



# RugbyApp

Rui Garcia  
João Ferreira

Orientador: Jorge Martins

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

Maio 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 de progresso realizado no âmbito de Projeto e Seminário  
Licenciatura em Engenharia Informática e de Computadores

Maio 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 duma equipa, esperamos no fim apresentar uma ferramenta que consiga aglomerar os aspetos principais de uma equipa 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

<b>1</b>	<b>Introdução</b>	<b>1</b>
1.1	Enquadramento . . . . .	1
1.2	Objetivos e Funcionalidades . . . . .	1
1.3	Organização do documento . . . . .	2
<b>2</b>	<b>Formulação do Problema</b>	<b>3</b>
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
<b>3</b>	<b>Aplicação Servidora</b>	<b>7</b>
3.1	Introdução e Estrutura da Aplicação Servidora . . . . .	7
3.1.1	<i>Model</i> . . . . .	8
3.1.2	<i>Repository</i> . . . . .	8
3.1.3	<i>Business</i> . . . . .	9
3.1.4	<i>Controller</i> . . . . .	10
<b>4</b>	<b>Aplicação Cliente</b>	<b>13</b>
4.1	Introdução e Estrutura da Aplicação Cliente . . . . .	13
4.1.1	<i>Angular Component</i> . . . . .	14
4.1.2	<i>Angular Service</i> . . . . .	15
4.1.3	<i>Angular Data Binding</i> . . . . .	17
4.1.4	<i>IONIC API</i> . . . . .	18
4.1.5	Angular Materials CDK . . . . .	23
4.2	Descrição dos <i>Endpoints</i> . . . . .	28
4.2.1	<i>Main Component</i> . . . . .	28
4.2.2	<i>Calendar</i> . . . . .	29
4.2.3	<i>Athletes/Staff</i> . . . . .	36
4.2.4	<i>Profile</i> . . . . .	37

4.2.5	<i>AthleteStats</i> . . . . .	38
4.2.6	<i>Events/Practices/Games/Tournaments/TrainingSchedules</i> . . . . .	40
4.2.7	<i>GameStats</i> . . . . .	44
4.2.8	<i>Forms</i> . . . . .	46
4.2.9	<i>[WIP]Injuries</i> . . . . .	48
<b>5</b>	<b>Testes</b>	<b>49</b>
5.1	Aplicação Servidora . . . . .	49
5.1.1	Testes de GET . . . . .	50
5.1.2	Testes de POST . . . . .	53
5.1.3	Testes de UPDATE . . . . .	55
5.1.4	Testes de DELETE . . . . .	58
5.2	Aplicação Cliente . . . . .	61
<b>6</b>	<b>Conclusões (até agora)</b>	<b>63</b>
6.1	Recapitulação . . . . .	63
	<b>Referências</b>	<b>65</b>
<b>A</b>	<b>Apêndice A</b>	<b>67</b>

# 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 um grupo de ferramentas que proporcionem uma melhor qualidade na organização e prática desta modalidade, este foi considerado pelos estudantes como um grupo escasso e dispendioso, pelo que foi optado como objectivo 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 e Funcionalidades

Esta secção aborda os objetivos e funcionalidades 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, afim de monitorizar aspetos críticos que avaliem não só o estado individual de cada jogador, como o estado atual de toda a equipa. Pressupõe-se que toda a informação recolhida consiga facilitar

aspectos chaves do funcionamento de uma equipa desportiva, auxiliando desde a face tática do desporto (aspectos como a constituição da equipa, o plano tático, a otimização de temáticas de treino), assim como a face menos técnica de uma equipa (aspectos 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 aglomerados 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 uma linha temporal sobre todos os eventos futuros contextuais com a equipa. No que toca à equipa técnica, esta também terá a funcionalidade de gerar jogos, manusear 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 adicionar estatísticas aos atletas, oferecendo uma percepção detalhada do desempenho desse atleta no jogo.

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	<b>Formulação do Problema</b> Formulação e Contextualização do Problema, Especificações Funcionais e Arquitetura da Solução.
Capítulo 3	<b>Aplicação Servidora</b> Aspetos relacionados com a aplicação servidora, como abordagens, metodologias e detalhes de implementação.
Capítulo 4	<b>Aplicação Cliente</b> Aspetos relacionados com a aplicação cliente, como abordagens, metodologias e detalhes de implementação.
Capítulo 5	<b>Testes</b> Testes executados sobre as diversas vertentes do projeto.
Capítulo 6	<b>Conclusões</b> 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 cujos objetivos e funcionalidades estão descritos 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. Conceito de Perfil de Atleta;
2. Conceito de Perfil de Equipa Técnica;
3. Conceito de Jogo;
4. Conceito de Estatísticas de Jogo;
5. Conceito de Treino;
6. Conceito de Planos de Treinos Físicos;
7. Conceito de Calendário de Eventos;
8. Conceito de Torneio;
9. Conceito de Evento;

Estes conceitos refletem 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 que 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**

Esta é a segunda sub-secção desta secção, que aponta os conceitos de algumas especificações secundárias. Conforme a disponibilidade, são especificações que poderão ser inseridas no contexto da aplicação, nomeadamente:

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

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*.

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

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

A aplicação servidora é uma das duas partições do nosso projeto. É a partição 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, de modo a ser mais perceptível a maneira como os dados iriam ser guardados persistentemente e ligados entre si, e qual os comportamentos que essas ligações iriam gerar.

No Apêndice A pode-se observar a descrição do modelo de dados.

Como descrito na secção 2.3, a aplicação servidor 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-seção 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.

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 tabelas da base de dados, que constituem a camada do Modelo. Na sub-seção seguinte é explicado como é que se interage com estas entidades.

