Universidade do Minho

Mestrado Integrado em Engenharia Informática

Engenharia Web 18/19

BetESS - Casa de apostas

João Pedro Ferreira Vieira A78468
Simão Paulo Leal Barbosa A77689

16 de Junho de 2019

Conteúdo

1 Introdução

O presente documento descreve o processo de desenvolvimento da segunda parte do trabalho prático da UC de *Engenharia Web*. O problema proposto passa pela implementação de uma casa de apostas *online*, depois de definido numa etapa anterior a interação dos diferentes utilizadores com a aplicação. É ainda pretendido que a implementação deste desafio seja conseguida com a utilização de uma arquitetura de micro-serviços.

Desta forma, nas próximas secções deste relatório é descrita a arquitetura adoptada para o sistema da casa de apostas de acordo com as escolhas tomadas pelo grupo de trabalho, é explicada a forma e as ferramentas com que a implementação do projeto foi desenvolvida e é ainda demonstrada a forma como é conseguido correr o sistema com a utilização das ferramentas `Docker` e `Docker-compose`.

2 Contexto do problema

Tendo em conta que na etapa anterior deste trabalho prático foram definidas as interações dos diferentes utilizadores com a aplicação com recurso à linguagem `IFML` (*Interaction Flow Modeling Language*) e à ferramenta `WebRatio`, nesta segunda fase o objetivo passa por implementar a aplicação com recurso às linguagens/*frameworks*/ferramentas à nossa escolha, sendo ainda desafiados a adoptar uma arquitetura de micro-serviços.

Importante referir que as funcionalidades a implementar nesta fase seguem as enunciadas na fase e relatório anterior. Referir ainda que toda a descrição feita no relatório anterior sobre a camada de dados mantêm-se, ainda que com pequenas alterações que mais à frente são referidas.

3 Arquitetura da solução

3.1 Diagrama da arquitetura

Começamos a descrição da arquitetura proposta para o problema apresentando o diagrama da mesma.

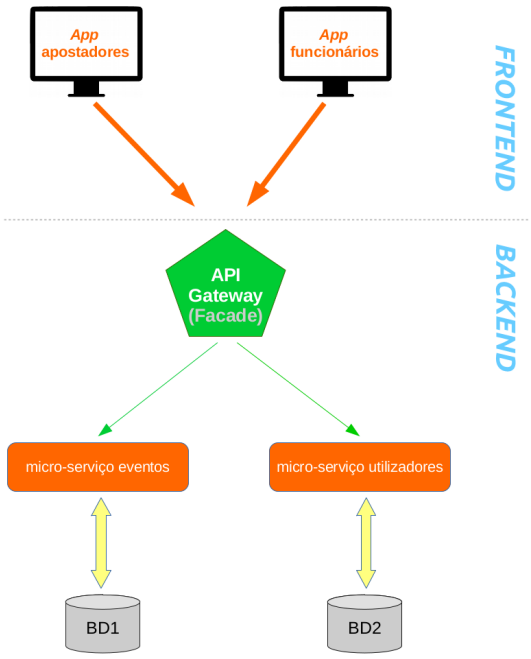

Figura 1: Arquitetura proposta

De seguida é explicado tanto o *backend* como o *frontend* aqui apresentado.

3.2 *Backend*

Através da análise da arquitetura, é possível ver que pretendemos adotar uma arquitetura virada para os micro-serviços (*Microservices Architecture*). Enquanto que

a grande maioria das aplicações desenvolvidas até ao momento por nós são de características monolíticas, neste trabalho é necessário adotar uma filosofia diferente, tendo em conta o objetivo e os requisitos de qualidade que pretendemos alcançar.

Numa arquitetura monolítica, os principais problemas passam pelo aumento da sua complexidade e tamanho ao longo do tempo, alta dependência de componentes de código face a muitas funções serem interdependentes, a escalabilidade do sistema é limitada e, para além disto, pode obrigar os *developers* a ficarem amarrados a uma certa tecnologia originalmente escolhida para o sistema (falta de flexibilidade) e leva ainda a um grande problema quando ao mundo real toca, visto ser complicado fazer alterações em produção, levando a reinicialização do sistema e aos erros que isso pode implicar.

