CLOUDFOUNDRY

Pivotal Cloud Foundry Developer

Lab Instructions

Application Deployment using Pivotal Cloud Foundry

Version 1.11.a

Pivotal

Copyright Notice

Pivotal

Table of Contents

Chapter 1. Service Broker

Estimated Time: 60 minutes

1.1. Preface

Cloudfoundry is an extensible platform. Managed services can easily be plugged in and made available in the services marketplace. In this lab, we'll use a feature known as *space-scoped service brokers* to register a custom managed service that provisions Mongodb databases for applications.

You'll deploy an application that implements the service broker API, register it as a service broker in Cloudfoundry, and then put it to use by instantiating a mongodb instance (with `create-service`), and attaching that resource (with `bind-service`) to an application that will use it to persist album information.

1.2. Exercises

1.2.1. Setup

1. The code for our service broker is in your lab files in `cloudfoundry-mongodb-service-broker-0.3.zip`. If you have it, skip to (3)/
2. Alternatively, download the following zip file, which contains the source code and precompiled jar file, ready for you to deploy (no building necessary). Copy the file to your `pivotal-cloud-foundry-developer-workshop` folder.
3. Extract the the zip file to … `/pivotal-cloud-foundry-developer-workshop/cloudfoundry-mongodb-service-broker`.
4. OPTIONAL STEP - Import applications into your IDE such as Spring Tool Suite (STS).

STS Import Help

1. Make sure the `Gradle Support` IDE extension is installed.
   a. To install `Gradle Support` select menu:Help[Dashboard]. This will open a new tab to the Dashboard.
   b. Click on "IDE Extensions" then search for "Gradle". If you can't find it or can't select it, it's already installed.

c. Select `Gradle Support` or `Gradle (STS Legacy) Support` and click the "Install" button.

d. Complete the installation wizard and restart STS.

2. Import the broker code

a. Select menu:File[Import…]

b. Then select Gradle -> Gradle Project.

c. On the Import Gradle Project page, browse to the directory where you extracted the zip.

d. Then push the "Build Model" button.

e. Select the project. Click "Finish".

1.2.2. Service Broker API Overview

1. Review the documentation. Specifically, the sequence diagram. This is what we will implement.

1.2.3. Create a MongoDB Service Broker

1.2.3.1. About this Broker

This broker is implemented with Spring Boot, and leverages the Cloud Foundry Service Broker project. This project reduces the effort required to write a service broker in Java to the implementation of a handful of Spring beans. This blog entry by Scott Frederick originally introduced this project to the Spring Cloud in the Summer of 2016.

1.2.3.2. Implement Catalog Management

1. Review the documentation on implementing catalog management.

2. We need to implement catalog management in our `mongodb-service-broker` application. Fortunately, all the Service Broker API endpoints have been mapped by the Spring Cloud Service Broker project. For instance, the https://github.com/spring-cloud/spring-cloud-cloudfoundry-service-broker/blob/master/src/main/java/org/springframework

3. We have an endpoint, but the Spring Cloud Service Broker can't provide all the implementation. We need to describe our catalog. To to that, all we need to do is provide a `Catalog` bean.

Review the following file:

…

`/pivotal-cloud-foundry-developer-workshop/cloudfoundry-mongodb-service-broker/src/main/java/org/springf`

```
@Configuration
public class CatalogConfig {

  @Bean
  public Catalog catalog() {
    return new Catalog(Collections.singletonList(
        new ServiceDefinition(
            getEnvOrDefault("SERVICE_ID","mongodb-service-broker"), //env variable
            getEnvOrDefault("SERVICE_NAME","MongoDB"), //env variable
            "A simple MongoDB service broker implementation",
            true,
            false,
            Collections.singletonList(
                new Plan(getEnvOrDefault("PLAN_ID","mongo-plan"), //env variable
                    "standard",
                    "This is a default mongo plan.  All services are created equally.",
                    getPlanMetadata(),
                    true)),
            Arrays.asList("mongodb", "document"),
            getServiceDefinitionMetadata(),
            null,
            null)));
  }
...
```

4. Push the `mongodb-service-broker` application.

```
cd .../pivotal-cloud-foundry-developer-workshop/cloudfoundry-mongodb-service-broker/
```

..and:

```
cf push mongodb-service-broker -p build/libs/cloudfoundry-mongodb-service-broker.jar -m 768M --random-route --no-start
```

5. Set environment variables.

   These environment variables get used by the broker to generate the catalog. These values should be unique across the entire Pivotal Cloud Foundry instance to meet the broker API specifications.

