Biblify

Universidade de Aveiro

Departamento de Eletrónica, Telecomunicações e Informática

Engenharia e Gestão de Serviços

Relatório

Alexey Kononov 89227 & António Domingues 89007 & Miguel Cabral 93091 & Rodrigo Santos 93173

Professores João Barraca e Diogo Gomes

Junho de 2022

Índice

Introdução

No âmbito da unidade curricular, o grupo teve iniciativa de abordar o tema de uma biblioteca. Muito resumidamente, a plataforma é dotada de capacidades de ver quantos livros existem em stock, quantos estão reservados, assim como os autores associados a cada livro. Numa fase mais avançada do projeto, caso tudo funcione como esperado, é suposto um determinado utilizador conseguir reservar um livro. Ao efetuar esta ação, o stock tem de ser atualizado. Contudo, o utilizador possui um perfil, onde pode visualizar a sua informação, assim como os livros que foram reservados e ou publicados por ele.

Como o tema não possuía serviços para todos os elementos do grupo, adicionou-se uma *feature* ao projeto: um serviço de notificação. Este serviço tem como objetivo gerir os utilizadores que têm subscrições de notificações, assim como enviar as devidas notificações aos utilizadores.

Assim, o tema foi dividido em 4 serviços, como vai ser explicado ao longo do presente documento.

O repositório do projeto, pode ser encontrado no seguinte endereço

https://github.com/AlexkononovV/egs

Arquitetura

Como exposto nas aulas, a nossa aplicação segue o princípio de uma arquitetura ***SOA***. Assim, visto que se possui três serviços principais (**stock, autenticação e notificação**) surgiu desde logo a necessidade de um quarto serviço, o orquestrador. Sem este, a aplicação não se iria interligar de maneira nenhuma, porque cada serviço é independente. Este estabelece as ligações às diversas *api*'s expondo o necessário para um front-end, para que um user possa interagir com os diversos serviços disponibilizados, sem se que seja necessário saber o como é que os serviços comunicam e estão implementados entre si.

Assim, chegou-se à seguinte arquitetura em termos de serviços.

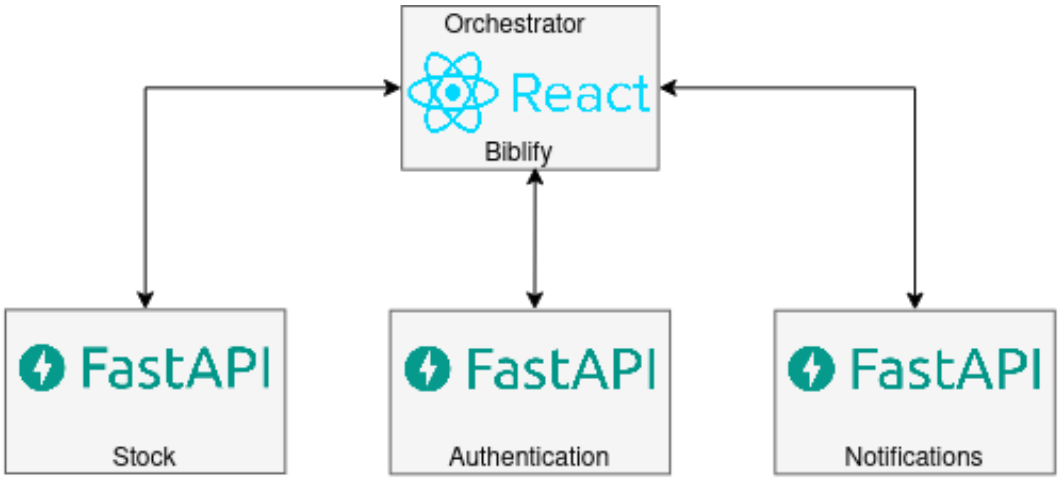

Fig. 1 - Arquitetura dos Serviços

Neste sentido, cada *api*, é dotada de uma base de dados, onde armazena todas as informações relevantes para o seu funcionamento.

