Sumário

[Introdução 3](#_Toc101952297)

[Objetivo 3](#_Toc101952298)

[Step 1 4](#_Toc101952299)

[Criação do Compartment 4](#_Toc101952300)

[Criação de Usuário 4](#_Toc101952301)

[Criação de Grupo 5](#_Toc101952302)

[Criação de Policies 6](#_Toc101952303)

[Criação de SSH-KEY 7](#_Toc101952304)

[Criação de Instancia VM 8](#_Toc101952305)

[Step 2 9](#_Toc101952306)

[Entrar na Instancia via shell 9](#_Toc101952307)

[Instalar Docker 9](#_Toc101952308)

[Instalar Docker-Compose 10](#_Toc101952309)

[Instalar Git 10](#_Toc101952310)

[Step 3 11](#_Toc101952311)

[Criação do Docker-Compose-Services 11](#_Toc101952312)

[Teste na OCI 14](#_Toc101952313)

[Spet4 17](#_Toc101952314)

[Requeriments 17](#_Toc101952315)

[Criação API 17](#_Toc101952316)

[Dockerfile 19](#_Toc101952317)

[Docker-Compose 19](#_Toc101952318)

[Step5 20](#_Toc101952319)

[Postgres 20](#_Toc101952320)

[Create\_tables 21](#_Toc101952321)

[Rabbit MQ 23](#_Toc101952322)

[Redis 24](#_Toc101952323)

[Step6 25](#_Toc101952324)

[Queues 25](#_Toc101952325)

[Consumo da classe RabbitQueue 28](#_Toc101952326)

[Routes 29](#_Toc101952327)

[Resumo 29](#_Toc101952328)

# Introdução

Nesse projeto, utilizarei algumas ferramentas contidas na OCI (Oracle Claud Infraestructure) visto que, recentemente passei na certificação “Oracle Cloud Infrastructure Foundations 2021 Associate“.

Utilizarei também:

- Python com framework Flash para criação de API Gateway;

- Persistência de dados em banco relacional com PostgreeSQL;

- Camada de cache em memória com Redis;

- Utilização de um ORM para manipulação dos dados;

- Docker para criação de containers de microsserviços.

Vamos começar...

## Objetivo

Criar duas aplicações básicas de microserviços:

O primeiro deles deverá ser um cadastro de usuários, contendo as seguintes informações:

- create\_user, show\_all\_user, show\_one\_user, edit\_user, edit\_password e delete\_user

Tabela de usuários “users” deverá conter os campos: user\_id, nick\_name, nick\_name, \_name, \_name, email, phone\_number, phone\_number, updated\_at.

O segundo será um serviço de OS (ordem de serviço) que deverá conter o user\_id contido no banco de dados. Deverá ter as seguintes informações:

- order\_id, user\_id, item\_description, item\_quantity, item\_price, total\_value, created\_at, updated\_at.

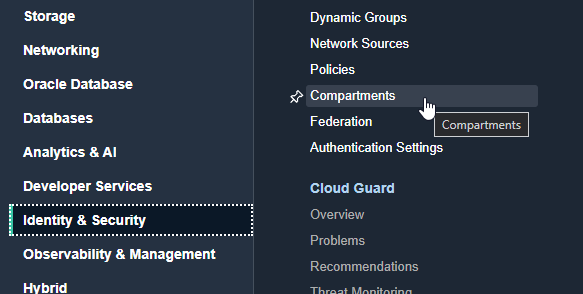
# Step 1

Nesse primeiro passo, configurei um ambiente OCI para rodar toda a aplicação em cloud.

## Criação do Compartment

Iniciei criando um *conpartiment.*

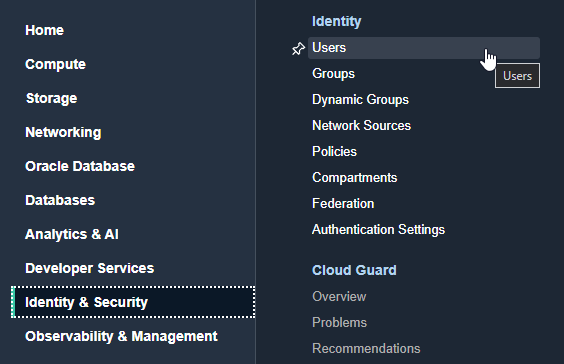
No ícone de hamburguer na página inicial do OCI, em *> Identity & Security > compartments* criei um compartimento com o nome de **Developmet.**

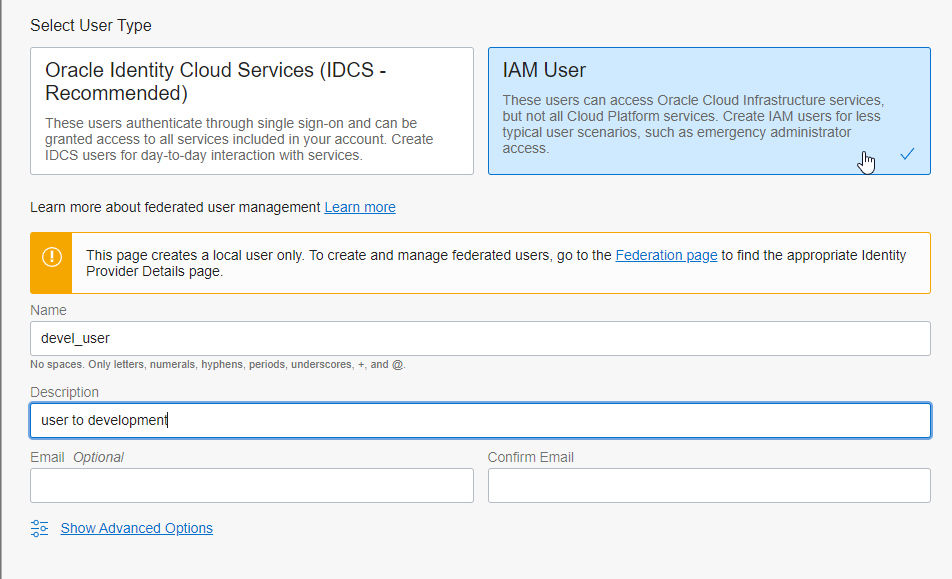


## Criação de Usuário

Após isso, criei um usuário de desenvolvimento para ter acesso somente ao compartimento **Developmet**.

