Notas sugeridas:

*Não tente descrever a função de cada componente, mas descreva detalhadamente a função de cada componente invulgar ou crítico*

*As figuras e as tabelas devem ser legíveis, instrutivas,*

*legendadas e ter título.*

*Todas as ilustrações devem ser descritas e referidas no texto*

*Os exemplos devem ser suficientemente detalhados para*

*ilustrar o conceito.*

**Relatório Final   
  
Resumo**

A Players Net é uma plataforma nova no mercado do desporto em Portugal. É uma rede social semelhante ao LinkedIn cujo objetivo é proporcionar a todos os clubes em Portugal uma ferramenta online que permita encontrar atletas, treinadores e fisioterapeutas, em qualquer contexto, permitindo a personalização da procura, tendo em conta os seus critérios de deteção de talento.

Para os atletas fornece uma ferramenta que proporciona uma forma de autopromoção perante os clubes, bem como uma vertente interativa e competitiva com os restantes atletas que utilizam a Players Net.

A visão deste projeto é desenvolver o desporto em Portugal fornecendo as ferramentas necessárias de promoção e deteção de talentos aos principais intervenientes do Desporto: Clubes, Atletas, Treinadores e Fisioterapeutas. Como se trata de um projeto de caris académico, vamos usar apenas o futebol como desporto “pivot”.

A propósito essencial é proporcionar a atletas e clubes um meio de interação onde os atletas possam promover as suas aptidões e competências, e os clubes possam detetar todos os talentos existentes em Portugal e futuramente no estrangeiro.

**Agradecimentos**Quero agradecer ao meu grande amigo Diogo Pires, que me ajudou de forma notória no desenvolvimento da parte servidora desta aplicação, bem como na conceção da lógica e modelo de negócio da mesma.  
  
Deixo também os meus mais sinceros agradecimentos ao meu orientador - professor Carlos Martins- , por me proporcionar as melhores e mais confortáveis condições desde o início do projeto, sendo que foi difícil conseguir um orientador cujo core tecnológico se assemelhasse ao máximo com o da plataforma, voluntariando-se sempre para me ajudar no que fosse preciso.

**Índice**

**Capítulo 1 – Introdução**  
  
Num mundo onde a automaticidade e rapidez de processos nunca teve tanto valor, há a recorrente necessidade agilizar processos que carecem de um bom investimento de tempo e trabalho. Foi nesse sentido que cresceu a conhecida plataforma LinkedIn e é igualmente nesse sentido que nasce a Players Net.  
  
A plataforma ataca duas grandes vertentes:   
  
Scouting  
  
Para um clube desportivo, fazer scouting de um atleta é um processo trabalhoso e utiliza imensos recursos. Existem base de dados com informação dos atletas federados das variadas modalidades desportivas existentes, no entanto, essa informação não só raramente está atualizada para os escalões menos seniores ou modalidades menos conhecidas, como não há uma constante validação dos atletas desses mesmos dados (porque não é do seu interesse) nem possui dados que não se consiga obter numa típica “folha de avaliação” desportiva.   
Com isto, os clubes têm de ter recursos exclusivamente alocados para scouting. Viajam pelo país para testemunhar presencialmente a performance dos atletas, enviam emails, fazem telefonemas, encontram-se com pessoas que possam também dar a sua opinião acerca do atleta. Tudo para, no final, terem um feedback mais real possível.  
  
Com a Players Net, os atletas atualizam os seus dados, sendo eles responsáveis pela informação introduzida na plataforma. Para além disso, outros atletas, treinadores ou clubes podem dar o seu parecer acerca da informação introduzida pelo atleta, através de recomendações escritas, votos, confirmações de “skills”, entre outros. O que faz com que a informação alcance o nível mais alto possível de veracidade.  
  
Os dados estatísticos são automaticamente atualizados pela plataforma, que recorre a outras fontes para o fazer, mas o utilizador **pode** e **deve** confirmar esses dados.   
Esses dados não só são confirmados pelo jogador como, sempre que há incoerência com outros dados obtidos através das fontes nas quais a Players Net confia, há um sistema de validação que só o rotula (através de um símbolo que representa um dado validado pela Players Net) como válido quando todos os intervenientes desse acontecimento também o validarem.

Por fim, há um sistema avançado de pesquisa de atletas que permite que o responsável pelo scouting de jogadores de um determinado clube - juntamente com o nível de validação de dados que a plataforma fornece, a atualização constante de informação por parte dos jogadores e outros intervenientes e o feedback de outras entidades através de recomendações escritas ou votos - faça o seu trabalho de scouting em meia dúzia de cliques.

Promoção dos atletas

Da mesma forma que o LinkedIn permite que os seus utilizadores se promovam perante os recrutadores de outras empresas, a Players Net permite que os seus atletas se promovam perante os recrutadores de outros clubes. Especialmente atletas com menos “visibilidade automática”.  
  
A possibilidade de os jogadores introduzirem vídeos ou outro tipo de informação digital, terem em destaque nas suas páginas as suas grandes conquistas individuais ou coletivas, os seus dados estatísticos e não estatísticos e partilharem com toda a comunidade do seu setor desportivo uma informação constantemente atualizada, são tudo fatores fundamentais para a sua promoção como atleta. Não há outra plataforma que permita este tipo de promoção pessoal a nível desportivo.

Conteúdo do relatório

Neste documento encontrar-se-á uma explicação detalhada do processo de desenvolvimento deste projeto.  
Não estando organizado de uma forma cronológica, o relatório aborda os aspetos essenciais para a compreensão da metodologia de trabalho utilizada, abordagem técnica, workflow funcional da plataforma, explicação da escolha das tecnologias, abordagem algorítmica e também uma partilha de opinião pessoal no desenvolvimento de software e plataformas web.

No 2º Capítulo, para além da terminologia do relatório, está enunciado em síntese o processamento de dados automático que a Players Net oferece, bem como os aspetos funcionais e enumeração de páginas da plataforma.  
  
No Capítulo seguinte é explicada a escolha da abordagem tecnológica para o desenvolvimento deste projeto. Numa vasta gama de linguagens e “Stacks de desenvolvimento” possíveis, aqui encontram-se as razões pelas quais a “MEAN Stack” foi a escolhida, incluindo as suas vantagens num ambiente que efetua “Web crawling”, benefícios em utilizar um sistema de base de dados não relacional (NoSQL) e o porquê de se decidir apresentar toda essa informação através de uma linguagem que essencialmente sustenta as “SPAs” - Angular.  
  
Um “Web Crawler”, por ser tipicamente concebido para adquirir dados que não nos são diretamente fornecidos (por exemplo, através de um web service), é uma ferramenta algo polémica no desenvolvimento de software. Esta questão e o seu conceito geral encontram-se melhor aprofundados no 4º Capítulo, seguido duma abordagem ao conceito de “Gamification” utilizado na Players Net presente no Capítulo seguinte.  
  
O 6º e 7º Capítulos falam sobre a arquitetura e descrição técnica respetivamente. Neste último está descrito com detalhe a solução técnica da Players Net, com exemplos práticos, definição e descrição dos componentes de serviço e da interface de utilizador, sendo que o seu detalhe funcional, acompanhado de um exemplo de manual de utilizador ilustrado com imagens da plataforma, encontra-se no penúltimo Capítulo deste relatório.  
  