Contrariando isto, adotando uma arquitetura com recursos a micro-serviços, passamos a ter uma maior facilidade e estabilidade de manutenção e evolução dos serviços, tendo este um baixo nível de acoplamento e interdependência. Permite ainda uma maior escalabilidade ao sistema desenvolvido podendo o *deploy* ser feito em múltiplas máquinas e servidores, etc, de forma independente. Isto permite ainda uma maior flexibilidade nas tecnologias usadas, podendo ser usada a melhor tecnologia para responder a cada caso particular e, ao contrário de uma arquitetura monolítica, as mudanças no sistema são feitas através de alterações e evoluções feitas nos serviços, não existindo um sistema que necessite de ser reinicializado para continuar a funcionar corretamente.

3.2.1 Micro-serviços

Tendo em conta a nossa intenção em reutilizar a base de dados utilizada na primeira parte deste projeto, construída com recurso ao *MySQL* e cujas relações *ER* são apresentadas de seguida:

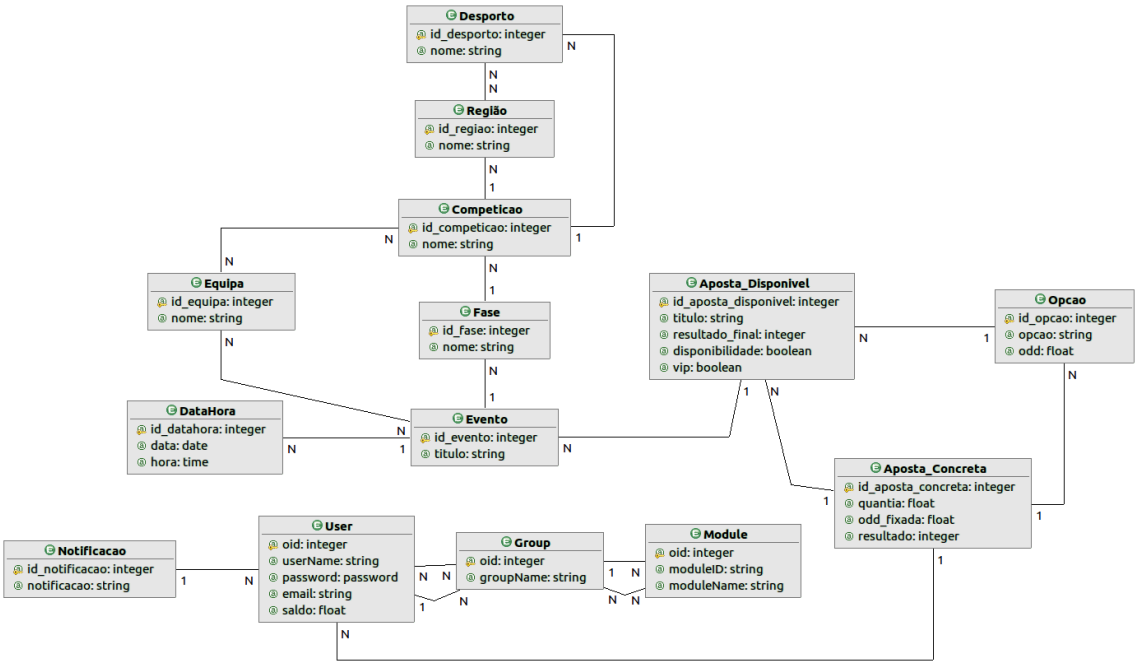

Figura 2: Modelo de dados usado na fase do projeto anterior

Existe então a necessidade de dividir um serviço e um base de dados única por mais

do que um micro-serviço, cada um com a sua própria base de dados.

Através da análise do sistema e das funções/métodos necessários implementar e com o intuito de reduzir o número de comunicações entre os micro-serviços (que no nosso caso irá sempre existir seja qual for a separação feita, o que idealmente poderia não ser o desejado para uma arquitetura deste tipo), a nossa solução contempla dois micro-serviços.

É ainda pretendido que as funções definidas nos micro-serviços sejam capazes de verificar se o utilizador que realizou o pedido tem autorização para receber as informações devolvidas.

3.2.1.1 Micro-serviço eventos

Este micro-serviço é responsável pela parte de gestão e apresentação de eventos e toda a sua categorização (*Desportos* → *Regiões* → *Competições* → *Fases* → *Eventos*), sendo desta forma neste micro-serviços que são criadas equipas, eventos, apostas disponíveis nos eventos e gestão das opções de cada uma destas aposta (assim como outras funções auxiliares).

Desta forma, a base de dados deste micro-serviço contêm as tabelas `Desporto`, `Região`, `Competição`, `Fase`, `Evento`, `DataHora`, `Equipa`, `Aposta_Disponivel` e `Opção` da base de dados anterior.