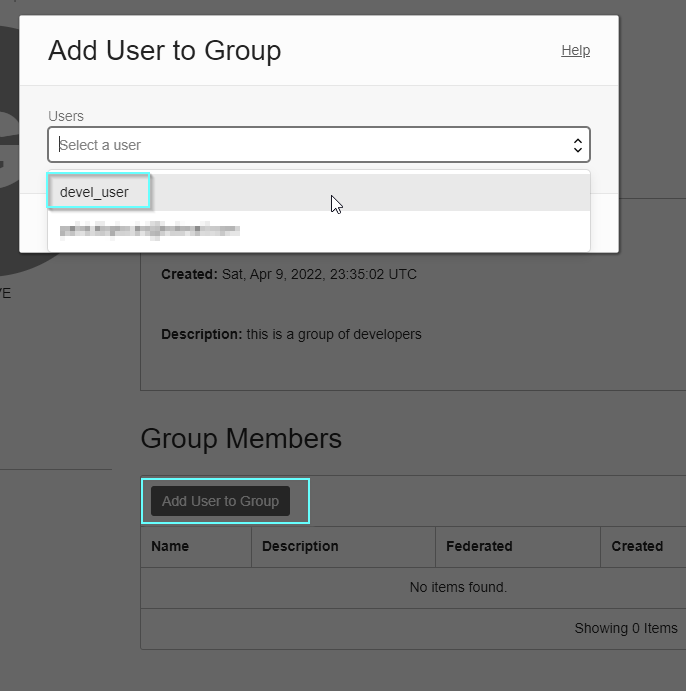
Novamente no menu lateral, em *> Identity & Security > users*, criei o usuário **devel\_user,** com o tipo IAM USER. Esses usuários podem acessar os serviços do Oracle Cloud Infrastructure, mas nem todos os serviços do Cloud Platform. Os usuários do IAM são cenários de usuário atípico, como acesso de administrador de emergência.





## Criação de Grupo

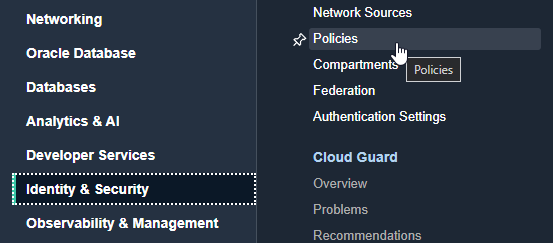
No menu lateral, em *> Identity & Security > group*s, criei um grupo com o nome de **Developer\_Group.**  Depois disso, adicionei o usuário **devel\_user** ao grupo clicando no botão “Add user to Group”.



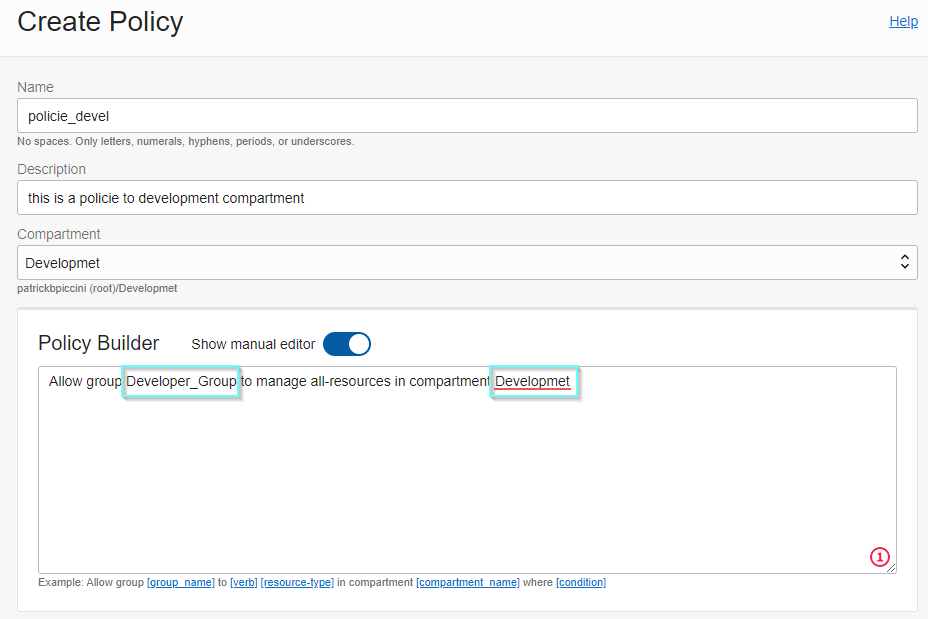
## Criação de Policies

Finalizando o processo de criação de Usuário, Grupo, e adição do usuário ao grupo, é necessário atribuir uma sequência de política de permissões ao grupo, para que assim, o grupo criado tenha acesso ao compartimento *Developer* para fazer as devidas interações*.*

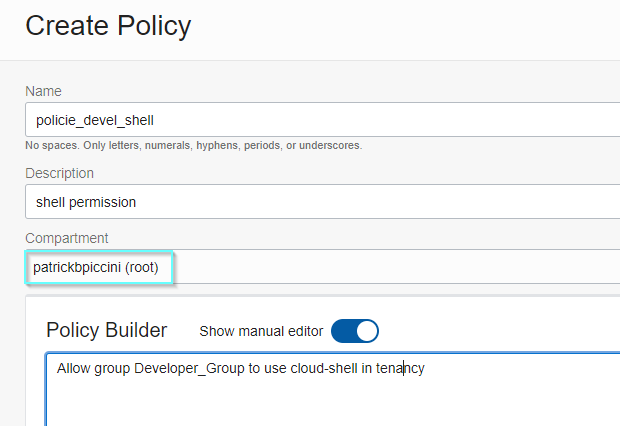
Vamos em *> Identity & Security > Policies*



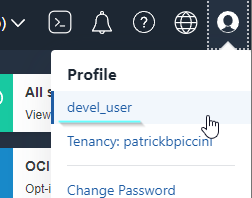
Após ser direcionado a página de policies, selecionei o compartimento Developmet para receber a política que criei. Na policie, permiti o Developer\_Group (e todos seus usuários) a utilização de todos os recursos no compartimento OCI.



Também será criado uma outra policie no compartimento **root**, para liberar o acesso ao terminal cloud shell, através dos usuários do grupo *Developmet.* Isso deverá ser feito devido ao fato de que precisarei de uma ssh-key disponibilizada pelo usuário, para conseguir criar nossa instancia no OCI.

