NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Relatório do 1º 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 implementar este sistema num ambiente cloud, foram-se utilizadas várias ferramentas da PaaS Azure, da Microsoft, para efetuar o lançamento da aplicação e das várias bases de dados dependentes desta em servidores da Microsoft, em vez de ser necessário gerir servidores para a aplicação.

Instruções e configurações

Este projeto apresenta a capacidade de escolher entre uma implementação que utiliza o “CosmosDB with NoSQL” ou uma que usa o “CosmosDB with PostgreSQL”. A escolha da implementação do “CosmosDB” é feita através do valor de “USE_SQL” definido no ficheiro "azurekeys-region.props". Se esse valor é *false* (o valor por defeito), usa-se “NoSQL”, caso contrário, se o valor é *true* usa-se o “PostgreSQL”.

Este projeto também apresenta a capacidade de escolher uma implementação do armazenamento dos blobs com através de “Azure Storage Account” (JavaAzureBlobs) ou a partir do sistema de ficheiros do computador (JavaFileBlobs) (Testado?). A escolha da implementação do armazenamento de blobs é feita através do valor de “USE_AZURE_BLOB_STORAGE” definido no ficheiro "azurekeys-region.props". Se esse valor é *false* , usa-se o sistema de ficheiros do computador, caso contrário, se o valor é *true* (o valor por defeito) usa-se o “Azure Storage Account”.

Utilização dos recursos Azure PaaS

App Service
Serviço da plataforma que permite que um utilizador possa lançar uma aplicação criada por este num ambiente cloud, utilizando recursos da Microsoft, sem ser necessário gerir servidores ou atribuir recursos para este fim.
Neste projeto, é usado para lançar a aplicação do Tukano na cloud.

Storage Account
Usado para armazenar os *blobs*.
Os *blobs* estão todos armazenados num *blob storage container* com o nome “blobs”. Dentro deste *container*, os *blobs* estão organizados em diversas diretorias, cada uma contendo os *blobs* referente aos *shorts* de um determinado *user* do sistema.
Esta organização de *blobs* em pastas foi feita para que não existisse mistura de *blobs* de *users* diferentes no sistema, removendo a necessidade de atribuir o nome do *user* no nome do blob no sistema, mas também porque esta organização facilita a operação de remoção de todos os *blobs* de um *user*, quando se quer remover este do sistema.

CosmosDB

Usado para armazenar os *Users*, *Shorts*, *Followings* e *Likes*.
Esta base de dados está dividida em 4 *containers*, uma para cada tipo de dados armazenado.
Implementou-se duas bases de dados de CosmosDB, uma baseada no uso de "NoSQL" e outra que utiliza o "PostgreSQL". A gestão dos dados armazenados em "CosmosDB with PostgreSQL" é possível através do "Hibernate", permitindo que o código do "JavaUsers" e do "JavaShorts" no código base pudesse ser aproveitado para gerir essa base de dados (na versão final do código, estes correspondem às classes "JavaHibernateUsers" e "JavaHibernateShorts", respetivamente).
As *PartitionKeys* usadas são:

- **Container *Users*:** /id – identificador único do *user* (anteriormente era "userId")
- **Container *Shorts*:** /ownerId – identificador do *user* que criou o *short*. Usado para facilitar a pesquisa de *shorts* criados por um determinado *user* quando se quer remover os shorts criados por este (operação "deleteAllShorts()").
- **Container *Followings*:** /followee – identificador do *user* a ser seguido. Usado para facilitar a pesquisa de *users* que seguem um determinado *user* (operação "followers()"). Uma alternativa seria usar /follower, que faria a operação "followers()" mais lenta para acelerar a operação "getFeed()".
- **Container *Likes*:** /userId – identificador do *user* que fez like. Serve para facilitar a remoção de todos os *likes* que um *user* criou quando se quer apagar este do sistema ("deleteUser()"). Uma alternativa para este seria /shortId, que facilitaria a execução da operação "likes()" para obter todos os utilizadores que fizeram like a um determinado short mas faria a operação "deleteUser()" mais lenta.

RedisCache

Esta cache guarda os *users* e os *shorts* criados/acedidos mais recentemente.
A RedisCache não é utilizada quando se utiliza o CosmosDB com o "PostgreSQL", pois a integração com a RedisCache não foi implementada no "DBHibernate".
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 e 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. No entanto, esta ordem não é muito adequada para alterações, pois não impede o caso em que uma entidade é alterada na cache mas, por ocorrência de erro, não na base de dados.

