## **Create Service Broker**

1. On your local machine create new java project. Don’t forget to replace YOUR\_USERNAME with your github username.

|  |
| --- |
| $ mvn archetype:generate -DgroupId=com.github.YOUR\_USERNAME.pgbroker -DartifactId=pgbroker -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false |

1. Add the following to your pom.xml file

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.5.8.RELEASE</version>  </parent>  <dependencies>  <dependency>  <groupId>postgresql</groupId>  <artifactId>postgresql</artifactId>  <version>LATEST</version>  </dependency>  <dependency>  <groupId>io.pivotal.ecosystem</groupId>  <artifactId>simple-broker</artifactId>  <version>1.1.0</version>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-jpa</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-cloud-connectors</artifactId>  </dependency>  </dependencies>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build> |

1. Add Application class

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

1. Add PostgresConfig. This class is responsible for creating all beans that later are going to be injected in the constructors of other classes; It takes postgres related parameters from environment variables and builds the following beans: datasource, jdbcTemplate, dbUrl, brokerApiVersion. Don't forget to replace YOUR\_PACKAGE with your package name.

|  |
| --- |
| @Configuration  @ComponentScan(basePackages = {"com.github.YOUR\_PACKAGE.pgbroker", "io.pivotal.ecosystem.servicebroker"})  public class PostgresConfig {  static final String POSTGRES\_DB = "POSTGRES\_DB";  static final String POSTGRES\_USER = "POSTGRES\_USER";  static final String POSTGRES\_PASSWORD = "POSTGRES\_PASSWORD";  static final String POSTGRES\_HOST\_KEY = "POSTGRES\_HOST";  static final String POSTGRES\_PORT\_KEY = "POSTGRES\_PORT";    static final String POSTGRES\_URI = "postgresuri";  static final String POSTGRES\_URI\_SCHEME = "postgresql";    @Bean  public DataSource datasource(Environment env) {  PGPoolingDataSource source = new PGPoolingDataSource();  source.setServerName(env.getProperty(POSTGRES\_HOST\_KEY));  source.setDatabaseName(env.getProperty(POSTGRES\_DB));  source.setUser(env.getProperty(POSTGRES\_USER));  source.setPassword(env.getProperty(POSTGRES\_PASSWORD));  return source;  }  @Bean  public JdbcTemplate jdbcTemplate(DataSource datasource) {  return new JdbcTemplate(datasource);  }  @Bean  public String dbUrl(Environment env) {  return POSTGRES\_URI\_SCHEME + "://" + env.getProperty(POSTGRES\_HOST\_KEY) + ":" + Integer.parseInt(env.getProperty(POSTGRES\_PORT\_KEY));  }  @Bean  public BrokerApiVersion brokerApiVersion() {  return new BrokerApiVersion();  }  } |

1. Add PostgresClient class. This class contains all postgres specific logic. It contains methods for DB schema creation and deletion as well as generation user credentials and assigning rights for that user to a particular database.

|  |
| --- |
| @Repository  class PostgresClient {  private static final Logger log = LoggerFactory.getLogger(PostgresClient.class);  private JdbcTemplate jdbcTemplate;  PostgresClient(DataSource dataSource) {  this.jdbcTemplate = new JdbcTemplate(dataSource);  }  String createDatabase() {  String db = getRandomId();  jdbcTemplate.execute("CREATE DATABASE " + db);  log.info("Database: " + db + " created successfully...");  return db;  }  void deleteDatabase(String db) {  jdbcTemplate.execute("DROP DATABASE IF EXISTS " + db);  log.info("Database: " + db + " deleted successfully...");  }  boolean checkDatabaseExists(String db) {  return jdbcTemplate.queryForObject("SELECT count(\*) from pg\_database WHERE datname = ?", new Object[]{db}, Integer.class) > 0;  }  private String getRandomId() {  String s= UUID.randomUUID().toString();  return “s” + s.replaceAll("[^a-zA-Z0-9]", "");  }    Map<String, String> createUserCreds(ServiceBinding binding) {  String db = binding.getParameters().get(PostgresConfig.POSTGRES\_DB).toString();  Map<String, String> userCredentials = new HashMap<String, String>();    userCredentials.put(PostgresConfig.POSTGRES\_USER, getRandomId());  userCredentials.put(PostgresConfig.POSTGRES\_PASSWORD, getRandomId());  userCredentials.put(PostgresConfig.POSTGRES\_DB, db);  log.debug("creds: " + userCredentials.toString());  jdbcTemplate.execute("CREATE USER " + userCredentials.get(PostgresConfig.POSTGRES\_USER) + " WITH PASSWORD '" + userCredentials.get(PostgresConfig.POSTGRES\_PASSWORD) + "'");  jdbcTemplate.execute("GRANT ALL PRIVILEGES ON DATABASE " + userCredentials.get(PostgresConfig.POSTGRES\_DB) + " to " + userCredentials.get(PostgresConfig.POSTGRES\_USER));  log.info("Created user: " + userCredentials.get(PostgresConfig.POSTGRES\_USER));  return userCredentials;  }  void deleteUserCreds(String uid) {  jdbcTemplate.execute("DROP USER IF EXISTS " + uid);  }  } |