Desenvolvimento

De seguida, é descrito, a maneira de como cada elemento implementou o seu serviço, quais as tecnologias utilizadas, a arquitetura do respetivo serviço e como foi efetuado o *deployment,* assim como, a documentação da *api*.

Autenticação

O principal objetivo deste serviço é o de permitir que os utilizadores se possam registar e autenticar na plataforma.

Na primeira etapa de modelação do serviço fez-se a construção da api. Para isso, foi utilizado o Swagger 2.0 de forma a criar a documentação. Temos assim os **schemas** que facilitam a distinção entre os vários modelos por forma a reduzir a redundância entre eles:

- User - define um utilizador geral:
    - username;
    - email;
- UserCreate - extende o User com o campo adicional da password;
- UserInDB - extende o User com o campo adicional da hash da password;
- Token - que garante a criação de um token:
    - access_token;
    - token_type;
- TokenData - atribuição de um token a um utilizador:
    - username;

Passada a implementação para FastAPI pelo bom desempenho, suporte de código assíncrono, fácil e rápido desenvolvimento, é necessário estabelecer a ligação entre a api e a base de dados. Para isso, é usado o SQLAlchemy que passando o url da base de dados é possível executar diferentes operações através da criação de queries. A base de dados é implementada em MySQL. O ficheiro que implementa estas operações segue uma estrutura CRUD (Create, Read, Update e Delete). Os endpoints são então criados na main que fazem uso dos métodos auxiliares implementados nestas classes.

A estrutura acima definida encontra-se no diretório *src/* juntamente com um script que inicia a api apenas quando a base de dados estiver ligada e ainda um ficheiro *.sql* para criar a base de dados necessária e a tabela no container.

Posto isto, testou-se localmente no docker-compose com os seguintes serviços:

- Services:
    - db: base de dados
        - usa base image mysql:8.0.21-alpine
        - usa 2 volumes: um para persistência de dados e outro para scripts de inicialização
        - expose na porta 3306
        - parâmetros nas variáveis de ambiente (user,password,nome da base de dados, password do root)
    - api: API
        - imagem criada com Dockerfile.api
        - expose porta 8000
        - depende do serviço de base de dados
        - variável de ambiente com c*onnection string* para a ligação à base de dados

O próximo passo é a instanciação no cluster Kubernetes . São criados pods e serviços para a API de autenticação e a respetiva base de dados além um *ingress* para a api para que possa ser acedido pelo serviço de *front-end* publicamente.

Os ficheiros de Docker contendo as imagens dos pods encontram-se no diretório deploy/ bem como os ficheiros de deployment. É neles que se especificam as imagens a serem usadas, os serviços e as portas de acesso onde o container vai ser exposto e também o volume de dados a ser usado através de um volume persistente.

Após a criação de todos estes ficheiros, é necessário dar build e push a ambas as imagens (API e base de dados) para depois se poder fazer *apply* no cluster.

Orquestrador

Neste serviço, o principal objetivo é fazer os diversos pedidos às restantes *api's* e fazer a articulação com todos estes serviços. Deste modo, não fora implementada nenhuma base de dados, porque não surge essa necessidade. Com isto, foi desenvolvido em **react**, um frontend para expor todos os casos de uso da aplicação acima mencionados. Para tal, utiliza-se um template, onde se procede à alteração de algumas páginas e se fazem os vários pedidos à api, através dos métodos ***post, get*** e ***delete***. Não se utilizaram nenhum dos restantes verbos dos métodos *http* porque não surge a sua necessidade.

Relativamente à primeira parte da unidade curricular, apenas se estabelecem os contactos com os vários endpoints necessários.

Numa iteração seguinte, já na temática da gestão de serviços, num primeiro incremento, trata-se de ***dockerizar*** este serviço, para de seguida utilizar ***kubernetes.*** Para tal acontecer, utiliza-se uma imagem ***node*** e define-se a *working directory*. Efetuam-se 2 *COPY* distintos, com intuito de dar cache, para que o tempo de *build* seja reduzido sempre que haja uma nova interação (excepto quando se adicionam novas dependências). Neste momento o container docker, corre na porta 8081, onde o **npm** internamente redireciona para http://192.168.1.6:8081/argon-dashboard-react. Os vários ficheiros javascript, que dizem respeito às páginas criadas, podem ser encontrados no diretório egs/biblify/react/app/src/views/examples/.

