Sumário

[Um projeto completo de Microservices 4](#_Toc104656395)

[Entendimento do problema 4](#_Toc104656396)

[Objetivo 4](#_Toc104656397)

[Arquitetura do Projeto Completo 5](#_Toc104656398)

[Step 1 7](#_Toc104656399)

[Criação do Compartment 7](#_Toc104656400)

[Criação de Usuário 7](#_Toc104656401)

[Criação de Grupo 8](#_Toc104656402)

[Criação de Policies 9](#_Toc104656403)

[Criação de SSH-KEY 10](#_Toc104656404)

[Criação de Instância VM 11](#_Toc104656405)

[Resumo 11](#_Toc104656406)

[Step 2 12](#_Toc104656407)

[Entrar na Instância via shell 12](#_Toc104656408)

[Instalar Docker 12](#_Toc104656409)

[Instalar Docker-Compose 13](#_Toc104656410)

[Instalar Git 13](#_Toc104656411)

[Resumo 13](#_Toc104656412)

[Step 3 14](#_Toc104656413)

[Criação do Docker-Compose-Services 14](#_Toc104656414)

[Teste na OCI 17](#_Toc104656415)

[Resumo 19](#_Toc104656416)

[Spet4 20](#_Toc104656417)

[Requirements 20](#_Toc104656418)

[Criação API 20](#_Toc104656419)

[Dockerfile 22](#_Toc104656420)

[Docker-Compose 22](#_Toc104656421)

[Resumo 23](#_Toc104656422)

[Step5 24](#_Toc104656423)

[Postgres 24](#_Toc104656424)

[Create\_tables 25](#_Toc104656425)

[Rabbit MQ 27](#_Toc104656426)

[Redis 28](#_Toc104656427)

[Resumo 28](#_Toc104656428)

[Step6 29](#_Toc104656429)

[Queues 29](#_Toc104656430)

[Consumo da classe RabbitQueue e ConnectionDatabase 32](#_Toc104656431)

[Routes 33](#_Toc104656432)

[Resumo da Funcionalidade 34](#_Toc104656433)

[Step7 37](#_Toc104656434)

[Config 37](#_Toc104656435)

[Requeriments 37](#_Toc104656436)

[Docker-Compose 37](#_Toc104656437)

[Dockerfile 38](#_Toc104656438)

[Main.py 38](#_Toc104656439)

[Resumo 39](#_Toc104656440)

[Step8 40](#_Toc104656441)

[Tag API 40](#_Toc104656442)

[Controlador DataBase 41](#_Toc104656443)

[Manipulação de datas 41](#_Toc104656444)

[Insert\_user 42](#_Toc104656445)

[Show\_all\_user 43](#_Toc104656446)

[Métodos Criados 44](#_Toc104656447)

[Controlador Rabbit 44](#_Toc104656448)

[Resumo 45](#_Toc104656449)

[Step9 47](#_Toc104656450)

[Argon2 47](#_Toc104656451)

[Hash de Senha 47](#_Toc104656452)

[Encapsulamento 47](#_Toc104656453)

[Controlador DataBase 49](#_Toc104656454)

[Resumo 51](#_Toc104656455)

[Step10 52](#_Toc104656456)

[Config 52](#_Toc104656457)

[Requeriments 52](#_Toc104656458)

[Docker-Compose 52](#_Toc104656459)

[Dockerfile 53](#_Toc104656460)

[Main.py 53](#_Toc104656461)

[Resumo 54](#_Toc104656462)

[Step11 55](#_Toc104656463)

[Tag API 55](#_Toc104656464)

[Controlador DataBase 56](#_Toc104656465)

[Manipulação de datas 56](#_Toc104656466)

[Create\_order 56](#_Toc104656467)

[List\_all\_orders 57](#_Toc104656468)

[Métodos Criado 58](#_Toc104656469)

[Controlador Rabbit 58](#_Toc104656470)

[Resumo 60](#_Toc104656471)

[Step12 61](#_Toc104656472)

[Instalação 61](#_Toc104656473)

[Inicialização do Redis 61](#_Toc104656474)

[Decoratos Cached 62](#_Toc104656475)

[Exemplo 62](#_Toc104656476)

[Finalização 63](#_Toc104656477)

# Um projeto completo de Microservices

Há um tempo eu desejava entender melhor como os tão famosos microsserviços conseguem trabalhar individualmente, porém todos conectados. Foi então que decidi projetar uma pequena aplicação onde iria me aprofundar nesses conhecimentos, e que também me desafiaria a criar uma aplicação completa seguindo o **Software Development Life Circle**. Partindo desde a criação da ideia, até o deploy.

Isso pode ser representado conforme o seguinte fluxo:



Então, meu nome é Patrick Berlatto Piccini, e esse é meu projeto completo de microservices.

## Entendimento do problema

Nesse projeto, utilizarei algumas ferramentas contidas na OCI (Oracle Claud Infraestructure) visto que, recentemente (Fevereiro 2022) passei na certificação “Oracle Cloud Infrastructure Foundations 2021 Associate”. Se você desejar seguir os passos da criação da aplicação e desenvolver junto comigo o código, é opcional utilizar o OCI.

Vamos começar...

### Objetivo

Criar duas aplicações básicas de microsserviços:

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, full\_name, password, cpf, email, phone\_number, created\_at, updated\_at.

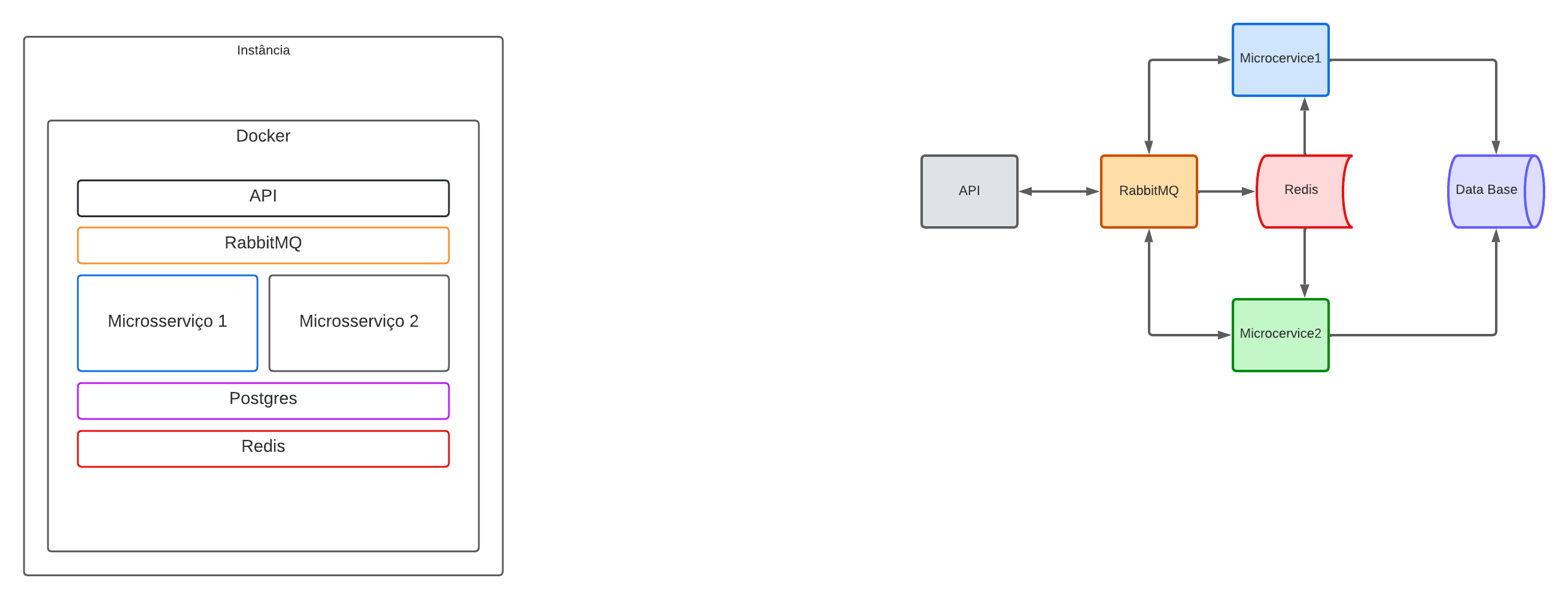
O segundo será um serviço de OS (ordem de serviço) que deverá conter no cadastro, o user\_id do usuário contido no banco de dados. Deverá ter as seguintes informações:

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

A arquitetura da aplicação será a seguinte: criaremos uma API que será responsável por distribuir as requisições através de um broker de mensagens chamado RabbitMQ, e também criar as filas e tabelas necessárias para a aplicação.

Nesse broker, terão duas filas onde a API fará a separação das mensagens e enviará ao seu devido destino, onde teremos dois microsserviços, uma para *usuários*, e outro para os *orders*. Cada microsserviço é conectado a um banco de dados Postgres onde serão armazenadas as informações dos *usuários* e dos *orders.*

Juntamente a API, haverá uma camada de memória cache onde utilizaremos o Redis para fazer essa função. Então, caso uma requisição já tenha sido feita, a API irá verificar antes nos dados em Cache se já existe essa informação. Assim, o usuário terá o retorno muito mais rápido.

**

## Arquitetura do Projeto Completo

MS-application

│ .gitignore

│ docker-compose-services.yml

│

├───API

│ │ docker-compose-api.yml

│ │ Dockerfile

│ │ requirements.txt

│ │ server.py

│ │

│ ├───config

│ │ database\_connection.py

│ │ rabbitmq\_connection.py

│ │ redis\_connection.py

│ │ \_\_init\_\_.py

│ │

│ └───rabbitmq\_controller

│ rabbit\_queues.py

│ \_\_init\_\_.py

│

├───MS1

│ │ docker-compose-microservice1.yml

│ │ Dockerfile

│ │ main.py

│ │ requirements.txt

│ │

│ ├───config

│ │ database\_connection.py

│ │ rabbitmq\_connection.py

│ │ \_\_init\_\_.py

│ │

│ ├───criptografy

│ │ hash\_password.py

│ │ \_\_init\_\_.py

│ │

│ ├───database\_controller

│ │ postgres\_worker.py

│ │ \_\_init\_\_.py

│ │

│ └───rabbitmq\_controller

│ rabbit\_worker.py

│ \_\_init\_\_.py

│

└───MS2

│ docker-compose-microservice2.yml

│ Dockerfile

│ main.py

│ requirements.txt

│

├───config

│ database\_connection.py

│ rabbitmq\_connection.py

│ \_\_init\_\_.py

│

├───database\_controller

│ postgres\_worker.py

│ \_\_init\_\_.py

│

└───rabbitmq\_controller

rabbit\_worker.py

\_\_init\_\_.py

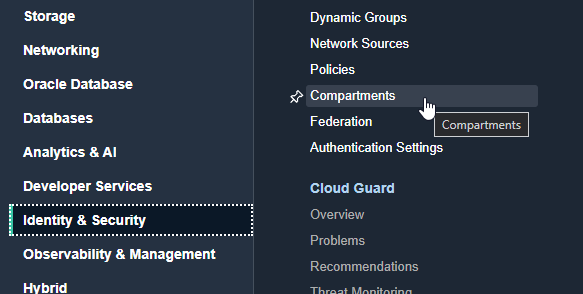
# Step 1

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

## Criação do Compartment

Criaremos um *conpartiment.*

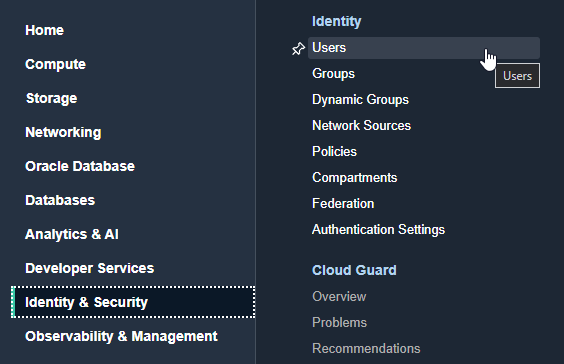
No ícone de hamburguer na página inicial do OCI, iremos em *> Identity & Security > compartments, e* criaremos um compartimento com o nome **Developmet.**

