Pivotal

Pivotal Cloud Foundry

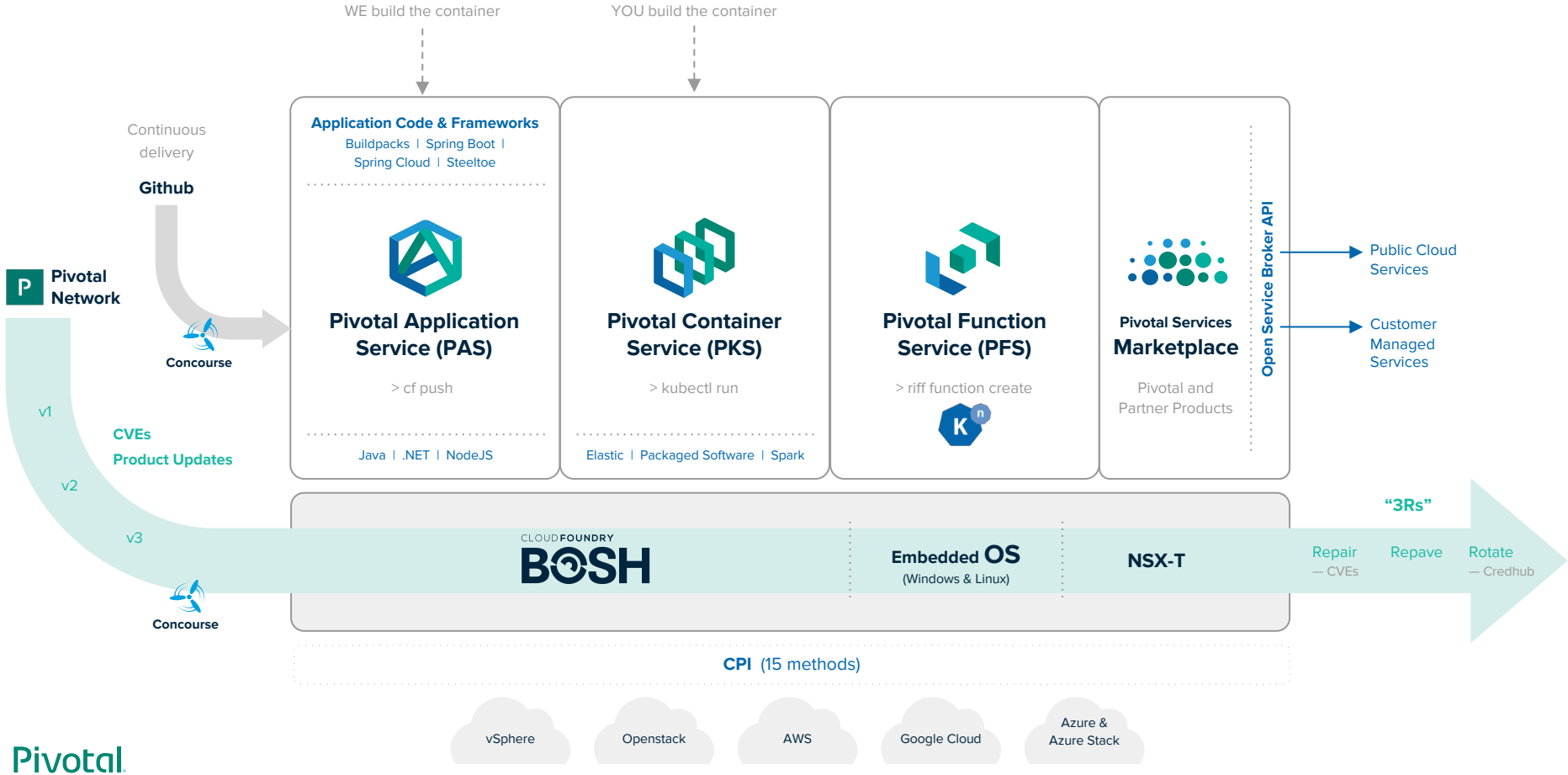
Spring Boot to PCF Developer Workshop

Pas Apicella = {
  Company: Pivotal,
  Advisory Platform Architect: http://pivotal.io,
  Email: papicella@pivotal.io
  Twitter: @PasApicella,
  Blog: http://theblasfrompas.blogspot.com.au/
}

High Level View

What is PCF?

Any Workload Anywhere Anytime



Pivotal

Pivotal Application Service (PAS): A Runtime for Apps

Increase speed and deploy code to production thousands of times per month. Use PAS to run Java, .NET, and Node apps.

Pivotal
Application Service™

Best runtime for Spring and Spring Boot — Spring's microservice patterns—and Spring Boot's executable jars—are ready-made for PAS.

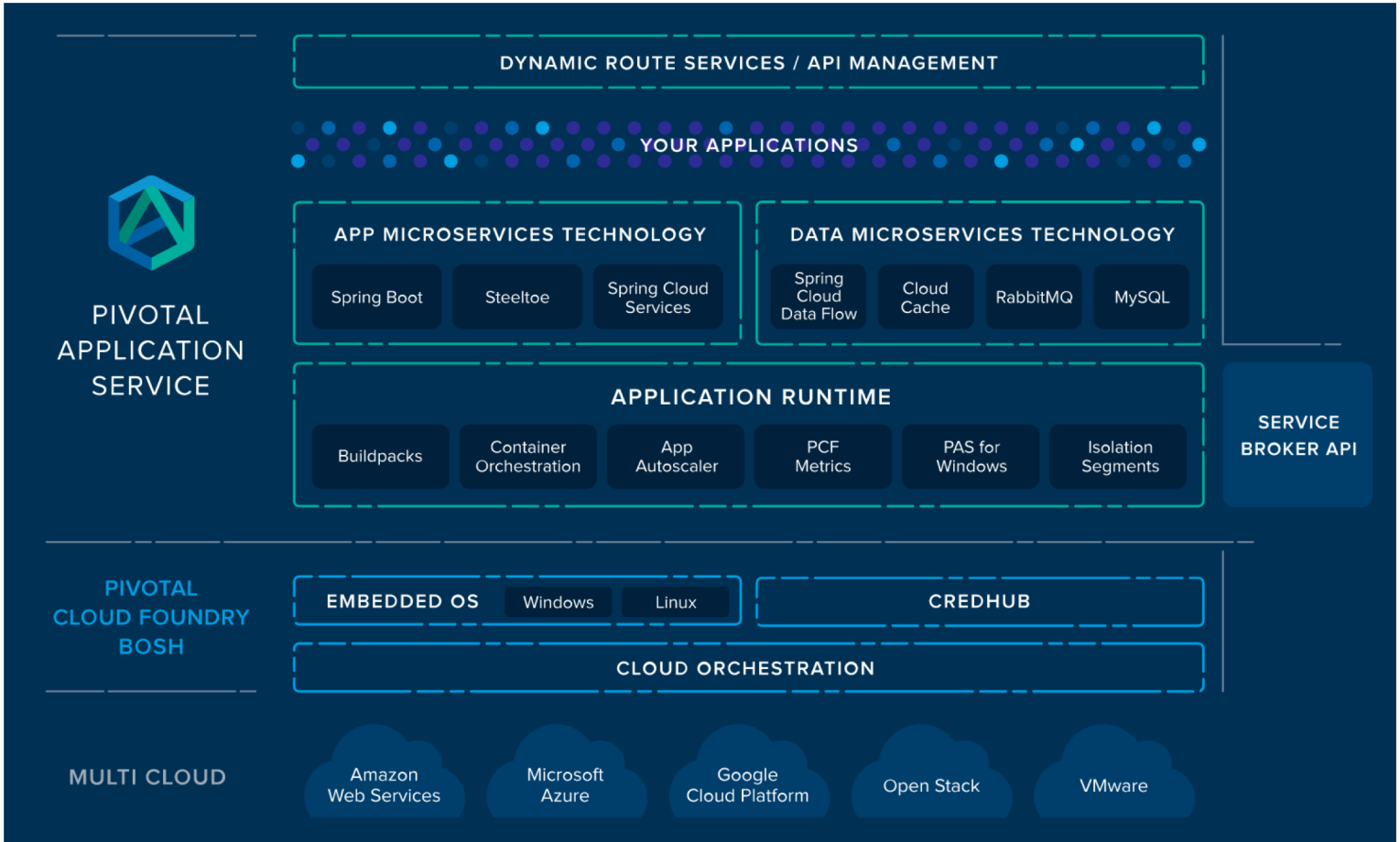
Turnkey microservices operations and security — Spring Cloud Services brings microservices best practices to PAS. It includes Config Server, Service Registry, and Circuit Breaker Dashboard.

A native Windows and .NET experience — Use PAS to run new apps built with .NET Core. Run your legacy .NET Framework apps on PAS too, using the .NET Hosted Web Core buildpack. Push applications to containers running on Windows Server 2016.

Built for apps — PAS has everything to need to run apps. Buildpacks manage runtime dependencies; metrics, logging, and scaling are done for you. Multitenancy, and blue/green deployment patterns are built-in. Extend apps with a rich service catalog.

Container-ready — PAS supports the OCI format for Docker images. Run platform-built and developer-built containers.

Spring Boot To PCF Developer Workshop



Spring Boot to PCF Developer Workshop

What Is Needed

Spring Boot To PCF Developer Workshop

- An account with Pivotal Web Services (https://run.pivotal.io/)
- At least 2G of memory and 1 service spare capacity on your PWS account
- CF CLI installed (https://github.com/cloudfoundry/cli)
- Maven CLI installed (https://maven.apache.org/download.cgi), for this workshop it was tested with “Apache Maven 3.3.9"
- Java 1.8 installed, this workshop was tested with “1.8.0_91”.
- IntelliJ IDEA , Eclipse or Spring Source Tool Suite or similar Java IDE
- CURL or HTTPie or a browser that can display RESTful JSON data through a plugin

Spring Boot to PCF Developer Workshop

What Is Covered

Spring Boot To PCF Developer Workshop

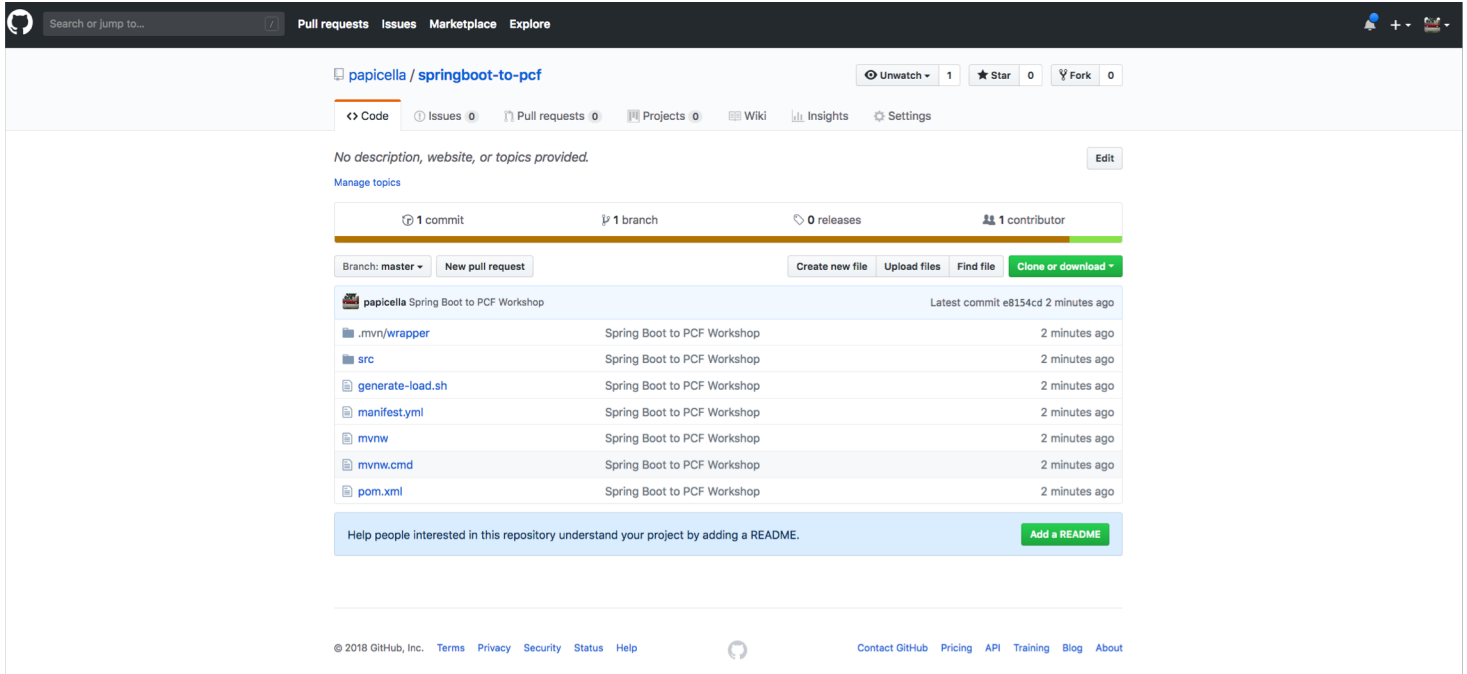
- Create a Data Bound Spring Boot Application Micro Service returning RESTful endpoints
- Tech Stack:
  - Hibernate JPA
  - Spring Boot
  - H2 In Memory Database as well as MySQL Database
  - Spring Data Repositories
  - Spring Boot Actuator
- Write some Java Code within an IDE (This is not a Spring Boot or Spring Course)
- Deploy to Pivotal Cloud Foundry
- Switch to a MySQL database in Pivotal Cloud Foundry without Code Changes
- All things application from
  - View application in “Application Manager”
  - Scale our application
  - View Logs
  - Monitor our application
  - Setup an Auto Scaling Policy
  - And more …..

Spring Boot to PCF Developer Workshop

Let’s Get Started

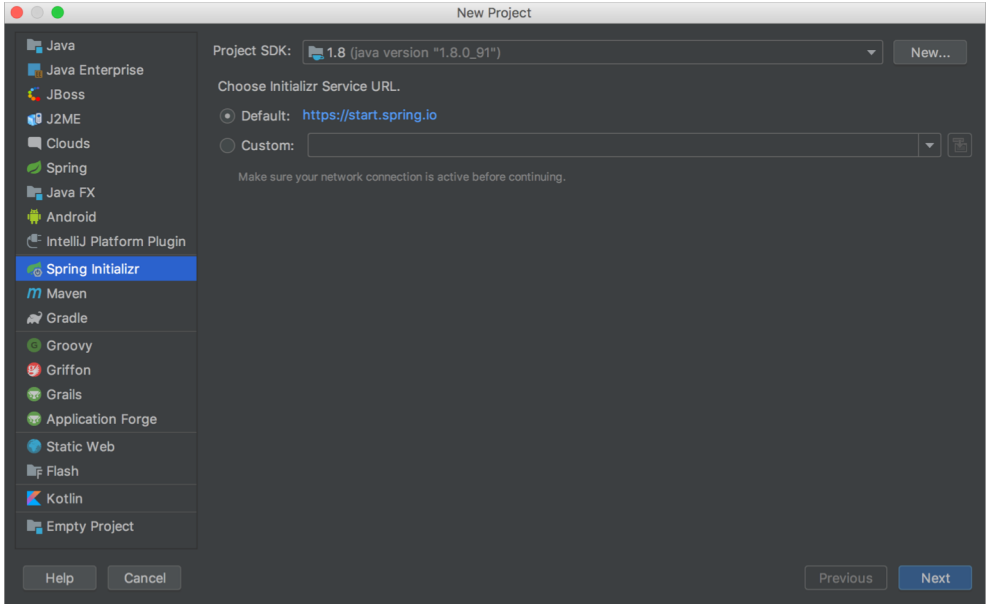
Spring Boot To PCF Developer Workshop

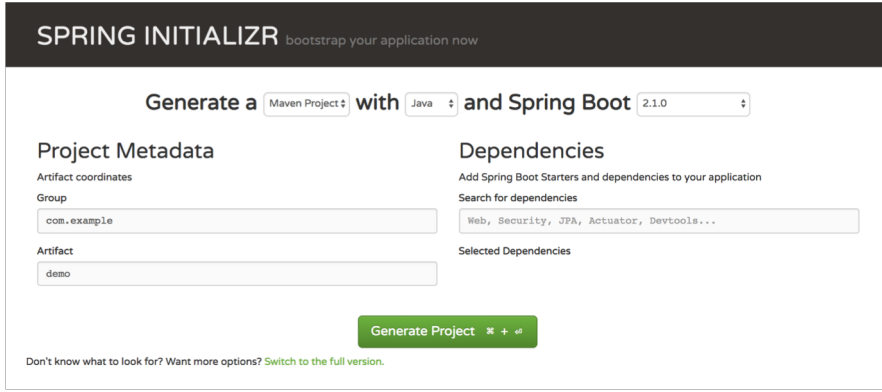
- The whole source code is on GitHub. If you decide to just copy and paste code text then use GitHub itself rather then copying/pasting from PPTX
    - https://github.com/papicella/springboot-to-pcf



Spring Boot To PCF Developer Workshop

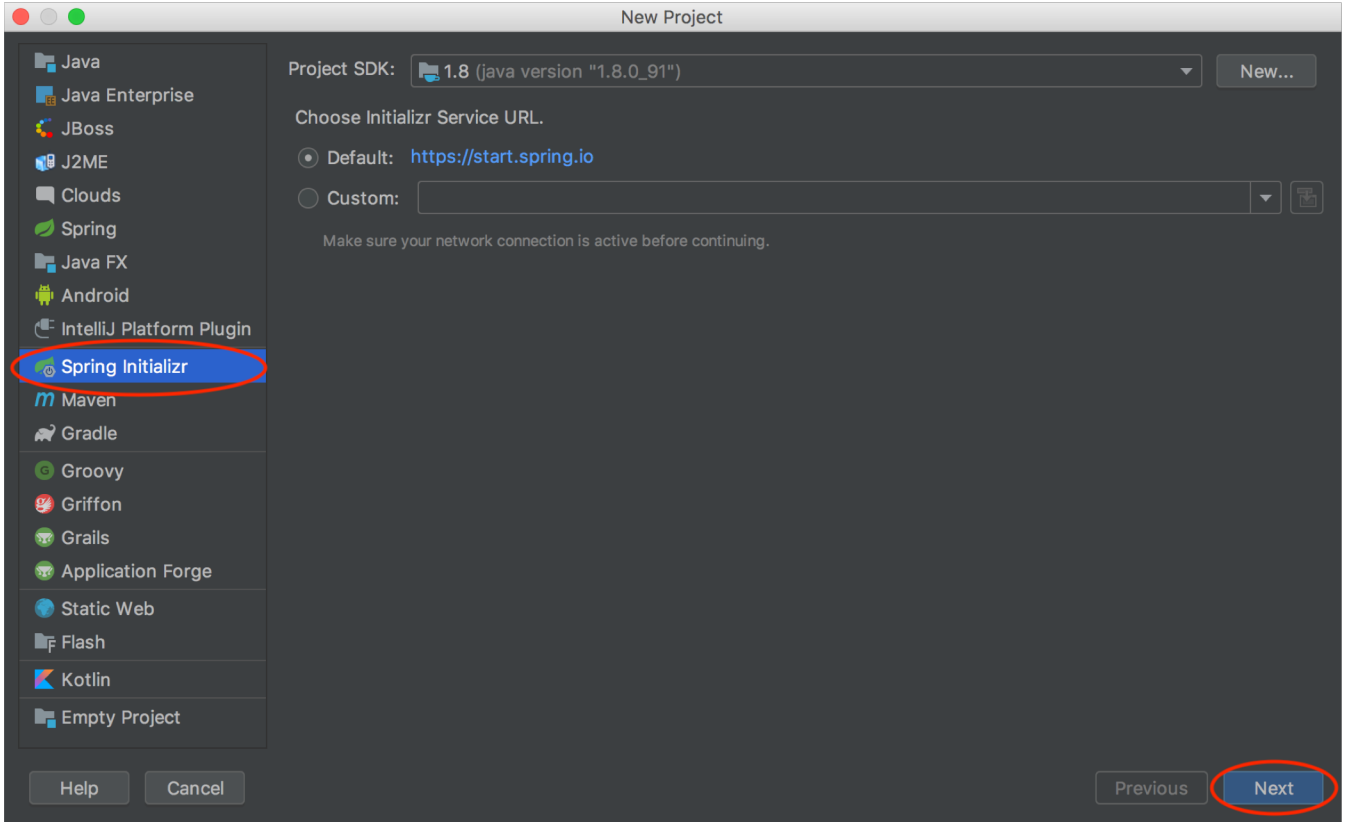
- Let's create a spring boot application one of two ways
  - http://start.spring.io (Web Browser) as shown to the right
  - Within your Java IDE (IntelliJ IDEA as shown below or Eclipse, STS)





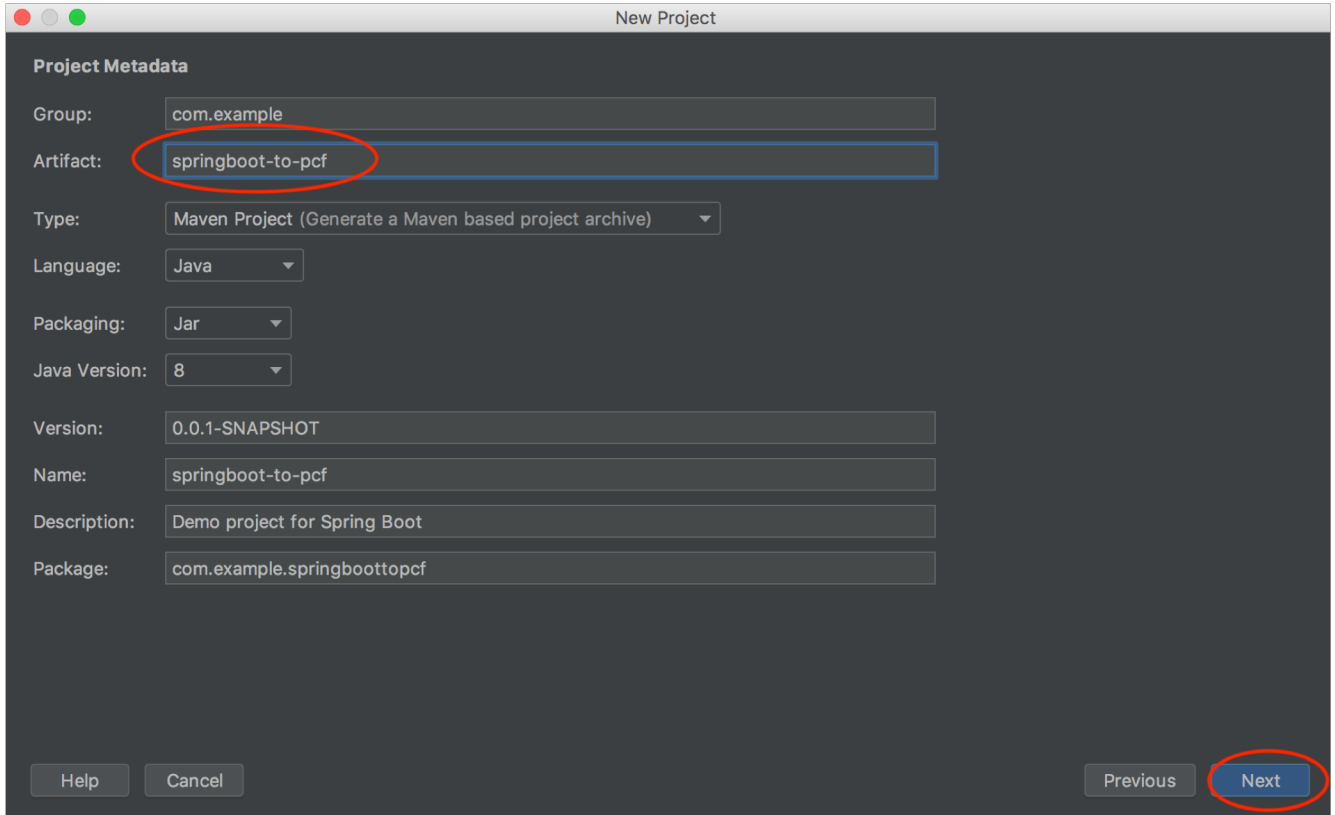
Spring Boot To PCF Developer Workshop

- Using IntelliJ IDEA, Select “Spring Initializer” and click “Next”



