Betting House Web App

Miguel Brito (pg38419)
Luís Dias (pg38418)

Junho 2019

Universidade do Minho
Escola de Engenharia

Engenharia Web
MEI – Universidade do Minho

1 Objetivo

Pretende-se desenvolver uma aplicação *web* de uma plataforma de apostas. Esta deverá possibilitar consultar eventos, realizar apostas, tal como permitir a um utilizador com privilégios especiais fazer a gestão de eventos. Deverá ainda cumprir algumas exigências no que diz respeito à responsividade, escalabilidade e acessibilidade.

2 Requisitos

2.1 Requisitos Funcionais

- Um apostador deverá registar-se no sistema através de: email, nome e creditar uma quantia para apostas;
- Ao aceder à plataforma, o apostador registado deverá ter disponível uma lista de eventos sobre os quais poderá apostar;
- O apostador ao efetuar uma aposta deverá indicar o resultado pretendido e o valor a apostar;
- O serviço deverá manter uma lista das apostas realizadas por cada apostador;
- Um evento suscetível de aposta pode encontrar-se no estado Aberto (disponível para uma aposta) ou Fechado (indisponível para aposta). Ao passar de Aberto para Fechado (o resultado do evento é conhecido), todos os apostadores devem ser notificados do resultado das suas apostas para o evento e, se for aplicável, o valor ganho;
- Para determinar o valor ganho numa aposta deverá ser definido para cada evento as *odds* para os possíveis resultados e sobre essas *odds* será calculado o valor ganho numa aposta;
- O administrador poderá gerir as informações da aplicação tais como eventos e apostas através de uma interface independente;
- Os utilizadores *premium*, que pagam por uma subscrição, terão acesso a eventos restritos.

2.2 Requisitos Não Funcionais

- Responsividade - a aplicação deve garantir que novas apostas não são efetuadas após um evento terminar.
- Escalabilidade - a aplicação poderá receber picos de carga elevados em eventos como a Liga dos Campeões, devendo ser escalável para responder aos mesmos.
- Interface - a aplicação deve seguir o nível A da WCAG(Web Content Ac- cessibility Guidelines).

3 Arquitetura

A arquitetura inicialmente proposta tratava-se de uma divisão em 3 camadas: *frontend*, *backend* e camada de dados, onde o *backend* se definia como um monólito, um único processo que expunha toda a lógica de negócio da aplicação

No entanto, dados os requisitos não funcionais relativos à escalabilidade optou-se pela transformação de uma arquitetura monolítica, numa arquitetura de micro-serviços onde poderá ser possível separar as preocupações da aplicação e escalar os serviços que mais necessitam de acordo com a carga a que estão submetidos.

A arquitetura de micro-serviços para além de permitir lidar com a escalabilidade permite também uma maior disponibilidade uma vez que caso um micro-serviço falhe os outros poderão não ser afetados caso não dependam diretamente eles e maior agilidade de desenvolvimento dado haver uma maior segregação de componentes. Relativamente ao *API Layer* que oferece uma interface de comunicação com a *API REST* optou-se pelo uso de um *Gateway* onde será feita a validação dos pedidos pela autenticação e se definirá que pedidos estão disponíveis para o exterior sendo também tratado do *load balancing* para as diferentes instâncias do mesmo serviço.

Optou-se por segregar a aplicação nos seguintes 4 micro-serviços:

3.1 Listar Eventos Disponíveis

Dado a maioria dos acessos à aplicação ser na sua maioria para consulta de eventos e só depois uma fração desses utilizadores passará à realização de apostas, este será um serviço bastante requisitado. Como tal, para evitar que escritas nos eventos e os consecutivos *locks* a essa base de dados atrasem o processo, este serviço terá armazenado uma lista de todos os eventos atualmente disponíveis que serão consultados na sua maioria para leituras havendo apenas escritas periódicas para a atualização dos eventos a decorrer.

3.2 Gestão de Apostas

Este micro-serviço será responsável por lidar com a realização das apostas, a sua validação após o fecho de um evento e disponibilização de um histórico de apostas por utilizador. Inicialmente ponderou-se a separação da realização de apostas com a sua validação, mas optou-se por mantê-lo num únic omicro-serviço dada a quantidade elevada de interação que haveria entre os micro-serviços.

3.3 Gestão de Eventos

Este micro-serviço será responsável pela gestão de eventos, desde a sua criação e edição ao seu fecho e adição de estatísticas.

3.4 Gestão de Utilizadores

Este micro-serviço permitirá a autenticação com os utilizadores em interação com o Gateway e toda a gestão relativa aos mesmos. Segue-se o diagrama de componentes referente à arquitetura proposta.