   As a convention, append your initials to where specified.

```
$ cf set-env mongodb-service-broker SERVICE_ID mongodb-service-broker-{{initials}}
$ cf set-env mongodb-service-broker SERVICE_NAME MongoDB-{{initials}}
$ cf set-env mongodb-service-broker PLAN_ID mongo-plan-{{initials}}
```

Note

You can safely ignore the "TIP: Use 'cf restage' to ensure your env variable changes take effect" message.

6. Start `mongodb-service-broker`

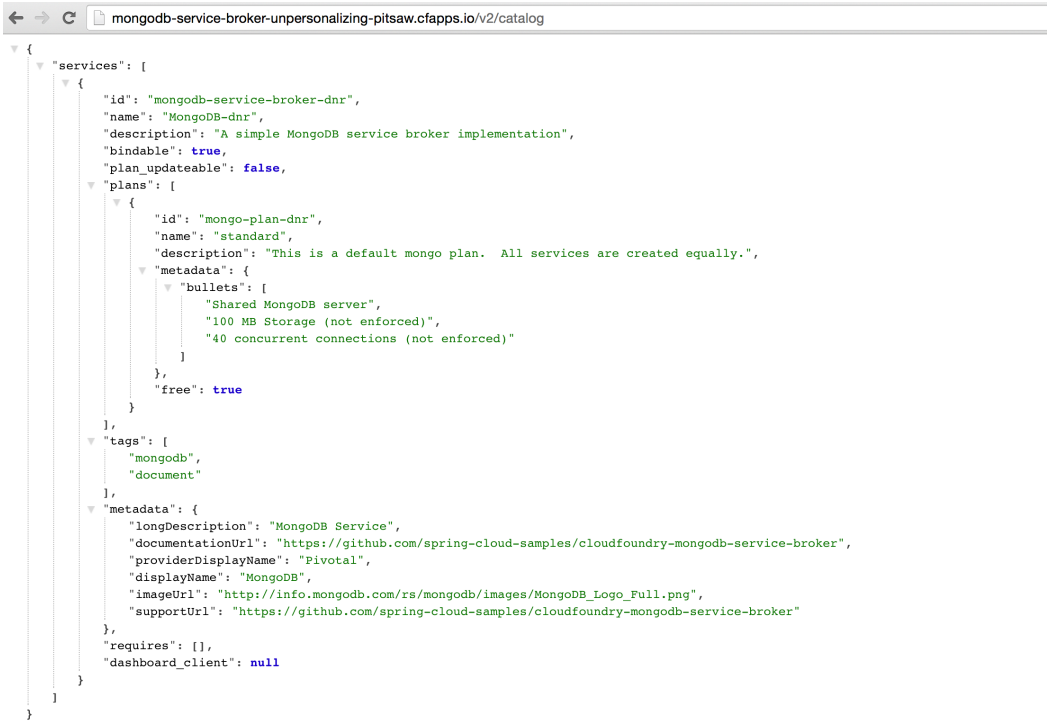

```
cf start mongodb-service-broker
```

7. Verify your work. Using an http client (e.g. a browser), visit the application's `v2/catalog` endpoint. Because the application is secured with Basic Auth you will need to provide credentials.

   Username: pivotal

   Password: keepitsimple

   You should see response similar to the following (pic is using the JSON Formatter for Chrome):



8. Register your Service Broker.

   We will be creating a Space-Scoped broker. Space-Scoped brokers help you during the development/testing of your service broker, because they are private to a space and don't require an `admin` to enable access (list it in the marketplace, provision service instances, etc).

A unique broker name is required. Use your initials.

```
cf create-service-broker mongodb-service-broker-{{initials}} pivotal keepitsimple {{service_broker_app_url}} --space-scoped
```

9. View the Service Brokers in your installation. You should see your new Service Broker.

```
cf service-brokers
```

10.Verify that your service is listed in the marketplace.

```
cf marketplace
```

Congratulations, you have implemented and tested the catalog endpoint in your service broker!

1.2.3.2.1. Questions

- Can a service broker support upgrade/downgrade of a service?

1.2.3.3. Implement Provisioning and Deprovisioning

1. Review the documentation on implementing provisioning and deprovisioning.
2. We need to implement provisioning/deprovisioning in our `mongodb-service-broker` application. To do so we just need to implement the ServiceInstanceService interface, because the Spring Cloud Service Broker project has already done the mapping.

