Fase 3 - Requisitos, Casos de Uso e Arquitetura

Pedro Carrega, nº49480 Vasco Ferreira, nº49470
Ye Yang, nº 49521

17 de Abril de 2020

Conteúdo

1 Motivação para Dataset e Serviços

O dataset ecommerce foi escolhido pelo grupo devido ao grande número de eventos gerados e consequente informação produzida durante a utilização de uma loja de ecommerce. Informação esta que pode ser utilizada de diversas formas através de um grande número de variados serviços. Essa mesma informação poderá ser utilizada em vários contextos, sendo que escolhemos os seguintes 5 serviços que demonstram diferentes tipos de informação sobre o dataset:

- api/products/listCategories: Fornece todas as diferentes categorias presentes nos dados
- api/products/popularBrands: Fornece a contagem de eventos associados a cada marca
- api/products/salesByBrand: Lista o numero de vendas de cada marca
- api/products/salePrice: Calcula o valor médio de venda de uma determinada marca
- api/events/ratio: Apresenta a distribuição relativa de cada tipo de evento, havendo os possíveis valores: view, cart e purchase

Os serviços foram escolhidos de forma a que consigam fazer diferentes tipos de operações sobre os dados, desde serviços mais específicos e por isso com menos carga na base de dados, a serviços mais abrangentes e consequente aumento de carga. Foram também escolhidos pois todos os serviços fornecem dados úteis para serem explorados no contexto de lojas de ecommerce.

2 Diagrama de Casos de Uso

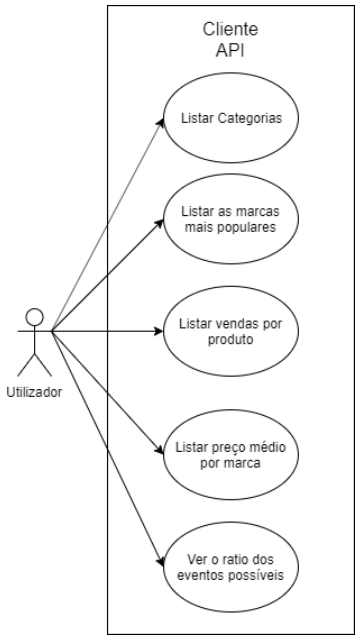

Figura 1: Diagrama de casos de uso

3 Requisitos

Requisitos Não Funcionais	**Descrição**
Portabilidade	Implementação de servidor em NodeJS e cliente em HTML de modo a facilitar o processo de mudança de plataforma do serviço
Legibilidade	A separação clara entre as camadas de apresentação, lógica de negócio e acesso à base de dados irá tornar o fluxo do sistema mais legível para os desenvolvedores
Estabilidade	A distribuição do sistema por diversas máquinas virtuais, que poderão estar distribuídas por diferentes fornecedores cloud e em diferentes Data Centers permitem obter um sistema estável
Elasticidade	O sistema deverá ser capaz de se adaptar à carga de trabalho através do provisionamento e desprovisionamento dos recursos de forma autónoma. Idealmente, de forma que em qualquer ponto do tempo, o sistema apenas utilize o número de máquinas necessárias de forma a corresponder à carga atual
Escalabilidade	Capacidade do sistema lidar com o crescimento de carga de trabalho. Pode-se associar à capacidade de elasticidade do sistema
Confiabilidade	Visto o sistema estar implementado na núvem, conseguimos garantir alta confiabilidade nos sistema, pois se um servidor no datacenter falhar, conseguimos facilmente migrar a VM para outro servidor funcional

Tabela 1: Requisitos Não Funcionais

Requisitos Funcionais	**Descrição**
Listar Categorias Disponíveis	Serviço que fornece aos clientes todas as categorias
Visualizar Popularidade das Marcas	Serviço que fornece cada marca associada com a sua popularidade
Visualizar Número de Vendas Individuais	Fornece o numero total de vendas de cada marca
Visualizar Preço Médio de Venda	Fornece o preço médio dos produtos vendidos de uma determinada marca
Visualizar Rácio de Tipo de Eventos	Fornece a percentagem de cada tipo de evento

Tabela 2: Requisitos Funcionais

4 Arquitetura da aplicação

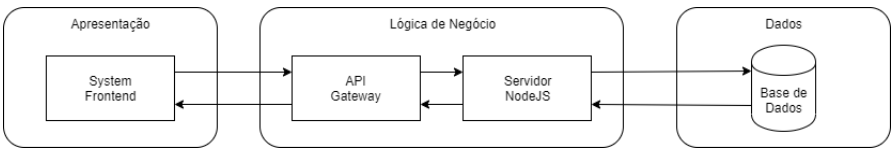

Figura 2: Arquitetura da aplicação

Após a definição da API na fase anterior, foi implementado de raiz um servidor em NodeJS que segue a API definida.