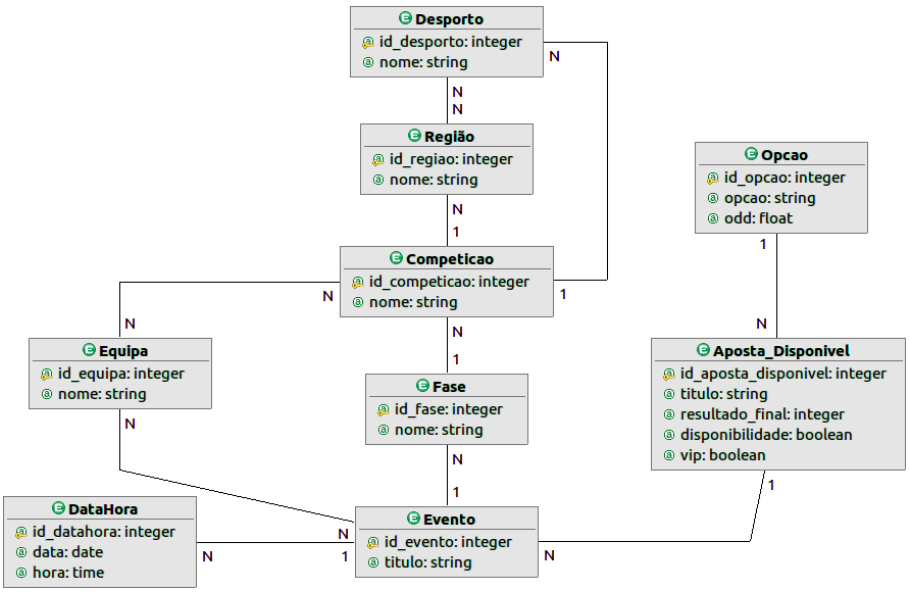

Figura 3: Base de dados do micro-serviço eventos

3.2.1.2 Micro-serviço utilizadores

Já este micro-serviço fica responsável pela informação e gestão dos utilizadores da aplicação, assim como as notificações a as apostas feitas pelos apostadores, sejam eles normais ou VIP.

Desta forma, o resto das tabelas da base de dados anterior migram para esta nova base de dados, com excepção para as tabelas `Group` e `Module`, sendo que o *id* do grupo a que cada utilizador pertence passa agora a estar como um atributo na sua própria entidade. Para além disto, é preciso realçar que como deixam de existir as relações de

uma `Aposta_Concreta` para a `Aposta_Disponivel` respetiva e para a devida `Opção`, os respetivos *id's* passam então a ser incluídos na tabela `Aposta_Concreta`, optando nós também pela inclusão de campos como o nome do evento, da aposta disponível, do evento e da opção desta aposta realizada nesta mesma tabela, de forma a não ter que interagir com o outro micro-serviço quando um apostador pretende por exemplo consultar informação sobre as suas apostas.

Sendo assim, a base de dados deste micro-serviço contêm as tabelas `User`, `Notificacao` e `Aposta_Concreta`.

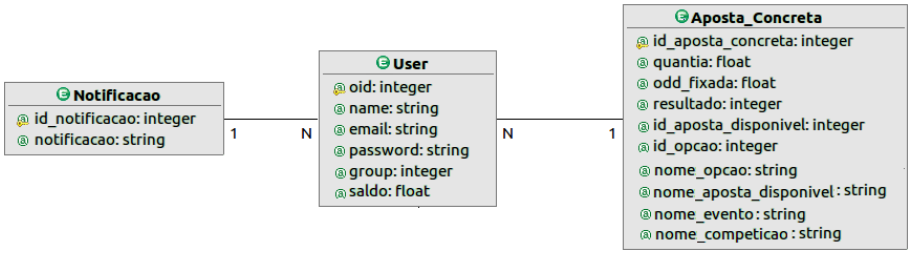

Figura 4: Base de dados do micro-serviço utilizadores

3.2.2 *API Gateway*

O objetivo deste componente passa por ser o único ponto através do qual o *frontend* deverá ter acesso, ou seja, funcionará como um *Facade* com a API que pode ser chamada de fora do sistema. Sendo assim este serviço conhece os micro-serviços disponíveis e trata de fazer reencaminhamento dos pedidos para cada um deles, sendo isto totalmente transparente para o cliente que faz o pedido.