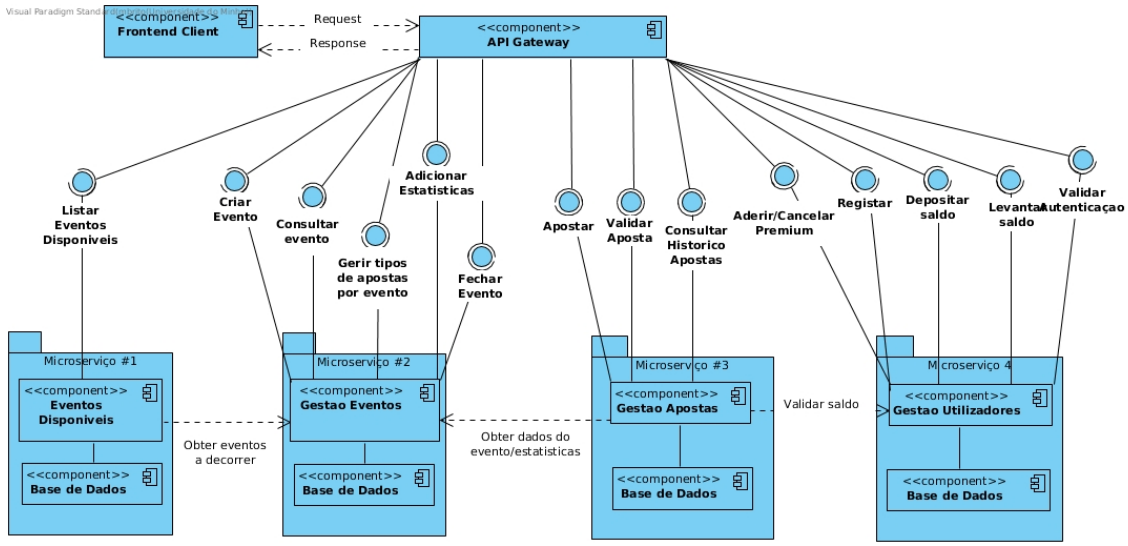

Figura 1: Arquitetura de micro-serviços

4 Infraestrutura de Suporte à Arquitetura

Estando definida uma arquitetura de micro-serviços é agora necessário criar uma infraestrutura que lhe dê suporte, que seja consistente e independente do ambiente sob o qual é executada e permita *deploys* expectáveis.

Neste sentido, fomos expostos ao *Vagrant* durante as sessões realizadas nas aulas, no entanto, pela pesquisa realizada e opinião dos próprios colaboradores da *Vagrant* [1], o facto de o *Docker* ser mais leve por ser apenas um gestor de *containers* e ser mais flexível para micro-serviços levou-nos a escolhe-lo como base para a definição da infraestrutura. Existia há partida alguma tendência para o *Docker*, dada a sua enorme popularidade e existir curiosidade em perceber como funciona.

4.1 *Docker*

Esta tecnologia fornece uma camada de abstração e automação de virtualização ao nível do sistema operativo, recorrendo a características do *kernel Linux* como é o caso do *cgroups*.

Recorrendo à virtualização o *docker* permite "empacotar"aplicações num local isolado denominados de *containers*. Sobre estes *containers* o programador define um conjunto de instruções a serem executadas, armazenadas num ficheiro denominado *Dockerfile*.

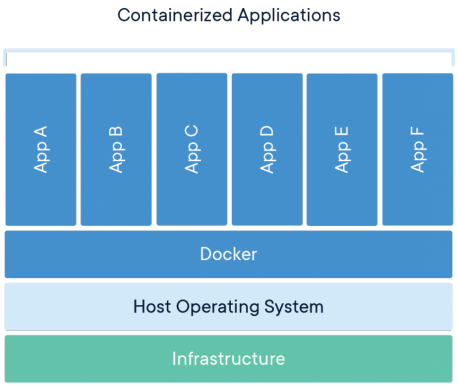

Figura 2: Arquitetura Docker

Segue-se um exemplo do *Dockerfile* para execução de uma aplicação em *node* que foi utilizado múltiplas vezes para a criação do *container* da lógica de negócio de cada micro-serviço. Neste *Dockerfile* começa-se for definir uma imagem base, sobre a qual as operações serão executadas. De seguida define-se o *WORKDIR* para onde será copiado o projeto e a partir daí será executado. Realiza-se a cópia do *package.json* e a respetiva instalação das independências necessárias e por último copia-se o projeto para o *WORKDIR* e executa-se o comando que expõe o serviço.

```
FROM node:8

WORKDIR /usr/src/app

COPY package*.json ./

RUN npm install
RUN npm install nodemon -g

COPY . ./

# Change to npm start for deploy
CMD ["nodemon"]
```

Feita a *dockerização* dos vários componentes é agora necessário realizar a integração e comunicação entre eles, para tal recorreu-se ao *Docker-compose*.

4.2 *Docker-Compose*

Docker Compose é uma ferramenta que permite definir e gerir vários *containers* de *Docker*. Esta definição é feita recorrendo a um ficheiro *YAML*. ***Apêndice A***

Segue-se um exemplo de definição de um micro-serviço no *docker-compose*, sendo cada micro-serviço sempre composto por um *container* para a base de dados e outro para a aplicação de lógica de negócio.

Define-se um *container* para a base de dados, neste caso *MySQL*, definindo variáveis de ambiente que pela imagem utilizada criam automaticamente uma base de dados com o respetivo nome e utilizador. Define-se um volume que permite que haja um mapeamento de um diretório no *host* para um diretório no *container* permitindo desta forma que os dados sejam persistidos, caso contrário um *restart* ao *container* resultaria na perda de dados. Segue-se o mapeamento do porto de comunicação, embora este apenas tenha sido utilizado para efeitos de *debug* dado que a comunicação entre serviços é feita pela rede interna definida em *networks*, não havendo interação com o exterior.

Relativamente à definição do container do *Node*, existem algumas semelhanças à definição do *container* anterior, no caso dos volumes, portos e *network*, no entanto este requer outros ajustes. O processo de *build* aqui realizado é feito recorrendo ao respetivo *Dockerfile* e não diretamente a uma imagem. Mais uma vez as variáveis de ambiente permitem definir variáveis necessárias à configuração do processo, neste caso, definição de configurações da base de dados, *secrets* necessários para validação de *JWT*.