4.1 Load Balancer

Foi implementado um Load Balancer que distribui os pedidos HTTP recebidos dos clientes browser e que reencaminha para os respetivos serviços.

4.2 Servidor NodeJS

O servidor irá receber os pedidos a partir do Load Balancer, e processá-los de acordo com a funcionalidade pretendida. A lógica de negócio é aqui tratada, realizando as queries necessárias para a base de dados. Ao receber os dados, processa-os de acordo com a funcionalidade, e encaminha o resultado para o Load Balancer.

4.3 Base de dados

A base de dados irá conter o conteúdo dos ficheiros .csv do nosso dataset, que são acedidos pelo servidor de modo a poder efetuar as leituras necessárias para produzir uma resposta para o cliente.

5 Arquitetura técnica

Para garantir os requisitos não funcionais mencionados a cima devemos ter em atenção os serviços da cloud escolhidos, pois proporcionam vários aspetos fundamentais dos requisitos.

A utilização dos serviços disponibilizados pela AWS permite cumprir alguns dos requisitos não funcionais mencionados, nomeadamente a Escalabilidade e a Elasticidade. Já o Load Balancer implementado com o Ingress permite que este serviço da cloud trate automaticamente da separação dos diferentes tipos de pedidos HTTP recebidos para os respetivos serviços. A base de dados utilizada consiste uma imagem docker baseada na imagem mongo que como o nome sugere é uma imagem de MongoDB. Foi escolhida esta alternativa de forma a não estar dependente dos serviços de Bases de Dados de um fornecedor específico, sendo que com esta opção a imagem pode ser implementada numa instância em qualquer fornecedor cloud. Quanto à escalabilidade a imagem poderia ser replicada por diversas réplicas de forma a distribuir os pedidos feitos pelas diferentes réplicas.

A Confiabilidade do sistema também é melhorada com a instalação em serviços na cloud, visto estas terem deteções automáticas de falhas de servidores, possibilitando a migração do sistema virtualizado para outro servidor funcional.

A Estabilidade é garantida nas várias ferramentas usadas na instalação do sistema. O Load Balancer como referido anteriormente faz a distribuição dos pedidos HTTP para os respetivos serviços, sendo que o Kubernetes atribui um DNS igual para todas as réplicas do serviço de maneira a que o Load Balancer apenas reencaminhe para o respetivo DNS, sendo o Kubernetes a distribuir a carga pelas respetivas réplicas do serviço. Isto permite que o sistema escale com facilidade já que aumentar o número de réplicas de um micro-serviço em Kubernetes é um processo rápido e sem necessitar de qualquer alteração no Load Balancer.

Já a Estabilidade é garantida devido a ter a Lógica de negócio distribuída por várias máquinas virtuais que cumprem a função de servidores.

A utilização do Docker permite-nos facilmente fazer deploy de um serviço,

em qualquer máquina e em qualquer um fornecedor de cloud. Isto deve-se ao Dockerfile garantir a instalação de todas as dependências do serviço, variando conforme a linguagem de implementação do mesmo, cumprindo assim o requisito de Portabilidade do sistema.

6 Implementação

6.1 Base de dados

Uma das maiores dificuldades que tivemos foi a passagem dos dados em formato CSV para uma tabela na DynamoDB. Começamos primeiro com a criação de scripts em Python, com a utilização de bibliotecas disponibilizadas pela AWS. Isto mostrou-se não fazível devido às limitações do tamanho do heap e da memória consumida, visto o ficheiro CSV ser de grandes dimensões.

Passamos para a implementação do mesmo script mas através do AWS Lambda mas deparamo-nos com o mesmo problema, sem sucesso na escrita para a DynamoDB.

Finalmente utilizamos a ferramenta do Data Pipeline, que por sua vez originou vários novos obstáculos:

1. Deparamo-nos com vários erros na passagem do CSV para a DynamoDB. Estes erros originaram do facto da DynamoDB ter um formato específico de JSON que contém não só o nome de cada coluna, como o tipo da coluna. Esta informação não se encontrava explicitamente documentada, logo para obter a estrutura tivemos que escrever uma linha manualmente numa tabela criada, exportar usando o Data Pipeline para um bucket S3 e daí conseguimos observar a estrutura. De seguida criamos um parser que percorria o dataset, eliminando espaços vazios que não são aceitadas pela DynamoDB, tornando cada linha no formato JSON que a DynamoDB conseguisse interpretar.

2. No início da utilização do Data Pipeline, o tempo de escrita para a tabela aparentava ser muito elevado, rondando os 15 minutos para um ficheiro de 8MB. Isto deve-se ao facto das tabelas da DynamoDB terem um controlo da taxa de transferência, que caso for demasiado elevada, consumia para além do que o free tier da AWS fornecia, consumindo recursos monetários em elevadas quantidades na nossa experiência.

