



RugbyApp

Rui Garcia
João Ferreira

Orientador: Jorge Martins

Relatório final realizado no âmbito de Projeto e Seminário
Licenciatura em Engenharia Informática e de Computadores

setembro de 2020

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

RugbyApp

40539 Rui Miguel Marques Garcia

40913 João Carlos Máximo Ferreira

Orientador Jorge Manuel Rodrigues Martins Pião

Relatório final realizado no âmbito de Projeto e Seminário
Licenciatura em Engenharia Informática e de Computadores

setembro de 2020

Resumo

O Rugby é um desporto que representa uma grande presença no quotidiano dos membros do nosso grupo, por sermos ou conhecermos praticantes ativos, e experienciarmos a maioria das vertentes deste desporto há um longo período de tempo. A falta de presença deste desporto no conceito geral da nossa cultura não o expõe a tanto apoio e auxílio como noutros desportos de grande renome, o que se constata de forma clara, sem a necessidade de uma busca intensiva.

A ideia deste projeto nasceu da nossa própria necessidade de criar uma aplicação que preencha essa lacuna. Com o foco primário em trazer às equipas deste desporto uma aplicação virada para a organização e gestão de informação dentro das equipas, esperamos no fim apresentar uma ferramenta que consiga aglomerar os aspetos principais da gestão de equipas de rugby num único sítio, e que proporcione um apoio extra à maioria das suas necessidades funcionais.

Agradecimentos

Agradecemos às equipas técnicas do Belas Rugby Clube e do Sporting Clube de Portugal pela disposição para partilha de ideias e comentários, afim de enriquecer e melhorar o nosso projeto. Agradecemos também ao engenheiro Jorge Martins por se disponibilizar para ser o orientador do nosso projeto.

Índice

1	Introdução	1
1.1	Enquadramento	1
1.2	Objetivos Funcionais	1
1.3	Organização do documento	2
2	Formulação do Problema	3
2.1	Formulação	3
2.2	Especificações Funcionais	3
2.2.1	Especificações Principais	3
2.2.2	Especificações Secundárias	4
2.3	Arquitetura da Solução	5
3	Aplicação Servidora	6
3.1	Introdução e Estrutura da Aplicação Servidora	6
3.1.1	<i>Model</i>	6
3.1.2	<i>Repository</i>	7
3.1.3	<i>Business</i>	8
3.1.4	<i>Controller</i>	9
4	Aplicação Cliente	11
4.1	Introdução e Estrutura da Aplicação Cliente	11
4.1.1	<i>Angular Component e NgModules</i>	11
4.1.2	<i>Angular Service</i>	12
4.1.3	<i>Angular Data Binding</i>	14
4.1.4	<i>IONIC API</i>	15
4.1.5	Angular Materials CDK	19
4.1.6	Classes e Entidades	21
4.1.7	IONIC Lifecycle	22
4.1.8	<i>Chart.js</i>	23

5	Testes	24
5.1	Aplicação Servidora	24
5.2	Aplicação Cliente	26
6	Conclusões (até agora)	27
6.1	Recapitulação	27
	Referências	28
A	Diagrama de Entidades	29
B	Árvore de Navegação	30
C	Informação do Dossiê de Projeto	31

Lista de Figuras

2.1	Arquitetura da nossa solução.	5
4.1	Vista inicial da aplicação. Observa-se o <i>ion-split-pane</i> a azul, o <i>ion-tabs</i> a vermelho e o <i>ion-header</i> a verde.	15
4.2	<i>Modal</i> da página do formulário de <i>Practice</i>	16
4.3	<i>Popover</i> da página <i>Event</i>	17
4.4	<i>Select</i> da página do formulário de <i>Event</i>	17
4.5	Tabela da página de <i>Games</i> , ordenada decrescentemente por nome.	20
4.6	<i>Profile</i> de um <i>Staff</i> , com as linhas da grelha <i>highlighted</i> para fins de visualização.	21
4.7	Aba <i>Graph</i> , com as estatísticas <i>Fouls</i> , <i>Errors</i> , <i>YellowCards</i> , <i>RedCards</i> , <i>turnOvers</i> , <i>tries</i> e <i>mauls</i> selecionados.	23
4.8	Aba <i>Graph</i> , com as estatísticas todas selecionadas. Quando o utilizador filtra os gráficos para apenas um atleta, a aplicação gera um gráfico de barras. . . .	23
5.1	Demonstração de todos os testes da aplicação servidora a concluírem com sucesso.	26
A.1	Diagrama de Entidades	29
B.1	Árvore de Navegação	30

Capítulo 1

Introdução

Este projeto foi desenvolvido no âmbito de Projeto e Seminário, no semestre de Verão de 2020 da Licenciatura em Engenharia Informática e de Computadores. Este capítulo está organizado em três secções que descrevem o enquadramento e objectivo do projeto, assim como a organização do documento.

1.1 Enquadramento

Este projeto tem como temática principal o desporto Rugby. Sendo uma atividade desportiva pouco reconhecida ou relevante no contexto da nossa cultura, é notória a falta de ferramentas que proporcionem suporte à prática desta atividade. Este problema foi apresentado aos diversos elementos do grupo devido a ligações interpessoais entre estes e o desporto, experienciando ativamente e lidando com este problema no próprio quotidiano.

Apesar de existir ferramentas que proporcionam uma melhor qualidade na organização e prática desta modalidade, estas são escassas e dispendiosas, pelo que é objetivo deste projeto criar uma aplicação que assente na ideia de ajudar ativamente equipas técnicas desta modalidade desportiva em vários temas relevantes à otimização e melhoria do desempenho da equipa ao longo da época desportiva.

1.2 Objetivos Funcionais

Esta secção aborda os objetivos funcionais principais e secundários. É de notar que entre a proposta de projeto inicial e o corrente relatório ocorreram diversas reuniões com as equipas técnicas do Belas Rugby Clube e Sporting Clube de Portugal, pelo que é notável alguma diversidade de objetivos entre ambos os documentos.

A ideia chave deste projeto é criar uma aplicação que seja capaz de recolher e analisar estatisticamente dados sobre o desempenho dos jogadores de uma equipa de Rugby, com o propósito de monitorizar aspetos críticos que avaliem não só o estado individual de cada jogador, mas também o estado atual de toda a equipa. Pressupõe-se que toda a informação

recolhida consiga facilitar aspetos chaves do funcionamento de uma equipa desportiva, auxiliando desde a face tática do desporto (aspetos como a constituição da equipa, o plano tático, a otimização de temáticas de treino), às faces menos técnicas das equipas (aspetos como a organização de treinos e jogos, a facilidade de acesso a informação pertinente, entre outros).

Pretende-se aglomerar todos estes aspetos numa única aplicação, que ofereça aos utilizadores, sendo eles atletas ou equipa técnica, uma plataforma onde possam observar os dados agregados referentes a cada jogador em particular, os dados referentes a jogos concretos ao longo da época, aceder a planos de treinos físicos propostos pela equipa técnica, e observar um calendário de eventos futuros. No que toca à equipa técnica, esta também terá a funcionalidade de adicionar jogos, seleccionar jogadores em contexto destes jogos (conceitos como lista de convocados, jogadores titulares, jogadores suplentes), assim como ter acesso a uma interface gráfica onde seja possível consultar estatísticas de atletas, oferecendo uma percepção detalhada do desempenho dos atleta nos jogos.

As funcionalidades são explicadas em mais detalhe na secção 2.2.

1.3 Organização do documento

O restante relatório encontra-se organizado da seguinte forma.

Capítulo 2	Formulação do Problema Formulação e Contextualização do Problema, Especificações Funcionais e Arquitetura da Solução.
Capítulo 3	Aplicação Servidora Aspetos relacionados com a aplicação servidora, como abordagens, metodologias e detalhes de implementação.
Capítulo 4	Aplicação Cliente Aspetos relacionados com a aplicação cliente, como abordagens, metodologias e detalhes de implementação.
Capítulo 5	Testes Testes executados sobre as diversas vertentes do projeto.
Capítulo 6	Conclusões Recapitulação das observações e conclusões importantes.

Capítulo 2

Formulação do Problema

Este capítulo está organizado em três secções, onde se descreve a formulação do problema e as suas especificações funcionais, assim como a arquitectura da solução.

2.1 Formulação

Esta secção aborda todos os aspetos referentes à Formulação do Problema.

Tema: Aplicação de Suporte a Equipas de Rugby
Problema : As equipas técnicas de Rugby têm ferramentas de suporte à sua organização?
Que aspetos são necessários implementar numa aplicação para garantir esse suporte?