Por fim, no último Capítulo, estão as conclusões tiradas no final da elaboração deste projeto, onde são descritas as lições aprendidas com o seu desenvolvimento, o que poderia ter sido diferente, em que medida foram atingidos os objetivos deste projeto e as perspetivas para o seu trabalho futuro.

**Capítulo 2 – Solução Funcional**

Terminologia

- Skill: Talento ou atributo mensurável de um determinado utilizador.  
 - Badge: Título em destaque atribuído a um utilizador quando preenche um determinado número de requisitos.  
 - Scouting: Procura ativa da prospeção de jogadores.Em síntese, a plataforma oferece a seguinte solução:  
  
Ações de utilizador  
  
- Criação e edição de uma conta pessoal.  
- Criação e edição de um perfil de utilizador (jogador)  
- Criação e edição de recomendações escritas nas páginas de perfil de jogador, treinador ou clube.  
- Votar em “skills” no perfil de um utilizador.  
- Adicionar conteúdo de multimédia (vídeos ou imagens) ao próprio perfil de utilizador.  
- Editar informação estatística dos jogos.  
- Validar informação estatística dos jogos.  
- Seguir um determinado perfil de utilizador.  
- Fazer uma pesquisa geral de todos os perfis de utilizador ativos na plataforma.   
- Fazer uma pesquisa filtrada de todos os jogadores ativos ativos na plataforma.  
- Como clube, criar um atleta privado e atribuir avaliações simples ou avançadas para esse mesmo atleta.  
- Como clube, criar um evento de treinos de captação.  
- Como clube, filtrar quais os perfis de utilizador podem ter acesso ao evento de captação e/ou recebem uma notificação do evento.  
- Comentar, gostar ou partilhar um conteúdo de “Media” na página de perfil de um utilizador ou na página de feed principal.

Ações automaticamente feitas pela plataforma:  
  
- Obtenção de todas as “Competições” de futebol nacional, independentemente do escalão.

- Obtenção de todas os clubes presentes nas competições obtidas.

- Obtenção de todos os jogadores presentes nas equipas obtidas.

- Obtenção de todos os jogos cuja “Competição” tenha sido previamente obtida  
- Obtenção constante de novos jogos e respetivos dados estatísticos.  
- Atualização de todos os dados estatísticos detetados nos jogos obtidos.  
- Atribuição de títulos (“Badges”) consoante um determinado acontecimento estatístico de um jogador.   
- Obtenção constante de informação de jogos futuros.  
- Atualização de todas as entidades com base na informação obtida acerca dos jogos futuros.  
- Atualização constante de rankings estatísticos e não estatísticos dos utilizadores.

Páginas da plataforma:  
  
- Inscrição/ Login.

- Criação de conta.   
- Feed de notícias.

- Perfil de jogador.   
- Edição de jogador.  
- Conteúdo “Media” de jogador.  
- Estatísticas de jogador.

- Perfil de treinador.   
- Conteúdo “Media” de treinador.  
- Estatísticas de treinador.

- Perfil de clube.   
- Conteúdo “Media” de clube.  
- Estatísticas de clube.  
- Plantel de clube.

- Jogo.   
- Pesquisa filtrada de jogadores.  
- Criação/ Edição de jogador privado de clube.  
- Lista de títulos (“Badges”).

- Lista de recomendações.

**Capítulo 3 – Porquê MEAN Stack?**

Escolha de DB:

Devido ao approach tomado desde o inicio do desenvolvimento, front to back, não se pode falar desta escolha sem levar em conta a interface visual, com as seguintes caracteristicas principais:

1 - Todas as paginas principais são altamente relacionais, no sentido em que, na maioria das entidades, temos presentes informações de outras.

2 - A procura customizada de entidades, muitas vezes envolve caracteristicas compostas Ex: somatorio de golos, golos por jogo (de jogadores e equipas).

3 - Conteudo dinamico em que a sua escala é pouco previsivel por ser uma rede social.

4 - A escolha de Angular, uma single-page application Framework. **- NodeJS versus outras linguagens**

**- Geral  
 - Event Driver  
 - Web Crawling  
 - Javascript/ Typed Script  
 - NoSQL (MongoDB) versus Modelos relacionais**

Face á emergencia de base de dados NoSQL, criadas com redes sociais e outros casos de uso Web 2.0, optamos pela escolha de MongoDB, uma das DB’s NoSQL mais maturas no mercado.

Como funciona MongoDB?

Como uma Base de dados orientada a Documentos, utiliza o conceito de dados e documentos auto-contidos e auto-descritivos, e isso implica que o documento em si já define como ele deve ser apresentado e qual é o significado dos dados armazenados na sua estrutura.

Ou seja, idealmente, por pagina irá apenas ser preciso 1 documento.

Com este conceito em mente, começamos por atacar as nossas principais características:

1 - Devido à informação relativa às restantes entidades ser, na sua maioria, dificilmente mutável (não se muda de nome todos os dias) e em pouca quantidade, a sua replicação não é, de todo, critica.

2 - As características compostas como somatórios e agregações estão auto contidas nos documentos, sendo a sua procura mais rápida, embora mais custosa em termos de actualização de base de dados.

3 - O ponto mais preocupante desta escolha no ponto de vista teórico, isto porque um documento pode ter a o tamanho máximo de 14MB, no ponto de vista prático isto não é tão realista, pois metendo em perspectiva:

O aclamado livro “A Guerra dos mundos “ por ​H. G. Wells​ de 287 páginas, tem apenas 400KB, e mesmo que uma bibliografia de um jogador seja superior, há métodos de divisão de documentos disponíveis.

4 - Quanto menos pedidos forem feitos á base de dados, mais responsive será uma single page application, por termos apenas que pedir 1 ou 2 documentos à base de dados invés de vários pedidos por informação relacional como em base de dados SQL, a nossa aplicação apresenta load times muito mais curtos.

Argumentos relativamente à optimização de queries e escalabilidade da base de dados:

-> O nosso conteúdo relacional é pouco clusterizado

Ex: os golos, assistências e cartões de um jogador,durante uma época, não ficam

armazenados em disco relativamente perto uns dos outros, num contexto de base de dados SQL.

Em MongoDB, todo esse conteúdo fica num único documento, evitando Fetch’s a várias secções do disco.

-> MongoDB tem um sistema de “sharding” que permite, com facilidade, uma escalabiilidade horizontal da base de dados.

**- Angular 5, Benefícios**

**Capítulo 4 – Web Crawler   
- Conceito geral   
- Tecnologias  
- Pros e Contras  
- Problemas encontrados e respetivas soluções  
(Especificação técnica ao detalhe apenas na arquitetura e descrição de componentes)**

Foi desenvolvido em NodeJS uma API que percorre o site *www.zerozero.pt* de forma a obter o máximo de dados de todos os jogadores, treinadores, clubes de futebol, resultados e classificações das primeiras divisões dos escalões sénior e últimos de formação do país.  
  
A API utiliza a biblioteca *Crawler-js* para ler os valores das páginas HTML do site acima referido, os valores são em seguida mapeados para os documentos (modelos) existentes na nossa base de dados em Mongo, através da biblioteca *Mongoose*.  
  
Para além de permitir assim uma sólida quantidade de informação dos utilizadores principais na plataforma independente de insumos dos mesmos, juntamente com o *dispatcher* - que determina os períodos de “fetch” de informação - é possível obter dados cuja necessidade de que estejam o mais atuais possível sejam constantemente atualizados em curtos períodos de tempo. Tais como resultados de jogos, estatística, entre outros.