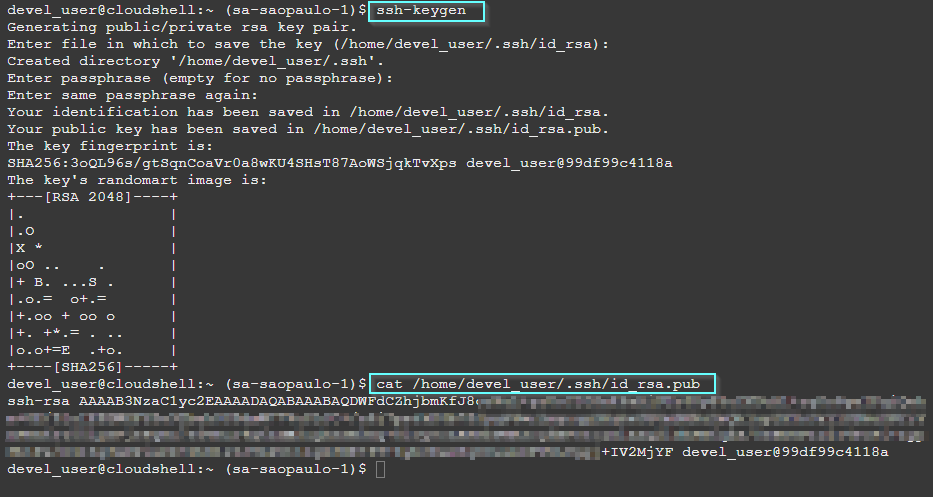


## Criação de SSH-KEY

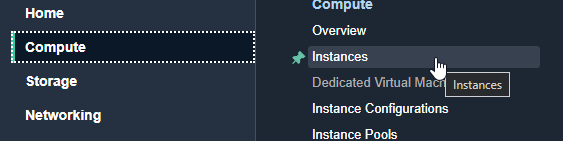
**Logado no usuário *devel\_user,* conectei ao Oracle Cloud Shell para adquirir uma ssh\_key que posteriormente irei utilizar. Para isso, usei alguns simples comandos para a criação dessa chave.

ssh-keygen – Para criação da chave

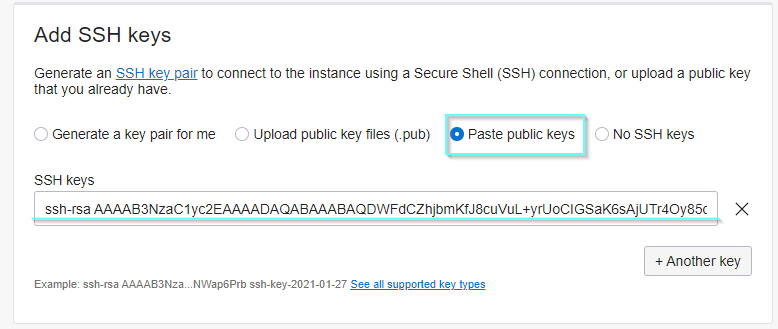
cat /home/your\_user/.ssh/id\_rsa.pub – Mostrará o conteúdo contido no arquivo id\_rsa.pub, que foi criado com o comando anterior



## Criação de Instancia VM

Ainda conectado ao usuário *devel\_user* criei uma instancia, que nada mais é que uma Virtual Macine. Em > *compute > instances > create instance.*

Ao criar uma instancia na página de configuração da VM, colei a ssh-key adquirida anteriormente no campo onde solicita essa chave. Isso é feito para que se consiga acessar a VM remotamente. As imagens do Oracle Linux, CentOS ou Ubuntu usam esse par de chaves SSH ao contrário de uma senha para autenticar um usuário.



OBS: A configuração da instancia que irei utilizar, são disponibilizadas pelo serviço *Oracle Cloud – Free Tier,* sendo ela 1 VMs de computação baseadas em AMD com 1/8 OCPU\*\* e 1 GB de memória cada.

Para mais informações dos serviços Free Tier, acesse:

<https://www.oracle.com/br/cloud/free/>

# Step 2

Após Configurar a infraestrutura no Oracle Cloud, necessita-se fazer algumas instalações de alguns programas que utilizarei. O Docker para a criação dos nossos containers de servidores e microsserviços, o Docker-Compose para facilitar a criação dos containers, e o Git, para versionamento de vocigo

Como desenvolvi os códigos fora de nossa VM da OCI, o Git é o fator principal para que todos os códigos sejam disponíveis facilmente.

## Entrar na Instancia via shell

Logado no usuário **devel\_user**, iniciei o cloud shell para conectar-se ao VM criada através do SSH Connection.

\* Detalhe, essa conexão pode ser feita de qualquer terminal ou comprador que tenha acesso ao SSH Connection, desde que se tenha cadastrado na hora da criação da VM a SSH-KEY da máquina que onde se conectará.

Com o comando ssh opc@<ip\_public> entrei na máquina e comecei a fazer a instalação com o pacote de instalação Dandified YUM (DNF) no Oracle Linux 8.

## Instalar Docker

Com a sequência de comando abaixo, foi instalado o Docker na VM:

> dnf install -y dnf-utils zip unzip

> dnf config-manager --add-repo=https://download.docker.com/linux/centos/docker-ce.repo

> dnf remove -y runc

> dnf install -y docker-ce --nobest

> systemctl enable docker.service

> systemctl start docker.service

> systemctl status docker.service

> docker version

Adicionar o usuário ao grupo docker para poder executar comandos Docker;

> sudo usermod -aG docker opc

> newgrp docker

Vemos que Docker está rodando e pronto para ser usando.

Texto

Descrição gerada automaticamente

## Instalar Docker-Compose

Foi instalado o Docker-Compose na VM com os seguintes comandos:

> sudo dnf -y install curl

> curl -s https://api.github.com/repos/docker/compose/releases/latest|grep browser\_download\_url|grep docker-compose-linux-x86\_64|cut -d '"' -f 4|wget -qi –

> ls -1 docker-compose-linux-x86\_64\*

> sha256sum -c docker-compose-linux-x86\_64.sha256

> chmod +x docker-compose-linux-x86\_64

> sudo mv docker-compose-linux-x86\_64 /usr/local/bin/docker-compose

> docker-compose version

Docker-Compose instalado com sucesso.

