Relatório Preliminar

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

28 de Abril de 2020

Indíce

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

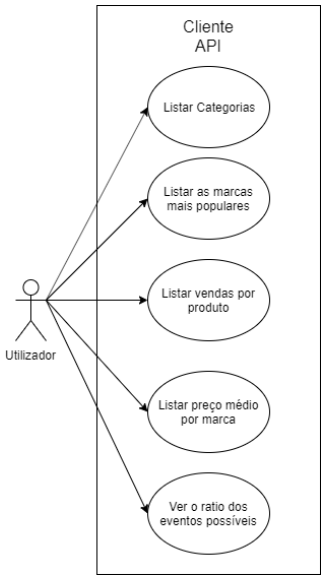

Figura 1: Diagrama de casos de uso

3 Requisitos

Requisitos 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 1: Requisitos Funcionais

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 2: Requisitos Não Funcionais

4 Arquitetura da aplicação

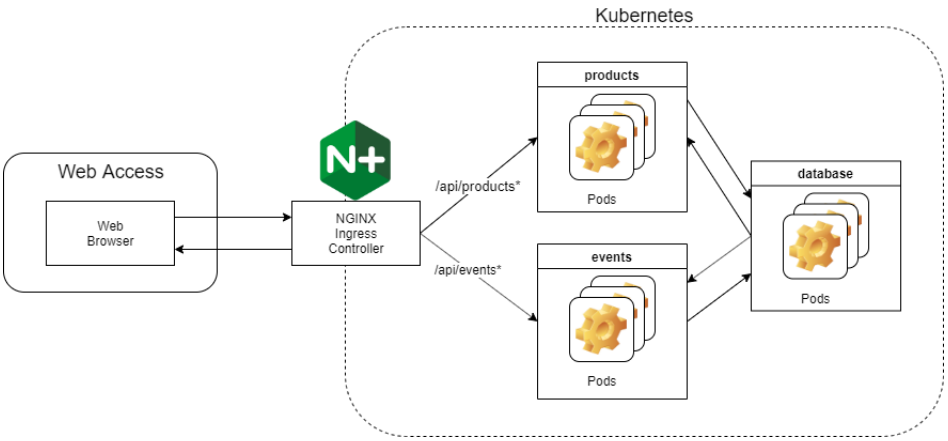

Figura 2: Arquitetura da aplicação

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 utilizando o Ingress que distribui os pedidos HTTP recebidos dos clientes browser e que reencaminha para os respetivos serviços.

4.2 Servidores NodeJS

Os servidores irão 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 numa 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.

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.

Ao apercebemo-nos de que DynamoDB não seria a melhor escolha para a base de dados tendo em conta os serviços definidos e o dataset usado, decidimos mudar para uma base de dados Mongo em que manteríamos uma base de dados NoSQL, já que nos permite uma escalabilidade horizontal que se adequa aos serviços fornecidos.

Verificamos que utilizar o serviço fornecido pelo MongoDB Atlas tinha custos dos quais não poderíamos suportar, optámos então por utilizar a imagem Mongo disponibilizada online. Mas isto não seria suficiente, pois queríamos que o processo fosse simples, tal como montar qualquer outro container, mas para isso queríamos que a imagem contivesse os dados do dataset para popular a base de dados com o respetivo ficheiro.

Para conseguir obter esta simplicidade, recorremos a um Dockerfile que usa como imagem base a imagem de Mongo mas para tal necessitaríamos de acrescentar um novo conjunto de operações. Portanto adicionamos a funcionalidade de, ao dar build, copiar o ficheiro CSV com os dados do dataset para a imagem a construir. De seguida necessitámos de definir um shell script que fosse executado depois de correr a imagem. Definimos um simples script que trataria de executar a base de dados e consequentemente importasse os dados contidos no ficheiro CSV presente na imagem, sendo o ficheiro apagado no fim do import de modo a não ocupar espaço de armazenamento. Ao definir o seguinte script, o mesmo foi acrescentado como entrypoint no Dockerfile para que fosse executado assim que a imagem fosse instalada. Foi também acrescentada a cópia do script para o container tal como foi feito anteriormente com o ficheiro CSV.

Depois de definir toda esta estrutura construimos a imagem e fizemos push para um repositório da AWS para que ficasse acessível. Assim conseguimos obter uma imagem de uma Base de Dados Mongo que é facilmente montada em qualquer máquina devido à utilização do Docker. Bastando puxar a imagem do repositório e correr como qualquer outra imagem.

