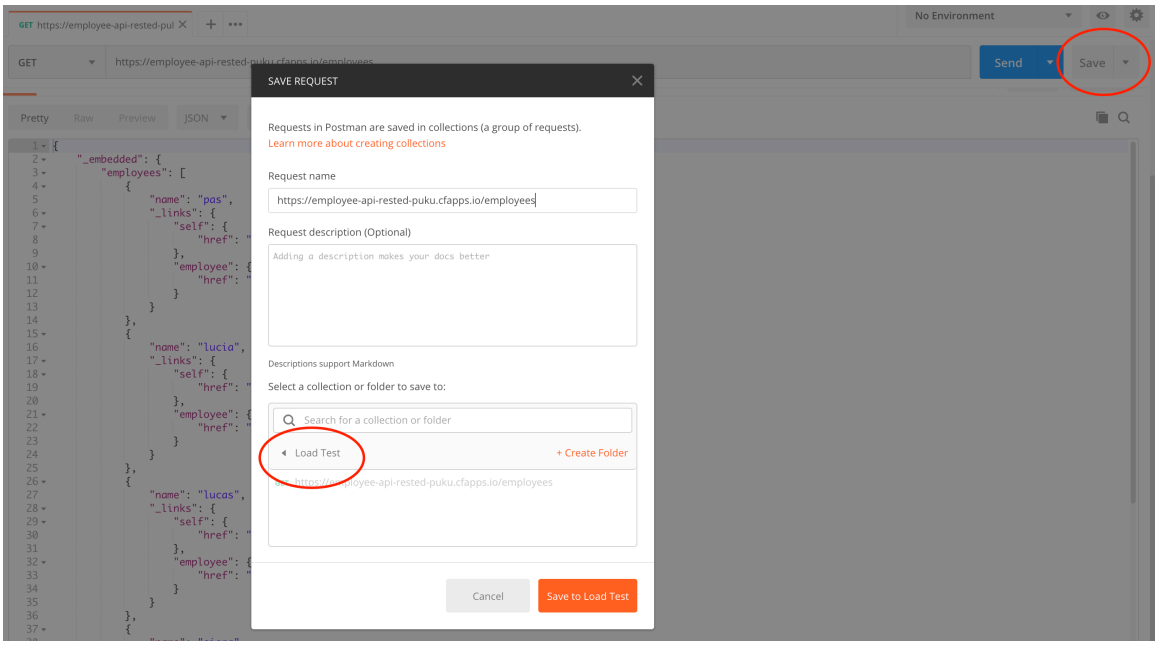
8. Click on “Manage Autoscaling”



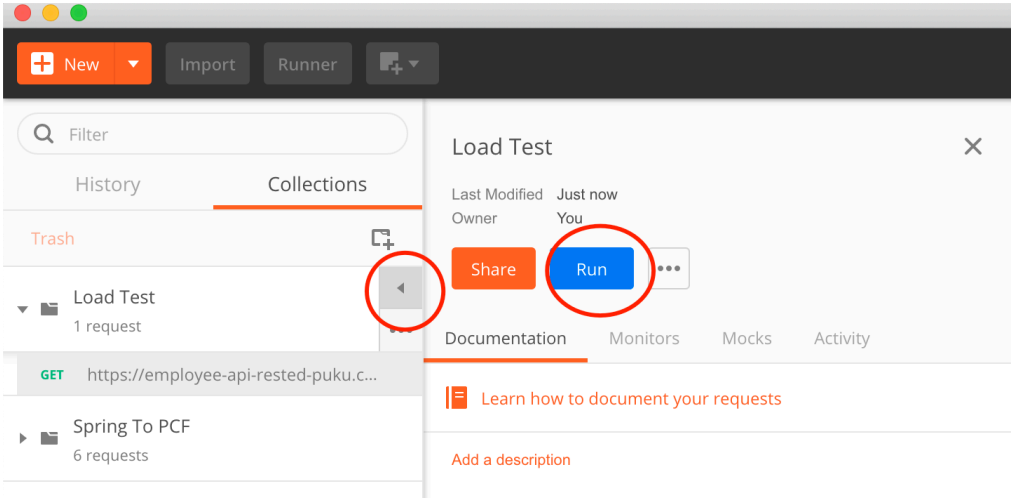
9. Add a scaling rule, starting by clicking on the “Edit” link. Add a CPU utilization rule as shown below. Note that the numbers used are for demo purposes only. In actual deployments, more sensible numbers should be used.