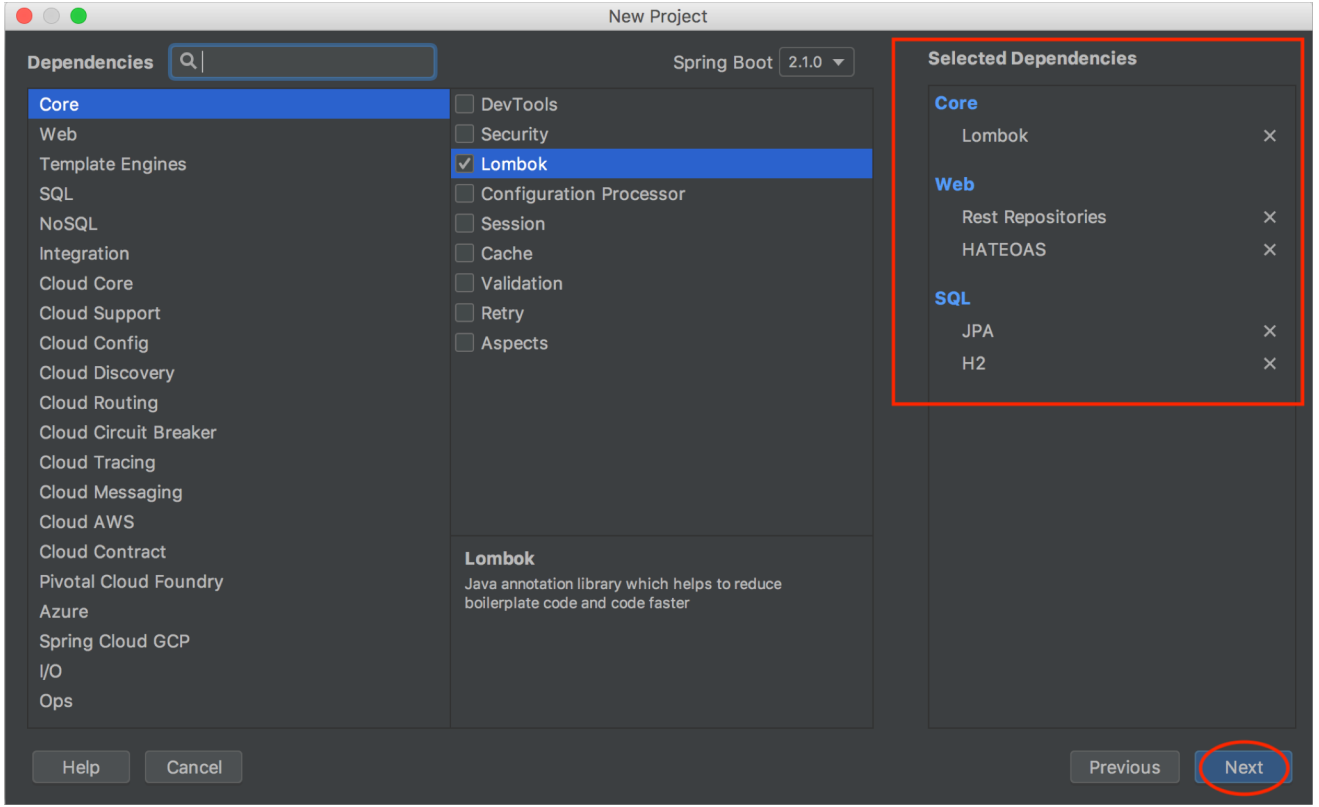
Spring Boot To PCF Developer Workshop

- Set the “Artifact” to “springboot-to-pcf” and click “Next”



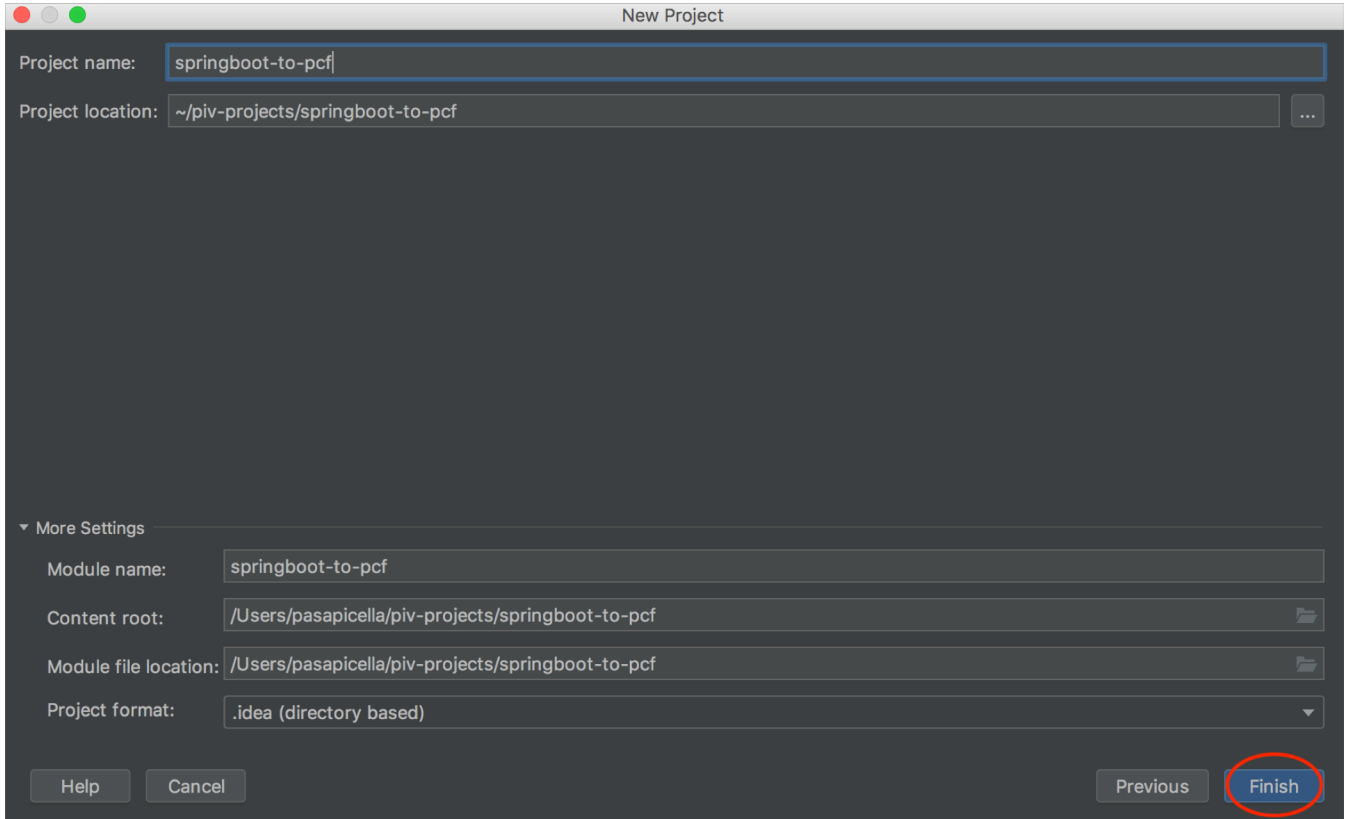
Spring Boot To PCF Developer Workshop

- Select the following dependencies and click “Next”
  - H2
  - JPA
  - Rest Repositories
  - Lombok



Spring Boot To PCF Developer Workshop

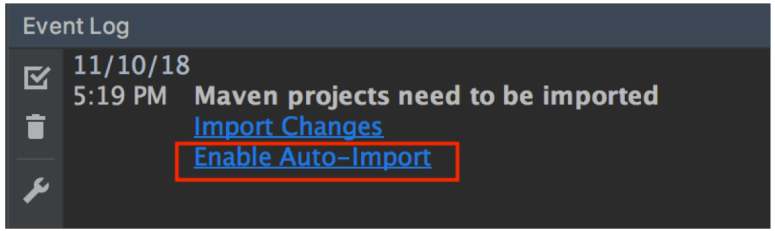
- Click “Finish” and wait for Maven to download the required JAR files

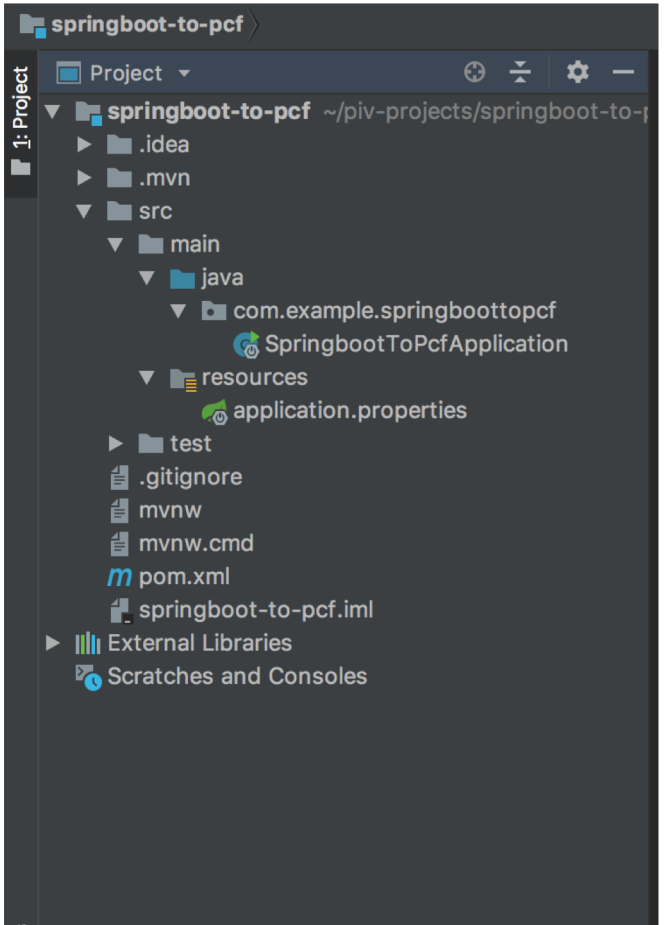


Spring Boot To PCF Developer Workshop

- Make sure you enable auto-import for maven as we will make changes at some point to our pom.xml file

- To the right is our Spring Boot project ready for us to write code





Spring Boot To PCF Developer Workshop

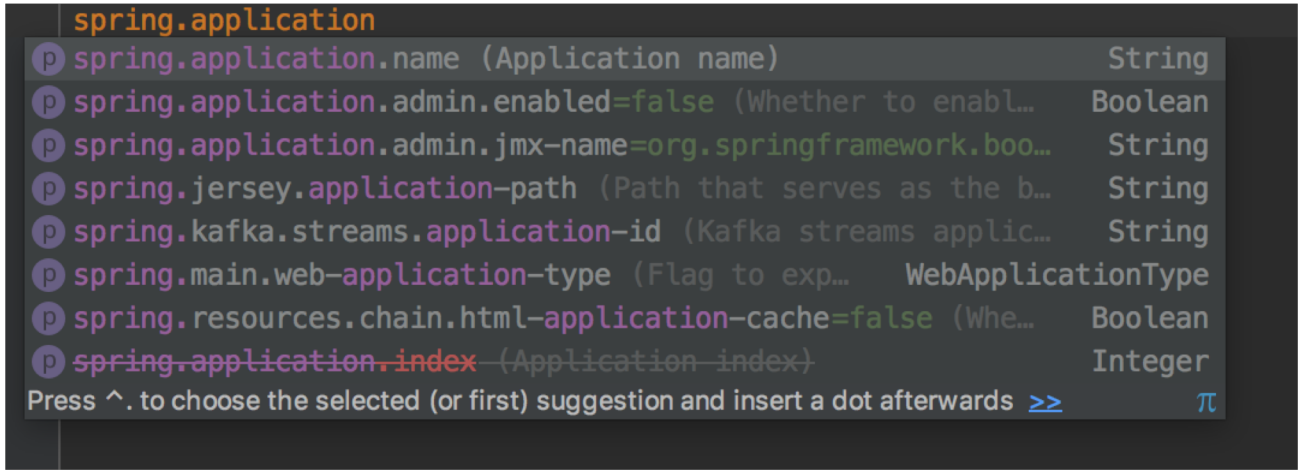
- Open the class file “SpringbootToPcfApplication.java”
- Note the use of the @SpringBootApplication annotation that makes this application a Spring Boot Application and auto configuration will then be applied to sensible defaults such as:
- H2: Spring boot will scan the class path and given H2 JAR files 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

```
@SpringBootApplication
public class SpringbootToPcfApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringbootToPcfApplication.class, args);
    }
}
```

Spring Boot To PCF Developer Workshop

- Lets actually set the Spring Boot application name by editing “application.properties” and adding a name as follows. You can use auto insight by typing just a few characters if you wish as shown below which is great way to see what other properties can be set

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



Spring Boot To PCF Developer Workshop

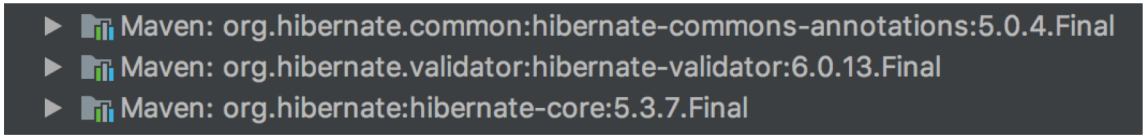
- Lets add an JPA Entity to our project with a class named “Employee”.

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

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

Spring Boot To PCF Developer Workshop

- Wondering what JPA provider we are using. Well it's Hibernate and Spring Boot starters are what have brought in the hibernate libraries.

```
<parent>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-parent</artifactId>
   <version>2.1.0.RELEASE</version>
   <relativePath/> <!-- lookup parent from repository -->
</parent>
```

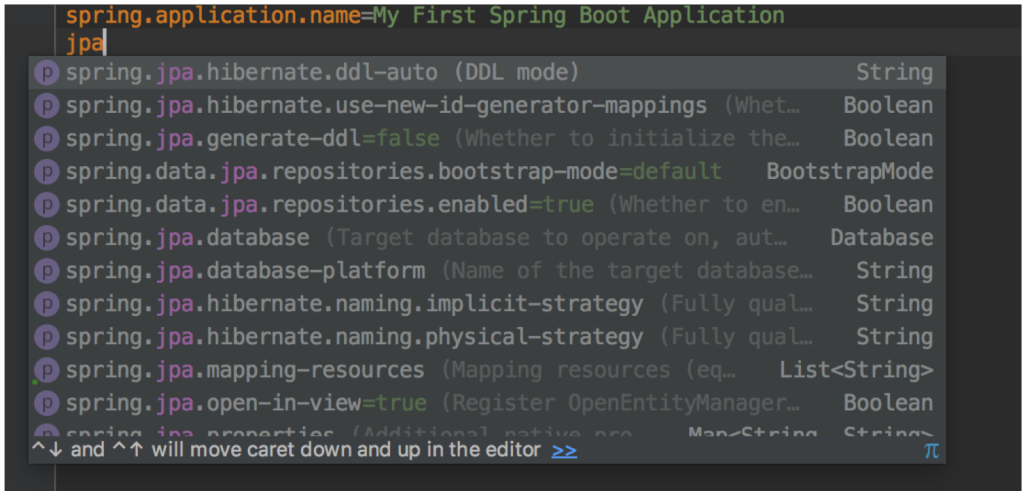



```
<dependencies>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
   </dependency>
```

Spring Boot To PCF Developer Workshop

- Hibernate will automatically create tables in the underlying database for us. It's good practice to set your desired behavior in the "application.properties" as follows in any case and here we clearly say DON'T do this. Once again you can use Auto Insight by typing a few characters as shown below

```
spring.application.name=My First Spring Boot Application
spring.jpa.hibernate.ddl-auto=none
spring.datasource.initialization-mode=always
```



Spring Boot To PCF Developer Workshop

- Lets create some RESTful end points for our Entity which we can do using Spring Rest Repositories
- Add an Interface as shown below
- Note the use of “@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

```
package com.example.springboottopcf;

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

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

Spring Boot To PCF Developer Workshop

- 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.
    - https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready
- Lets add it to our pom.xml as follows, yep that's all we need to do and we get RESTful end points added to our application automatically.

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

Spring Boot To PCF Developer Workshop

- We will also need to add the following “management” properties to our “application.properties” file as shown below to ensure our actuator endpoints are enabled and accessible.

```
spring.application.name=My First Spring Boot Application
spring.jpa.hibernate.ddl-auto=none
spring.datasource.initialization-mode=always
management.endpoint.health.enabled=true
management.endpoint.health.show-details=always
management.endpoints.enabled-by-default=true
management.endpoints.web.exposure.include=*
```

Spring Boot To PCF Developer Workshop

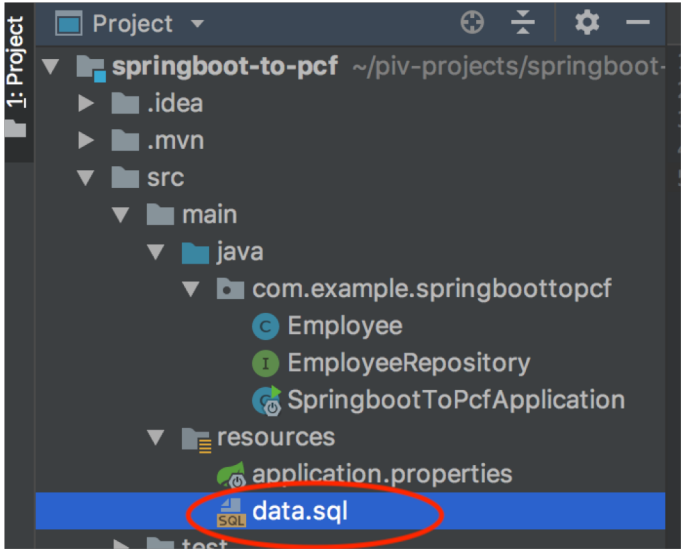
- Spring Boot Actuator Endpoints can be accessed using /actuator/{name} with examples below
    - beans: Displays a complete list of all the Spring beans in your application
    - env: Exposes properties from Spring's ConfigurableEnvironment
    - health: Shows application health information
    - autoconfig: Displays an auto-configuration report showing all auto-configuration candidates and the reason why they 'were' or 'were not' applied
    - metrics: Shows 'metrics' information for the current application
    - trace: Displays trace information (by default the last 100 HTTP requests)
    - mappings: Displays a collated list of all @RequestMapping paths

Many more …

Spring Boot To PCF Developer Workshop

- Now lets add some Employee data through our Entity. There are various ways but using Spring Boot we can add a “data.sql” file with some SQL inserts which will be inserted at runtime for us.

- Create a file called “data.sql” under the resources folder as follows

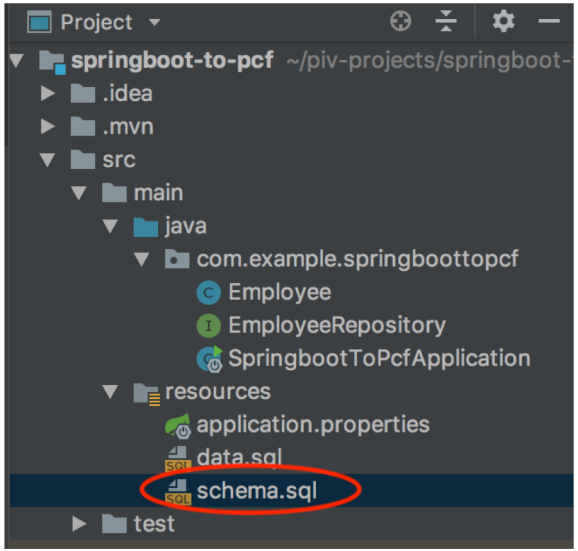



```
insert into employee (name) values ('pas');
insert into employee (name) values ('lucia');
insert into employee (name) values ('lucas');
insert into employee (name) values ('siena');
```

Spring Boot To PCF Developer Workshop

- Like we created a data.sql file a schema.sql file is required as we explicitly told hibernate not to create the table data for us and so Spring Boot is doing the schema/table initialization

- Create a file called “schema.sql” under the resources folder as follows



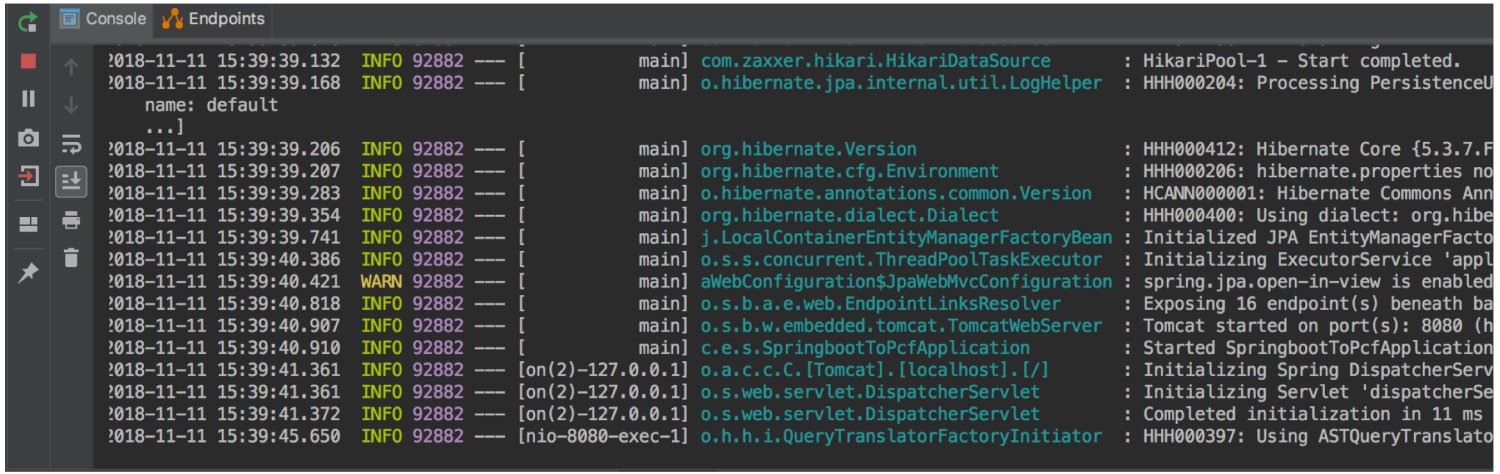

```
DROP TABLE IF EXISTS employee;


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

Spring Boot To PCF Developer Workshop

- Now lets run our Spring Boot application by right clicking on “SpringbootToPcfApplication” and selecting “Run”

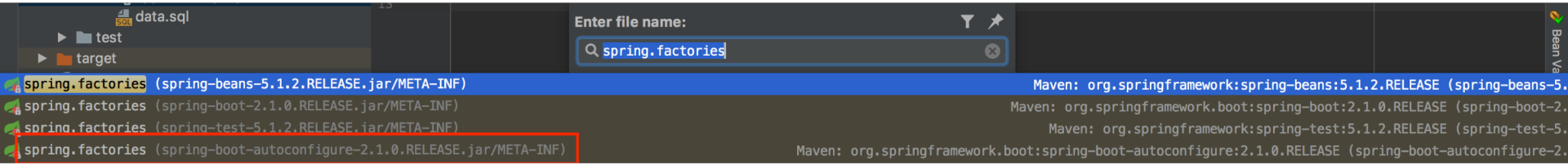
- Few things to Note:

    - 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



Spring Boot To PCF Developer Workshop

- Various Spring Boot Auto Configurations have been run the full list can be found using “Command - SHIFT - 0” and finding the file “spring.factories” in the spring auto configuration JAR file




```
# Initializers
org.springframework.context.ApplicationContextInitializer=\
org.springframework.boot.autoconfigure.SharedMetadataReaderFactoryContextInitializer,\
org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLoggingListener

# Application Listeners
org.springframework.context.ApplicationListener=\
org.springframework.boot.autoconfigure.BackgroundPreinitializer

# Auto Configuration Import Listeners
org.springframework.boot.autoconfigure.AutoConfigurationImportListener=\
org.springframework.boot.autoconfigure.condition.ConditionEvaluationReportAutoConfigurationImportListener

# Auto Configuration Import Filters
org.springframework.boot.autoconfigure.AutoConfigurationImportFilter=\
org.springframework.boot.autoconfigure.condition.OnBeanCondition,\
org.springframework.boot.autoconfigure.condition.OnClassCondition,\
org.springframework.boot.autoconfigure.condition.OnWebApplicationCondition

# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,\
org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration,\
```

Spring Boot To PCF Developer Workshop

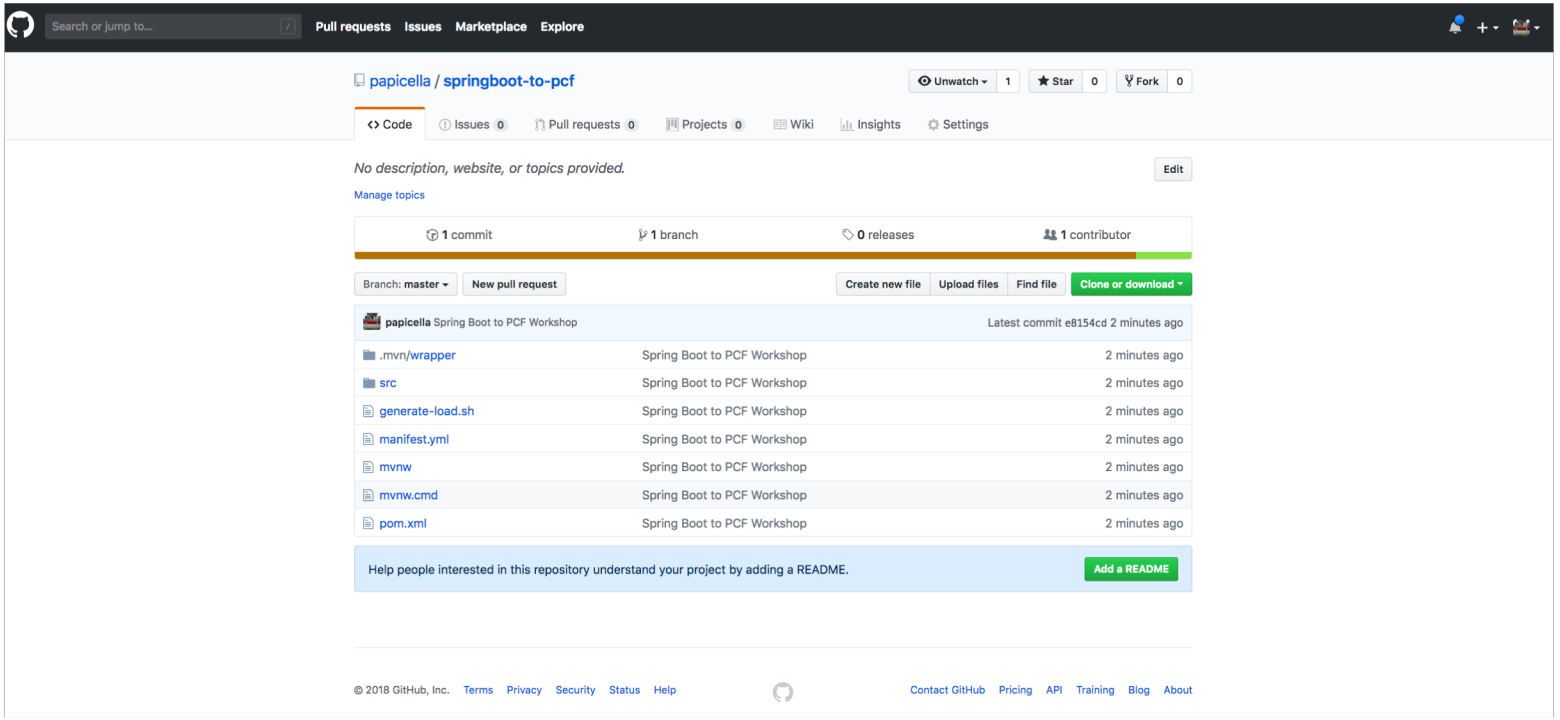
- Using either CURL, HTTPie or even a browser you can hit the main endpoint as follows.
    - http://localhost:8080/employees
- Various Actuator end points exist using:
    - /actuator/health, /actuator/autoconfig, /actuator/beans, /actuator/threaddump, /actuator/env , /actuator/mappings ,

```
pasapicella@pas-macbook:~$ http http://localhost:8080/actuator/health
HTTP/1.1 200
Content-Type: application/vnd.spring-boot.actuator.v2+json;charset=UTF-8
Date: Sun, 11 Nov 2018 04:52:54 GMT
Transfer-Encoding: chunked

{
    "details": {
        "db": {
            "details": {
                "database": "H2",
                "hello": 1
            },
            "status": "UP"
        },
        "diskSpace": {
            "details": {
                "free": 106433597440,
                "threshold": 10485760,
                "total": 500068036608
            },
            "status": "UP"
        }
    },
    "status": "UP"
}
```

```
pasapicella@pas-macbook:~$ http http://localhost:8080/employees
HTTP/1.1 200
Content-Type: application/hal+json;charset=UTF-8
Date: Sun, 11 Nov 2018 04:51:19 GMT
Transfer-Encoding: chunked

{
    "_embedded": {
        "employees": [
            {
                "_links": {
                    "employee": {
                        "href": "http://localhost:8080/employees/1"
                    },
                    "self": {
                        "href": "http://localhost:8080/employees/1"
                    }
                },
                "name": "pas"
            },
            {
                "_links": {
                    "employee": {
                        "href": "http://localhost:8080/employees/2"
                    },
                    "self": {
                        "href": "http://localhost:8080/employees/2"
                    }
                },
                "name": "lucia"
            },
```

Spring Boot To PCF Developer Workshop

- Had trouble along the way? The whole source code is on GitHub
  - https://github.com/papicella/springboot-to-pcf



Spring Boot to PCF Developer Workshop

What Cloud Native Platform can we deploy our employee microservice too?

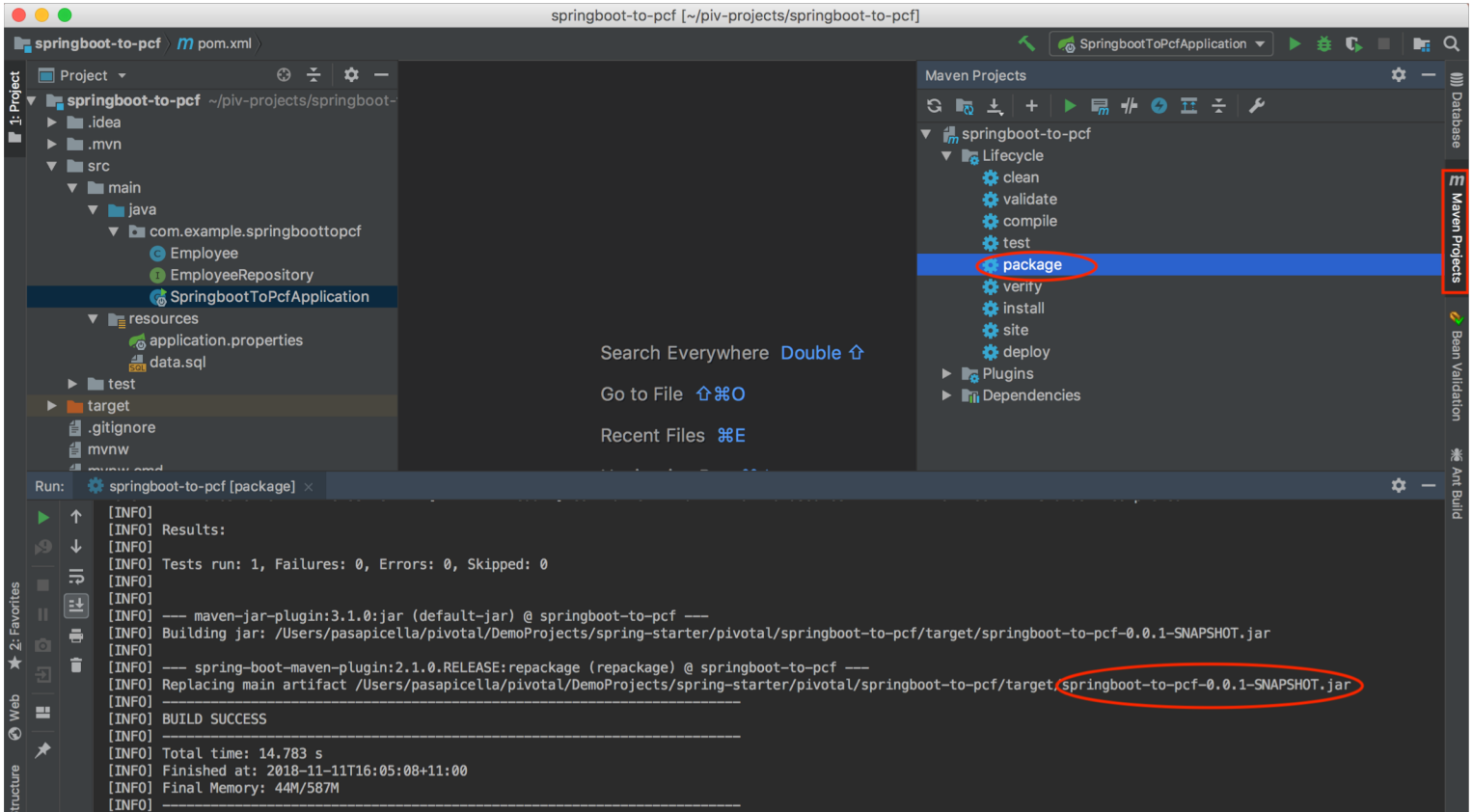
Spring Boot to PCF Developer Workshop

Well Pivotal Cloud Foundry of course

Pivotal

Spring Boot To PCF Developer Workshop

- Using “Maven Projects” tab package the application as shown below



Spring Boot To PCF Developer Workshop

- Installing the Cloud Foundry CLI

[{“cf help” : “provides command line options such as push etc”,
“cf help <command>” : “provides help for each command option”}]

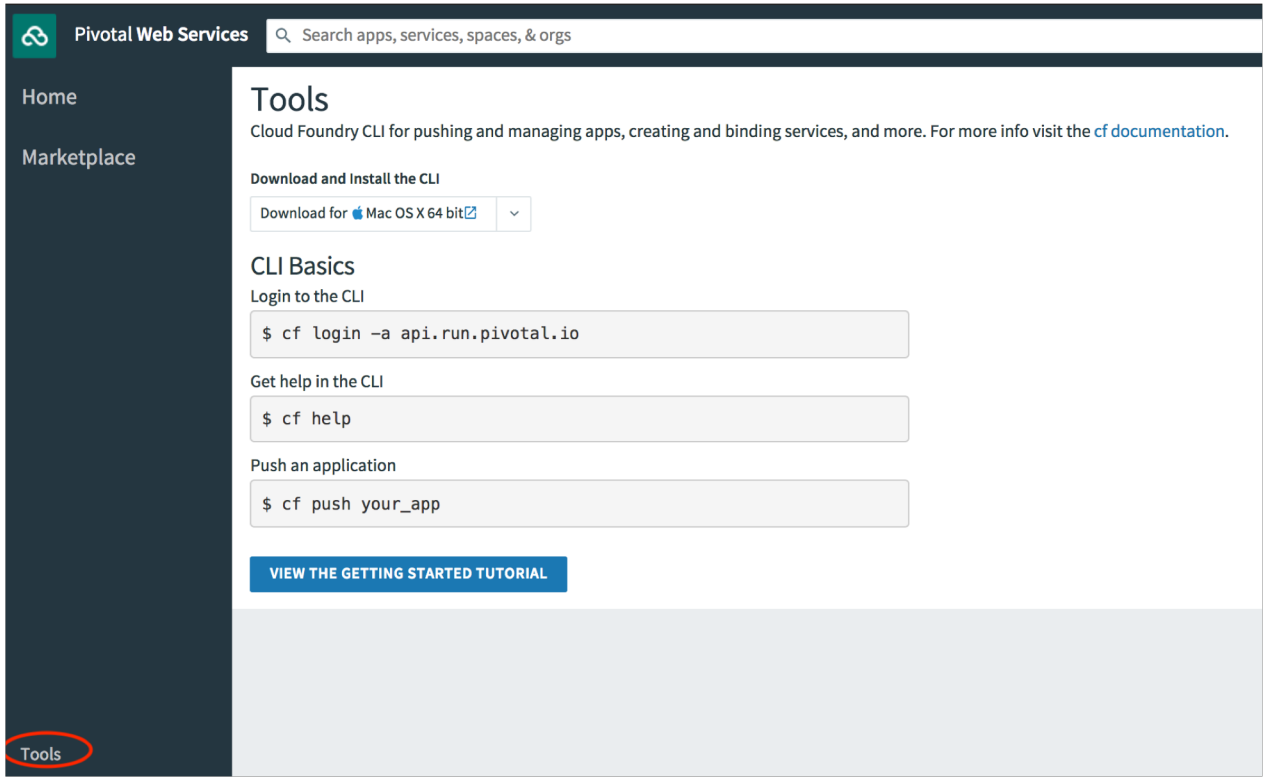
- Install using Github project as follows
    - $ brew install cloudfoundry/tap/cf-cli
- Easily install using BREW if on a MAC

$ brew tap cloudfoundry/tap
$ brew install cf-cli

```
pasapicella@pas-macbook:~$ cf --version
cf version 6.39.0+607d4f8be.2018-09-11
```

Spring Boot To PCF Developer Workshop

- Can also download directly from Application Manager UI as shown below, make sure you select the right O/S for your laptop



Spring Boot To PCF Developer Workshop

- Target cloud using “cf api” as shown below
- Only a single PUBLIC Instance of PWS exists in the USA, BUT PCF can also be installed on AWS, GCP, Azure or within your Firewall locally on VMware or OpenStack

[{“USA Instance” : “https://api.run.pivotal.io”}]

```
pasapicella@pas-macbook:~$ cf api https://api.run.pivotal.io
Setting api endpoint to https://api.run.pivotal.io...
OK

api endpoint:   https://api.run.pivotal.io
api version:    2.125.0
Not logged in. Use 'cf login' to log in.
```

Spring Boot To PCF Developer Workshop

- Login using “cf login” as shown below. Your username, organization name and space will be different then the example below of course. You should already have login details and those details available to be able to perform a login

[{“USAGE” : “cf login [-a API_URL] [-u USERNAME] [-p PASSWORD] [-o ORG] [-s SPACE]”}]

- $ cf login -u papicella@pivotal.io -p ***** -o papicella-org -s apple

```
pasapicella@pas-macbook:~$ cf login -u papicella@pivotal.io -o papicella-org -s apple
API endpoint: https://api.run.pivotal.io

Password>
Authenticating...
OK

Targeted org papicella-org

Targeted space apple



API endpoint:   https://api.run.pivotal.io (API version: 2.125.0)
User:           papicella@pivotal.io
Org:            papicella-org
Space:          apple
```

Spring Boot To PCF Developer Workshop

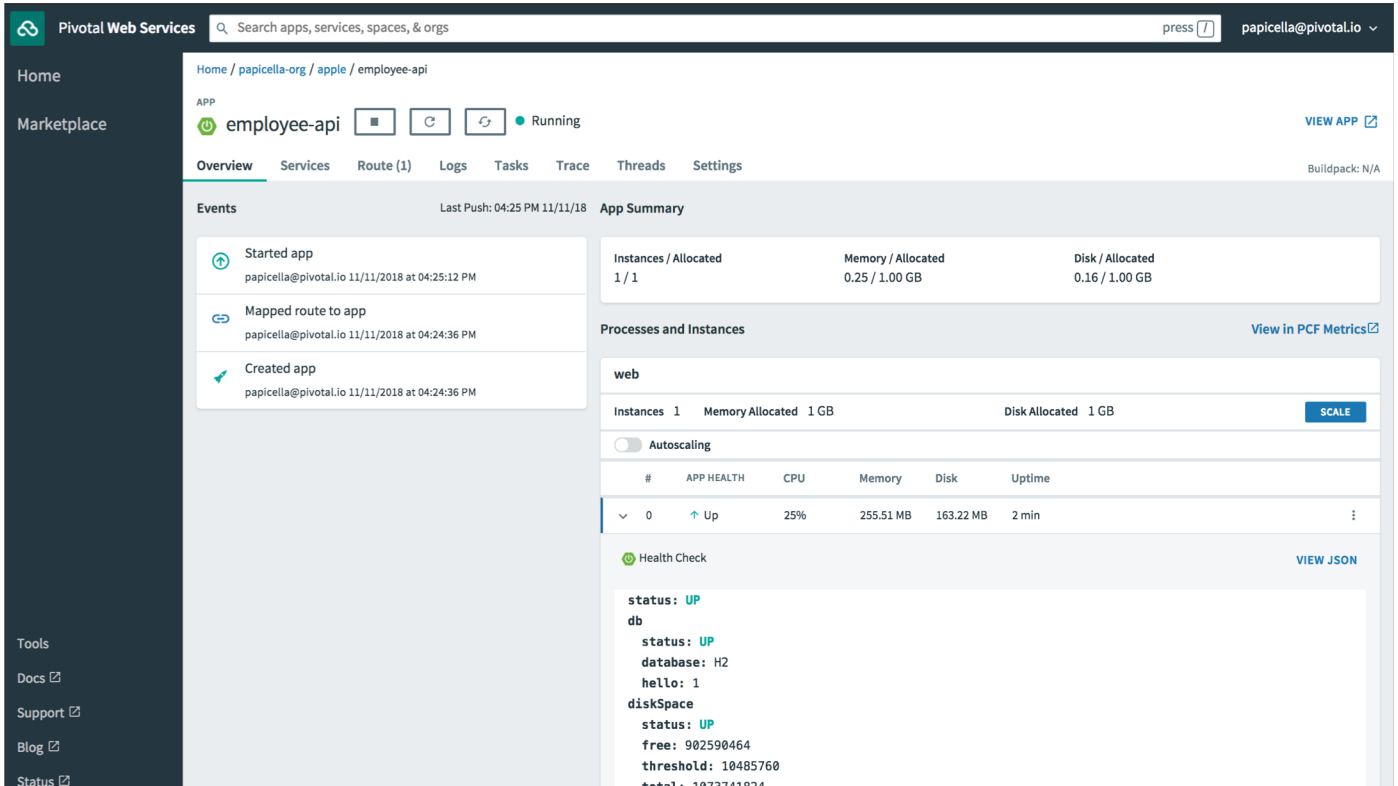
- Deploy using “cf push” as shown below

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

```
Waiting for app to start...

name:              employee-api
requested state:   started
routes:            employee-api-excellent-eland.cfapps.io
last uploaded:     Sun 11 Nov 16:25:37 AEDT 2018
stack:             cflinuxfs2
buildpacks:        client-certificate-mapper=1.8.0_RELEASE container-security-provider=1.16.0_RELEASE jav
                   java-main java-opts java-security jvmkill-agent=1.16.0_RELEASE open-jd...

type:            web
instances:       1/1
memory usage:    1024M
start command:   JAVA_OPTS="-agentpath:$PWD/.java-buildpack/open_jdk_jre/bin/jvmkill-1.16.0_RELEASE=printh
                 -Djava.ext.dirs=$PWD/.java-buildpack/container_security_provider:$PWD/.java-buildpack/op
                 $JAVA_OPTS" && CALCULATED_MEMORY=$($PWD/.java-buildpack/open_jdk_jre/bin/java-buildpack-m
                 -stackThreads=250 -vmOptions="$JAVA_OPTS") && echo JVM Memory Configuration: $CALCULATED_
                 exec $PWD/.java-buildpack/open_jdk_jre/bin/java $JAVA_OPTS -cp $PWD/. org.springframewor
     state     since                  cpu      memory         disk
#0   running   2018-11-11T05:26:10Z   129.9%   147.8M of 1G   163.2M of 1G
```

Spring Boot To PCF Developer Workshop

- Navigate to Pivotal Applications Manager UI and verify your deployed application is up and running
  - http://run.pivotal.io



Spring Boot To PCF Developer Workshop

- Can even view applications using the CF CLI as shown below
  - cf apps
  - cf app employee-api

```
pasapicella@pas-macbook:~$ cf apps
Getting apps in org papicella-org / space apple as papicella@pivotal.io...
OK

name           requested state   instances   memory   disk   urls
employee-api   started           1/1         1G       1G     employee-api-excellent-eland.cfapps.io
pasapicella@pas-macbook:~$ cf app employee-api

Showing health and status for app employee-api in org papicella-org / space apple as papicella@pivotal.io...

name:              employee-api
requested state:   started
routes:            employee-api-excellent-eland.cfapps.io
last uploaded:     Sun 11 Nov 16:25:37 AEDT 2018
stack:             cflinuxfs2
buildpacks:        client-certificate-mapper=1.8.0_RELEASE container-security-provider=1.16.0_RELEASE java-bu
                   java-main java-opts java-security jvmkill-agent=1.16.0_RELEASE open-jd...

type:           web
instances:      1/1
memory usage:   1024M
     state     since                  cpu    memory         disk
#0   running   2018-11-11T05:26:10Z   2.7%   262.7M of 1G   163.2M of 1G
```

Spring Boot To PCF Developer Workshop

- Lets just view one RESTful endpoint
    - $ http http://{url}/employees/2

Note: The URL can be obtained using “cf apps” as per the output in the previous slide.

```
pasapicella@pas-macbook:~$ http http://employee-api-excellent-eland.cfapps.io/employees/2
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 239
Content-Type: application/hal+json;charset=UTF-8
Date: Sun, 11 Nov 2018 05:33:19 GMT
X-Vcap-Request-Id: f23a1bda-029a-4f6d-62a1-b6b59fbe94ae

{
    "_links": {
        "employee": {
            "href": "http://employee-api-excellent-eland.cfapps.io/employees/2"
        },
        "self": {
            "href": "http://employee-api-excellent-eland.cfapps.io/employees/2"
        }
    },
    "name": "lucia"
}
```

Spring Boot To PCF Developer Workshop

- So how can we use a different RDBMS service like MySQL rather then H2?