Desta forma, o objetivo passa por impedir que os micro-serviços sejam contactados diretamente pelo cliente, podendo este apenas conhecer o endereço do *API Gateway* e todos os métodos que este fornece.

Um dos objetivos passa por realizar a autenticação dos utilizadores, por forma a que os micro-serviços apenas respondam ao que o *gateway* reencaminha e não precisem de lidar com autenticações dos utilizadores. Para tal ser conseguido, tem de ser feito um pedido ao micro-serviço utilizadores para validar os dados do utilizador (*username*, *password*) provenientes do cliente, e obter o seu *id* e grupo (*group*).

3.2.3 Outras considerações

Outras considerações relevantes passam por referir que é pretendido que cada um dos micro-serviços use uma *API REST*, assim como o *Gateway*, sendo as respostas fornecidas com recurso a *JSON*. É também relevante referir que os pedidos efetuados entre micro-serviços não devem ser realizados de forma direta mas sim através do *API Gateway*.

Para além disto, pretendeu-se desenvolver o *backend* deste projeto com recurso a `Node js` e ao conjunto de *frameworks* e livrarias a si associadas e que são disponibilizadas (tais como `Express`, `Sequelize`, etc...).

3.3 *Frontend*

Para o *frontend* desta aplicação, pensamos que a melhor opção passaria por desenvolver dois clientes independentes:

- Um para os apostadores - sejam eles apostadores regulares ou VIP
- Outro para os funcionários - que funcionam como administradores do sistema

Pensamos ser a melhor opção tendo em conta as semelhanças que existem entre os dois tipos de apostadores e sendo que a interface para os dois será muito semelhante com exceção para certos conteúdos que um tem acesso e outro não (como descrito no relatório anterior). Por outro lado, o funcionário terá acesso a vários ecrãs diferenciados e várias opções em certos ecrãs a que os clientes não têm acesso, pelo que pensamos ser mais facilitador terem uma aplicação destinada só a eles.

Para além disto, em termos tecnológicos foi planeado desenvolver ambos os clientes com recurso a *React* em conjunto com outras livrarias que nos sejam úteis.

4 Implementação da solução

4.1 *API Gateway*

Tal como referido anteriormente, todo o *backend* foi implementado com recurso a `Node js`, tendo em conta o facto de ser bastante utilizado nos dias hoje, o facto de ter um grande suporte *online*, e o facto de possuir um grande conjunto de livrarias à disponibilidade do utilizador para os mais diferentes objetivos.

Sendo assim, para criar esta máquina são utilizadas dependências como:

- `express` - pacote utilizado para criar `web APIs` e `web apps` facilmente
- `morgan` - utilizado como um *logger* para perceber através do terminal o que está a acontecer
- `helmet` - pacote utilizado para adicionar proteção a 11 tipos comuns de pedidos maliciosos, adicionando assim alguma segurança ao *API Gateway*
- `express-http-express` - pacote utilizado para redirecionar os pedidos para os micro-serviços definidos
- `http` - módulo básico para criar um servidor `HTTP` básico

A partir daqui, são então definidos os endereços dos micro-serviços, e construídas as *proxys* que são utilizadas para reencaminhar os pedidos para os mesmos.

```
const users_microservice = 'http://10.1.0.11:3001/'
const events_microservice = 'http://10.1.0.12:3002/'

const usersMS = httpProxy(users_microservice);
const eventsMS = httpProxy(events_microservice);
```

São ainda definidas as rotas que os clientes podem chamar. Desta forma, as rotas utilizadas são `'/api_users'` (para pedidos relativos ao micro-serviço utilizadores) e `'/api_eventos'` (para pedidos relativos ao micro-serviço eventos). Por exemplo, para pedidos `GET` relativo ao micro-serviço utilizadores:

```
app.get('/api_users*', verifyJWT, (req, res, next) => {
    req.body["user_group_id"] = req.group
    req.body["user_oid"] = req.userId
    req.url = req.url.replace('/api_users', '')
    usersMS(req, res, next);
})
```

O *middleware* `verifyJWT` é utilizado para realizar a autenticação dos utilizadores. Os clientes da casa de apostas assim que fazem *login* os seus dados são enviados para o micro-serviço utilizadores que devolve uma resposta que diz ao *API Gateway* se os dados estão ou não corretos. Caso estejam, com recurso à dependência `jwtwebtoken` que com *id* e *group* do utilizador juntamente com um segredo definido nesta máquina é capaz de gerar um *token* para o mesmo, que deve ser utilizado nos próximos pedidos a serem efetuados.

```
...
const id = login.data.oid;
const group = login.data.group;
var token = jwt.sign({ id, group }, process.env.SECRET, {
    expiresIn: 18000 // expires in 5 hours
});
res.status(200).send({
    success: true,
    token: token,
    id: id,
    group: group
});
...
```