Após de realizar os passos referidos tivemos de converter todas as queries anteriormente definidas para cada serviço. A principal vantagem nesta nova base de dados seria o uso do distinct, sendo que o Mongo apesar de tal como a Dynamo ser também NoSQL, permite uma mais fácil implementação de queries mais complexas, melhorando consideravelmente o desempenho dos serviços disponibilizados.

Deparamo-nos então com um problema semelhante à paginação da Dynamo mas agora em Mongo, neste caso uma query ao devolver demasiados resultados devolve um cursor que tem de ser iterado para devolver passo a passo, todos os dados resultantes da query. Inicialmente deparamo-nos com alguma dificuldade na utilização do cursor não conseguindo encontrar documentação muito clara e completa. Depois de alguma insistência da parte do grupo, conseguimos através de várias tentativas, compreender como iterar sobre um cursor e assim tirar proveito das queries.

Após ultrapassada esta dificuldade, todas as queries foram facilmente implementadas, sendo que foi usada principalmente a operação de aggregate que permite uma maior complexidade das queries permitindo assim filtrar ao máximo a resposta da base de dados, reduzindo a latência de resposta.

Com esta implementação das queries de forma mais detalhada, conseguimos um aumento de performance muito significativo relativamente à Dynamo, provando que a mudança de Base de dados apesar de ter sido demoroso para efetuar, foi a escolha certa a fazer tornando os nossos serviços mais eficientes, passando a ter respostas da parte da base de dados na ordem de poucos segundos ao contrário dos anterior minutos necessários para devolver uma resposta.

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.

Na implementação do Ingress controller deparamo-nos com problemas no deployment do Ingress em si, pois este não mostrava o endereço necessário para realizar o load balancing, não conseguimos antes da primeira entrega encontrar o problema no nosso deployment, pelo que tivemos que fazer expose dos 2 serviços individualmente para ter um sistema que responda a queries.

Já na segunda entrega fizemos uma nova implementação do controlador Ingress, desta vez dando as permissões necessárias e acrescentado os dados corretos no ficheiros YAML relacionados com o Ingress, resultand num deployment correto do load balancer.

7 Lançamento em Kubernetes

Para realizar o deployment do sistema é usado apenas um script de forma a tornar todo o processo mais prático, automatizando todo o deployment.

Antes da execução do script 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 o prefixo eksctl-cluster
3. AWS Policy com nome ALBIngressControllerIAMPolicyEcommerce

Para a execução do script é necessário ter a ferramenta eksctl.

A região a escolher para o lançamento poderá ser qualquer um, porém recomendamos a região eu-west-1.

Com a utilização do Ingress, reparámos 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 deploy.sh

O script de deployment recebe 2 argumentos na seguinte ordem:

1. A região onde a Stack e o Cluster vão ser lançados (ex.: **eu-west-1**)

2. O nome do Cluster que também terá de ser único (ex.: **ecommerce-cluster**)

3. Exemplo de execução: *./deploy.sh eu-west-1 e-cluster*

A execução inicial do script irá demorar cerca de 10 a 15 minutos até que passe para a fase de preparar as imagens a correr no cluster. Após esta fase inicial de criação do cluster no EKS, inicia-se o processo de preparação das imagens que irão ser usadas para ser corridas em containers e que compõem o sistema. Para este processo vai ser necessário que insira as credenciais de IAM que darão acesso aos repositórios por nós criados. Esses credenciais encontram-se no ficheiro **credenciais.txt** juntamente com a região onde os repositórios se encontram. Depois de realizar o pull das imagens, irá ser pedido para introduzir de novo as credenciais IAM mas desta vez para a conta de AWS pessoal de modo a poder criar os repositórios e fazer push das imagens. Depois da seguinte fase todo o processo não irá requerer qualquer input da parte do utilizador até o fim de execução do script.

7.2 Acesso aos serviços

Após a execução do script de deployment, é possível aceder ao sistema executando o comando:

- **kubectl get ingress -n kube-system**

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).

Deparámo-nos com um problema no load balancing na kubernetes, que também foi verificado no caso da implementação com o ELB da AWS na fase anterior à entrega das kubernetes. Executando kubectl expose sobre os deployments events e products, as queries retornam normalmente. Acreditamos que o problema se encontre no load balancer do Ingress.