Tela preta com letras brancas

Descrição gerada automaticamente

## Instalar Git

Comandos para instalação do Git:

> yum install curl-devel expat-devel gettext-devel openssl-devel zlib-devel -y

> yum install gcc perl-ExtUtils-MakeMaker -y

> cd /usr/local/

> wget https://mirrors.edge.kernel.org/pub/software/scm/git/git-2.35.1.tar.gz

> tar zxvf git-2.35.1.tar.gz

> cd git-2.35.1/

> make prefix=/usr/local/git all

> make prefix=/usr/local/git install

> echo "export PATH=$PATH:/usr/local/git/bin" >> /etc/bashrc

> source /etc/bashrc

# Step 3

Nesse próximo passo, parto do pressuposto que meu ambiente já está configurado e com todas as instalações feitas, assim, posso começar a criar as primeiras linhas de código.

Começo criando uma pasta chamada *MS-application.* Essa pasta será onde irei criar todos os microserviços, configuração de containers de serviço, e mais alguns detalhes, que será mostrado no decorrer das publicações.

Na raiz da pasta, crio um arquivo chamado *docker-compose-services.yml.*

Ms-Application

↪ *docker-compose-services.yml*

## Criação do Docker-Compose-Services

Como vou usar containers para praticamente tudo, será necessário que os servidores Rabbit, Redis, Postgres, e interfaces de manager sejam criadas separadamente em seus devidos containers. É necessária essa separação para que nenhum servidor dependa do outro para funcionar. Com isso, já temos os primeiros passos na criação da arquitetura de microcerviços.

Para inicializar esses serviços rapidamente, criei um Docker-Compose configurando uma *Network*, onde posteriormente estarão todos os serviços dentro dessa mesma rede.

A rede foi nomeada de *internal-network,* nela, configuro o endereço 10.5.0.0 sendo uma rede classe A, com uma máscara de rede /16. Tambem já defino o gateway da rede para 10.5.0.1.

version: "3.7"

networks:

network-service:

driver: bridge

name: network-internal

ipam:

config:

- subnet: 10.5.0.0/16

gateway: 10.5.0.1

Nos serviços, começo configurando o RabbitMQ. Utilizo a imagem *rabbitmq:3-management-alpine,* que é uma imagem mínima do Docker baseada no Alpine Linux com um índice de pacotes completo e apenas 5 MB de tamanho*.* Nas variáveis de ambiente é deixado com o valor padrão o user, password e hots, que será utilizado posteriormente para conectar ao manager do rabbit. Nas portas, deixo 15672 para o maneger e 5672 para o servidor. E por fim, configuro o contêiner para ficar dentro da Network que foi criada anteriormente, apontando também um ip estático para essa instancia.

<https://hub.docker.com/_/alpine>

services:

rabbitmq:

image: "rabbitmq:3-management-alpine"

environment:

RABBITMQ\_DEFAULT\_USER: "guest"

RABBITMQ\_DEFAULT\_PASS: "guest"

RABBITMQ\_DEFAULT\_VHOST: "/"

ports:

- "15672:15672"

- "5672:5672"

labels:

NAME: "rabbitmq1"

networks:

network-service:

ipv4\_address: 10.5.0.10

A estrutura dos próximos serviços é bem semelhante a essa primeira, com apenas alguns detalhes como diferença. No postgres, utilizei a imagem *postgres:13, indiquei as v*ariáveis padrões do user e password para conexão a instancia, apontei a porta padrão 5432, criei um volume compartilhado de *.data* para */data/, configurei a* Network com o ip estático, e um detalhe muito importante, coloquei o parâmetro *restart: Always*, que fara com que a instancia seja reiniciada caso pare de funcionar. Se for interrompido manualmente, ele será reiniciado somente quando o daemon do Docker for reiniciado ou o próprio contêiner for reiniciado manualmente.

postgres:

container\_name: postgres\_container

image: postgres:13

restart: always

environment:

POSTGRES\_USER: postgres

POSTGRES\_PASSWORD: postgres

POSTGRES\_DB: baseapplication

ports:

- "5432:5432"

volumes:

- ./data:/data/

networks:

network-service:

ipv4\_address: 10.5.0.11

Para ter uma parte visual do banco de dados, faço a criação de uma instancia com o Pgadmin que poderemos conectar ao servidor do postgres via Web. Sendo assim, utilizei a imagem *dpage/pgadmin4,* setei as variáveis de email e password para entrarmos na pagina web, Apontei a porta de 16543/80 (external/internal), compartilhei os volumes ./data/:/data/ e ./postgres-backup:/var/lib/postgresql/backups, configurei a network com um ip estático, e coloquei uma dependência, no parâmetro *depends\_on – postgres.*isso fará com o a instancia do Pgadmin só seja criada, após a criação da instancia do postgres.

pgadmin:

container\_name: pgadmin\_container

image: dpage/pgadmin4

environment:

PGADMIN\_DEFAULT\_EMAIL: admin@admin.com

PGADMIN\_DEFAULT\_PASSWORD: admin

ports:

- "16543:80"

depends\_on:

- postgres

volumes:

- ./data/:/data/

- ./postgres-backup:/var/lib/postgresql/backups

networks:

network-service:

ipv4\_address: 10.5.0.12

E para camada de cache, criei uma instancia com Redis utilizando a imagem *redis:alpine*, apontei a porta padrão 6379, e configurei a Network com um ip estático.

redis:

image: redis:alpine

container\_name: redis-container

ports:

- "6379:6379"

networks:

network-service:

ipv4\_address: 10.5.0.9

No final, terei vários containers rodando dentro de uma Rede, tendo algo parecido com isso:Diagrama

Descrição gerada automaticamente

## Teste na OCI

Agora que tenho pronto o meu Docker-Compose com todos os serviços que irei utilizar, faço um teste dentro da VM que criamos na OCI. Para conseguir compartilhar o código, faço *um git commit* de minha pasta Ms-application, e dentro da instancia OCI dou um *git pull* para baixar.

OBS: Não entrarei em detalhes nos comando do git, caso tenha alguma especifica, consulte a documentação oficial: <https://comandosgit.github.io/>

Dentro a instancia, inicio meus serviços com o comando *docker-compose -f docker-compose-services.yml up,* e passo o nome do arquivo que deverá ser iniciado.

Nesse momento começara a baixar as imagens dos containeres.

Interface gráfica do usuário

Descrição gerada automaticamente com confiança média

Texto

Descrição gerada automaticamente

Texto

Descrição gerada automaticamente

Agora que está tudo rodando, pude fazer um teste me conectando ao manager do RabbitMQ pela Web. Porém como a aplicação está em cloud, é necessário que nas configurações da VNC do projeto, seja expostas as portas que iremos utilizar para nos conectar externamente.