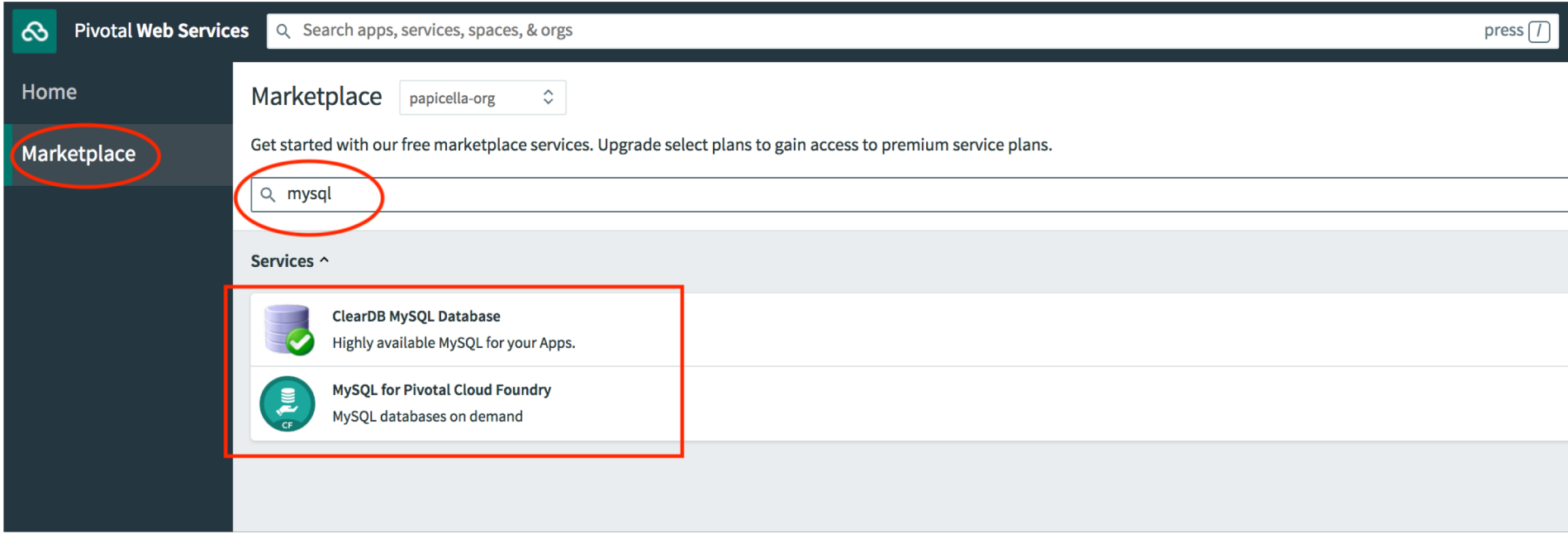
Lets start my adding MySQL Dependency to the maven pom.xml as shown below and then repackage to create a new JAR file. Put the MySQL driver before the H2 driver dependency

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

- That's it, Pivotal Cloud Foundry's buildpack along with Spring Boot will expect a MySQL database service to be bound to the application and then will create a DataSource to MySQL. If such a service isn't bound it will use H2 instead without any code changes!!!

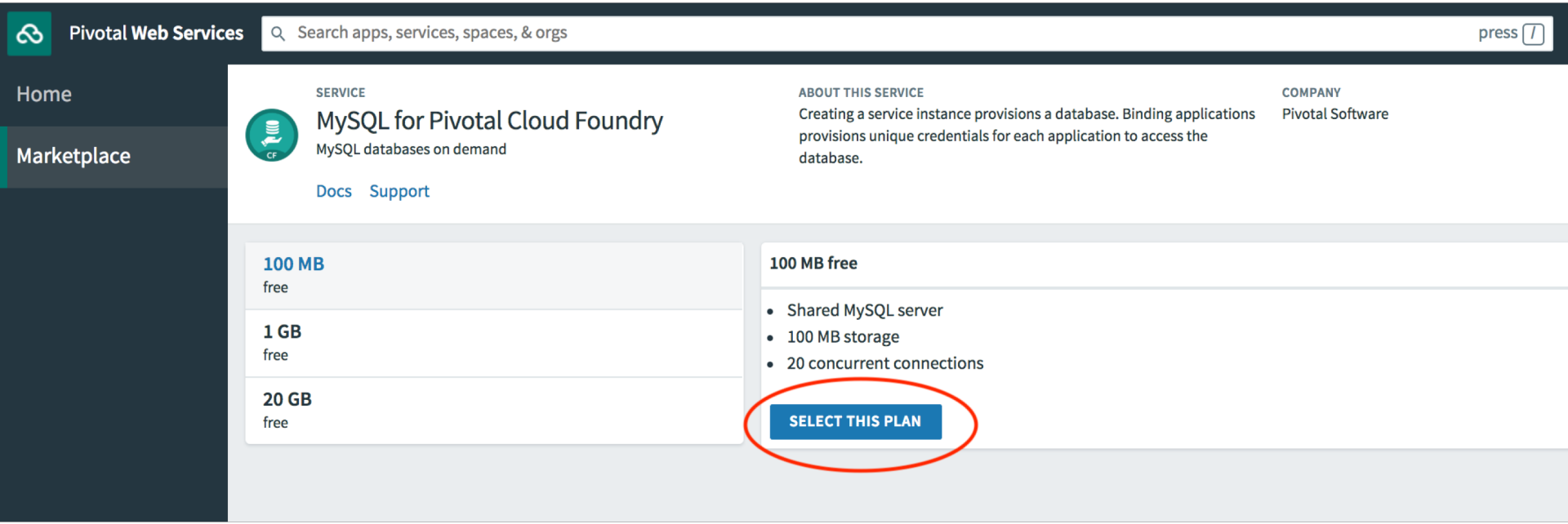
Spring Boot To PCF Developer Workshop

- Now using the Pivotal Applications Manager UI click on “Marketplace” and browse for “mysql” database instances as shown below



Spring Boot To PCF Developer Workshop

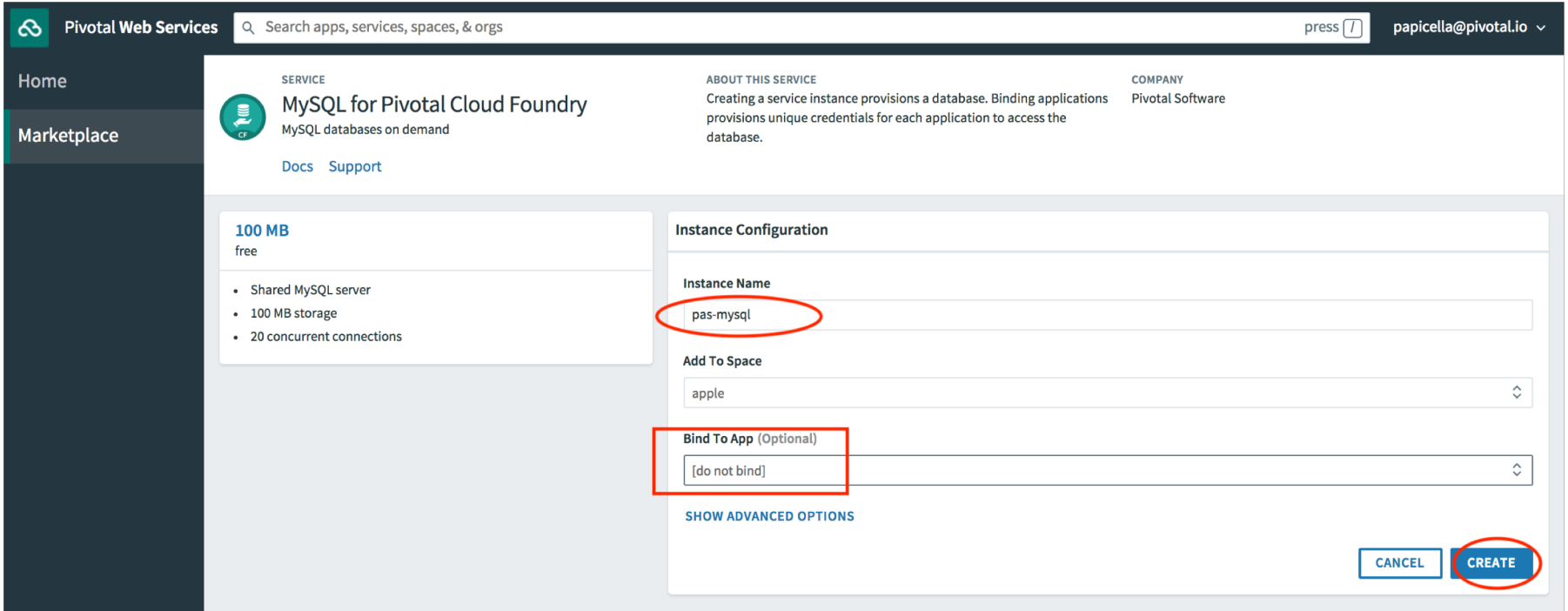
- Click on one of the “mysql” instances in the next few steps we select the Pivotal MySQL service. Once selected click on “Select This Plane” button



Spring Boot To PCF Developer Workshop

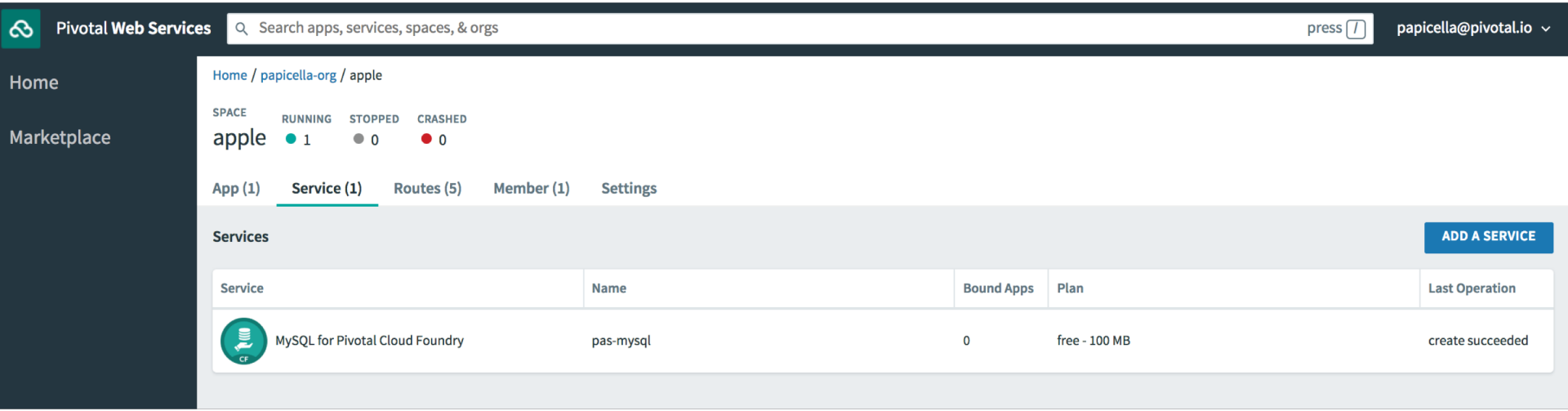
- In this screen lets give our service the following details then click create

Name: pas-mysql
Bind To App: [do not bind]



Spring Boot To PCF Developer Workshop

- Verify it has successfully created our MySQL service using Pivotal Applications Manager UI as well as the CF CLI as shown below

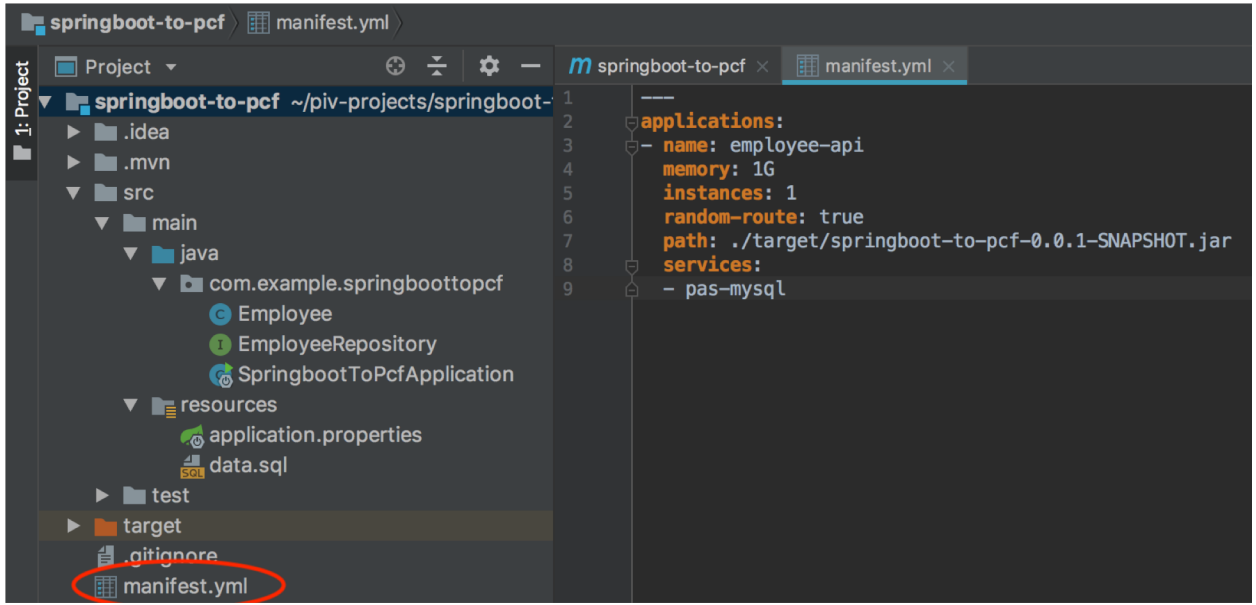



```
pasapicella@pas-macbook:~$ cf services
Getting services in org papicella-org / space apple as papicella@pivotal.io...

name         service   plan    bound apps   last operation
pas-mysql    p-mysql   100mb                create succeeded
```

Spring Boot To PCF Developer Workshop

- Now returning to our project let's create a file called "manifest.yml" with contents as shown below

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



Spring Boot To PCF Developer Workshop

- Now let's re-push our application which should have been packaged with the MySQL driver as per a previous step so it will use the MySQL database service we created

$ cf push

```
Waiting for app to start...

name:              employee-api
requested state:   started
routes:            employee-api-excellent-eland.cfapps.io
last uploaded:     Sun 11 Nov 18:03:45 AEDT 2018
stack:             cflinuxfs2
buildpacks:        client-certificate-mapper=1.8.0_RELEASE container-securit
                   java-main java-opts java-security jvmkill-agent=1.16.0_RE

type:            web
instances:       1/1
memory usage:    1024M
start command:   JAVA_OPTS="-agentpath:$PWD/.java-buildpack/open_jdk_jre/bir
                 -Djava.ext.dirs=$PWD/.java-buildpack/container_security_pro
                 $JAVA_OPTS" && CALCULATED_MEMORY=$($PWD/.java-buildpack/ope
                 -stackThreads=250 -vmOptions="$JAVA_OPTS") && echo JVM Memo
                 exec $PWD/.java-buildpack/open_jdk_jre/bin/java $JAVA_OPTS
     state     since                  cpu    memory         disk
#0   running   2018-11-11T07:04:14Z   0.0%   145.5M of 1G   165.3M of 1G
```

Spring Boot To PCF Developer Workshop

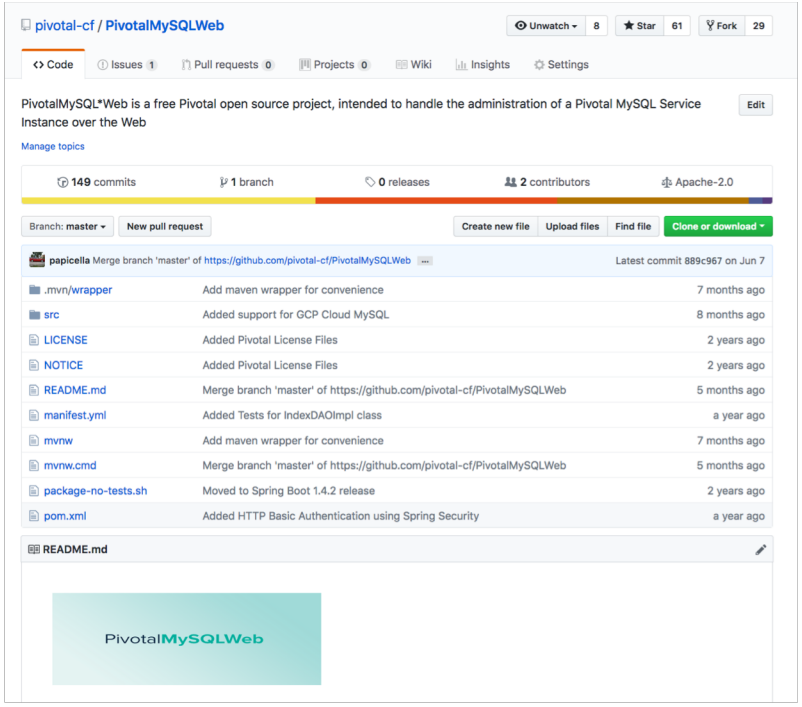
- And that's it our Spring Boot application using Hibernate can easily switch to another RDBMS and the cloud foundry buildpack along with Spring Boot will automatically create a Data Source from the bound service for us

View all employees using http://{URL}/employees

```
pasapicella@pas-macbook:~$ http http://employee-api-accountable-hartebeest.cfapps.io/employees/4
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 251
Content-Type: application/hal+json;charset=UTF-8
Date: Sun, 11 Nov 2018 10:00:00 GMT
X-Vcap-Request-Id: e7715169-782d-479c-6daf-e17fa601e735

{
    "_links": {
        "employee": {
            "href": "http://employee-api-accountable-hartebeest.cfapps.io/employees/4"
        },
        "self": {
            "href": "http://employee-api-accountable-hartebeest.cfapps.io/employees/4"
        }
    },
    "name": "pas"
}
```

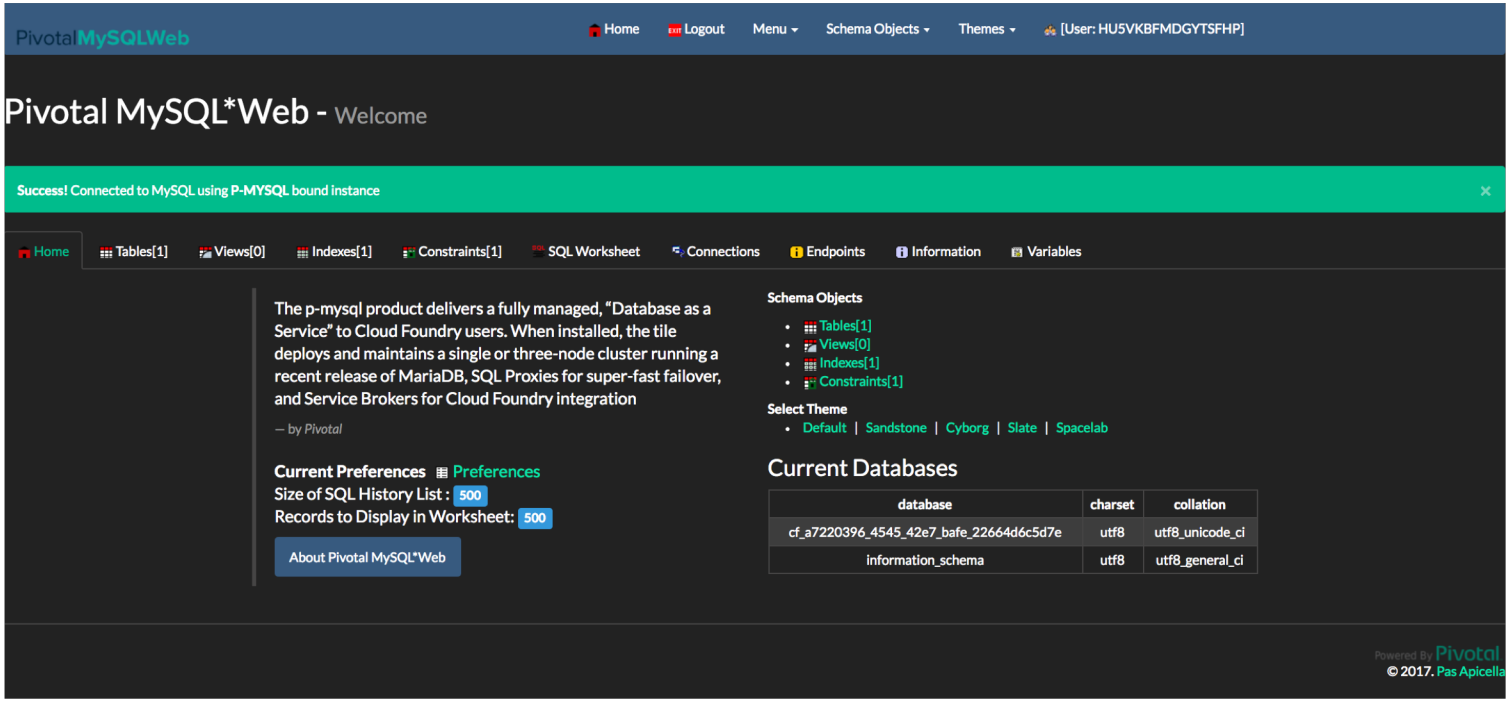
Spring Boot To PCF Developer Workshop

- I suppose we better prove we are actually using the MySQL database. To do that let's deploy Pivotal MySQL*Web using the GitHub URL below. Instructions on how to deploy are on the GitHub link
    - https://github.com/pivotal-cf/PivotalMySQLWeb



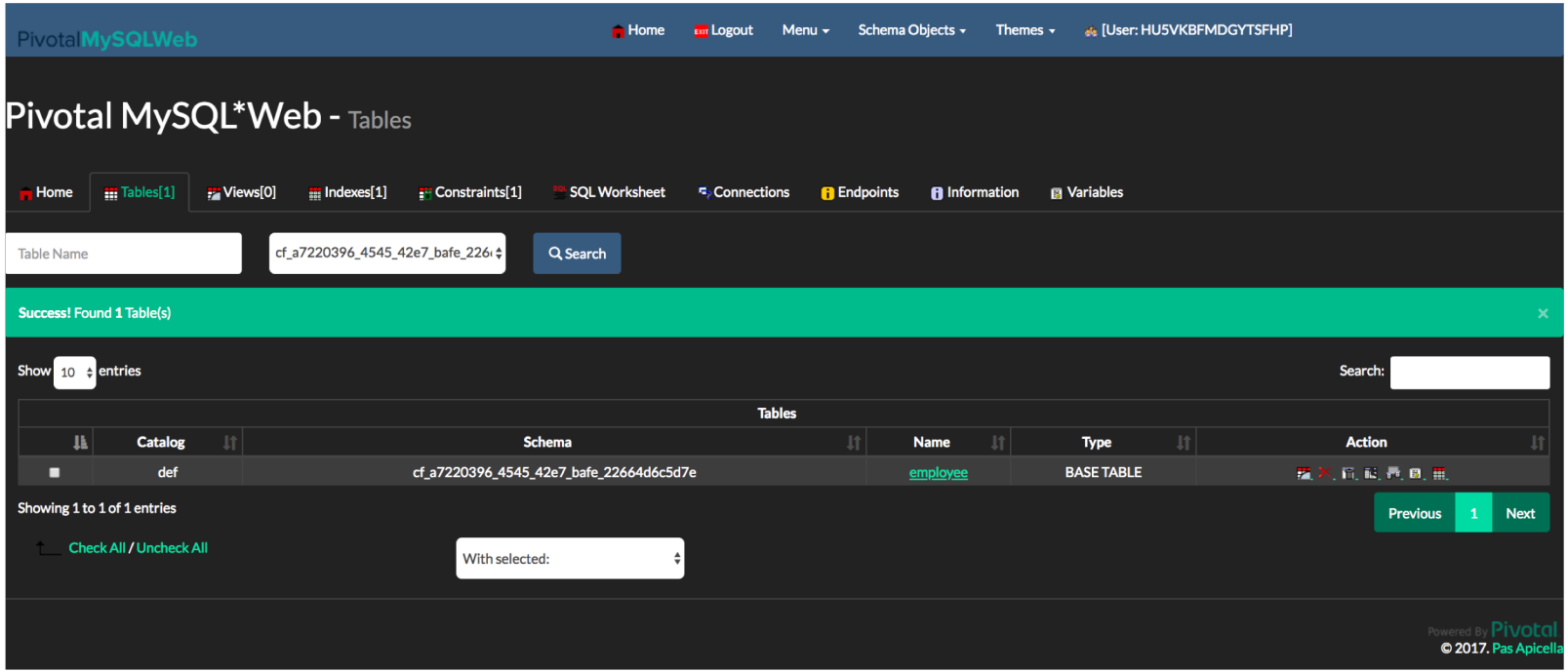
Spring Boot To PCF Developer Workshop

- Once Pivotal MYSQL*Web is deployed accessing it using it's URL and you get to a home page as follows providing we have bound the application to the same service name our "employee-api" using which is "pas-mysql"