### 3.1.2 *Repository*

A camada *Repository*, referida nesta sub-seção 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 persistência na base de dados. Quando a aplicação interage com os repositórios (através da chamada a métodos de *queries*), o JPA gera as ligações à base de dados e garante a comunicação 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 troço 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         return eventRepository.findAll();
8     }
9
10    public Long postEvent(Event event){
11        return eventRepository.save(event).getId();
12    }
13
14    public Event findEventById(Long id){
15        return eventRepository.findById(id)
16            .orElseThrow(() -> new ResourceNotFoundException("Event", "Id", id));
17    }
18
19    public Long updateEvent(Event event){
20        eventRepository.findById(event.getId())
21            .orElseThrow(() -> new ResourceNotFoundException("Event", "Id", event.getId()));
22        return eventRepository.save(event).getId();
23    }
24
25    public void deleteEvent(Event event){
26        eventRepository.findById(event.getId())
27            .orElseThrow(() -> new ResourceNotFoundException("Event", "Id", event.getId()));
28        eventRepository.delete(event);
29    }
30 }

```

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, e contem um comportamento que será incremental ao longo da realização do projeto.

### 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 troço 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        logger.info("On method GET event/all");
13        return eventBusiness.findAllEvents();
14    }
15
16    @GetMapping ("/findById/{id}")
17    public Event findEventById(@PathVariable Long id){
18        logger.info("On method GET event/findById/{id} with id: "+ id);
19        return eventBusiness.findEventById(id);
20    }
21
22    @PostMapping ("/post")
23    public Long postEvent(@RequestBody Event event){
24        logger.info("On method POST event/post");
25        return eventBusiness.postEvent(event);
26    }
27
28    @PutMapping ("/update")
29    public Long putEvent(@RequestBody Event event){
30        logger.info("On method PUT event/update");
31        return eventBusiness.updateEvent(event);
32    }
33
34    @DeleteMapping ("/delete")
```

```

34 public ResponseEntity<?> deleteStats(@RequestBody Event event){
35     logger.info("On method GET event/all");
36     eventBusiness.deleteEvent(event);
37     return ResponseEntity.ok().build();
38 }
39 }

```

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* 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 um *Router* completo para todos os *endpoints* da nossa aplicação. Também esta camada tem comportamento que será incremental ao longo do desenvolvimento do projeto.



## 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

A aplicação cliente é a segunda partição do nosso projeto. É a partição onde se encontra a interface de utilizador, painéis de controlo e alguma lógica de negócio adicional.

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 import {Profile} from './profile';
2
3 export class Event {
4   constructor(
5     private id?: number,
6     private name?: string,
7     private description?: string,
8     private date?: Date,
9     private local?: string,
10    private profiles?: Profile[]
11  ) {
12    this.id = id ? id : 0;
13    this.description = description ? description : '';
14    this.date = date ? date : new Date(0);
15    this.local = local ? local : '';
16    this.name = name ? name : '';
17    this.profiles = profiles ? profiles : [];
18  }
19 }
```

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*.

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.

#### 4.1.1 *Angular Component*

Um *Component* em *Angular* é uma peça visual de uma aplicação, que pode estender desde uma página a uma tabela ou a um *menu*. Não existe qualquer requisito sobre que tipo de objeto o componente representa, e esse tipo é meramente para fins de organização da aplicação.

No caso da nossa aplicação cliente, cada especificação principal tem o seu *endpoint* e consequentemente o seu *Component*. Dado que algumas regras de negócio implicam que certos *endpoints* necessitem de componentes adicionais, como as estruturas *Modal* ou *Popover* 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* das páginas dos *endpoints*, que são invocados nas circunstâncias apropriadas.

Cada *Component* tem associado um *template*, que representa o ficheiro *HTML* com a vista do componente, e um ficheiro *TypeScript* que representa a instância da classe 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, como os métodos invocados pelos eventos de *click* de um botão ou *check/uncheck* de uma *checkbox*, métodos que chamam os serviços que fazem o *data-fetching*, métodos de *redirect* da página, ou métodos que invocam os *Controllers* dos componentes adicionais mencionados 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 re-utilizadas 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 contem todos os métodos que fazem as chamadas feitas à *web API* exposta pela aplicação servidora.

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

```
1  import { Injectable } from '@angular/core';
2  import { HttpClient, HttpHeaders } from '@angular/common/http';
3  import { Event } from '../classes/event';
4  import { Observable, throwError } from 'rxjs';
5
6  @Injectable ({
7    providedIn: 'root'
8  })
9
10 export class EventService {
11   private BASE_URL = 'http://localhost:8080/event';
12   private httpOptions = {
13     headers: new HttpHeaders({
14       'Content-Type': 'application/json',
15       Authorization: 'my-auth-token',
16       'Access-Control-Allow-Origin': '*'
17     })
18   };
19
20   constructor(private http: HttpClient) { }
21
22   getEvents(): Observable<Event[]> {
23     const url = `${this.BASE_URL}/all`;
24     return this.http.get<Event[]>(url, this.httpOptions);
25   }
26
27   getEventsById(id: any) {
28     const url = `${this.BASE_URL}/findById/${id}`;
29     return this.http.get(url, this.httpOptions);
30   }
31
32   postEvent(event: Event): Observable<Event> {
33     const url = `${this.BASE_URL}/post`;
34     return this.http.post(url, event, this.httpOptions);
35   }
36
37   updateEvent(event: Event): Observable<any> {
38     const url = `${this.BASE_URL}/update`;
39     return this.http.put(url, event, this.httpOptions);
40   }
41
42   deleteEvent(id: number): Observable<any> {
```

```

43     const url = `${this.BASE_URL}/delete/${id}`;
44     return this.http.delete(url, this.httpOptions);
45   }
46 }

```

Certas entidades dispõem também de um serviço próprio (agrupados no *package app/components/services*) que contêm todos os métodos que envolvem a lógica de regras de negócio adicionais dessas entidades. Podemos observar no trecho 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;
22   }
23 }
24 }

