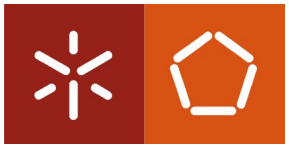

Universidade do Minho
Mestrado Integrado em Engenharia Informática
Departamento de Informática

Engenharia Web

BetESS Web Platform

Daniel Fernandes Veiga Maia A77531
Vitor Emanuel Carvalho Peixoto A79175

Ano Letivo 2018/2019

Resumo

Este relatório foi desenvolvido com o intuito de especificar e justificar as decisões tomadas no decorrer da segunda parte do desenvolvimento deste projeto. Nesta, foi implementado o sistema descrito no relatório anterior, tendo por base a arquitetura de serviços especificada. O sistema desenvolvido é composto por uma página *web*, onde se inclui uma interface gráfica, serviços de ligação à base de dados e gestão de pedidos e uma camada intermediária entre a interface e os serviços.

Conteúdo

1 Introdução

Antes de se iniciar o desenvolvimento de qualquer aplicativo ou página *web*, é necessário haver um conhecimento explícito das tecnologias ao nosso dispor e das necessidades, não só atuais, mas também a longo prazo, do projeto a desenvolver. Nesta perspetiva, o processo de levantamento de requisitos e de estudo do contexto atual do mercado é importante não só para o sucesso do projeto, mas para a sua utilidade e longevidade.

Tendo como objetivo o desenvolvimento de um *website* de apostas *responsive* e acessível, a atenção deve recair sobre a arquitetura do sistema. Um serviço monolítico é uma válida abordagem ao desenvolvimento de qualquer projeto *web*, mas com o aparecimento de dispositivos móveis e outras gamas de aparelhos para além de *desktops*, este tipo de arquiteturas perecem, devido à maior probabilidade de erros e pior ainda, reduzem drasticamente a escalabilidade da solução.

Uma das soluções existentes é a aplicação de uma arquitetura baseada em microserviços. Este tipo de arquiteturas permitem separar o código em várias partes, que correm em processos separados, numa orquestração de serviços independentes e comunicativos. O projeto, como um todo, torna-se mais tolerante a faltas, acessível e escalável.

Para além disso, dada a existência de uma larga variedade de dispositivos ao qual podemos aceder a um *website*, a sua acessibilidade e adaptabilidade deve ser vista como um ponto de extrema importância.

É sobre estes pontos que o trabalho a ser desenvolvido deve incidir, tendo como objetivo final o desenvolvimento de um serviço *web* (*BetESS*) que permita efetuar apostas em eventos, com um conjunto de funcionalidades definidas e que corresponda aos requisitos de sistema abordados.

2 Estudo inicial

Recapitulando o que foi introduzido no capítulo anterior, a necessidade de construir um sistema acessível, escalável e responsivo remete-nos para a construção de um sistema baseado em **microserviços**.

Para além disso, esses requisitos serão validados com a utilização de *frameworks*. A utilização de *frameworks* revela-se bastante vantajosa neste caso uma vez que se trata de uma solução testada e eficiente para problemas que iremos encontrar. A maioria destas soluções encontra-se vastamente documentada e permite uma integração fácil e segura com as tecnologias a serem utilizadas.

Qualquer aplicativo *web* segue uma arquitetura básica onde o serviço é dividido em duas partes, *backend* e *frontend*, responsáveis pelo processamentos de dados e a ligação aos dados da aplicação e pela recolha e apresentação de dados aos utilizadores, respetivamente.

Baseando nestes princípios, foi desenvolvido uma proposta de arquitetura dos componentes que irão compor o sistema.