Esta cache não é utilizada em queries de SQL para recolher entidades, isto é, queries de "SELECT" devido ao facto que a RedisCache não é adequada para o uso com queries, pois estas envolvem procurar pela base de dados completa, por isso a cache faria o processo mais lento. No entanto, certificou-se que, sempre que se quer remover várias entidades numa só operação, como no "deleteAllShorts()", certificou-se que as entidades eram removidas da cache antes de se efetuar a operação na base de dados.

Outras Decisões

Manter a versão original do JavaBlobs (JavaFileBlobs) e o FileSystemStorage

Decidimos manter a versão original do JavaBlobs e do FileSystemStorage porque, apesar de não serem relevantes para o armazenamento dos blobs através de serviços do Azure, considerámos relevante possibilitar o uso de armazenamento de blobs local em simultâneo com o uso de serviços de Azure para implementar o JavaUsers e o JavaShorts.

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

Como o CosmosDB não permite que sejam criadas entidades que não têm um valor de "id", foi necessário adicionar este valor nas entidades do sistema para que o seu armazenamento fosse possível.
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".

Inserção de transaction no DBCosmos

Como o código do Hibernate apresentava suporte para transaction, concluímos que seria necessário implementar uma função semelhante para o DBCosmos, para que, caso fosse necessário utilizar transações para realização de funções do sistema, o DBCosmos já estará pronto para tratar desta funcionalidade.

Métricas de Desempenho

Os testes usados para a criação da tabela em baixo são os testes de Artillery criados pelo professor.

Os testes de “NoSQL” e de “PostgreSQL” foram feitos a partir de um lançamento da aplicação no Azure, enquanto que os testes do “Projeto base”, por não ter integração com as ferramentas do Azure, foram feitas com a aplicação lançada no Tomcat.

O “Projeto base” tem algumas alterações para que fosse possível testar essa no Tomcat. No entanto, estas mudanças causaram problemas do tipo 500 no “getShorts()” e no “getFeed()”, por isso, os resultados podem não estar completamente certos.

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

(Nota: os nomes dos endpoints não são literais)

	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

Considerando os dados adquiridos com os testes do Artillery, podemos primeiro observar que o Projeto base tem valores muito melhores que os outros Deployments. Isto deve-se ao facto que, ao contrário dos outros testes, o Projeto base esteve a correr num ambiente Tomcat local, por isso, não puderam ocorrer problemas de latência. Não chegamos a ter o projeto base a correr com deployment Azure, uma vez que estava a dar erros (e o código base não estava feito para ser usado no Azure). Agora, apercebemos-nos que nos faltou

capacidade de previsão, para podermos ter arranjado uma amostra de controlo mais relevante.
Outra diferença óbvia, é que apesar de terem performances muito piores, os projetos deployed em Azure são mais seguros, resilientes, escaláveis que o projeto em tomcat (entre todos os outros benefícios presentes na utilização de recursos Azure PaaS).

Olhando para a comparação do projeto com NoSQL com e sem cache. Podemos observar que o uso da cache causa melhorias em funções de GET mas causa um aumento de latência notável para funções de escrita do sistema (POST). Isto deve-se ao facto que a procura de entidades na cache é mais rápida que a procura na base de dados, mas a escrita de dados é mais lenta devido a ser necessário introduzir os dados na cache em adição à base de dados, devido ao método usado para a cache ser o write-through.

As funções que dependem apenas no uso de queries de procura (“SELECT”) de SQL não são afetadas pelo uso de cache, devido a não termos integrado a cache com o SQL.

Acreditamos que a longo prazo a cache poderia apresentar latências melhores nas funções de GET.

A segunda comparação é entre o NoSql e o Postgres. É de se notar que não chegamos a implementar a cache no Postgres. Como tal, o projeto com Postgres consegue apenas comparar resultados com NoSql sem cache.
Entre ambos os projetos NoSql e Postgres sem cache, o postgres teve índices de performance superiores a ambas as implementações do NoSQL.

As razões disto são, possivelmente, a melhor performance natural dos queries. Internamente, o Postgres é um produto open source que recebe muita atenção regular. Como tal, ser resultado de um simples melhor produto.
Outra possível conclusão pode ser a interferência do hibernate. A framework pode oferecer uma melhoria na performance que não existe na utilização do NoSql.

Acreditamos que se tivéssemos colocado cache no PostgreSQL, teríamos resultados muito superiores ao que obtivemos sem cache.