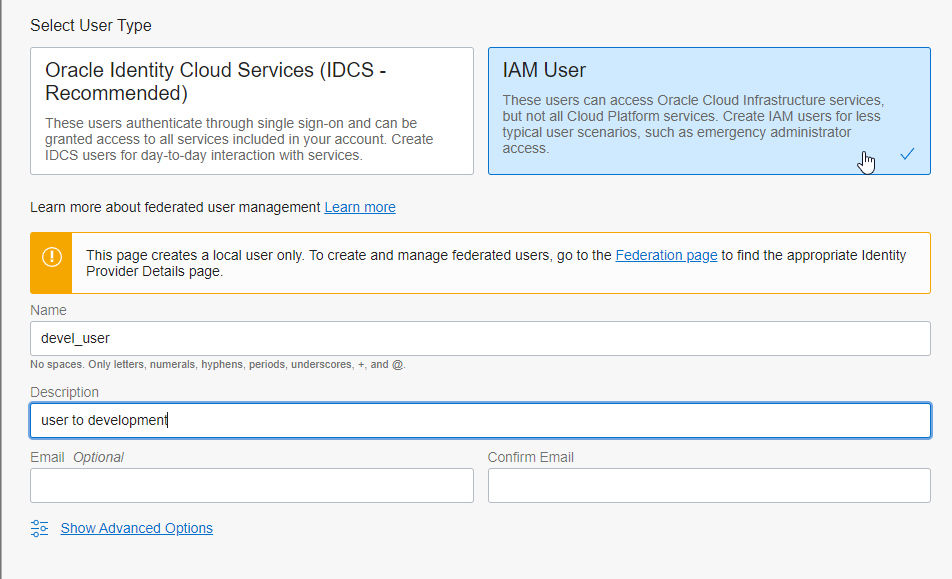


## Criação de Usuário

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

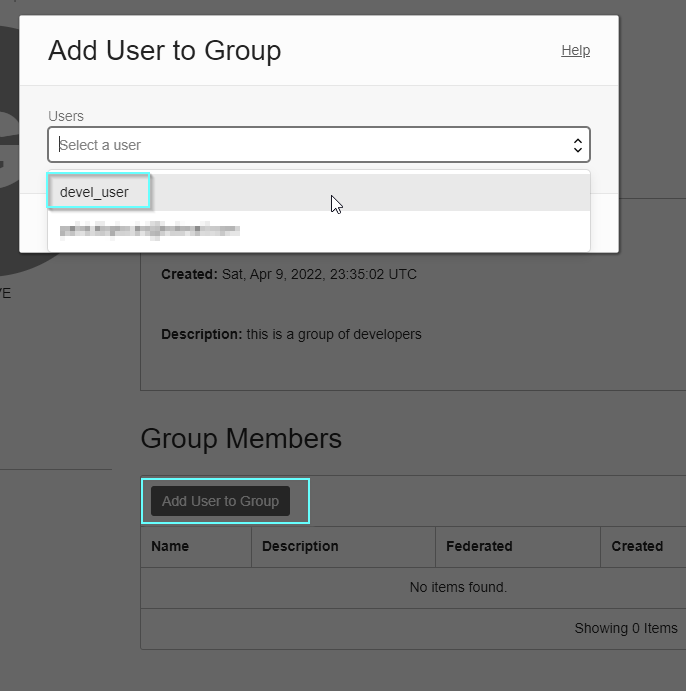
Novamente no menu lateral, em *> Identity & Security > users*, iremos criar o usuário **devel\_user,** com o tipo IAM USER. Esses usuários podem acessar os serviços do Oracle Cloud Infrastructure, mas nem todos os serviços do Cloud Platform. Os usuários do IAM são cenários de usuário atípico, como acesso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





## Criação de Grupo

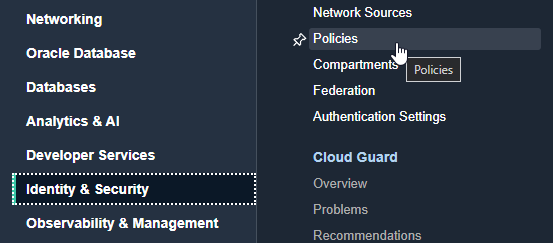
No menu lateral, em *> Identity & Security > group*s, criaremos um grupo com o nome de **Developer\_Group.**  Depois disso, iremos adicionar o usuário **devel\_user** ao grupo clicando no botão “Add user to Group”.



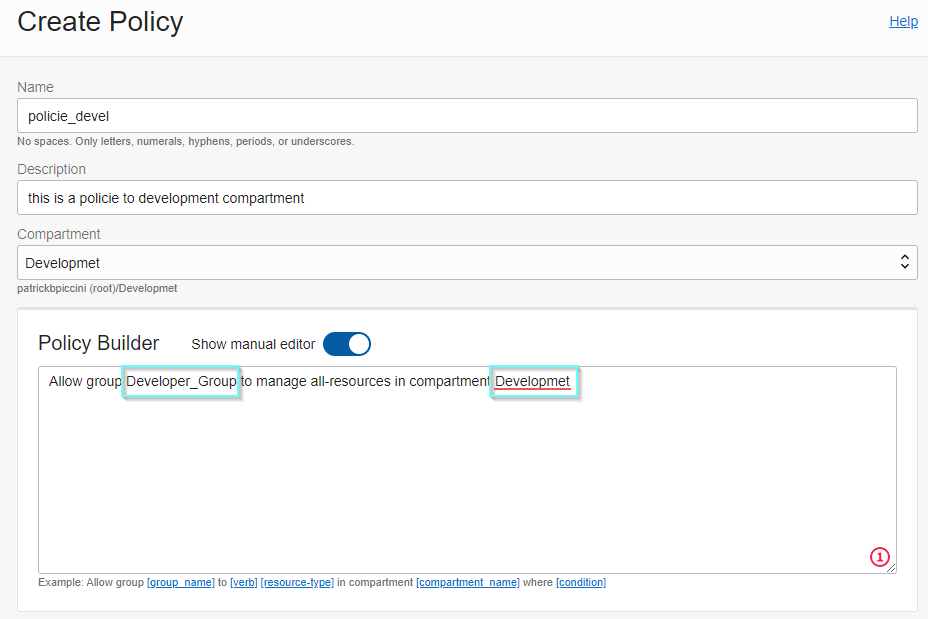
## Criação de Policies

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

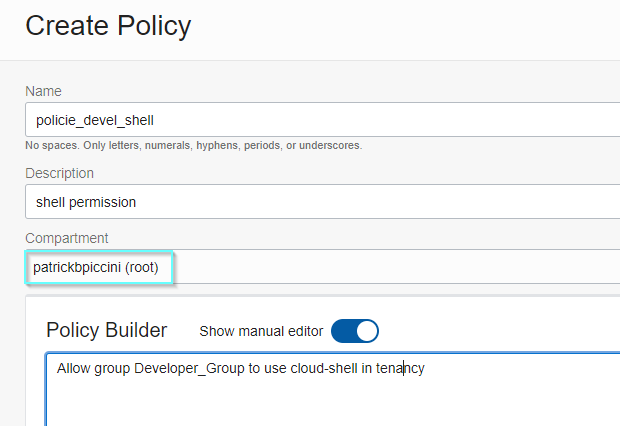
Vamos em *> Identity & Security > Policies*



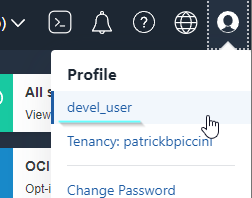
Após ser direcionado a página de policies, selecionamos o compartimento **Developmet** para receber a política que criamos. Na policie, iremos permitir o Developer\_Group (e todos seus usuários) a utilização de todos os recursos no compartimento OCI.



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 precisaremos de uma ssh-key disponibilizada pelo usuário para conseguir criar nossa instancia no OCI.

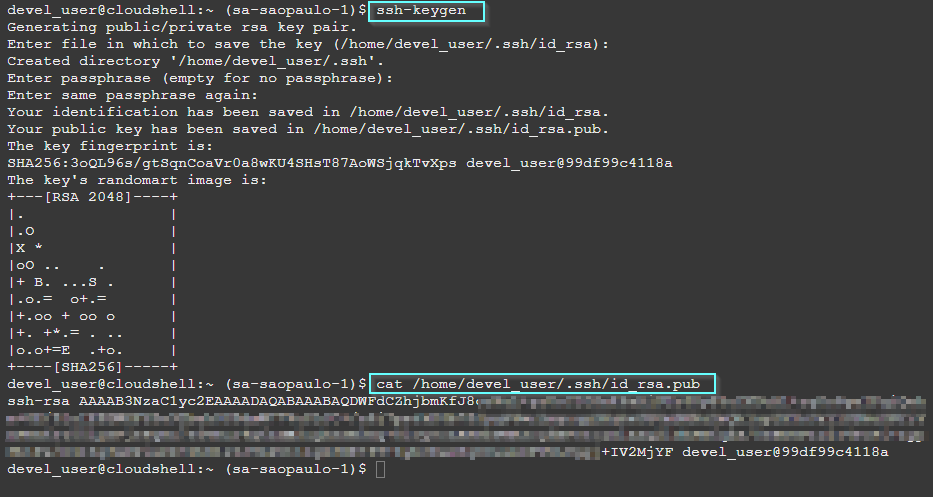


## Criação de SSH-KEY

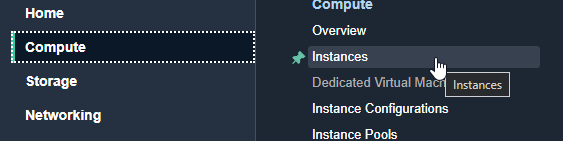
**Logado no usuário *devel\_user,* conectaremos a Oracle Cloud Shell para adquirir uma ssh\_key que posteriormente iremos utilizar. Para isso, vamos usar alguns simples comandos para a criação dessa chave.

ssh-keygen – Para criação da chave

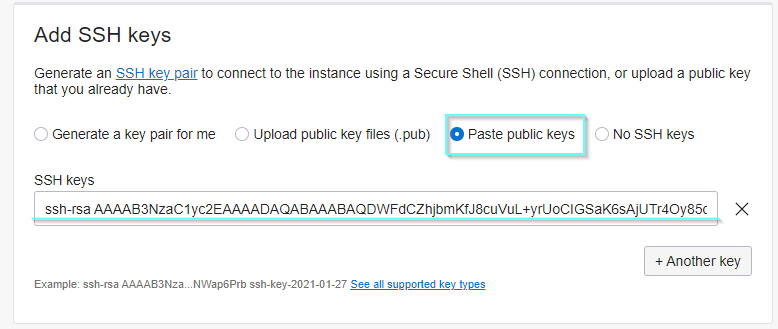
cat /home/your\_user/.ssh/id\_rsa.pub – Mostrará o conteúdo contido no arquivo id\_rsa.pub, que foi criado com o comando anterior



## Criação de Instancia VM

Ainda conectado ao usuário *devel\_user* criaremos uma instância, que nada mais é que uma Virtual Machine. Em > *compute > instances > create instance.*

Ao criar uma instância na página de configuração da VM, devemos colar 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 instância que irei utilizar, são disponibilizadas pelo serviço *Oracle Cloud – Free Tier,* sendo ela 1 VM 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/>

## Resumo

Nesse Step, começamos as criar o ambiente onde rodaremos nossa aplicação. Configuramos um compartment, usuário e grupo de usuário, que será o responsável pelo acesso ao desenvolvimento na instância OCI. Definimos algumas políticas de usuário. Também criamos uma ssh-key que utilizamos para a criação da instancia OCI.