**Escolha de Framework**

O processo de crawling está intimamente ligado á logica de negócio, estamos a falar de centenas de equipas e milhares de jogadores, com potencial de escalar varios niveis de grandeza no futuro, logo a framework tem que ser robusta e versatil.

A framework que escolhemos usar chama-se crawler-js:

-> Funciona a partir de uma fila de pedidos com varios workers, esta fila é altamente configuravel, podendo-se configurar proxies, tempos maximos entre pedidos, entre outros.

-> Pode-se usar a interface de JQuery para navegar pelo DOM, algo bastante util devido a nossa experiencia tecnologica com a mesma.

-> Open Source e sem custos.

**Hierarquia de Entidades e Paralelismo**

Devido à quantidade de páginas a ser crawled, optarmos por um processo que seja possivel paralelizar, seria um ideal, embora isto implique abordarmos vários problemas:

-> Atomaticidade das operações.

Em MongoDB as operaçoes sao atomicas ao nivel do documento, inclusive o lock do mesmo, logo qualquer operação, desde que seja atómica e auto contida por si ( ou seja, ausencia de SET’s, tudo a base de incrementaçoes e outras operaçoes sem conhecer o estado corrente )

-> Dependencias entre os modelos e Concorrência

Devido a Modularização dos modelos em si (nenhum destes preciso obrigatoriamente de outro para ter alguma identidade ), a hierarquia é feita pelo quão contidas estão as entidades com a subida do nível hierárquico.

Competição:

Contém poucas dezenas de equipas   
 Contém centenas de jogadores   
 Contém centenas de jogos

Equipa:

Contem dezenas de jogadores

Contem centenas de jogos mas de competiçoes diferentes

Jogo:

Contem 2 equipas

Dezenas de jogadores

Pertence apenas a 1 competição.

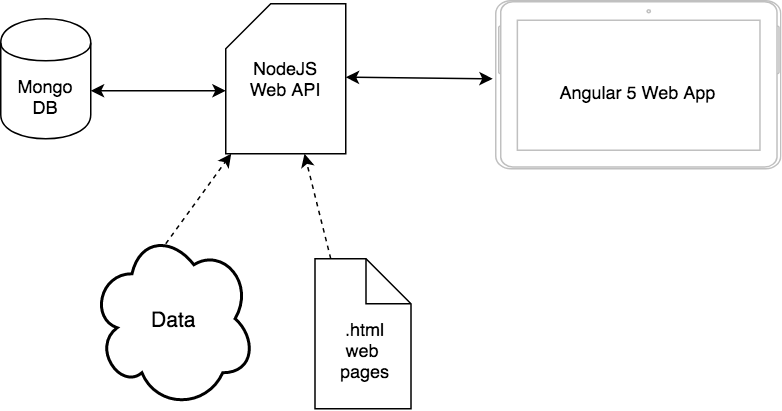
Os jogos serão crawled por base das competições (todos os jogos de uma competiçao, ao inves de todos os jogos de uma equipa), para garantir que quando os mesmos estão a ser processados, não exista um constante lookup da existencia dos jogadores ou equipas do mesmo, pois estes já vão estar devidamente processados, para isto acontecer modificamos a estrutura da pilha da framework para uma priority queue em que nenhum pedido de uma prioridade inferior pode começar sem serem completos todos os pedidos de hierarquias anteriores.

Como estamos a fazer crawling de modelos numa base de dados externa que funciona por REST, podemos usar os ID’s de todas as entidades como chaves da nossa procura, não sendo preciso reconhecer o ID na nossa base de dados, isto torna o processo de atualização dos modelos muito mais simples.

**Capítulo 5 – Gamification   
- Significado  
- Aplicação na Players Net  
 - Badges  
 - Skills  
 - Pesquisa filtrada (FM)   
- Benefícios**

**Capítulo 6 – Arquitetura   
- Geral**

**Arquitetura Geral do Projeto**

****  
 figura X

Tal como a maioria dos projetos aplicacionais, a PlayersNet divide-te em 3 camadas – Base de dados em MongoDB, Servidor em NodeJS, Front-end em Angular 5.   
Para além dos dados inseridos por parte dos utilizadores, o servidor está preparado para obter também informação de outras fontes (outras bases de dados ou outros sites).

**- Definição e explicação dos componentes (sem grande detalhe técnico)**Back End Node Architecture overview

Front End Angular Architecture overview

Angular is a platform and framework for building client applications in HTML and TypeScript. Angular is written in TypeScript. It implements core and optional functionality as a set of TypeScript libraries that you import into your apps.

The basic building blocks of an Angular application are NgModules, which provide a compilation context for components. NgModules collect related code into functional sets; an Angular app is defined by a set of NgModules. An app always has at least a root module that enables bootstrapping, and typically has many more feature modules.

Components define views, which are sets of screen elements that Angular can choose among and modify according to your program logic and data.

Components use services, which provide specific functionality not directly related to views. Service providers can be injected into components as dependencies, making your code modular, reusable, and efficient.

Both components and services are simply classes, with decorators that mark their type and provide metadata that tells Angular how to use them.

The metadata for a component class associates it with a template that defines a view. A template combines ordinary HTML with Angular directives and binding markup that allow Angular to modify the HTML before rendering it for display.

The metadata for a service class provides the information Angular needs to make it available to components through dependency injection (DI).

An app's components typically define many views, arranged hierarchically. Angular provides the Router service to help you define navigation paths among views. The router provides sophisticated in-browser navigational capabilities.

Modules

Angular NgModules differ from and complement JavaScript (ES2015) modules. An NgModule declares a compilation context for a set of components that is dedicated to an application domain, a workflow, or a closely related set of capabilities. An NgModule can associate its components with related code, such as services, to form functional units.

Every Angular app has a root module, conventionally named AppModule, which provides the bootstrap mechanism that launches the application. An app typically contains many functional modules.

Like JavaScript modules, NgModules can import functionality from other NgModules, and allow their own functionality to be exported and used by other NgModules. For example, to use the router service in your app, you import the Router NgModule.

Organizing your code into distinct functional modules helps in managing development of complex applications, and in designing for reusability. In addition, this technique lets you take advantage of lazy-loading—that is, loading modules on demand—to minimize the amount of code that needs to be loaded at startup.

For a more detailed discussion, see Introduction to modules.

Components

Every Angular application has at least one component, the root component that connects a component hierarchy with the page document object model (DOM). Each component defines a class that contains application data and logic, and is associated with an HTML template that defines a view to be displayed in a target environment.

The @Component() decorator identifies the class immediately below it as a component, and provides the template and related component-specific metadata.

Decorators are functions that modify JavaScript classes. Angular defines a number of decorators that attach specific kinds of metadata to classes, so that the system knows what those classes mean and how they should work.

Learn more about decorators on the web.

Templates, directives, and data binding

A template combines HTML with Angular markup that can modify HTML elements before they are displayed. Template directives provide program logic, and binding markup connects your application data and the DOM. There are two types of data binding:

Event binding lets your app respond to user input in the target environment by updating your application data.

Property binding lets you interpolate values that are computed from your application data into the HTML.

Before a view is displayed, Angular evaluates the directives and resolves the binding syntax in the template to modify the HTML elements and the DOM, according to your program data and logic. Angular supports two-way data binding, meaning that changes in the DOM, such as user choices, are also reflected in your program data.

Your templates can use pipes to improve the user experience by transforming values for display. For example, use pipes to display dates and currency values that are appropriate for a user's locale. Angular provides predefined pipes for common transformations, and you can also define your own pipes.