A hipótese de resposta a esta pergunta foi adquirida da noção pessoal dos estudantes, como indivíduos com ligações interpessoais com o desporto, e das ideias que resultaram do diálogo com as duas equipas referidas neste documento. Após diversas reuniões com foco na recolha de ideias, foi atingida a hipótese de resposta descrita na secção 1.2. Apesar do problema apresentar algum teor subjetivo (equipas distintas operam e organizam-se de formas distintas, e sentem necessidades distintas em fatores distintos), foi possível alcançar uma solução que aglomera os fatores mais importantes para garantir a utilidade e a cobertura necessárias no contexto desta aplicação.

2.2 Especificações Funcionais

Esta secção enumera as especificações funcionais da nossa solução, separando-as em especificações principais e especificações secundárias.

2.2.1 Especificações Principais

A primeira sub-secção desta secção lista todos os conceitos chave que se pretendem desenvolver como especificações principais:

1. Perfil de Atleta;
2. Perfil de Equipa Técnica;
3. Jogo;
4. Estatísticas de Jogo;
5. Treino;
6. Planos de Treinos Físicos;
7. Calendário de Eventos;
8. Torneio;
9. Evento;

Estes conceitos definem a estrutura da nossa aplicação.

A nossa aplicação irá implementar um perfil de Atleta, onde se pode observar a informação correspondente do atleta, como a idade, peso, altura, posições, assim como as suas estatísticas ao longo da época. Também será possível observar uma lista dos jogos onde foi convocado, ligações para as suas estatísticas nos mesmos, e uma lista de treinos e eventos a que compareceu. A aplicação irá também implementar perfis dedicados aos integrantes das equipas técnicas, para adicionar alguma coesão sobre a informação global da equipa.

A nossa aplicação irá implementar um menu de jogo, com as estatísticas da equipa no contexto desse jogo, os jogadores convocados e titulares, o oponente, e comentários adicionais.

A nossa aplicação irá implementar um menu de treinos, com as datas e locais de treinos, a lista de comparecentes, e comentários adicionais. A lista de comparecentes irá conseguir diferenciar os atletas que compareceram como ativos no treino, os que compareceram sem participar no treino ou os que compareceram para outra atividade ligada ao treino, como treinos físicos e de recuperação de lesões.

A nossa aplicação irá implementar um menu de planos de treino, onde a equipa técnica poderá fazer *upload* de planos de treino físicos ou de ginásio, e indicar as datas onde estes planos se devem concretizar e os atletas a quem os planos se dirigem.

A nossa aplicação irá implementar um calendário, onde irão estar demonstrados todos os jogos, treinos, torneios ou outros eventos adicionados pela equipa técnica e a sua respetiva data de concretização.

2.2.2 Especificações Secundárias

Nesta secção apresentam-se algumas especificações secundárias. Conforme a disponibilidade, são especificações que poderão ser inseridas no contexto da aplicação, nomeadamente:

1. Fisioterapeuta;
2. Lesão;
3. Campeonato;
4. Estatísticas Gráficas;
5. Exportação de Dados;

Estes conceitos refletem a possibilidade de monitorizar e documentar lesões, e apresentá-las de forma organizada numa interface própria para uso pelos fisioterapeutas, assim como de organizar os diversos jogos da época num campeonato com respetivas classificações. Também propõe a possibilidade de exportar dados em formatos de texto para serem consumidos por outros meios, assim como de representar visualmente as estatísticas dos jogos com auxílio gráfico.

2.3 Arquitetura da Solução

Esta secção explicita a arquitetura da nossa solução.

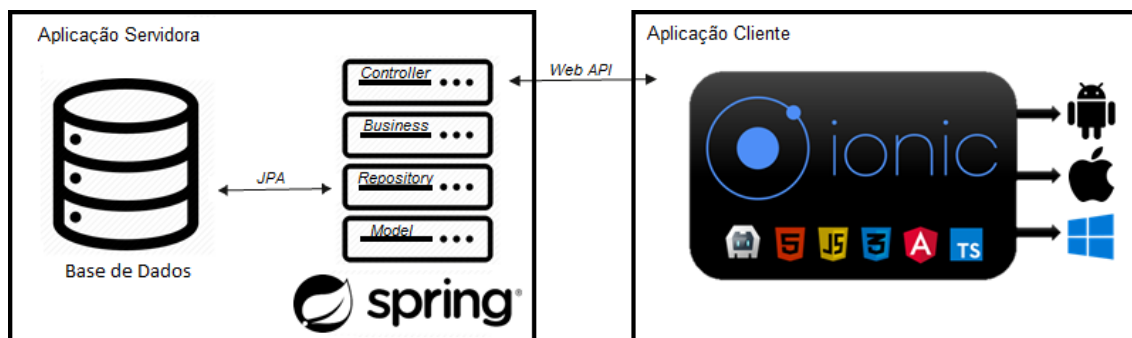


Figura 2.1: Arquitetura da nossa solução.

O nosso projeto irá ser dividido entre aplicação servidora e aplicação cliente.

A aplicação servidora irá ser programada em *Java* com uso da *Spring Boot framework*. A base de dados irá ser programada em *MySQL* com a ligação entre estes componentes feitos com o auxílio de *JPA - Java Persistence API*. As diferentes camadas da aplicação servidora são abordadas no capítulo 3.

A aplicação cliente irá ter ambas as vertentes *Web* e *mobile*, programadas em *TypeScript*, com o uso de *Angular framework* e *IONIC framework*.

A camada *Controller* da aplicação servidora expõe uma *Web API*, e são as chamadas feitas a essa *API* pelo lado da aplicação cliente que fazem a ligação entre as duas.

Entende-se que a maioria destes domínios sejam familiares ao leitor.

Capítulo 3

Aplicação Servidora

Este capítulo vai apresentar a nossa solução para o lado da aplicação servidora.

3.1 Introdução e Estrutura da Aplicação Servidora

Como é observável no diagrama da secção 2.3, a aplicação servidora é uma das duas peças principais do nosso projeto. É onde se encontra a base de dados, o modelo de dados, e todo o comportamento que os interliga um com o outro, assim como com a aplicação cliente, sejam estas leituras e escritas, algoritmos de pesquisa ou *routing*.

A partir das especificações funcionais principais discutidas na sub-secção 2.2.1, foi possível desenvolver a estrutura do nosso modelo de dados, observável no Apêndice A.

Como descrito na secção 2.3, a aplicação servidora irá ser desenvolvida com uso da *Spring Boot framework* e de *JPA - Java Persistence API*. Dadas as funcionalidades acrescentadas destas *framework* e *API*, a nossa solução separa a aplicação servidora em quatro camadas distintas:

1. *Model*
2. *Repository*
3. *Business*
4. *Controller*

As seguintes sub-secções explicam como cada camada funciona e como é que estas interagem entre si.

3.1.1 *Model*

A camada *Model*, referida nesta sub-secção como camada do Modelo, representa o modelo de dados. É aqui que encontramos todas as Entidades que estruturam o modelo de dados, assim como as relações entre elas.

O *JPA* é uma ferramenta que trata a forma como os objetos *Java* são persistidos, ou seja, como são guardados fora da aplicação que os criou. Apesar de nem todos os objetos precisarem de ser guardados, objetos como as instâncias de todas as entidades necessitam de ser persistidos fora da nossa aplicação, pois todos esses dados não podem ser perdidos sempre que a aplicação servidora for reiniciada. Foi decidido utilizar o *JPA* pela sua simplicidade nesta vertente de persistência. Em vez de implementarmos como é que os nossos objetos são guardados e recuperados, é definido o mapeamento entre os objetos e a base de dados, e a invocação do *JPA* gera as ligações e faz a persistência automaticamente.

Uma das características chave do *JPA* é a possibilidade de criar classes com a anotação *@Entity*, que indica ao *JPA* que esta classe é uma Entidade, e assim, a ferramenta mapeia-a diretamente para uma tabela na base de dados com o mesmo nome e propriedades.

Podemos observar no troço de código seguinte, como exemplo, a classe *Event* inserida na camada do Modelo, que representa um Evento.

```
1 @Entity
2 @Table (name = "event")
3 @Data
4 public class Event {
5
6     @Id
7     @GeneratedValue (strategy = GenerationType.AUTO)
8     private Long id;
9
10    @Column
11    private String name;
12
13    @Column
14    private String description;
15
16    @Column
17    private Date date;
18
19    @Column
20    private String local;
21
22    @Column
23    @OneToMany (mappedBy = "events")
24    private List<Profile> profiles;
25 }
```

Replicando este conceito para todas as outras entidades, obtemos uma camada de Modelo onde estão geradas todas as classes *@Entity* que o *JPA* mapeia para as tabelas da base de dados, que constituem a camada do Modelo. Na sub-secção seguinte é explicado como é que se interage com estas entidades.

3.1.2 *Repository*

A camada *Repository*, referida nesta sub-secção como camada de Repositório, representa os repositórios para cada entidade. Outra das características chave do *JPA* é a habilidade de criar interfaces de repositórios associadas às entidades do modelo, permitindo gerar per-

sistência na base de dados. Quando a aplicação interage com os repositórios (através da chamada a métodos de *queries*), o JPA faz a gestão das ligações à base de dados, garantindo a comunicação eficiente entre o modelo físico e o modelo de dados.