# Step 2

Após Configurar a infraestrutura no Oracle Cloud, necessitamos fazer instalações de alguns programas que iremos utilizar. 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 código.

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 computador 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

## Resumo

Nesse Step, nos conectamos a instância da OCI e instalamos algumas dependências para o nosso ambiente de produção. Instalamos o Docker, Docker-Compose e Git.

# 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.

Devemos começar criando uma pasta chamada *MS-Application.* Essa pasta será onde iremos criar todos os microsserviços, configuração de containers de serviço, e mais alguns detalhes, que serão mostrados no decorrer das publicações.

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

**MS-Application**

**│ docker-compose-services.yml**

## Criação do Docker-Compose-Services

Como vamos 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 microsserviços.

Para inicializar esses serviços rapidamente, iremos desenvolver um Docker-Compose e configuraremos uma *Network*, onde posteriormente estarão todos os serviços dentro dessa mesma rede.

A rede foi nomeada de *internal-network,* nela, iremos configurar o endereço 10.5.0.0 sendo uma rede classe A, com uma máscara de rede /16. Também iremos definir o gateway da rede para 10.5.0.1.

version: "3.7"

networks:

network-service:

driver: bridge

name: internal-network

ipam:

config:

- subnet: 10.5.0.0/16

gateway: 10.5.0.1

Nos serviços, iniciamos configurando o RabbitMQ. Utilizaremos 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 host, que será utilizado posteriormente para conectar ao manager do rabbit. Nas portas, devemos colocar 15672 para o manager e 5672 para o servidor. E por fim, iremos configurar 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, utilizaremos a imagem *postgres:13*, iremos indicar as variáveis padrões do user e password para conexão a instância, apontaremos a porta padrão 5432, criaremos um volume compartilhado de *.data* para */data/,* configuraremosaNetwork com o ip estático, e um detalhe muito importante, será colocado o parâmetro *restart: Always* que fará com que a instância 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, faremos a criação de uma instância com o PgAdmin, que poderemos conectar ao servidor do postgres via Web. Sendo assim, iremos utilizar a imagem *dpage/pgadmin4,* com as variáveis de e-mail e password para entrarmos na página web. Iremos apontar a porta de 16543/80 (external/internal), compartilharemos os volumes./data/:/data/ e ./postgres-backup:/var/lib/postgresql/backups, iremos configurar a network com um ip estático, e colaremos uma dependência no parâmetro *depends\_on – postgres.* Isso fará com o a instância do PgAdmin só seja criada, após a criação da instância 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, iremos criar uma instância com Redis utilizando a imagem *redis:alpine*, apontando a porta padrão 6379, e configurando 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, teremos vários containers rodando dentro de uma Rede, tendo algo parecido com a seguinte imagem:

Diagrama

Descrição gerada automaticamente

## Teste na OCI

Agora que temos pronto o Docker-Compose com todos os serviços que utilizaremos, devemos fazer 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 instância OCI dou um *git pull* para baixar.

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

Dentro a instância, 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 começará a baixar as imagens dos contêineres.

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, sejam 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. iremos adicionar as portas que utilizaremos.

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

## Resumo

Nesse Step, criamos o arquivo *docker-compose-services.yml* que é responsável por iniciar os servidores RabbitMQ, Postgres e Redis, juntamente as interfaces PgAdmin e RabbitManager. Foram liberadas as portas dos servidores para visualização externa, e também foi feito um teste acessando as interfaces externamente à OCI, tendo sucesso na conexão.

# Spet4

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

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

MS-application

└───API

│ docker-compose-api.yml

**│ Dockerfile**

**│ requirements.txt**

**│ server.py**

## Requirements

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

*requirements.txt*

psycopg2-binary

Flask-Caching

threaded

flask

redis

pika

uuid

Comando para instalação:

> pip install -r requeriments.txt

## Criação API

Dentro o arquivo *server.py* iremos 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, e **show\_all\_user** para mostrar todos os usuários. A mesma lógica será aplicada referente aos **orders**, como **create\_order, list\_order** e assim sucessivamente.

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

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

Devemos criar uma classe chamada **Api\_server** e instanciar o Flask a variável **app.** Após isso, para definir uma rota iremos utilizar o @app.route(“rota”, method), onde apontaremos *"/user/create\_user/"* como destino*,* sendo o **POST** o único método que essa rota irá aceitar**.**

Abaixo da rota fica a função que será disparada após ser acessada, que nomearemos de **create\_user().** Dentro da ‘def’, faremos uma verificação se o method é o mesmo que esperamos 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(): # <- metodo 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 iremos instanciar a classe que criamos, e utilizaremos 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 ao 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, basta ir ao repositório da aplicação e olhar as outras rotas.

## Dockerfile

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

FROM python:3.8-alpine <- Imagem base utilizada

COPY . /app/ <- cópia 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='10.0.0.25' <- ip do redis

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

ENV CACHE\_REDIS\_DB=0

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

CMD 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 da API, e apontamento da Network que foi criada no Docker-Compose, onde 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

## Resumo

Nesse Step, criamos o arquivo *requeriments.txt* que é responsável por conter todas as bibliotecas que iremos utilizar. Foi explicado como funciona o sistema de rotas do Flask e também criamos uma rota como exemplificar. Por fim, desenvolvemos os arquivos Dockerfile e Docker-Compose, que serão responsáveis pela inicialização da API.

# Step5

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

Nesse passo, mostrarei como configurar a conexão com os servidores do Rabbit, Redis e Postgres, e como planejei a arquitetura de pastas e módulos para ter uma aplicação organizada.

Dentro da pasta API, devemos criar uma subpasta chamada config com os seguintes arquivos:

MS-application

└───API

│ docker-compose-api.yml

│ Dockerfile

│ requirements.txt

│ server.py

│

**├───config**

**│ database\_connection.py**

**│ rabbitmq\_connection.py**

**│ redis\_connection.py**

**│ \_\_init\_\_.py**

## Postgres

Antes de continuar, deve-se ter certeza de que executou todas as instalações das bibliotecas, contidas 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 Postgres para nossa aplicação, na criação do Dockerfile foi criado uma variável de ambiente chamada HOST\_DATABASE, contendo o ip privado da instância 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 de desenvolvimento em Homologação, utilizando os serviços que já estão rodando.

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

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

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

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

HOST\_DATABSE = '144.22.193.219'

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()

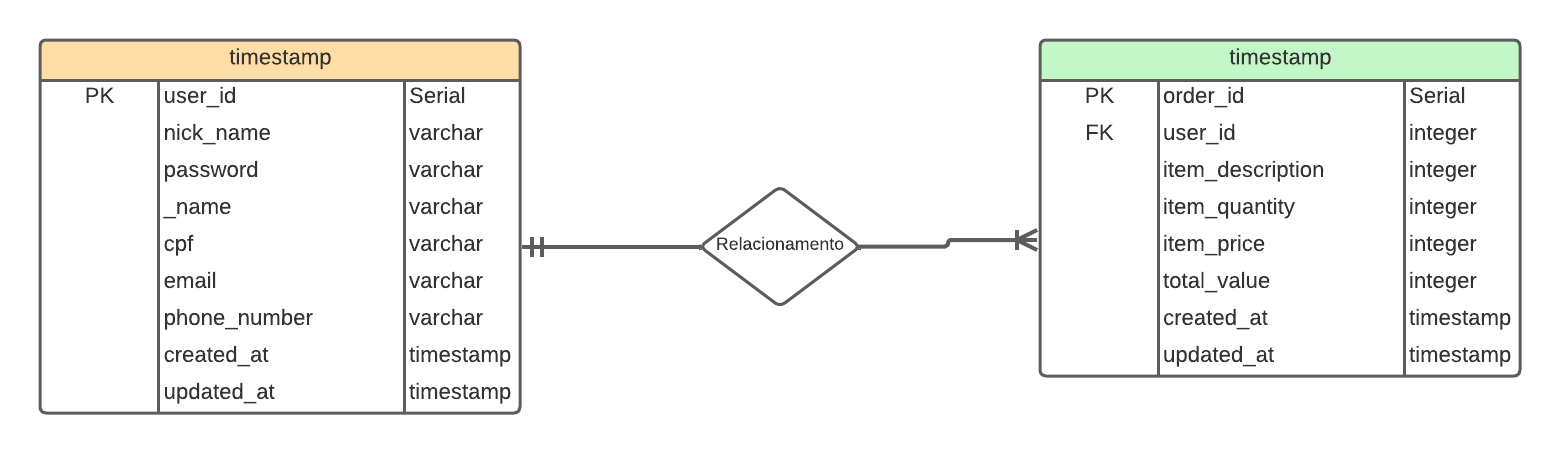
print('[✓] Connected to Postgres')

except Exception as error:

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

### Create\_tables

Antes de começar a fazer a função que criará as tabelas, vamos analisar o que precisará ser feito com o modelo ER abaixo:



Iremos criar uma estrutura simples. Serão duas tabelas, *users* e *orders*. Nelas conterão 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 é o *user\_id*, que será tratado como uma chave estrangeira (ForeignKey).

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

Tela de celular com texto preto sobre fundo branco

Descrição gerada automaticamente com confiança média

Após essa análise, devemos criar o método *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 criaremos um select da versão do Postgres, para que toda a vez que seja iniciado a conexão, seja mostrado a versão. Logo após, devemos criar 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\_\_,* será criado as tabelas ao instanciar a classe **ConnectionDatabase***.*

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,

full\_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 muitos outros recursos, porém, não irei detalhar eles nesse projeto.

Nesse momento, faremos 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 público, somente para desenvolver a aplicação local, utilizando o servidor que já estão 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\_DATABSE = '144.22.193.219'

Agora iremos criar uma classe chamada *RabbitConnection(),* e faremos a conexão no método *\_\_init\_\_.* Utilizando o função de conexão da biblioteca pika chamada *BlockingConnection(),* passaremos como parâmetros, através do metodo *ConnectionParameters()* o host, contida na variável 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, utilizaremos a variável de ambiente CACHE\_REDIS\_HOST, porém, para a fim de desenvolvimento, criarei uma variável local contendo o ip público da instancia.