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

Na OCI, em Networking > Virtual Cloud Networks > sua\_vnc > Security List Details > Ingress Rules. Irei adicionar as portas que utilizarei.

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

Após a liberação da porta, consegui ter acesso aos meus contêineres, tanto o do rabbit quando ao pgadmin, que também liberei a porta. Percebam que para me conectar aos meus contêineres, utilizo o ip public da VM que estou utilizando.

Interface gráfica do usuário

Descrição gerada automaticamente

Interface gráfica do usuário, Site

Descrição gerada automaticamente

# Spet4

Nesse passo, comecei a criar o rosto da API. Irei explicar como estruturei as rotas para a conexão utilizando Flask. Como funciona o disparo das funções das rotas. Criei a minha própria imagem para futuramente rodar a aplicação utilizando Dockerfile, juntamente com um Docker-Compose para subir as instancias Docker, e por fim, farei um teste de conexão localmente as rotas que criei.

No diretório raiz, crio a pasta API com os seguintes arquivos:

MS-Application

API

↳ docker-compose-api.yml

↳ Dockerfile

↳ requirements.txt

↳ server.py

## Requeriments

No arquivo *requeriments.txt* irão ficar todas as instalações de bibliotecas e framework que será utilizado na aplicação da API. As bibliotecas devem ficar separadas por linha, visto que usaremos um comando para uma instalação recurciva.

*requirements.txt*

flask

Flask-Caching

pika

psycopg2-binary

uuid

Comando para instalação:

> pip install -r requeriments.txt

## Criação API

Dentro o arquivo *server.py* irei criar uma espécie de API-Gateway, onde ela será responsável apenas por distribuir as informações para as filas do rabbit. Cada rota terá sua funcionalidade, como **create\_user** para criar um usuário, **show\_all\_user** para mostrar todos os usuários. A mesma logica será aplicada referente aos **orders**, como **create\_order, list\_order**  e assim sucessivamente.

Irei mostrar apenas a criação de uma das rotas, pois a estrutura será a mesma para todas as outras, basta altera o nome da rota e função que será disparada após acessá-la.

Primeiro, deve-se importar o framework **Flask** juntamente com a funcionalidade **request.**

Criei uma classe chamada **Api\_server** e instanciei o Flask a variável **app.** Após isso, para definir uma rota utilizei @app.route(“rota”, method), onde apontei *"/user/create\_user/"* como destino*,* e o método que essa rota irá aceitar será somente **POST.**

Abaixo da rota fica a função que será disparada após ser acessada, que nomeei de **create\_user().** Dentro da ‘def’, faço uma verificação se o method é o mesmo que espero nessa função. Caso isso seja atendido, coloquei como teste, um return que apenas mostrará o que enviamos no corpo da requisição. Caso não atenda, será mostrado um erro referente ao método enviado.

from flask import Flask, request

class Api\_server():

app = Flask(\_\_name\_\_)

@app.route("/user/create\_user/", methods=['POST']) # <- rota que será acessada

def create\_user(): # <- methodo que será disparado ao acessar a rota acima

if request.method == 'POST':

payload = request.get\_json()

return {'Status': 200, 'Message': payload}

else:

return {'Status': 404, 'Message': 'Erro no envio do method'}

Para executar o servidor, nas ultimas linhas instanciei a classe que criei, e utilizei o comando app.run(host, porta), para definir o host e porta que irá rodar a API.

if \_\_name\_\_ == '\_\_main\_\_':

APP = Api\_server()

APP.app.run('0.0.0.0', 7000)

Para testar, executei o arquivo *server.py* e enviei um JSON como corpo da requisição para a rota que criei anteriormente, juntamente com o método POST:

Texto

Descrição gerada automaticamente

Tela de celular

Descrição gerada automaticamente

OBS: Criei todas as demais rotas, mas não irei exemplificar cada uma delas pois a funcionalidade é a mesma. Se tiverem curiosidade, bata ir ao repositório da aplicação e olhar as outras rotas.

## Dockerfile

No arquivo *Dockerfile* irei criar uma imagem personalizada, adequada a aplicação que irei criar:

FROM python:3.8-alpine <- Imagem base utilizada

COPY . /app/ <- copia da pasta API para /app/

ENV HOST='10.0.0.34' <- ip privado

ENV HOST\_DATABASE='10.5.0.11' <- ip do postgres

ENV CACHE\_TYPE=redis <- type do redis

ENV CACHE\_REDIS\_HOST='ip\_do\_redis' <- ip do redis

ENV CACHE\_REDIS\_PORT='6379' <- porta do redis

EXPOSE 7000 <- porta que será a API

WORKDIR /app <- diretório de trabalho

RUN pip install --upgrade pip <- instalação do pip

RUN pip install -r requirements.txt <- instalação das libs

RUN python server.py <- comando para iniciar a API

## Docker-Compose

O Docker-Compose para subir a aplicação é bem simples. Terá apenas um service onde irá rodar o Dockerfile no ‘*build: . ’*.A exposição da porta 7000, que será a porta do API, e apontamento da Network que foi criada no Docker-Compose, ondem é preciso especificar que a rede que estará sendo utilizada é externa, na parte *external: true*

version: "3.7"

services:

api:

build: .

ports:

- "7000:7000"

networks:

- internal-network

networks:

internal-network:

external: true

# Step5

Como visto no Step4, foi iniciado os primeiros passos pra interagir com a API gateway, e também, foi criado as configurações para a inicialização do container, quando for subir para a OCI.

Nesse passo, mostrai como configurei a conexão com os servidores do Rabbit, Redis e Posgres. E como planejei a arquitetura de pastas e módulos, para ter uma aplicação organizada.

Dentro da pasta API, crio uma subpasta chamada config com os seguintes arquivos

API

↳ config

↳ database\_connection.py

↳ rabbitmq\_connection.py

↳ redis\_connection.py

↳ docker-compose-api.yml

↳ Dockerfile

↳ requirements.txt

↳ server.py

## Postgres

Antes de continuar, deve-se ter certeza de que executou todas as instalações das bibliotecas, contida no arquivo *requeriments.txt.*

Para começarmos, devemos importar os módulos *psycopg2* e *os,* que iremos utilizar para a conexão.

import psycopg2, os

Como teremos que apontar o host do servidor Postger para nossa aplicação, na criação do Dockerfile foi criado uma variável de ambiente chamada HOST\_DATABASE, contendo o ip privado da instancia da OCI. Porém, essa variável de ambiente só será utilizada em Produção, sendo assim, criei uma variável dentro do arquivo *database\_connection.py* contendo o ip público, somente para fins desenvolvimento em Homologação, utilizando os serviços que já estão rodando.