Podemos observar no troço de código seguinte, como exemplo, a interface *EventRepository* inserida na camada do Repositório.

```
1 public interface EventRepository extends CrudRepository<Event, Long> {
2     List<Event> findByDate(Date date);
3 }
```

Ao estender da interface *CrudRepository*, já implementada na biblioteca do JPA, as nossas classes de repositório herdam métodos para trabalhar com a persistência dos nossos objetos (neste caso, do *Event*), através de operações *Create*, *Read*, *Update* e *Delete*. Conforme a necessidade e o contexto, é possível adicionar outras *queries* (ex.: *findByDate*) diretamente a estas interfaces, sem a necessidade de as implementar.

O agrupamento de todos os repositórios de todas as nossas entidades constituem a nossa camada de Repositório.

3.1.3 *Business*

A camada *Business*, referida nesta sub-secção como camada de Negócio, representa todos os comportamentos referentes ao nosso modelo de negócios. É nesta camada que se encontra toda a algoritmia dedicada aos comportamentos da aplicação. Foram geradas classes *Business* para cada entidade, que contém comportamentos relacionados com procura e verificação de recursos. Esta camada liga diretamente aos repositórios.

Podemos observar no excerto de código seguinte, como exemplo, a classe *EventBusiness* inserida na camada de Negócio.

```
1 @Component
2 public class EventBusiness {
3     @Autowired
4     EventRepository eventRepository;
5
6     public Iterable<Event> findAllEvents(){
7         Iterable<Event> allEvents = eventRepository.findAll();
8         for (Event event : allEvents) {
9             event.getProfiles().forEach(profile -> profile.setEvents(Collections.
10                 emptyList()));
11         }
12         return allEvents;
13     }
14
15     public Long postEvent(Event event){
16         event.getProfiles().forEach(profile -> profile.getEvents().add(event));
17         event.setId(null);
18         return eventRepository.save(event).getId();
19     }
20
21     public Event findEventById(Long id){
22         Event event = eventRepository.findById(id).orElseThrow(() -> new
23             ResourceNotFoundException("Event", "Id", id));
24     }
25 }
```

```

22     event.getProfiles().forEach(profile -> profile.setEvents(null));
23     return event;
24 }
25
26 /* . . . */
27 }

```

A anotação *@AutoWired* garante a injeção do *EventRepository* quando a classe *EventBusiness* é criada. A anotação *@Component* permite que as classes sejam injetadas com *@AutoWired*. Cada um destes métodos tem uma interação diferente com o repositório, e nos casos justificados, faz a verificação da existência do objeto no repositório antes de o alterar/remover/retornar.

Este modelo de negócios garante a comunicação entre as camadas de controlo e repositório, servindo de camada intermédia onde é feita a verificação dos objetos antes de serem feitas alterações persistentes à base de dados.

3.1.4 *Controller*

A camada *Controller*, referida nesta sub-secção como camada de Controlo, representa todo o *routing* do exterior para a aplicação servidor. É a camada que gera todos os *endpoints*, assim como os métodos associados a estes *endpoints*. Esta camada expõe uma *web API*, que responde a pedidos feitos, no caso do nosso projeto, pela aplicação cliente.

Podemos observar no excerto de código seguinte, como exemplo, a classe *EventController* inserida na camada de Controlo.

```

1  @RestController ()
2  @RequestMapping ("/event")
3  public class EventController {
4      private static final Logger logger = LoggerFactory.getLogger(
5          RugbyApplication.class);
6
7      @Autowired
8      EventBusiness eventBusiness;
9
10     @RequestMapping ("event/all")
11     public Iterable<Event> findAllEvents() {
12         return eventBusiness.findAllEvents();
13     }
14
15     @GetMapping ("/findById/{id}")
16     public Event findEventById(@PathVariable Long id){
17         return eventBusiness.findEventById(id);
18     }
19
20     /* . . . */
21 }

```

A anotação *RestController* serve para implementar classes de controlo, que contém métodos capazes de processar pedidos HTTP, ao mesmo tempo que converte os objetos de retorno destes métodos para *HttpResponse*. Ou seja, todos os métodos desta classe são mapeados para um *endpoint* diferente e processam os pedidos para esse *endpoint*. A anotação *@RequestMapping*

ping recebe os parâmetros de mapeamento, podendo especificar-se o *endpoint* que o método ou classe de controlo vão processar, assim como outros parâmetros.

É de salientar que

@GetMapping corresponde a *@RequestMapping(method = RequestMethod.GET)*

@PostMapping corresponde a *@RequestMapping(method = RequestMethod.POST)*

@PutMapping corresponde a *@RequestMapping(method = RequestMethod.PUT)*

@DeleteMapping corresponde a *@RequestMapping(method = RequestMethod.DELETE)*

Podemos então observar que todos os pedidos para o caminho */event* serão processados por esta classe, onde cada método de cada pedido é processado num método da classe.

Após replicar este comportamento para as classes das diversas entidades, obtemos o *routing* completo para os *endpoints* da nossa aplicação.

Capítulo 4

Aplicação Cliente

Este capítulo vai apresentar a nossa solução para o lado da aplicação cliente.

4.1 Introdução e Estrutura da Aplicação Cliente

Como é observável na secção 2.3, a aplicação cliente é a segunda peça principal do nosso projeto. É onde se encontra a interface de utilizador, painéis de controlo e alguma lógica de negócio adicional. No apêndice B é observável um diagrama em árvore da navegação da aplicação cliente.

Como descrito na secção 2.3, a aplicação cliente foi desenvolvida com o uso de *Angular* e *Ionic*. O tipo de organização e estrutura de uma aplicação cliente que o *Angular* incentiva a aplicar é a estrutura *Component - Service*, e é esta estrutura que é seguida no nosso projeto. As seguintes sub-secções explicam as ideias chaves de ambas as *frameworks*.

4.1.1 *Angular Component e NgModules*

Um *Component* em *Angular* é uma peça visual de uma aplicação, que pode estender desde uma página a uma tabela ou a um *menu*. No caso da nossa aplicação cliente, cada especificação principal tem o seu componente. Dado que algumas regras de negócio implicam que certas páginas necessitem de componentes adicionais, como as estruturas *Modal* ou *Pop-over* que existem na *Ionic Framework* (explicadas em detalhe na sub-secção 4.1.4), foi tomada como regra de decisão gerar estas estruturas num *Component* separado do *Component* principal das páginas, que são invocados nas circunstâncias apropriadas.

Cada *Component* tem associado um *template*, que representa o *HTML* com a vista do componente, e um ficheiro *TypeScript* que representa o objeto do próprio *Component*, onde se encontram as definições das estruturas de dados internas do *Component*, os *imports* necessários ao *Component*, assim como métodos chamados pelo *template* com alguns comportamentos visuais, métodos que chamam os serviços que fazem o *data-fetching*, métodos de *redirect* da página, ou métodos que invocam os *Controllers* dos componentes adicionais men-

cionados anteriormente.

Cada Component tem também o seu *NgModule*. *NgModules* são um tipo de estrutura disposta no *Angular*, que ajuda a organizar a aplicação em módulos que podem ser importados ou importar outros módulos. A nossa aplicação cliente está assim estruturada para cada *Component* ter o seu próprio módulo (os componentes adicionais são inseridos no mesmo módulo que o componente a que estão contextualmente associados), assim como o seu próprio *routing module*, inserido dentro do módulo do componente, que trata do *routing* dentro do módulo. Esta abordagem permite-nos ter o *routing* todo da aplicação re-partido pelos módulos, em vez de estar todo centralizado num único componente.

4.1.2 *Angular Service*

Típicamente em arquitetura de *software*, o termo *Service* (ou serviço) é um termo utilizado para denominar uma peça de *software* que tem um conjunto de funcionalidades específicas e limitadas, que estão por norma ligadas e contextualizadas, e que podem ser utilizadas e reutilizadas por diferentes partes de uma aplicação.

No caso do *Angular*, um *Service* não é mais que uma classe onde são escritas funcionalidades, e que pode ser anotada como *@Injectable* para que o *Angular* consiga injectar essas funcionalidades num *Component* através de um *injector*.

No caso da nossa aplicação cliente, cada entidade dispõe de um serviço *HttpService* (agrupados no *package app/http/services*) que contém todos os métodos que fazem as chamadas à *web API* exposta pela aplicação servidora.