For a more detailed discussion of these concepts, see Introduction to components.

Services and dependency injection

For data or logic that isn't associated with a specific view, and that you want to share across components, you create a service class. A service class definition is immediately preceded by the @Injectable() decorator. The decorator provides the metadata that allows your service to be injected into client components as a dependency.

Dependency injection (DI) lets you keep your component classes lean and efficient. They don't fetch data from the server, validate user input, or log directly to the console; they delegate such tasks to services.

For a more detailed discussion, see Introduction to services and DI.

Routing

The Angular Router NgModule provides a service that lets you define a navigation path among the different application states and view hierarchies in your app. It is modeled on the familiar browser navigation conventions:

Enter a URL in the address bar and the browser navigates to a corresponding page.

Click links on the page and the browser navigates to a new page.

Click the browser's back and forward buttons and the browser navigates backward and forward through the history of pages you've seen.

The router maps URL-like paths to views instead of pages. When a user performs an action, such as clicking a link, that would load a new page in the browser, the router intercepts the browser's behavior, and shows or hides view hierarchies.

If the router determines that the current application state requires particular functionality, and the module that defines it hasn't been loaded, the router can lazy-load the module on demand.

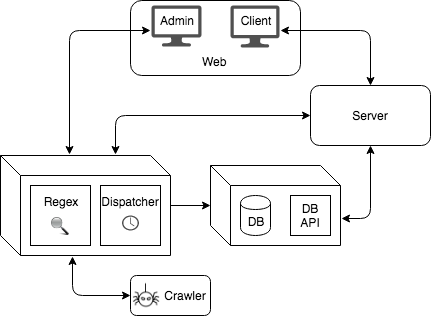
The router interprets a link URL according to your app's view navigation rules and data state. You can navigate to new views when the user clicks a button or selects from a drop box, or in response to some other stimulus from any source. The router logs activity in the browser's history, so the back and forward buttons work as well.

To define navigation rules, you associate navigation paths with your components. A path uses a URL-like syntax that integrates your program data, in much the same way that template syntax integrates your views with your program data. You can then apply program logic to choose which views to show or to hide, in response to user input and your own access rules.

****

Na camada de apresentação, apesar de se tratar de desenvolvimento front-end, como está a ser desenvolvida em Angular 5 existe um padrão de desenho bem explícito na sua estrutura.  
  
Sendo que esta camada se pode dividir em “controladores”, “modelos” e “vistas”, a estrutura Angular é muitas vezes confundida com um padrão de desenho MVC ou MVVM. No entanto, não há por defeito controladores ou *ViewModels* no Angular realmente, há sim componentes que derivam de *templates*, serviços e modelos.  
  
Posto isto, na camada de front-end tem-se uma subcamada de modelos que representam os que se encontram na base de dados, outra de serviços onde se encontram os pedidos ao servidor que mapeiam os modelos e também outros serviços como os de lógica de autenticação e autorização, e, por fim, a subcamada visual que, com o Angular, nos permite dividir por cada componente diferente (i.e. player-component, team-component, header-component, footer-component, chat-component, skills-component) especificando um ficheiro .html, .css e .ts (este último funciona como a classe da componente) por cada um deles tornando possível ter características bastante específicas por componente.   
  
Olhando para a estrutura completa desta camada, pode-se notar uma orientação a objetos subtil na forma como a mesma é implementada.

**Capítulo 7 – Descrição técnica  
- Detalhe técnico explícito de todos os componentes de Back-End**

****

Web  
Componente Front-End da aplicação que, não só acarreta a *user interface* do projeto (client), mas também páginas que permitem a configuração específica de alguns eventos (admin).   
  
Desta forma, alterações de configuração tais como:   
 - O intervalo de tempo em que corre o crawler (comunicação com o *dispatcher*),  
 - Critérios de prémios dos jogadores (comunicação com o *regex*),   
 - Critérios de atribuição de badgers a jogadores ou treinadores (comunicação com o *regex*)   
  
passam a poder ser feitas por um utilizador administrador.

Dispatcher  
Componente responsável por lançar eventos automáticos que o *crawler* irá consumir. Para além disso, é no *dispatcher* que se encontram as ações que determinam as configurações definidas pela componente *admin –* referidas na componente anterior e é nesta componente que é são escritos na base de dados os valores obtidos pelo crawler. O *dispatcher* está também encarregue por lançar eventos detetados na informação obtida pelos crawlers (como por exemplo, um jogador marcou um hat-trick, ou é a décima vitória consecutiva do clube XPTO). A deteção da maioria desses eventos é definida pela componente *regex*.

Regex  
Nesta componente encontra-se toda a lógica que determina as condições necessárias para que um evento seja despoletado.   
De forma a reduzir a quantidade de condições necessária para que se capture os inúmeros eventos pretendidos (tais como 3 golos marcados num jogo, 5º jogo consecutivo sem marcar golos, 34º jogo consecutivo sem perder, etc.) optou-se por atribuir a cada entidade (clube, jogador ou treinador) um texto codificado que identifica diferentes eventos, por exemplo, “G” representa um golo, “W” representa uma vitória e “L” representa uma derrota.   
Assim, através de um sistema de regular expressions que vai percorrer os enormes textos dos utilizadores em questão, obtêm-se as respostas pretendidas e consegue-se detetar a ocorrência dos eventos definidos.

Crawler  
Componente responsável por recolher dados automaticamente de outros sites e retorná-los ao *dispatcher*. A única informação que o *crawler* recebe são dados de configuração e estes vêm por parte do *dispatcher*.

DB Api  
Componente que trata de toda a comunicação com a base de dados. Esta componente recebe os valores que o *dispatcher* recebe do crawler, os valores que o *dispatcher* recebe dos administradores, envia (quer sejam os valores automáticos, quer os que vêm da componente web).

Nao vou ter tempo de fazer a DB API, e ainda o Regex, Dispatcher e Crawler, vao estar no mesmo "Quadrado" onde os dados fluem bilateralemente ( Ao contrario de 1 way como está acima).

**Logo vais ter que alterar a imagem para reflectir isto.**

Se achas que vais ter tempo de fazer qualquer coisa no admin front-end, we keep things as they are, se nao, apaga todas as mentions de tal feature.

DB  
Base de Dados MongoDB.  
(Aqui vou meter os JSONs todos)

Server  
Componente servidora principal. Contém todos os endpoints da aplicação (Web Api) e é responsável por todo o processamento de Back-End da aplicação, tais como: autenticação, autorização, mensagens, envio automático de emails e comunicação com a componente de armazenamento de dados.

**Web Crawler**

Foi desenvolvido em NodeJS uma API que percorre o site [*www.zerozero.pt*](http://www.zerozero.pt/) de forma a obter o máximo de dados de todos os jogadores, treinadores, clubes de futebol, resultados e classificações das primeiras divisões dos escalões sénior e últimos de formação do país.  
  
A API utiliza a biblioteca *Crawler-js* para ler os valores das páginas HTML do site acima referido, os valores são em seguida mapeados para os documentos (modelos) existentes na nossa base de dados em Mongo, através da biblioteca *Mongoose*.  
  
Para além de permitir assim uma sólida quantidade de informação dos utilizadores principais na plataforma independente de insumos dos mesmos, juntamente com o *dispatcher* - que determina os períodos de “fetch” de informação - é possível obter dados cuja necessidade de que estejam o mais atuais possível sejam constantemente atualizados em curtos períodos de tempo. Tais como resultados de jogos, estatística, entre outros.  
  
