NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Relatório do 2º Projeto

Ano Letivo: 2024/25 Semestre: 1º Cadeira: SCC

Número de Aluno: 70252

Nome de Aluno: Diogo Alexandre Envia Matos

Email: dae.matos@campus.fct.unl.pt

Número de Aluno: 54471

Nome de Aluno: Henrique Malato Reynolds de Sousa

Email: hm.sousa@campus.fct.unl.pt

Introdução

O Tukano é um sistema de partilha de vídeos inspirado por sistemas como o TikTok, que implementa a capacidade de gestão de utilizadores, partilha de vídeos, gostar de vídeos e seguir outros utilizadores.
Para este projeto, tomou-se partido do portefólio de IaaS do ambiente Azure, da Microsoft para a efetuar o lançamento da aplicação e dos serviços dos quais esta depende como containers geridos pelo Kubernetes, deixando a criação de pods para este.
A diferença entre utilizar o portfólio de IaaS em relação à utilização de PaaS é que, com PaaS, a gestão de recursos da infraestrutura interna da Microsoft usada para a implementação da aplicação era feita apenas pelo sistema Azure, sendo que o programador não tinha possibilidade de definir os recursos a serem usados, enquanto que com IaaS o programador tem de definir quais são os recursos necessários para a aplicação e definir o seu funcionamento interno.

Instruções e configurações

Para correr o programa, é preciso obter o EXTERNAL-IP do “tukano-service”, o LoadBalancer do sistema, usando o comando “kubectl get service” depois de se fazer deploy do “tukano-service”. Depois, é preciso adicionar este EXTERNAL-IP como environment variable do deployment do tukano-image (no ficheiro “one_yaml_to_rule_them_all.yaml”, variável “KUBERNETES_CLUSTER_DNS”).
Por fim, corre o comando “kubectl apply -f one_yaml_to_rule_them_all.yaml” na pasta “SCC-Project-2” (a pasta “root” do projeto), depois de trocar o valor do “KUBERNETES_CLUSTER_DNS” pelo EXTERNAL-IP.
No entanto, o postgres não fica ativo imediatamente, é necessário tentar fazer um pedido no tukano para que a base de dados comece a ser criada.

Utilização dos recursos Azure IaaS

Kubernetes
Este serviço do sistema Azure disponibiliza um ambiente na cloud Microsoft que permite o lançamento de containers geridos pelo Kubernetes num cluster, com a associação dos recursos necessários para cumprir os requisitos definidos pelo programador neste sendo feita pela Azure.
Este serviço não disponibiliza funcionalidades para facilitar o lançamento de uma aplicação como um conjunto de containers no Kubernetes, com a tarefa da definição dos ficheiros “.yaml” utilizados para a criação da aplicação caber ao programador.

Microsserviços da aplicação

Aplicação: Tukano
A docker image usada para criar este microsserviço foi criada por nós:

hemareso/tukano-server
Este é o único microsserviço no cluster que permite receber pedidos HTML de elementos que não pertencem ao cluster, que significa que é o único que pode receber pedidos do cliente. Por isso, este microsserviço define a forma como o cliente pode interagir com o sistema.
Este microsserviço contém a lógica relacionada com a gestão de utilizadores e de shorts e é responsável por redirecionar pedidos para a execução de funções nos blobs para o microsserviço de "Blob Storage".

Blob Storage

A docker image usada para criar este microsserviço foi criada por nós:
diogomatos1232/tukano-server:latest
Este microsserviço é responsável pela lógica interna de gestão de blobs, implementando as funções de upload, download e remoção de blobs.
Os blobs são armazenados num Persistent Volume associado a este, que implementa a política de retenção "Delete", que significa que, quando este Volume é eliminado, é-se também eliminado os seus dados na estrutura de armazenamento. Isto não significa que os dados são apagados quando o Pod é apagado, os dados apenas são apagados quando se remove o Persistent Volume Claim.

Base de dados - PostgreSql

Este microsserviço foi criado usando uma docker image obtida a partir da internet
postgres (imagem oficial do Docker)
Usado para armazenar os *Users*, *Shorts*, *Followings* e *Likes*.
A manipulação dos dados armazenados neste microsserviço é feito através das funções definidas no microsserviço "Tukano".
A gestão dos dados armazenados neste microsserviço é possível através do uso "Hibernate", permitindo que o código do "JavaUsers" e do "JavaShorts" no código base pudesse ser aproveitado para gerir essa base de dados.