7.3 Desconstrução do deployment

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

1. Usar o seguinte comando que listará os comandos que irão ser usado nos próximos comandos. **kubectl get svc –all-namespaces**

2. Apagar todos os serviços listados pelo comando anterior com o seguinte comando, **kubectl delete svc <service-name>**

3. 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.

4. Executar o comando **eksctl delete cluster --name=<cluster-name>** que será responsável por desmantelar os componentes criados no início do script.

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

6. Na consola de IAM Policies, eliminar a policy com o seguinte nome: **ALBIngressControllerIAMPolicyEcommerce**. Após a eliminação, navegar para a secção dos Roles, e apagar os roles com o prefixo eksctl-cluster.

8 Cenários de Validação

- Listar Categorias:
    1. Inserir no navegador o URL anteriormente mencionado acrescentado /**api**/**products**/**listCategories**
    2. Após alguns segundos irá aparecer no ecrã o resultado.
- Listar a popularidade das marcas:
    1. Inserir no navegador o URL anteriormente mencionado acrescentado /**api**/**products**/**popularBrands**
    2. Após alguns segundos irá aparecer no ecrã o resultado.
- Listar vendas de cada marca:
    1. Inserir no navegador o URL anteriormente mencionado acrescentado /**api**/**products**/**salePrice**
    2. Após alguns segundos irá aparecer no ecrã o resultado.
- Listar o preço médio de uma marca:
    1. Inserir o url mencionado posteriormente adicionado "/<Marca>", aonde <Marca> representa a marca que o utilizador quer analisar.
    2. Após alguns segundos irá aparecer no ecrã o resultado.
- Ver o ratio dos eventos possíveis:
    1. Inserir no navegador o URL anteriormente mencionado acrescentado /**api**/**events**/**ratio**
    2. Após alguns segundos irá aparecer no ecrã o resultado.
- Portabilidade:

1. Escolher 2 diferentes fornecedores de serviço cloud
2. Implementar os serviços criados, nas plataformas diferentes utilizando os docker containers fornecidos.
3. Verficar que ambos os serviços se encontram funcionais

Os docker containers tornam os sistema muito portável dado que é uma ferramenta comum entre os vários fornecedores de serviços cloud. Daí o deployment dos serviços é facilitada caso seja necessário a mudança de fornecedor, sendo apenas necessário tratar dos aspetos técnicos em termos de balanceamento de carga e quantidade de instâncias ativas nos serviços utilizando as ferramentas de cada fornecedor.

- Elasticidade:

1. Verfique o numero de instâncias virtuais ativas
2. Assumindo uma baixa carga de sistema, sobrecarregue o serviço com pedidos
3. Visualize o numero de maquinas virtuais ativas aumentar de forma a poder satisfazer os diferentes pedidos sem afetar a qualidade do serviço

9 Discussão e Conclusões

Durante o desenvolvimento do projeto foram encontrados bastantes obstáculos, seja na implementação dos serviços, na escolhe e consequente aplicação da base de dados ou na automatização dos scripts de implementação dos serviços desenvolvidos. Todos estes problemas resultaram em muitas horas de trabalho extra por parte do grupo, mas serviram também de aprendizagem, seja em como trabalhar com fornecedores de serviços na nuvem, ou como desenvolver scripts de automatização de implementação. Apesar de todos os problemas, o grupo foi capaz de os solucionar de forma eficiente como mencionados na secção da Arquitetura técnica.

O DynamoDB aparenta ser uma boa solução para fazer hosting de bases de dados onde a procura seja limitada a um conjunto de valores limitados. Isto torna-o pouco útil para a nossa implementação, porém, continua a ser uma boa ferramenta para os casos mencionados se a base de dados for de dimensões elevadas, sendo difícil fazer o hosting local da mesma.

Os serviços da cloud no geral, se utilizadas de forma correta, aparentam ser poderosas em sentido computacional, tendo uma grande variedade de ferramentas que possibilita a implementação de aplicações, serviços ou outra qualquer funcionalidade.

Na nossa experiência pessoal, a documentação da AWS foi mais escassa em comparação ao GCP. Dado a escolha entre ambas recomendaríamos fortemente a utilização do GCP para futuras implementações.