(Esta componente encontra-se em fase de teste)

**- Detalhe técnico explícito de todos os componentes de Front-End**Models

Guards  
 - Administrator (Explicar Processo JWT através de um Post para o servidor)  
 - Authorization (Explicar Processo de validação da Token – envio da token corrente em todos os pedidos que assim o exigem)  
  
Helpers  
  
Modals  
Esta pasta inclui os “Modals” principais da plataforma. Os “Modals” são páginas que se sobrepõem a outras páginas de HTML principais com um conteúdo reduzido. Normalmente contém uma seleção de elementos, adição de informação ou ficheiros ou uma visualização especial de algum elemento.  
Funciona exatamente como um componente (Component) e possui uma estrutura igual.  
  
 - Recommendation Modal  
 Modal responsável pela criação ou edição de uma recomendação.  
 Os componentes *Team* e *User Info Profile* utilizam este Modal.  
   
 - Team Player Evaluation modal  
 Modal responsável pela criação ou edição de uma avaliação avançada de um jogador (Shadow Player).  
 Este Modal é apenas utilizado pelo componente *Team*.

- Tryout Modal  
 Modal responsável pela criação ou edição de um treino de captação.  
 Este Modal é apenas utilizado pelo componente *Team*.

Modules

Angular apps are modular and Angular has its own modularity system called NgModules. NgModules are containers for a cohesive block of code dedicated to an application domain, a workflow, or a closely related set of capabilities. They can contain components, service providers, and other code files whose scope is defined by the containing NgModule. They can import functionality that is exported from other NgModules, and export selected functionality for use by other NgModules.

Every Angular app has at least one NgModule class, the root module, which is conventionally named AppModule and resides in a file named app.module.ts. You launch your app by bootstrapping the root NgModule.

While a small application might have only one NgModule, most apps have many more feature modules. The root NgModule for an app is so named because it can include child NgModules in a hierarchy of any depth.  
  
Metadata  
  
An NgModule is defined by a class decorated with @NgModule(). The @NgModule() decorator is a function that takes a single metadata object, whose properties describe the module. The most important properties are as follows.

declarations: The components, directives, and pipes that belong to this NgModule.

exports: The subset of declarations that should be visible and usable in the component templates of other NgModules.

imports: Other modules whose exported classes are needed by component templates declared in this NgModule.

providers: Creators of services that this NgModule contributes to the global collection of services; they become accessible in all parts of the app. (You can also specify providers at the component level, which is often preferred.)

bootstrap: The main application view, called the root component, which hosts all other app views. Only the root NgModule should set the bootstrap property.  
  
  
Services  
  
Service is a broad category encompassing any value, function, or feature that an app needs. A service is typically a class with a narrow, well-defined purpose. It should do something specific and do it well.

Angular distinguishes components from services to increase modularity and reusability. By separating a component's view-related functionality from other kinds of processing, you can make your component classes lean and efficient.

Ideally, a component's job is to enable the user experience and nothing more. A component should present properties and methods for data binding, in order to mediate between the view (rendered by the template) and the application logic (which often includes some notion of a model).

A component can delegate certain tasks to services, such as fetching data from the server, validating user input, or logging directly to the console. By defining such processing tasks in an injectable service class, you make those tasks available to any component. You can also make your app more adaptable by injecting different providers of the same kind of service, as appropriate in different circumstances.  
  
Portanto, nesta pasta encontram-se todos os serviços utilizados pela plataforma que comunicam diretamente com a componente servidora em NodeJS. Aqui poder-se-iam encontrar web services REST ou SOAP também, mas a plataforma delega quaisquer comunicações com o “exterior” à componente servidora.  
Apesar dos web-services serem internos, as chamadas são feitas em client-side, portanto é sempre feito um pedido http para aceder a um serviço na componente servidora.   
  
Toda a comunicação com os serviços do servidor é feita através de promessas (Promises).  
  
 - Authentication Service  
 Nesta classe encontram-se todos as chamadas aos web services responsáveis pela lógica de autenticação em cada página da plataforma. Incluindo o Login e Logout.

- Competition Service  
 Classe responsável por comunicar com os web-services que despoletam ações sobre uma determinada competição.

- Generic User Service

Classe responsável por comunicar com os web-services que efetuam pesquisas simples ou avançadas sobre todos os utilizadores existentes.

- Match Service

Classe responsável por comunicar com os web-services que despoletam ações sobre um determinado Jogo.

- Team Service

Classe responsável por comunicar com os web-services que despoletam ações sobre uma determinada Equipa.

- User Service  
 Esta classe é única e exclusivamente responsável por comunicar com os web-services alusivos ao utilizador corrente.

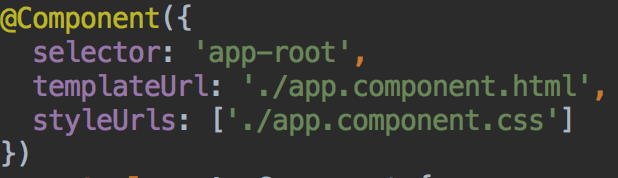
- User Info Service  
 Classe responsável por comunicar com os web-services que despoletam ações sobre um determinado Jogador.

Validators  
  
Components

Um componente contém uma “view” em html (e respetiva classe de estilos .css) e o seu controlador.   
  
É definida a lógica do componente — o que faz para suportar a view—numa classe. A classe interage com a “view” através de uma API com propriedades e métodos.

No nosso caso, todos os componentes têm um objeto “viewModel” que é alimentado pelo serviço presente no construtor que é fornecido através de “Dependency injection”.  
  
O Angular cria, atualiza e destrói componentes à medida que o utilizador se move na aplicação. São despoletadas ações durante o período ativo do componente através das funções que são chamadas em diferentes alturas do ciclo de vida do mesmo, como por exemplo o ngOnInit(), o ngAfterInit() ou o ngOnDestroy().

Todos os componentes são definidos no corpo da sua classe .ts através de uma anotação igual ou semelhante à seguinte:

  
Figura X

As configurações da figura X são as do componente App, aqui seguem exemplos das opções de configuração de componentes mais úteis e comuns:

selector: Um seletor CSS seletor que diz ao Angular para criar e inserir uma instância do componente em questão sempre que encontrar a “Tag” correspondente num template HTML.

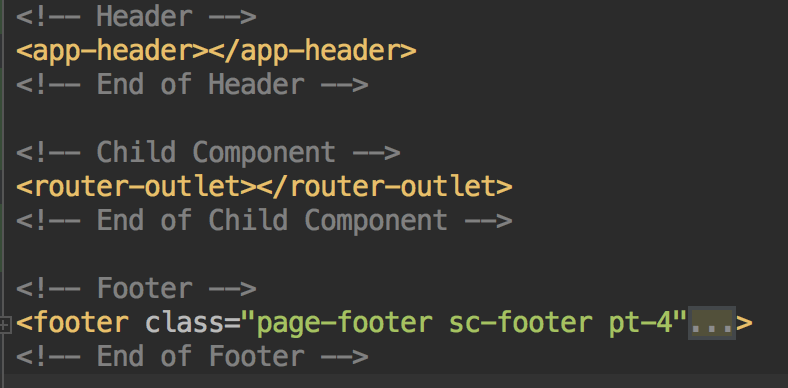
templateUrl: O endereço “module-relative” do template HTML deste componente. Este template define a view do componente.

providers: Um array de “providers” para os serviços que o componente requer. Desta forma, este array diz ao Angular como providenciar instâncias de serviços ao componente cujo construtor usará para obter os dados que necessita e apresentá-los através da sua “view”.

