Sistema de Computação em Cloud

2021/2022

Relatório 1º Projeto

Elaborado por:

André Matos 55358

João Palma 55414

Ruben Belo 55967

Docentes:

Nuno Preguiça

João Leitão

Dezembro 3, 2021

Para o desenvolvimento do trabalho optámos pela utilização do Spring.

Para as interações com o blob storage criamos um controlador(/media) como indicado, onde é possível o upload, download e listagem dos elementos existentes. Para esta concretização foi necessario criar ainda as classes StorageConfig e StoreProperties que permitem a ligação com o blob storage(Nota: Todas as credenciais necessárias foram introduzidas no ficheiro application.properties).

Relativamente à segurança implementada, optamos por um sistema de Login/Logout com JWT Tokens. Na classe SecurityConfig são definidos quais os pedidos que não requerem autenticação. Todos os outros passam pelo TokenValidationFilter em que verificamos a validade do token e emitimos um token novo na resposta. É também nesta classe que guardamos o user como principal para usos futuros no servidor. O token JWT carrega consigo o username do utilizador, o tempo em que o token foi emitido, assim como o tempo em que irá expirar(10min). O token é encriptado com um algoritmo de hashing(HS512) e com um SECRET definido pelo servidor e apenas este tem conhecimento do seu valor.

Ainda sobre a segurança, possuímos também um encoder(BCryptPasswordEncoder) de passwords para que as palavras-passe dos utilizador não fiquem visíveis na base de dados. Certificamo-nos ainda que sempre que a resposta do servidor devolvesse um utilizador com a palavra-passe, esta não é retornada mesmo estando encriptada(Com exceção do registro de contas apenas devido a testes no artillery).

No CosmosDB estão presentes 3 repositórios(Users,Channels,Messages) e para a interação com o servidor foi apenas necessário introduzir os dados no ficheiro application.properties.

Existem 3 entidades, são estas:

Users que possuem id(Id, gerado pelo servidor), name, pwd(password), imageId(id da sua imagem no blobStorage), username(único, utilizado para o login) e um HashSet channelIds(com todos os ids dos canais em que o utilizador é dono/membro ou está subscrito).

Channels que possuem id(Id, gerado pelo servidor), name, privateChannel(true/false), ownerId e uma lista de membros.

Messages que possuem id(Id, gerado pelo servidor), creationDate(introduzido pelo servidor com o intuito de ordenar em listagens), replyTo(id da mensagem a dar reply), channel(id do channel para que foi enviada), user(id do utilizador que a enviou), text(o conteúdo da mensagem) e imageId(id da sua imagem no blobStorage).

EndPoints
/rest/user:

POST registerUser - Requer: pwd,name,username Optional: imageId - Cria um utilizador - Retorna:Utilizador
PUT (requer autenticação) updateUser - Optional:imageId, name, pwd - Atualiza os dados de um utilizador -Retorna:Utilizador
PUT /login login - Requer: username,pwd -Inicia sessão
PUT /logout (requer autenticação) logout
GET /{userId}(requer autenticação) getUser- Requer: userId- Devolve um utilizador -Retorna:Utilizador
DELETE/{userId}(requer autenticação) deleteUser -Requer:userId - Elimina um utilizador
PUT (requer autenticação) /subscribe/{channelId} subscribeChannel -Requer:channelId- Subscreve o user num canal
PUT (requer autenticação) /unSubscribe/{channelId} unSubscribeChannel -Requer:channelId- Remove a subscrição de um canal
GET(requer autenticação) /channels/own getUserChannels - Devolve as lista de canais em que o user é dono, membro ou está subscrito

/rest/channel(requer autenticação):

POST createChannel -Requer:name,privateChannel(true/false) Opcional:members- Cria um canal -Retorna:Canal
PUT updateChannel - Requer:id, name - Atualiza o nome do canal -Retorna:Canal
DELETE /{id} - Requer: id do canal -Elimina um canal
GET/{id} - Requer: id do canal -Devolve um canal-Retorna:Canal
PUT /{id}/addMember/{userId} - Requer: id do canal, userId - Adiciona um mebro ao canal-Retorna:Canal
PUT /{id}/removeMember/{userId} - Requer: id do canal, userId - Remove um mebro do canal-Retorna:Canal
GET /{id}/messages getChannelMessages- Requer: id do canal, número de items na página e número da página. Devolve as mensagens de um channel, em paginação.
GET /{id}/messages/search/{text} getChannelMessagesTextSearch- Requer: id do canal, text, número de items na página e número da página. Devolve as mensagens de um channel que contêm um dado texto, em paginação.