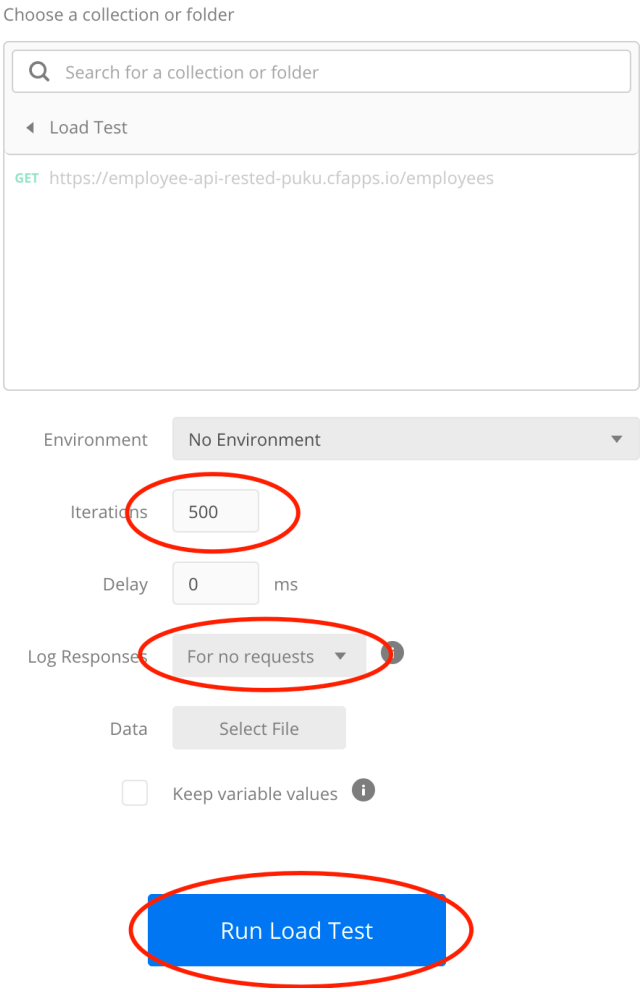
10. In Postman, save your employees request to a new collection called “Load Test’.



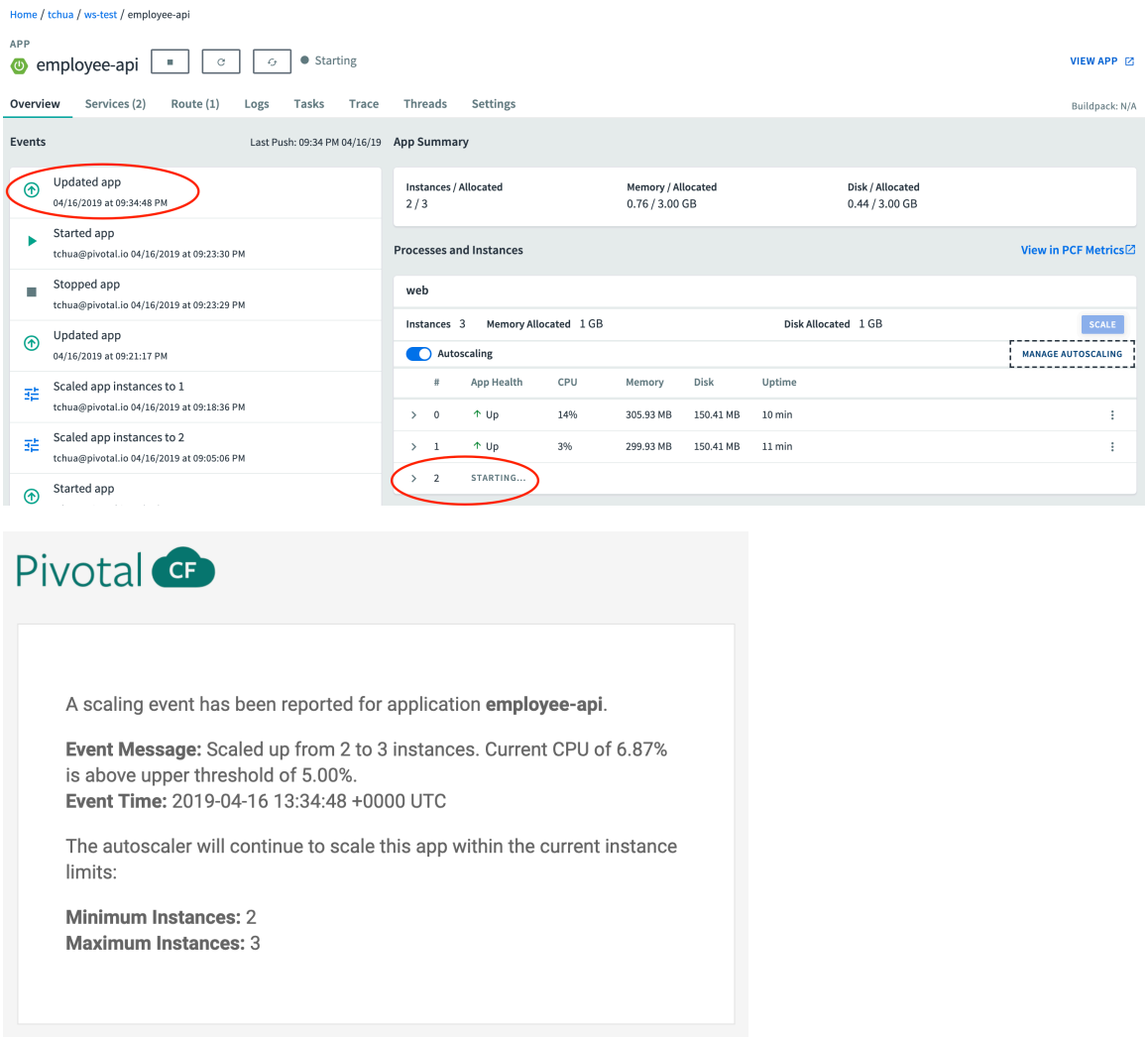
11. On the left panel, expan the arrow next to “Load Test” collection and click on the “Run” button.