Os Componentes Angular da aplicação

App

Este é o componente core da aplicação.  
Aqui se insere a “View” presente em todas as páginas do projeto, constituída pelo “Header” e “Footer”, sendo que o “Header”, devido ao seu nível de complexidade, tem um outro componente individual que o representa.

  
Figura X   
  
  
Na figura X está representada a classe app.component.html.   
A “Tag” <app-header> representa o componente “Header”, portanto, este será invocado na construção da view da componente “app” e a sua classe .html inserida, pela ordem apresentada, nesta classe.  
O mesmo se passa com a “Tag” seguinte - <router-outlet> que, neste caso, representa a “view” do componente invocado pelo “router”.

Header

No componente de cabeçalho (Header), insere-se a lógica de Login (através de outro componente independente), de pesquisa de utilizadores, visualização e navegação das notificações do utilizador e visualização e edição dos dados de perfil do utilizador.

O Login aparece apenas quando não é verificado nenhuma “token” autorizada a estar na página requisitada, ou seja, quando ainda não foi feito nenhum login ou que a “token” tenha expirado.

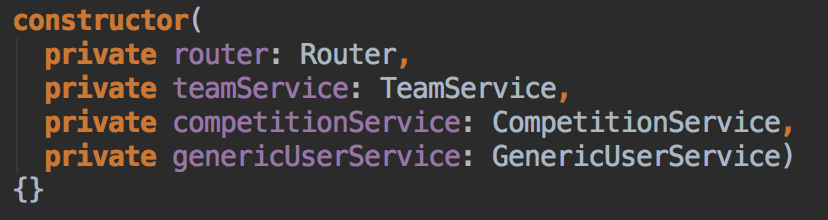
Home

Apesar de ser no componente Create Account onde se cria uma conta, é no componente Home que se captura o primeiro grupo de informação do novo utilizador.

Login

Componente responsável por efetuar o Login na aplicação através da obtenção das credenciais do utilizador.

Create Account

Este componente é responsável pela lógica e apresentação da “User interface” da criação de um novo perfil de utilizador.  
  
  
Figura X  
  
Tal como o seu construtor, representado na figura X indica, indica, este componente consome os serviços Team, Competition e Generic User para criar a sua lógica aplicacional e alimentar a sua “view”.

Competition

Este componente representa a entidade “Competição” da plataforma.  
A sua “view” é alimentada pelo seu controlador “Competition.component.ts” que, por sua vez, consome os serviços de da classe “Competition.Service.ts”.