```

### 4.1.3 *Angular Data Binding*

O *Angular* também dispõe de um sistema de sintaxe 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> | Corresponde a ligar uma expressão que é calculada quando o <i>template</i> é gerado a um elemento <i>HTML</i> dentro desse <i>template</i> .<br>A sintaxe deste <i>binding</i> é <code>{{expression}}</code> .   |
| <i>Property</i>      | Corresponde a ligar 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> , que altera dinâmicamente o valor do elemento para o valor da propriedade, mesmo quando o valor da propriedade é alterado em <i>run-time</i> .<br>A sintaxe deste <i>binding</i> é <code>[target]="expression"</code> .                   |
| <i>Event</i>         | Corresponde a ligar um evento de um elemento <i>HTML</i> a uma declaração do <i>Component</i> . É uma ligação denominada <i>View-to-Source</i> , que conecta um evento na vista ( <i>click</i> de um botão, <i>check/uncheck</i> de uma <i>checkbox</i> ) a uma declaração (podendo ser um método do <i>Component</i> ).<br>A sintaxe deste <i>binding</i> é <code>(target)="statement"</code> . |
| <i>Two-Way</i>       | Corresponde a uma 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. Esta ligação é normalmente usada em Formulários. 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 programaticamente alguns aspetos visuais do *template* sem querer fazer essa ligação a propriedades no *Component*. Estes tipos especializados são

- Attribute*    Corresponde a ligar um atributo a um elemento *HTML* do *template*. Este tipo de ligação é pouco usual, porque é normalmente preferível fazer uma ligação de uma propriedade ao elemento. Mas em certos casos (por exemplo, quando se utiliza ARIA - Accessible Rich Internet Applications, ou SVG - Scalable Vector Graphics), o objetivo da ligação não é ligar uma propriedade ao elemento, visto que não existe uma propriedade neste contexto onde fazer a ligação, mas sim alterar atributos do elemento. Neste caso, utiliza-se o *Attribute Binding*. A sintaxe deste *binding* é igual ao *Property Binding*, substituindo o nome da propriedade pelo nome do atributo `[attr.attribute-name]="expression"`.
- Class*        Corresponde a ligar uma classe *CSS* a um elemento *HTML* do *template*. Esta ligação é usada quando se quer que um elemento do *template* possa ter dois tipos de classes *CSS*, alternáveis em *run-time* pelo valor de uma expressão. A sintaxe deste *binding* é igual ao *Property Binding*, substituindo o nome da propriedade pelo nome da classe *CSS* `[class.name]="expression"`.
- Style*        Corresponde a ligar um estilo específico a um elemento *HTML* do *template*. No contexto do nosso projeto, este *binding* é utilizado no formulário de adição de estatísticas de um atleta, onde o valor de uma estatística demonstra gradualmente através de cores diferentes o carácter do valor (cores na gama do azul e verde são consideradas cores positivas, apresentadas em estatísticas com valores com percentagem de sucesso acima dos 50%, enquanto que cores na gama do laranja e vermelho são consideradas cores negativas, apresentadas em estatísticas com valores com percentagem de sucesso abaixo dos 50%). A sintaxe deste *binding* é igual ao *Property Binding*, substituindo o nome da propriedade pelo nome do estilo `[style.name]="expression"`;

#### 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, e se esconde automaticamente quando o tamanho do écran é menor que um determinado *threshold*, ou se esconde manualmente em algumas páginas que apresentam elementos de grande dimensão (ex.: a tabela de *AthleteStats*).
- ion-tabs* Barra de navegação, que aparece no *footer* da página, com *icons* com *routing* para todas as páginas da aplicação. Apesar de indicar alguma redundância quando utilizado em conjunto com o *ion-split-pane*, em contextos onde este colapsa, como referido anteriormente, o *ion-tabs* torna-se o menu que melhor facilita a navegação pela aplicação. Este menu é lateralmente *scrollable*, tornando-o extremamente responsivo em qualquer tipo de dispositivo.
- ion-header* Barra de navegação, que aparece no *header* da página, com uma *ion-toolbar* onde podem ser inseridos botões ou segmentos. No contexto da nossa aplicação cliente, dependendo da página, este componente tem por norma um *back button*, botão de adicionar um elemento, botão de informação sobre a página e título da página. No caso do calendário, tem botões para o filtro e para a barra de procura, e em outros casos, segmentos com abas para os diversos tipos de informação a mostrar (calendário com *Upcomming*, *All* e *Past*, estatísticas com *Stats* ou *Graph*).

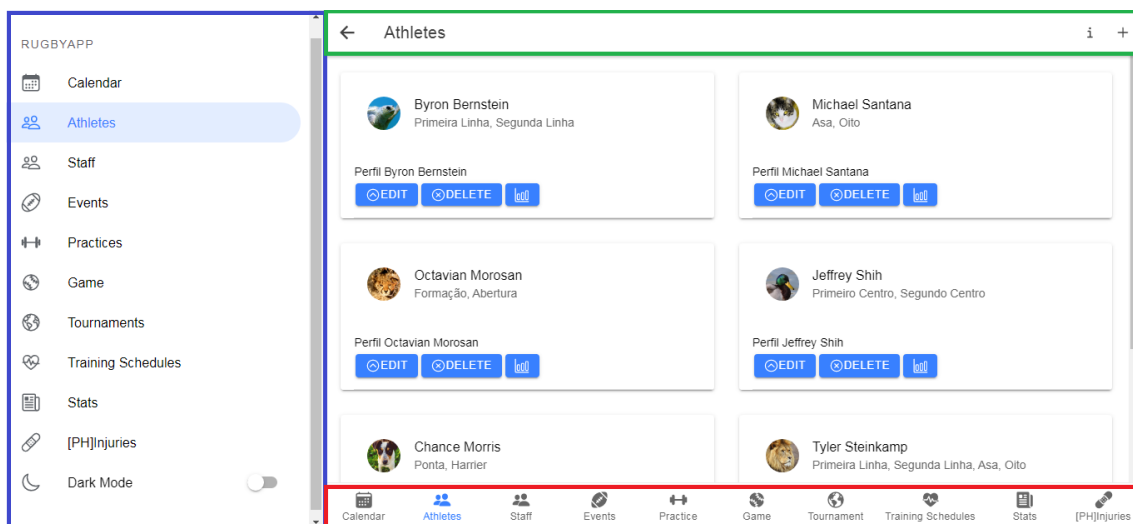


Figura 4.1: Figura da vista inicial da aplicação. Observa-se o *ion-split-pane* a azul, o *ion-tabs* a vermelho e o *ion-header* a verde.