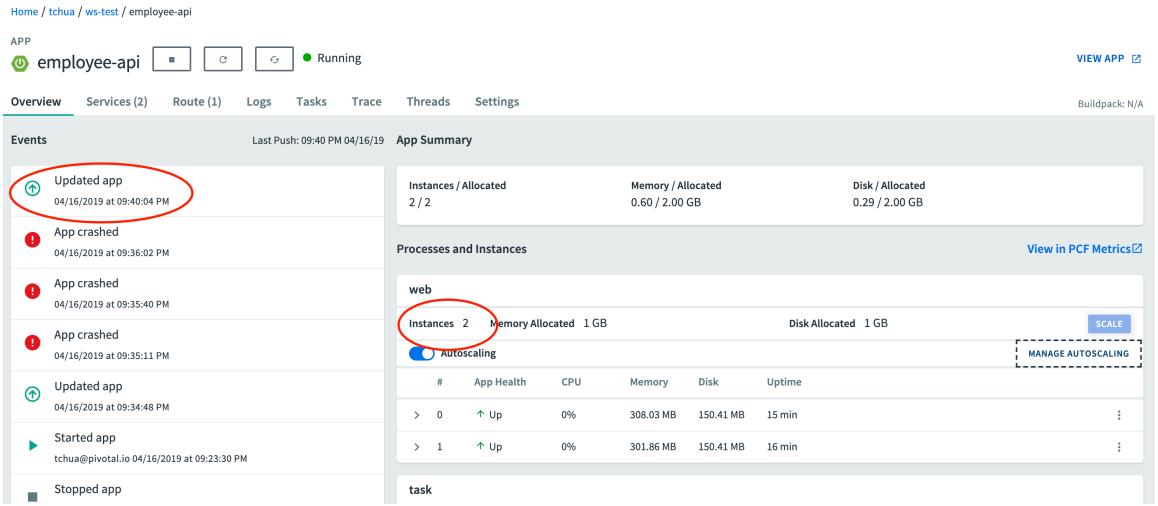
12. In the window that pops up, set the number of Iterations to 500, the Log Responses to “For no requests” and then hit the “Run Load Test” button.



13. Switch back to the PWS console and observe your app. You should see the autoscaling kick in and spin up the third instance. You will also receive a notification in your email inbox.
Note: You might find the app crashing after some time. This is normal for this test. The app crashes due to the connections to the database running out because we are running on the free plan.



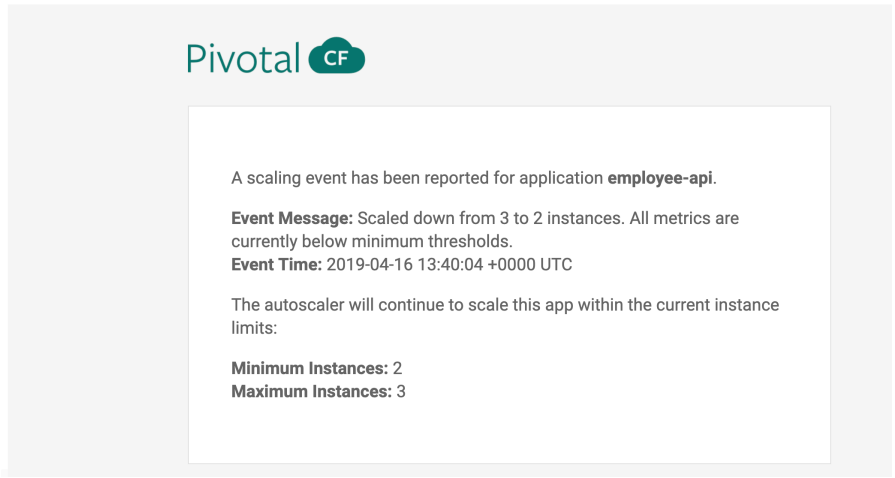
14. After the tests completes, observe a while and you should see the number of instances dropping back to 2. An email notification will also be sent to you.