Desta forma, o que o *middleware* `verifyJWT` faz é verificar a autenticação dos pedidos feitos antes dos mesmos poderem vir a ser enviados aos micro-serviços. Sendo assim, pedidos que não contêm o *token* ou que o mesmo é inválido não obtêm acesso às informações pretendidas, e os que passam estas condições, através da descodificação do *token* são obtidos dados como o *group* ou o *id* do utilizador enviados para os micro-serviços de forma a estes verificarem questões de autorização.

4.2 Micro-serviços

Quanto aos micro-serviços, os dois foram construídos de forma semelhante, com recurso à *framework* `Express`.

Tendo em conta que tínhamos em posse o ficheiro de criação da base de dados gerado pelo `WebRatio` na etapa anterior, começamos por separar manualmente o mesmo em dois ficheiros distintos de acordo com as bases de dados enunciadas em cima. Após isto, foi decidido que utilizaríamos o ORM (*Object-Relational Mapper*) `Sequelize`, que comunica com o `MySQL` e permite-nos utilizar um conjunto de funcionalidades de forma a não necessitarmos de utilizar *querys* `SQL` para obtermos os dados desejados. Para utilizar o `Sequelize` em `Node js` utilizamos uma ferramenta que converte automaticamente o nosso *schema* das bases de dados para modelos que podemos utilizar no código. Um exemplo de uma função que utiliza esta *framework* é apresentado de seguida, na qual se pretende depositar uma certa quantia no saldo de um apostador.

```
module.exports.adicionarQuantia = (userId, valor) => {
    return models.user.increment(
        ['saldo'],
        {
            by: valor,
            where: {oid: userId}
        }
    )
}
```

Para garantir que um dado utilizador só tem acesso às informações a que o mesmo deve ter, é utilizado um *middleware* na definição das rotas, por exemplo:

```
router.post('/delete/:oid', mw.verifyFuncionario, async (req, res) => {
  UserController.remove(req.params.oid).then(() => {
    res.send({success: true});
```

```
  })
  .catch(err => {
    res.status(500).send(err);
  })
})
```

Desta forma, este *middleware* tenta garantir que apenas um funcionário possa conseguir eliminar um determinado utilizador. Neste caso em específico, este *middleware* utiliza o *group* do utilizador que foi descodificado no *API Gateway* e enviado para este micro-serviço para verificar a autenticação do pedido recebido:

```
exports.verifyFuncionario = function (req, res, next){
    if (req.body.user_group_id == 2)
        next();
    else
        return res.status(500).send({
            auth: false,
            message: 'Falha de autenticação. Apenas Funcionários permitidos.'
        });
};
```

4.3 *Frontend*

Para desenvolver as duas aplicações pretendidas (uma para apostadores e outra para funcinários) foi utilizado `React` tendo em conta a grande utilização do mesmo nos dias de hoje para construir aplicações Web e assim aproveitarmos para aprender a utilizar o mesmo. A juntar a isto, o `React` tem ainda uma grande documentação e suporte *online*, um vasto leque de *frameworks* e livrarias que fornecem componentes já feitos e estilização de componentes de acordo com os padrões atuais de desenvolvimento de interfaces Web, e a juntar a isto, esta tecnologia permite uma "fácil"reutilização de componentes construídos ao longo da aplicação.

Como ponto de partida para o desenvolvimento das aplicações, é definido em cada um dos projetos um ficheiro `Api.js` no qual são declaradas as funções que vão comunicar com o *backend* e que permite assim serem chamadas nos vários componentes definidos, sendo que qualquer alteração numa destas funções fica assim a ser apenas necessário mudar num único ficheiro. Por exemplo, a função que permite efetuar *login* é implementada da seguinte forma (sendo `jsonFetch` e `generateUrl` métodos auxiliares):

```
export const login = (body) => (
    jsonFetch(generateUrl(BASE_URL, '/login'), { method: 'post', body })
);
```

A partir daqui, e para construir as páginas necessárias, são utilizados também alguns componentes e estilização fornecidos pela ferramenta `Semantic UI React`.