Como mencionado na secção 4.1.1, existem *endpoints* 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. É utilizado no formulário de *Practices* para o utilizador escolher na folha de presenças os diversos tipos de treino suportados pelo nosso modelo que o atleta realizou.
- 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. É utilizado nas diversas tabelas com a lista de *Events*, *Games*, *Tournaments*, *Practices*, etc, para "esconder" a lista de presenças atrás de um botão, para que as listas fiquem mais organizadas e visualmente "limpas".
- ion-select* Apesar de não requerir 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, das quais este escolhe uma ou várias, dependendo do contexto, e o *ion-select* junta-as e guarda-as numa *string*). É utilizado nos nossos formulários para dar ao utilizador uma lista de escolhas possíveis, com que ele possa interagir para fazer as suas escolhas.

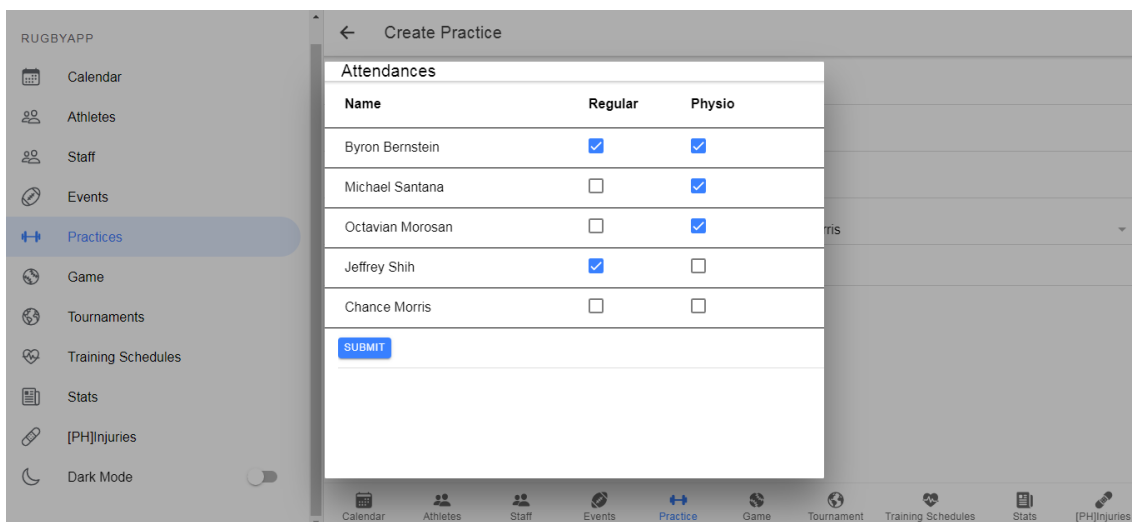


Figura 4.2: Figura do *Modal* do *endpoint* do formulário de *Practice*.

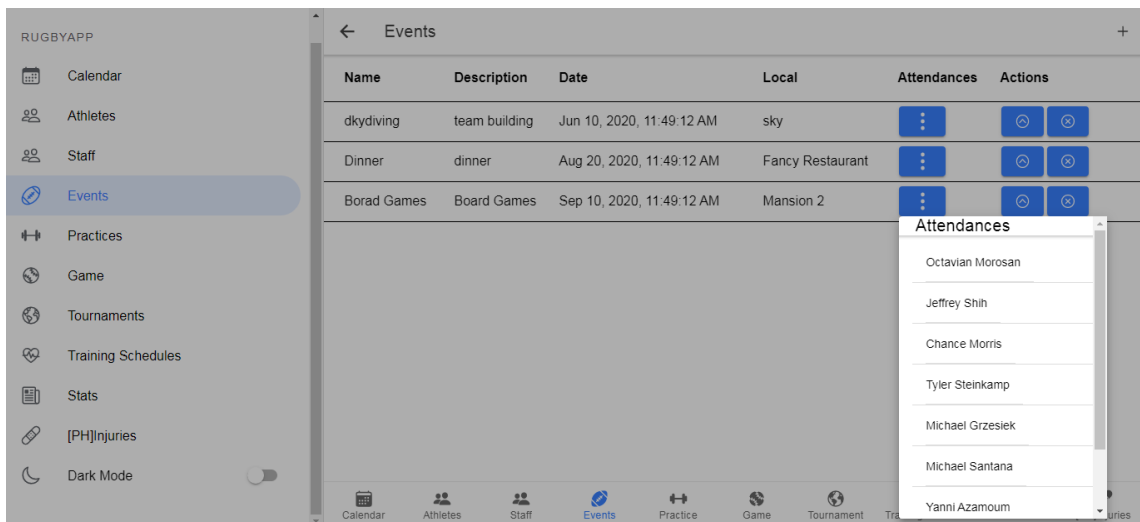


Figura 4.3: Figura do *Popover* do *endpoint Event*.

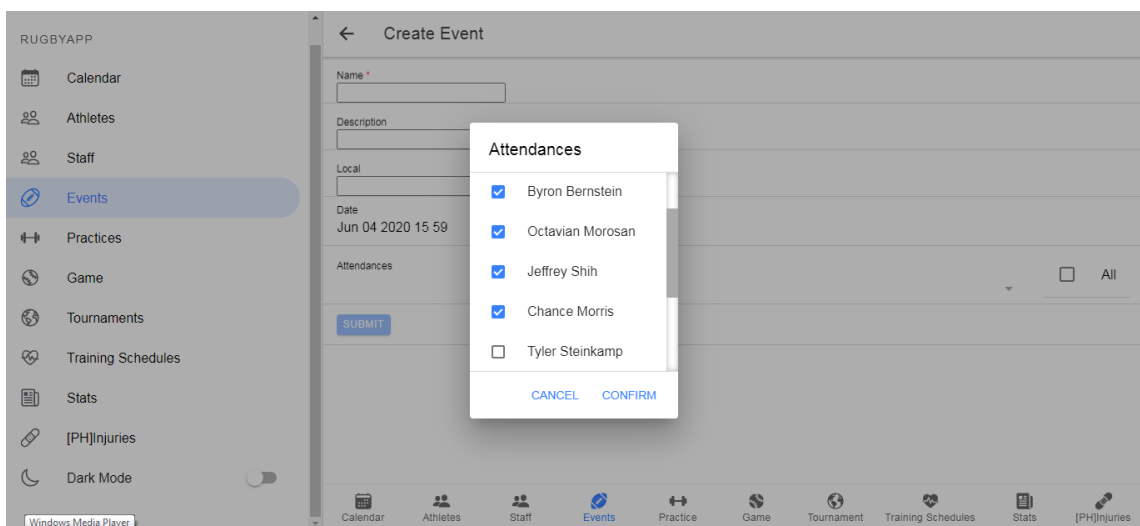


Figura 4.4: Figura do *Select* do *endpoint* do formulário de *Event*.