Pivotal

CF Notification: Scaling Down Inbox ×

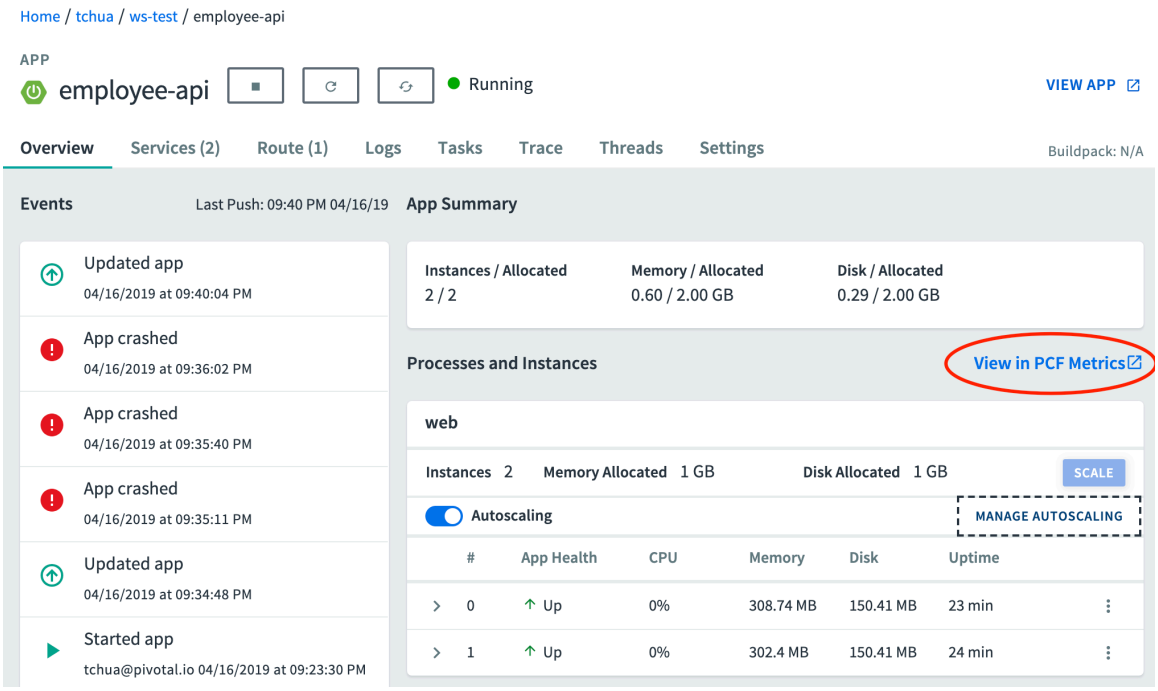
no-reply@run.pivotal.io 9:40 PM (50 minutes ago)
to me