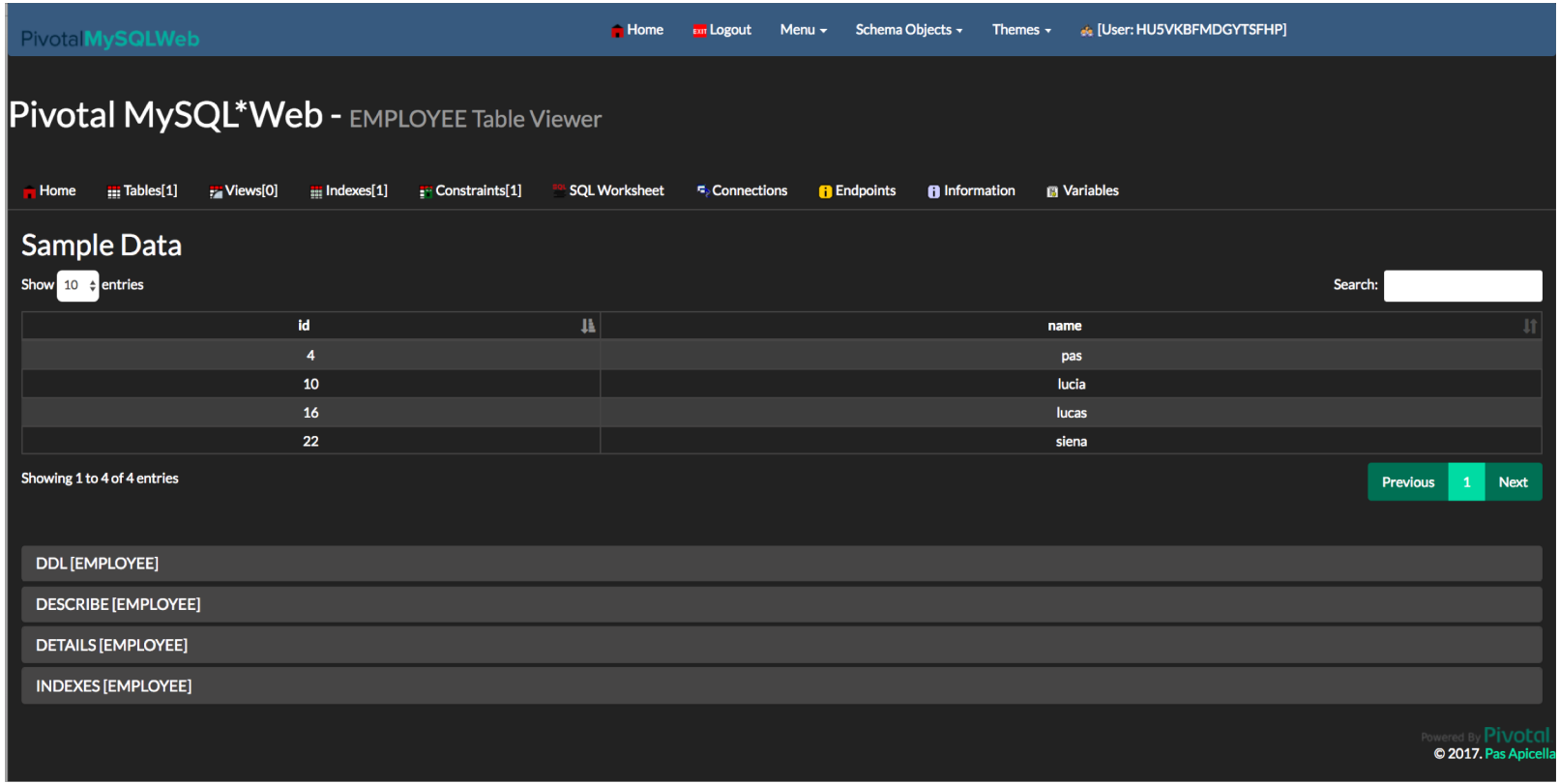
Spring Boot To PCF Developer Workshop

- Click on tables then click on “employee” table and you will see the data has been loaded into the MySQL database and we are not using H2 anymore



Spring Boot To PCF Developer Workshop

- His our “employee” data

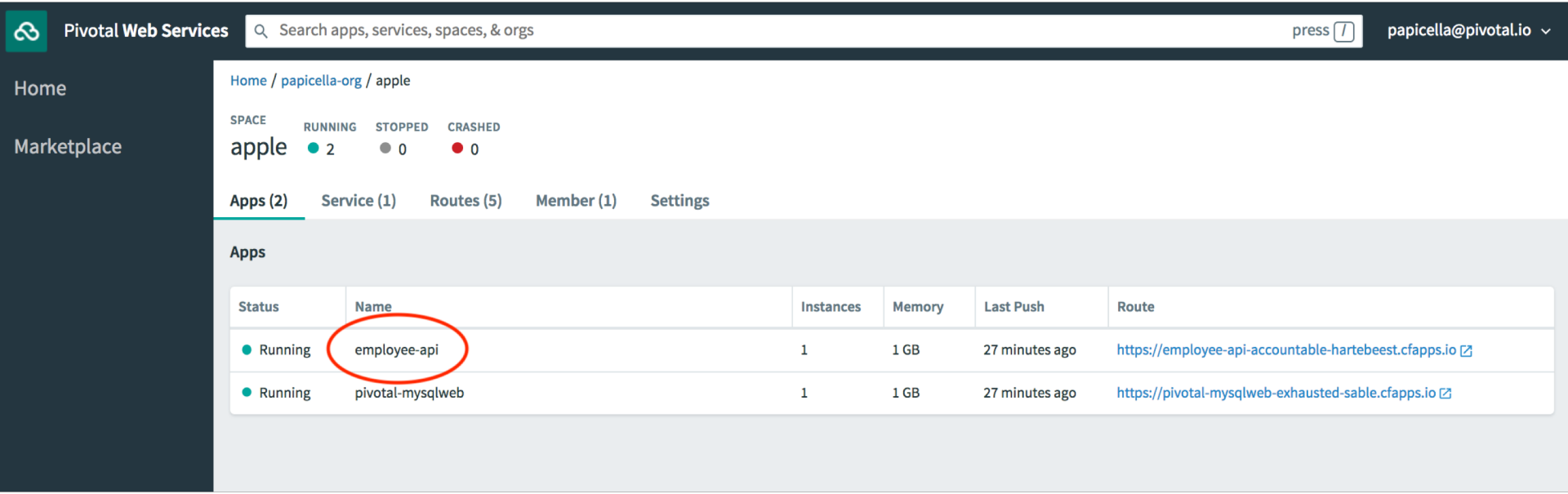


Spring Boot to PCF Developer Workshop

My application on PCF

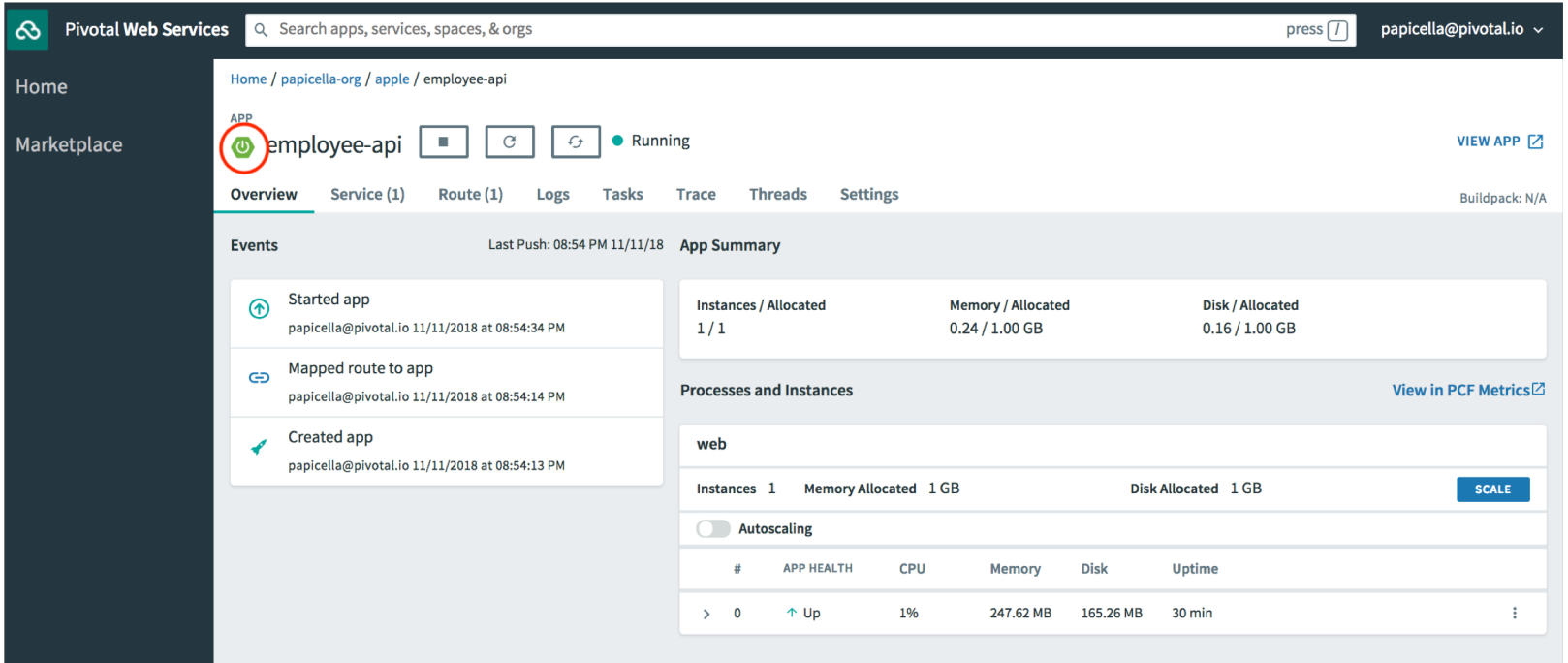
Spring Boot To PCF Developer Workshop

- Let's start by navigating to Applications Manager UI. We see we have two applications here. Our employee-api micro service and Pivotal MySQL*Web. Click on "employee-api" as shown below



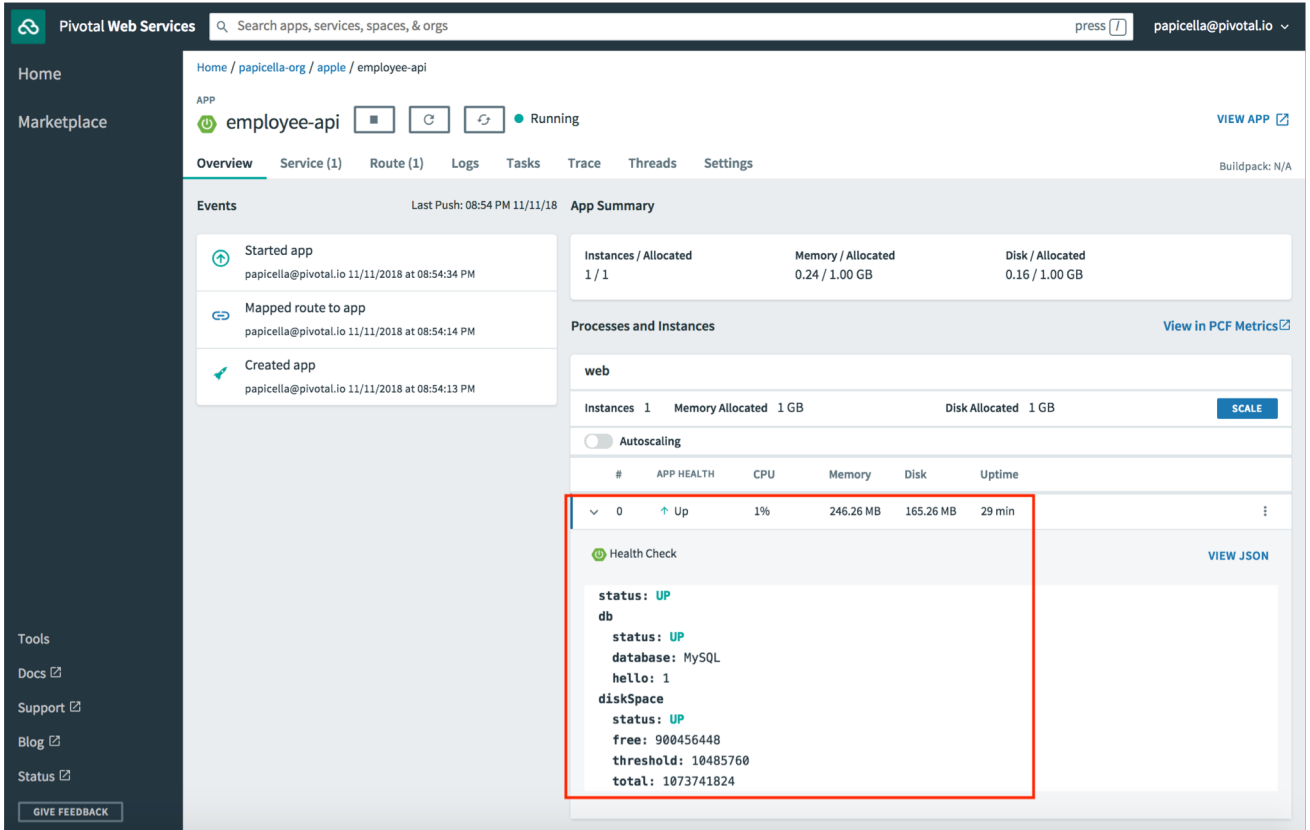
Spring Boot To PCF Developer Workshop

- From here you can view logs, events, routes, env variables etc
- Notice how a Spring ICON appears on this page that's because the application is using the Spring Actuator Library which the Applications Manager then provides further details for us