O facto do *container* do *node* depender do *container* da base de dados uma vez que necessita de realizar o proceso de *ORM* na sua inicialização exige configuração extra: quer por definir uma dependência o que determina a ordem de execução dos *containers*; quer pela necessidade de recorrer a um *script*, *wait-for-it.sh*, para garantir que a base de dados já se encontra funcional, uma vez que o *depends_on* não garante que o processo disponibilizado já se encontre operacional quando o *container* que cria a dependência é executado.

```
# MS - USER_MANAGER
    db_user_manager:
        image: mysql:5.7.26
        environment:
            MYSQL_DATABASE: db_user_manager
            MYSQL_USER: db_user_manager
```

```
        MYSQL_PASSWORD: db_user_manager
        MYSQL_ROOT_PASSWORD: root-password
    volumes:
        - ~/databases/db_user_manager/data:/var/lib/mysql
    ports:
        - 3300:3306
    networks:
        - network_backend

ms_user_manager:
    build: ./microservices/user-manager
    volumes:
        - ./microservices/user-manager:/usr/src/app
        - /usr/src/app/node_modules
    environment:
        DB_USERNAME: db_user_manager
        DB_PASSWORD: db_user_manager
        DB_NAME: db_user_manager
        DB_HOST: db_user_manager

        FILE_STORE_SECRET_KEY: 'bettingwebapp'
        PASSPORT_SECRET_KEY: 'bettingwebapp'
        JWT_SECRET_KEY: 'bettingwebap'

        MS_EVENTS: 'http://ms_event_manager:3000'
        MS_BETS: 'http://ms_bet_manager:3000'
    ports:
        - 3001:3000
    networks:
        - network_backend
    depends_on:
        - db_user_manager
    restart: always
    command: ["./wait-for-it.sh", "db_user_manager:3306", "--", "nodemon"]
```

4.3 *API Layer - Gateway*

A *API Layer* é um componente essencial na infraestrutura da aplicação dado que servirá como *middleware* para todos os pedidos feitos: quer pelo cliente, quer pela comunicação entre micro-serviços.

Dado o uso de *express* para exposição das rotas *REST* recorreu-se ao *express-gateway* dado ter uma integração facilitada por seguir várias das normas definidas pelo *express*.

Para além de interagir como *middleware* entre os pedidos o *express-gateway* oferece uma variadade de *plugins* que permitem realizar: autenticação, *load balancing proxy*, *logging*, limitação de pedidos, etc.

Há semelhança do *docker* e *docker-compose* toda a configuração e definição desta infraestrutura é feita por um ficheiro de configuração, denominado de *gateway-config.yml*, ***B***.

A configuração de um micro-serviço é dividida em 3 passos principais:

- Definição dos *API Endpoint* - que rotas estarão expostas ao exterior e disponíveis para que *hosts*

```
bet-manager:
    host: '*'
    paths: ['/bet/', '/bet/*', '/bettype', '/bettype/*']
```

- Definição dos *Service Endpoints* - em que *host* está cada um dos micro-serviços

```
bet-manager-service:
    url: ${MS_BET_MANAGER_URL} # Obtido pelas variáveis de ambiente
```

- Definição das *Pipelines* - definição da lógica de funcionamento entre as camadas anteriores e lógica extra, como por exemplo validação da autenticação. É aqui que na aplicação em questão o *jwt* é validado de acordo com o gerado no micro-serviço de autenticação.

```
bet-manager-pipe:
    apiEndpoints: bet-manager
    policies:
      - jwt-db:
        - action:
            secret: "secret"
      - proxy:
        - action:
            serviceEndpoint: bet-manager-service
```

5 Tecnologias

5.1 Frontend

O *React* foi a *framework* selecionada para o desenvolvimento do cliente, a interface, dada a sua popularidade e suporte, o que resulta num vasto repositório de tutoriais e documentação sobre a mesma. É também uma das *frameworks* com mais *packages* disponíveis no *npm* o que permite que haja uma maior reutilização de funcionalidade já implementada e não obrigue à criação de certos componentes de raiz, como é o caso de gestor de rotas, formulários, notificações, etc.

Inicialmente considerou-se o uso de *Vue* por ser conhecido como uma *framework* com boa documentação e que pretende colmatar as falhas do *React* e *Angular*, mas as sessões apresentadas em *React* influenciaram-nos a seguir o *React*.

A nivel de funcionamento, cada componente implementa um método *render* que retorna o que se pretende que seja visualizado. Para definição dos elementos visuais recorre-se a *JSX* (*JavaScipt XML*), uma combinação de *JavaScipt* com *XML*. O uso de *JSX* é opcional mas o seu uso é padrão. Cada componente pode receber dados de outros componentes, dados estes denominados de *props*, pode também manter o seu estado interno que a cada alteração resultará num *trigger* para execução do método de *renderização* do componente. Os componentes são compostos por um grupo de métodos que podem ser implementados para serem ativados ao longo do seu ciclo de vida, ver **Figura 3**.