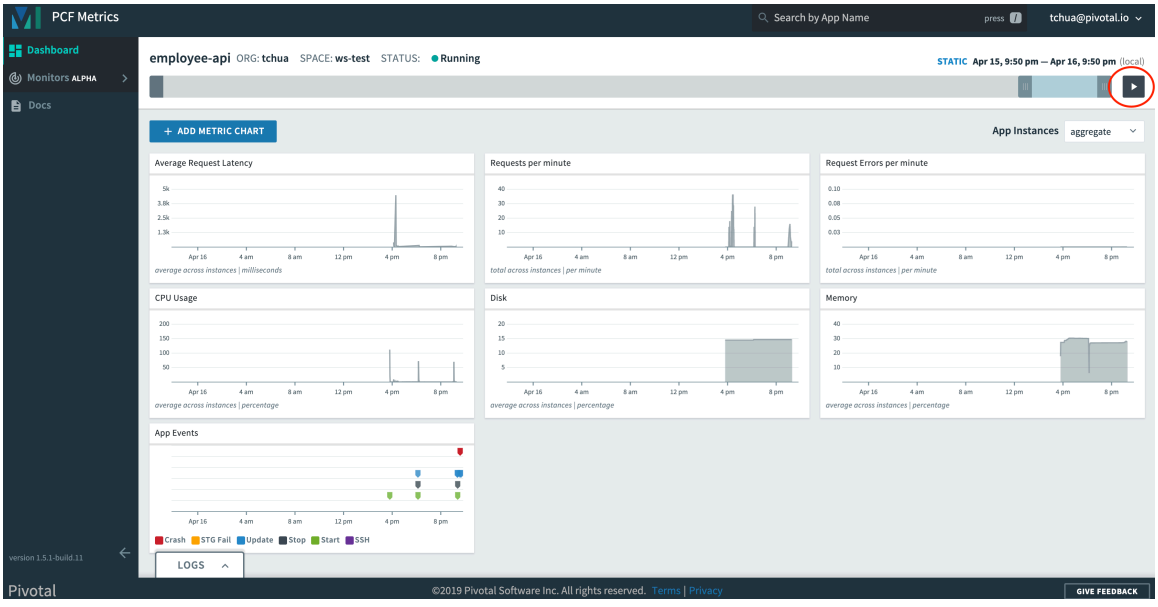
Pivotal

6. Monitoring and troubleshooting your app

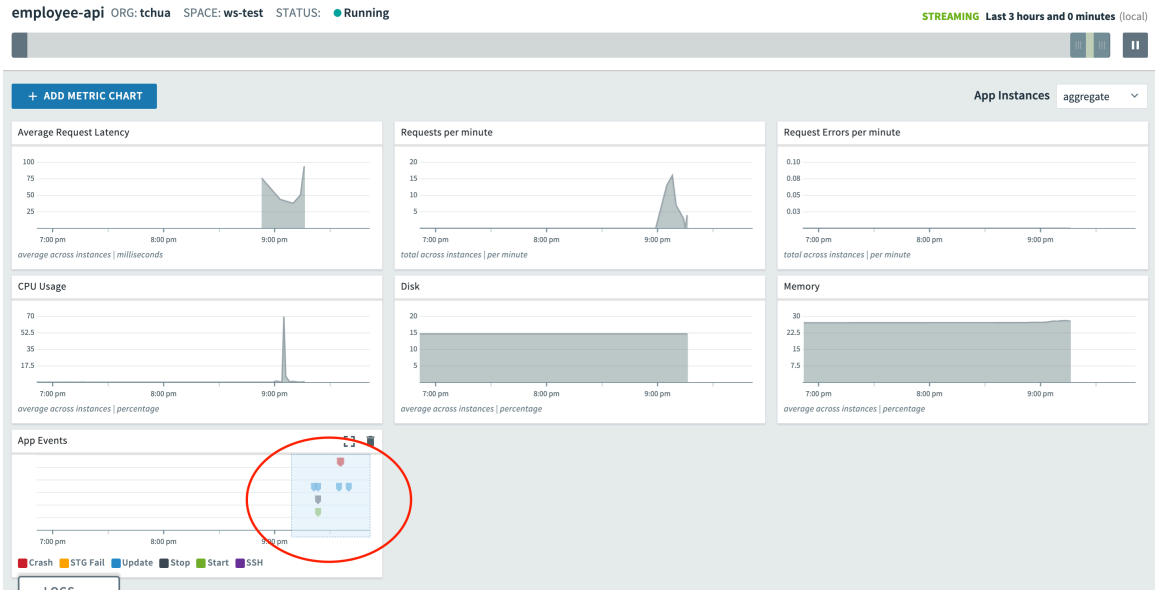
1. At PWS console, go to your app and click “View in PCF Metrics”.



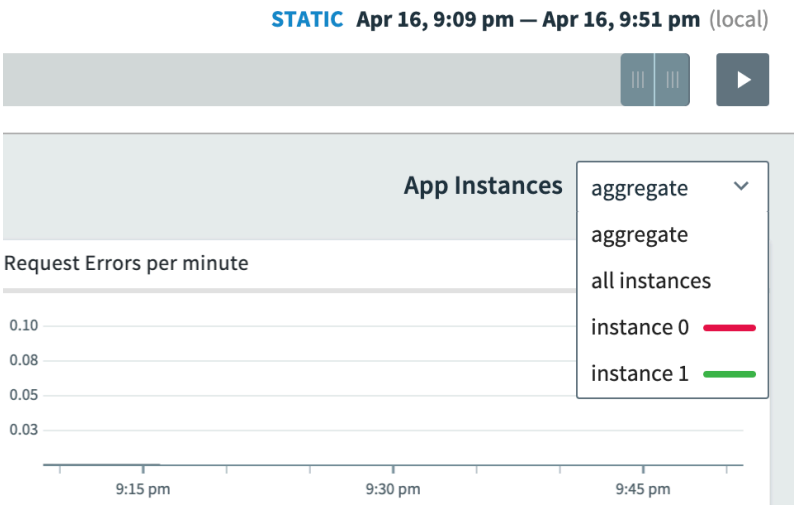
2. A new tab will open up to the PCF Metrics page, showing all the different metrics that are captured per application instance. You can view the metrics at different time periods by sliding and stretching the time bar at the top. You can also click on the play button to the right of the time bar as illustrated. This will put the page into live mode, showing constantly the different performance areas of the app during the last 3 hours.