Já que iremos nos conectar a um banco te dados, essa deve ser uma das primeiras coisas que deve acontecer quando a API for inicializada. Para isso, foi criada uma classe chamada *ConnectionDatabase(),* onde em sei *\_\_init\_\_* iremos fazer esse processo de conexão.

Dentro da função connect da biblioteca psycopg2, vamos colocar os parâmetros *host,* que será o ip da instancia OCI, *port* contendo a numeração padão da porta postgres(já foi liperada a porta no OCI para a conexão externa), *badabase*, contendo o nome do banco, *user* e *password,* contendo usuário e senha que foi colocado na imagem que subimos do postres.

Após o apontamento dos parâmetros obrigatórios para a conexão, é criado um atributo chamado *cursor,* onde ele será o responsável por executar qualquer interação com o banco de dados. Logo abaixo há um método *create\_tables*, que iremos criar logo em seguida. Para finalizar, tudo isso esta dentro de um try-catch, para caso haja alguma falha na conexão, seja lançado um except com o erro.

# HOST\_DATABSE = os.environ['HOST\_DATABASE']

HOST\_DATABSE = '144.22.130.193'

class ConnectionDatabase():

def \_\_init\_\_(self):

try:

self.connection = psycopg2.connect(

host=HOST\_DATABSE,

port=5432,

database="baseapplication",

user="postgres",

password="postgres")

self.cursor = self.connection.cursor()

# start tables

self.create\_tables()

print('[✓] Connected to Postgres')

except Exception as error:

print(f'[X] CONNECTING POSTGRES ERROR: {error}')

### Create\_tables

Antes de começar a fazer a função que irá criar as tabelas, vamos analisar o que precisará ser feito no modelo ER abaixo:

Uma imagem contendo Diagrama

Descrição gerada automaticamente

Iremos criar uma estrutura simples. Serão duas tabelas, *users* e *orders*. Nelas terão seus atributos, com seus determinados tipos. Note que na tabela orders, iremos herdar um atributo da tabela user, que será o *user\_id*, onde na tabela user é a PrimaryKey. Já na tabela orders, o atributo herdado *user\_id*, será tratado como uma chave estrangeira ForeignKey.

Em relação a cardinalidade vamos ter 1 usuario pode ter vários orders, e 1 order pode ter somente um usuário, representados pelos símbolos.

Tela de celular com texto preto sobre fundo branco

Descrição gerada automaticamente com confiança média

Após essa analize, devemos criar o metodo *create­\_table, e criar as tabelas.* Para executar comandos SQL através da biblioteca, basta escrever em uma string o comando SQL que deseja executar, e passar como parâmetro para função *execute.* Visto isso, primeiramente criei um select da verção do postegres, para que toda a vez que seja iniciado a conexão, seja mostrado a versão. Logo após foi criado um atributo contendo toda a query de criação das duas tabelas, baseada do modelo ER visto anteriormente. Sendo assim, como método já foi chamado no *\_\_init\_\_.*

def create\_tables(self):

self.cursor.execute("SELECT version();")

record = self.cursor.fetchone()

print("[✓] You are connected to - ", record)

create\_table\_query = '''

CREATE TABLE IF NOT EXISTS users (

user\_id SERIAL NOT NULL,

nick\_name varchar(50) UNIQUE NOT NULL,

password varchar(256) NOT NULL,

\_name varchar(50) NOT NULL,

cpf varchar(11) NOT NULL,

email varchar(50) NOT NULL,

phone\_number varchar(50) NOT NULL,

created\_at TIMESTAMP,

updated\_at TIMESTAMP,

PRIMARY KEY (user\_id)

);

CREATE TABLE IF NOT EXISTS orders (

order\_id SERIAL NOT NULL,

user\_id integer NOT NULL,

item\_description varchar(256) NOT NULL,

item\_quantity integer NOT NULL,

item\_price integer NOT NULL,

total\_value integer NOT NULL,

created\_at TIMESTAMP,

updated\_at TIMESTAMP,

PRIMARY KEY (order\_id),

FOREIGN KEY(user\_id) REFERENCES users(user\_id)

);'''

self.cursor.execute(create\_table\_query)

self.connection.commit()

print('[✓] Created tables on DataBase')

## Rabbit MQ

O Rabbit é um message borker open source fácil e leve de ser implementado, tanto local, quanto em nuvem. O rabbit suporta vários tipos de protocolo de mensagens, para haver facilidade na comunicação entre aplicações. Existem muito outros recursos, porém, não irei detalhar eles nesse projeto.

Nesse momento, iremos fazer somente a conexão com o servidor Rabbit, onde usaremos uma variável de ambiente como no postgres, no entanto ela será a HOST, contida no dockerfile. Da mesma forma que na conexão do postgres, criei uma variável estática contendo o ip publico, somente para desenvolver a aplicação local, utilizando o servidor que já esta rodando os serviços.

Começamos com a importação das bibliotecas necessárias e o apontamento da variável com o IP Público da instancia OCI.

import pika, os

# HOST\_RABBIT = os.environ['HOST']

HOST\_RABBIT '144.22.130.193'

Agora iremos criar uma classe chamada *RabbitConnection(),* e faremos a conexão no metodo *\_\_init\_\_.* Utilizando o methodo de conexão da biblioteca pika chamada *BlockingConnection(),* passaremos como parâmetros, através do motodo *ConnectionParameters()* o host, contida na vaviavel HOST\_RABBIT, e a porta padrão 5672. Em seguida, será criado um atributo chamado *channel*, que será responsável por executar toda e qualquer interação com o servidor rabbit, tudo isso foi criado em um try-catch, para caso haja alguma falha na conexão, seja lançado um except com o erro.

class RabbitConnection():

def \_\_init\_\_(self) :

try:

self.connection = pika.BlockingConnection(

pika.ConnectionParameters(host=HOST\_RABBIT, port=5672))

self.channel = self.connection.channel()

print('[✓] Connected to RabbitMQ server')

except Exception as error:

print(f'[X] CONNECTING RABBIT MQ ERROR: {error}')

## Redis

Redis é um armazenamento de estrutura de dados na memória de código aberto (licenciado BSD), usado como banco de dados, cache, agente de mensagens e mecanismo de streaming. Um dos serviços que se destaca é o cache, que iremos utilizar nessa aplicação.

