universidade de aveiro

Departamento de Eletrónica, Telecomunicações e Informática

Gestão Integrada de Redes e Sistemas

Projeto Funkwhale

2020/2021

Alunos:

Rodrigo Santos, 89180

Daniel Lopes, 87881

Introdução

Neste relatório encontram-se alguns aspetos relativos ao projeto de implementação da aplicação Funkwhale num cluster de Kubernetes. Este projeto é do âmbito da cadeira de Gestão Integrada de Redes e Sistemas, do curso MIECT da Universidade de Aveiro.
Será abordado neste relatório o produto a fornecer bem como um cenário devidamente caracterizado, o software utilizado e o seu modo de funcionamento, decomposição do produto sobre a infraestrutura de Kubernetes, uma descrição da instalação tendo em consideração aspetos como a persistência do armazenamento, rede e mecanismos de balanceamento de carga, uma análise dos recursos necessários para suportar os pods alocados e também uma descrição dos fluxos de ingress de tráfego e visibilidade dos endpoints.

Produto e Cenário

O produto a implementar e gerir é o Funkwhale. O Funkwhale é um servidor de áudio baseado na web, semelhante em termos de objetivos e recursos definidos a vários projetos existentes como o Sonerezh e Airsonic.
O Funkwhale é uma aplicação mais adaptada a pequenas e médias comunidades e foi desenvolvido para não ser só um servidor de reprodução de música mas também um sítio para socializar em torno da música.

O cenário de utilização a executar será baseado no upload e reprodução de uma ou mais músicas por parte de um ou mais utilizadores. Tendo como objetivo fornecer o serviço de forma rápida e sem que o utilizador sinta a carga imposta no serviço em qualquer instante da sua utilização.

Software

Quanto ao software utilizado e o seu modo de funcionamento distribuído pela aplicação Funkwhale, a sua arquitetura é composta por vários módulos diferentes interligados de modo a fornecer o serviço de uma forma coerente e eficaz. Na figura a seguir é possível analisar esta arquitetura.