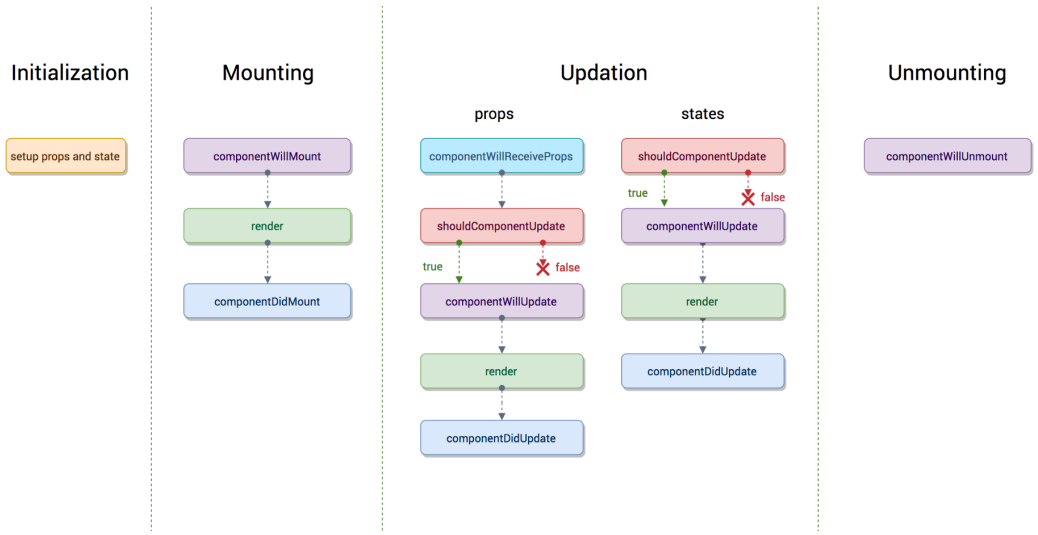

Figura 3: Ciclo de vida de um componente em *React*

Destes métodos destacam-se os seguintes:

- ***shouldComponentUpdate*** - Permite ao desenvolvedor adicionar lógica para definir se o componente deve ser re-renderizado ou não.
- ***componentDidMount*** - Método chamado assim que o componente é criado e associado ao DOM. Normalmente utilizado para fazer fetch de dados de uma API.
- ***componentWillUnmount*** - Método chamado imediatamente antes do componente ser removido do DOM. Pode ser utilizado para forçar a remoção/limpeza de dependências associadas ao componente que não o são automaticamente.

A construção de aplicações modulares é um conceito nuclear do *React*. Esta modularidade pode ser obtida recorrendo à partição bem definida de componentes e permite que recorrendo a uma Virtual DOM apenas os componentes que são alterados e os seus filhos são renderizados de novo, algo que não acontece com o tıpico DOM, onde cada mudança exige uma renderização completa da página o que implica um maior uso de recursos e maior tempo de processamento. Esta diferença pode ser observada na **Figura 4**.

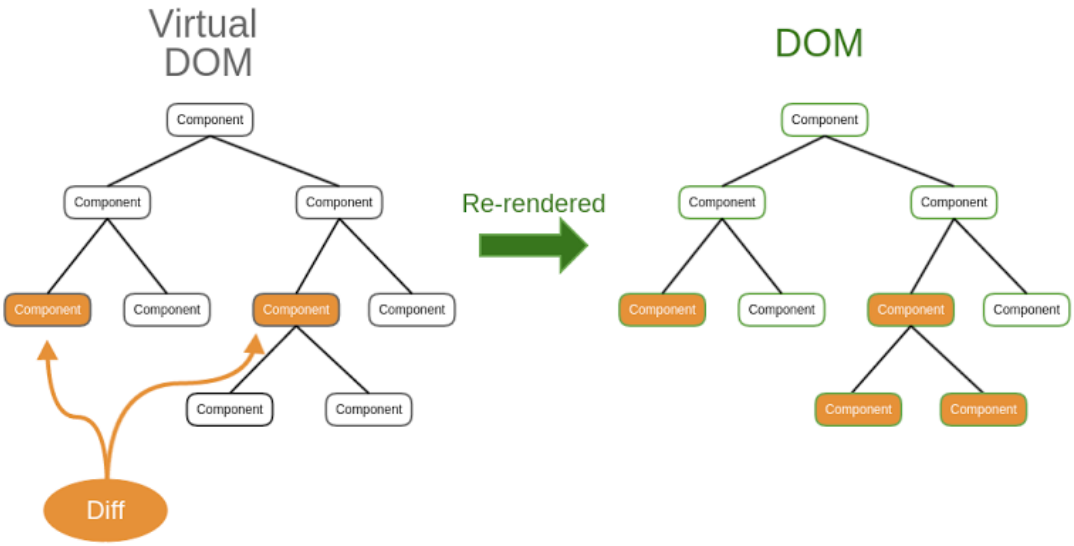

Figura 4: Virtual DOM vs DOM

5.2 Backend

Para o *backend* foi utilizado *node.js*, quer pela sua popularidade quer para mantermos um sistema homogéneo no que diz respeito à linguagem usada, neste caso *JavaScript*. Considerámos o uso da *framework Spring* para *Java*, mas é reconhecida como uma *framework* com uma curva de aprendizagem elevada, por ser bastante robusta.

Uma das principais características que diferencia o *node.js* de um servidor que utiliza linguagens tradicionais, por exemplo PHP, Java, entre outras, é o facto de este realizar a sua execução através de *single-thread*.

Apesar do *node.js* ser *single-thread* consegue tratar dos pedidos de forma concorrente tal como acontece noutros servidores. Isto é possível devido ao facto de serem utilizadas chamadas não-bloqueantes na entrada e saída de dados. Por outras palavras, signfica que as operações de entrada e saída de dados (operações CRUD) são assíncronas e como tal não bloqueiam a *thread* como acontece nos servidores tradicionais em que a thread fica à espera que as operações sejam concluídas para que seja possível continuar a execução.

Através do uso da assincronia o *node.js* possibilita que sejam desenvolvidas aplicações escaláveis. Dito isto, possibilita que o número de pedidos referentes a um serviço possa aumentar de forma uniforme sem que o seu comportamento a nível de performance seja alterado.

A elevada popularidade do *node.js* possibilitou que o NPM (*Node Package Manager*), que é o gerenciador de pacotes do *node.js*, seja um dos maiores repositórios do mundo o que permite que o *node.js* tenha boa flexibilidade e que possa ser adaptado a diversas situações.