/rest/messages(requer autenticação):

GET /{id} getMessage -Requer: Id da mensagem. Devolve a mensagem com o id dado, caso o user logado seja o dono.
DELETE /{id} deleteMessage - Requer: Id da mensagem. Elimina a mensagem com o id dado, caso o user logado seja o dono.
POST /messages/ deleteMessage - Adiciona uma mensagem. Retorna:Id da mensagem.

Function

Nas funções achámos interessante fazer 4 que irão ser descritas de seguida, sendo que 3 delas complementam o projeto em si e a outra é apenas informativa. Esta última diz quantas mensagens foram enviadas no último dia, portanto é um timer trigger que ativa de 24 em 24 horas e assim podemos ter uma noção do fluxo que existe diariamente. Relativamente às outras, começamos pela deleteUser:

Quando é chamado o endpoint que apaga um user, o que fazemos é mudar a palavra passe deste para "DELETED" e depois com um timer trigger executar todas as operações adicionais de 5 em 5 horas. Quando um user é apagado, todas as mensagens desse mesmo user precisam ser alteradas, mudando o user da mensagem para "Deleted User". Também é necessário apagar os channels quando o user apagado é owner destes, e remover estes channels de todos os users que estavam subscritos. Por último, quando o user não é o dono mas pertence aos channels, temos de o ir tirar das lista dos membros destes. No final é que apagamos os users da base de dados.

No deleteMessage optámos por uma abordagem semelhante:

Quando é chamado o endpoint que apaga uma mensagem, mudamos o replyTo desta para "DELETED", depois de 5 em 5 horas vamos a todas as mensagens que respondem às mensagens apagadas e o replyTo para "Deleted Message" e apagamos da base de dados as devidas mensagens.

No fim destas duas operações damos a informação de quantos users/messages foram apagados. Para finalizar, a última função é um BlobTrigger, e serve para replicar as imagens que são colocadas no servidor WestEurope para o servidor EastUS.

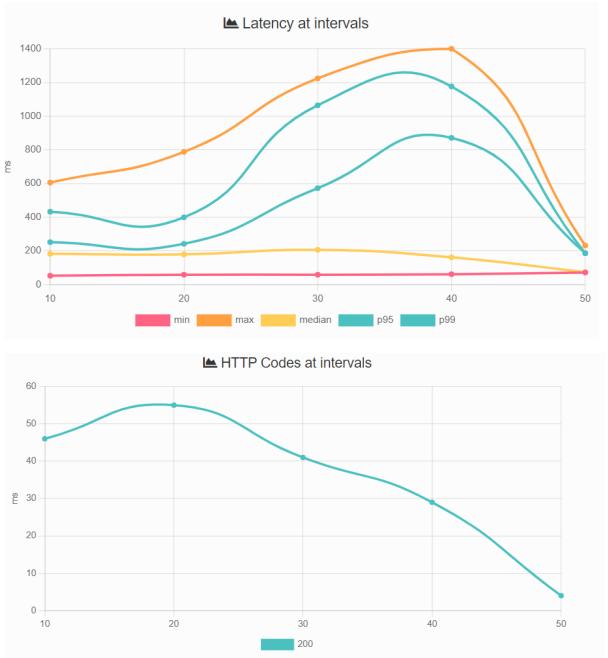
Artillery/testes

Para avaliar a aplicação desenvolvida utilizamos um conjunto de testes feitos com o artillery em combinação de testes feitos por nós.
Os testes criados e realizados pela equipa consistem em pedidos http singulares realizados no postman ou no IntelliJ e tinham como objetivo verificar a consistência e fidelidade do código ao executar vários testes tais como:
Apenas o owner de um canal conseguir adicionar membros;
Apenas o owner da mensagem conseguir apagar uma mensagem dele;
Um user só consegue criar um mensagem se for membro do canal;
Apagar um user e uma mensagem e verificar se estes são apagados com as functions que criamos;
Entre outros testes;