3. Instead of using the time bar, you can narrow down your focus by clicking and dragging to zoom into a specific time period as illustrated. Doing so will change the time period shown across all metrics chart.

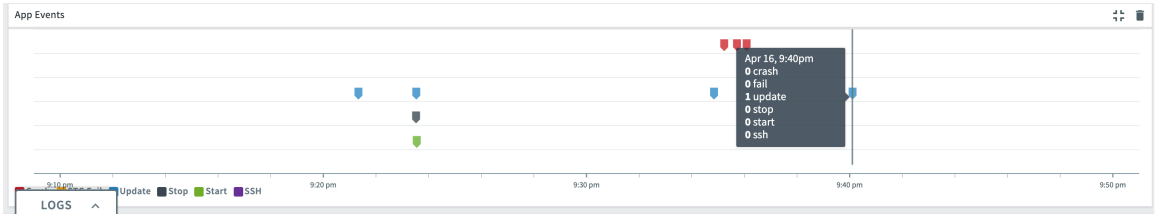


4. You can also select different views by selecting the “App Instances” drop down list as shown.

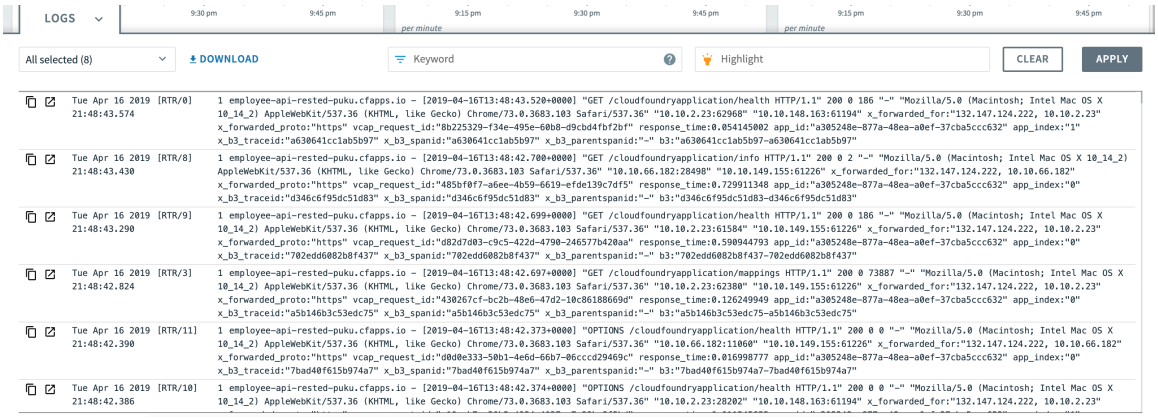


5. You can expand a chart to have more display area.

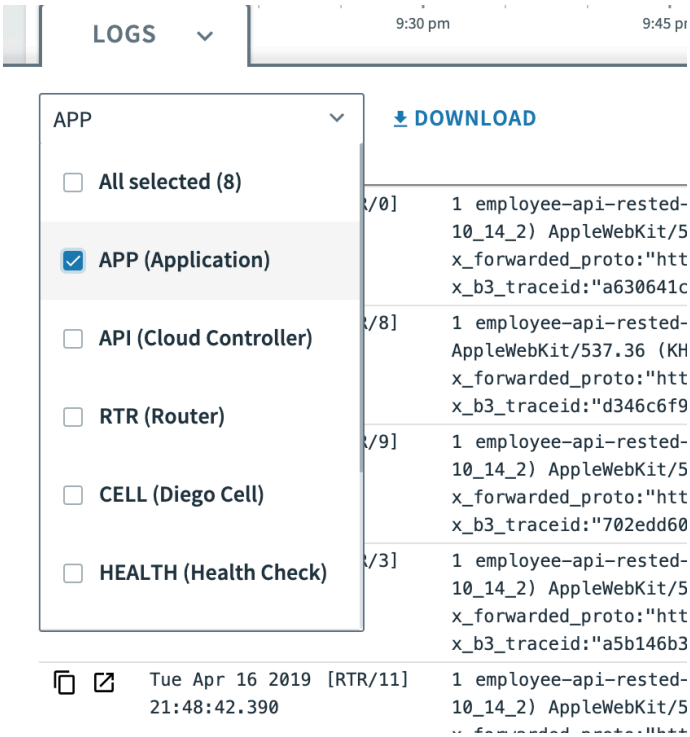
6. You can also mouse over to display relevant details pertaining to a specific metric.