Para criar a conexão com o servidor Redis, iremos utilizar a variável de ambiente CACHE\_REDIS\_HOST, porem, para a fim de desenvolvimento, irei criar uma variável local contendo o ip público da instancia.

Nessa configuração iremos fazer algo um pouco diferente. Como o redis e flasck conseguem trabalhar junto naticamente, a configuração para a conexão ao servidor, será feita através do arquivo principal *server.py*, porem, será mostrado como fazer isso no próximo passo.

No momento a configuração será a seguinte, importar a biblioteca *os,* criar uma classe chamada *BaseConfig(),* e dentro dela, apontar as variáveis de ambiente contidas no Dockerfile, veja:

import os

class BaseConfig(object):

CACHE\_TYPE = os.environ['CACHE\_TYPE']

CACHE\_REDIS\_HOST = os.environ['CACHE\_REDIS\_HOST']

CACHE\_REDIS\_PORT = os.environ['CACHE\_REDIS\_PORT']

CACHE\_REDIS\_DB = os.environ['CACHE\_REDIS\_DB']

Como irei testar localmente para homologação, criarei a mesma classe, porém, com atributos com valor estático do servdor.

class BaseConfig(object):

CACHE\_TYPE = 'redis'

CACHE\_REDIS\_HOST = '144.22.130.193'

CACHE\_REDIS\_PORT = '6379'

CACHE\_REDIS\_DB = 0

# Step6

Nessa etapa, iremos criar as filas que necessitamos para a comunicação com os microcervices. Iremos instanciar a classe de configuração do Rabbit para inicializar juntamente com a API, estabelecendo uma conexão, e criando automaticamente as filar para consumo. Iremos ver também como funciona o recurso RPC, que irei detalhar posteriormente como funciona. Tambem, será explicado cada funcionalidade da criação da fila do rabbit que iremos utilizar. Por fim, irei mostrar no manager, a parte visual de como irá ficar nossas filas. No final dessa etapa, será resumindo como ira funcionar o código, por uma visão geral.

Dentro da pasta API, crie uma subpasta chamada rabbitmq\_controller com o arquivo rabbit\_queues.py.

API

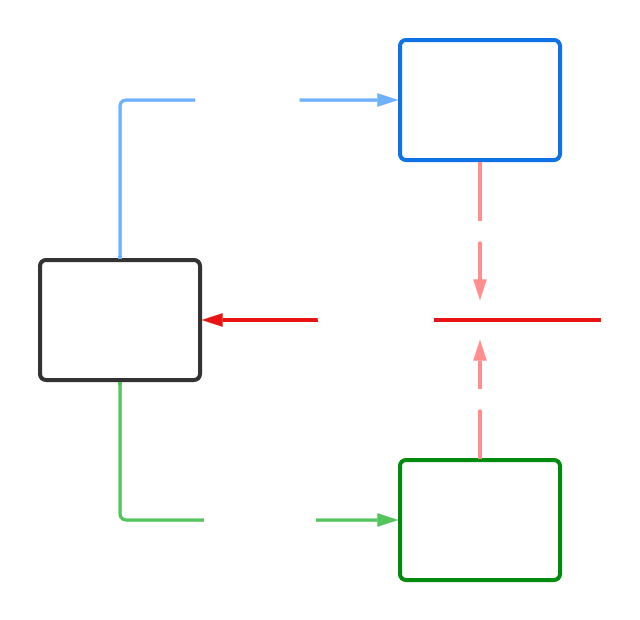
↳ rabbitmq\_controller

↳ rabbit\_queues.py

## Queues

O que iremos criar será algo relativamente simples. Como vamos ter os microcervices *users* e *orders,* nossa API terá que se comunicar com eles de algum jeito. Para isso, iremos utilizar o massenge broker do rabbit. A ideia é termos duas filas para comunicação entre API/Serviço, porém existe um pequeno detalhe. Como em alguns casos iremos fazer uma consulta no banco de dados, desejando enviar um response ao usuário. Nas filas do rabbit existe um sistema de ACK para confirmação de entrega de mensagens ao destinatário, porém é algo relativamente simples, não podendo ter uma personalização da mensagem de confirmação.

O que precisaremos fazer, será utilizar um recurso chamado Remote Procedure Call (RPC). Que consiste em um remetente enviar uma mensagem e aguardar o destinatário processar a informação, para que depois possa retornar uma resposta. Resumindo, uma comunicação assíncrona. Sendo assim, será necessária uma fila auxiliar para transmitir esse response. Abaixo há uma ilustração da arquitetura que iremos criar.



Agora que temos a arquitetura do que precisamos construir, podemos começar.

Primeiramente precisamos fazer a sequência de imports das bibliotecas, e também importar a classe da configuração do rabbit, que fizemos no passo anterior.

import pika, uuid, threading, time

from config.rabbitmq\_connection import RabbitConnection

Logo após, vamos criar uma classe chamada *RabbitQueue(),* contendo duas variáveis que iremos utilizar posteriormente.

class RabbitQueue():

internal\_lock = threading.Lock()

queue = {}

No método \_\_init\_\_.py vamos ter crias o seguinte atributo. Primeiro precisamos instanciar a classe de conexão com o servidor Rabbit para posteriormente criar as filas. Por enquanto precisamos saber só disso, os demais atributos irão ser abordados posteriormente.

def \_\_init\_\_(self) :

self.RMQ = RabbitConnection()

result = self.RMQ.channel.queue\_declare(queue='', exclusive=True)

self.callback\_queue = result.method.queue

self.\_create\_process\_thread()

Agora, iremos criar um método chamado *create\_queues()*, pra criar as filas que precisamos consumir. Para isso será utilizado a o atributo channel, da classe de conexão, e a função queue\_declare, para declarar as filas.

def create\_queues(self):

self.RMQ.channel.queue\_declare(queue='user')

self.RMQ.channel.queue\_declare(queue='order')

print(' [✓] Queues created successful!')

Vamos criar agora, o método chamado \_*create\_process\_thread(),* ele será responsável por criar uma thread de processamento, fazendo com que o consumo de diferentes processos sejam independentes.

def \_create\_process\_thread(self):

thread = threading.Thread(target=self.\_process\_data\_events)

thread.setDaemon(True)

thread.start()

No metodo *rpc\_async()*, terá como parâmetros o **payload**, que será passado a informação que será lançado na fila, e o parâmetro **route,** que será o nome da fina que será utilizada.