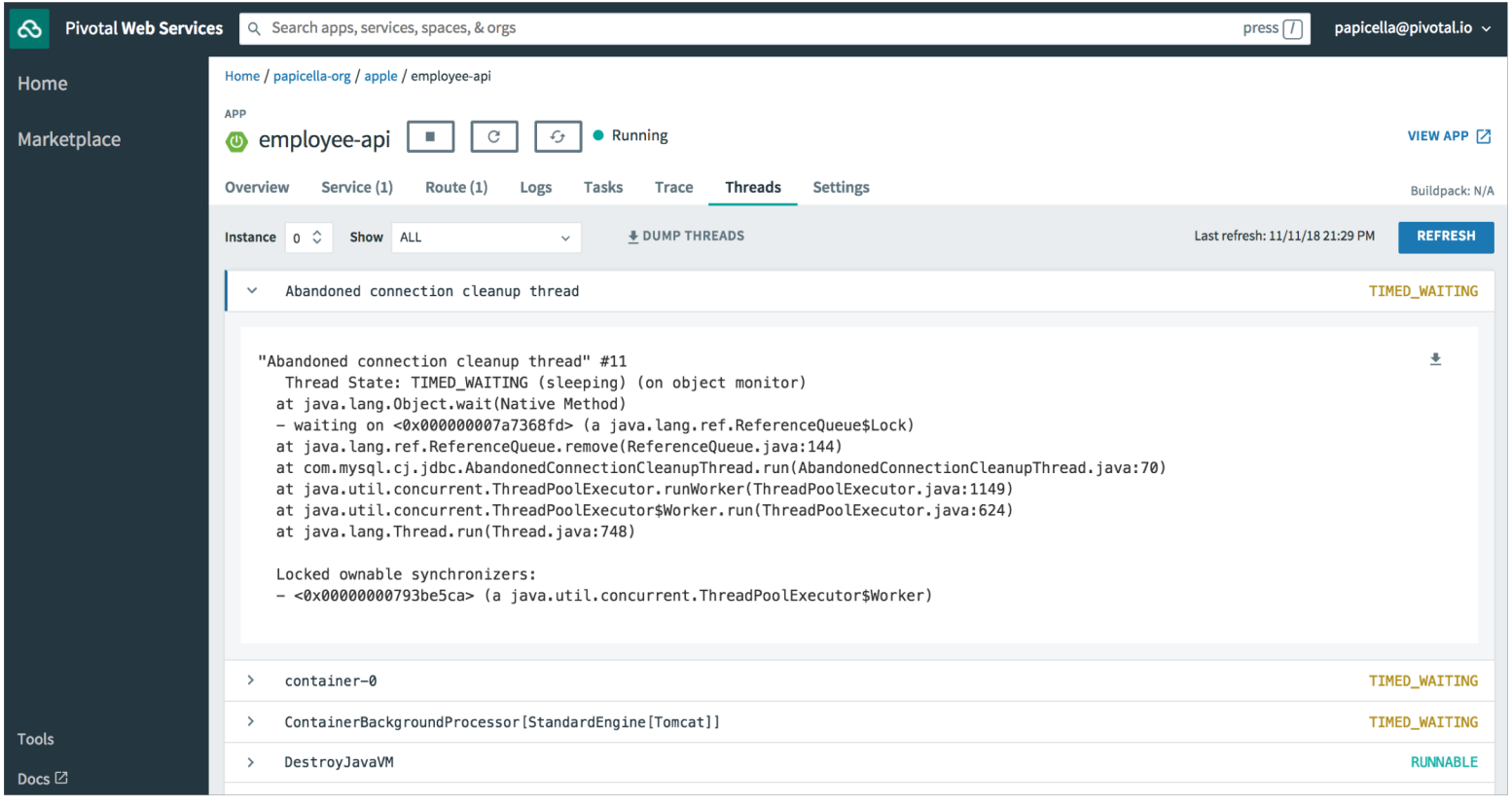
Spring Boot To PCF Developer Workshop

- Here we can view a Health Check for example as shown below



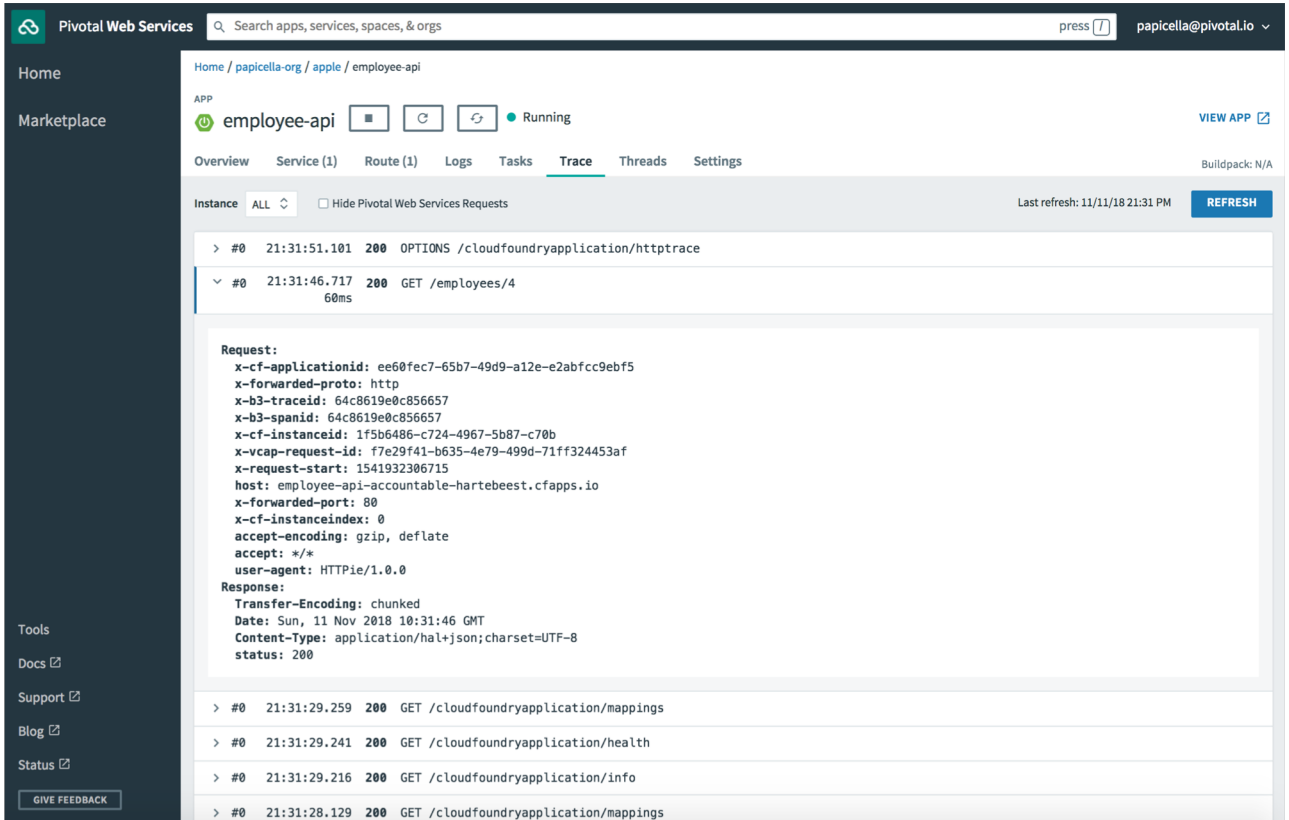
Spring Boot To PCF Developer Workshop

- Here we can view the application threads clicking on “Threads” as shown below



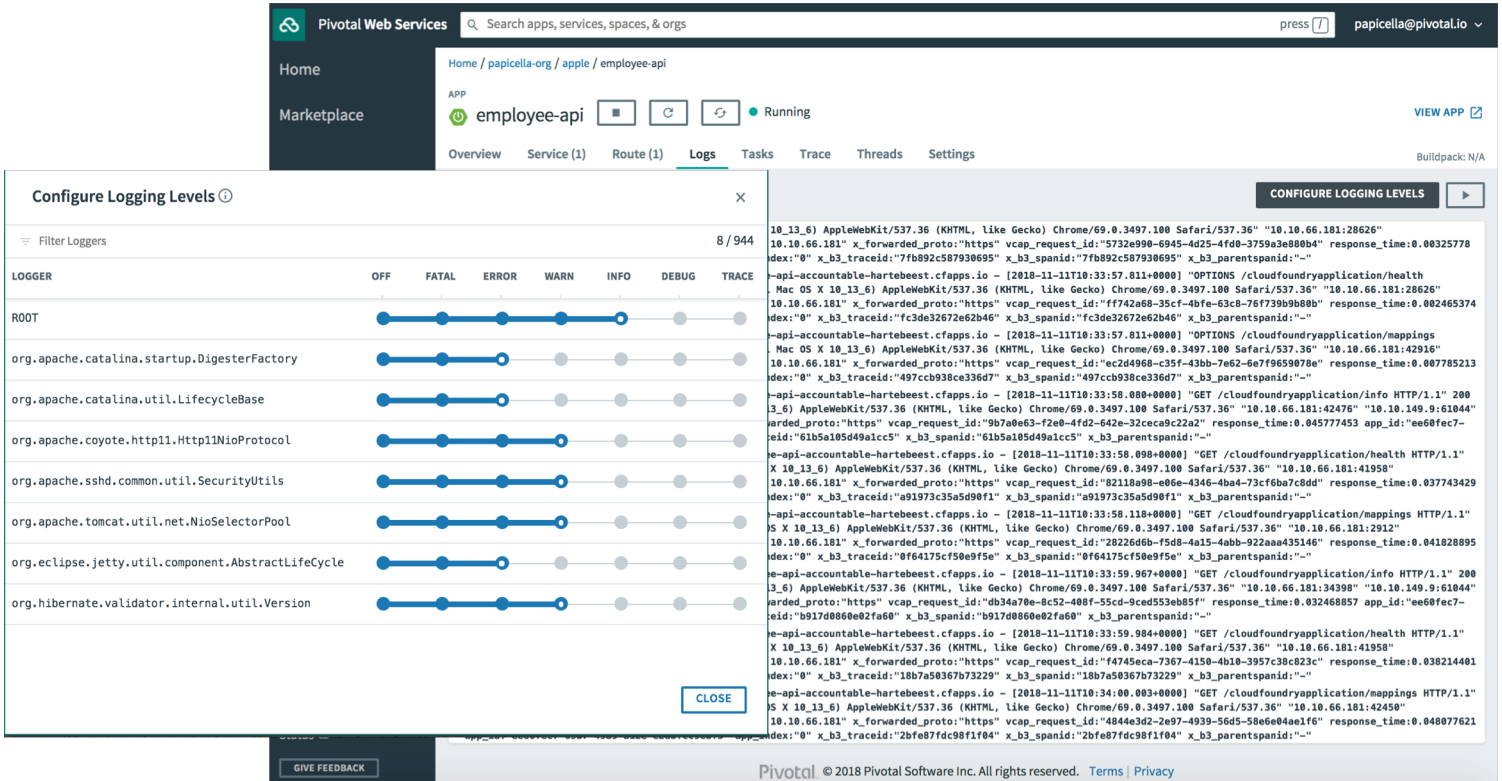
Spring Boot To PCF Developer Workshop

- Here we can view the trace information by clicking on “Traces” as shown below



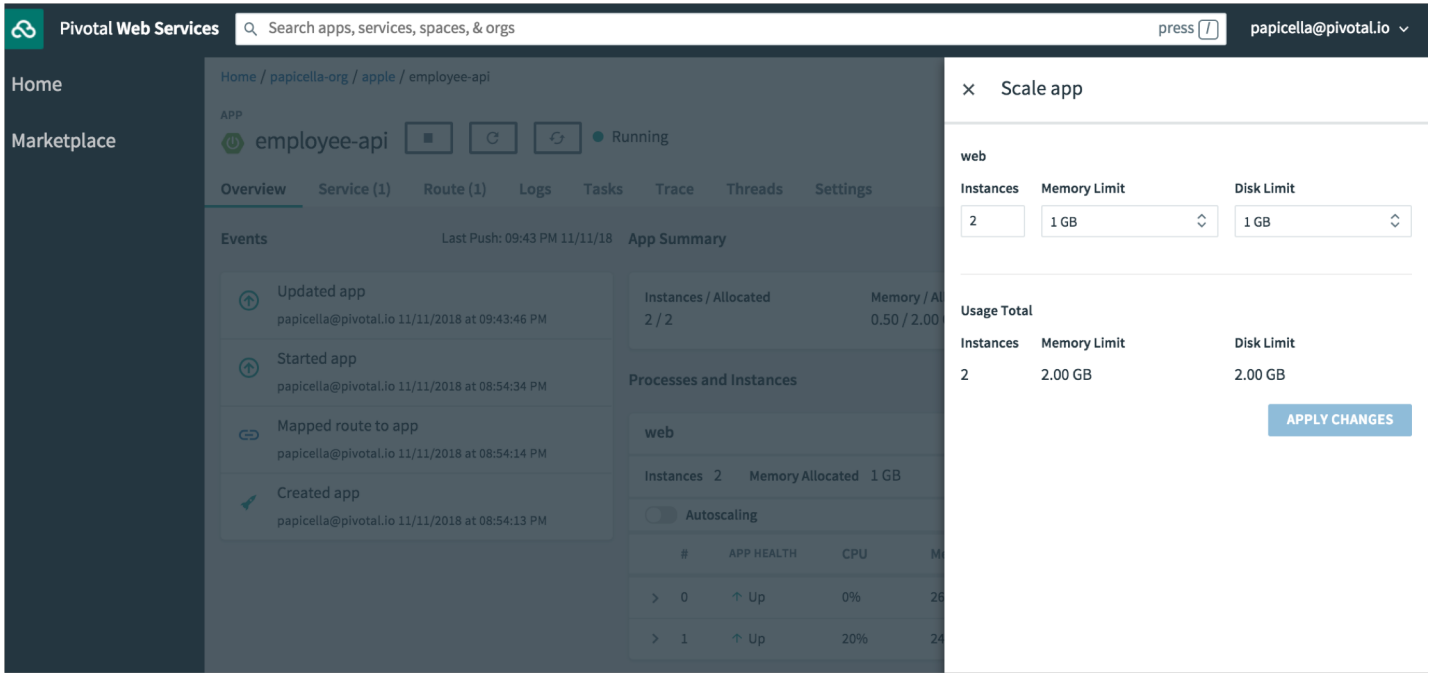
Spring Boot To PCF Developer Workshop

- Click on the logs tab as shown below to view logs and we can also configure Log Levels without re-starting the application



Spring Boot To PCF Developer Workshop

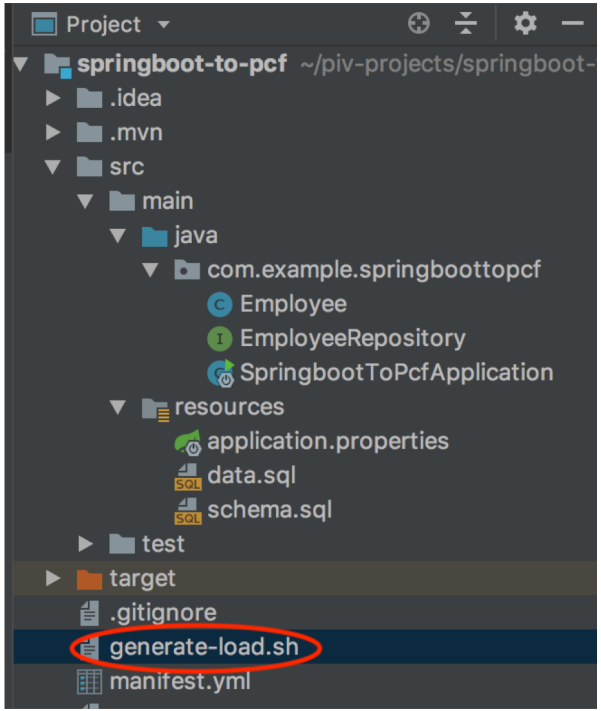
- Let's scale our application by adding a second instance either using Applications Manager UI or from the command line using
  - $ cf scale employee-api -i 2



Spring Boot To PCF Developer Workshop

- Let's generate some load on our application so we can show how we monitor applications on PCF. In your project create a file called "generate-load.sh" with contents as shown below. If you don't have HTTPie installed use "curl" instead

```
#!/bin/bash
while true
do
  echo "Press [CTRL+C] to stop.."
      http http://$1/employees
  sleep 2
done
```



Spring Boot To PCF Developer Workshop

- Start your script by ensuring that it's executed and you pass in the URL of your employee-api micro service

```
$ cf apps | grep employee-api

employee-api       started          2/2         1G       1G     employee-api-accountable-hartebeest.cfapps.io

$ chmod +x generate-load.sh

$ ./generate-load.sh employee-api-accountable-hartebeest.cfapps.io
```

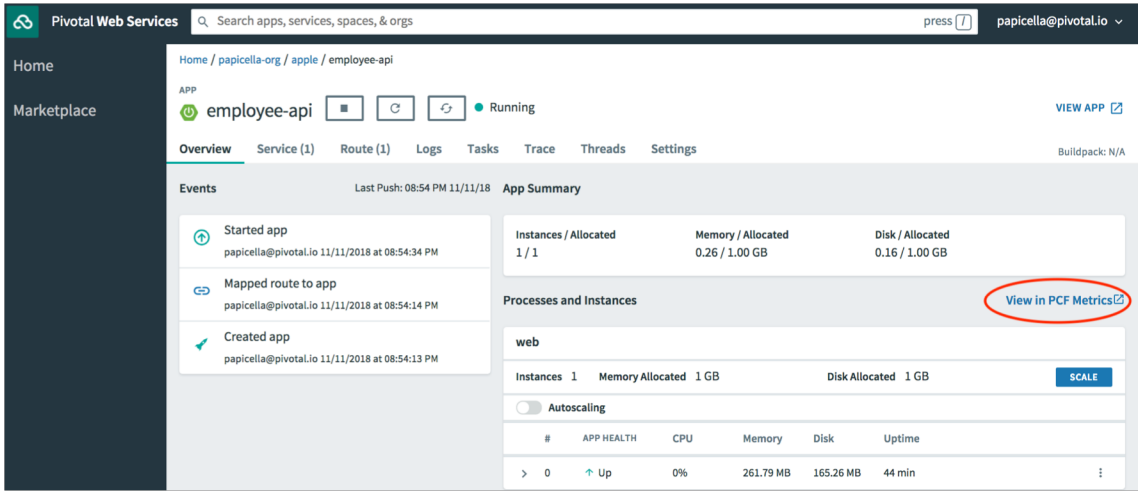
This will just call our "employee" RESTful endpoint over and over again until we use CNTRL-C. Leave it running as we need some application activity for the next part of this workshop

Spring Boot To PCF Developer Workshop

Built in Metrics on PCF can be accessed from the application home page as shown to the right. Click on “View in PCF Metrics” link

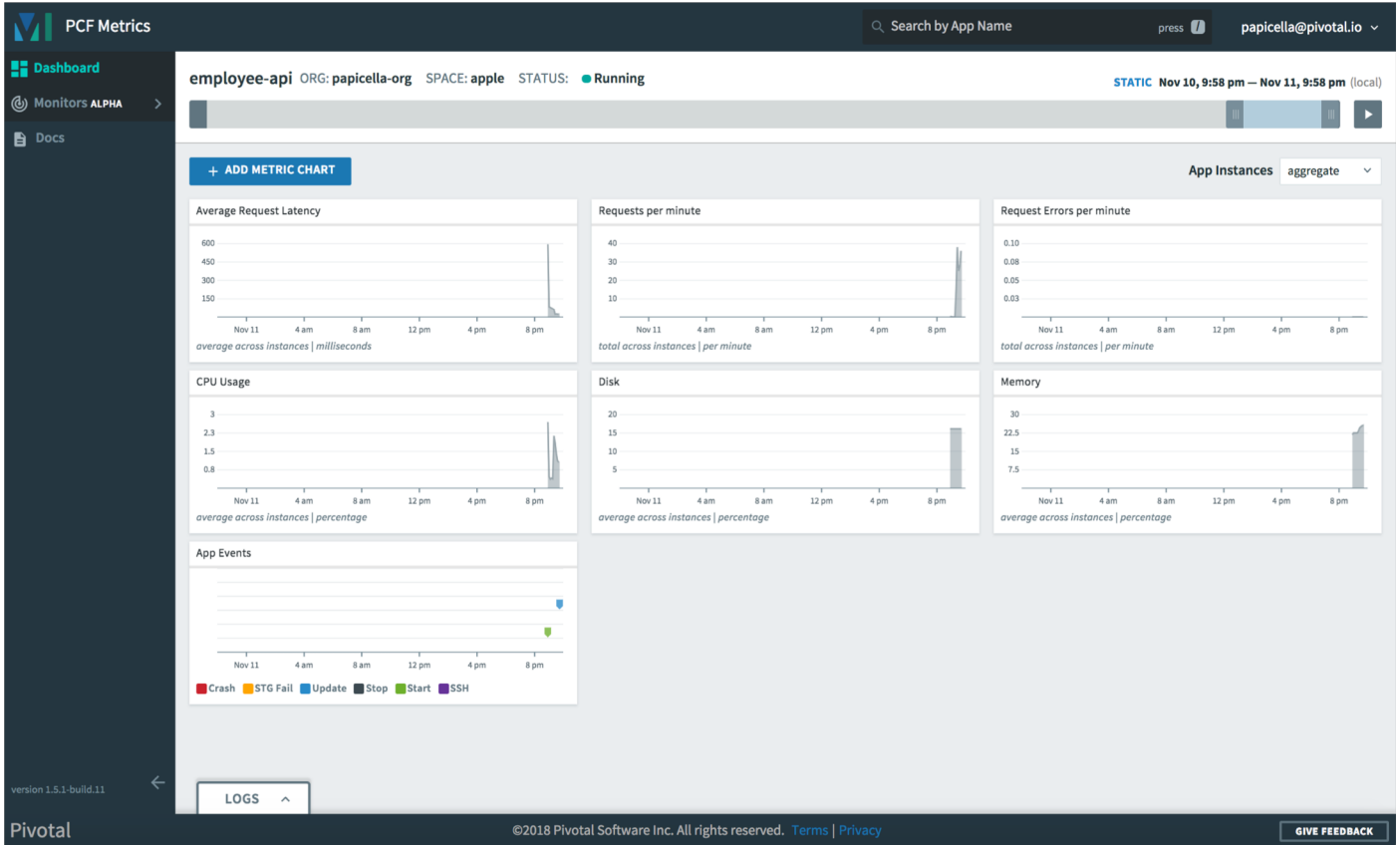
- Metrics include:
    - HTTP requests and errors
    - Avg. response latency
    - CPU, memory and disk
    - App events like start, stop, scale, update, and crash
    - Application logs
    - Request tracing

All without any agent, and no application configuration!



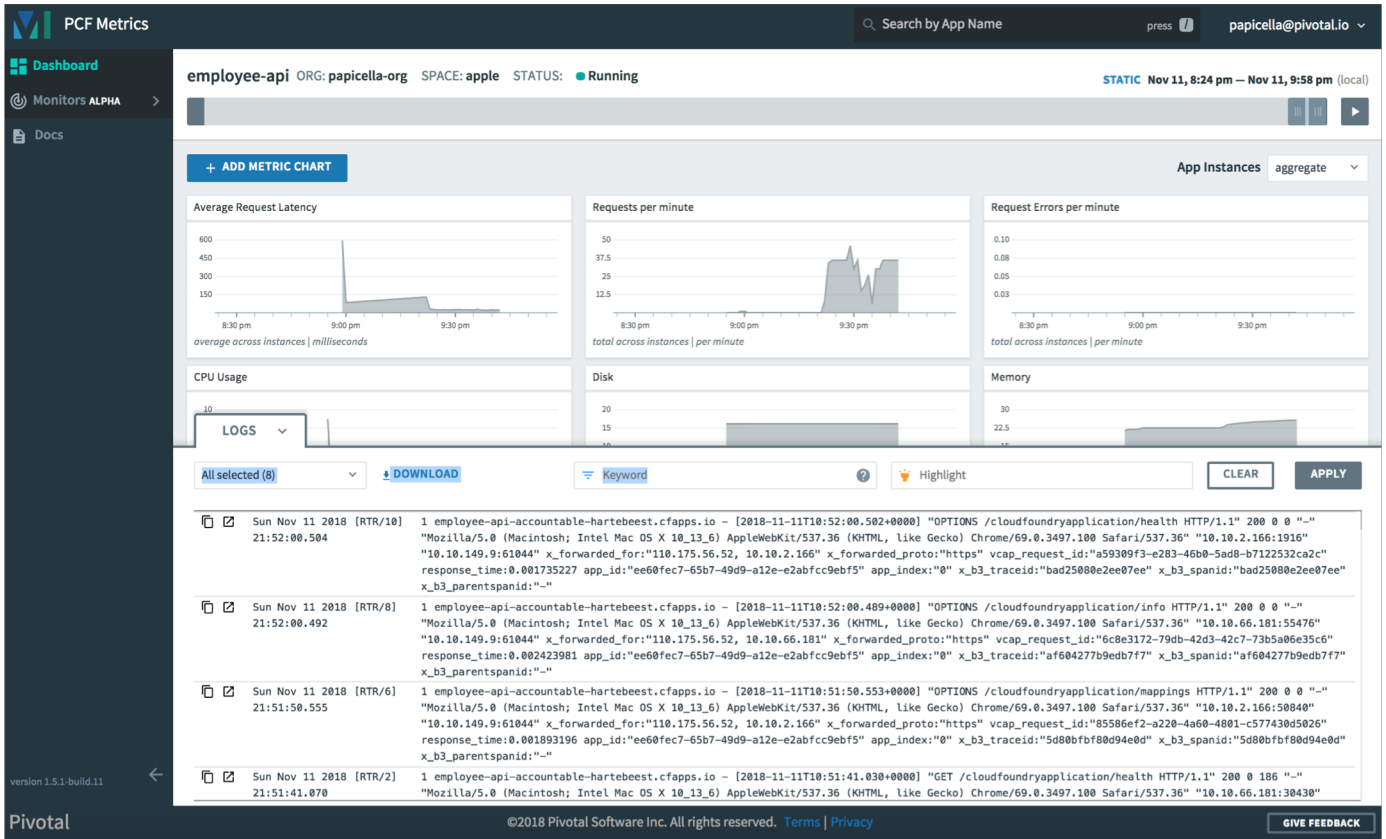
Spring Boot To PCF Developer Workshop

- We should see some activity on the graphs as shown below



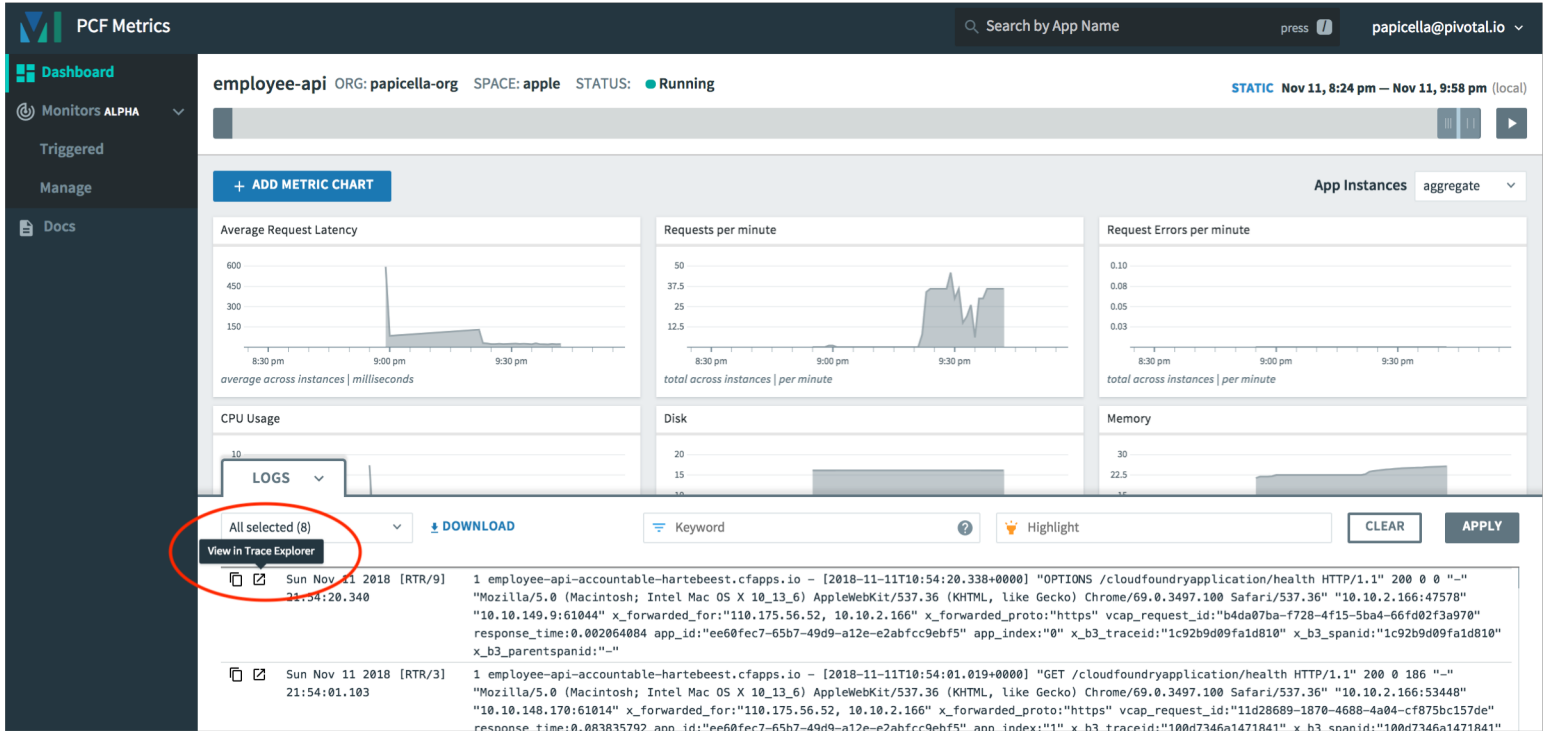
Spring Boot To PCF Developer Workshop

- Click on the “Logs” tab to verify the logs at any point in time on the graph



Spring Boot To PCF Developer Workshop

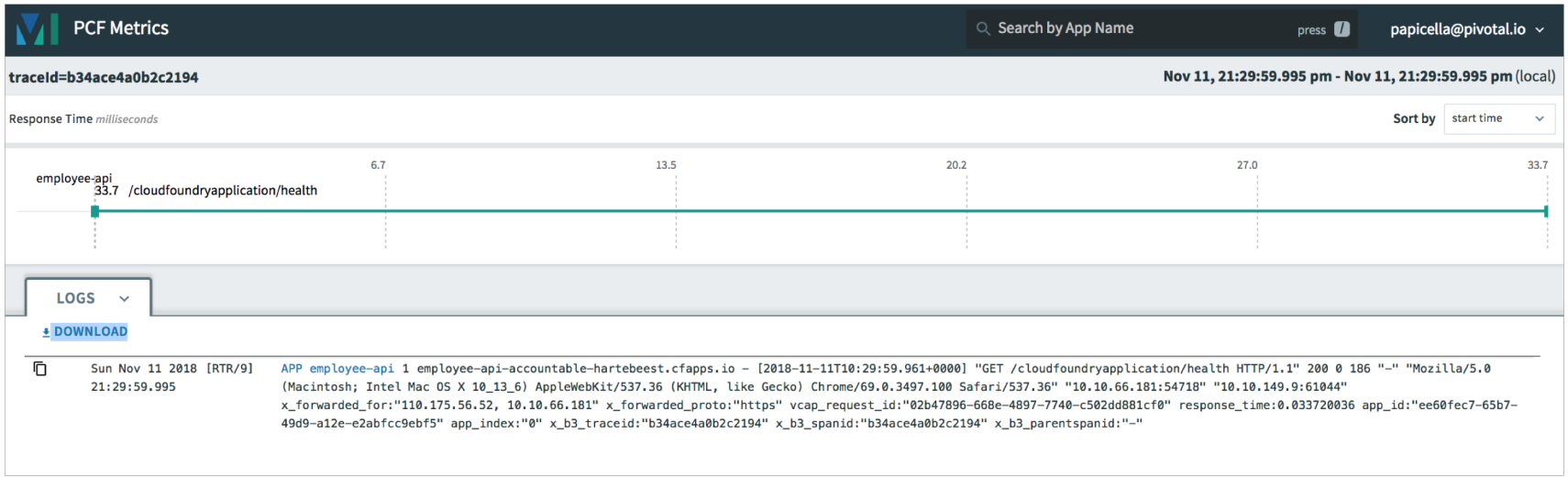
- We can even trace micro service calls using the “Trace Explorer” icon on a specific log entry