Quando um utilizador realiza o *login* na aplicação, o *token* que é devolvido pelo *API Gateway* é guardado em memória de forma a ser utilizado em todos os restantes pedidos e a poder ser verificado que um dado utilizador já está ou não autenticado. Para além disto, é ainda guardado o *group* a que o utilizador pertence de forma a conseguir mostrar apenas aquilo a que o mesmo tem direito (sejam páginas, funcionalidades numa página, etc...), assim como o *id* do mesmo e o seu *username* que podem dar jeito ao

longo do desenvolvimento das aplicações. Por exemplo, na aplicação dos apostadores, caso a autenticação seja realizado com sucesso, os dados são guardados da seguinte forma:

```
...
if (response.success){
    localStorage.setItem('token', response.token);
    localStorage.setItem('username', username);
    localStorage.setItem('user_id', response.id);

    if (response.group === 1)
        localStorage.setItem('userType', 'normal');
    else if (response.group === 3)
        localStorage.setItem('userType', 'vip');

    window.location.reload();
}
...
```

4.3.1 Nivel A *WCAG*

De forma a conseguir uma orientação para o desenvolvimento da interface , seguimos as regras de nível A de *Web Content Accessibility Guidelines (WCAG)*.

A maioria dos botões, *links*, possíveis ações do utilizador seguem o seguinte padrão: *icon* e texto para descrição visual e textual da função. Alguns botões fogem à regra quando o *icon* é suficientemente explícito, mas nessas casos é possível ver a descrição textual ao passar o rato no botão. Um exemplo deste padrão está na seguinte imagem:

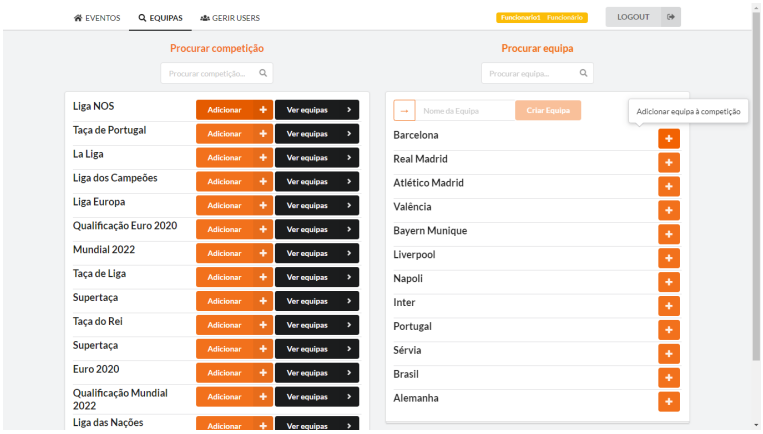

Figura 5: Página de gestão de equipas na aplicação para funcionários

São também utilizadas cores nos botões de forma consistente para descrever tipos de ações semelhantes. Nestes exemplos de seguida, quando o utilizador escolhe para terminar evento (botão vermelho) tem acesso aos botões com as opções em vermelho para escolher o resultado final e terminar evento. Mas quando o utilizador escolhe para mudar odds (botão azul), as opções tem a cor azul.

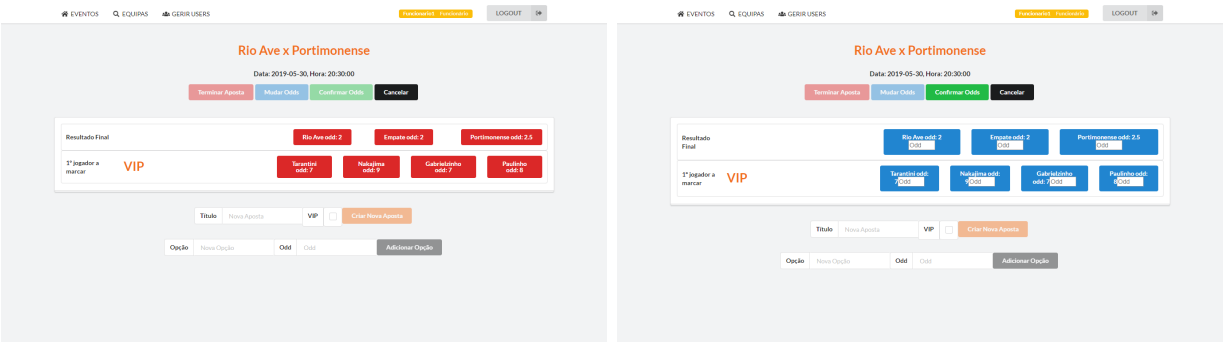

Figura 6: Terminar evento e alterar *odd's* de um evento na aplicação para funcionários

Foi tida também a preocupação para deixar a apresentação responsiva relativamente ao tamanho do ecrã, sendo que a barra do menu da aplicação muda de forma para ecrãs de menor dimensões. Este exemplo em concreto é apresentado de seguida.