Nessa configuração, iremos fazer algo um pouco diferente. Como o Redis e Flask conseguem trabalhar juntos ativamente, a configuração para a conexão ao servidor, será feita através do arquivo principal *server.py*. Contudo, será demonstrado 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 testarei localmente para homologação, criarei a mesma classe, porém, com atributos com valor estático do servidor.

class BaseConfig(object):

CACHE\_TYPE = 'redis'

CACHE\_REDIS\_HOST = '144.22.193.219'

CACHE\_REDIS\_PORT = '6379'

CACHE\_REDIS\_DB = 0

## Resumo

Nesse Step, criamos as conexões com os servidores Rabbit, Redis e Postgres que já estavam rodando a instancia da OCI. Criamos a conexão com o Postgres. Foi explanado o modelo de entidade relacional que posteriormente foi desenvolvido, criando as devidas tabelas planejadas. Vimos um breve resumo do que é o rabbit e criamos a conexão com o servidor que está rodando na OCI. E por fim, apontado a conexão com o Redis, porém não utilizaremos nesse momento.

# Step 6

Nessa etapa, criaremos as filas que necessitamos para a comunicação com os microservices. Iremos instanciar a classe de configuração do Rabbit para inicializar juntamente com a API, estabelecendo uma conexão, e criando automaticamente as filas para consumo. Veremos também como funciona o recurso RPC, que irei detalhar posteriormente como funciona. Também, será explicado cada funcionalidade da criação da fila do rabbit que utilizaremos. Por fim, mostrarei no manager, a parte visual de como irá ficar nossas filas. No final dessa etapa, será resumido como funcionará o código, por uma visão geral.

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

MS-application

└───API

│ docker-compose-api.yml

│ Dockerfile

│ requirements.txt

│ server.py

│

├───config

│ database\_connection.py

│ rabbitmq\_connection.py

│ redis\_connection.py

│ \_\_init\_\_.py

**│**

**└───rabbitmq\_controller**

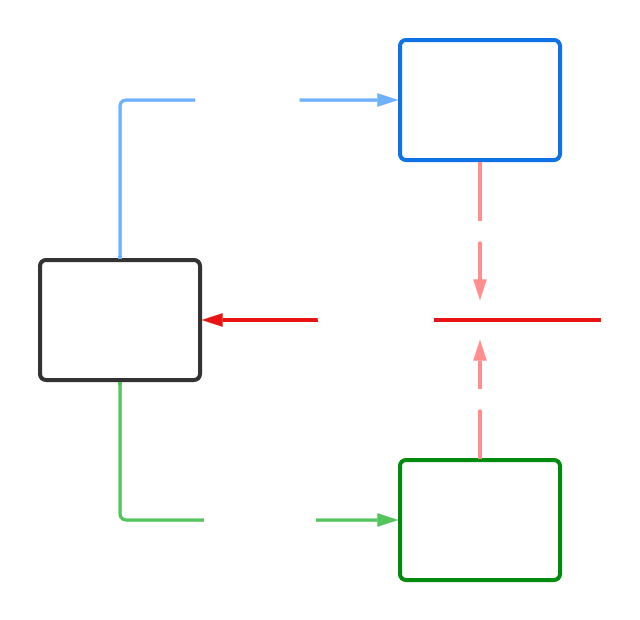
**rabbit\_queues.py**

**\_\_init\_\_.py**

## Queues

O que iremos criar será algo relativamente simples. Como vamos ter os microservices *users* e *orders,* nossa API terá que se comunicar com eles de alguma maneira. Para isso, utilizaremos o message broker do rabbit. A ideia é termos duas filas para comunicação entre API/Serviço, porém existe um pequeno detalhe, como em alguns casos que 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 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 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, criaremos 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 seja independente.

def \_create\_process\_thread(self):

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

thread.setDaemon(True)

thread.start()

No método *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 fila que será utilizada.

\*\*verificar - Nesse método faremos 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 passada a *route* com o 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 usados para nomear uma fila de retorno de chamada. Nele é passado o atributo *callback\_queue* que foi criado no \_\_init\_\_.

Também há 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, passado 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 e ConnectionDatabase

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

Antes de tudo, devemos importar todas as bibliotecas que precisaremos.

import json,time

from flask import Flask, request

from config.database\_connection import ConnectionDatabase

from rabbitmq\_controller.rabbit\_queues import RabbitQueue

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,* criaremos o construtor \_\_init\_\_ e chamaremos o método *create\_queues,* para criar as filas, e também iremos chamar a classe **ConnectionDatabase**,criada no Step5. Isso será responsável por se conectar aos dois servidores de serviço e criar as tabelas no banco, e filas no rabbit.

rabbit\_queues = RabbitQueue()

class Api\_server():

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

ConnectionDatabase()

rabbit\_queues.create\_queues()

Uma pergunta que você pode estar se questionando é “porque não criar o atributo *app* e a criação das filas do rabbitdentro de um \_\_init\_\_?”.

Bom, na maioria dos projetos que podem ser encontrado na internet relacionada ao Flask, você simplesmente irá se deparar com um arquivo com uma rota chamando uma função abaixo, sem segredos 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 queira.

Agora, respondendo à pergunta, o simples fato de colocar um “self” antes do *app* ou da criação das filas, já faria com que todo o código parasse de funcionar, 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 possamos enviar o playload para a fila desejada. Para isso, utilizarei o mesmo exemplo que foi utilizado no Step4, com a rota /user/create\_user/.

Chamaremos 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

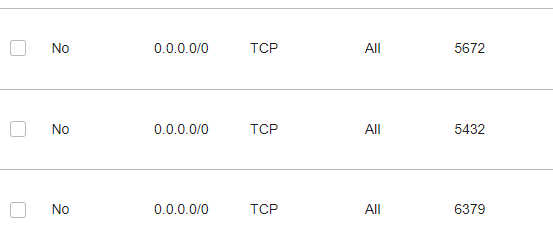
E chegamos ao fim do envio de mensagens a fila. Como podem perceber, são muitas coisas divergentes sendo processadas, enviadas e recebidas ao mesmo tempo, podendo ser um pouco complexo de entender inicialmente. Basta montar o condigo na ordem que foi mostrada acima, e olhá-lo de uma visão panorâmica,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

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

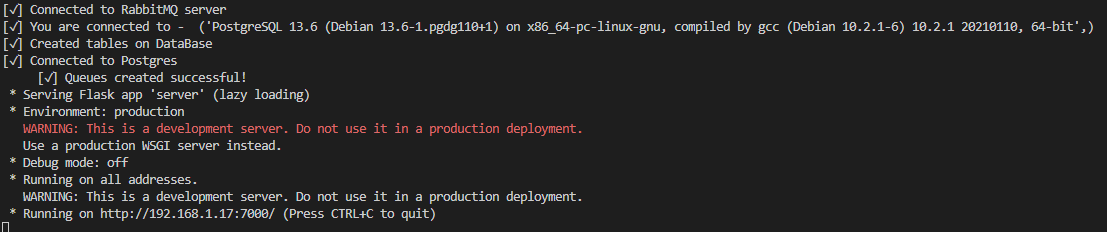
1. Será chamado o método **rpc\_async** passando a mensagem e a fila que deverá ser enviada.
2. No **\_\_init\_\_** da classe **RabbitQueue,** estabelecemos uma conexão com servidor Rabbit, e declaramos um **callback\_queue** exclusivamente para receber resposta da fila de controle. É criada uma thead que ficará monitorando a fila de controle.
3. No **rpc\_async** é onde 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.

Agora que já fizemos tanto a parte de conexão e criação das tabelas do banco de dados, quanto a conexão e criação das filas do rabbit, vou fazer um teste inicializando a API, verificando se todas as coisas que criamos estão rodando como deveriam.

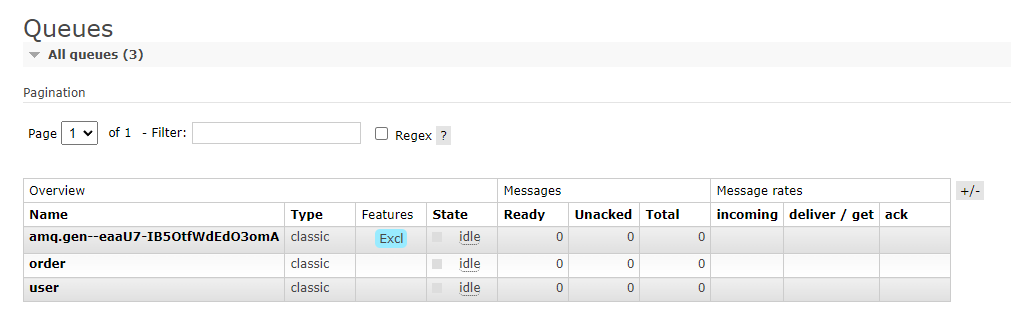
Como quero testar utilizando os servidores que já estão rodando na instância OCI, liberei as portas dos serviços para que eu possa fazer o teste localmente de meu computador.



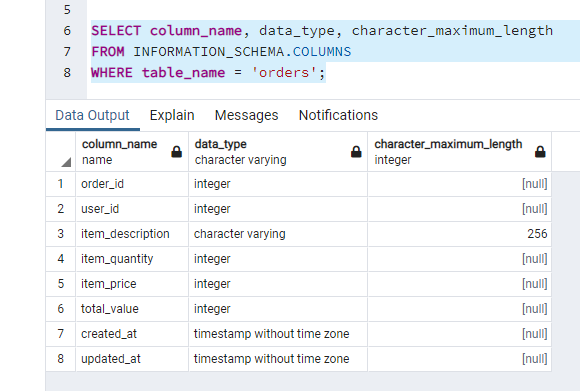
Após rodar a API aparecerão algumas informações como a conexão com o rabbit e a criação das filas feitas, e também informações sobre o Flask.

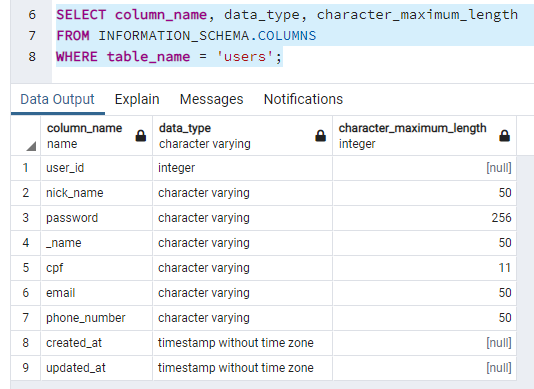


Já na Manager do rabbit podemos ver a criação das filas *order* e *user*, juntamente com a fila de controle.



E no PgAdmin fiz um select para verificar se as tabelas foram criadas.





# Step7

Agora vamos para o próximo nível da nossa aplicação. Como já temos a API pré-configurada com as filas e as tabelas do banco criados, começaremos a criar os microservices para consumi-las. A arquitetura do serviço será bem semelhante ao que criamos na API, contendo *requirements*, *dockerfile*, *docker-compose*, pasta de *config* e mais algumas coisas.

Na pasta raiz da aplicação MS-Application, vamos criar os seguintes arquivos:

MS-application

└───MS1

**│ docker-compose-microservice1.yml**

**│ Dockerfile**

**│ main.py**

**│ requirements.txt**

**│**

**├───config**

**│ database\_connection.py**

**│ rabbitmq\_connection.py**

**│ \_\_init\_\_.py**

## Config

Na pasta **config**, os arquivos *database\_connection.py* e *rabbitmq\_connection.py* são exatamente as mesmas configurações dos arquivos contido na API, então apenas vamos copiar aos arquivos da API.