Podemos observar no excerto de código seguinte, como exemplo, o serviço *HttpEventService*.

```
1  /* . . . */
2  export class EventService {
3  private BASE_URL = 'http://localhost:8080/event';
4  private httpOptions = {
5  headers: new HttpHeaders({
6  'Content-Type': 'application/json',
7  Authorization: 'my-auth-token',
8  'Access-Control-Allow-Origin': '*'
9  })
10 };
11
12 constructor(private http: HttpClient) { }
13
14 getEvents(): Observable<Event[]> {
15 const url = `${this.BASE_URL}/all`;
16 return this.http.get<Event[]>(url, this.httpOptions);
17 }
18
19 getEventsById(id: any) {
20 const url = `${this.BASE_URL}/findById/${id}`;
21 return this.http.get(url, this.httpOptions);
22 }
23 /* . . . */
```

Um dos aspetos principais dos serviços do *Angular* são os *Observables*. Os *Observables* são um tipo de objetos implementados na biblioteca *RxJS* que é utilizada pelo *Angular*. Os *Observables* atuam da mesma maneira que as *Promises* (objeto *Javascript* que representa a eventual conclusão de uma operação assíncrona e o seu resultado), com algumas diferenças chave:

<i>Lazy</i>	Os <i>Observables</i> , ao contrário das <i>Promises</i> , são <i>Lazy</i> . A função de <i>callback</i> passada ao <i>Observable</i> só é invocada quando se chama o método <i>subscribe</i> do <i>Observable</i> .
<i>Synchronous</i>	Os <i>Observables</i> podem ser ambos síncronos e assíncronos, enquanto que as <i>Promises</i> são apenas assíncronas.
Multiplos Valores	Os <i>Observables</i> podem emitir multiplos valores. Um objeto <i>Promise</i> retorna sempre apenas um objeto (podendo este ser um <i>array</i> de valores, mas continua a ser um único objeto). Um <i>Observable</i> pode ser <i>subscribed</i> a várias alturas da execução, e de cada vez que é <i>subscribed</i> , executa o seu comportamento novamente.
<i>Operators</i>	A biblioteca <i>RxJS</i> apresenta um conjunto de <i>operators</i> que podem ser aplicados ao <i>stream</i> do <i>Observable</i> (como o <i>map</i>). Este tipo de comportamento não existe para <i>Promises</i> .

Certas entidades dispõem também de um serviço próprio (agrupados no *package app/components/services*) que contêm todos os métodos que envolvem a algoritmia de regras de negócio adicionais dessas entidades. Podemos observar no troço de código seguinte, como exemplo, o serviço *AthleteGameStatsService*, que contém um método *getTotal()*, que retorna um objeto *Stats* com o somatório de todos os *Stats* dentro do array de *AthleteGameStats*, utilizado na geração da tabela de estatísticas de um atleta.

```
1
2 import { Injectable } from '@angular/core';
3 import { AthleteGameStats } from "../../classes/associations/AthleteGameStats";
4 import { Stats } from "../../classes/stats";
5
6 @Injectable ({
7   providedIn: 'root'
8 })
9
10 export class AthleteGameStatsService {
11
12   constructor() { }
13
14   getTotal(stats: AthleteGameStats[]) {
15     let acc: Stats = new Stats();
16     Object.keys(acc).forEach( key => {
17       stats.forEach( stat => {
18         acc[key] += stat.stats[key];
19       });
20     });
21     return acc;}}
```

4.1.3 Angular Data Binding

O *Angular* também dispõe de um sistema de *binding* de *templates* que é utilizado com regularidade ao longo da aplicação. Dado que cada *Component* tem a sua instância de classe e *template* com a sua vista, o *Angular* permite que estas vertentes comuniquem uma com a outra programáticamente através do chamado *Data Binding*. Os principais tipos de *Data Binding* que existem no *Angular* são:

- | | |
|----------------------|--|
| <i>Interpolation</i> | Ligação de uma expressão a um elemento <i>HTML</i> do <i>template</i> .
A sintaxe deste <i>binding</i> é <code>{{expression}}</code> . |
| <i>Property</i> | Ligação de uma propriedade do <i>Component</i> a um elemento <i>HTML</i> do <i>template</i> . É uma ligação denominada <i>Source-to-View</i> .
A sintaxe deste <i>binding</i> é <code>[target]="expression"</code> . |
| <i>Event</i> | Ligação de um evento de um elemento <i>HTML</i> a uma declaração do <i>Component</i> . É uma ligação denominada <i>View-to-Source</i> .
A sintaxe deste <i>binding</i> é <code>(target)="statement"</code> . |
| <i>Two-Way</i> | Ligação dupla entre uma propriedade e um elemento.
O valor da propriedade transita até ao elemento, onde este pode ser alterado através da interação do utilizador, e o novo valor transita de volta até à propriedade. A sintaxe deste <i>binding</i> é <code>[(target)]="expression"</code> . |

O *Angular* também contém alguns tipos de *binding* especializados para estilos. Estes tipos de *binding* são utilizados quando se quer alterar programáticamente alguns aspetos visuais do *template* sem querer fazer essa ligação a propriedades no *Component*. Estes tipos especializados são:

- | | |
|------------------|--|
| <i>Attribute</i> | Ligação de um atributo a um elemento <i>HTML</i> do <i>template</i> . A sintaxe deste <i>binding</i> é igual ao <i>Property Binding</i> , substituindo o nome da propriedade pelo nome do atributo <code>[attr.attribute-name]="expression"</code> . |
| <i>Class</i> | Corresponde a ligar uma classe <i>CSS</i> a um elemento <i>HTML</i> do <i>template</i> .
A sintaxe deste <i>binding</i> é igual ao <i>Property Binding</i> , substituindo o nome da propriedade pelo nome da classe <i>CSS</i> <code>[class.name]="expression"</code> . |
| <i>Style</i> | Corresponde a ligar um estilo específico a um elemento <i>HTML</i> do <i>template</i> .
A sintaxe deste <i>binding</i> é igual ao <i>Property Binding</i> , substituindo o nome da propriedade pelo nome do estilo <code>[style.name]="expression"</code> ; |

4.1.4 IONIC API

A *API* do *Ionic* dispõe de uma lista de vários componentes visuais que podem ser intercalados com os do *Angular* para criar uma aplicação responsiva e adaptável ao dispositivo em que é executada. Em [1] podemos encontrar a lista completa de todos os componentes disponíveis na *API* do *Ionic*. Ao longo deste projeto foram usados diversos componentes, nos casos que foram considerados apropriados, para garantir que eram cumpridas todas as regras de negócio aplicáveis, assim como garantir que a interface do utilizador seja clara e explícita na sua utilização. Ao longo do desenvolvimento da aplicação cliente, foram utilizados diversos componentes em diversos contextos. No que toca a menus, foram utilizados:

- ion-split-pane* Menu lateral que aparece constantemente em toda a aplicação, onde se encontram ligações para as diversas páginas.
- ion-tabs* Barra de navegação, que aparece no *footer* da página, com *icons* com *routing* para todas as páginas da aplicação.
- ion-header* Barra de navegação, que aparece no *header* da página, com uma *ion-toolbar* onde podem ser inseridos botões ou segmentos.

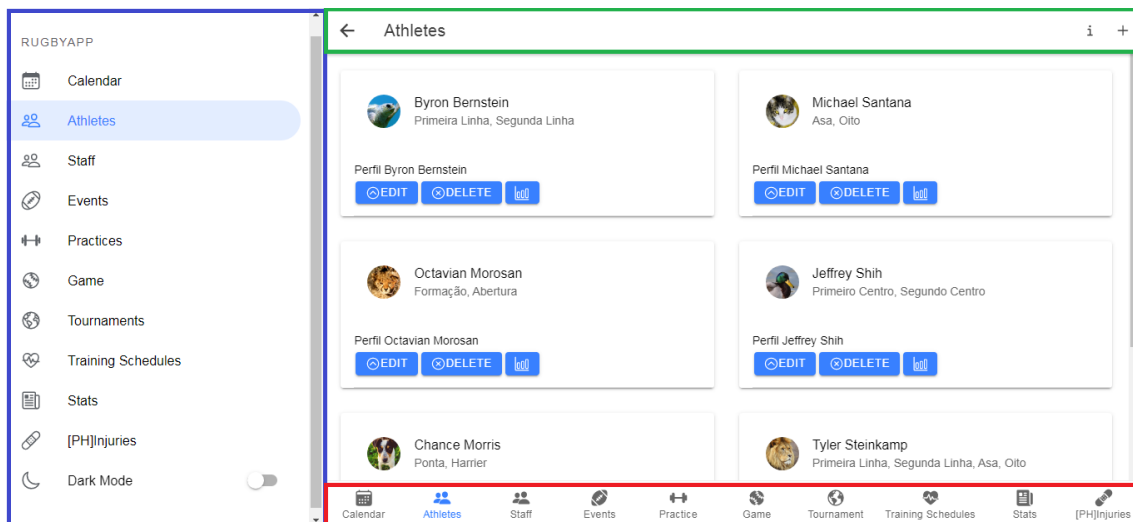


Figura 4.1: Vista inicial da aplicação. Observa-se o *ion-split-pane* a azul, o *ion-tabs* a vermelho e o *ion-header* a verde.