De seguida, após o professor fornecer todos os dados necessários para a utilização de ***kubernetes***, trata-se de definir o ficheiro *frontend.yml*. Neste ficheiro, declara-se um deployment, que cria 2 réplicas ou seja ***pods***, para que posteriormente seja feito load balancing aos vários pedidos. Aqui especifica-se o nome da app (*biblify-frontend*), assim como a imagem a usar. No passo seguinte, define-se o serviço, estabelecendo as várias portas necessárias. Sendo assim, tem de ir buscar o container que está a ser exposto na porta 8081 e expô-lo como serviço na porta que se definiu (sendo a porta 3000).

Por último , declara-se um ***ingress*** *(ingress.yaml)* para expor todo este serviço ao público, pois sem este passo, não é possível que um utilizador externo aceda ao mesmo. Efetua-se a descrição do campo *host*, para se definir o nome em que fica exposto. Desta forma, o mesmo pode ser alcançado pelo seguinte endereço http://biblify-frontend.k3s. É utilizado nginx como proxy. Ou seja, ingress expõe o serviço nginx que por sua vez serve o serviço de front-end.

O deployment, service e ConfigMap com configurações de nginx são declarados em *nginxproxy.yaml* . No *ingress* declarado, existem 3 hosts para cada um dos serviços: ‘biblify-frontend’, ‘biblify-stock’, ‘biblify-auth’.

Internamente, em termos de código *React*, apenas são alterados os endpoints de acesso aos vários serviços, trocando as portas que antes são definidas por *localhost,* pelo nome do serviço definido pelo deployment dos outros serviços.

O *deployment* deste serviço (*frontend*.yml) está no diretório *egs/biblify* e o *Dockerfile* da imagem em uso encontra-se em *egs/biblify/react/deploy*

Stock

A função do serviço stock é fornecer os dados sobre os livros existentes na biblioteca e a sua disponibilidade

As principais funcionalidades do serviço:

- adicionar e apagar livros
- obter informações sobre cada livro (autor, quantidade em stock etc)
- criar empréstimos de livros (requisição de um livro inclui informação como: requisitor, período, estado do empréstimo)
- consultar empréstimos em curso

Na primeira etapa de modelação do serviço, fez-se especificação do serviço e da respetiva API. Seguiu-se com a implementação do serviço e conteinerização do mesmo para colocar no cluster de kubernetes.

A documentação da API é feita com ajuda de Swagger. Na especificação de API são seguidas especificações OpenApi 2.0 e boas práticas ensinadas nas aulas.

Endpoints:

- /v1/book: GET e POST
- /v1/book/{bookId}: DELETE
- /v1/author: GET,POST
- /v1/author/{id}: DELETE
- /v1/orders : GET,POST
- /v1/orders/{id}: GET,DELETE
- /v1/user: GET, POST
- /v1/user/{id}: DETELE

Os endpoints relacionados com user foram criados com intuito de integrar a autenticação, mas não se chegou a implementar a autenticação na API.

A documentação de API pode ser visualizada em:

“http://biblify-stock.k3s/docs/” *

* Se o serviço de stock estiver a correr no cluster

Para a implementação de API usa-se FastApi com Sqlalchemy para operações com base de dados, base de dados PostgreSQL e Alembic para migrações.

A escolha de *FastAPI* justifica-se pelo bom desempenho, suporte de código assíncrono, fácil e rápido desenvolvimento.

O diretório stock/src contém o código da API implementada.