Figura 7: Menu vertical utilizado para ecrãs com dimensões menores

De forma a tornar a aplicação mais reativa e rápida, evitámos fazer *reload* de uma página sempre que fazemos uma alteração do conteúdo com pedidos ao *backend*, como por exemplo na página do que permite aos funcionários adicionar e remover equipas de competição, tornar e retirar a categoria de VIP a apostadores da casa de apostas, ou a capacidade de os apostadores limparem a sua própria secção de notificações.

Em todos estes casos, fazemos as alterações ao estado dos componentes `React` quando os pedidos foram feitos com sucesso, e assim nunca é necessário fazer *reload* da página para atualizar o conteúdo alterado.

A procura de eventos, principalmente para o apostador é muito intuitiva, visto poder escolher eventos por desporto, região, competição e fase. Tudo acessível numa só página:

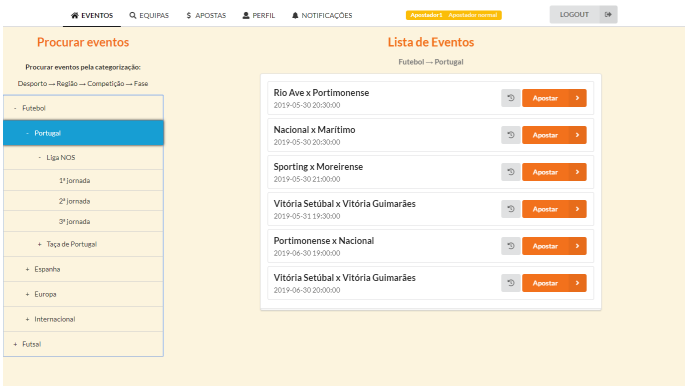

Figura 8: Página de procura de eventos na aplicação para apostadores

4.3.2 *Real-time responsiveness* e *Scalability*

Foram tidas diversas precauções relativamente à realização de apostas por parte dos utilizadores e ao término de apostas disponíveis, de forma a não ser possível apostar em apostas já terminadas. Quando um utilizador aposta, é feito um pedido ao *backend* do sistema que trata de realizar a verificação da disponibilidade desta aposta. Caso já não esteja disponível o apostador é assim notificado.

Um processo semelhante ocorre em relação às *odd's* aquando do efetuar de uma aposta, visto que as mesmas podem mudar ao longo do tempo. Assim, quando um apostador se encontra no processo de apostar, mas em simultâneo, foram alteradas as *odd's* dessa aposta, o utilizador é notificado e a aposta não é realizada, visto que é realizada uma verificação do valor da *odd* no momento da aposta.

5 *Deploy* utilizando `Docker`

Para "montar"o *backend* deste projeto, foi utilizado o `Docker-compose`, uma ferramenta que permite a definição e a execução de aplicações `Docker` *multi-container*. O ficheiro `docker-compose.yaml` é apresentado de seguida.

```
version: '3.5'

# ------------------------
# BetESS containers network
# ------------------------

networks:
  frontNetwork:
    name: front-network
    driver: bridge
    ipam:
      driver: default
      config:
        - subnet: 10.1.0.0/24
  backNetwork:
    name: back-network
    driver: bridge
    ipam:
      driver: default
      config:
        - subnet: 172.28.0.0/16

   # -------------------------
   # BetESS backend containers
   # -------------------------

   services:
     api-gateway:
       build: ./api-gateway
       image: betess-api
       container_name: betess-api
       networks:
         frontNetwork:
           ipv4_address: 10.1.0.10
       ports:
         - '3000:3000'

     BetESSserver_users:
       build: ./BetESSserver_users
       image: betess-msu
       container_name: betess-msu
       networks:
         frontNetwork:
           ipv4_address: 10.1.0.11
         backNetwork:
           ipv4_address: 172.28.0.11
       ports:
         - '3001:3001'
       depends_on:
         - db

BetESSserver_events:
  build: ./BetESSserver_events
  image: betess-mse
  container_name: betess-mse
  networks:
    frontNetwork:
      ipv4_address: 10.1.0.12
    backNetwork:
      ipv4_address: 172.28.0.12
  ports:
    - '3002:3002'
  depends_on:
    - db

db:
  build: ./MySQL_scripts
  container_name: mysqldb
  image: mysql
  restart: always
  ports:
    - "8998:3306"
  networks:
    backNetwork:
      ipv4_address: 172.28.0.13
  environment:
    MYSQL_DATABASE: betess
    MYSQL_ROOT_PASSWORD: password
```