1. Add PostgresBroker class. This is the main class that actually implements service broker API. We are going to add API method handlers in the following sections.

|  |
| --- |
| @Service  public class PostgresBroker extends DefaultServiceImpl {  private static final Logger log = LoggerFactory.getLogger(PostgresBroker.class);  private PostgresClient client;  private String dbUrl;  public PostgresBroker(PostgresClient client, String dbUrl) {  super();  this.client = client;  this.dbUrl = dbUrl;  }  } |

1. Add createInstance method. This method is called each time when you execute ‘cf create-service’ command. It redirects to the potgressClient a task of database creation and saves generated database name as instance parameter, so that is will be available for other API call handlers.

|  |
| --- |
| @Override  public LastOperation createInstance(ServiceInstance instance) {  log.info("creating database...");  try {  String db = client.createDatabase();  log.info("database: " + db + " created.");  instance.getParameters().put(PostgresConfig.POSTGRES\_DB, db);  } catch (Throwable t) {  log.error("error creating database.", t);  return new LastOperation(LastOperation.CREATE, LastOperation.FAILED, t.getMessage());  }  return new LastOperation(LastOperation.CREATE, LastOperation.SUCCEEDED, instance.getId() + " creating.");  } |

1. Implement deleteInstance method. This method is called each time when you execute ‘cf delete-service’ command. It is responsible for deleting the database and associated user.

|  |
| --- |
| @Override  public LastOperation deleteInstance(ServiceInstance instance) {  try {  String db = instance.getParameters().get(PostgresConfig.POSTGRES\_DB).toString();  String user = instance.getParameters().get(PostgresConfig.POSTGRES\_USER).toString();  log.info("deleting database: " + db);  client.deleteDatabase(db);  log.info("\*\*\*\*\*\*\*\*DELETED database: " + db);  client.deleteUserCreds(user);  log.info("\*\*\*\*\*\*\*\*DELETED User creds: " + user);  } catch (Throwable t) {  log.error("error deleting database.", t);  return new LastOperation(LastOperation.DELETE, LastOperation.FAILED, t.getMessage());  }  return new LastOperation(LastOperation.DELETE, LastOperation.SUCCEEDED, instance.getId() + " deleting.");  } |

1. Implement updateInstance method. This method is called each time you execute ‘cf update-service’ command. Our broker is not going to support service instance updates, so we just return an error here

|  |
| --- |
| @Override  public LastOperation updateInstance(ServiceInstance instance) {  log.info("update not yet implemented");  return new LastOperation(LastOperation.UPDATE, LastOperation.FAILED, instance.getId() + " updating.");  } |

1. Implement createBinding method. This method is called each time you bind you service to an application. It is responsible for creating user credentials and saving them as binding parameters.

|  |
| --- |
| @Override  public LastOperation createBinding(ServiceInstance instance, ServiceBinding binding) {  String db = instance.getParameters().get(PostgresConfig.POSTGRES\_DB).toString();  binding.getParameters().put(PostgresConfig.POSTGRES\_DB, db);  Map<String, String> userCredentials = client.createUserCreds(binding);  binding.getParameters().put(PostgresConfig.POSTGRES\_USER, userCredentials.get(PostgresConfig.POSTGRES\_USER));  binding.getParameters().put(PostgresConfig.POSTGRES\_PASSWORD, userCredentials.get(PostgresConfig.POSTGRES\_PASSWORD));  log.info("bound app: " + binding.getAppGuid() + " to database: " + db);  return new LastOperation(LastOperation.BIND, LastOperation.SUCCEEDED, "bound.");  } |

1. Implement deleteBinding method. This method is called each time we unbound service from an application. In our case nothing is done here.

|  |
| --- |
| @Override  public LastOperation deleteBinding(ServiceInstance instance, ServiceBinding binding) {  log.info("unbinding app: " + binding.getAppGuid() + " from database: " + instance.getParameters().get(PostgresConfig.POSTGRES\_DB));  return new LastOperation(LastOperation.UNBIND, LastOperation.SUCCEEDED, "bound.");  } |

1. Implement getCredentials method. This method is responsible for generating credentials, that Cloud Foundry later will expose to an application as VCAP\_SERVICE environment variable.