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*. Passando esse mesmo evento ao método *create* do *controller*, fazemos com que o *Popover* seja apresentado visualmente ligado ao botão que o criou, contrariamente a não ser passado um evento ao *create*, que cria apenas um *Popover* no topo da página.

```
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 do *endpoint Event*.

```
1 <ion-header>
2   <ion-title>Attendances</ion-title>
3 </ion-header>
4
5 <ion-content>
6   <ion-list>
7     <ion-item *ngFor="let profile of this.navParams.get('profiles')">
8       <ng-template [ngIf]="profile.athlete" [ngIfElse]="staff">
9         <ion-item detail="false" (click)="close()" routerLink="/app/athlete/
10         athlete-profile/{{profile.id}}">
11           <ion-label>
12             <h3>{{profile.name}}</h3>
13           </ion-label>
14         </ion-item>
```



```

14     </ng-template>
15     <ng-template #staff>
16         <ion-item detail="false" (click)="close()" routerLink="/app/staff/
staff-profile/{{profile.id}}">
17             <ion-label>
18                 <h3>{{profile.name}}</h3>
19             </ion-label>
20         </ion-item>
21     </ng-template>
22
23 </ion-item>
24 </ion-list>
25 </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 o *endpoint* desse *Profile*. É usado a terminologia *ngIf* do *Angular* para decidir entre gerar um *Link* para o perfil de um atleta ou de um *staff*.

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*, exceto que neste caso não se passa um evento para garantir que o *Modal* fica visualmente ligado ao componente que o invocou, visto que o *Modal* é sempre apresentado no meio da página.

#### 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 diversos <i>endpoints</i> da nossa aplicação cliente. Esta tabela foi escolhida pelo facto de ser possível 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.

Podemos observar no troço 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"
7    }} </td>
8
9    </ng-container>
10
11    <ng-container matColumnDef="local">
12      <th mat-header-cell *matHeaderCellDef mat-sort-header> Local </th>
13      <td mat-cell *matCellDef="let element"> {{element.local}} </td>
14    </ng-container>
15
16    <ng-container matColumnDef="opponent">
17      <th mat-header-cell *matHeaderCellDef> Opponent </th>
18      <td mat-cell *matCellDef="let element">
19        <ion-item>
20          <ion-thumbnail slot="start"> <img [src]="element.opponent.photo">
21        </ion-thumbnail>
22        <ion-label> {{element.opponent.name}} </ion-label>
23      </ion-item>
24    </td>
25  </ng-container>
26
27  <ng-container matColumnDef="score">
28    <th mat-header-cell *matHeaderCellDef> Score </th>
29    <td mat-cell *matCellDef="let element"> {{element.teamScore}} - {{
30  element.opponentScore}} </td>
31  </ng-container>
32
33  <ng-container matColumnDef="comment">
34    <th mat-header-cell *matHeaderCellDef> Comment </th>
35    <td mat-cell *matCellDef="let element"> {{element.comment}} </td>
36  </ng-container>
37
38  <ng-container matColumnDef="athletes">
39    <th mat-header-cell *matHeaderCellDef> Call </th>
40    <td mat-cell *matCellDef="let element">
41      <ion-button (click)="createPopover(element.athletes,$event)">
42      <ion-icon slot="icon-only" name="ellipsis-vertical"></ion-icon>
43    </ion-button>
44  </td>
45  </ng-container>
46
47  <ng-container matColumnDef="actions">
48    <th mat-header-cell *matHeaderCellDef> Actions </th>
49    <td mat-cell *matCellDef="let element">
50      <ion-button routerLink="/app/game/update/{{element.id}}">
51      <ion-icon name="chevron-up-circle-outline"> </ion-icon>
52    </ion-button>
53    <ion-button (click)="presentAlert(element)">
54    <ion-icon name="close-circle-outline"></ion-icon>
55  </ion-button>
56    <ion-button routerLink="/app/game/{{element.id}}/roster">
57    <ion-icon name="person-add-outline"></ion-icon>
58  </ion-button>
59  </td>
60  </ng-container>
61
62  <tr mat-header-row *matHeaderRowDef="displayedColumns"></tr>
63  <tr mat-row *matRowDef="let row; columns: displayedColumns;"></tr>
64</table>
65</ion-content>

```

As *mat-table* do *Angular Materials* funcionam com base num objeto *DataSource*. O *Component* de cada *endpoint* 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 }
```

Cada *Component* também tem um *decorator* *@ViewChild* para injetar uma referência para os elementos do *template* do *MatSort* nas tabelas. Afetando a propriedade *sort* do nosso *dataSource* para esta referência, conseguimos ativar a funcionalidade de ordenação nas nossas colunas. Como é observável no troço de código *HTML* da tabela, as colunas que são ordenáveis contém o atributo *mat-sort-header*. Quando a tabela é carregada na página da aplicação, ao meter o cursor por cima da coluna, se esta for ordenável, é visível uma seta preta. Cada *click* na coluna altera o estado desta seta, entre apontar para cima (a tabela está ordenada crescentemente pelos valores da coluna), apontar para baixo (a tabela está ordenada decrescentemente pelos valores da coluna) ou desaparecer (e a tabela fica organizada mediante o seu estado inicial).

← Games + +




















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

Figura 4.5: Figura do da tabela do *endpoint* de *Games*, ordenada crescentemente por nome.