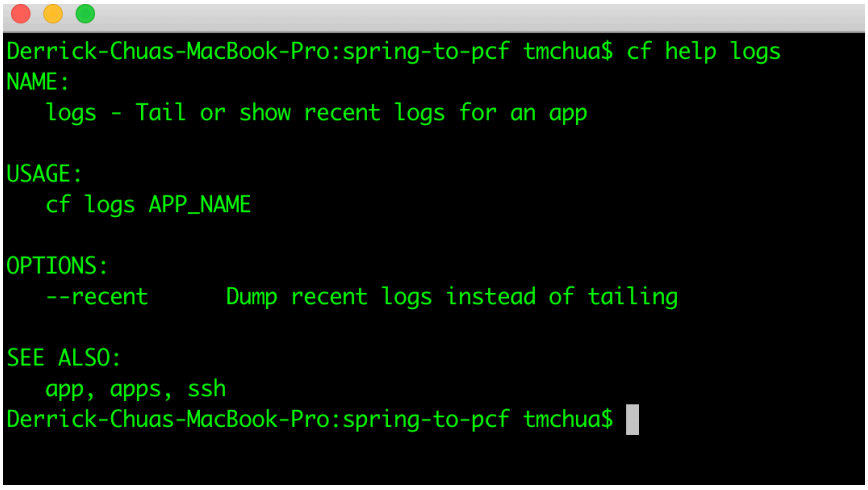
7. You can also view the logs across all instances by clicking on the “Logs” tab at the bottom.



8. Within the “Logs” panel, you can do different filters, such as filtering by log types as shown. This can help filter out the noise to aid your troubleshooting.

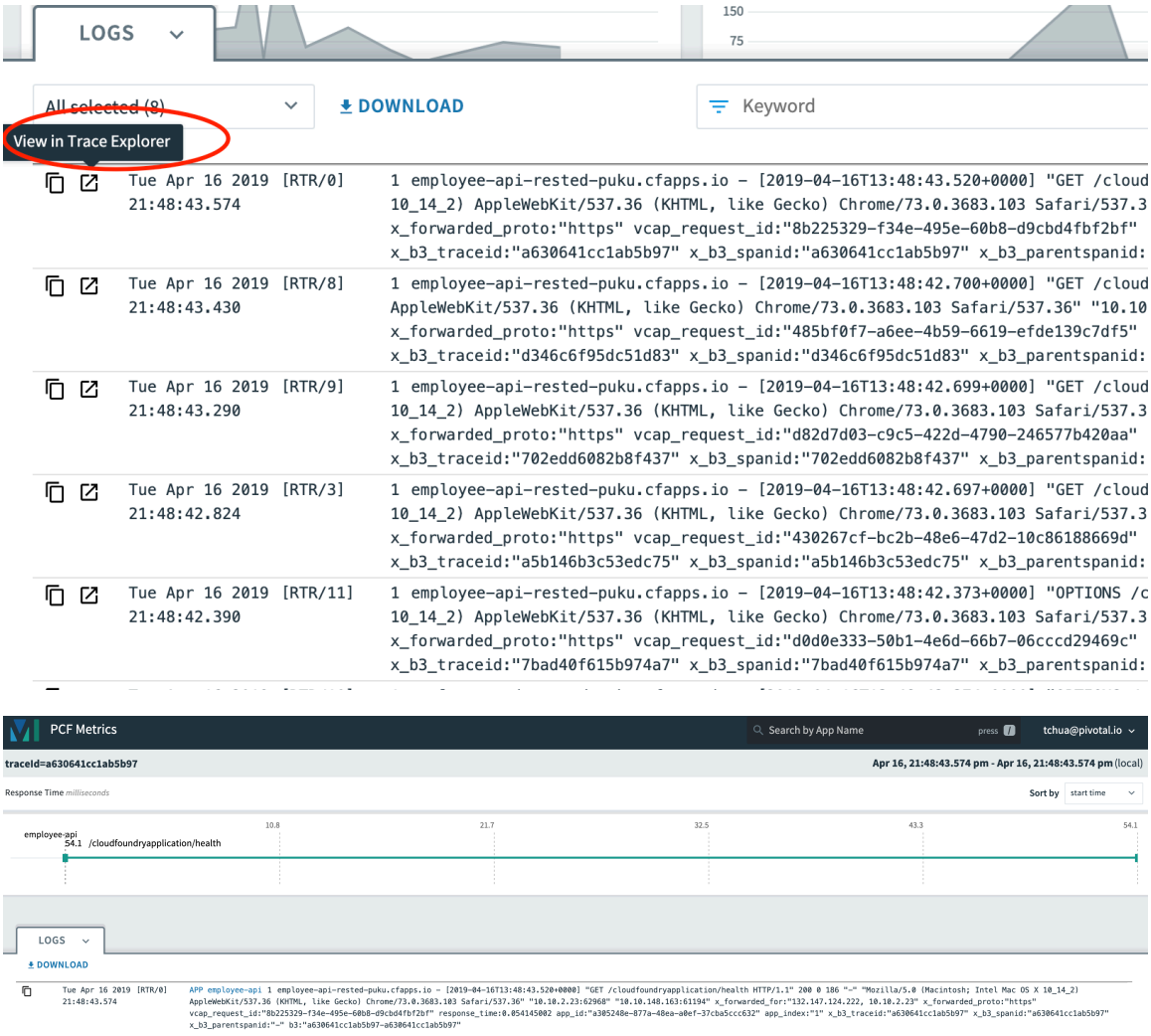


9. You also have the option to retrieve or tail your app logs using ‘cf logs’ command in the cf cli. Note that the logs are automatically aggregated across all apps.