5.3 Persistência de Dados

Relativamente à persistência de dados recorreu-se ao *MySQL* dado ser uma base de dados com a qual somos familiar e existirem vários *packages* para *nodejs* que permitem realizar ORM (*Object Relational Mapping*). O uso de ORM cria um nível de abstração entre a aplicação e a base de dados, retirando a necessidade de criar *queries* e mapeamentos à mão para interação com a base de dados. Dos *packages* para ORM encontrados optámos pelo uso do *sequelize* por ser um dos mais populares e ter uma documentação detalhada com exemplos de utilização.

6 Implementação

A implementação foi essencialmente dividida em 4 fases:

- Definição/Implementação da infraestrutura e criação de projeto base:

  Nesta fase tendo em conta a arquitetura criaram-se os *Dockerfiles* e o *docker-compose* para a interação entre componentes, também se criaram projetos base para o *frontend* e para um micro-serviço contendo o *nodejs*, *express*, *sequelize*, etc, que mais tarde foram replicados para cada um dos micro-serviços.

- Implementação dos micro-serviços:

  Deu-se início com a criação dos respetivos modelos por *ORM* recorrendo ao *sequelize* de acordo com a partição da base de dados necessária. Já com os modelos prontos para interação com a base de dados realizou-se a implementação de toda a lógica de negócio da aplicação.

  A implementação das rotas *REST* foi feita num estilo *routes/controllers* em que nos ficheiros *routes* se definem as rotas e tratamento de dados obtidos e dados a enviar, e toda a lógica inerente à aplicação é realizada nos *controllers*.

- Implementação do *frontend*

  Esta fase iniciou-se com a criação de interface ainda sem dados do *backend*, recorrendo a dados tipo. As principais preocupações que aqui foram estabelecidas foram a criação de componentes reduzidos com uma função muito específica para permitiram a sua fácil reutilização em outras secções da página.

  Houve uma preocupação com a responsividade da página para *mobile* e outras resoluções que foi tida em conta pelo uso de *Bootstrap* mas também pela criação de *media queries* para casos mais específicos.

- Integração dos dados com o *frontend*

  Por último, já com praticamente todas as rotas dos micro-serviços disponíveis para "consumo", realizou-se a integração dos dados e lógica com a interação do cliente. Nesta fase houve a preocupação em transmitir ao utilizador o que está a acontecer (mensagens de sucesso/erro, recorrendo a notificações).

Recorremos ao gestor de versões *git* e todo o desenvolvimento e evolução da aplicação se encontra registado em http://github.com/miguelfbrito/EW

Seguem-se as principais interfaces implementadas:

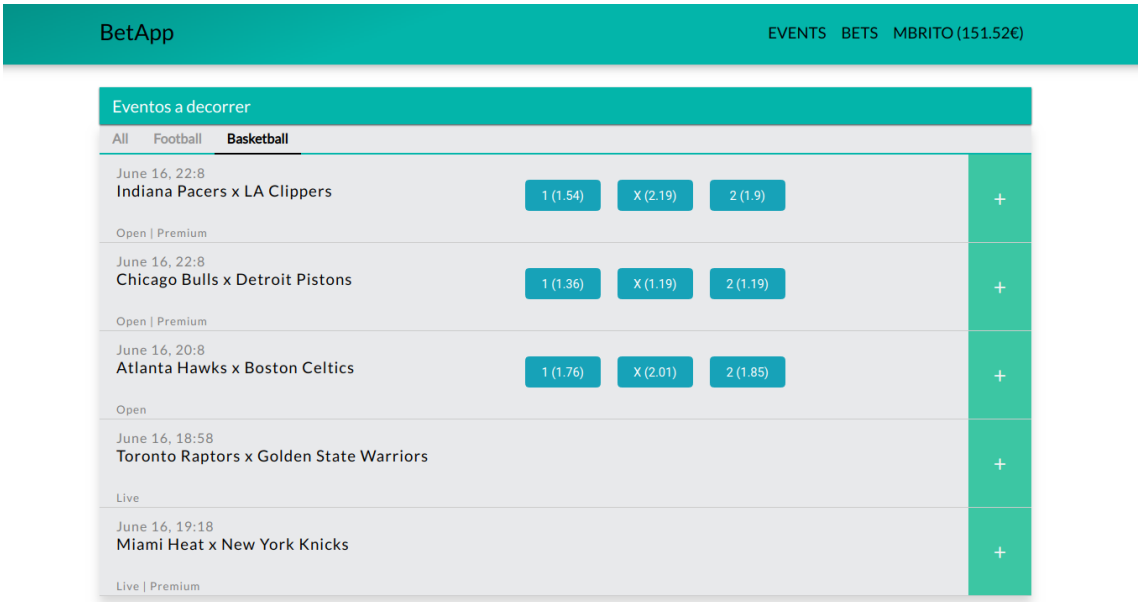

Figura 5: Listagem de eventos disponíveis

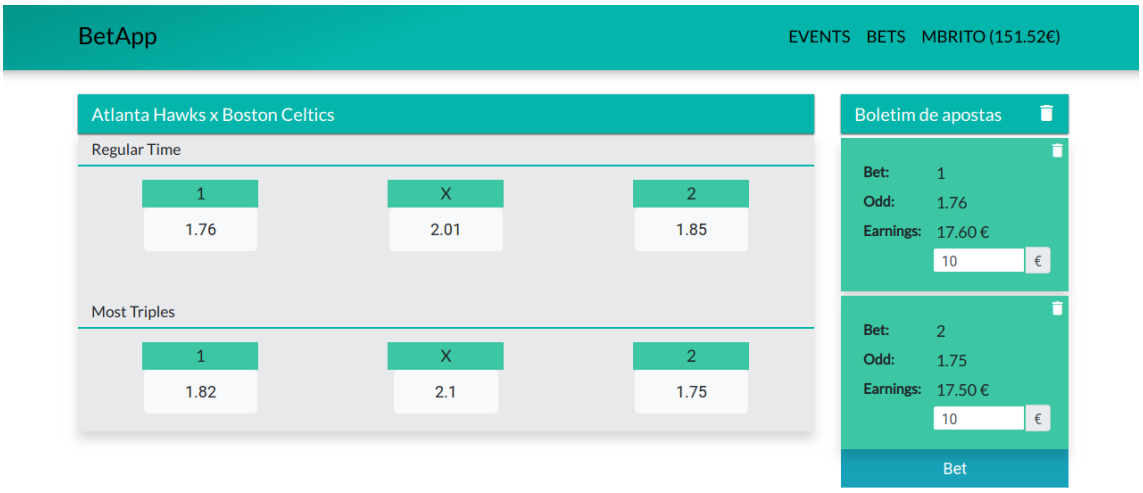

Figura 6: Realização de apostas

Figura 7: Listagem de apostas abertas

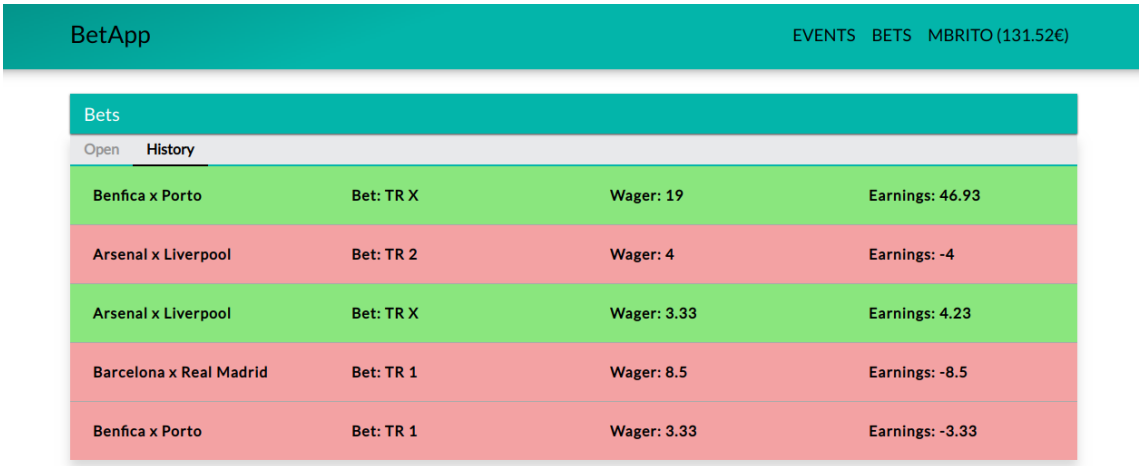

Figura 8: Histórico de apostas

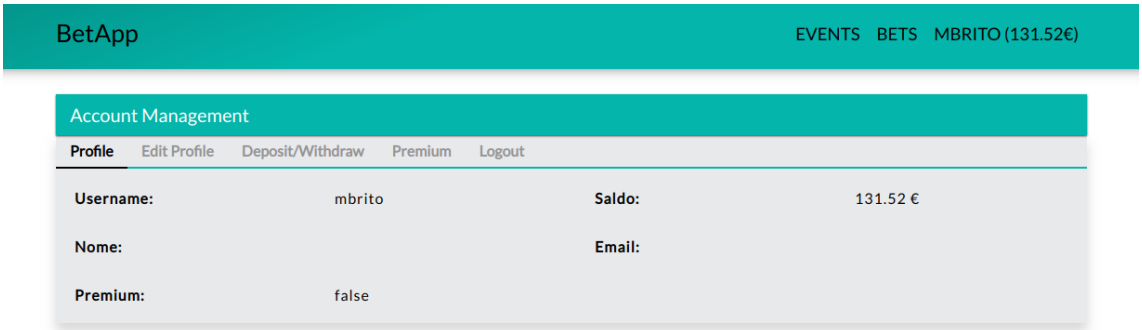

Figura 9: Gestão de Utilizador

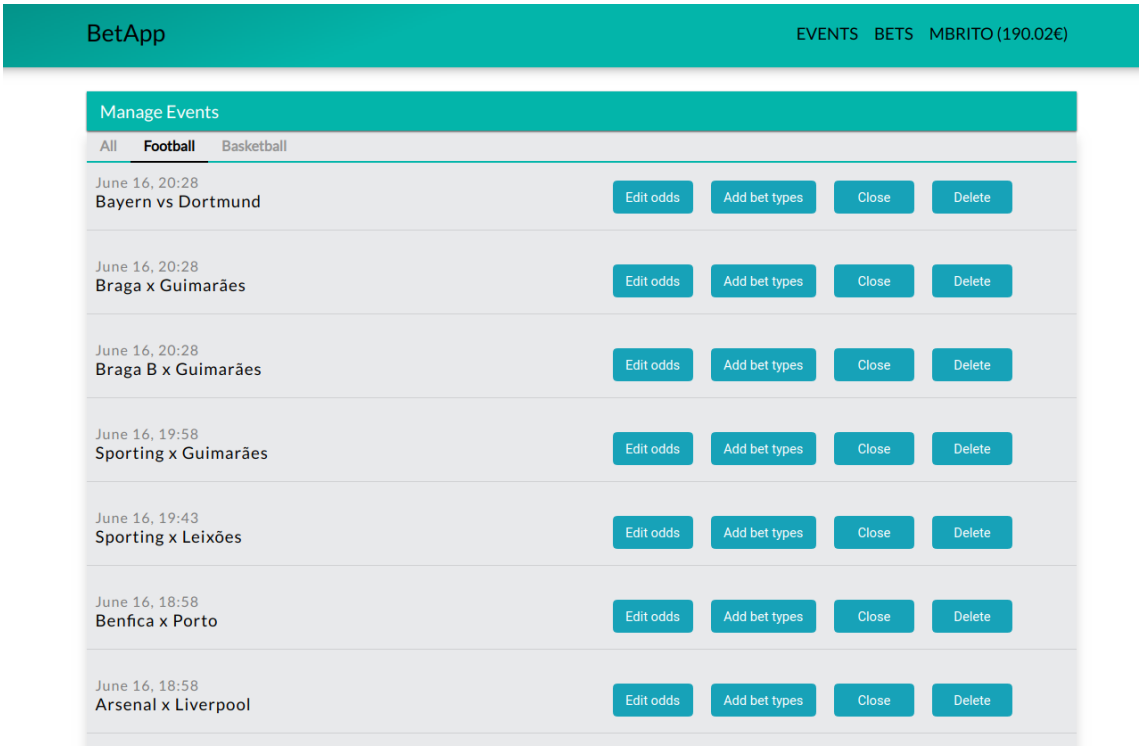

Figura 10: Gestão de Eventos pelo Administrador

7 Conclusão

O facto de termos optado por uma arquitetura de micro-serviços permitiu-nos obter e pôr em prática conhecimentos desta arquitetura que se têm vindo a tornar cada vez mais popular. Popularidade que advêm da necessidade de escalar aplicações e agilizar o processo de desenvolvimento.

Consideramos este conhecimento crucial para o nosso futuro no mercado do trabalho, no entanto, em contexto de trabalho sentimos que apenas lidámos com as desvantagens da arquitetura de micro-serviços, ou seja, maior investimento inicial necessário a definir a infraestrutura, em especial por ser a primeira vez que trabalhamos com tecnologias deste género, e necessidade de criação de uma camada extra de abstração para lidar com a interação entre micro-serviços.

Este investimento feito na arquitetura limitou de certa forma o tempo disponível para outras áreas da aplicação, as quais gostaríamos de melhorar.

Como pontos principais retirados da implementação do *frontend* fica a quase obrigatoriedade da criação de componentes reduzidos com objetivos bem definidos permitindo assim a sua reutilização e código mais limpo. Um caso particular no desenvolvimento foi o componente *Event* que é reutilizado em várias listagens em secções diferentes.

Como trabalho futuro seria interessante adicionar testes unitários para integração com o *Jenkins* uma vez que só realizar um *pull* do repositório e executar o *docker-compose* não tem grande interesse do ponto de vista do processo de integração contínua.