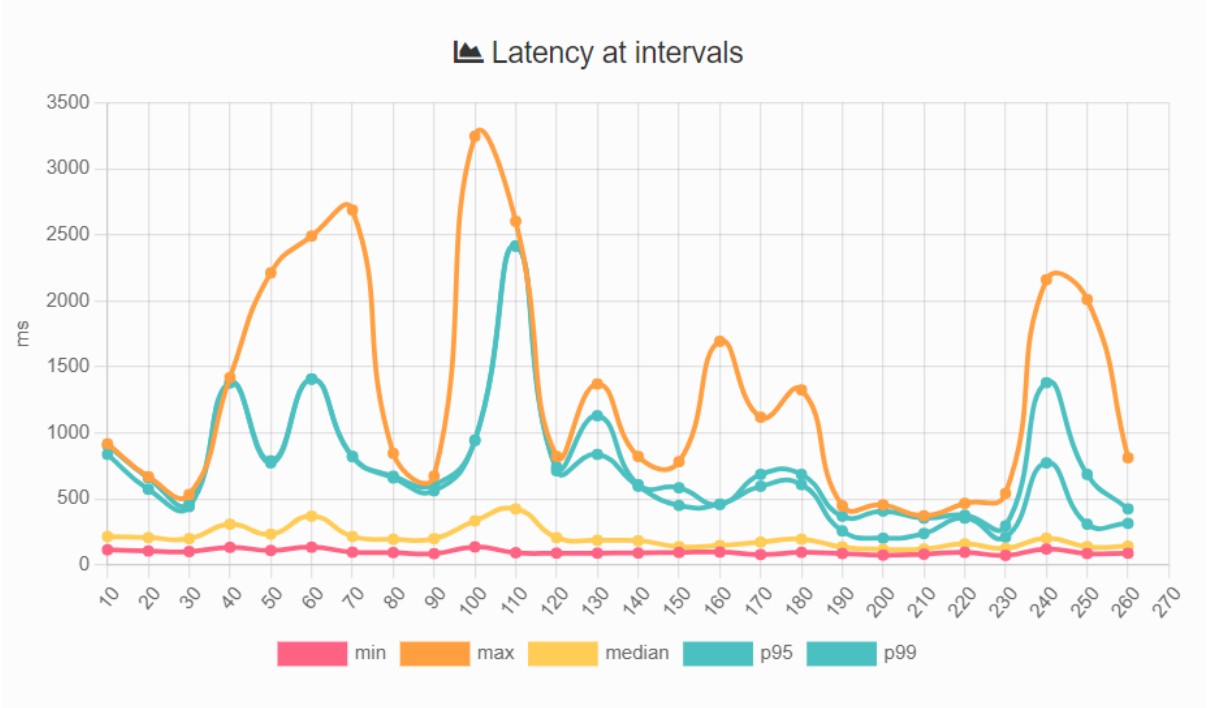
Ao executar os testes do artillery verificamos que o nosso servidor funcionava como desejado.
Com estes testes do artillery averiguamos várias situações:
Ao tentar subscrever ou adicionar um user a um canal onde já está subscrito, obtemos no body da resposta erro code 409;
Ao tentar subscrever ou adicionar o owner a um dos seus canais obtemos o erro code 403.