3. Na implementação das queries para a DynamoDB, reparamos numa limitação dos resultados obtidos com o despoletar de uma query. Isto deve-se ao facto da DynamoDB limitar os resultados de cada query a "páginas"de 1MB, sendo que uma página é um conjuntos das entradas filtradas pela query. Tivemos então que lidar com a paginação, que provou ser pouco eficiente, baseando-se na utilização da última entrada da

página anterior, e despolentando uma nova query a partir dessa entrada, logo necessitando de *TAMANHO_DATASET*/1MB queries.

4. As queries da DynamoDB são de 2 tipos: **Scan** e **Query**. Inicialmente usamos Scans para alguns serviços disponibilizados, porém, estes eram muito ineficientes em comparação com as queries dado que necessitavam de percorrer a tabela inteira. Recriamos então a tabela, de modo a fornecer uma Primary Key comum a todas as entradas e com uma coluna **event_id** que era denominada como Sorting Key, distinguindo todas as entradas. Isto possibilitou a utilização de queries da DynamoDB que aumentou o desempenho para certos serviços.

5. Porém, existiam outros serviços (como o caso do */api/events/ratio*) que necessitavam de percorrer a base de dados na sua totalidade para mostrar as estatísticas pretendidas, pelo que não se verificou um aumento de desempenho significante entre a utilização de Queries ou Scans. Estas queries demoravam em média 10 a 15 minutos por cada pedido realizado, não sendo um tempo de execução aceitável.

6. Foi concluído que a DynamoDB não era uma base de dados adequada para a realização de análise de dados. Daí, mudamos a implementação para uma imagem Docker de Mongo.

6.2 Web Server

O Web Server foi inicialmente implementado em NodeJS com um router Koa. Porém a documentação relativa ao router era escassa em comparação a outras alternativas, razão pela qual mudamos para um router Express que se encontrava com melhor documentação.

As queries para a base de dados na DynamoDB foram implementadas em NodeJS, onde foi necessário tratar da paginação. A documentação relativa à paginação das queries encontrava-se disponível sendo a implementação relativamente simples, excluindo as tentativas de melhorar o desempenho.

Mudando a implementação da base de dados para uma de MongoDB, deparamo-nos com pequenos problemas de implementação das queries, mas que foram eventualmente resolvidos, observando-se um aumento de desempenho a várias ordens de magnitude em comparação com a implementação na DynamoDB (queries que demoravam 10-15 minutos na Dynamo não excediam os 20 segundos no mongo).

6.3 Load Balancer

Para o balanceamento da carga no deployment em Kubernetes, utilizamos um controlador de Ingress, o NGINX. Este controlador encaminha os vários pedidos para os serviços correspondentes. No nosso caso concretamente, os

pedidos com o URL */api/events** e */api/products** são encaminhados para os seus respetivos serviços que estão deployed nas Kubernetes. Os serviços por sua vez fazem o balanceamento de carga automaticamente pelos vários pods por nós lançados que contém a implementação do Web Server.

7 Lançamento em Kubernetes

O scripts de deployment do sistema foram separados em 4 devido à necessidade dos clusters e node groups estarem ativos, antes de proceder aos próximos passos. Existe também uma secção de edição de ficheiro manual, o que fez a quebra entre o terceiro e o quarto script.

Antes da execução dos scripts é necessário verificar a existência dos seguintes repositórios, roles e policies e caso existam, precisam de ser **eliminados**:

1. Repositórios com nomes *products* e *events*
2. AWS Role com nome eksServiceRole
3. AWS Policy com nome ALBIngressControllerIAMPolicyEcommerce

Para a execução dos scripts são necessárias as seguintes ferramentas:

- AWS CLI
- eksctl
- kubectl

A região a escolher para o lançamento poderá ser qualquer um, porém recomendamos a região eu-west-1. Esta região terá de ser a mesma nos argumentos de todos os scripts que requeiram a mesma.

Com a utilização do Ingress, reparamos que existe a possibilidade deste poder usar VPCs antigas, para prevenir que tal ocorra, sugerimos apagar Target Groups que não estejam a ser utilizados.

7.1 deploy1.sh

O primeiro script de deployment recebe 3 argumentos na seguinte ordem:

1. A região onde a Stack e o Cluster vão ser lançados (ex.: **eu-west-1**)
2. O nome da Stack que terá de ser único (nenhuma outra Stack na CloudFormation da conta pessoal poderá ter o mesmo nome) para o script funcionar corretamente (ex.: **ecommerce-stack**)
3. O nome do Cluster que também terá de ser único (ex.: **ecommerce-cluster**)

4. Exemplo de execução: *./deploy1.sh eu-west-1 e-stack e-cluster*

A criação da stack e do cluster irá demorar cerca de 10 a 20 minutos até ficarem ativos, após o qual poderemos proceder à execução do segundo script. O estado do script pode ser verificado com o seguinte comando:

- *aws eks describe-cluster --name CLUSTER_NAME* , mudando CLUSTER_NAME para o nome do cluster dado nos argumentos

A execução do segundo script só deve ser feita quando o estado do cluster estiver em **ACTIVE**.