Edit User Info

Filter User Info

Match

Team

Team Media

Team Player

Team Profile

Team Roster

Team Stats

Tryout Modal

User Info Profile

User Info

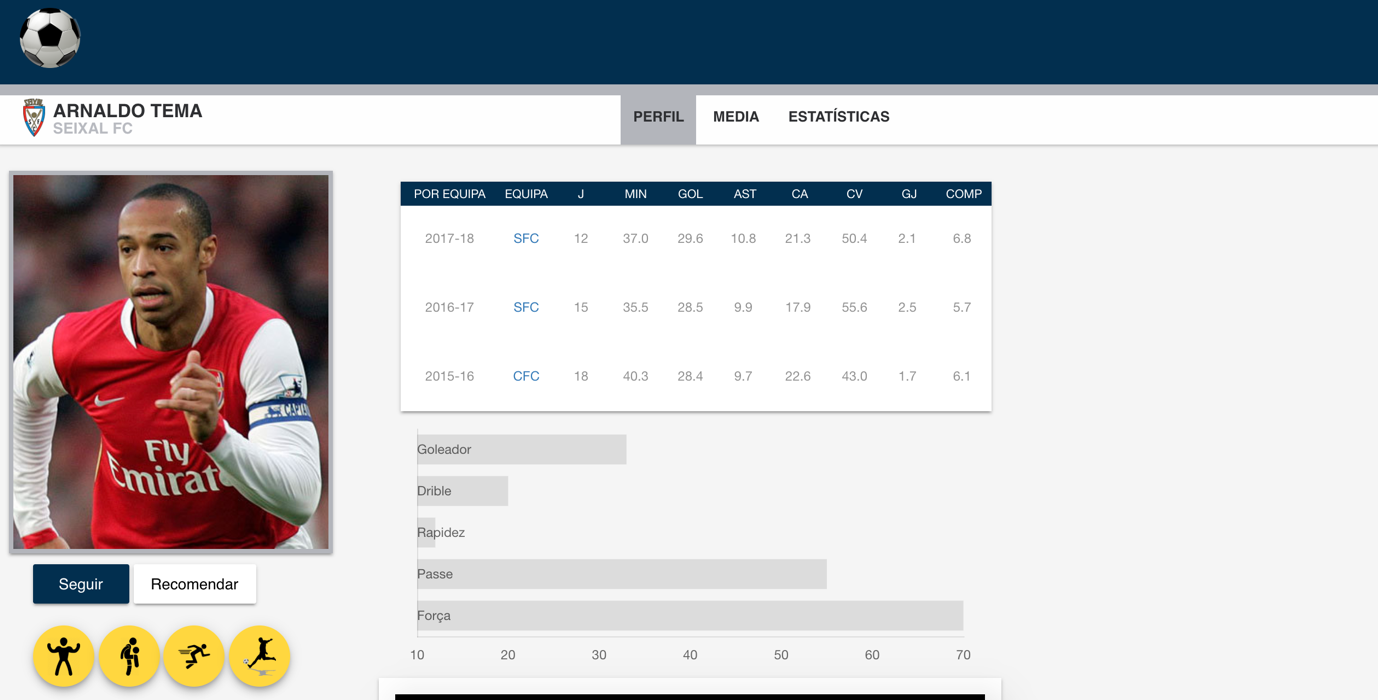
User Info Media

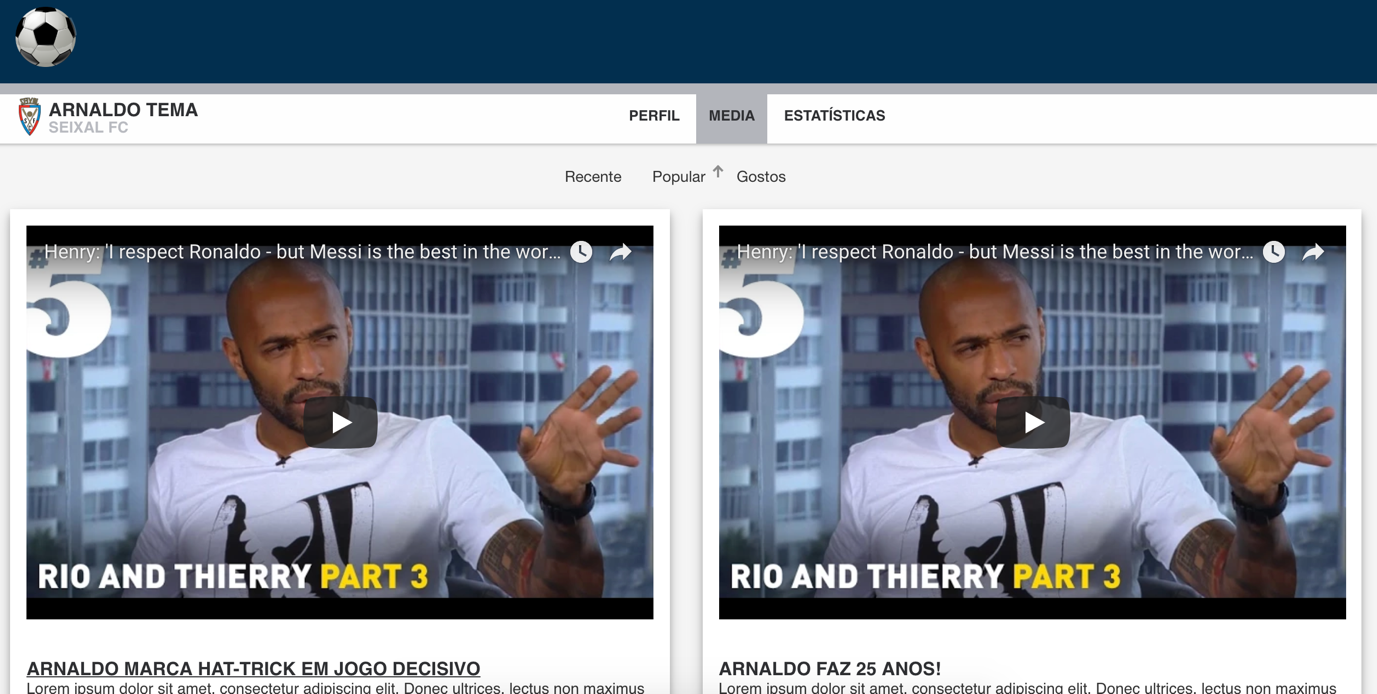
User Info Stats

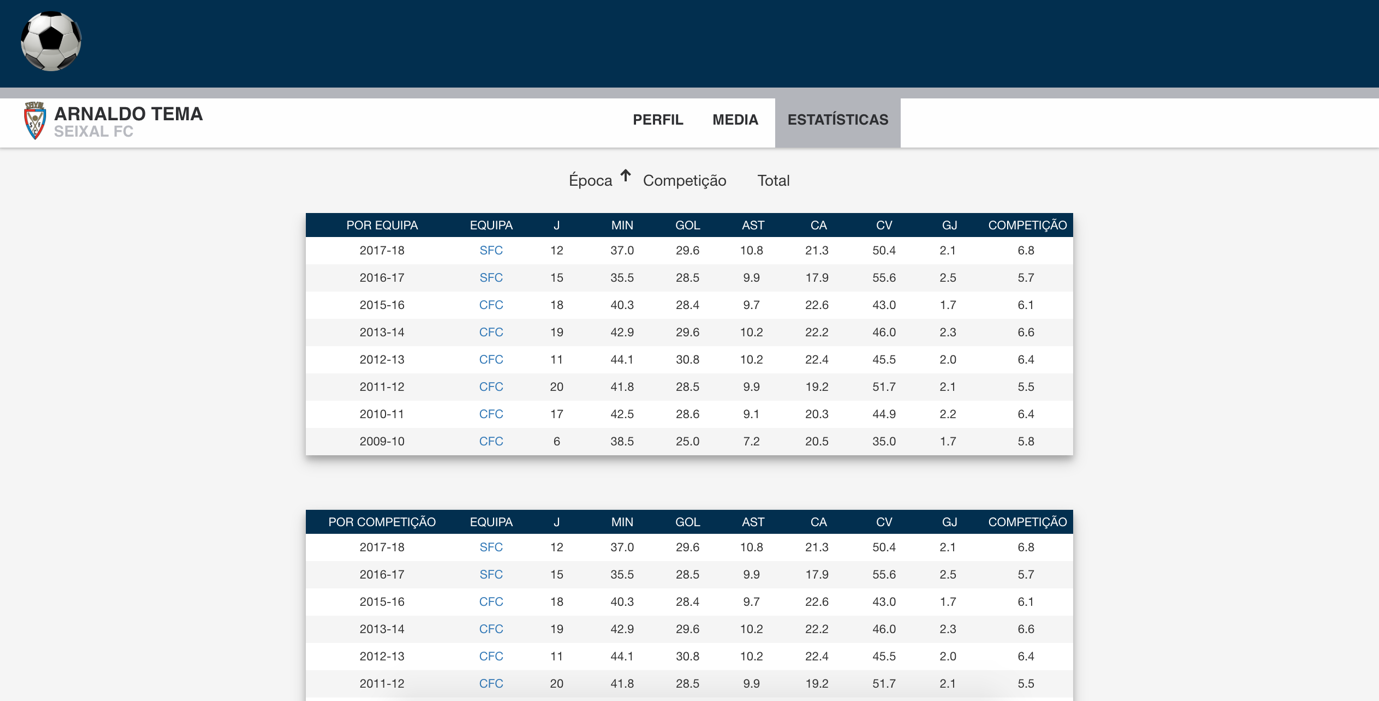
**Capítulo 9 – Descrição funcional  
- Manual de utilizador com exemplos ilustrativos da plataforma.**