10. Another handy feature within the logs panel are the links to the trace explorer. Clicking on a link opens up the transaction trace for that request. If a call was to traverse multiple hops across different microservices, the traversal path will show up, giving breakdown such as latency per hop. Such information is useful for understanding the cause of slowness or errors across a multi-hop call.

Pivotal


LOGS
150
75
All selected (8)
DOWNLOAD
Keyword
View in Trace Explorer
Tue Apr 16 2019 21:48:43.574 [RTR/0] 1 employee-api-rested-puku.cfapps.io - [2019-04-16T13:48:43.520+0000] "GET /cloud
10_14_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/73.0.3683.103 Safari/537.3
x_forwarded_proto:"https" vcap_request_id:"8b225329-f34e-495e-60b8-d9cbd4fbf2bf"
x_b3_traceid:"a630641cc1ab5b97" x_b3_spanid:"a630641cc1ab5b97" x_b3_parentspanid:
Tue Apr 16 2019 21:48:43.430 [RTR/8] 1 employee-api-rested-puku.cfapps.io - [2019-04-16T13:48:42.700+0000] "GET /cloud
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/73.0.3683.103 Safari/537.36" "10.10
x_forwarded_proto:"https" vcap_request_id:"485bf0f7-a6ee-4b59-6619-efde139c7df5"
x_b3_traceid:"d346c6f95dc51d83" x_b3_spanid:"d346c6f95dc51d83" x_b3_parentspanid:
Tue Apr 16 2019 21:48:43.290 [RTR/9] 1 employee-api-rested-puku.cfapps.io - [2019-04-16T13:48:42.699+0000] "GET /cloud
10_14_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/73.0.3683.103 Safari/537.3
x_forwarded_proto:"https" vcap_request_id:"d82d7d03-c9c5-422d-4790-246577b420aa"
x_b3_traceid:"702edd6082b8f437" x_b3_spanid:"702edd6082b8f437" x_b3_parentspanid:
Tue Apr 16 2019 21:48:42.824 [RTR/3] 1 employee-api-rested-puku.cfapps.io - [2019-04-16T13:48:42.697+0000] "GET /cloud
10_14_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/73.0.3683.103 Safari/537.3
x_forwarded_proto:"https" vcap_request_id:"430267cf-bc2b-48e6-47d2-10c86188669d"
x_b3_traceid:"a5b146b3c53edc75" x_b3_spanid:"a5b146b3c53edc75" x_b3_parentspanid:
Tue Apr 16 2019 21:48:42.390 [RTR/11] 1 employee-api-rested-puku.cfapps.io - [2019-04-16T13:48:42.373+0000] "OPTIONS /c
10_14_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/73.0.3683.103 Safari/537.3
x_forwarded_proto:"https" vcap_request_id:"d0d0e333-50b1-4e6d-66b7-06cccd29469c"
x_b3_traceid:"7bad40f615b974a7" x_b3_spanid:"7bad40f615b974a7" x_b3_parentspanid:

PCF Metrics
Search by App Name
tchua@pivotal.io
traceId=a630641cc1ab5b97
Apr 16, 21:48:43.574 pm - Apr 16, 21:48:43.574 pm (local)
Response Time milliseconds
Sort by
start time
10.8
21.7
32.5
43.3
54.1
employee-api
54.1 /cloudfoundryapplication/health
LOGS
DOWNLOAD
Tue Apr 16 2019 21:48:43.574 [RTR/0]
APP employee-api 1 employee-api-rested-puku.cfapps.io - [2019-04-16T13:48:43.520+0000] "GET /cloudfoundryapplication/health HTTP/1.1" 200 0 186 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_2)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/73.0.3683.103 Safari/537.36" "10.10.2.23:62968" "10.10.148.163:61194" x_forwarded_for:"132.147.124.222, 10.10.2.23" x_forwarded_proto:"https"
vcap_request_id:"8b225329-f34e-495e-60b8-d9cbd4fbf2bf" response_time:0.054145002 app_id:"a305248e-877a-48ea-a0ef-37cba5ccc632" app_index:"1" x_b3_traceid:"a630641cc1ab5b97" x_b3_spanid:"a630641cc1ab5b97"
x_b3_parentspanid:"-" b3:"a630641cc1ab5b97-a630641cc1ab5b97"