Como mencionado na secção 4.1.1, existem páginas na nossa aplicação cliente que requerem mais do que um *Component* para garantir certas regras de negócios sem que a informação nas páginas fique sobrelotada e proporcionando a melhor experiência de utilização. Existem componentes na *API* do *Ionic* (alguns também mencionados na secção 4.1.1) que foram utilizados para este mesmo propósito:

- ion-modal* *Dialog* (componente visual que se sobrepõe ao contexto atual da página e requer interação do utilizador para desaparecer), normalmente utilizado para apresentar uma página onde o utilizador tem diversas opções de interação.
- ion-popover* *Dialog* (assim como o *ion-modal*) que é geralmente usado para conter acções ou informação que não pode ser mostrada na totalidade sem comprometer visualmente os elementos da página.
- ion-select* Apesar de não requerer um *controller* ou um *component* próprio para ser programado, o *ion-select* é um *ion-modal* pré-definido na *API* exclusivamente para *input/output* (ou seja, não é possível atribuir-lhe diretamente outro comportamento que não o de dar ao utilizador diversas opções de escolha).

Name	Regular	Physio
Byron Bernstein	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Michael Santana	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Octavian Morosan	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Jeffrey Shih	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Chance Morris	<input type="checkbox"/>	<input type="checkbox"/>

SUBMIT

Figura 4.2: *Modal* da página do formulário de *Practice*.

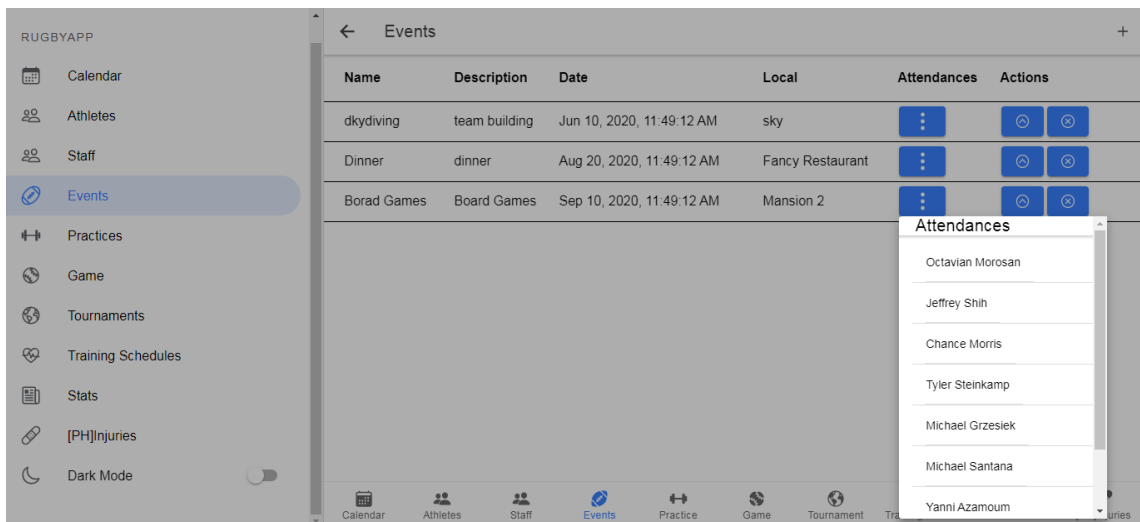


Figura 4.3: *Popover* da página *Event*.

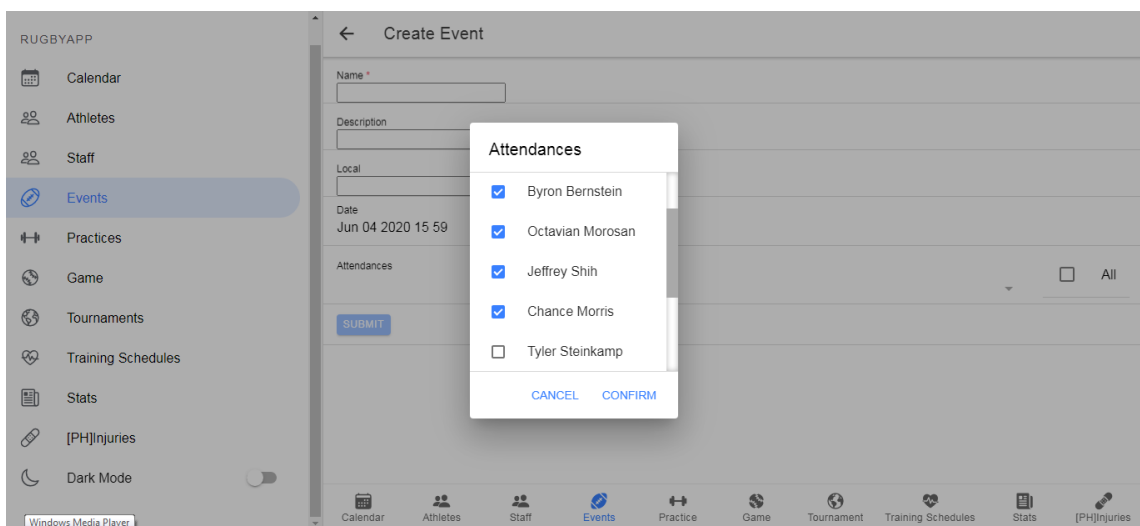


Figura 4.4: *Select* da página do formulário de *Event*.

Como referido anteriormente, ao contrário do *ion-select*, o *ion-modal* e o *ion-popover* são componentes próprios e requerem um *controller* para serem gerados e criados por outros componentes. Podemos observar no troço de código seguinte, o método *createPopover()* inserido no *event.component*

```
1 constructor(private eventService: EventService, private popoverController:
   PopoverController, private alertController: AlertController) { }
2
3 /* . . . */
4
5 async createPopover(profiles: Profile[], ev) {
6   const popover = await this.popoverController.create({
7     component: EventPopoverComponent,
8     componentProps: { profiles },
9     event: ev
10  });
11  return await popover.present();
12 }
```

O *Component event.component* importa o módulo *PopoverController* do *package* do *Ionic*, e cria a sua instância no construtor. Atribuindo o método *createPopover* a um evento, é assim invocada a criação de um *Popover*.

```
1 <ng-container matColumnDef="profiles">
2   <th mat-header-cell *matHeaderCellDef> Attendances </th>
3   <td mat-cell *matCellDef="let element">
4     <ion-button (click)="createPopover(element.profiles,$event)">
5       <ion-icon slot="icon-only" name="ellipsis-vertical"></ion-icon>
6     </ion-button>
7   </td>
8 </ng-container>
```

Cada elemento da coluna de *Attendances* tem o seu próprio botão que vai criar o seu próprio *Popover*. Passando a lista de *Profiles* associadas a cada elemento, podemos criar um *Popover* com cada uma das listas de *Profiles* na tabela da página *Event*.

```
1 <ion-content>
2   <ion-list>
3     <ion-item *ngFor="let profile of this.navParams.get('profiles')">
4       /* . . . */
5     </ion-item>
6   </ion-list>
7 </ion-content>
```

Através de um módulo existente no *package* do *Ionic*, chamado *NavParams*, é possível passar informação ao *Controller* do *Popover*, e é possível do lado do *Popover* obter essa informação para ser consumida. Neste caso, o *Popover* é meramente uma lista de itens com os nomes dos *Profiles*, com um *Link* para a página desse *Profile*.

Do lado do *ion-modal*, a ideia-chave é idêntica, exceto que o módulo que é importado pelo *component* passa a ser o *ModalController* em vez de *PopoverController*. Continua-se a usar o *NavParams* para passar informação ao *Modal*.

4.1.5 Angular Materials CDK

O *Angular* também dispõe de um *CDK* (*Component Dev Kit*) chamado *Angular Materials*, onde encontramos vários componentes visuais. Em [2] podemos encontrar a lista completa de todos os componentes disponíveis neste *CDK*. Apesar da nossa aplicação cliente explorar maioritariamente componentes da *IONIC API*, também são usados alguns componentes deste *CDK*, nomeadamente

<i>mat-table</i>	Tabela usada para mostrar dados em diversas páginas da nossa aplicação cliente, com a possibilidade de adicionar comportamentos adicionais à tabela, como paginação, linha de rodapé, filtro e ordenação.
<i>mat-grid-list</i>	Grelha que permite variar os tamanhos que cada item ocupa na grelha, ambos em número de colunas ou número de linhas, criando dinamismo na organização da grelha sem a comprometer visualmente.
<i>mat-form-field</i>	Componente que representa o campo de um formulário, onde se podem aplicar estilos de texto como <i>Placeholder</i> , <i>Hint</i> ou texto de erro.