Review the following file:

+ …
`/pivotal-cloud-foundry-developer-workshop/cloudfoundry-mongodb-service-broker/src/main/java/org/springfra`

Provisioning Code:

```
@Service
public class MongoServiceInstanceService implements ServiceInstanceService {
...

  @Override
  public CreateServiceInstanceResponse createServiceInstance(CreateServiceInstanceRequest request) {
    // make sure we haven't provisioned this before (check broker database)
    ServiceInstance instance = repository.findOne(request.getServiceInstanceId());
    if (instance != null) {
      throw new ServiceInstanceExistsException(request.getServiceInstanceId(), request.getServiceDefinitionId());
    }

    instance = new ServiceInstance(request);

    if (mongo.databaseExists(instance.getServiceInstanceId())) {
      // ensure the instance is empty
      mongo.deleteDatabase(instance.getServiceInstanceId());
    }
```



```
    DB db = mongo.createDatabase(instance.getServiceInstanceId());
    if (db == null) {
      throw new ServiceBrokerException("Failed to create new DB instance: " + instance.getServiceInstanceId());
      }
    //save to broker database for record keeping
    repository.save(instance);

    return new CreateServiceInstanceResponse();
  }
...
```

What's happening?

The `createServiceInstance` method is where our broker provisions the database. But to do so two things must happen:

1. Record details in the broker database that we are provisioning a service instance (a MongoDB database)
2. Create the database

Deprovisioning Code:

```
@Service
public class MongoServiceInstanceService implements ServiceInstanceService {
...
  @Override
  public DeleteServiceInstanceResponse deleteServiceInstance(DeleteServiceInstanceRequest request) throws   MongoServiceException {
    String instanceId = request.getServiceInstanceId();
    //locate record in broker database
    ServiceInstance instance = repository.findOne(instanceId);
    if (instance == null) {
      throw new ServiceInstanceDoesNotExistException(instanceId);
    }

    // delete mongo database
    mongo.deleteDatabase(instanceId);
    // delete record from broker database
    repository.delete(instanceId);
    return new DeleteServiceInstanceResponse();
  }
}
```

What's happening?

The `deleteServiceInstance` method is where our broker deprovisions the database. But to do so two things must happen:

1. Delete the database
2. Delete the record of the service instance in the broker database

1.2.3.3.1. Questions

- The broker is required by the Cloud Controller to respond within how many seconds?
- Does provisioning have to be done synchronously?

1.2.3.4. Implement Binding and Unbinding

1. Review the documentation on implementing binding and unbinding.
2. We need to implement binding/unbinding in our `mongodb-service-broker` application. To do so we just need to implement the ServiceInstanceBindingService interface, because the Spring Cloud Service Broker project has already done the mapping.

Review the following file:

+ …
`/pivotal-cloud-foundry-developer-workshop/cloudfoundry-mongodb-service-broker/src/main/java/org/springfra`

Binding Code:

```
@Service
public class MongoServiceInstanceBindingService implements ServiceInstanceBindingService {
...
  @Override
  public CreateServiceInstanceBindingResponse createServiceInstanceBinding(CreateServiceInstanceBindingRequest request) {

    String bindingId = request.getBindingId();
    String serviceInstanceId = request.getServiceInstanceId();

    ServiceInstanceBinding binding = bindingRepository.findOne(bindingId);
    if (binding != null) {
      throw new ServiceInstanceBindingExistsException(serviceInstanceId, bindingId);
    }

    String database = serviceInstanceId;
    String username = bindingId;
    String password = "password";


    mongo.createUser(database, username, password);

    Map<String, Object> credentials =
        Collections.singletonMap("uri", (Object) mongo.getConnectionString(database, username, password));

    binding = new ServiceInstanceBinding(bindingId, serviceInstanceId, credentials, null, request.getBoundAppGuid());
    bindingRepository.save(binding);

    return new CreateServiceInstanceAppBindingResponse().withCredentials(credentials);
  }
...
```

What's happening?