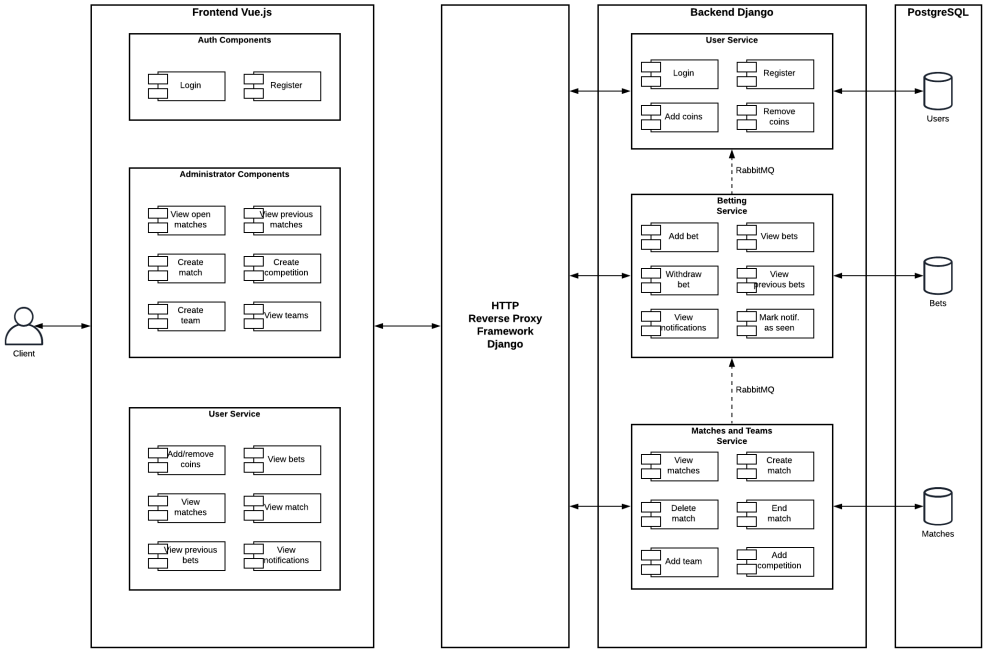

Figura 1: Diagrama de componentes

3 Server-side

No seguimento do que foi abordado anteriormente, a divisão do sistema em microserviços deve ser coerente e responsável tendo em conta as funcionalidades e dados que o sistema lida. Deste modo, a divisão foi efetuado tendo em conta três serviços muito distintos: um serviço que gere os utilizadores do sistema, incluindo a sua autenticação e os seus dados pessoais onde se encontra o seu saldo atual (*User Service*); um serviço responsável pelas equipas, bem como as suas partidas (*Matches and Teams Service*). Este serviço é facilmente partilhável entre várias outras aplicações, podendo futuramente ser gerido por uma API externa, por exemplo; e finalmente um serviço responsável pelas apostas, onde são também geradas as respetivas notificações associadas a cada aposta (*Betting Service*). Todos os serviços implementados operam totalmente independentes entre si.

Cada serviço contém a sua base de dados independente, de modo a garantir a disponibilidade parcial do sistema em caso de falha de um microserviço e facilitar a escalabilidade da solução.

3.1 Framework

A escolha de uma *framework* deve ser estudada e aplicada ao contexto do desenvolvimento e do projeto. Para tal, foi feita uma pesquisa sobre a oferta disponível e por fim efetuada uma escolha. Esta recaiu em ***Django***.

O *Django* é uma *framework Python* com vantagens que beneficiam o objetivo deste projeto. Para começar, providencia um mecanismo de autenticação baseado em *tokens*, assegurando um sistema seguro e protegido contra *third-party attacks*. Para além disso, permite desenvolver um sistema escalável, uma vez que é baseado numa arquitetura "*share nothing*"que permite a adição de novas funcionalidades em qualquer estágio do desenvolvimento. Ainda oferece grande compatibilidade com várias outras *frameworks*, incluíndo uma fácil integração com o *client-side* através da sua *REST framework*.

A estrutura de dados pode ser criada automaticamente nesta *framework* com recurso a estruturas modulares e uma fácil integração com um vasto leque de sistemas de gestão de bases de dados relacionais, sendo que a opção escolhida foi ***PostgreSQL***.