|  |
| --- |
| @Override  public Map<String, Object> getCredentials(ServiceInstance instance, ServiceBinding binding) {  log.info("returning credentials.");  Map<String, Object> m = new HashMap<String, Object>();  m.put(PostgresConfig.POSTGRES\_URI, dbUrl + "/" + binding.getParameters().get(PostgresConfig.POSTGRES\_DB).toString());  m.put(PostgresConfig.POSTGRES\_USER, binding.getParameters().get(PostgresConfig.POSTGRES\_USER));  m.put(PostgresConfig.POSTGRES\_PASSWORD, binding.getParameters().get(PostgresConfig.POSTGRES\_PASSWORD));  m.put(PostgresConfig.POSTGRES\_DB, binding.getParameters().get(PostgresConfig.POSTGRES\_DB));  return m;  } |

1. Indicate that our broker is synchronous.

|  |
| --- |
| @Override  public boolean isAsync() {  return false;  } |

1. Add catalog.json to application resources. This is a metadata that our service returns to Cloud Foundry. The most important fields here are id, name and list of service plans.

|  |
| --- |
| {  "services":[  {  "id":"aUniquePostgresId",  "name":"postgres",  "description":"Hi I'm Mr.Postgres! I share VMs",  "bindable": true,  "tags":[  "hello"  ],  "metadata":{  "displayName":"Postgress",  "documentationUrl":"https://docs.cloudfoundry.org/services/api.html"  },  "plans":[  {  "id":"anotherUniquePostgresId",  "name":"SharedVMs",  "description":"I share because I care",  "metadata": {  "cost": 0  },  "free": true  }  ]  }  ]  } |

1. Add logback.xml to configure logging. Don’t forget to replace YOUR\_PACKAGE with your package name.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <configuration>  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">  <layout class="ch.qos.logback.classic.PatternLayout">  <Pattern>  %d{yyyy-MM-dd HH:mm:ss} %-5level %logger{36} - %msg%n  </Pattern>  </layout>  </appender>  <logger name="YOUR\_PACKAGE" level="info" additivity="false">  <appender-ref ref="STDOUT"/>  </logger>  <root level="error">  <appender-ref ref="STDOUT"/>  </root>  </configuration> |

1. Create application manifest. Postgres connection information is hardcoded in the manifest.

|  |
| --- |
| ---  applications:  - name: my-postgres-broker  memory: 1G  instances: 1  path: target/pgbroker-1.0-SNAPSHOT.jar  services: [ redis-for-postgres ]  env:  SPRING\_PROFILES\_ACTIVE: cloud  POSTGRES\_HOST: 54.200.42.160  POSTGRES\_PORT: 5432  POSTGRES\_USER: postgres  POSTGRES\_PASSWORD: some\_password  POSTGRES\_DB: template1  SECURITY\_USER\_NAME: admin  SECURITY\_USER\_PASSWORD: admin |

## 

## 

## **Create the postgresql security group**

1. Create the postgresql.json file with the following content:

|  |
| --- |
| [  {  "destination": "0.0.0.0/0",  "ports": "5432",  "protocol": "tcp"  }  ] |

## 

1. Create the postgresql security group

|  |
| --- |
| $ cf create-security-group postgresql postgresql.json |

## 

1. Bind the security group to your org and space

|  |
| --- |
| $ cf bind-security-group postgresql pcfdev-org pcfdev-space |

## 

## 

## 

## **Push and register the broker in the Cloud Foundry**

1. Create redis service instance. Service broker need redis to persist information about created instances.

|  |
| --- |
| $ cf create-service p-redis shared-vm redis-for-postgres |

1. Build and push broker to CF

|  |
| --- |
| $ mvn clean install  $ cf push |

1. Check the following url to ensure that your service is working correctly. You will have to provide credentials to authenticate with the broker, those can be found in the deployment manifest.

|  |
| --- |
| http://my-postgres-broker.local.pcfdev.io/v2/catalog |

1. Register it as a service broker

|  |
| --- |
| $ cf create-service-broker postgresql admin admin http://my-postgres-broker.local.pcfdev.io  $ cf enable-service-access postgres |

## 

## 

## **Test the broker**

1. Clone sample application

|  |
| --- |
| $ git clone https://github.com/Altoros/cf-example-sinatra  $ cd cf-example-sinatra  $ git checkout postgres |

1. Create service instance

|  |
| --- |
| $ cf create-service postgres SharedVMs pgsql-service |

1. Push the application

|  |
| --- |
| $ cf push |

1. Check the following url to see whether the application successfully connected to postgres db.

|  |
| --- |
| http://pg-app.local.pcfdev.io |

f