← 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 4.6: Figura do da tabela do *endpoint* 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  <mat-grid-tile [colspan]="1" [rowspan]="1">
14    <ion-label>
15      <div class="ion-text-center"> <b> Tipo de Staff </b> </div>
16      <div class="ion-text-center"> {{staff?.staffType.name}} </div>
17    </ion-label>

```

```

18 </mat-grid-tile>
19
20 <mat-grid-tile [colspan]="1" [rowspan]="1">
21   <ion-label>
22     <div class="ion-text-center"> <b> Numero de Staff </b> </div>
23     <div class="ion-text-center"> {{staff?.staffNumber}} </div>
24   </ion-label>
25 </mat-grid-tile>
26
27 <mat-grid-tile [colspan]="1" [rowspan]="1">
28   <ion-label>
29     <div class="ion-text-center"> <b> Data de Nascimento </b> </div>
30     <ion-datetime displayFormat="MMM DD, YYYY" value="{{staff?.profile.
31       birth}}"></ion-datetime>
32   </ion-label>
33 </mat-grid-tile>
34
35 <mat-grid-tile [colspan]="1" [rowspan]="1">
36   <ion-label>
37     <div class="ion-text-center"> <b> Morada </b> </div>
38     <div class="ion-text-center"> {{staff?.profile.address}} </div>
39   </ion-label>
40 </mat-grid-tile>

```

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.


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

Figura 4.7: Figura do *Profile* de um *Staff*, com as linhas da grelha *highlighted* para fins de visualização.

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

## 4.2 Descrição dos *Endpoints*

Esta secção aborda os componentes dos diversos *endpoints* implementados na aplicação cliente.

### 4.2.1 *Main Component*

O Angular gera automaticamente um *root Module*, chamado *AppModule*, e um *AppComponent*, que servem de raiz a todos os componentes e módulos da aplicação. O nosso *AppComponent* contém, como foi mencionado anteriormente e é observável na figura 4.1, um *ion-split-pane* e um *ion-tabs*. A instância da classe deste *Component* contém um *array AppPages*, onde cada objeto dentro desse *array* tem um *title*, um *url* e um *icon*, e o *template* deste *component* gera os diferentes botões do menu através de um *ngFor*. O *ion-tabs*, por outro lado, não é diretamente implementado no *AppComponent*. A nossa aplicação cliente parte o *routing* em diversos componentes, sendo o principal o *ion-tabs*. Foi implementado a maioria do *routing* para os diversos *endpoints* no *ion-tabs* para garantir que em dispositivos demasiado pequenos, em termos de tamanho de ecrã, para o *ion-split-pane* ser criado, que o *routing* da aplicação não seria comprometido. Por outras palavras, a entrada na aplicação dá-se, em termos de *routing*, pelo *ion-split-pane*, e todos os *endpoints* a partir da entrada passam pelo *ion-tabs*, para garantir a) que este componente é sempre criado, e b) o *routing* da aplicação é sempre mantido em qualquer circunstância.

### 4.2.2 Calendar

O *Calendar*, referido nesta sub-secção como Calendário, é o *endpoint* principal da aplicação cliente. É a primeira página a ser mostrada quando um utilizador entra na aplicação.

UPCOMMING	ALL	PAST	Q	≡
May 13, 2020, 7:30:00 PM Practice at Track1				
May 15, 2020, 1:33:17 PM Game vs Galacticos at madrid				
May 22, 2020, 1:33:17 PM Game vs SuperTeam at unsiverse				
May 27, 2020, 3:00:00 AM Rui Miguel Marques Garcia at Lisbon				
May 29, 2020, 1:33:17 PM Game vs Team1 at seol				
Jun 10, 2020, 11:49:12 AM Event at sky				
Jun 10, 2020, 11:49:12 AM Torneio das Caldas at somewhere				
Jun 27, 2020, 12:00:00 AM Practice at Field 1				