Appendices

A docker-compose.yml

```
version: '3.3'
services:

# Gateway
    gateway:
        build: ./gateway
        volumes:
            - ./gateway:/usr/src/app
            - /usr/src/app/node_modules
        environment:
            MS_USER_MANAGER_URL: 'http://ms_user_manager:3000'
            MS_BET_MANAGER_URL: 'http://ms_bet_manager:3000'
            MS_EVENT_MANAGER_URL: 'http://ms_event_manager:3000'
            MS_LIST_EVENTS_URL: 'http://ms_list_events:3000'
        ports:
            - 8081:8080
        networks:
            - network_backend
            - network_frontend
        depends_on:
            - ms_user_manager
            - ms_bet_manager
            - ms_event_manager
        restart: always

# MS - LIST EVENTS
    db_list_events:
        image: mysql:5.7.26
        environment:
            MYSQL_DATABASE: db_list_events
            MYSQL_USER: db_list_events
            MYSQL_PASSWORD: db_list_events
            MYSQL_ROOT_PASSWORD: root-password
        volumes:
            - ~/databases/db_list_events/data:/var/lib/mysql
        ports:
            - 3304:3306
        networks:
            - network_backend

    ms_list_events:
        build: ./microservices/list-events
        volumes:
            - ./microservices/list-events:/usr/src/app
            - /usr/src/app/node_modules
        environment:
            DB_USERNAME: db_list_events
            DB_PASSWORD: db_list_events
            DB_NAME: db_list_events
            DB_HOST: db_list_events
```

```
            MS_EVENTS: 'http://ms_event_manager:3000'
            MS_BETS: 'http://ms_bet_manager:3000'
        ports:
            - 3004:3000
        networks:
            - network_backend
        depends_on:
            - db_list_events
        restart: always
        command: ["./wait-for-it.sh", "db_list_events:3306", "--", "nodemon"]

# MS - USER_MANAGER
    db_user_manager:
        image: mysql:5.7.26
        environment:
            MYSQL_DATABASE: db_user_manager
            MYSQL_USER: db_user_manager
            MYSQL_PASSWORD: db_user_manager
            MYSQL_ROOT_PASSWORD: root-password
        volumes:
            - ~/databases/db_user_manager/data:/var/lib/mysql
        ports:
            - 3300:3306
        networks:
            - network_backend

    ms_user_manager:
        build: ./microservices/user-manager
        volumes:
            - ./microservices/user-manager:/usr/src/app
            - /usr/src/app/node_modules
        environment:
            DB_USERNAME: db_user_manager
            DB_PASSWORD: db_user_manager
            DB_NAME: db_user_manager
            DB_HOST: db_user_manager

            FILE_STORE_SECRET_KEY: 'bettingwebapp'
            PASSPORT_SECRET_KEY: 'bettingwebapp'
            JWT_SECRET_KEY: 'bettingwebap'

            MS_EVENTS: 'http://ms_event_manager:3000'
            MS_BETS: 'http://ms_bet_manager:3000'
        ports:
            - 3001:3000
        networks:
            - network_backend
        depends_on:
            - db_user_manager
        restart: always
        command: ["./wait-for-it.sh", "db_user_manager:3306", "--", "nodemon"]

# MS - BET-MANAGER
    db_bet_manager:
        image: mysql:5.7.26
        environment:
            MYSQL_DATABASE: db_bet_manager
```

```
            MYSQL_USER: db_bet_manager
            MYSQL_PASSWORD: db_bet_manager
            MYSQL_ROOT_PASSWORD: root-password
        volumes:
            - ~/databases/db_bet_manager/data:/var/lib/mysql
        ports:
            - 3301:3306
        networks:
            - network_backend

    ms_bet_manager:
        build: ./microservices/bet-manager
        volumes:
            - ./microservices/bet-manager:/usr/src/app
            - /usr/src/app/node_modules
        environment:
            DB_USERNAME: db_bet_manager
            DB_PASSWORD: db_bet_manager
            DB_NAME: db_bet_manager
            DB_HOST: db_bet_manager

            MS_USERS: 'http://ms_user_manager:3000'
            MS_EVENTS: 'http://ms_event_manager:3000'

        ports:
            - 3002:3000
        networks:
            - network_backend
        depends_on:
            - db_bet_manager
        restart: always
        command: ["./wait-for-it.sh", "db_bet_manager:3306",  "--", "nodemon"]

# MS - EVENT-MANAGER
    db_event_manager:
        image: mysql:5.7.26
        environment:
            MYSQL_DATABASE: db_event_manager
            MYSQL_USER: db_event_manager
            MYSQL_PASSWORD: db_event_manager
            MYSQL_ROOT_PASSWORD: root-password
        volumes:
            - ~/databases/db_event_manager/data:/var/lib/mysql
        ports:
            - 3302:3306
        networks:
            - network_backend

    ms_event_manager:
        build: ./microservices/event-manager
        volumes:
            - ./microservices/event-manager:/usr/src/app
            - /usr/src/app/node_modules
        environment:
            DB_USERNAME: db_event_manager
            DB_PASSWORD: db_event_manager
            DB_NAME: db_event_manager
```

```
            DB_HOST: db_event_manager

            MS_USERS: 'http://ms_user_manager:3000'
            MS_BETS: 'http://ms_bet_manager:3000'
            MS_LIST_EVENTS: 'http://ms_list_events:3000'

        ports:
            - 3003:3000
        networks:
            - network_backend
        depends_on:
            - db_event_manager
            - ms_bet_manager
            - ms_user_manager
            - ms_list_events
        restart: always
        command: ["./wait-for-it.sh", "db_event_manager:3306", "--", "nodemon"]

    frontend:
        build: ./frontend
        ports:
            - 3333:3000
        volumes:
            - ./frontend:/usr/src/app
            - /usr/src/app/node_modules
        networks:
            - network_frontend

networks:
    network_backend:
        driver: "bridge"
    network_frontend:
        driver: "bridge"
```

B *gateway-config.yml*

```
http:
  port: 8080
  admin:
  port: 9876
  host: localhost

apiEndpoints:

  list-events-no-auth:
    host: '*'
    paths: ['/available-events']

  list-events:
    host: '*'
    paths: ['/available-events/*']

  user-no-auth-required:
    host: '*'
    paths:
      - '/user/login'
```

```
      - '/user/signup'

  user-manager:
    host: '*'
    paths: '/user/*'

  event-manager:
    host: '*'
    paths: ['/event', '/event/*', '/availablebettype/', '/availablebettype/*', '/stats/', '/stats/

  bet-manager:
    host: '*'
    paths: ['/bet/', '/bet/*', '/bettype', '/bettype/*']

serviceEndpoints:
  user-manager-service:
    url: ${MS_USER_MANAGER_URL}
  bet-manager-service:
    url: ${MS_BET_MANAGER_URL}
  event-manager-service:
    url: ${MS_EVENT_MANAGER_URL}
  list-events-service:
    url: ${MS_LIST_EVENTS_URL}

policies:
  - proxy
  - jwt-db

pipelines:
  user-no-auth-required:
    apiEndpoints: user-no-auth-required
    policies:
      - proxy:
        - action:
            serviceEndpoint: user-manager-service

  event-manager-pipe:
    apiEndpoints: event-manager
    policies:
      - jwt-db:
        - action:
                secret: "secret"
      - proxy:
        - action:
            serviceEndpoint: event-manager-service

  bet-manager-pipe:
    apiEndpoints: bet-manager
    policies:
      - jwt-db:
        - action:
                secret: "secret"
      - proxy:
        - action:
            serviceEndpoint: bet-manager-service

  list-events-pipe:
```

```
  apiEndpoints: list-events
  policies:
    - jwt-db:
      - action:
              secret: "secret"
    - proxy:
      - action:
          serviceEndpoint: list-events-service

list-events-no-auth-pipe:
  apiEndpoints: list-events-no-auth
  policies:
    - proxy:
      - action:
          serviceEndpoint: list-events-service

user-manager-pipe:
  apiEndpoints: user-manager-service
  policies:
    - jwt-db:
      - action:
              secret: "secret"
    - proxy:
      - action:
          serviceEndpoint: user-manager-service
```

Referências

[1] *Vagrant vs Docker* `https://www.vagrantup.com/intro/vs/docker.html`

[2] Docker Compose Docs `https://docs.docker.com/compose/overview/`