Podemos observar no excerto de código seguinte o *template* da *mat-table* de *Games*.

```
1 <ion-content>
2 <table mat-table [dataSource]="this.dataSource" matSort class="mat-elevation-
  z8">
3
4 <ng-container matColumnDef="date">
5 <th mat-header-cell *matHeaderCellDef mat-sort-header> Date </th>
6 <td mat-cell *matCellDef="let element"> {{element.date | date:"medium"}} </td
  >
7 </ng-container>
8
9 /* . . . */
10
11 <tr mat-header-row *matHeaderRowDef="displayedColumns"></tr>
12 <tr mat-row *matRowDef="let row; columns: displayedColumns;"></tr>
13 </table>
14 </ion-content>
```

As *mat-table* do *Angular Materials* funcionam com base num objeto *DataSource*. O *Component* principal de cada página tem uma propriedade com este objeto, e após fazer *data-fetching*, afeta esta propriedade com a informação para popular a tabela com os dados. Cada *Component* tem também um *array* com o nome das colunas da tabela. Da forma que o *mat-table* é implementado, podemos organizar os nomes neste *array* com a ordem que se pretende que as colunas sejam apresentados visualmente, e a tabela será desenhada com essa ordem, independentemente da ordem como está definido o HTML de cada coluna.

```
1 export class GameComponent implements OnInit {
2   games: Game[];
3   displayedColumns: string[] = ['date', 'local', 'opponent', 'score', 'comment',
    , 'athletes', 'actions'];
4   dataSource: any;
```

```

5 @ViewChild (MatSort, {static: true}) sort: MatSort;
6
7 /* . . . */
8
9 showGames() {
10 this.gameService.getGames().subscribe(games => {
11 this.games = games;
12 this.dataSource = new MatTableDataSource(this.games);
13 this.dataSource.sort = this.sort;
14 });
15 }
16 /* . . . */
17 }

```

← Games + +						
Date	Local ↓	Opponent	Score	Comment	Call	Actions
May 22, 2020, 1:33:17 PM	unsiverse	 SuperTeam	13 - 45 vs superteam			  
May 29, 2020, 1:33:17 PM	seol	 Team1	15 - 15 t1			  
Jul 11, 2020, 1:33:17 PM	nowhere	 Team do Boss	13 - 12 tdb			  
May 15, 2020, 1:33:17 PM	madrid	 Galacticos	50 - 10 Vs galaticos			  

Figura 4.5: Tabela da página de *Games*, ordenada decrescentemente por nome.

No que toca ao *mat-grid-list*, podemos observar no troço de código seguinte como é que este componente é utilizado no *Profile* de um *Staff*.

```

1 <mat-grid-list cols="3" rowHeight="100px">
2 <mat-grid-tile [colspan]="1" [rowspan]="3">
3 <img [src]="staff?.profile.photo">
4 </mat-grid-tile>
5
6 <mat-grid-tile [colspan]="2" [rowspan]="1">
7 <ion-label>
8 <div class="ion-text-center"> <b> Nome </b> </div>
9 <div class="ion-text-center"> {{staff?.profile.name}} </div>
10 </ion-label>
11 </mat-grid-tile>
12
13 /* . . . */
14 </mat-grid-list>

```

Como referido anteriormente, uma das características chave do *mat-grid-list* é a possibilidade de atribuir tamanhos diferentes aos diferentes itens da grelha, permitindo-os organizar visualmente enquanto se mantem as proporções corretas da tabela.

É possível aferir, com base na imagem anterior, que a grelha gerada é uma grelha 3x3, no


	Nome Rui Miguel Marques Garcia	
	Tipo de Staff Coach	Número de Staff 123
	Data de Nascimento May 27, 2020	Morada

Figura 4.6: *Profile* de um *Staff*, com as linhas da grelha *highlighted* para fins de visualização.

entanto, o item da imagem do perfil ocupa 1x3, o nome do perfil ocupa 2x1, e os restantes itens ocupam 1x1. É possível também aferir, apesar dos tamanhos variados, que a grelha fica organizada e ocupa espaço proporcional em ambos os eixos.

4.1.6 Classes e Entidades

Foram geradas na aplicação cliente as classes correspondentes às entidades da aplicação servidora em *TypeScript*. Podemos observar no troço de código seguinte, como exemplo, a classe *Event.ts* inserida no *package* de classes da nossa Aplicação Cliente.

```

1 export class Event {
2   constructor(
3     private id?: number,
4     private name?: string,
5     private description?: string,
6     private date?: Date,
7     private local?: string,
8     private profiles?: Profile[]
9   ) {
10    this.id = id ? id : 0;
11    this.description = description ? description : '';
12    this.date = date ? date : new Date(0);
13    this.local = local ? local : '';
14    this.name = name ? name : '';
15    this.profiles = profiles ? profiles : [];
16  }
17 }

```

Um dos aspetos principais a salientar na implementação dos construtores das Entidades na aplicação cliente é a questão das propriedades poderem ser *nullable*. Organizando os construtores para que todas as propriedades sejam *nullable* enquanto se faz a verificação no corpo do construtor para a ausência destas propriedades, permite-se construir objetos atribuindo valores padrão a todas as propriedades que não existem na altura da criação. Este detalhe de implementação ajuda a gerar objetos vazios sem os problemas que ocorrem frequentemente na manipulação de valores *null*.

4.1.7 IONIC Lifecycle

Em diversas partes da aplicação cliente, é necessário garantir que alguns comportamentos são executados em alturas específicas da geração e carregamento do componente da página. Por exemplo, em certas páginas que contêm demasiada informação em relação ao tamanho da página, é necessário que o *ion-split-pane* seja escondido para garantir que a página não fica visualmente comprometida pelo tamanho dos componentes que são mostrados. Para atingir este objetivo, o *IONIC* dispõe de um *lifecycle*, com vários ganchos que são executados em alturas específicas do carregamento do componente. Este *lifecycle* segue a seguinte estrutura ordenada

<i>constructor</i>	É executado quando a página é iniciada. É o melhor sitio para definir valores <i>default</i> para as variáveis do componente.
<i>ionViewDidLoad</i>	É executado quando a página foi carregada. Este evento é só executado uma vez por cada criação de cada página. Se a página for recarregada mas estiver em <i>cache</i> , este evento não é executado outra vez.
<i>ionViewWillEnter</i>	É executado quando a página está prestes a entrar e a tornar-se a página ativa.
<i>ionViewDidEnter</i>	É executado quando a página entrou por completo e é agora a página ativa. Este evento é sempre executado, independentemente de ser o primeiro carregamento da página ou se a página for carregada da <i>cache</i> .
<i>ionViewWillLeave</i>	É executado quando a página está prestes a sair e deixar de ser a página ativa.
<i>ionViewDidLeave</i>	É executado quando a página acabou de sair e já não é a página ativa.
<i>ionViewWillUnload</i>	É executado quando a página está prestes a ser destruída e os seus elementos a serem removidos.

Seguindo o exemplo referido anteriormente, o *ion-split-pane* é escondido no gancho *ionViewDidEnter* e é recuperado no gancho *ionViewDidLeave*.

4.1.8 *Chart.js*

A geração de gráficos é feita com base numa biblioteca chamada *Chart.js*. *Chart.js* é uma biblioteca *open-source* com o foco principal em criar grafos responsivos. Em [3] encontramos a referência para esta biblioteca.

Esta biblioteca contém a implementação de um objeto *Chart*, que recebe, entre outra informação, o tipo de gráfico, as *labels*, algumas opções de renderização, e *Data Sets*. Cada *Data Set* representa um bloco de informação no gráfico (no caso de um gráfico de barras, cada barra é um *data set*, no caso de um gráfico de linhas, cada linha é um *data set*).

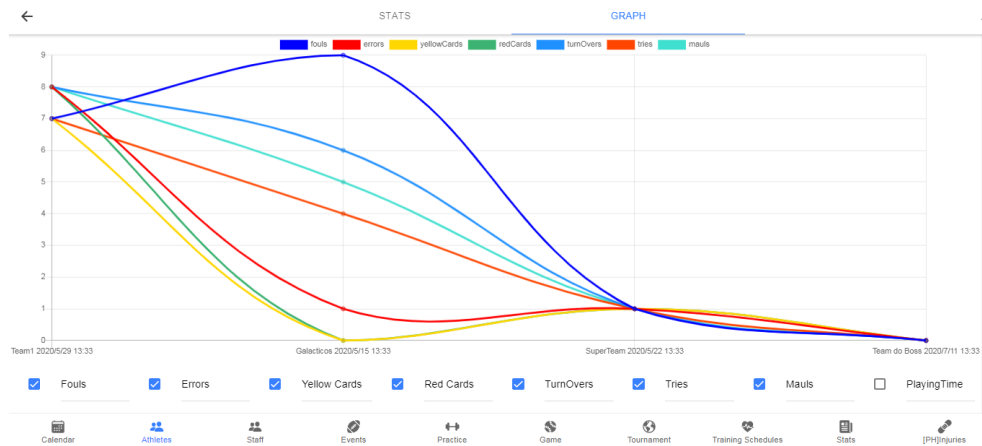


Figura 4.7: Aba *Graph*, com as estatísticas *Fouls*, *Errors*, *YellowCards*, *RedCards*, *turnOvers*, *tries* e *mauls* selecionados.