7.2 deploy2.sh

O segundo script recebe os mesmos argumentos que o primeiro, todos na mesma ordem. Neste script vão ser criados os node groups e o pull das imagens dos serviços.

Para realizar o pull, irá ser pedido para inserir os credenciais IAM que dão acesso aos repositórios por nós criados. Estes credenciais encontram-se no ficheiro **credenciais.txt** juntamente com a região onde os repositórios se encontram.

No fim da execução do script, é necessário verificar o estado dos node groups antes de proceder ao próximo script. O estado pode ser verificado com o seguinte comando:

- *aws eks describe-nodegroup --cluster-name CLUSTER_NAME --nodegroup STACK_NAME* , mudando CLUSTER_NAME e STACK_NAME para os respetivos nomes dados nos argumentos

No final irá ser pedido para introduzir os credenciais de IAM para a conta de AWS pessoal de modo a poder criar os repositórios e fazer push das imagens no script seguinte. Após o estado do node group passar para **ACTIVE** podemos proceder À execução do terceiro script.

7.3 deploy3.sh

O terceiro script recebe 2 argumentos na seguinte ordem:

- A região que deverá ser idêntica às inseridas nos scripts anteriores
- O nome do cluster que também deverá ser o mesmo

Para a correto funcionamento deste script, salienta-se a necessidade de apagar os repositórios, roles e policies anteriormente referidos.

Após a criação dos repositórios e o push das imagens para os mesmos, será pedido para mudar os credenciais IAM para os que se encontram no

ficheiro **credenciais_mongo.txt** para realizar o pull da imagem da base de dados.

Depois de realizar o pull irá ser pedido para mudar os credenciais de novo para a conta pessoal de AWS para a criação do repositório da base de dados e o push da imagem.

7.4 deploy4.sh

Antes da execução do 4 ficheiro de deployment, as seguintes mudanças terão de ser realizadas nos ficheiros **ingress/ingress-rbac.yaml** e **ingress/alb-ingress-controller.yaml**.

7.4.1 ingress/ingress-rbac.yaml

Neste ficheiro é necessário inserir manualmente os dados relativos ao role criado no ficheiro na seguinte linha:

- **eks.amazonaws.com/role-arn:**

Para aceder ao valor do ARN do role criado, basta inserir na linha de comandos as seguintes instruções:

- **aws iam get-role --role-name eksServiceRole**

7.4.2 ingress/alb-ingress-controller.yaml

Neste ficheiro é necessário inserir a informação relativa ao cluster criado para que o Ingress utilize os recursos corretos. Para tal é necessário alterar no ficheiro, na linha **- --cluster-name=CLUSTER_NAME**, alterando **CLUSTER_NAME** pelo nome do cluster dado nos argumentos dos scripts anteriores.

Após a execução destes 2 passos, executa-se o quarto e último script, **deploy4.sh** sem parâmetros de entrada.

7.5 Acesso aos serviços

Após a aplicação dos ficheiros yaml do Ingress, é possível aceder ao sistema executando o comando:

- **kubectl get ingress**

Deste comando retira-se o endereço que se encontra em *Address* e adiciona-se os paths para os serviços disponibilizados (ex.: <address>/api/products/listCategories).

7.6 Desconstrução do deployment

Para remover a montagem do sistema, seguem-se os seguintes passos:

1. Na consola de EKS, dependendo da região escolhida, selecionar o cluster e apagar o node group correspondente. Após dos node groups terem sido eliminados, proceder com a eliminação do cluster.

2. Na consola de EC2, eliminar os load balancers criados. Na mesma consola navegar para os target groups e apagar aqueles que estejam relacionados com o VPC criado.

3. Na consola de VPC, eliminar as NATs criadas. Após a eliminação, navegar para a secção dos VPCs e eliminar as VPCs correspondentes que contém o nome dado à stack.

4. Na consola de ECR, eliminar os seguintes repositórios: *database*, *events* e *products*.

5. Na consola de IAM Policies, eliminar a policy com o seguinte nome: **ALBIngressControllerIAMPolicyEcommerce**. Após a eliminação, navegar para a secção dos Roles, e apagar o role com o seguinte nome: **eksServiceRole**.

6. Na consola de CloudFormation, eliminar a stack criada com o nome dado nos scripts anteriores.