3.2 Comunicação

Apesar da sua relativa separação, os microserviços não deixam de conter relações de dados entre si. Como tal, caso uma alteração nos dados de um microserviço que afete os dados de outro, é necessário que esta atualização seja comunicada. Para esse efeito, recorreu-se ao *software open-source* de *message-broker* ***RabbitMQ***, o qual permite assegurar que as bases de dados dos microserviços se mantenham consistentes entre si, e caso uma atualização aos dados ocorra enquanto um dos servidores se encontre indisponível, é guardada a atualização numa *queue* até que este volte a estar *online*.

Assim, toda a comunicação estabelecida entre os microserviços é efetuada através de *queues* utilizando o *RabbitMQ*. Exemplos práticos da aplicação deste *software* são encontrados em diversas das funcionalidades do sistema, como por exemplo o término de um evento. Esta é porventura a funcionalidade mais exigente no sistema, uma vez que implica a atualização do resultado de um evento no *Matches and Teams Service*, a atualização do lucro das apostas relativas a esse evento e a geração das respetivas notificações no *Betting Service* e a atualização do saldo do utilizador caso a sua aposta tenha sido vencedora no *User Service*. Como tal, no momento da finalização de um evento, este insere uma mensagem na *queue* do serviço de apostas para sinalizar que este deve fechar as apostas efetuadas e notificar os apostadores dos resultados.

4 Client-side

O *frontend* do sistema fornece aos utilizadores uma forma de aceder e interagir com os dados. É através deste que os utilizadores se autenticam, visualizam os eventos, efetuam apostas e todas as restantes funcionalidades do sistema.

4.1 Framework

Tal como na escolha da *framework backend* a utilizar como base de desenvolvimento do projeto, também a escolha da *framework frontend* deve ser estudada e aplicada ao contexto do desenvolvimento e dos objetivos do projeto. Para tal, é necessário ter em conta a utilização de *Django* como *framework backend* e aplicar uma que seja complementar.

Após uma pesquisa efetuada, a escolha incidiu na *framework* baseada em *JavaScript*, ***Vue.js***. A utilização desta *framework* permite desenvolver uma *interface* responsiva e dinâmica, sendo capaz de renderizar páginas em memória, tornando-as mais rápidas dada a sua menor utilização do *browser*.

Este *client-side* é então único e integra todos os três microserviços existentes, pedindo e fornecendo dados de cada um deles, de forma independente.

4.2 Interface

De modo a que cumprir os requisitos de uma interface fácil de utilizar, responsiva e reativa para diferentes plataformas (*web*, *mobile*, etc.) foram aplicados um conjunto de *frameworks* que auxiliam no desenvolvimento e permitem criar um sistema mais adaptativo, escalável e rápido de entender. De entra as *frameworks* gráficas implementadas destaca-se o *Bootstrap* e *Vue Material*.

Todas as *frameworks* e pacotes utilizados fornecem uma integração suave com o *Vue.js*, aumentando de largo modo a facilidade de uso do aplicativo, bem como a adaptabilidade do sistema ao largo leque de dispositivos que podem utilizar o sistema.

Verifica-se também que a *interface* desenvolvida cumpre com sucesso os critérios para o nível A das *Web Content Accessibility Guidelines 2.0* (WCAG).

Abaixo mostramos algumas imagens do resultado final da *interface*:

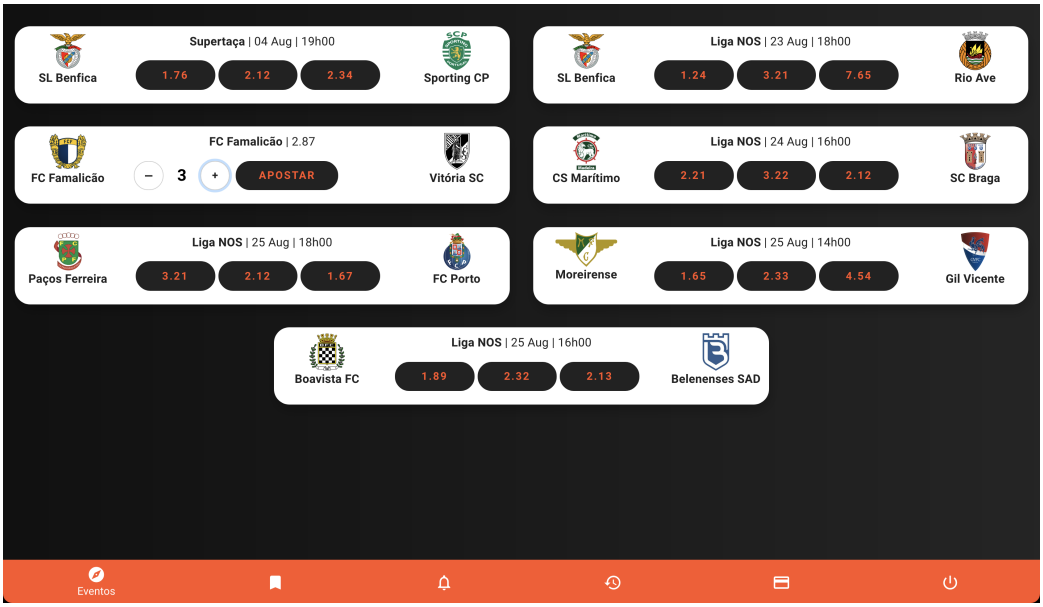

Figura 2: Página de visualização de eventos e efetuação de apostas (Utilizador).

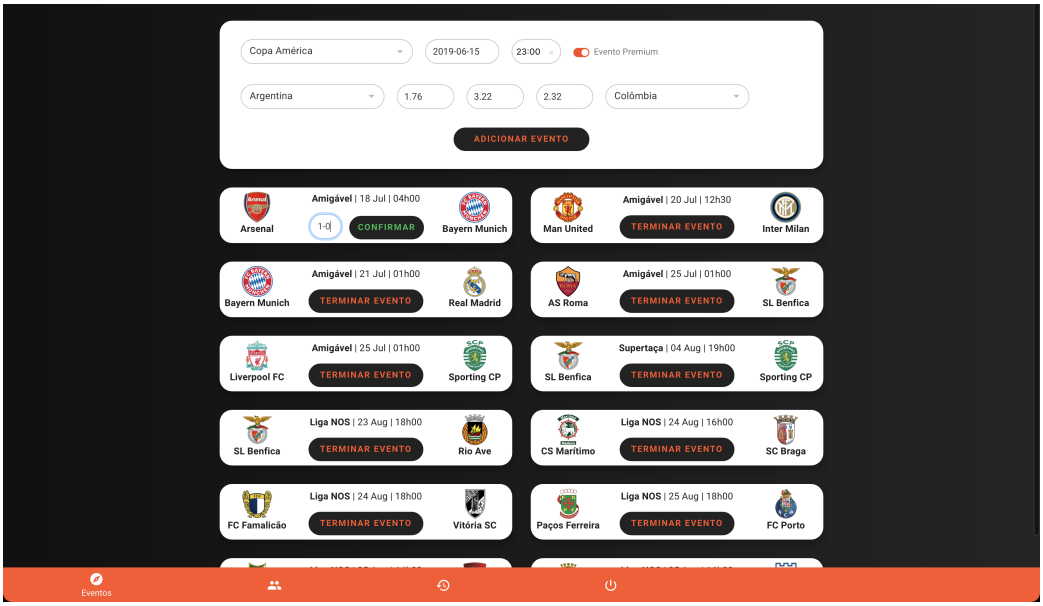

Figura 3: Página de gestão (adição e fecho) de eventos (Administrador).

5 API Layer

Para abstrair os microserviços ao cliente, foi construído um serviço de acesso na forma de um servidor *reverse proxy*. Este providencia um método simples e eficiente de distribuir os pedidos dos clientes que permita também distribuir a carga proveniente dos clientes pelos microserviços. Dado que se prevê um ambiente de elevado trânsito de pedidos, optou-se por manter a lógica desta API mínima para evitar a criação de um *bottleneck* no sistema.