RedisCache

Este microsserviço foi criado usando uma docker image obtida a partir da internet
redis (imagem oficial do Docker)
Esta cache guarda os *users* e os *shorts* criados/acedidos mais recentemente.
As entidades guardadas na cache são identificados pelo seu "id" precedido por um identificador do tipo da entidade ("user:" ou "short:"). Este identificador garante que não possam acontecer conflitos devido ao tentar guardar um *user* e um *short* com o mesmo id.
Decidiu-se escolher apenas guardar *users* e *shorts* e não *followings* ou *likes* devido ao facto que a maioria das operações dependem de *users* ou de *shorts* e as operações que envolvem o uso de *followings* ou *likes* existentes no sistema irão ser usados com pouca frequência, por isso armazenar estes na cache seria um gasto de tempo e de espaço na cache.
Esta cache suporta inserções, alterações e remoções de uma entidade, usando o método write-through.

Para inserções, certifica-se que a entidade nova é adicionada na base de dados antes de ser adicionada na cache, para garantir que a cache não guarda entidades que não existem na BD.
Para alterações, primeiro remove-se da cache entidade a ser alterada antes de se fazer a alteração na base de dados, após o qual insere-se na cache a versão atualizada da cache. Desta maneira, garantimos que não é possível que a atualização seja feita apenas na cache ou na base de dados, com o outro armazenamento ficando com valores desatualizados.
Para remoções, as operações são efetuadas primeiro na cache antes de se fazerem na base de dados. Nas remoções, esta ordem impede que aconteça o caso em que a cache mantenha valores que foram removidos da base de dados.
Esta cache não é utilizada em queries de SQL, pois a cache não implementa integração com SQL. Para garantir que isto não apresente um problema, as únicas operações de SQL utilizadas neste programa são de recolha de dados (pois estes não alteram o estado da base de dados), e que nos casos em que se quer apagar vários elementos da base de dados (deleteShorts() e deleteAllShorts()), obtem-se primeiro uma lista dos elementos a serem removidos e removem-se estes da cache antes de se removerem da base de dados.

Outras Decisões

Introdução de um valor "id" nas entidades *user*, *short*, *following* e *like*

Para simplificar o processo de procura de entidades na base de dados ou na cache, decidimos que seria útil estas entidades terem um valor "id" (como no projeto anterior).
Os ids usados são:

- ***User –*** "userId" no código original (identificador único do *user*).
- ***Short –*** "shortId" no código original (identificador único do *short*).
- ***Following –*** "<follower>-<follower>". Como esta entidade originalmente tinha 2 ids, sendo estes os valores "follower" e "followee", foi preciso criar uma nova variável para servir como id único da entidade. Como todos os pares de "follower" e "followee" são únicos, o valor "id" único pode ser uma string construída a partir do follower e do followee.
- ***Like –*** "<userId>-<shortId>". A escolha deste valor como "id" é por razões semelhantes à entidade *following*, com "userId" e "shortId" em troca de "follower" e "follower".

Usar tukano como "middle-man" para a comunicação entre o cliente e o Blob Service

Decidimos que seria melhor o tukano servir de "middle-man" para a comunicação entre o cliente e o Blob Service porque esta forma permite que o cliente possa efetuar todas as operações deste sistema a partir do mesmo URL base (o do Tukano).
A alternativa ao "middle-man" seria permitir que o cliente tivesse de comunicar diretamente com o Blob Service para efetuar as operações relacionadas com *blobs*, que é mais inconveniente para o utilizador porque esta implementação iria requerer que o cliente precisasse de conhecer dois URLs base diferentes para poder efetuar todas as operações do sistema, sendo um deles o URL do Tukano, que permitiria realizar as operações

relacionadas com o *user* e os *shorts*, e o outro seria o do Blob Service, que seria utilizado para realizar as operações de gestão de *blobs*.

Domain name incorreto no blobUrl

Devido à dificuldade que tivemos para obter o EXTERNAL-IP dentro do projeto, não foi possível colocar o endpoint correto no *blobUrl*. Como tal, todos os *blobUrl* devolvidos estão com o endpoint interno do blob-storage, e não com o endpoint exterior do cluster. Assim, apenas é relevante o blobId e token deste url.
Queriamos ter corrigido este problema, se tivessemos tido mais tempo.

Métricas de Desempenho

Os testes usados para a criação das tabelas abaixo são os testes de Artillery criados pelo professor.