## Requeriments

No arquivo *requeriments.txt* ficarão todas as instalações de bibliotecas e framework que serão utilizados na aplicação do microservices. As bibliotecas devem ficar separadas por linha, visto que usaremos um comando para uma instalação recursiva.

*requirements.txt*

psycopg2-binary

argon2-cffi

flask

pika

Comando para instalação:

> pip install -r requeriments.txt

## Docker-Compose

No arquivo *docker-compose-microservice1.yml,* configuraremos a inicialização da imagem do serviço. Será feito o Build da imagem que criaremos posteriormente, apontado a Network que criamos juntamente aos serviços logo no início do projeto.

version: "3.7"

services:

microservice1:

build: .

networks:

- internal-network

networks:

internal-network:

external: true

## Dockerfile

No arquivo Dockerfile iremos configurar a imagem de nosso primeiro microservices, contendo as variáveis de ambiente adequadas para o serviço. Diferente da imagem do python que usamos na API, no microservice precisaremos instalar a imagem completa do python, sendo ela: python:3.8

FROM python:3.8

COPY . /app/

ENV HOST='10.0.0.25'

ENV HOST\_DATABASE='10.5.0.11'

WORKDIR /app

RUN pip install --upgrade pip

RUN pip install -r requirements.txt

CMD python main.py

## Main.py

O arquivo *main.py* será responsável por interceptar as mensagens enviadas para a fila de usuário.

Primeiramente, devemos importar os arquivos de conexão com o postgres e rabbit. Após isso, criaremos a classe *Main(),*  onde no método *\_\_init\_\_,* chamaremos as conexões com os servidores.

from config.database\_connection import ConnectionDatabase

from config.rabbitmq\_connection import ConnectionRabbitMq

from rabbitmq\_controller.rabbit\_worker import RabbitWorker

class Main():

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

self.PSQL = ConnectionDatabase()

self.RMQ = ConnectionRabbitMq()

self.RMQ\_WORKER = RabbitWorker()

Feito isso, criaremos um método chamado *consume\_queue,* que consumirá a fila que desejamos. Nesse caso será *user,* e posteriormente o consumo disparará uma função para o processamento do dado recebido da fila. Nesse momento não iremos criar a função de processamento, porem iremos apenas nomeá-la no lugar correto.

Com o comando *basic\_consume* consumiremos a fila *user*, no on\_message\_callback será disparado a função para o processamento da informação recebida na fila.

Com o comando *start\_consuming,*  é processado o evento de I/O até que todos as mensagens sejam processadas.

def consume\_queue(self):

self.RMQ.channel.basic\_qos(prefetch\_count=1)

self.RMQ.channel.basic\_consume(queue='user', on\_message\_callback=self.RMQ\_WORKER.callback)

print(' [⇄] Waiting for messages. To exit press CTRL+C')

self.RMQ.channel.start\_consuming()

Finalizaremos com a chamada do método *consume\_queue* para inicializar o serviço.

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

MA = Main()

MA.consume\_queue()

## Resumo

Nesse Step criamos nosso primeiro microservices para o processamento dos dados recebidos através da fila de usuário. Copiamos os arquivos de conexão *database\_connection.py* e *rabbitmq\_connection.py* da API, visto que é a mesma conexão que precisaremos fazer nos microservices. Também criamos o arquivo *main.py* que será o responsável por inicializar o serviço e disparar o método *consume\_queue,* que como o próprio nome diz, irá consumir a fila de usuários.

# Step8

No passo anterior concluímos a arquitetura do Microsserviço, e fizemos a conexão com os serviços do Rabbit e Postgres. Agora, vamos fazer a lógica do sistema de usuário, onde iremos diferenciar as interações com o banco de dados através de uma tag, enviada pela API, e dependendo da interação de for solicitada, haverá o acionamento de uma função equivalente a solicitação.

Na pasta raiz da aplicação MS-Application, vamos criar os seguintes arquivos:

MS-application

└───MS1

│ docker-compose-microservice1.yml

│ Dockerfile

│ main.py

│ requirements.txt

│

├───config

│ database\_connection.py

│ rabbitmq\_connection.py

│ \_\_init\_\_.py

│

**├───database\_controller**

**│ postgres\_worker.py**

**│ \_\_init\_\_.py**

**│**

**└───rabbitmq\_controller**

**rabbit\_worker.py**

**\_\_init\_\_.py**

## Tag API

Antes de começarmos a codar, vamos entender o cenário onde paramos. Temos uma API que recebe as informações de uma criação de usuário, por exemplo, e envia essas informações para o microsserviço. Para o microsserviço entender o que deve ser feita com aquelas informações recebidas, a API deverá enviar uma tag juntamente ao corpo das informações, que posteriormente será identificada. Entendendo esse fluxo, vamos voltar ao código da API e fazer essa pequena alteração da inserção da tag.

No arquivo *server.py, vamos* ao método de criação de usuário que abordamos nos Steps anteriores, e inserir ao payload que recebemos item chamado *type*, com o valor *create*.

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

def create\_user():

if request.method == 'POST':

payload = request.get\_json()

payload['type'] = 'create'

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'}

## Controlador DataBase

O controlador do database será o responsável por fazer o famoso **CRUD.** CRUD é a composição da primeira letra de 4 funções básicas de um sistema que trabalha com banco de dados:

✔️ C: Create (criar) - criar um novo registro.

👀 R: Read (ler) - ler (exibir) as informações de um registro.

♻️ U: Update (atualizar) - atualizar os dados do registro.

❌ D: Delete (apagar) - apagar um registro.

No desenvolvimento desse estágio mostrarei apenas uma interação com o banco de dados, visto que o desenvolvimento é muito complexo, e a leitura ficará muito maçante. O que faremos agora será a inclusão de um usuário ao banco. Para isso, devemos inserir as informações através de um comando SQL.

Primeiramente devemos importar todas as dependências necessárias, que serão a conexão com o banco, feita no *database\_connection.py,* e a biblioteca datetime.

from config.database\_connection import ConnectionDatabase

from datetime import datetime

Vamos criar uma Classe chamada **PostgresWorker**, com o construtor *\_\_init\_\_* contendo a conexão com o banco, captura do horário atual, e a formatação da data.

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

self.PSQL = ConnectionDatabase()

self.date\_time = datetime.now()

self.date\_time\_formate = self.date\_time.strftime('%Y/%m/%d %H:%M')

### Manipulação de datas

Algo a ser ressaltado! No código que iremos desenvolver, fiz todas as manipulações de datas usando o formato DATE, para uma explicação mais dinâmica e simples. Contudo, em projetos reais é **MUITO RECOMENDADO** a utilizar o padrão **Unix Timestamp.**

**Unix Timestamp** é um ponto fixo na história da computação, onde em 1º de janeiro de 1970 às 00:00 foi iniciado a contagem dos segundos. Essa data é considerada o início dos tempos pelo Linux. O motivo da recomendação do uso desse tipo de formato é pelo fato de o valor retornado do timestamp ser um float numeric. Sendo assim, é facilmente manipulado de forma binária e é muito curto e rápido de ser calculado.

A imagem abaixo mostra como uma determinada data e hora é representada em diferentes formatos.

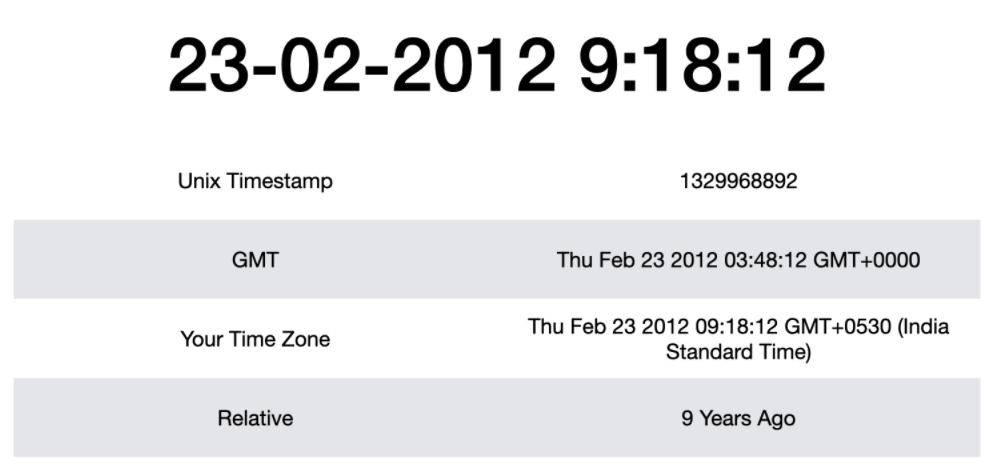


Imagem 1: <https://pynative.com/python-timestamp/>

Agora que entendemos a importância do uso do formato Unix Timestamp, podemos continuar com o desenvolvimento.

## Insert\_user

O que faremos agora será criar uma função chamada *insert\_user* que fará a inserção/criação de usuário no nosso banco de dados. O método receberá como parâmetro *data*, que serão os dados para fazermos a interação com o banco.

Dentro do método teremos a variável *query\_insert* que conterá a query para a execução do comando. Na variável *vars\_query* terão as informações recebidas da fila, que serão inseridas da query. A função *“cursor.execute(query\_update, vars\_query)”* é responsável pela execução do comando, onde pegará a *query\_update*, inserirá os valores de *vars\_query,*  e executará a instrução. A função *“connection.commit”*  é responsável por fazer as alterações do banco para a persistência do database. Por fim, é retornado alguma informação. Tudo isso ficará dentro de um try-except, para caso haja alguma falha na interação com o banco, seja lançado um except com o erro.

Como esse caso é apenas uma inserção no banco, apenas retornará uma mensagem ao usuário. Entretanto em casos de retorno de alguma informação do banco ao usuário, essas informações serão passadas no return.

def insert\_user(self, data):

try:

query\_insert = 'INSERT INTO users (full\_name, nick\_name ,password ,cpf , email, phone\_number, created\_at, updated\_at)VALUES (%s,%s,%s,%s,%s,%s,%s,%s)'

vars\_query = (data['name'], data['nick\_name'], data['password'], data['cpf'],

data['email'], data['phone\_number'], self.date\_time\_formate, self.date\_time\_formate)

self.PSQL.cursor.execute(query\_insert, vars\_query)

self.PSQL.connection.commit()

print('[✓] INSERTION DONE IN POSTGRES!')

return '[✓] User created successfully! '

except Exception as error:

print(error)

return f'[X] ERROR INSERTING IN POSTGRES! {error}'

finally:

self.PSQL.cursor.close()

## Show\_all\_user

Para exemplificar uma situação em que precisamos retornar uma informação do banco de dados para o usuário, vamos criar a função *show\_all\_user.* A estrutura da função será muito semelhante a que criamos acima. Teremos somente uma query de instrução que será um select de toda a tabela. Serão capturadas as informações através do comando “cursor.fetchall”, e manipuladas para mostrar mais organizado ao usuário. Tudo isso ficará dentro de um try-except, para caso haja alguma falha na interação com o banco, seja lançado um except com o erro.

def show\_all\_user(self):

try:

sql\_select\_query = 'SELECT \* FROM users'

self.PSQL.cursor.execute(sql\_select\_query)

record = self.PSQL.cursor.fetchall()

self.PSQL.connection.commit()

dict\_all\_users = []

for index in range(len(record)):

dict\_response = {

'nick\_name': record[index][1],

'name': record[index][3],

'email': record[index][5],

'phone\_number': record[index][6],

'created\_at': str(record[index][7]),

'updated\_at': str(record[index][8])

}

dict\_all\_users.append(dict\_response)

print('[✓] SELECT DONE SUCCESSFULLY IN POSTGRES!')

return {'users': dict\_all\_users}

except Exception as error:

print(error)

return f'[X] ERROR ON SELECT IN POSTGRES! {error}'

finally:

self.PSQL.cursor.close()

Com isso, podemos ter uma base de como fazer o nosso CRUD. As demais funções poderão ser visualizadas no repositório git do projeto.