Para a implementação deste servidor *reverse proxy* foi utilizada como base um projeto *Django* com recurso à *framework* ***Django-RevProxy*** que permite de um modo fácil redirecionar os pedidos efetuados pelo *frontend* para o respetivo *microserviço*. A utilização desta camada intermédia aumenta a escalabilidade do sistema, na medida em que o *client-side* lida apenas com um endereço. Para além disso permite uma maior segurança do sistema, impedindo contacto direto entre o cliente e o servidor.

6 Resultado Final

Tendo como ponto de partida a modelação dos fluxos de interação desenvolvidos na primeira parte deste projeto e a implementação efetuada com recurso ao *WebRatio*, podemos concluir que o sistema desenvolvido nesta etapa cumpre corretamente com todos os requisitos funcionais implementados na etapa anterior. Todas as páginas e componentes projetados nessa fase foram replicados e melhorados nesta fase, permitindo ter um aplicativo com uma *interface* agradável, adaptável e de fácil uso.

Relativamente aos requisitos de qualidade do sistema, podemos concluir que o resultado final revelou-se satisfatório uma vez que a arquitetura proposta foi corretamente aplicada permitindo obter uma solução fragmentada em microserviços com a aplicação correta de diversas *frameworks* que permitiram o desenvolvimento de um aplicativo responsivo, reativo e escalável.

Seguindo em frente para um ambiente de produção será necessário efetuar o *deployment* dos microserviços. Este é um processo que poderá requerer a replicação de cada um destes numa nova máquina. Para tal, é necessário efetuar um conjunto de ações:

- Obter pacotes de instalação dos programas necessários.
- Obter as dependências dos programas utilizados. Este é de particular importância no *Django*.
- Criar ficheiros de configuração dos programas, nomeadamente, de criação de bases de dados.
- Replicar os dados das bases de dados, caso existirem.

Já no caso do *scaling* dos microserviços, a resposta não é tão simples. Ao contrário de uma arquitetura monolítica, na qual se pode simplesmente alocar mais recursos ou gerar várias instâncias do serviço, numa arquitetura de micro-serviços, é necessário ter em conta as necessidades de cada um dos micro-serviços de modo a que estes cresçam em conjunto. Como tal, seria necessário efetuar uma análise da carga de trabalho imposta sobre cada um deles com o objetivo de determinar a melhor política de alocação de recursos que beneficie não apenas cada um dos micro-serviços como o sistema em geral.

7 Conclusões e Trabalho Futuro

Tendo sido dado como terminado o desenvolvimento da implementação do sistema *BetESS*, podemos afirmar que a implementação de uma arquitetura de microserviços permite a criação de um sistema com um índice de disponibilidade mais elevado, enquanto providenciando também uma estrutura modular que permite uma fácil introdução de novos microserviços, bem como a gestão dos existentes. Por outro lado, a flexibilidade do *Vue.js*, tanto na construção de páginas como no seu mecanismo de *routing*, torna a personalização do cliente acessível e compreensiva enquanto ainda sendo bastante reutilizável com o seu uso de componentes.

Ainda assim, esta arquitetura comportou um conjunto de desafios de implementação, nomeadamente, a sua complexidade. De facto, a necessidade de coordenar os três serviços torna o desenvolvimento do sistema bastante mais delicado comparadamente à gestão de uma arquitetura monolítica, sendo que é introduzido também o problema da consistência de dados.

Pode-se de certa forma concluir que a implementação de uma arquitetura baseada em microserviços deve ser estudada uma vez que não é a solução ideal para qualquer tipo de *software*. Aplicativos simples acabam por não ter necessidade de um sistema tão complexo, no entanto o crescimento futuro desse mesmo aplicativo deve ser estudado, pois a longo prazo a utilização de uma arquitetura monolítica pode limitar o crescimento do aplicativo.

Concluindo, a implementação da plataforma *BetESS* revelou-se bastante produtiva. Obteve-se desta forma uma melhor compreensão das arquiteturas frequentemente utilizadas por serviços *web* na atualidade e uma aprendizagem geral sobre as diversas *frameworks* aplicadas, bem como as vantagens da sua aplicação.