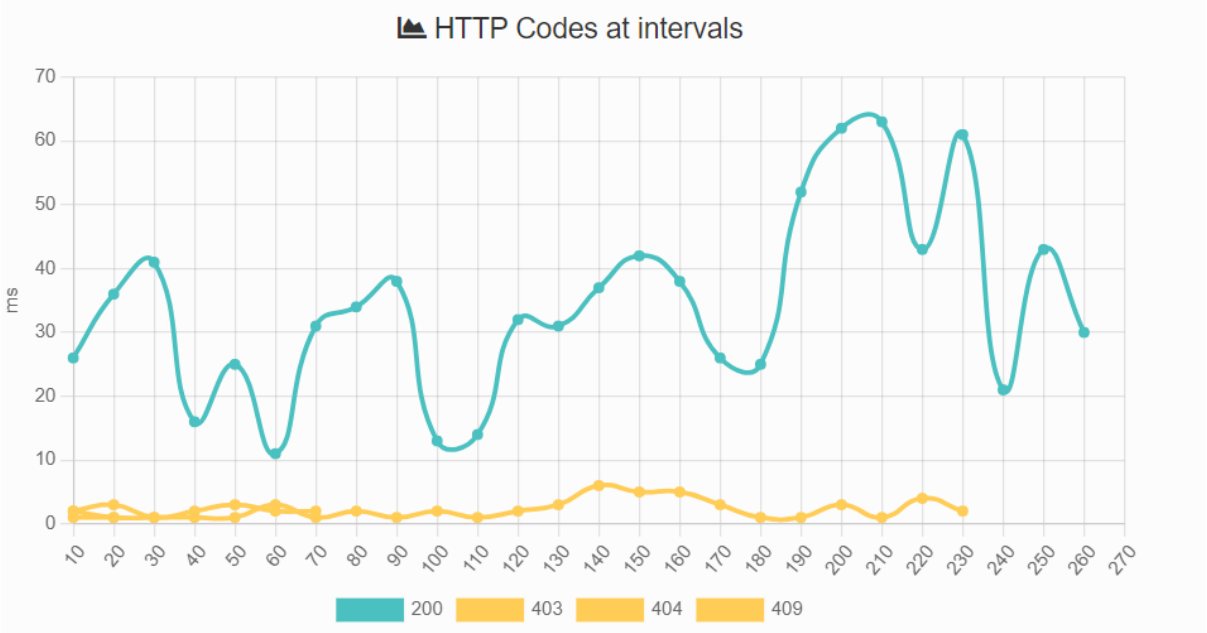
Decidimos acrescentar 2 novos testes ao artillery para demonstrar novos endpoints que adicionamos à aplicação. Um dos testes demonstra a adição e a remoção de um user a um canal privado e o segundo demonstra o Azure Cognitive Search fazendo print na consola às mensagens de um canal que contém o texto de input.

Estatísticas para os testes do artillery:

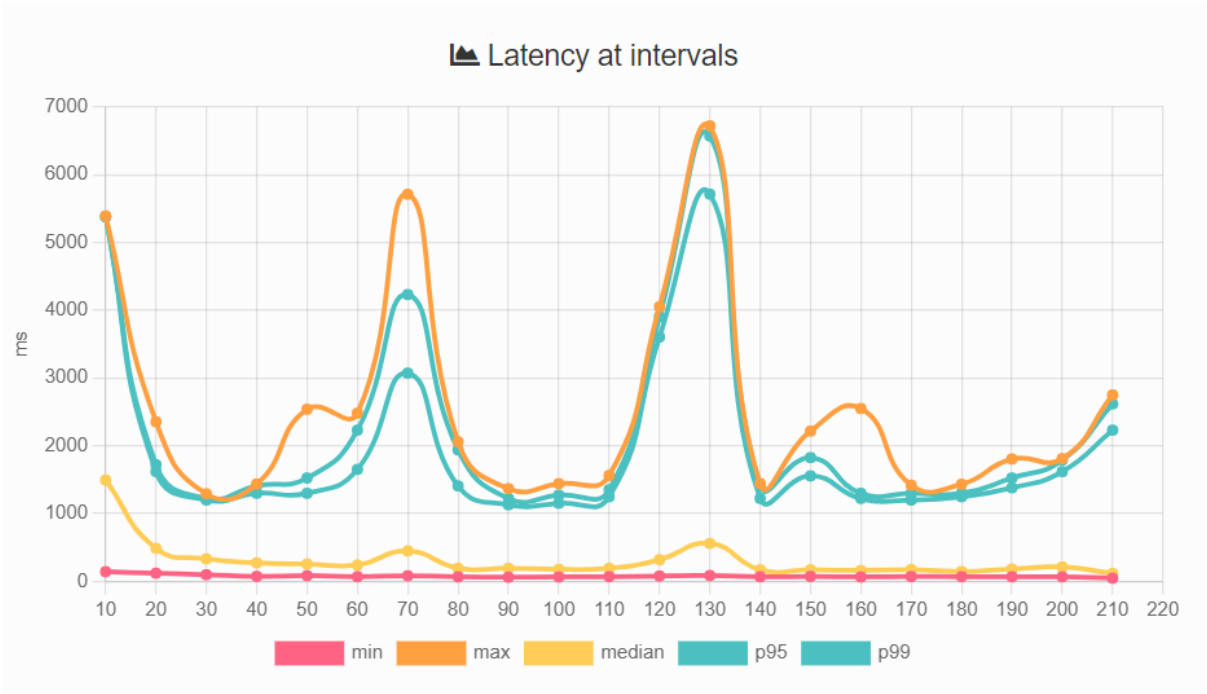


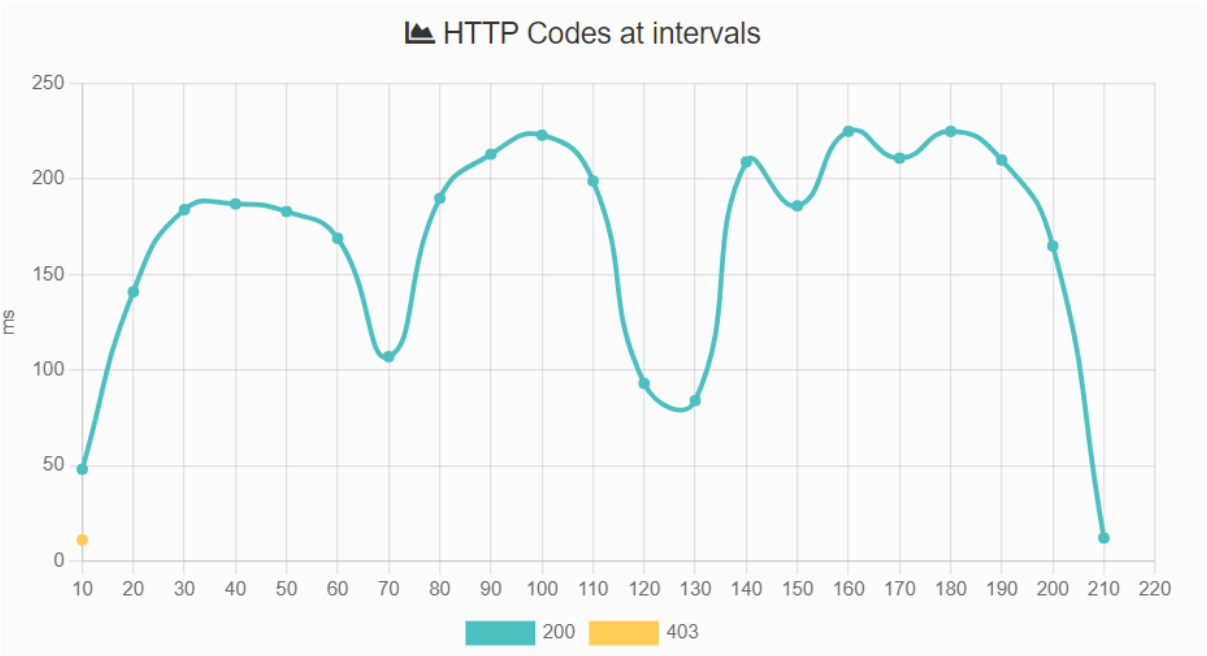
Estatísticas para o create-user.





Estatísticas para o create-channels





Estatísticas para o create-messages