## Métodos Criados

Os métodos precisaremos criar são:

**insert\_user** – Criação de usuário;

**alter\_user** – Alterar informações do usuário exceto senha;

**alter\_password** – Alterar senha do usuário;

**show\_all\_user** – Mostrar todos os usuários cadastrados;

**show\_one\_user** – Mostrar informações de um usuário através do NickName;

**delete\_user** – Deletar um usuário através do NickName;

**information\_user** – Selecionar todas as informações de um usuário;

**take\_pass** – Selecionar senha do usuário.

## Controlador Rabbit

O controlador do rabbit será o responsável por identificar a tag recebida e irá disparar uma função para a interação com o banco de dados. Contudo o retorno do bando de dados deve ser retornado também ao usuário que consumiu a API, para isso, iremos publicar esse response através da fila de controle que criamos no Step6.

No arquivo *rabbit\_worker.py* vamos iniciar importando as dependências.

import json, pika

from database\_controller.postgres\_worker import PostgresWorker

Vamos criar uma classe chamada **RabbitWorker,** e em seu construtor *\_\_init\_\_* criaremos um atributo vazio chamada *self.data.*

class RabbitWorker():

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

self.data = ''

Logo abaixo, criaremos o método que chamamos no arquivo *main.py,* chamado de *call-back.* Ele é o responsável por processar os dados recebidos, e retornar a fila de controle, um response ao usuário. Nos parâmetros iremos deverá ter *ch, method, props* e *body.* No atributo *self.data* iremos carregá-lo com as informações recebidas do body. Logo abaixo, criaremos uma variável onde chamaremos o metodo *database\_manipulation(self.data)* que criaremos posteriormente. Após isso, através do atributo *ch*, iremos criar a parte da publicação da mensagem de response na fila de controle e dando um ACK na fila de controle.

def callback(self, ch, method, props, body):

self.data = json.loads(body)

response\_work = self.database\_manipulation(self.data)

ch.basic\_publish(exchange='',

routing\_key=props.reply\_to,

properties=pika.BasicProperties(

correlation\_id=props.correlation\_id),

body=json.dumps(response\_work))

ch.basic\_ack(delivery\_tag=method.delivery\_tag)

Como dito anteriormente, agora vamos criar o método *database\_manipulation*, que é responsável por identificar a tag que há no corpo dos dados recebidos da API, e acionar os métodos contidos no arquivo *postgres\_worker.py* para a fazer a ação solicitada da requisição.

O que faremos é bem simples. Quando for recebido uma mensagem na fila de usuário, a mensagem será consumida. Será identificado o type da mensagem, que no caso é nossa tag, e após isso, chamaremos uma função equivalente ao type, para executar um CRUD no banco de dados.

Essa identificação será através de um *if-elif* da informação “type”, contida em data.

|  |  |  |
| --- | --- | --- |
| Type | Function | Objective |
| Create | insert\_user | Criar usuário |
| Update | alter\_user | Alterar informações de usuário |
| update\_password | alter\_password | Alterar senha de usuário |
| show\_all | show\_all\_user | Mostrar todos os usuários |
| show\_one | show\_one\_user | Mostrar um usuário |
| delete\_user | delete\_user | Deletar um usuário |

def database\_manipulation(self, data):

# start connection whith postgres

psql = PostgresWorker()

if data['type'] == 'create':

return psql.insert\_user(data)

elif data['type'] == 'update':

return psql.alter\_user(data)

elif data['type'] == 'update\_password':

return psql.alter\_password(data)

elif data['type'] == 'show\_all':

return psql.show\_all\_user()

elif data['type'] == 'show\_one':

return psql.show\_one\_user(data)

elif data['type'] == 'delete\_user':

return psql.delete\_user(data)

## Resumo

Nesse passo fizemos uma pequena alteração na API, inserindo uma nova informação no payload que será o ponto crucial para a identificação da ação que deve ser feita no banco de dados. Criamos o database\_worker onde há as funções de interação/manipulação do banco de dados, onde foi feito um CRUD. Foi explicado um pouco sobre o uso do UNIX Timestap para gravação de data-hora no banco de dados. Criamos o rabbit\_worker, que é o responsável por identificar o que está sendo solicitado através da tag, pegar as informações recebidas e utilizá-las nas funções contidas no database\_worker. Por fim, é publicado na fila auxiliar uma mensagem de response, onde o usuário receberá essa informação já processada.

# Step9

Nesse step iremos abordar um assunto muito importante, onde devemos sempre reservar um tempo para desenvolvimento, que é a criptografia de informações sensíveis. O que faremos será uma classe para criptografar a senha do usuário antes de salvar no banco de dados.

Na pasta raiz da aplicação MS-Application, vamos criar os seguintes arquivos:

MS-application

└───MS1

│ docker-compose-microservice1.yml

│ Dockerfile

│ main.py

│ requirements.txt

│

├───config

│ database\_connection.py

│ rabbitmq\_connection.py

│ \_\_init\_\_.py

│

**├───criptografy**

**│ hash\_password.py**

**│ \_\_init\_\_.py**

│

├───database\_controller

│ postgres\_worker.py

│ \_\_init\_\_.py

│

└───rabbitmq\_controller

rabbit\_worker.py

\_\_init\_\_.py

## Argon2

Para o hash de senhas, utilizaremos um algoritmo campeão da competição de Hashing de Senha em julho de 2015. Argon2 é um algoritmo de hash de senha seguro. Ele foi projetado para ter um tempo de execução configurável e consumo de memória. Isso significa que você pode decidir quanto tempo leva para fazer o hash de uma senha e quanta memória é necessária.

[argon2: <https://argon2-cffi.readthedocs.io/en/stable/argon2.html>]

## Hash de Senha

O tema de criptografia de senha pode ser muitas vezes assustador, contudo, nos dias atuais temos diferentes algoritmos e bibliotecas que podem nos auxiliar a aplicar uma complexidade maior no nível de segurança das informações. Por isso, faremos uma implementação simples usando a biblioteca **argon2** e iremos usar um conceito básico de O.O(Orientação a Objetos), que será o encapsulamento de dados.

### Encapsulamento

O encapsulamento consiste em evitar que esses dados sofram modificações ou acessos indevidos. Para isso, é criada uma estrutura que contém métodos chamados ***getters*** e ***setters***, que poderão ser utilizados em qualquer outra classe, sem causar inconsistência na criação do código.

***Getter*** – O método getter tem como objetivo retornar o valor que foi solicitado, porém, de forma que não prejudique a exatidão do dado.

***Setter*** – O método setter, recebe um valor como atributo de qualquer tipo suportado pela linguagem, podendo assim, acessar o dado bloqueado e fazer sua devida modificação.

Agora que temos conhecimento do que iremos utilizar, vamos para o desenvolvimento.

Primeiramente importaremos a biblioteca para fazer o hash das senhas

from argon2 import PasswordHasher

Vamos criar uma classe chamada **EncriptPassword** e em seu constructor \_\_init\_\_ iremos instanciar a biblioteca **PasswordHasher.** Tambémcriaremos os atributo *self.\_\_password* e *self.\_\_hash\_password* sendo privados, utilizando o “\_\_”*.* É nesses atributos que serão encapsulados

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

self.ph = PasswordHasher()

self.\_\_password = ''

self.\_\_hash\_password = ''

Agora criaremos os construtores getters e setters da senha e do hash da senha.

#password

def get\_pass(self):

return self.\_\_password

def set\_pass(self, password\_imput):

self.\_\_password = password\_imput

#hash password

def get\_hash\_pass(self):

return self.\_\_hash\_password

def set\_hash\_pass(self, hash\_pass):

self.\_\_hash\_password = hash\_pass

Nesse momento, criaremos o método que fará o hash da senha. Para isso usaremos a função *hash* e passaremos o *self.\_\_password* como atributo. Essa senha será hasheada e atribuída ao atributo privado *\_\_hash\_password.* Assim, classes externas não poderão alterar a senha já criptografada.

def hash\_password(self):

self.\_\_hash\_password = self.ph.hash(self.\_\_password)

Criaremos também um método que será responsável por comparar se a senha hasheada é equivalente à senha normal. Esse método será muito importante para troca de senha, edição das informações do usuário e deletar usuário, onde será verificado se a senha postada na requisição é compatível com a senha existente no banco de dados.

def verify\_hash(self):

try:

return self.ph.verify(self.\_\_hash\_password, self.\_\_password)

except Exception as error:

print('Erro in verify hash',error)

return False

## Controlador DataBase

Agora que temos nossa classe para criptografar as senhas, precisaremos criar alguns métodos a mais em nosso *postgres\_worker*.

Devemos lembrar de importar a classe **EncriptPassword**¸ ao arquivo*postgres\_worker.py.*

from config.database\_connection import ConnectionDatabase

from criptografy.hash\_password import EncriptPassword

from datetime import datetime

O primeiro método é o *verify\_password\_database(self, db\_password, old\_pass).* Ele será o responsável por receber como parâmetro a senha já criptografada, e a senha normal em formato string. Como resultado, nos retornará um valor booleano que será o que nos autorizará a continuar com algumas funções do sistema como: alterar a senha, alterar as informações do usuário e deletar um usuário. Caso a senha seja incompatível com a que há no banco de dados, a modificação será negada, e o usuário receberá um aviso como response.

def verify\_password\_database(self, db\_password, old\_pass):

try:

EP = EncriptPassword()

EP.set\_pass(old\_pass)

EP.set\_hash\_pass(db\_password[0])

response\_verify = EP.verify\_hash()

if response\_verify is True:

return True

return False

except Exception as error:

print(error)

Também será criado um método para criptografar a senha, onde novamente utilizaremos as funções da classe **EncriptPassword.** O retorno desse método deverá ser a senha já hasheada, visto que será ela que colocaremos no banco de dados.

def encript\_password(self, data):

try:

HS = EncriptPassword()

HS.set\_pass(data)

HS.hash\_password()

return HS.get\_hash\_pass()

except Exception as error:

print(error)

return f'[X] ERROR ON INCRIPTED PASSWORD! \

{error}'

Como exemplo de onde usaríamos esses novos métodos, deixarei abaixo um exemplo de uma criação de usuário, em que precisaremos colocar a senha. E a função de deletar um usuário, onde só será deletado caso a senha seja a mesma contida no banco de dados.

def insert\_user(self, data):

try:

encripted\_password = self.encript\_password(data['password'])

query\_insert = 'INSERT INTO users (full\_name, nick\_name ,password ,cpf , email, phone\_number, created\_at, updated\_at)VALUES (%s,%s,%s,%s,%s,%s,%s,%s)'

vars\_query = (data['name'], data['nick\_name'], encripted\_password, data['cpf'],

data['email'], data['phone\_number'], self.date\_time\_formate, self.date\_time\_formate)

self.PSQL.cursor.execute(query\_insert, vars\_query)

self.PSQL.connection.commit()

print('[✓] INSERTION DONE IN POSTGRES!')

return '[✓] User created successfully! '

except Exception as error:

print(error)

return f'[X] ERROR INSERTING IN POSTGRES! {error}'

finally:

self.PSQL.cursor.close()

def delete\_user(self, data):

try:

result\_psw = self.take\_pass(data['nick\_name'])

verify = self.verify\_password\_database(

result\_psw, data['password'])

if verify == False:

return f'[X] PASSWORD NOT EQUAL!'

sql\_delete\_query = 'DELETE FROM users WHERE nick\_name=%s'

vars\_query\_select = data['nick\_name']

self.PSQL.cursor.execute(sql\_delete\_query, (vars\_query\_select,))

row\_count = self.PSQL.cursor.rowcount

self.PSQL.connection.commit()

print('[✓] DELETE DONE SUCCESSFULLY IN POSTGRES!')

return {'Altered Lines': row\_count}

except Exception as error:

print(error)

return f'[X] ERROR ON DELETE IN POSTGRES! {error}'

finally:

self.PSQL.cursor.close()

OBS: lembrando novamente que todas as funções de interação com o banco de dados estão disponíveis no repositório git do projeto.