\*\*verificar - Nesse método iremos fazer a configuração da fila de controle, que citei anteriormente. Para isso, será criado um ID para fila, utilizando a biblioteca *uuid.*

No with, será verificado a variável *internal\_lock* que criamos, nela é atribuído um bloqueio primitivo. Uma vez que um thread adquiriu um bloqueio, as tentativas subsequentes de adquiri-lo bloqueiam, até que ele seja liberado. Após isso será passa a *route* como nome da rota, e como propriedade, há um *reply\_to*, que é um recurso que permite que clientes RPC (solicitação/resposta) evitem declarar uma fila de resposta por solicitação, comumente usado para nomear uma fila de retorno de chamada. Nele é passado o atributo *callback\_queue* que foi criado no \_\_init\_\_.

Tambem a o *correlation\_id*, que é útil para correlacionar respostas RPC com solicitações. É passado o id criado anteriormente, que será o nome da fila de controle.

E por fim, o body, contendo a informação passada através do payload.

def rpc\_async(self, payload, route):

corr\_id = str(uuid.uuid4())

self.queue[corr\_id] = None

with self.internal\_lock:

self.RMQ.channel.basic\_publish(exchange='',

routing\_key=route,

properties=pika.BasicProperties(

reply\_to=self.callback\_queue,

correlation\_id=corr\_id,

),

body=payload)

return corr\_id

No método *\_on\_response()* será apenas armazenado o response da fila em um dicionário, passa como key, a ID que foi criado no metodo *rpc\_async,* e no value o body do response.

def \_on\_response(self, ch, method, props, body):

"""On response we simply store the result in a local dictionary."""

self.queue[props.correlation\_id] = body

O método *\_process\_data\_events(),* será responsável por ser uma espécie de vigia de eventos da fila de controle, e marcar um ACK ao evento, para consumi-lo.

def \_process\_data\_events(self):

self.RMQ.channel.basic\_consume(

queue=self.callback\_queue,

on\_message\_callback=self.\_on\_response,

auto\_ack=True,

)

while True:

with self.internal\_lock:

self.RMQ.connection.process\_data\_events()

time.sleep(0.1)

## Consumo da classe RabbitQueue

Agora, só precisamos apontar a classe **RabbitQueue** no arquivo *server.py* para que possamos começar a consumir as filas que criamos.

Fora da classe *Api\_server* iremos instanciar **RabbitQueue** que criamos, a uma variável chamada *rabbit\_queues*. Ela ficará fora da classe pois queremos que ela seja vista globalmente no arquivo. Dentro da classe *Api\_server,* iremos criar o construtor \_\_init\_\_ e chamaremos o método *create\_queues,* para criar as filas.

rabbit\_queues = RabbitQueue()

class Api\_server():

app = Flask(\_\_name\_\_)

def \_\_init\_\_(self):

rabbit\_queues.create\_queues()

Uma pergunta que você pode estar se questionando é “porque não criar o atributo *app* dentro de um \_\_init\_\_?”.

Bom, na maioria dos projetos que pode ser encontrado na internet relacionada ao Flask, você simplesmente irá se deparar com um arquivo com uma rota chamando uma função abaixo, não tem muito segredo em relação a isso. Podemos compreender que o Flask não foi feito para seguir os padrões de Orientação a Objetos, tanto que é que se você tentar colocar um “self”, na def abaixo da rota, simplesmente não irá funcionar. Visto isso, apenas atribui a estrutura de rotas e funções dentro de uma classe como uma boa prática, porém, não é necessário fazer isso caso você não queria.

Agora, respondendo à pergunta, o simples fato de colocar um “self” antes do *app*, já deixaria de funcionar todo código, visto que no marcador @ não iria identificar o “self.app” que foi criado no init.

## Routes

Na chamada das routes, precisaremos invocar o método *rpc\_async* para que possemos enviar o playload para a fila desejada. Para isso, irei utilizar o mesmo exemplo que foi utilizado no Step4, com a rota /user/create\_user/.

Iremos criar chamar o método *rpc\_async,* passando como parâmetros o payload recebido na requisição, e a string “user”, que será a fila que será enviada a mensagem. Tudo isso será atribuído para uma variável chamada *corr\_id,* que terá como return o *correlation\_id*. Após isso, ficará sendo verificado o atributo *queue,* da classe **RabbitQueue,** se contêm alguma informação. Caso tenha, essa informação será retornada ao usuário

@app.route("/user/create\_user/", methods=['POST'])

def create\_user():

if request.method == 'POST':

payload = request.get\_json()

corr\_id = rabbit\_queues.rpc\_async(json.dumps(payload),"user")

while rabbit\_queues.queue[corr\_id] is None:

time.sleep(0.1)

return {'Status': 200, 'Message': json.loads(rabbit\_queues.queue[corr\_id])}

else:

return {'Status': 404, 'Message': 'Erro no envio do method'}

## Resumo da Funcionalidade

E chegamos ao fim do envio de mensagens a fila. Como podem perceber, são muitas coisas divergentes sendo processadas, enviadas e recebidas ao mesmo tempo, chegando a ser até um pouco complexo de entender de primeiro momento. Basta montar o condigo na ordem que foi mostrada acima, e olha-lo de uma visão mais panorâmica, que as peças irão se encaixar.

Para facilitar um pouco mais, vou resumir em tópicos o que vai acontecer quando enviarmos uma mensagem para alguma fila.

1. Será chamado o método **rpc\_async** passando a mensagem e a fila que deverá ser enviada.
2. No **\_\_init\_\_** da classe **RabbitQueue,** estabelecemos uma conexão com servidor Rabbit, e declaramos um **callback\_queue** exclusivamente para receber resposta da fila de controle. E é criado uma thead que ficara monitorando a fila de controle.
3. No **rpc\_async** é o que será feita a solicitação RPC.
4. Dentro do **rpc\_async** é gerado um **correlation\_id** exclusivo e o salvamos. O método **\_on\_response** usará esse valor para criar um dicionário, que será atribuído a *key.*
5. Também no método **rpc\_async,** publicamos a mensagem com duas propriedades, *reply\_to* e *correlation\_id*.
6. O **callback\_queue** será marcado para que possamos receber respostas RPC.
7. O **\_process\_data\_events**, terá a função de ficar monitorando a fila de controle para poder gravar o response do parâmetro *on\_message\_callback*, passado para o **\_on\_response**.
8. No **\_on\_response**, será montado um dicionário, que irá conter o *correlation\_id* e o *body* capturado da fila de controle. O body será o response da fila.
9. Por fim, no arquivo *server.py,* será verificado se a variável *queue* contém alguma informação para ser repassada como response da requisição a API.