Spring Boot To PCF Developer Workshop

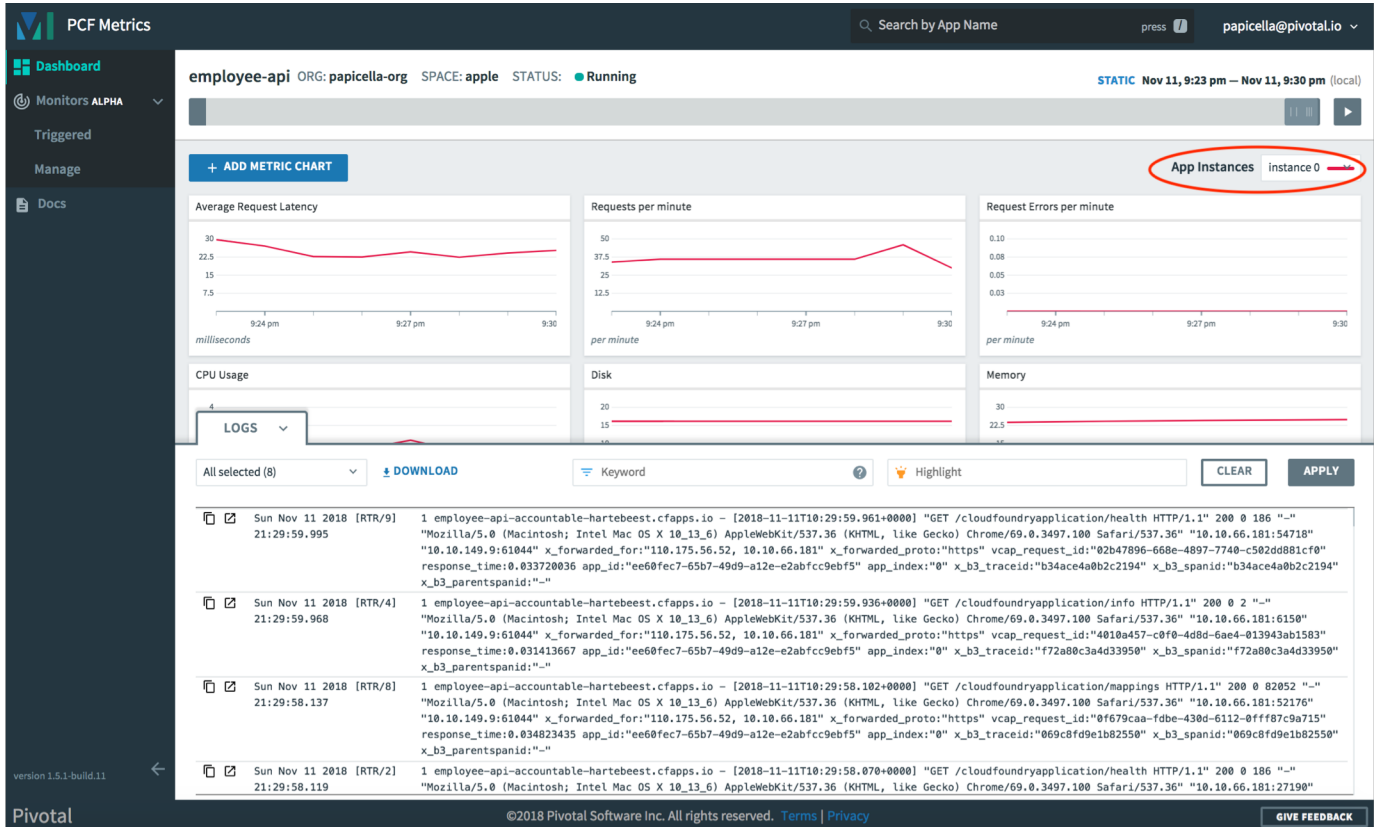
- To use the “Trace Explorer” from Spring Boot you need to ensure your using the dependency “org.springframework.cloud:spring-cloud-starter-sleuth”. The Trace Explorer displays the apps and endpoints involved in completing a request, along with the corresponding logs

- For more information refer to this link
  - https://docs.pivotal.io/pcf-metrics/1-5/using.html#trace



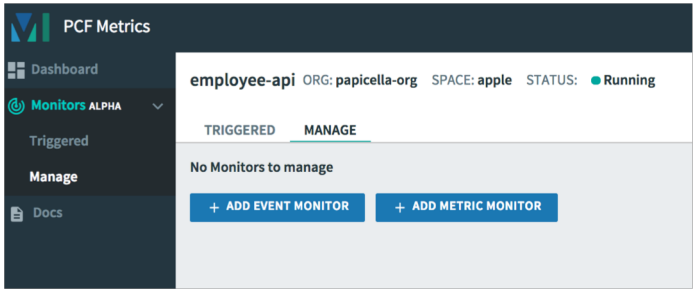
Spring Boot To PCF Developer Workshop

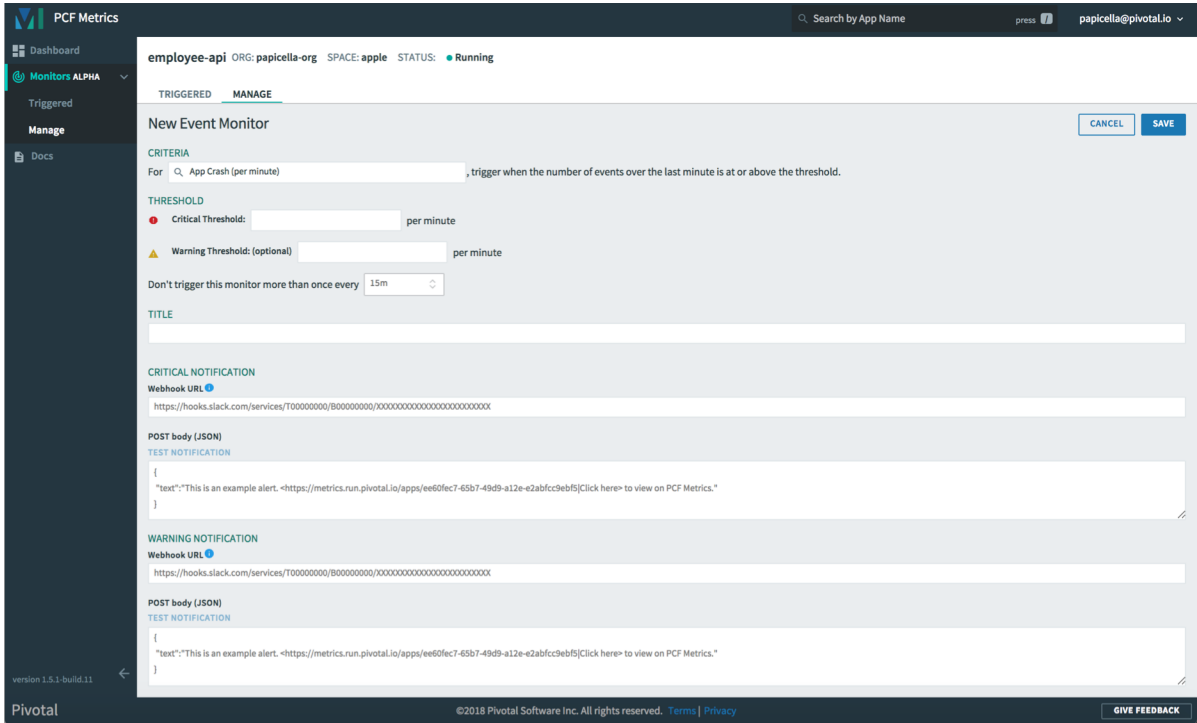
- You can view an application instance itself as shown below



Spring Boot To PCF Developer Workshop

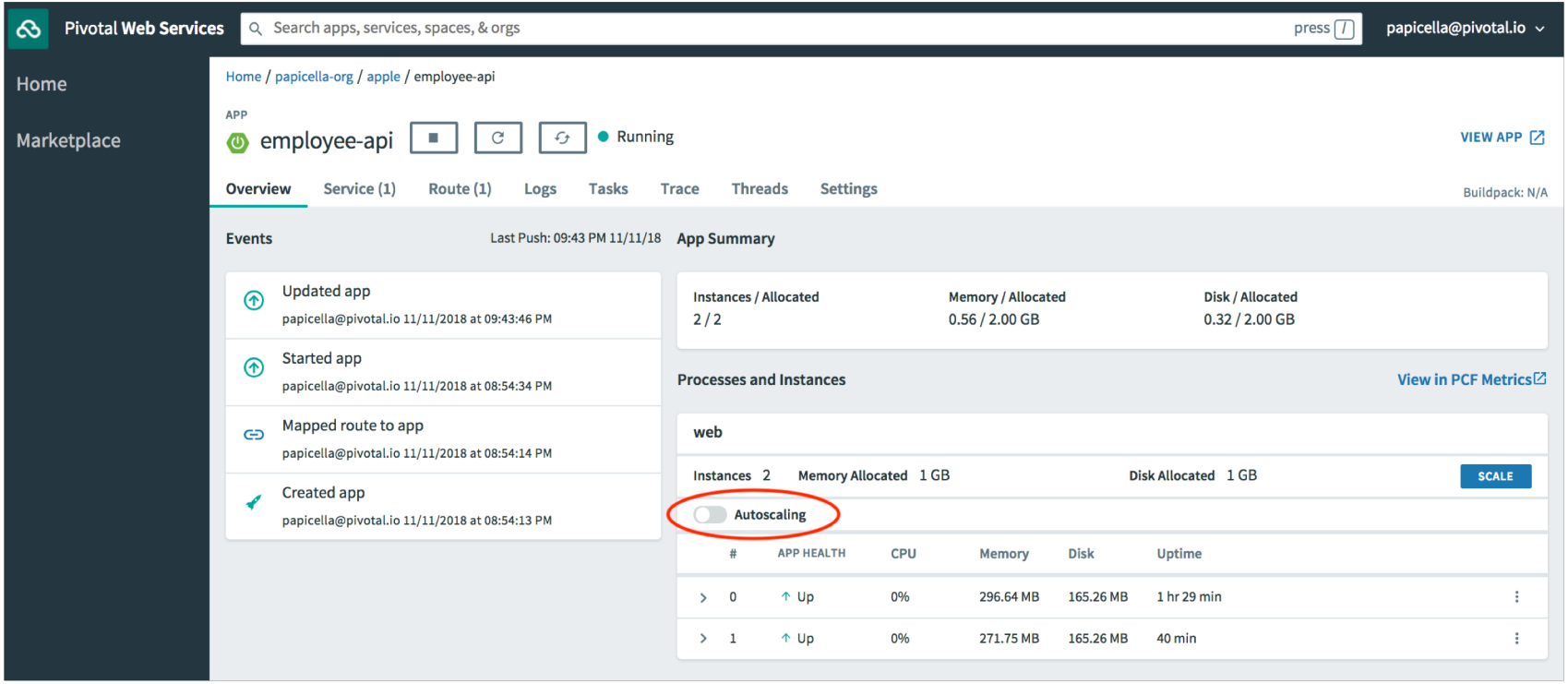
- Currently in ALPHA we will soon have the ability to add an “Event Monitor” or “Metric Monitor” as shown below. At this point we can use CNTRL-C to stop our script which is constantly hitting our micro service





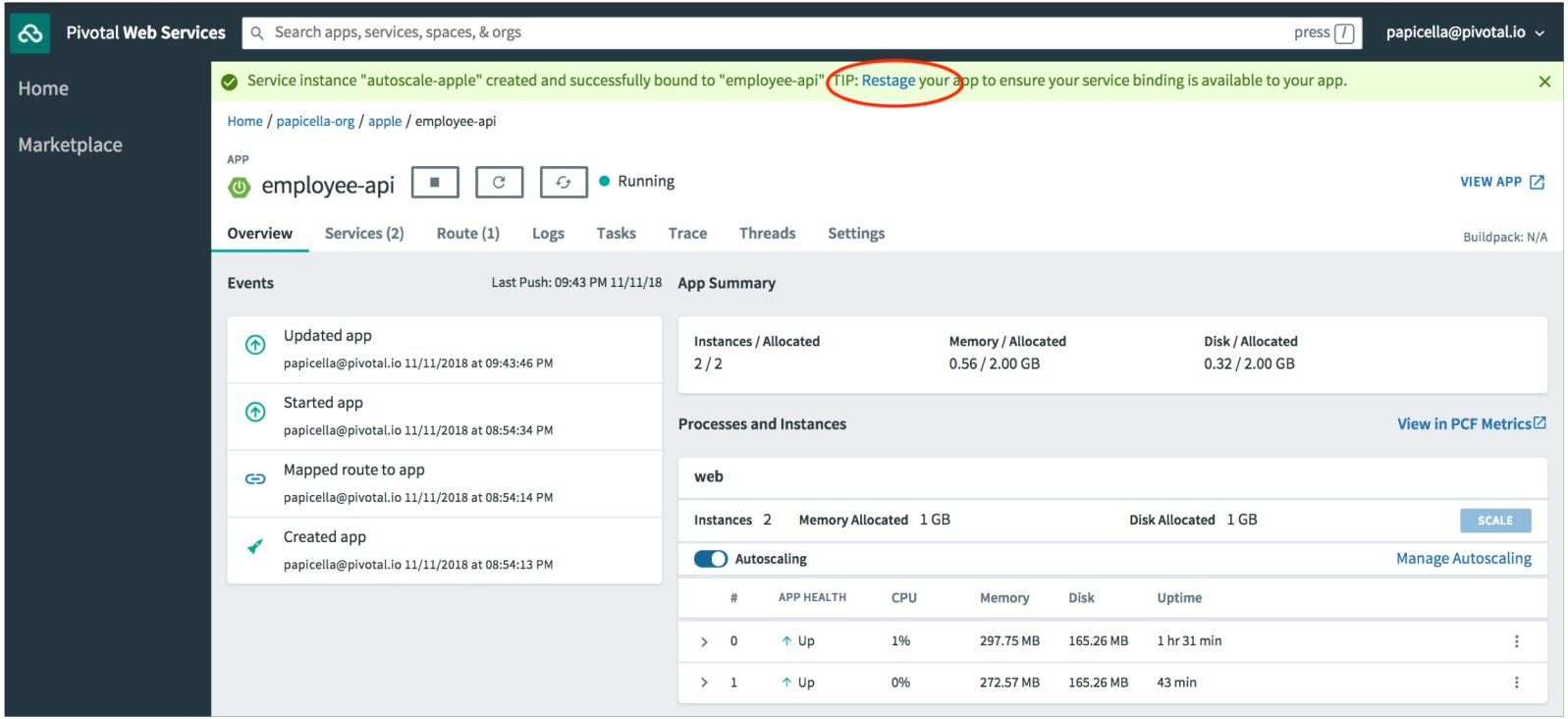
Spring Boot To PCF Developer Workshop

- Let's add a scaling policy for our application so we can scale up / down as required. Start by clicking on the "Autoscaling" radio checkbox to add the "Auto Scaler" service and bind it to out applcation



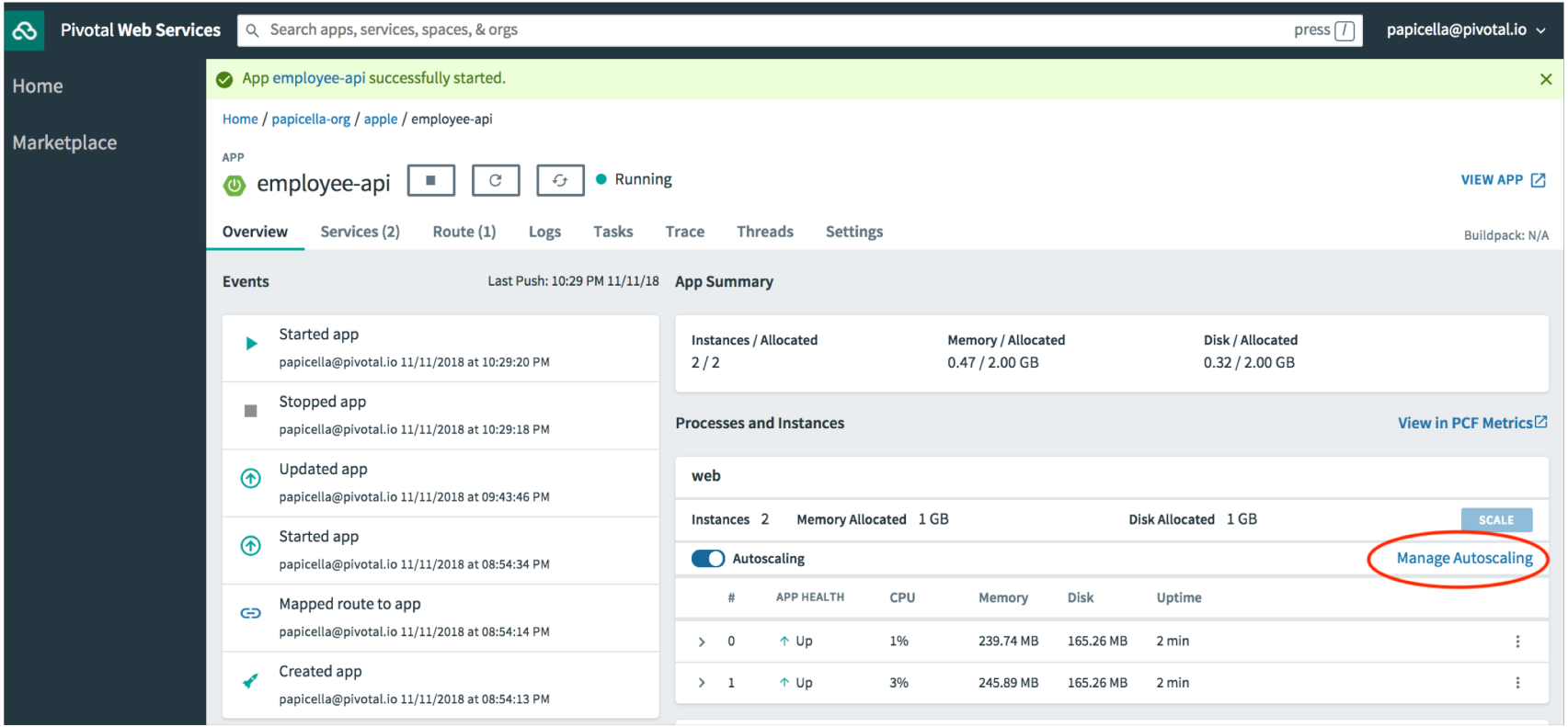
Spring Boot To PCF Developer Workshop

- Now we will need to re-stage the application to pick up the service details and inject those into the application



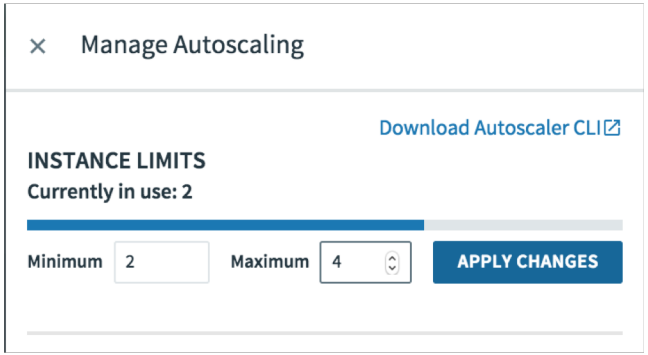
Spring Boot To PCF Developer Workshop

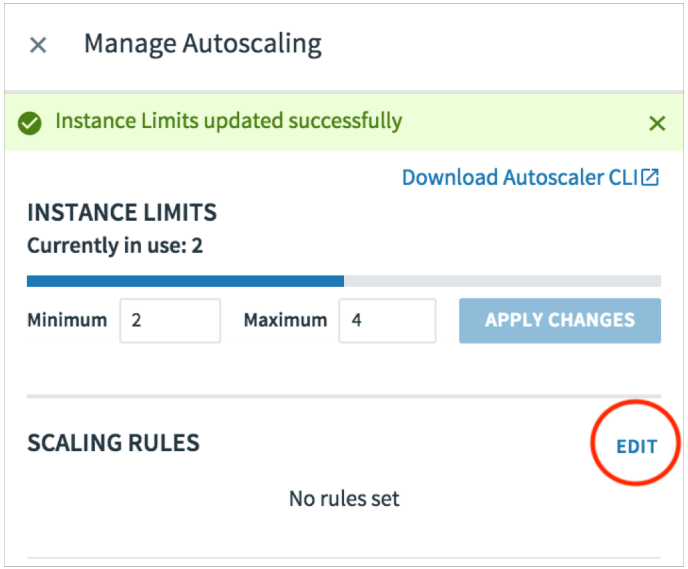
- Now click on “Manage Autoscaling” link as shown below



Spring Boot To PCF Developer Workshop

- Set the maximum instance limit to “4” and then “Apply Changes”. Once that is done click on the “Edit” link to add a scaling rule

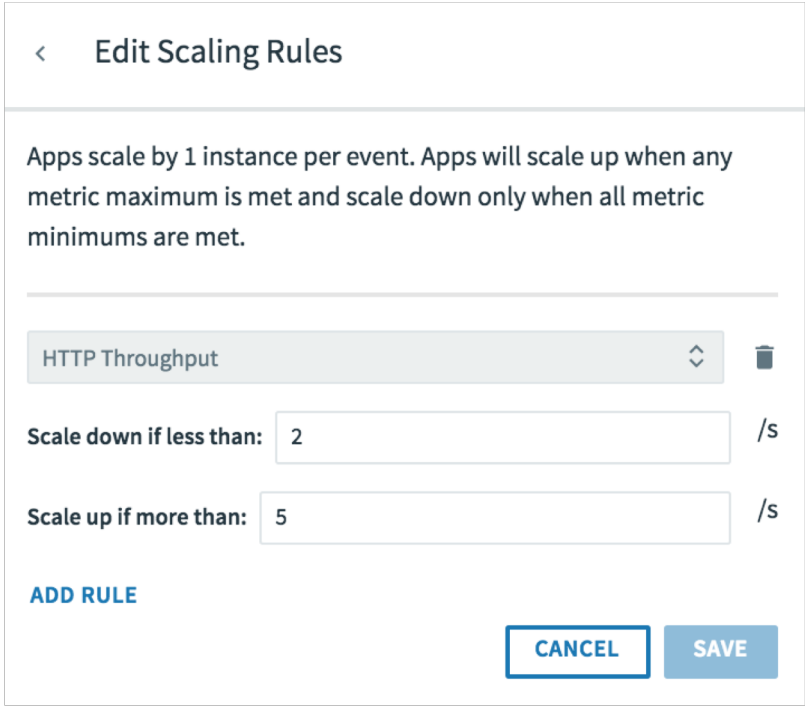


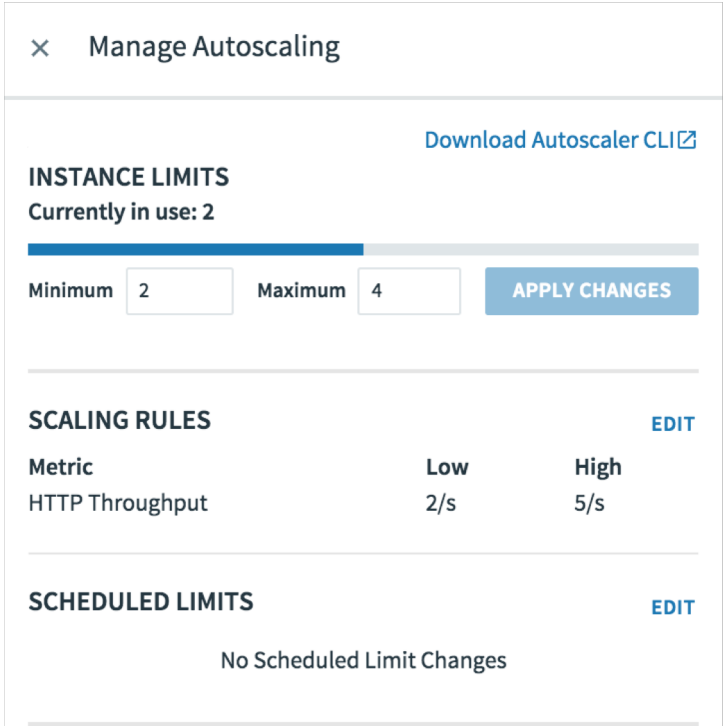


- Full example exists here
  - http://theblasfrompas.blogspot.com/2018/04/usingverifying-autoscale-service-from.html