## Resumo

Nesse passo abordamos assuntos muito importantes como a criptografia da senha do usuário e um ramo da Programação Orientada a Objetos (POO), chamada de encapsulamento. Criamos os métodos de criptografia de senha usando o algoritmo argon2. Por fim, implementamos novas funções no *postgres\_worker.py,* que serão responsáveis por verificar a senha contida no banco, e também criptografar a senha para salvar no banco de dados.

Aqui, terminamos o desenvolvimento do primeiro Microservice (MS1). Na próxima etapa do projeto, iremos desenvolver o segundo Microservice (MS2), que será responsável por criar as orders no banco de dados.

# Step10

Com a finalização do MS1, vamos criar o segundo microservice que será o responsável por criar a manipular as orders relacionada a um usuário. A arquitetura de pastas será semelhante ao microservice 1, contendo *requirements*, *dockerfile*, *docker-compose*, pasta de *config* e mais algumas coisas.

Na pasta raiz da aplicação MS-Application, vamos criar os seguintes arquivos:

MS-application

└───MS2

**│ docker-compose-microservice2.yml**

**│ Dockerfile**

**│ main.py**

**│ requirements.txt**

**│**

**├───config**

**│ database\_connection.py**

**│ rabbitmq\_connection.py**

**│ \_\_init\_\_.py**

## Config

Como no primeiro microsserviço, os arquivos de conexão serão exatamente iguais, então apenas copiaremos e colararemos a pasta **config** para o MS2. Nela, deve conter *database\_connection.py* e *rabbitmq\_connection.py* para a conexão com os serviços.

## Requeriments

No arquivo *requeriments.txt* ficarão todas as instalações de bibliotecas e framework que serão utilizados na aplicação do microservices. As bibliotecas devem ficar separadas por linha, visto que usaremos um comando para uma instalação recursiva.

*requirements.txt*

psycopg2-binary

flask

pika

Comando para instalação:

> pip install -r requeriments.txt

## Docker-Compose

No arquivo *docker-compose-microservice2.yml,* iremos configurar a inicialização da imagem do serviço. Será feito o Build da imagem que criaremos posteriormente, apontado a Network que criamos juntamente aos serviços logo no início do projeto.

version: "3.7"

services:

microservice2:

build: .

networks:

- internal-network

networks:

internal-network:

external: true

## Dockerfile

No arquivo Dockerfile configuraremos a imagem do microservice, contendo as variáveis de ambiente adequadas para o serviço. Diferente da imagem do python que usamos na API, no microservice precisaremos instalar a imagem completa do python, sendo ela: python:3.8

FROM python:3.8

COPY . /app/

ENV HOST='10.0.0.25'

ENV HOST\_DATABASE='10.5.0.11'

WORKDIR /app

RUN pip install --upgrade pip

RUN pip install -r requirements.txt

CMD python main.py

## Main.py

O arquivo *main.py* será responsável por interceptar as mensagens enviadas para a fila de orders.

Primeiramente, devemos importar os arquivos de conexão com o postgres e rabbit. Após isso, criaremos a classe *Main(),*  onde no método *\_\_init\_\_,* chamaremos as conexões com os servidores.

from config.database\_connection import ConnectionDatabase

from config.rabbitmq\_connection import ConnectionRabbitMq

from rabbitmq\_controller.rabbit\_worker import RabbitWorker

class Main():

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

self.PSQL = ConnectionDatabase()

self.RMQ = ConnectionRabbitMq()

self.RMQ\_WORKER = RabbitWorker()

Feito isso, criaremos um método chamado *consume\_queue,* que consumirá a fila que desejamos. Nesse caso será *order,* e posterior o consumo, irá disparar uma função para o processamento do dado recebido da fila. Nesse momento não iremos criar a função de processamento, porém iremos apenas nomeá-la no lugar correto.

Com o comando *basic\_consume* consumiremos a fila *order*, e no on\_message\_callback será disparado a função para o processamento da informação recebida na fila.

Com o comando *start\_consuming,* é processado o evento de I/O até que todos as mensagens sejam processadas.

def consume\_queue(self):

self.RMQ.channel.basic\_qos(prefetch\_count=1)

self.RMQ.channel.basic\_consume(queue='order', on\_message\_callback=self.RMQ\_WORKER.callback)

print(' [⇄] Waiting for messages. To exit press CTRL+C')

self.RMQ.channel.start\_consuming()

Finalizaremos com a chamada do método *consume\_queue* para inicializar o serviço.

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

MA = Main()

MA.consume\_queue()

## Resumo

Nesse Step, criamos o microservice 2 para o processamento dos dados recebidos através da fila de order. Copiamos os arquivos de conexão *database\_connection.py* e *rabbitmq\_connection.py* do Microservice 1, visto que é a mesma conexão que precisaremos fazer. Também criamos o arquivo *main.py* que será o responsável por inicializar o serviço e disparar o método *consume\_queue,* que como o próprio nome diz, consumirá a fila de usuários.

# Step11

No passo anterior concluímos a arquitetura do Microsserviço 2, e fizemos a conexão com os serviços do Rabbit e Postgres. Agora, vamos fazer a lógica do sistema dos orders, onde diferenciaremos as interações com o banco de dados através de uma tag, enviada pela API, e dependendo da interação de for solicitada, haverá o acionamento de uma função equivalente a solicitação.

Na pasta raiz da aplicação MS-Application, vamos criar os seguintes arquivos:

MS-application

└───MS1

│ docker-compose-microservice1.yml

│ Dockerfile

│ main.py

│ requirements.txt

│

├───config

│ database\_connection.py

│ rabbitmq\_connection.py

│ \_\_init\_\_.py

│

**├───database\_controller**

**│ postgres\_worker.py**

**│ \_\_init\_\_.py**

**│**

**└───rabbitmq\_controller**

**rabbit\_worker.py**

**\_\_init\_\_.py**

## Tag API

Antes de começarmos a codar, vamos entender o cenário onde paramos. Temos uma API que recebe as informações de uma criação de um order, por exemplo, e envia essas informações para o microsserviço. Para o microsserviço entender o que deve ser feito com aquelas informações recebidas, a API deverá envia uma tag juntamente ao corpo das informações, que posteriormente será identificada. Entendendo esse fluxo, vamos voltar ao código da API e fazer essa pequena alteração da inserção da tag.

No arquivo *server.py, vamos* ao método de criação de usuário que abordamos nos Steps anteriores, e inserirmos ao payload que recebemos um item chamado *type*, com o valor *create\_order*.

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

def create\_order():

if request.method == 'POST':

payload = request.get\_json()

payload['type'] = 'create\_order'

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

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'}

## Controlador DataBase

O controlador do database será o responsável por fazer o famoso **CRUD.** CRUD é a composição da primeira letra de 4 funções básicas de um sistema que trabalha com banco de dados:

✔️ C: Create (criar) - criar um novo registro.

👀 R: Read (ler) - ler (exibir) as informações de um registro.

♻️ U: Update (atualizar) - atualizar os dados do registro.

❌ D: Delete (apagar) - apagar um registro.

No desenvolvimento desse estágio mostrarei apenas uma interação com o banco de dados, visto que o desenvolvimento é muito complexo, e a leitura ficará muito maçante. O que faremos agora será a inclusão de um order ao banco. Para isso devemos inserir as informações através de um comando SQL.

Primeiramente, devemos importar todas as dependências necessárias, que serão a conexão com o banco, feita no *database\_connection.py,* e a biblioteca datetime.

from config.database\_connection import ConnectionDatabase

from datetime import datetime

Vamos criar uma Classe chamada **PostgresWorker**, com o construtor *\_\_init\_\_* contendo a conexão com o banco, captura do horário atual, e a formatação da data.

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

self.PSQL = ConnectionDatabase()

self.date\_time = datetime.now()

self.date\_time\_formate = self.date\_time.strftime('%Y/%m/%d %H:%M')

### Manipulação de datas

Para entender um pouco melhor sobre a manipulação de datas, confira o tópico “Manipula de datas” do Step 8.

## Create\_order

O que faremos agora será criar uma função chamada *create\_order* que fará a inserção/criação de um order no nosso banco de dados. O método receberá como parâmetro *data*, que serão os dados para fazermos a interação com o banco.

Devemos verificar antes se existe cadastrado o usuário no banco de dados. Para isso passaremos o nick\_name da pessoa a uma função chamada *information\_user.* Dependendo do retorno, o usuário receberá uma mensagem informando que não existe esse usuário cadastrado no manco de dados.

Dentro do método também teremos a variável *query\_insert* que conterá a query para a execução do comando. Na variável *vars\_query* terão as informações recebidas da fila, que serão inseridas da query. A função *“cursor.execute(query\_update, vars\_query)”* é responsável pela execução do comando, onde pegará a *query\_update*, inserir os valores de *vars\_query,*  e executar a instrução. A função *“connection.commit”*  é responsável por fazer as alterações do banco para a persistência do database. Por fim, é retornado alguma informação. Tudo isso ficará dentro de um try-except, para caso haja alguma falha na interação com o banco, seja lançado um except com o erro.

Como esse caso é apenas uma inserção no banco, apenas retornará uma mensagem ao usuário. Contudo, em casos de retorno de alguma informação do banco ao usuário, essas informações serão passadas no return.