**Interface da plataforma**(Todos os dados são fictícios e nenhuma das páginas está no seu estado final)

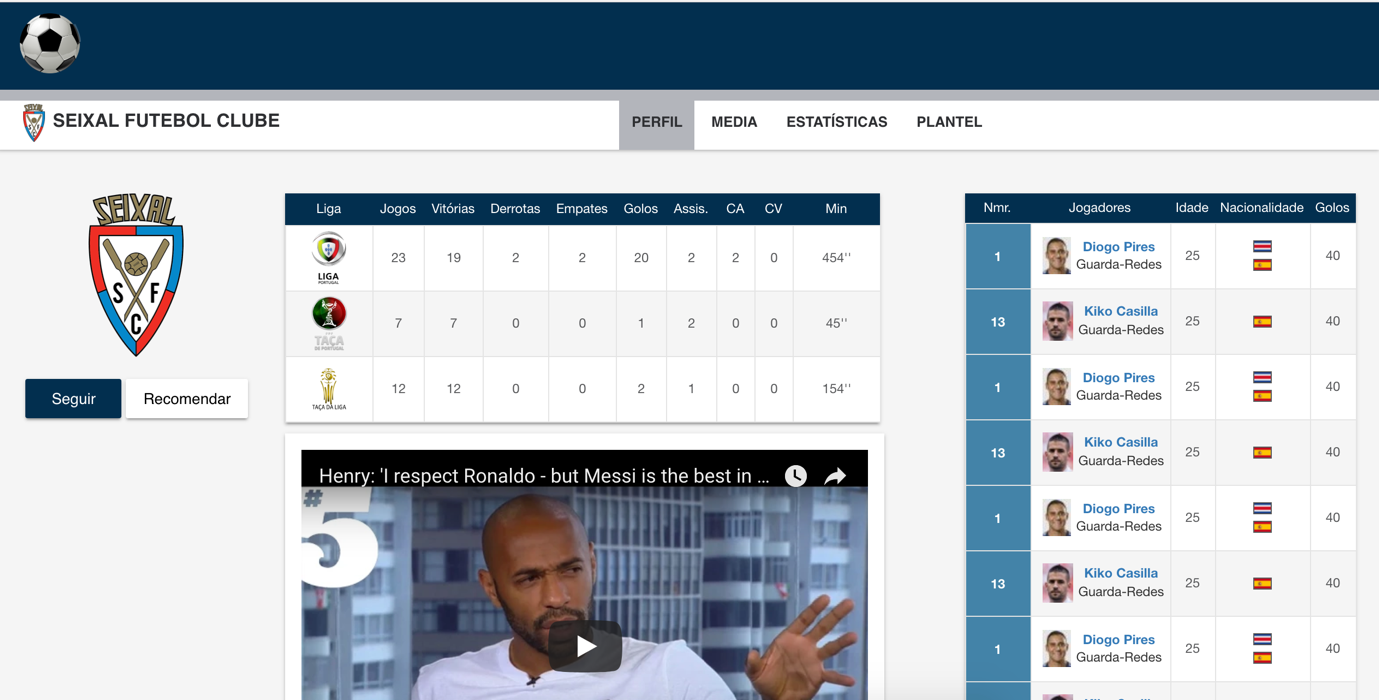
Página de perfil de jogador

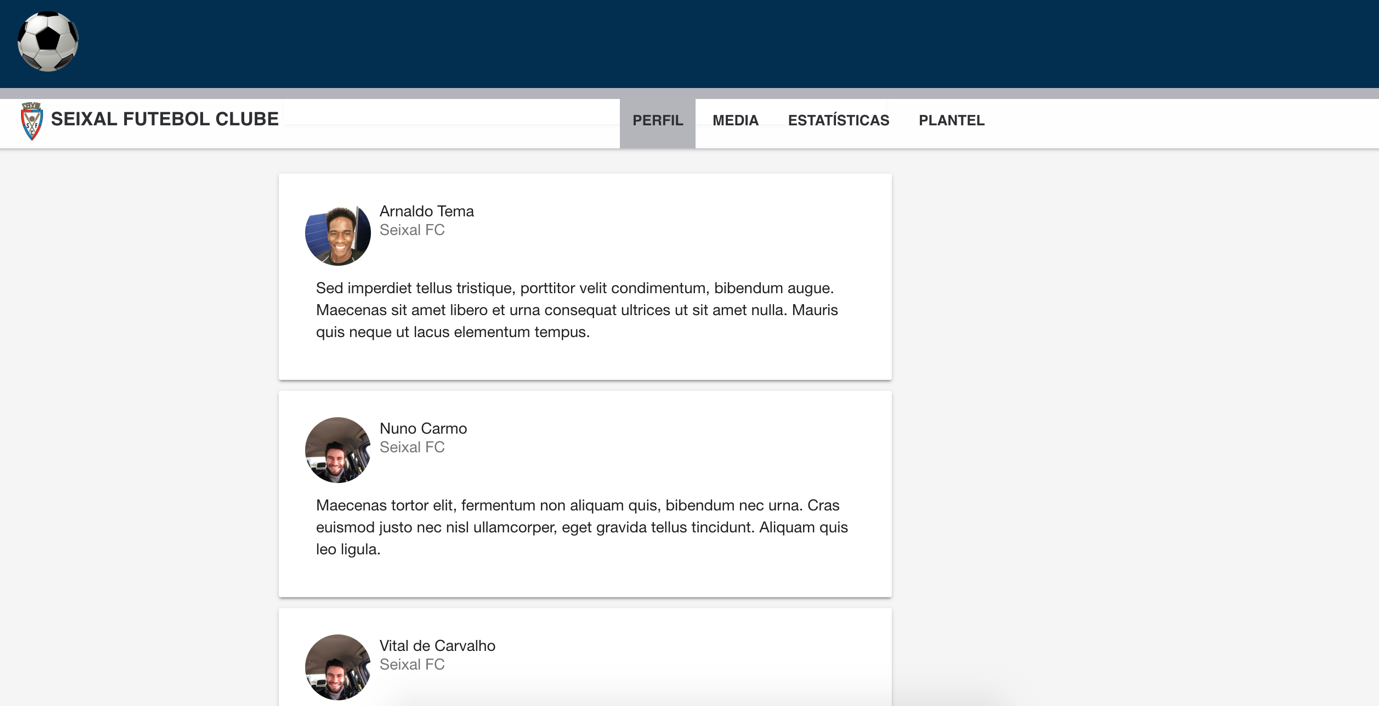
****

Página de media (vídeos e fotos) do jogador(Neste exemplo, ordenada pelas notícias mais populares) ****

Página das estatísticas do jogador(Dados fictícios) ****

Página de perfil de equipa

****

Área de recomendações escritas ****

**Capítulo 10 – Conclusões  
- Lições aprendidas com o desenvolvimento da plataforma.  
- O que é que poderia/deveria ter sido diferente.**

**Problemas**

**Alteração da arquitetura**

Como consta no capítulo que explica a arquitetura do projeto, houve uma significativa alteração do que tinha sido inicialmente definido que iria ser a estrutura da aplicação.  
  
A primeira abordagem encontrava-se pouco dividida, o que acrescia o número de dependências entre componentes, tornava a aplicação pouco suscetível a alterações futuras que, num projeto desta categoria, tende a acontecer à medida que se vai desenvolvendo e havia componentes (como o *crawler*) que eram responsáveis por vários tipos de processamento.  
  
Na abordagem corrente já se pode encontrar uma arquitetura mais segmentada, bastante sensível a alterações e melhorias sem que interfira com a lógica funcional do projeto, com componentes com propósitos bem definidos e de melhor compreensão aos olhos de quem tenta interpretar a lógica do projeto.  
  
No entanto, esta alteração, bem como a definição dos modelos da base de dados não relacional (MongoDB funciona apenas por referências entre entidades), foram causa de uma demora maior nesta fase de desenvolvimentos, o que trará consequências nos tempos estimados para outros desenvolvimentos.

**Dependência dos Sites para os Crawlers**

Tal como se verifica nas explicações do funcionamento do *web cralwer* desta aplicação, há uma **enorme** dependência das fontes do mesmo para que o funcionamento saia ileso e com sucesso. Ou seja, basta que uma das páginas lidas pelo *crawler* seja alterada para que o mesmo deixe de funcionar corretamente.  
  
Para solucionar este problema, foi necessário fazer uma análise da probabilidade da alteração das páginas em questão junto com os administradores do site das mesmas da qual se concluiu que, apesar de muito remota, existe obviamente essa possibilidade. Portanto, terá de ser criado no *dispatcher* um sistema de Logging que notifique essas alterações para que rapidamente se possa atualizar o crawler em função das mesmas. Não sendo esta uma definitiva, o objetivo é que, futuramente, se consiga obter todos os dados diretamente da mesma fonte onde o site de onde, neste momento, se está a extrair informação a consegue obter.

**Tempo para o desenvolvimento Front-End**

Seguindo a explicação no primeiro problema apontado neste capítulo, o tempo escasseia-se para alguns desenvolvimentos que ainda restam acontecer.   
  
Visto que todo o projeto carece de um workflow altamente elaborado, os desenvolvimentos de back-end não poderão, de forma alguma, sair prejudicados, pondo em causa o funcionamento geral da aplicação. Logo, se realmente houver necessidade de abdicar de alguns desenvolvimentos, será a camada de apresentação a sacrificada.   
  
Apontando sempre para uma apresentação perfeitamente funcional e apresentável, a versão final deste projeto poderá ficar apenas menos “bonita” do que uma eventual versão futura onde haja uma maior alocação de horas de desenvolvimento para a componente visual.

**- Em que medida foram atingidos os objetivos.  
- Trabalho futuro.**

**Referências**

**Apêndices  
- Listagem de programas.  
- Definição tecnológica da plataforma.**