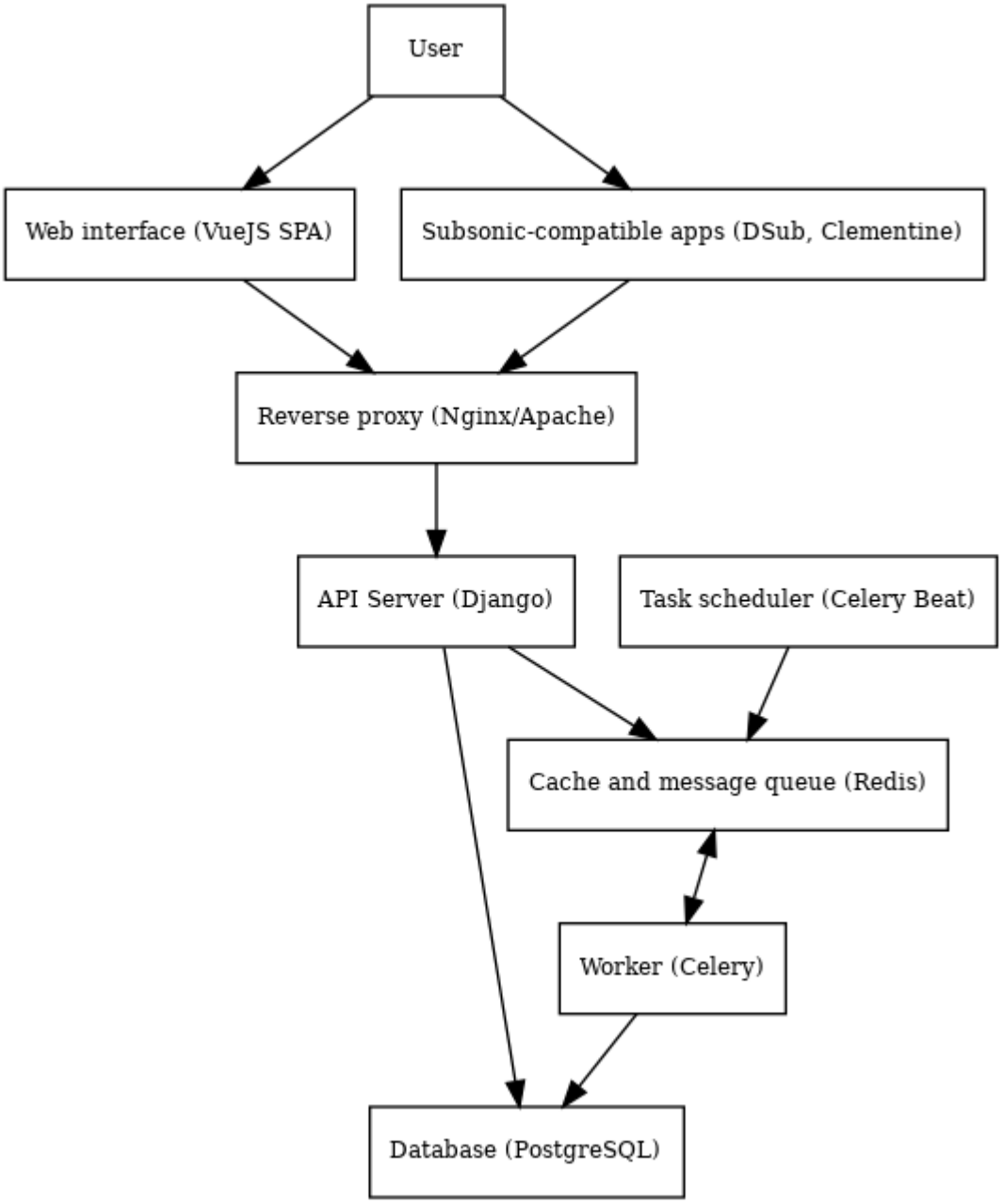

Figura 1: Arquitetura da Aplicação

Componentes:

- Web interface (VueJS SPA) - aplicação Single Page escrita em Vue JS, esta aplicação interage com a API do Funkwhale para recolher ou escrever dados.

- Subsonic-compatible apps - aplicações que podem ser usadas em conjunto ou em lugar da web interface, sendo os dados subjacentes os mesmos. Não será utilizada nenhuma aplicação deste género para estes propósitos.

- Reverse proxy (Nginx or Apache) - O reverse proxy do serviço recebe os requests HTTP ou HTTPS e faz o redirecionamento destes para a API do servidor, também serve os ficheiros estáticos pedidos tais como ficheiros de áudio, javascript, stylesheets, etc..

- API server (Django) - Peça central do projeto, este componente é responsável por responder e processar os pedidos dos utilizadores, manipular dados da base de dados, e enviar tarefas de longa duração para os workers.

- Database (PostgreSQL) - Maior parte dos dados como contas de utilizador, favoritos, metadados das músicas e playlists são guardados numa base de dados PostgreSQL.

- Cache and message queue (Redis) - Procurar e receber dados da base de dados pode por vezes ser um processo lento e intensivo em recursos utilizados, para reduzir a carga, o Redis age como uma cache para dados o que torna este acesso consideravelmente mais rápido. Também funciona como uma lista de mensagens que entrega tarefas aos workers.

- Worker (Celery) - Algumas operações são demasiado longas para viver num ciclo de request/response HTTP, como por exemplo importar um conjunto de músicas pode demorar 1 ou 2 minutos, logo para manter os tempos de resposta o mais curto possível estas operações são colocadas em processos Celery no background.

- Task Scheduler (Celery Beat) - Responsável por dar início e colocar na lista de mensagens dos workers, as tarefas de cache cleaning e atualização de metadados de músicas.

Decomposição do produto

No processo de deploy do produto para o cluster de Kubernetes, foi necessário decompor o produto a implementar, para um melhor funcionamento e uma distribuição mais eficiente e coerente dos diversos componentes utilizados pelo produto.
Após analisar a arquitetura do sistema, foi decidido que a melhor abordagem para a decomposição do produto seria criar um namespace para a aplicação, de modo a não interferir com nenhum outro projeto ou componente existente no cluster, e dentro desse namespace fazer a seguinte decomposição:

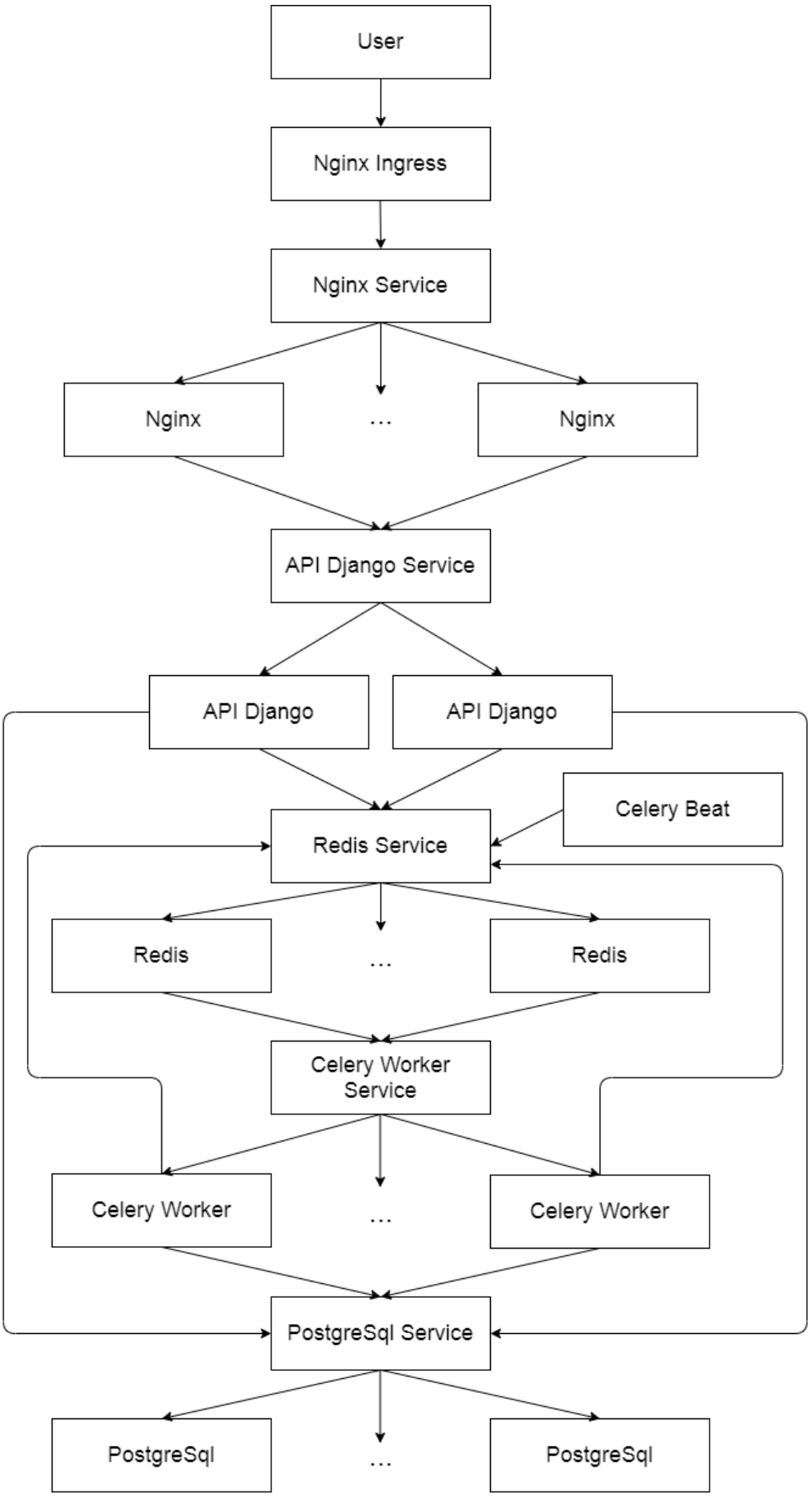

Figura 2: Decomposição do Produto no cluster de Kubernetes

- 1 *deployment* e 1 serviço para comunicação entre componentes, para a base de dados **PostgreSQL**;

- 1 *deployment* e 1 serviço para interligação de componentes para **Redis**;

- 1 *deployment* e 1 serviço para a **API Django**;

- 1 *deployment* e 1 serviço para o ***Celery worker**;*

- 1 *deployment* e 1 serviço para o **Celery beat**;

- 1 *deployment,* 1 serviço e 1 *ingress* para o Reverse Proxy **Nginx**.