Jul 11, 2020, 1:33:17 PM Game vs Team do Boss at nowhere				
Aug 20, 2020, 11:49:12 AM Event at Fancy Restaurant				
Sep 10, 2020, 11:49:12 AM Event at Mansion 2				
Calendar Athletes Staff Events Practice Game Tournament Training Schedules Stats [PH]Injuries				

Figura 4.8: Figura da aba *All* do Calendário.

O Calendário mostra todas as entradas de *Games*, *Events*, *Practices* e *Tournaments*, através da sua data e localização.

O Calendário contem as abas *Upcomming*, *All* e *Past*. A aba *Upcomming* mostra todas as entradas que irão decorrer nos próximos N dias, sendo o valor por omissão de N 15 dias. No menu de filtro existe um campo onde é possível aumentar ou diminuir este valor de N, consoante a informação que se quer mostrar. A aba *Past* mostra todas as entradas cuja data de concretização já ocorreu. A aba *All* mostra todas as entradas. No caso da aba *Upcomming*, o calendário também mostra o tempo, em semanas, dias, horas, e minutos, até a entrada de concretização.





O Calendário implementa uma barra de pesquisa, que pesquisa por nome, local ou oponente (caso se aplique), e apresenta apenas as entradas encontradas nessa pesquisa. Esta pesquisa é aplicada às três abas do Calendário.

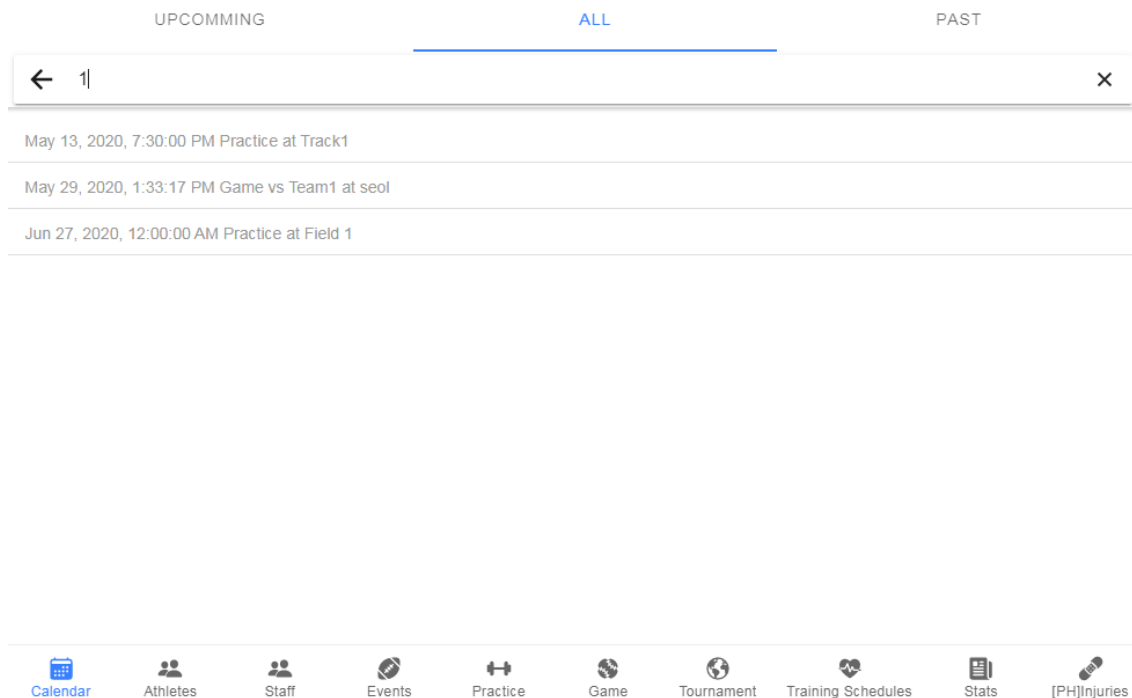


Figura 4.11: Figura da barra de pesquisa do Calendário. A pesquisa pela *string* "1" mostra todas as entradas onde o nome, local ou oponente contém essa *string*.

O Calendário implementa um menu de filtros, que permite filtrar por *Games*, *Events*, *Practices* ou *Tournaments*. Este filtro é aplicado às três abas do Calendário. Como referido anteriormente, também permite filtrar o número de dias de alcance da aba *Upcomming*.

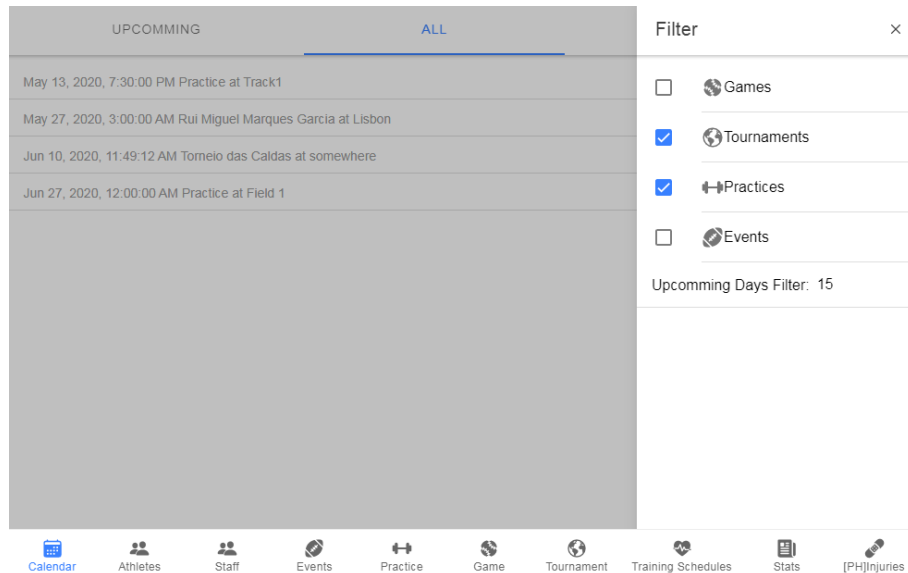


Figura 4.12: Figura do Calendário filtrado sem *Games* e *Events*.

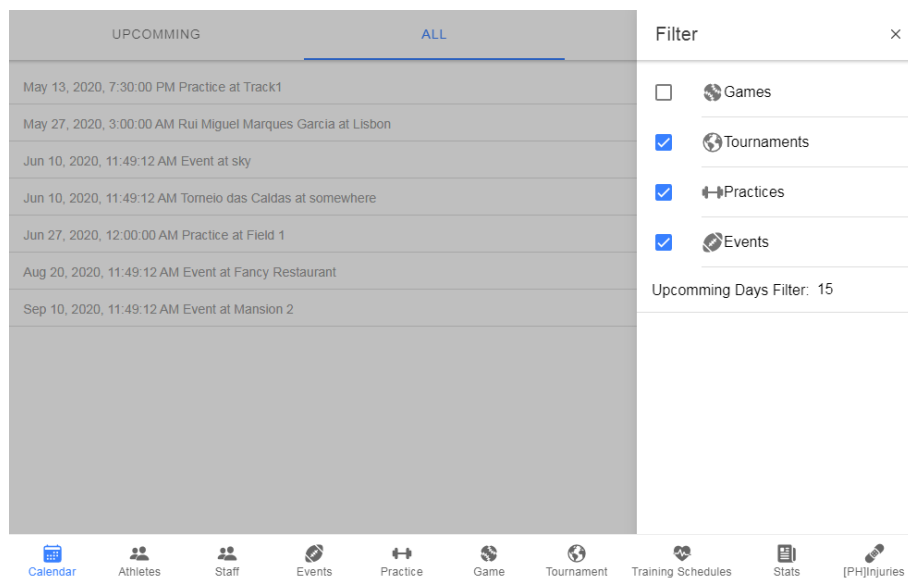


Figura 4.13: Figura do Calendário filtrado sem *Games*.

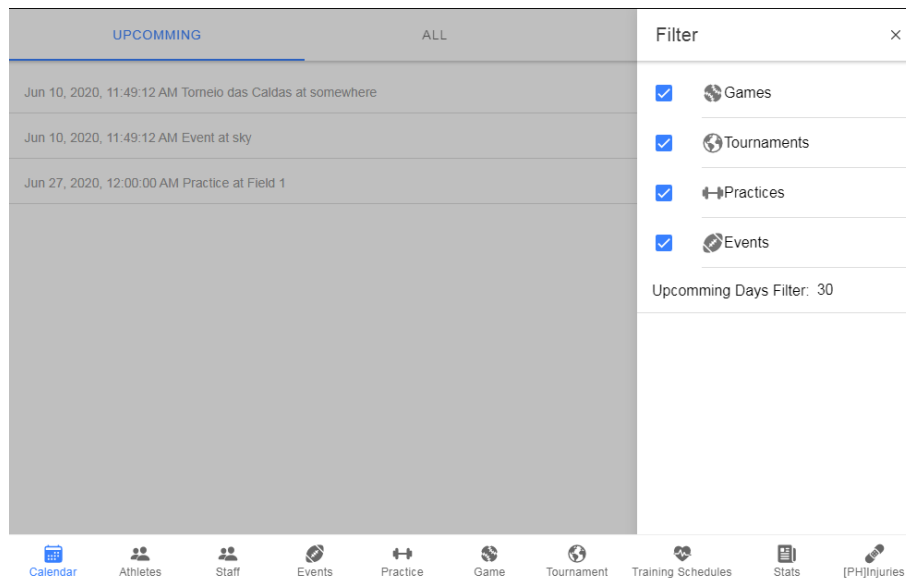


Figura 4.14: Figura da aba *Upcomming* filtrada com 30 dias.

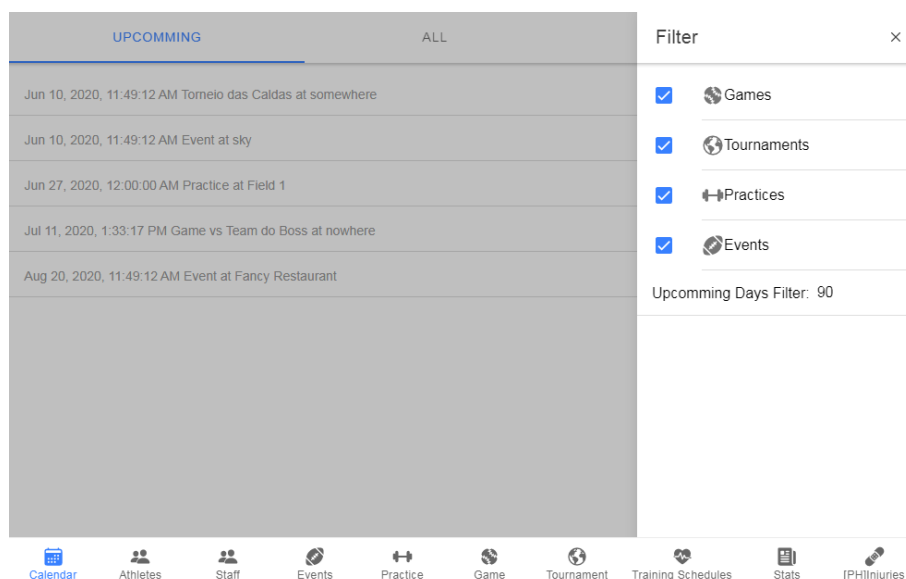


Figura 4.15: Figura da aba *Upcomming* filtrada com 90 dias.

Programaticamente, o Calendário recebe a informação através de um *forkJoin*, garantido que o *subscribe* do componente só é realizado quando a informação dos quatro pedidos aos quatro *endpoints* da *web API* da aplicação servidora forem concretizados no seu total. O Calendário contem 5 *arrays* para guardar a informação que vai demonstrar, sendo estes o *info*, *show*, *showFilter*, *past* e *upcomming*.

O *info* é o *array* que é retornado do *subscribe*, que contem toda a informação agrupada num unico objeto.

O *show* é uma cópia do *info*, para a informação que veio da *web API* não ser diretamente

alterada.

O *showFilter* é o *array show* filtrado com a *string* adquirida da barra de pesquisa.

O *past* e *upcomming* agem como filtros do *show*, onde um é um filtro por data com base no valor da variável *upcomming\_days* (que é alterável no campo do *menu* de filtro), e outro é um filtro por data com base na data corrente. Ambos são gerados através dos métodos *getPast()* e *getUpcomming()*. Ambos estes filtros observam o valor da barra de pesquisa, guardado numa variável *searchtext*, para contextualizar se devem ser gerados a partir do *show* ou do *showFilter*.

O Calendário tem um método *getDisplayInfo()*, que deduz através de uma variável *segment* com o nome da aba que está ativa no momento, qual dos *arrays* deve mostrar.