The `createServiceInstanceBinding` method is where our broker binds an application to the

provisioned service instance (database). But to do so two things must happen:

1. Create a unique set of credentials for this binding request in MongoDB
2. Create a record of the binding in the broker database

Unbinding Code:

```
@Service
public class MongoServiceInstanceBindingService implements ServiceInstanceBindingService {

  @Override
  public void deleteServiceInstanceBinding(DeleteServiceInstanceBindingRequest request) {
    String bindingId = request.getBindingId();
    ServiceInstanceBinding binding = getServiceInstanceBinding(bindingId);

    if (binding == null) {
      throw new ServiceInstanceBindingDoesNotExistException(bindingId);
    }

    mongo.deleteUser(binding.getServiceInstanceId(), bindingId);
    bindingRepository.delete(bindingId);
  }
}
```

What's happening?

The `deleteServiceInstanceBinding` method is where our broker unbinds an application to the provisioned service instance (database). But to do so two things must happen:

1. Delete the credentials (user) for this binding request in MongoDB
2. Delete the record of the binding in the broker database

Congratulations! You have created a simple service broker.

1.2.3.4.1. Questions

- Do all services have to be bindable?

1.2.4. Use the MongoDB Service Broker

1. Configure the `mongodb-service-broker` application to use a MongoDB instance.

   A MongoDB instance can be obtained in the following ways:

a. Your instructor will provision MongoDB and provide connectivity details to you

b. Use a MongoDB instance in your environment

c. Provision a MongoDB instance on AWS (see the Appendix)

You'll need to communicate the mongo db hostname (or ip address) and password that the service broker will use via environment variables:

```
cf set-env mongodb-service-broker MONGODB_HOST {{mongodb_ip}}
```

..and..

```
cf set-env mongodb-service-broker MONGODB_PASSWORD {{mongodb_password}}
```

2. Restart the application.

```
cf restart mongodb-service-broker
```

3. You used `spring-music.jar` in the first lab. We are going to use it again.

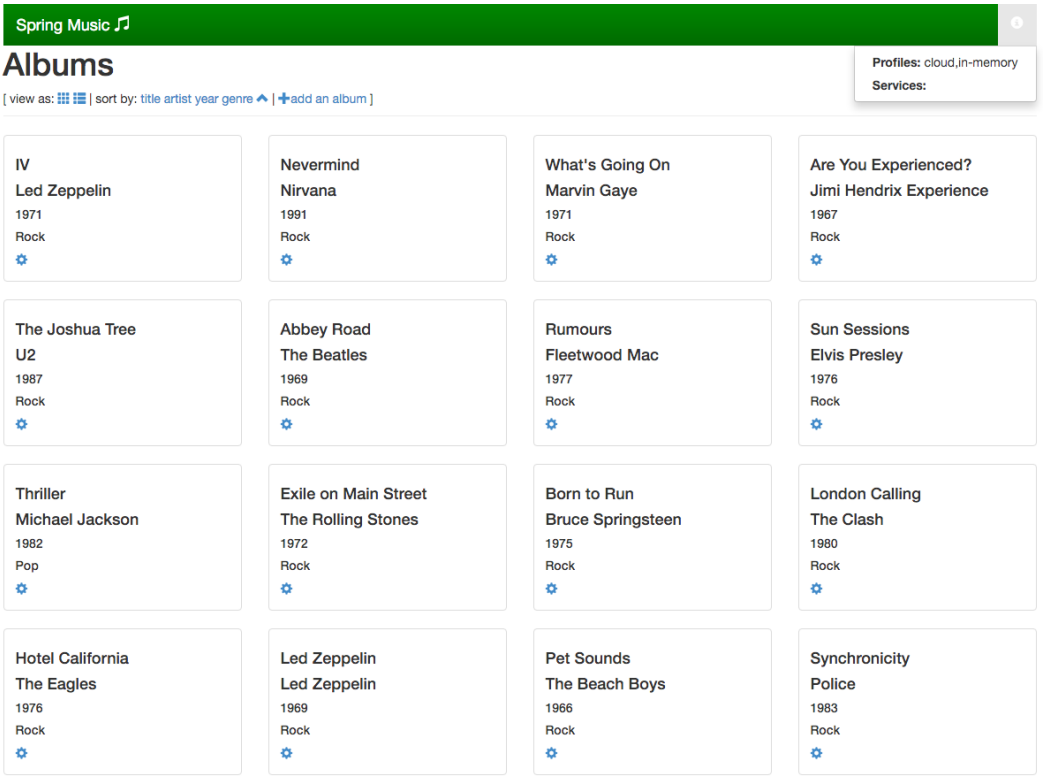
Source is not required, but you may be curious how it works as you move through the rest of this lab.