Como se pode verificar pelo ficheiro colocado em cima, são criadas duas redes, uma para a comunicação entre o *API Gateway* e os micro-serviços (`10.1.0.0/24`), e outra para a comunicação dos micro-serviços com a base de dados (`172.28.0.0/16`). Os serviços declarados funcionam como quatro *containers* que se pretendem construir:

- `api-gateway` - *container* para executar o *API Gateway*

  O `build: ./api-gateway` indica que na pasta `api-gateway` se encontra a `Dockerfile` deste *container*, cujo conteúdo é o seguinte:

  ```
  FROM node:8-alpine

  # Create app directory
  RUN mkdir -p /usr/src/app
  WORKDIR /usr/src/app

  # Install dependencies
  COPY package.json .
  RUN npm install

  # Bundle app source
  COPY . .

  # Exports
  EXPOSE 3000
  CMD [ "npm", "start" ]
  ```

- `BetESSserver_users` e `BetESSserver_events` - representam os *containers* para execução dos micro-serviços. O `Dockerfile` de ambos é semelhante ao apresentado em cima em `api-gateaway` (visto serem ambos aplicações `Node js`). De referir que ambos dependem do *container* da base de dados (`depends_on`).

- `db` - representa o *container* para execução das bases de dados `MySQL`. O `Dockerfile` para este *container* tem o seguinte conteúdo:

  ```
  FROM mysql

  COPY create_schema_users.sql /docker-entrypoint-initdb.d/
  COPY populate_database_users.sql /docker-entrypoint-initdb.d/
  COPY create_schema_events.sql /docker-entrypoint-initdb.d/
  COPY populate_database_events.sql /docker-entrypoint-initdb.d/
  COPY access.sql /docker-entrypoint-initdb.d/
  COPY my.cnf /docker-entrypoint-initdb.d/
  ```

  Desta forma, são copiados os ficheiros `SQL` que serão executados ao fazer *build* do *container*. Os quatro primeiros são relativos à criação dos esquemas das duas bases de dados e à sua população inicial. O ficheiro `access.sql` tem como intuito a criação de um utilizador que tem total acesso às bases de dados e cujas credenciais são utilizadas nos micro-serviços.

  ```
  CREATE USER 'betess'@'%' IDENTIFIED BY 'pass';
  GRANT ALL PRIVILEGES ON betess_users.* TO 'betess'@'%' WITH GRANT OPTION;
  GRANT ALL PRIVILEGES ON betess_events.* TO 'betess'@'%' WITH GRANT OPTION;
  ALTER USER 'betess'@'%' IDENTIFIED WITH mysql_native_password BY 'pass';
  FLUSH PRIVILEGES;
  ```

Já o ficheiro `my.cnf` tem como objetivo editar as configurações da base dados, visto inicialmente termos tido alguns problemas com carateres especiais, que foram contornados com este ficheiro de configuração.

```
[client]
default-character-set = utf8mb4

[mysql]
default-character-set = utf8mb4

[mysqld]
init-connect='SET NAMES utf8mb4'
collation_server=utf8mb4_unicode_ci
character_set_server=utf8mb4
skip-character-set-client-handshake
```

6 Conclusão

Podemos concluir, através da realização deste trabalho prático, que consideramos ter atingidas as metas esperadas para o mesmo.

Numa fase inicial, pusemos em prática os conceitos abordados nas aulas da UC, de forma a construir um *data model* de acordo com o tema proposto, assim como adquirir e implementar um site funcional com recurso à linguagem IFML e à ferramenta `WebRatio`.

Na segunda fase, conseguimos construir um *backend* adoptando uma aquitetura de micro-serviços que pensamos ir de encontro ao objetivo definido, seguindo-se da implementação das duas aplicações para os utilizadores do sistema e ainda da utilização do `Docker` e `Docker-compose` para realizar o *deploy* do *backend* do sistema.