Deste modo temos um maior controlo sobre todos os componentes de forma individual, facilitando assim a gestão do sistema. De notar que foi decidido manter a API Django e os componentes *Celery worker* e *Celery beat* num único deployment visto que são componentes que comunicam e se interligam diretamente entre si, e são todos componentes desenvolvidos em Python, não havendo assim necessidade de criar imagens com a mesma imagem base para componentes diferentes.

Descrição da instalação

No processo de instalação do produto no cluster de Kubernetes, tal como foi mencionado na secção anterior, o primeiro passo foi criar um namespace na infraestrutura para não interferir com nenhum outro serviço que se encontrasse na infraestrutura.
Após a criação do namespace "funkwhale" deu-se início à instalação dos diversos componentes constituintes da arquitetura do produto.

Para garantir a coesão nas dependências necessárias por cada componente, foi primeiro instalado e alocado um pod para um servidor Redis, foi construída a sua imagem Docker com base numa imagem oficial de Redis, efetuado o push e por fim feito e aplicado o deployment e serviço de Redis na infraestrutura. Verificou-se que tanto o serviço como o Deployment se encontravam operacionais e seguiu-se a instalação do próximo componente que não necessita de nenhum outro componente para a sua execução inicial, que é a base de dados PostgreSql. O procedimento foi em todo igual com a instalação do servidor Redis, foi construída uma imagem de Docker, efetuado o push e aplicado o deployment. De seguida, foram instalados os componentes Django, Celery Worker e Celery Beat, para os quais foi usada a imagem de Docker direta da documentação do Funkwhale, efetuamos o build e o push e aplicado o respectivo deployment. Por fim, foi feita a instalação do servidor web reverso Nginx, o processo seguido foi primeiramente foi feita a construção da imagem de Docker tendo por base uma imagem oficial de Nginx, de seguida efetuou-se o push para o cluster e por fim feito e aplicado o respectivo deployment, o serviço e o ingress. Com o Nginx a correr no cluster, foram instalados e configurados diretamente no pod alocado para o Nginx os ficheiros de configuração necessários à correta execução do servidor proxy. Verificamos por fim que todos os componentes se encontravam a correr corretamente.

Análise de Recursos

Uma vez que temos todos os componentes da aplicação separados em pods e containers diferentes, a utilização de recursos por cada um não será elevada. Os seguintes valores de alocação de memória e CPU foram estimados de acordo com o que será expectável da necessidade para a correta execução de cada um:

- **Redis**: CPU limit - 500m (0.5 cores)
  Memory limit - 128 MiB

  CPU request - 10m (0.01 cores)
  Memory request - 32 MiB

- **PostgreSql**: CPU limit - 1000m (1 cores)
  Memory limit - 256 MiB

  CPU request - 20m (0.02 cores)
  Memory request - 64 MiB

- **API Django**: CPU limit - 500m (0.5 cores)
  Memory limit - 128 MiB

  CPU request - 10m (0.01 cores)
  Memory request - 32 MiB

- **Celery Beat**: CPU limit - 500m (0.5 cores)
  Memory limit - 128 MiB

  CPU request - 10m (0.01 cores)
  Memory request - 32 MiB

- **Celery Worker**: CPU limit - 500m (0.5 cores)
  Memory limit - 128 MiB

  CPU request - 10m (0.01 cores)
  Memory request - 32 MiB

- **Nginx**: CPU limit - 750m (0.75 cores)
  Memory limit - 256 MiB

  CPU request - 15m (0.015 cores)
  Memory request - 64 MiB

No total, tendo em conta todos os valores necessários no limite por cada componente, temos um total de 3,75 cores do CPU e 1024 MiB de memória.

Descrição dos fluxos Ingress e Egress

Para a implementação do produto no Cluster foi configurado o Traefik no deployment do Nginx ao qual foi configurado o ingress que está conectado ao exterior pela porta 80. Sendo que deste modo, este componente é o responsável pela disponibilização dos conteúdos da aplicação aos clientes. Na implementação do serviço não foi configurado mais nenhum fluxo de Ingress ou Egress, uma vez que mais nenhum outro componente necessita nem faria um uso adequado de acesso ao exterior.