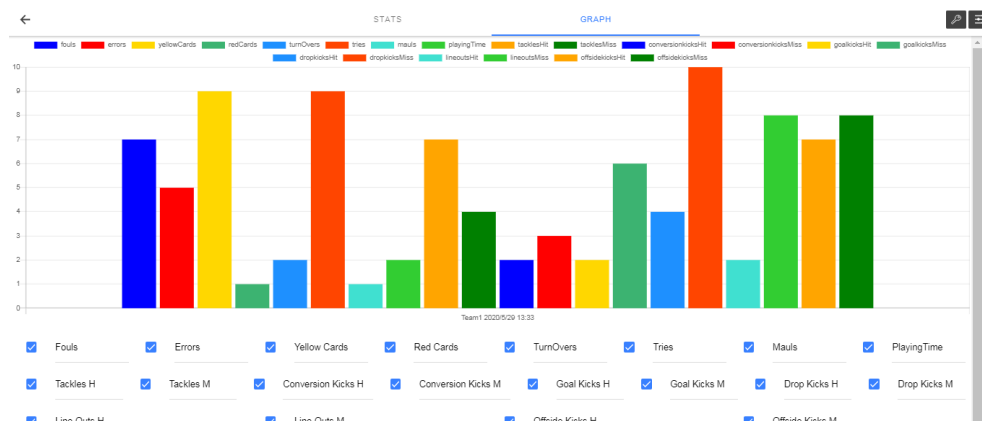


Figura 4.8: Aba *Graph*, com as estatísticas todas selecionadas. Quando o utilizador filtra os gráficos para apenas um atleta, a aplicação gera um gráfico de barras.

Capítulo 5

Testes

Este capítulo aborda os testes executados no projeto.

5.1 Aplicação Servidora

A partir desta fase do projeto, todos os testes feitos à aplicação servidora passaram a ser feitos a partir de uma *framework* chamada *Mockito*. Em [4] encontramos a referência para esta *framework*.

O *Mockito* é uma *framework Java* focada em testes unitários, que permite injetar *mocks* nas classes que queremos testar, permitindo assim criar um *man-in-the-middle* que intercepta chamadas a métodos dessas classes. Através da lógica *when(calledMethod) - thenReturn(desiredValue)*, podemos atribuir comportamentos aos *mocks* para que quando um certo método é chamado, em vez da chamada acontecer, o *mock* retorna um valor pré-fabricado. Esta *framework* é especialmente útil para testar a camada de *Business* da nossa aplicação servidora, sem que os testes tenham de fazer *GET* ou *POST* sobre a informação da base de dados.

Podemos observar, no troço de código seguinte, alguns dos testes feitos ao *AthleteBusiness*:

```
1 public class AthleteBusinessTest {
2
3     @InjectMocks
4     private AthleteBusiness business;
5
6     @Mock
7     AthleteRepository repository;
8
9     @Mock
10    ProfileRepository profileRepository;
11
12    /* . . . */
13
14    @Test
15    public void postProfileTest(){
16        Athlete athlete = createAthlete();
17        Mockito.when(repository.save(Mockito.any(Athlete.class))).thenReturn(
18            athlete);
19        Mockito.when(profileRepository.save(Mockito.any(Profile.class))).
20            thenReturn(createProfile());
```

```

19     Long id = business.postAthlete(athlete);
20     Assert.assertEquals(athlete.getId(), id);
21 }
22
23 @Test
24 public void getExistingProfileById(){
25     Athlete athlete = createAthlete();
26     Mockito.when(repository.findById(Mockito.any(Long.class))).thenReturn(
Optional.of(athlete));
27     Athlete athlete2 = business.findAthleteById(1L);
28     Assert.assertEquals(athlete.getTrainingSchedules().size(), athlete2.
getTrainingSchedules().size());
29     Assert.assertEquals(athlete.getAthleteGameStats().size(), athlete2.
getAthleteGameStats().size());
30     Assert.assertEquals(athlete.getAthletePractices().size(), athlete2.
getAthletePractices().size());
31     Assert.assertEquals(athlete.getActiveRosters().size(), athlete2.
getActiveRosters().size());
32     Assert.assertEquals(athlete.getGames().size(), athlete2.getGames().size()
);
33     Assert.assertEquals(athlete.getId(), athlete2.getId());
34     Assert.assertEquals(athlete.getAthleteNumber(), athlete2.getAthleteNumber
());
35     Assert.assertEquals(athlete.getComment(), athlete2.getComment());
36     Assert.assertEquals(athlete.getHeight(), athlete2.getHeight());
37     Assert.assertEquals(athlete.getWeight(), athlete2.getWeight());
38     Assert.assertEquals(athlete.getPositions(), athlete2.getPositions());
39 }
40
41 @Test(expected = ResourceNotFoundException.class)
42 public void updateNotExistingProfile(){
43     Athlete athlete = createAthlete();
44     Mockito.when(repository.findById(Mockito.any(Long.class))).thenReturn(
Optional.empty());
45     Mockito.when(profileRepository.findById(Mockito.any(Long.class))).
thenReturn(Optional.empty());
46     business.updateAthlete(athlete);
47 }

```

A anotação *@Mock* serve para criar *Mocks* que vão suportar a classe que vai ser testada (neste caso, os repositórios das entidades da classe a ser testada). A anotação *@InjectMocks* cria uma instância da classe que irá ser testada.

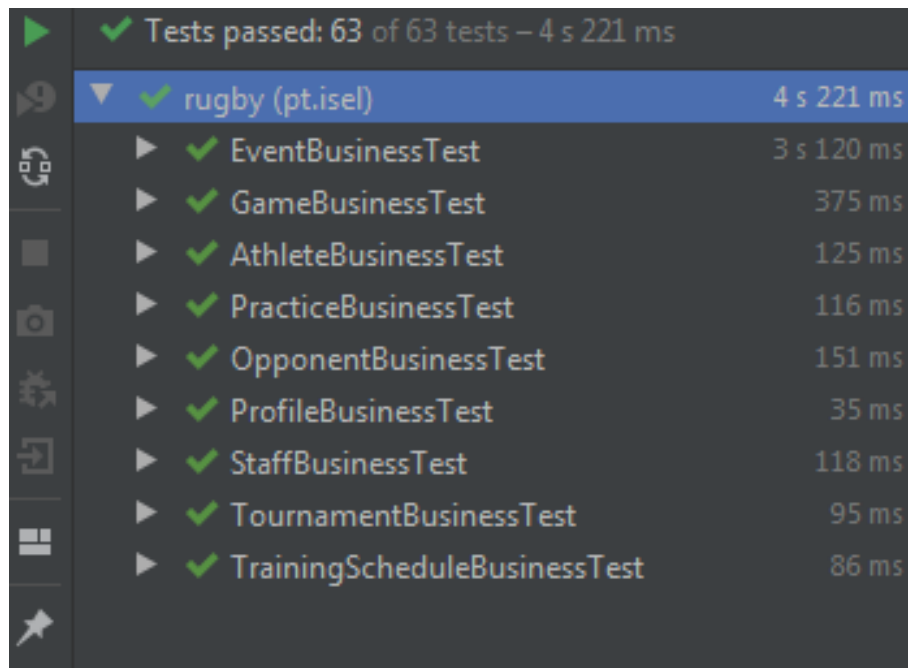


Figura 5.1: Demonstração de todos os testes da aplicação servidora a concluírem com sucesso.

5.2 Aplicação Cliente

Todos os testes feitos à aplicação cliente foram executados com ligação direta à *web API* exposta pela aplicação servidora. Todos os exemplos e demonstrações apresentados ao longo do Capítulo 4 têm como base informação que provem diretamente das chamadas aos *endpoints* da aplicação servidora, e toda a aplicação cliente foi testada com esses dados.

Capítulo 6

Conclusões (até agora)

Este capítulo descreve as conclusões que foram adquiridas do trabalho feito até ao momento.

6.1 Recapitulação

Como referido na secção 2.1, foi possível, através de reuniões com clubes deste desporto, definir as propriedades principais e secundárias da nossa aplicação. Após estas propriedades estarem definidas e estruturadas, foi gerado o modelo de entidades onde assenta a nossa aplicação (demonstrado no Apêndice A).

No que toca à aplicação servidora, como referido no capítulo 3, foram definidas todas as estruturas, tanto em termos de entidades e informação persistente, como em termos do formato em que a aplicação no geral irá ter acesso a esta informação. Distribuindo os focos principais nas camadas de Modelo, Repositório, Negócio e Controlador, podemos de uma forma organizada separar todo o processo que envolve o caminho desde o *browser* até à base de dados. Com o auxílio das ferramentas apresentadas na secção 2.3, conseguimos organizar toda esta partição e mantê-la consistente.