# create a order in database

def create\_order(self, data):

try:

user\_id = self.information\_user(data['nick\_name'])

if user\_id == None:

return f"[X] DON'T HAVE THIS USER IN DATA-BASE!"

total\_value = data['item\_quantity'] \* data['item\_price']

query\_insert = 'INSERT INTO orders (user\_id ,item\_description ,item\_quantity , item\_price, total\_value, created\_at, updated\_at)VALUES (%s,%s,%s,%s,%s,%s,%s)'

vars\_query = (user\_id[0], data['item\_description'], data['item\_quantity'],

data['item\_price'], total\_value, self.date\_time\_formate, self.date\_time\_formate)

self.PSQL.cursor.execute(query\_insert, vars\_query)

self.PSQL.connection.commit()

print('[✓] INSERTION DONE IN POSTGRES!')

return '[✓] Order created successfully! '

except Exception as error:

print(error)

return f'[X] ERROR INSERTING IN POSTGRES! {error}'

finally:

self.PSQL.cursor.close()

## List\_all\_orders

Para exemplificar uma situação em que precisamos retornar uma informação do banco de dados para o usuário, vamos criar a função *list\_all\_orders.* A estrutura da função será semelhante a que criamos acima. Teremos somente uma query de instrução, que será um select dos *order\_id* e *item\_description* de toda a tabela. Serão capturadas as informações através do comando “cursor.fetchall”, e manipuladas aos dados recebidos do banco, para mostrar mais organizado ao usuário. Tudo isso ficará dentro de um try-except, para caso haja alguma falha na interação com o banco, seja lançado um except com o erro.

# Show all orders of batabase

def list\_all\_orders(self):

try:

sql\_select\_query = 'SELECT order\_id, item\_description FROM orders'

self.PSQL.cursor.execute(sql\_select\_query)

record = self.PSQL.cursor.fetchall()

self.PSQL.connection.commit()

dict\_all\_orders = []

for index in range(len(record)):

dict\_respose={

'order\_id': record[index][0],

'item\_description': record[index][1],

}

dict\_all\_orders.append(dict\_respose)

print('[✓] SELECT ALL USER DONE SUCCESSFULLY IN POSTGRES!')

return dict\_all\_orders

except Exception as error:

print(error)

return f'[X] ERROR ON SELECT IN POSTGRES! {error}'

finally:

self.PSQL.cursor.close()

Com isso podemos ter uma base de como fazer o nosso CRUD. As demais funções poderão ser visualizadas no repositório git do projeto.

## Métodos Criado

Os métodos precisaremos criar são:

**create\_order** ­– Cria um order;

**edit\_order** – Edita as informações de um order;

**list\_all\_orders** – Lista todos os orders cadastrados;

**list\_per\_users** – Lista os orders por usuário;

**show\_order** – Mostra todas as informações de um order;

**delete\_order** – Deleta um order;

**information\_user** – Captura o user\_id de um usuário;

**information\_order** – Seleciona todas as informações de um order.

## Controlador Rabbit

O controlador do rabbit será o responsável por identificar a tag recebida e disparará uma função para a interação com o banco de dados. Porém o retorno do bando de dados deve ser retornado também ao usuário que consumiu a API. Para isso, publicaremos esse response através da fila de controle que criamos no Step6.

No arquivo *rabbit\_worker.py* vamos iniciar importando as dependências.

import json, pika

from database\_controller.postgres\_worker import PostgresWorker

Vamos criar uma classe chamada **RabbitWorker,** e em seu construtor *\_\_init\_\_* criaremos um atributo vazio chamada *self.data.*

class RabbitWorker():

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

self.data = ''

Logo abaixo, criaremos o método que chamamos no arquivo *main.py,* chamado de *call-back.* Ele é o responsável por processar os dados recebidos, e retornar à fila de controle, um response ao usuário. Nos parâmetros deverão conter *ch, method, props* e *body.* No atributo *self.data* iremos carregá-lo com as informações recebidas do body. Logo abaixo, criaremos uma variável onde chamaremos o método *database\_manipulation(self.data)* que criaremos posteriormente.

Após isso, através do atributo *ch*, iremos criar a parte da publicação da mensagem de response na fila de controle e dando um ACK na fila de controle.

def callback(self, ch, method, props, body):

self.data = json.loads(body)

response\_work = self.database\_manipulation(self.data)

ch.basic\_publish(exchange='',

routing\_key=props.reply\_to,

properties=pika.BasicProperties(

correlation\_id=props.correlation\_id),

body=json.dumps(response\_work))

ch.basic\_ack(delivery\_tag=method.delivery\_tag)

Como dito anteriormente, agora vamos criar o método *database\_manipulation*, que é responsável por identificar a tag que há no corpo dos dados recebidos da API, e acionar os métodos contidos no arquivo *postgres\_worker.py* para a fazer a ação solicitada da requisição.

O que faremos é bem simples. Quando for recebido uma mensagem na fila de usuário, a mensagem será consumida. Será identificado o type da mensagem, que no caso é nossa tag, e após isso, chamaremos uma função equivalente ao type para executar um CRUD no banco de dados.

Essa identificação será através de um *if-elif* da informação “type”, contida em data.

|  |  |  |
| --- | --- | --- |
| Type | Function | Objective |
| create | insert\_user | Criar usuário |
| update | alter\_user | Alterar informações de usuário |
| update\_password | alter\_password | Alterar senha de usuário |
| show\_all | show\_all\_user | Mostrar todos os usuários |
| show\_one | show\_one\_user | Mostrar um usuário |
| delete\_user | delete\_user | Deletar um usuário |

def database\_manipulation(self, data):

# start connection whit postgres

psql = PostgresWorker()

if data['type'] == 'create\_order':

print('entrou no type')

return psql.create\_order(data)

elif data['type'] == 'update\_order':

return psql.edit\_order(data)

elif data['type'] == 'list\_all\_orders':

return psql.list\_all\_orders()

elif data['type'] == 'list\_per\_users':

return psql.list\_per\_users(data)

elif data['type'] == 'show\_order':

return psql.show\_order(data)

elif data['type'] == 'delete\_order':

return psql.delete\_order(data)

## Resumo

Nesse passo fizemos uma pequena alteração na API, inserindo uma nova informação no payload que será o ponto crucial para a identificação da ação que deve ser feita no banco de dados. Criamos o database\_worker onde há as funções de interação/manipulação do banco de dados, onde foi feito um CRUD. Criamos o rabbit\_worker, que é o responsável por identificar o que está sendo solicitado através da tag, pegar as informações recebidas e utilizá-las nas funções contidas no database\_worker. Por fim, é publicado na fila auxiliar uma mensagem de response, onde o usuário receberá essa informação já processada.

# Step12

Findando o nosso projeto, precisaremos ainda aplicar a camada de cache em todas as rodas que nos retornam alguma informação vinda do banco de dados. Como estamos usando o Flask em nossa aplicação, existe uma extensão de cache próprio da biblioteca chamada Flask-Caching. Flask-Caching é uma extensão do Flask que adiciona suporte de cache para vários back-ends a qualquer aplicativo Flask. Ao ser executado em cima do [cachelib](https://github.com/pallets-eco/cachelib) , ele suporta todos os back-ends de cache originais do werkzeug por meio de uma API uniforme. Também é possível desenvolver seu próprio backend de armazenamento em cache por meio de subclasses de flask\_caching.backends.base.BaseCache classe, mas isso não vem ao caso nessa aplicação.

[Flask-Caching - <https://flask-caching.readthedocs.io/en/latest/>]

## Instalação

Antes de qualquer coisa, certifique-se de ter instalado todos os módulos contidos no arquivo *requirements.txt* da API.

*requirements.txt*

psycopg2-binary

Flask-Caching

threaded

flask

redis

pika

uuid

Comando para instalação:

> pip install -r requeriments.txt

Primeiramente faremos o import da biblioteca *Cache* que iremos utilizar no arquivo *server.py*.

from flask import Flask, request

from flask\_caching import Cache # Import Cache from flask\_caching module

from config.database\_connection import ConnectionDatabase  
...

## Inicialização do Redis

Dentro da classe *Api\_server*, apontaremos a conexão com o servidor Redis, visto que já configuramos todas as informações no arquivo *redis\_connection.py*  no Step5, precisaremos chama-las no arquivo da API.

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

app.config.from\_object('config.redis\_connection.BaseConfig')

cache = Cache(app) # Initialize Cache

ConnectionDatabase()

rabbit\_queues.create\_queues()

## Decoratos Cached

Nas rotas que serão retornadas alguma informação da banco de dados iremos colocar um novo decorator. Para armazenar funções em cache, usaremos o comando **cached**. Por parâmetro, haverá a opção *timeout*, que será o responsável por manter o cache por determinado tempo, que no nosso caso, será 30 segundos.

@app.route("/user/show\_all\_user/", methods=['GET'])

@cache.cached(timeout=30, query\_string=True)

def list\_user():

...

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

@cache.cached(timeout=30, query\_string=True)

def show\_user():

...

@app.route("/order/list\_all\_orders/", methods=['GET'])

@cache.cached(timeout=30, query\_string=True)

def list\_order():

...

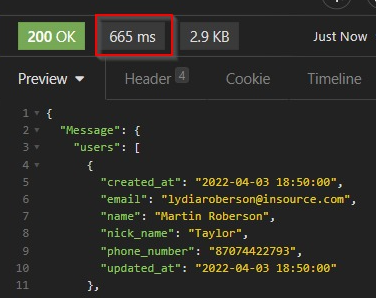
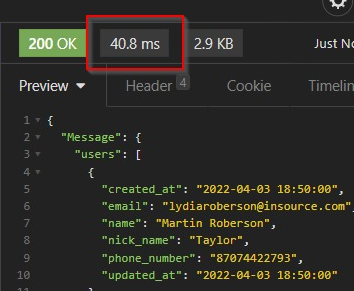
@app.route("/order/list\_per\_users/", methods=['GET'])

@cache.cached(timeout=30, query\_string=True)

def list\_per\_users():

## Exemplo

Fazendo um teste, o ganho é significativo, chegando até ser 20x mais rápido do que uma consulta padrão do postgres.

# Finalização

Então, chegamos ao fim desse projeto.

Durante o desenvolvimento do código, me esbarrei em diversos erros e problemas que pareciam não ter mais sentido, principalmente nas partes onde precisava fazer algum teste no ambiente da OCI. Como ainda estava aprendendo algumas coisas novas sobre a criação de aplicação em nuvem, passei por problemas essenciais para aprender e entender como devo arquitetar todo um local de produção. Também aprendi diversas coisas sobre persistência de dados, e como trabalhar com as manipulações das informações disponibilizadas pelo usuário. Sobre Docker, aprendi muito como criar Docker-Compose, visto que é extremamente importante saber isso para criação de containers rapidamente, e pude ver um pouco mais sobre Dockerfile, e buildar minhas próprias imagens, de acordo com a necessidade da aplicação. Pude me aprofundar na criação e gerenciamento do broker de mensagem utilizando Rabbit, e em como criar uma estrutura complexa utilizando o conceito RPC(Remote Procedure Call), para o retorno de informações ao requisitor, que no nosso caso é o usuário. Com todos esses aprendizados pude concluir o objetivo final que era criar microsserviços independentes e caso algum deles pare de funcionar, não cause indisponibilidade a todo a aplicação.

Uma coisa que deixei de fazer nesse projeto, mas que pretendo abordar em outro momento, é a criação de testes unitários, visto que isso é um requisito quase obrigatório no mercado de trabalho dos desenvolvedores. Então, podemos riscar de nosso fluxo de desenvolvimento essa etapa.

Logotipo

Descrição gerada automaticamente com confiança média

Espero que tenha aproveitado todo o desenvolvimento do projeto, e que sua leitura tenha agregado conhecimentos.