Para conteinerização foi criado docker-compose, em /stock/deploy. Tem seguintes componentes:

- Services:
  - web: servidor nginx como proxy
    - utiliza base imagem nginx:alpine
    - depende do serviço stock
    - corre na porta 80
  - db: base de dados
    - usa base image postgres:12.0-alpine
    - usa 2 volumes: um para persistência de dados e outro para scripts de inicialização
    - expose na porta 5432
    - parâmetros nas variáveis de ambiente (user,password,nome de bd)
  - stock: API
    - imagem criada com /src/Dockerfile
    - expose porta 8000
    - depende de serviço de base de dados
    - corre comando para dar start a aplicação
    - variável de ambiente com c*onnection string* para base de dados
- network:
  - *nginx_network* - bridge: para a intercomunicação dos serviços

A instanciação no cluster Kubernetes é bastante simples. São criados pods e serviços para stock API (stock/deploy/k8s_deploy/stockdeployment.yaml, stock/deploy/k8s_deploy/stockservice.yaml) e a respetiva base de dados (stock/deploy/k8s_deploy/postgresdeployment.yaml, stock/deploy/k8s_deploy/postgresservice.yaml). Para persistência de dados na BD, foi criado volume persistente (stock/deploy/k8s_deploy/storage.yaml).

É usado secret (postgres-user-pass) para passar credenciais de acesso à base de dados nas variáveis de ambiente.

Para maior disponibilidade de serviço foram criadas 3 réplicas de API. A base de dados só tem uma réplica. Pretendia-se ter várias réplicas de base de dados com volumes persistentes sincronizados. Para isso seria necessário criar StatefulSet para aplicação de base de dados com um volumeClaimTemplate com o qual stateful set conseguia criar volumes para réplicas.

Com a ferramenta RepMgr, incluída na imagem de postgres, é possível fazer replicação contínua nos pods. Para a replicação ocorrer tem de haver consistência nos pods, daí necessidade de usar stateful set para que os pods tenham sempre o mesmo nome. Na pasta /stock/deploy/k8s_deploy existe definição de stateful set, que usa imagem Bitnami postgres e a estratégia para aplicar configurações certas aos pods, para replicação contínua, é usar um script, em ConfigMap, que é executado quando o pod está a parar, para verificar se é master, caso seja, não termina, até existir um novo master nomeado no set. O nome do pod é obtido através de metadata.name. Um dos requisitos de Postgres stateful é que o *service* seja headless e não tenha IP atribuído pelo cluster.

Apenas se soube desta possível solução para replicação de base de dados na véspera da entrega, por isso não houve tempo suficiente para conseguir fazer deployment da mesma.

Na seguinte figura, pode visualizar-se o diagrama do deployment do projeto completo