4. Push `spring-music`

```
$> cd .../pivotal-cloud-foundry-developer-workshop/demo-apps/spring-music/

$> cf push spring-music -p ./spring-music.jar -m 768M --random-route
```

5. View `spring-music` in a browser. Click on the button on the top right of the screen. Notice that there are no services attached and `spring-music` is using an embedded (in-memory) database.



6. Create a MongoDB service instance.

   For Example:

```
cf create-service MongoDB-{{initials}} standard mongo-service
```

7. Bind the `spring-music` to `mongo-service`.

```
cf bind-service spring-music mongo-service
```

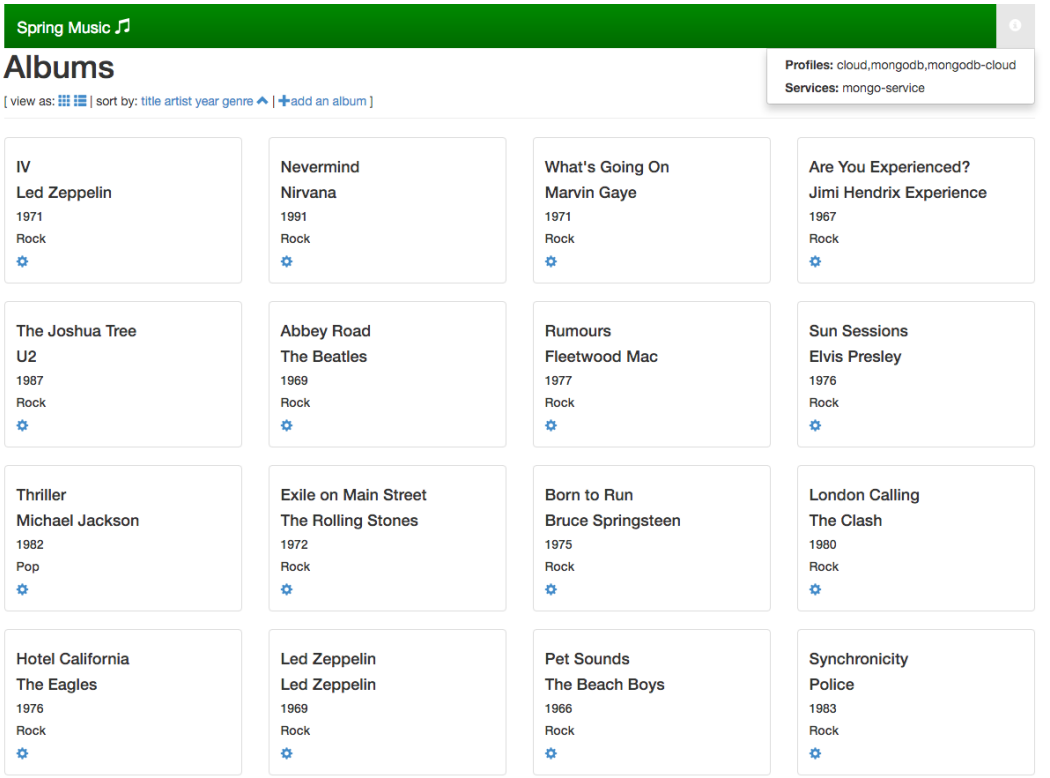
Note

You can safely ignore the "TIP: Use 'cf restage spring-music' to ensure your env variable changes take effect" message.

8. Restart `spring-music`

```
cf restart spring-music
```

9. Refresh `spring-music` in the browser. Click on the `i` button in the top right of the screen. You are now using MongoDB!



10.**Optional Step:** If you have access. View the data in MongoDB.

1.2.5. Clean up

1. Delete `spring-music`.

```
cf delete spring-music
```


2. Delete the `mongo-service` service instance.

```
cf delete-service mongo-service
```

3. Delete the service broker.

   For example:

```
cf delete-service-broker mongodb-service-broker-{{initials}}
```

4. Delete `mongodb-service-broker` application.

```
cf delete mongodb-service-broker
```

5. If provisioned by you, terminate your AWS MongoDB instance by going to your AWS EC2 dashboard, selecting the MongoDB instance, and clicking Actions -> Instance State -> Terminate. The VM should stop, change to "Terminated" status and will eventually be removed from your list of VMs (removal may not happen immediately).

1.3. Beyond the class

Review other sample brokers.