Do lado da aplicação cliente, como apresentado no capítulo 4, foram geradas as classes que correspondem às classes de Entidades da aplicação servidora, e criados as diversas páginas para cada informação de cada entidade relevante. A estrutura da aplicação cliente está organizada entre componentes e serviços, onde os componentes têm todo o código necessário à sua representação visual, e os serviços tratam da algoritmia complementar para garantir o funcionamento dos componentes.

Todos os testes feitos ao lado da aplicação servidora permitem observar o comportamento dos *endpoints* e a persistência dos dados na base de dados.

Todos os testes feitos ao lado da aplicação cliente foram feitos com base nos dados persistentes na aplicação servidora.

Referências

- [1] Ionic Docs. 2020. API Index - Ionic Documentation. [online] Available at: <https://ionicframework.com/docs/api> [Accessed 4 June 2020].
- [2] Team, A., 2020. Angular Material. [online] Available at: <https://material.angular.io/> [Accessed 4 June 2020].
- [3] Chartjs.org. 2020. Chart.js — Open Source HTML5 Charts For Your Website. [online] Available at: <https://www.chartjs.org/> [Accessed 15 June 2020].
- [4] Site.mockito.org., 2020. Mockito Framework Site. [online] Available at: <https://site.mockito.org/> [Accessed 15 June 2020].

Apêndice A

Diagrama de Entidades

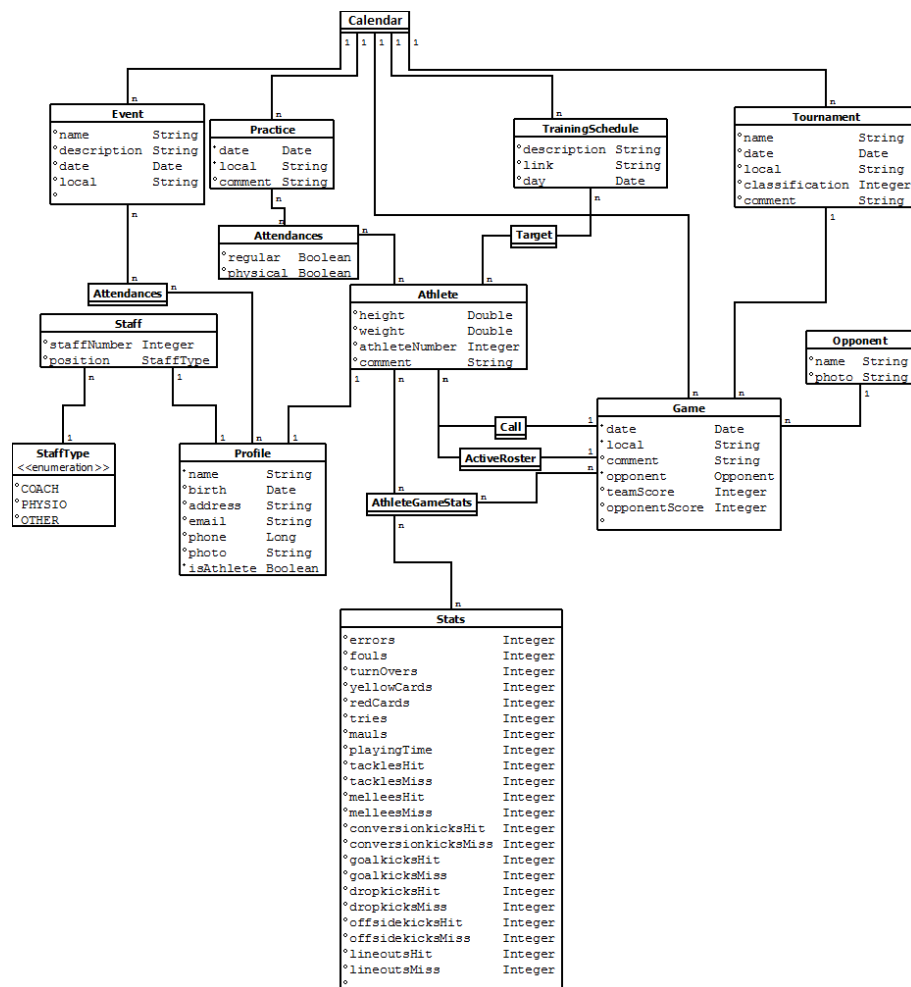


Figura A.1: Diagrama de Entidades

Apêndice B

Árvore de Navegação

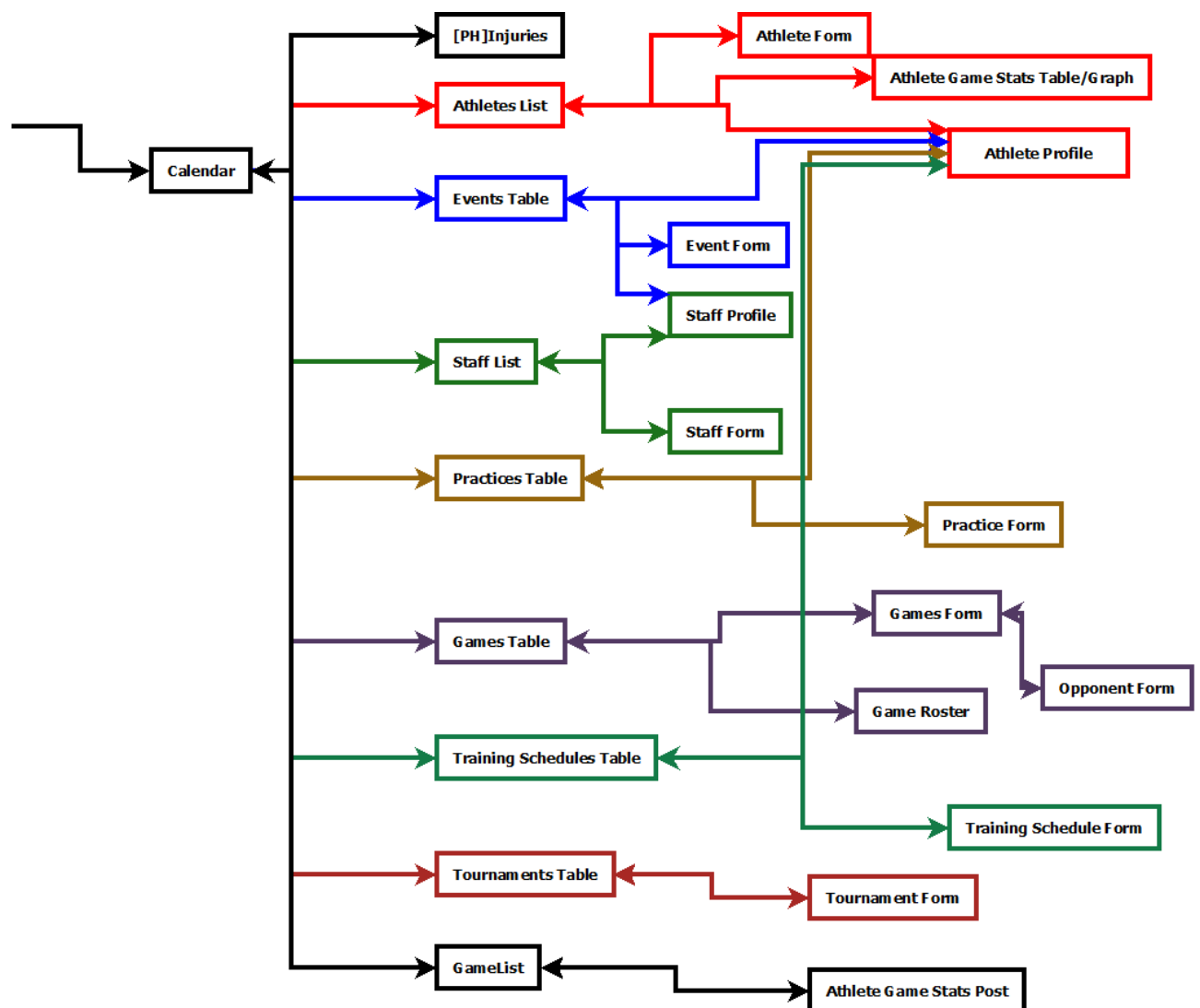


Figura B.1: Árvore de Navegação

Apêndice C

Informação do Dossiê de Projeto

O nosso projeto está alojado num repositório privado do *GitHub*, em <https://github.com/ruigarcia7/ps-1920-sv>. Para obter acesso a este repositório, o requerente deverá enviar um e-mail para o endereço **A40539@alunos.isel.pt** com o seu nome de utilizador do *GitHub*.