Média de Latência (L) and Throughput (T) de cada endpoint:

(Nota: os nomes dos endpoints não são literais)

Tabela obtida no primeiro projeto:

	NoSQL (com cache)		PostgreSQL (sem cache)		NoSQL (sem cache)		Projeto base	
estatísticas	L	T	L	T	L	T	L	T
/blobs	82,5	12,12	79,5	12,58	92	10,87	0,8	1250
/shorts/{{ shortId }}	81,5	12,27	66,5	15,038	85,7	11,67	1,9	526,32
/shorts/{{ shortId }}/likes?pwd={{ pwd }}	101,5	9,85	72,5	13,79	88,1	11,35	2,3	434,78
/shorts/{{ shortId }}/{{ userId }}/likes?pwd={{ pwd }}	94,6	10,57	72	13,89	109,5	9,13	2,2	454,55
/shorts/{{ userId }}/feed?pwd={{ pwd }}	74,2	13,48	67,7	14,77	86,8	11,52	2,3	434,78
/shorts/{{ userId }}/followers?pwd={{ pwd }}	69,8	14,33	69,7	14,35	71,8	13,93	1,8	555,56
/shorts/{{ userId }}/shorts	69,7	14,35	65,7	15,22	73,5	13,61	2,3	434,78
/shorts/{{ userId }}?pwd={{ pwd }}	78,6	12,72	64,7	15,46	77,7	12,87	2	500
/shorts/{{ userId1 }}/{{ userId2 }}/followers?pwd={{ pwd }}	94,3	10,60	75,1	13,32	85,6	11,68	2,1	476,19
/users	69	14,49	64,4	15,53	69,8	14,33	1,5	666,67

Tabela obtida no segundo projeto:

	PostgreSQL (com cache)		PostgreSQL (sem cache)		1º Projeto (sem cache)	
estatísticas	L	T	L	T	L	T
/blobs	94,3	10,60	104,5	9,57	79,5	12,58
/shorts/{{ shortId }}	83,7	11,94	85,9	11,64	66,5	15,038
/shorts/{{ shortId }}/likes?pwd={{ pwd }}	83	12,04	77,8	12,85	72,5	13,79
/shorts/{{ shortId }}/{{ userId }}/likes?pwd={{ pwd }}	90,7	11,03	79,3	12,61	72	13,89
/shorts/{{ userId }}/feed?pwd={{ pwd }}	77,3	12,94	85,2	11,74	67,7	14,77
/shorts/{{ userId }}/followers?pwd={{ pwd }}	74,8	13,37	101,8	9,823	69,7	14,35
/shorts/{{ userId }}/shorts	81,1	12,33	83,8	11,93	65,7	15,22
/shorts/{{ userId }}?pwd={{ pwd }}	78,8	12,69	82,7	12,09	64,7	15,46
/shorts/{{ userId1 }}/{{ userId2 }}/followers?pwd={{ pwd }}	86,5	11,56	96,2	10,40	75,1	13,32
/users	79,9	12,52	83,8	11,93	64,4	15,53
	70,1	14,27	82,2	12,17		

Notas sobre os testes

Para fazer os testes, foi necessário ir buscar o EXTERNAL-IP do LoadBalancer que dá acesso ao projeto. Colocamos o valor nos testes de artillery para os podermos correr.

Analise dos Dados

Considerando os dados adquiridos com os testes do Artillery, podemos fazer as seguintes conclusões:

Cache vs Sem Cache
Regra geral ter cache reduz a latência dos pedidos. Os dados apresentam algumas excepções (nos endpoints de *like*), mas acreditamos que esses casos devem-se a não só à cache já existente no postgres, como à reduzida amostra de dados.
Os restantes endpoints mostram que a cache é uma solução é um investimento relevante para o nosso cluster.

Primeiro Projeto vs Segundo Projeto
No primeiro projeto, apenas implementamos a solução de Postgres sem cache, por isso, é a nossa única referência para comparar com a utilização do Kubernetes.
Comparando os projetos sem cache, o segundo projeto tem piores métricas. Elas devem

ser resultado de uma maior necessidade de comunicação entre peças dentro do cluster. Mas também é de se notar, que não estamos a usar várias réplicas. Acreditamos que com o scale up da aplicação, devido ao uso de *LoadBalancers* e mais recursos disponibilizados pelo Kubernetes, poderíamos aumentar a performance sem um grande esforço da parte dos programadores.

O projeto com Kubernetes pode ter resultados menos rápidos, mas tem um aumento considerável na sua facilidade de manutenção (uso de kubectl) e gestão de recursos (simples valores em ficheiros yaml).