Spring Boot To PCF Developer Workshop

- Add a Scaling rule as shown below





Spring Boot To PCF Developer Workshop

- The Auto Scaler has a CF CLI Plugin which you can find out more details about below as well as install it as shown in the link below.
    - https://docs.run.pivotal.io/appsman-services/autoscaler/using-autoscaler-cli.html#install

```
pasapicella@pas-macbook:~$ cf plugins
Listing installed plugins...

plugin                version   command name                 command help
App Autoscaler        1.0.66    autoscaling-apps             Displays apps bound to the autoscaler
App Autoscaler        1.0.66    autoscaling-events           Displays previous autoscaling events for the app
App Autoscaler        1.0.66    autoscaling-rules            Displays rules for an autoscaled app
App Autoscaler        1.0.66    configure-autoscaling        Configures autoscaling using a manifest file
App Autoscaler        1.0.66    create-autoscaling-rule      Create rule for an autoscaled app
App Autoscaler        1.0.66    delete-autoscaling-rule      Delete rule for an autoscaled app
App Autoscaler        1.0.66    delete-autoscaling-rules     Delete all rules for an autoscaled app
App Autoscaler        1.0.66    disable-autoscaling          Disables autoscaling for the app
App Autoscaler        1.0.66    enable-autoscaling           Enables autoscaling for the app
App Autoscaler        1.0.66    update-autoscaling-limits    Updates autoscaling instance limits for the app
```

Spring Boot To PCF Developer Workshop

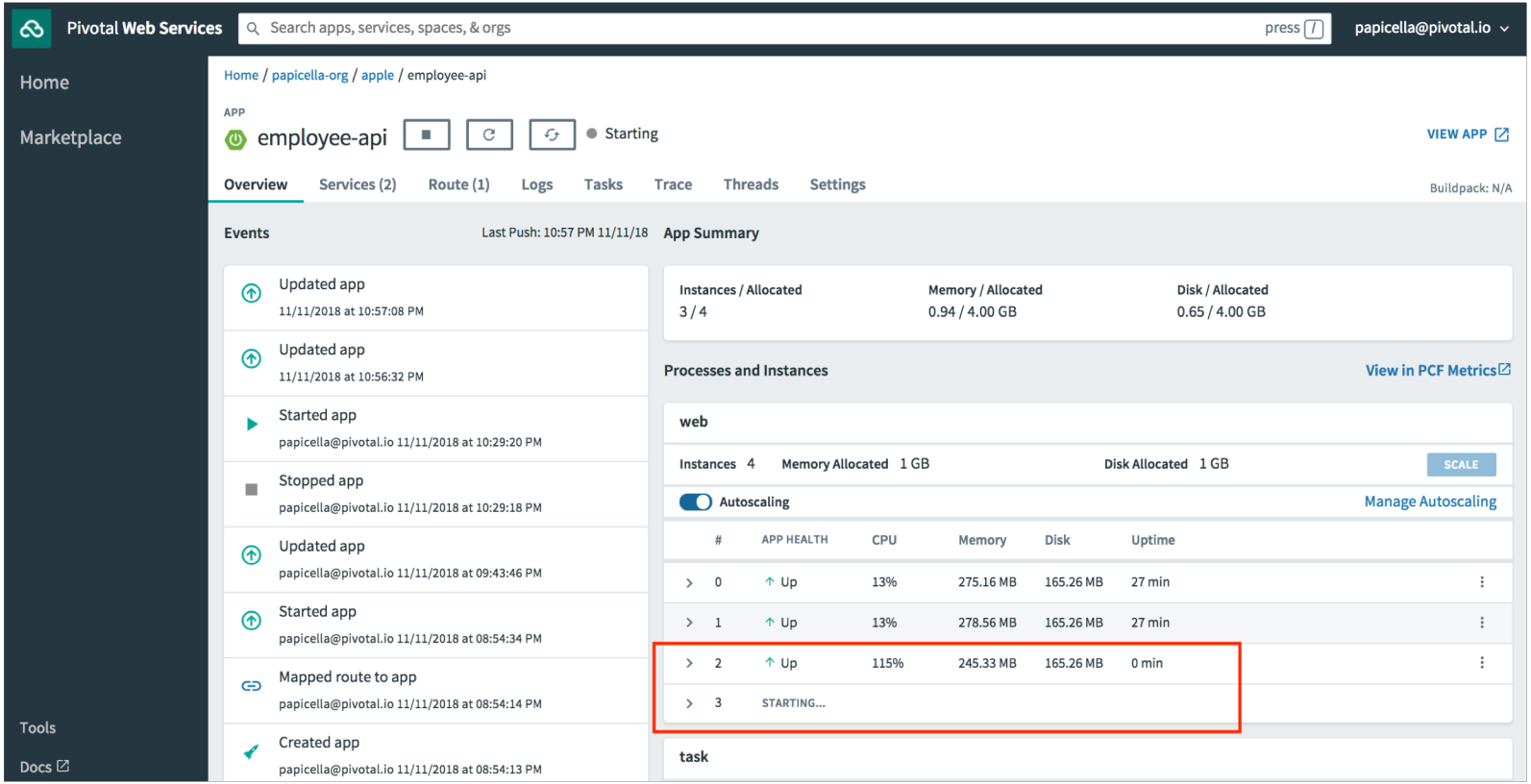
- To be able to test our Auto Scaling service we will need to generate some load to generate enough HTTP Throughput to force the Auto Scaler service to create new instances. Few ways to do that here is two of them

  - Use Apache JMeter (https://jmeter.apache.org/)
  - Use “ab” from your MacBook pro or Linux file system
    - $ ab -n 10000 -c 25 http://{URL}/employees

```
pasapicella@pas-macbook:~$ ab -n 10000 -c 25 http://employee-api-accountable-hartebeest.cfapps.io/employees
This is ApacheBench, Version 2.3 <$Revision: 1826891 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking employee-api-accountable-hartebeest.cfapps.io (be patient)
Completed 1000 requests
Completed 2000 requests
Completed 3000 requests
Completed 4000 requests
Completed 5000 requests
```

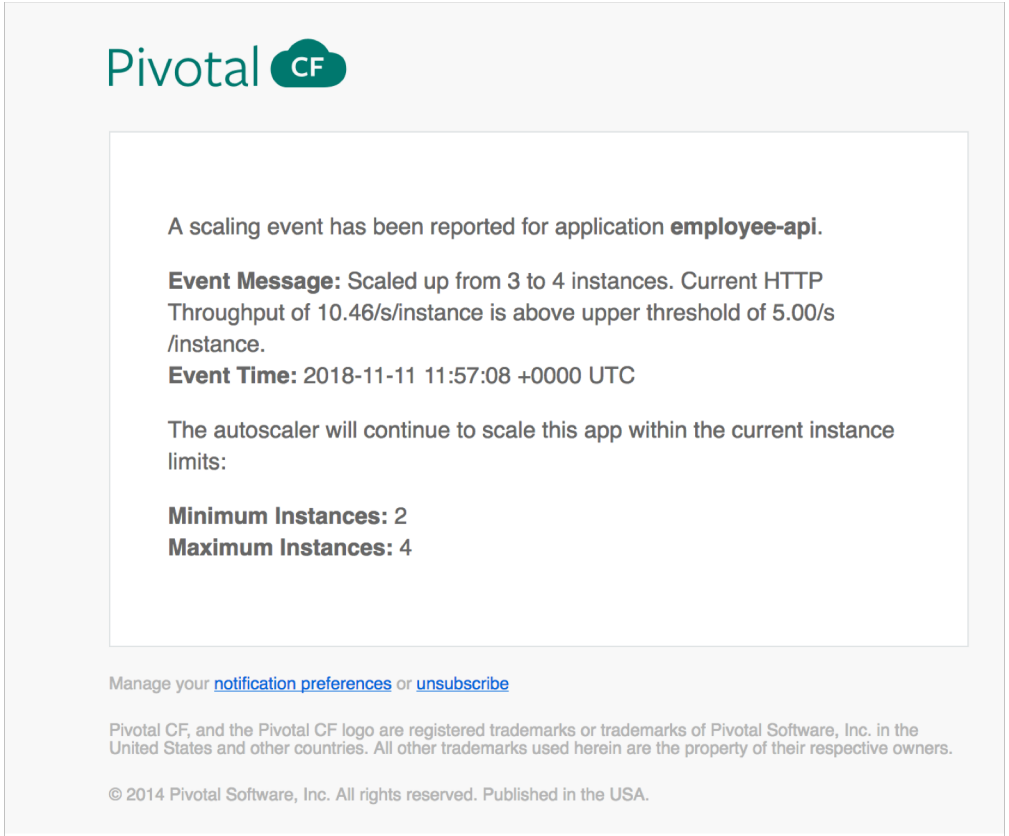
Spring Boot To PCF Developer Workshop

- It won't take long before we see the Auto Scaler service kick in as shown below



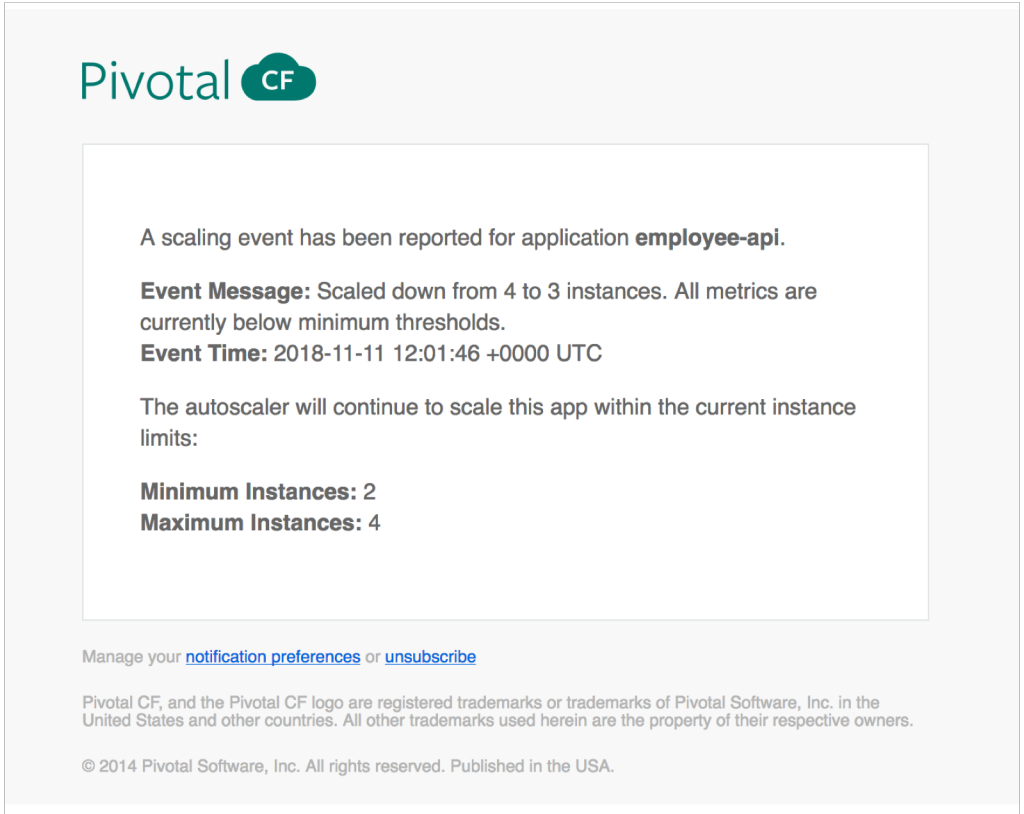
Spring Boot To PCF Developer Workshop

- In fact the Auto Scaler service even emails you!!!



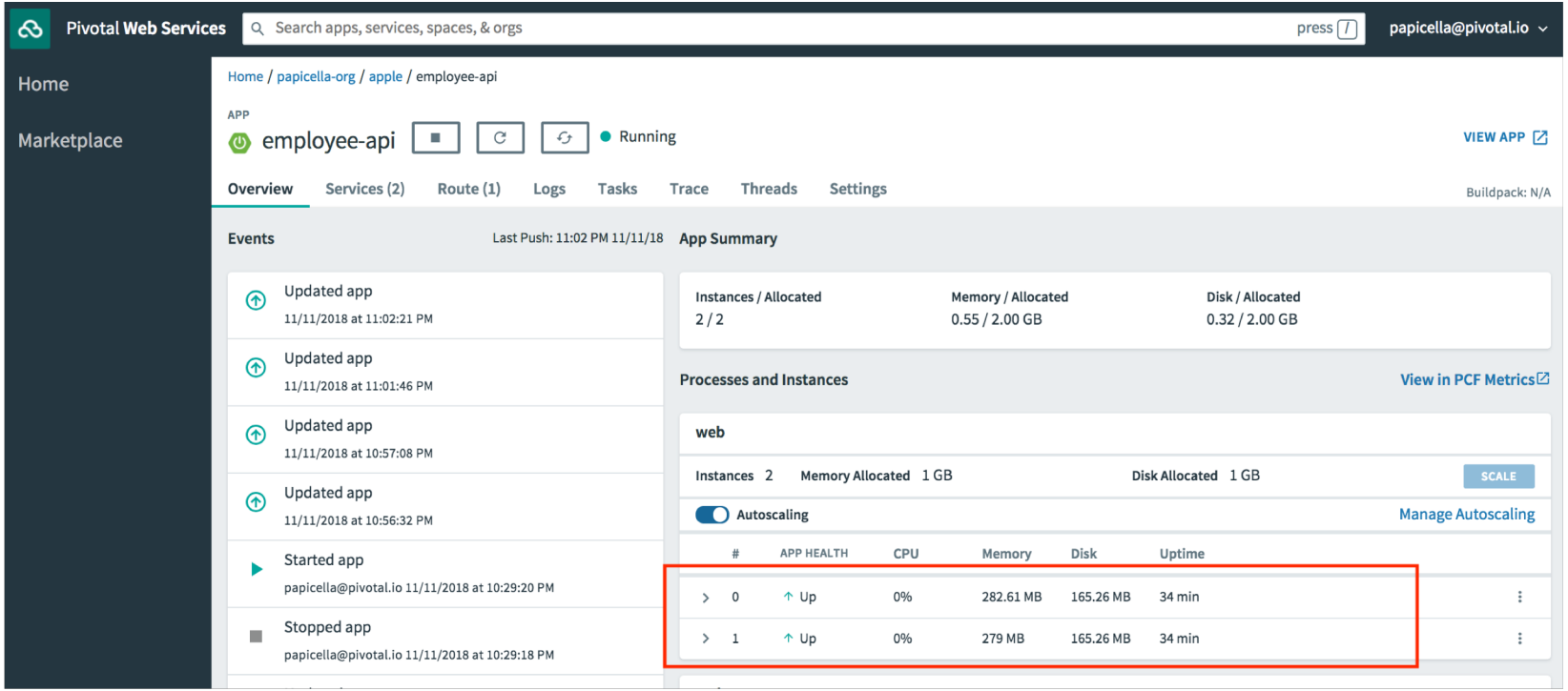
Spring Boot To PCF Developer Workshop

- In fact the Auto Scaler service even emails you when it starts to scale back down as well



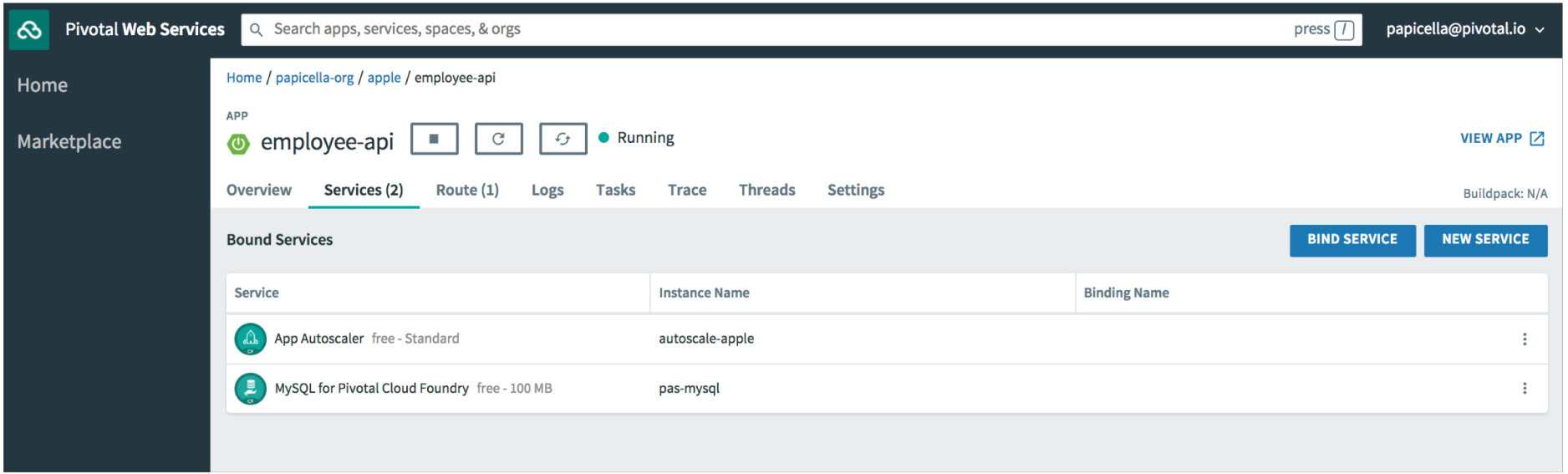
Spring Boot To PCF Developer Workshop

- Finally back to normal after all our load test has ended!!!



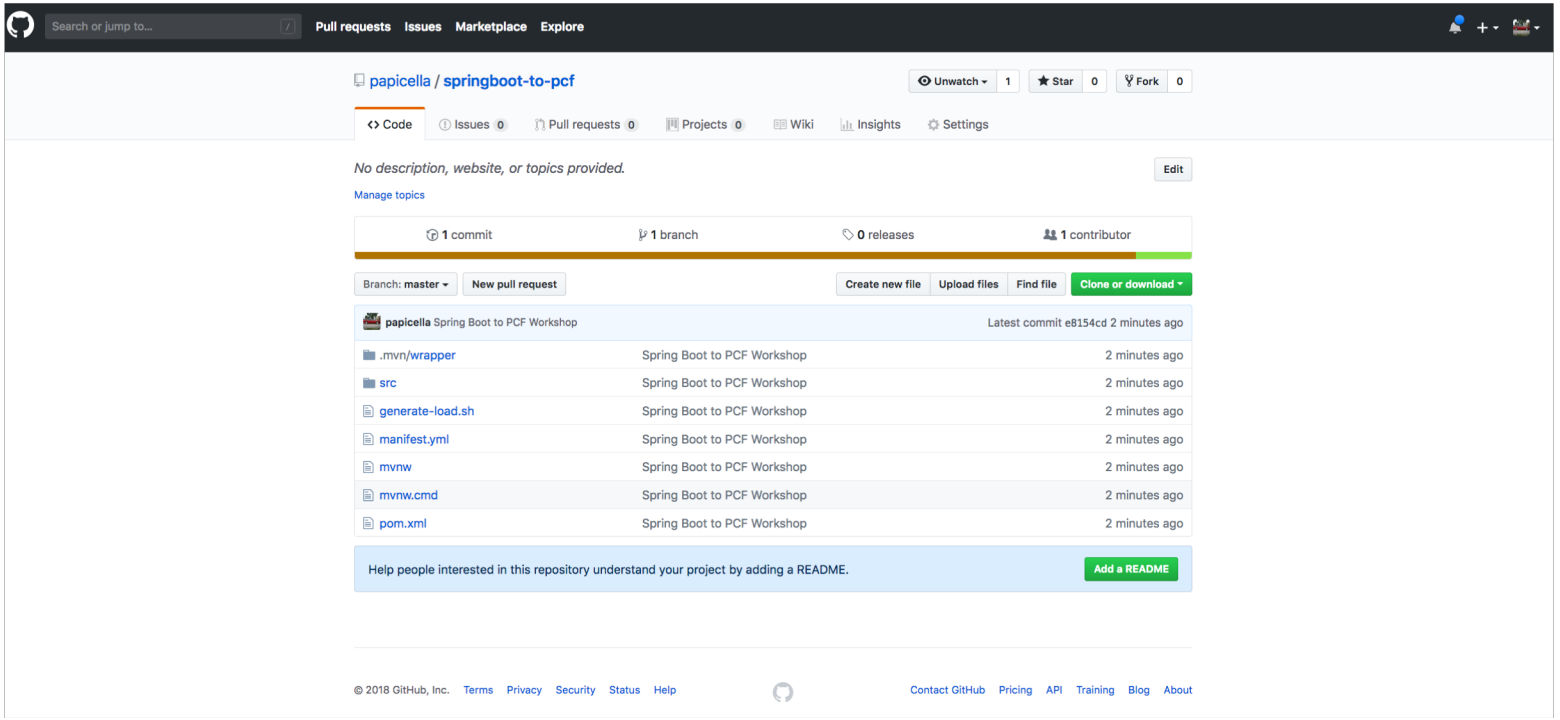
Spring Boot To PCF Developer Workshop

- From the applications Manager UI you will see we have two services bound to the “employee-api” application as the Auto Scaler service is a service itself



Spring Boot To PCF Developer Workshop

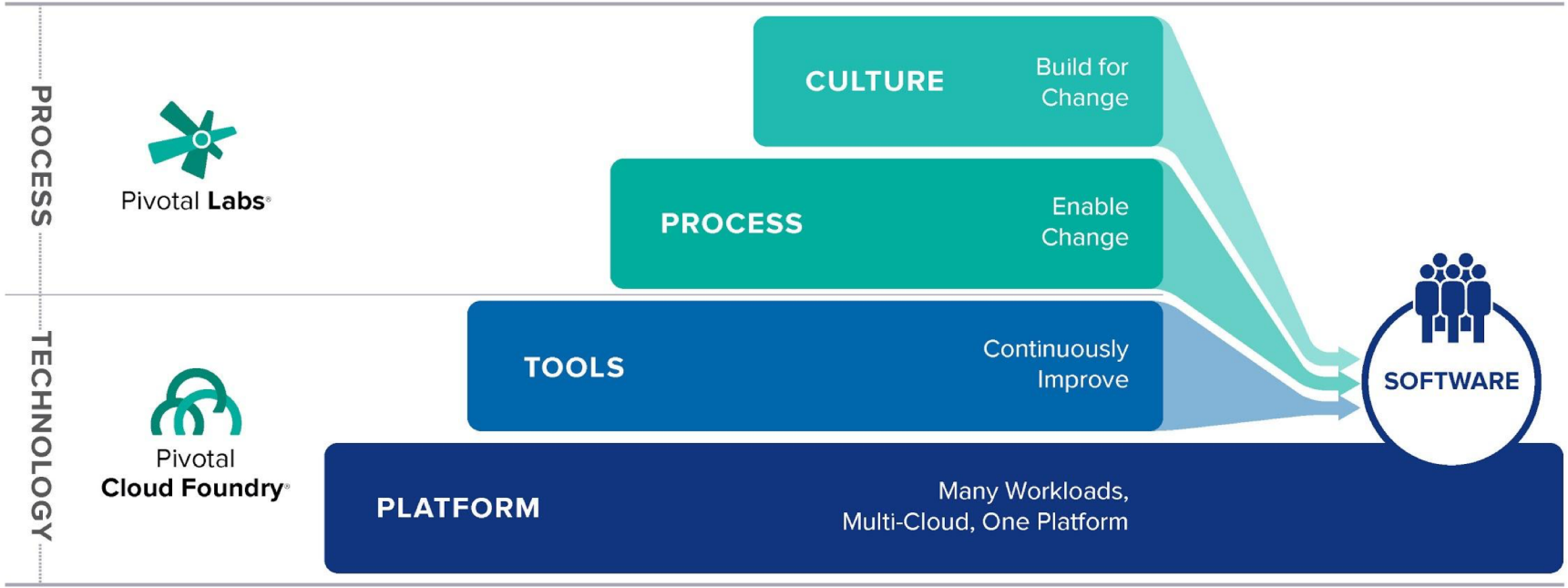
- Had trouble along the way? The whole source code is on GitHub
  - https://github.com/papicella/springboot-to-pcf



Spring Boot to PCF Developer Workshop

Summary

Pivotal Enables Cloud-Native Transformation



Spring Boot to PCF Developer Workshop

Questions?

Transforming How The World Builds Software