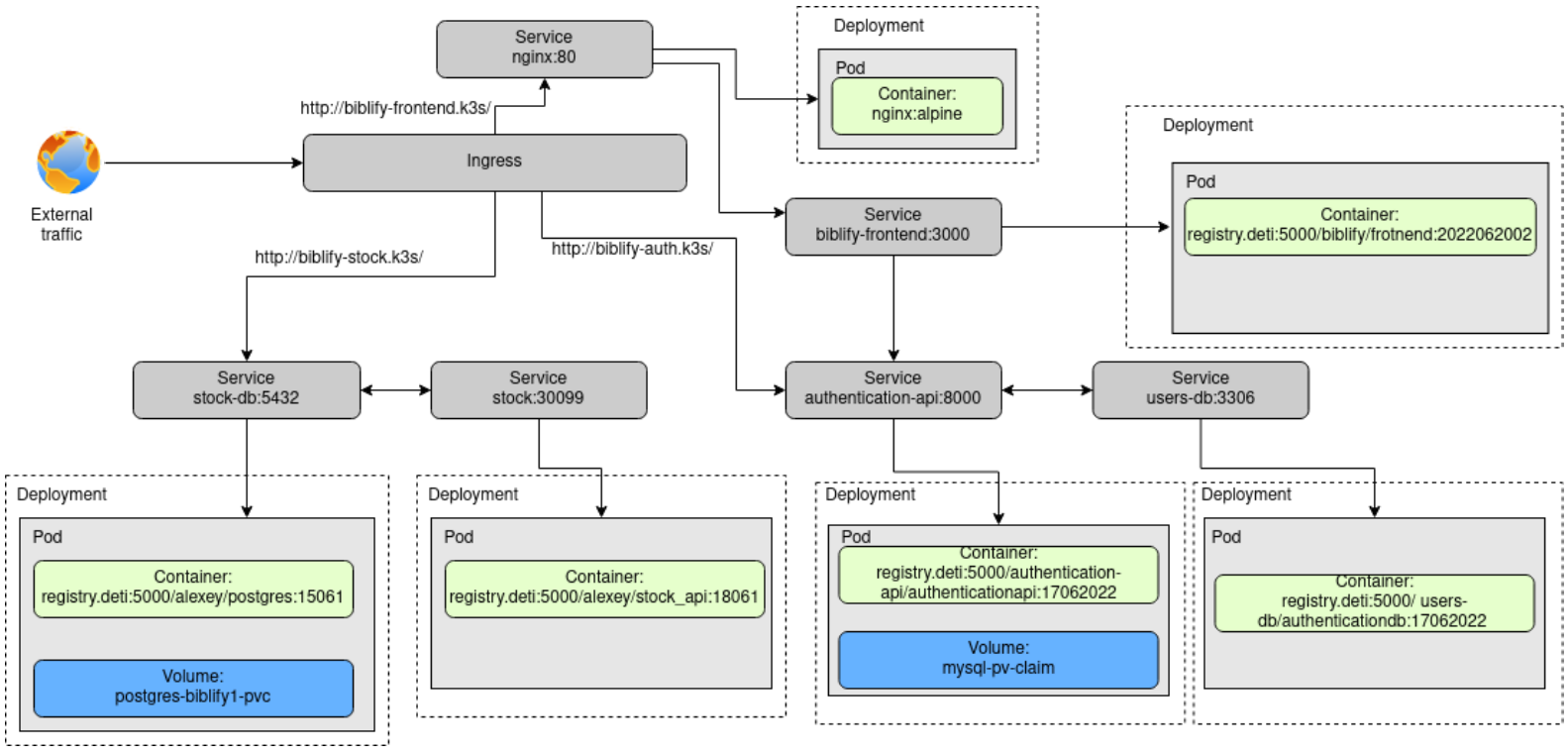

Fig. 2 - Kubernetes Deployment

Conclusão e Resultados

O resultado final é uma aplicação web que permite consultar e adicionar livros. Porém ainda regista e loga um utilizador. É ainda de notar, que o serviço de notificações não está implementado pelo facto do nosso colega Miguel Cabral não o ter implementado por razões externas ao grupo.

Dificuldades encontradas:

Na implementação de serviço de stock, tentou-se traduzir o código de especificação de API do Swagger para código de FastAPI mas não se conseguiu. Daí houve simplificação de API para reduzir a complexidade de implementação. Uma das vulnerabilidades existentes que têm de ser corrigidas no futuro é as credenciais da BD no docker-compose devem ser passados com secret, em vez de ser diretamente com environment variables. No cluster de kubernetes pretendia-se que a API de stock não fosse acessível do exterior, mas a única maneira com que front-end conseguisse comunicar com API era criando ingress. Deve-se ao facto de que app de front-end faz GET/POST à API do browser, fora de cluster.

No que diz respeito ao orquestrador, a sua implementação nem sempre é o mais linear. Visto que foi o primeiro contacto com a tecnologia *react*, nem tudo ficou o mais fluido possível. As ligações são feitas com os serviços que são implementados, mas o uso de certas funcionalidades, podiam dar mais consistência e robustez ao resultado final.

No serviço de autenticação, tentou-se traduzir a documentação da API do Swagger para código do FastAPI mas sem sucesso tendo que ser feito à mão. Perceber como iam funcionar os tokens que tinham de ser passados para as outras apis também foi uma tarefa complicada bem como as ligações à base de dados.

Bibliografia

Template REACT: https://www.creative-tim.com/product/argon-dashboard-react

Documentação para auxilio de composição dos deplyment: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

Tutorial de FastAPI com SQLAlchemy, PostgreSQL e Alembic: https://ahmed-nafies.medium.com/fastapi-with-sqlalchemy-postgresql-and-alembic-and-of-course-docker-f2b7411ee396