```
1 export class CalendarComponent implements OnInit {
2   upcoming_days = 15;
3   segment = 'upcomming';
4   searchtext = '';
5   showSearchbar: boolean;
6   info: (Event|Game|Practice|Tournament)[] = [];
7   show: (Event|Game|Practice|Tournament)[] = [];
8   showFilter: (Event|Game|Practice|Tournament)[] = [];
9   upcomming: (Event|Game|Practice|Tournament)[] = [];
10  past: (Event|Game|Practice|Tournament)[] = [];
11
12  constructor(private calendarService: CalendarService, private
13    menuController: MenuController) { }
14
15  ngOnInit() {
16    this.getInfo();
17  }
18
19  getInfo() {
20    this.calendarService.getInfo().subscribe( resList => {
21      this.info.push.apply(this.info, resList[0]);
22      this.info.push.apply(this.info, resList[1]);
23      this.info.push.apply(this.info, resList[2]);
24      this.info.push.apply(this.info, resList[3]);
25      this.info = this.calendarService.sortByDate(this.info);
26      this.show = this.info;
27      this.showFilter = this.info;
28      this.getUpcomming();
29      this.getPast();
30    });
31  }
32
33  getPast() {
34    this.past = this.searchtext ? this.calendarService.getPast(this.
35      showFilter)
36      : this.calendarService.getPast(this.show);
37  }
38
39  getUpcomming() {
40    this.upcomming = this.searchtext ?
41      this.calendarService.getUpcomming(this.showFilter,
42      this.calendarService.addDays(new Date(), this.upcomming_days)) :
43      this.calendarService.getUpcomming(this.show,
44      this.calendarService.addDays(new Date(), this.upcomming_days));
45  }
```

```

44  getDisplayInfo() {
45      if (this.segment === 'past') return this.past;
46      if (this.segment === 'upcomming') return this.upcomming;
47      return this.searchtext ? this.showFilter : this.show;
48  }
49  /* . . . */
50  }

```

Cada *checkbox* do menu de *filtros* está ligada a um método próprio (*gameCheckChange()*, *eventCheckChange()*, etc.) que faz *listen* ao evento de *check/uncheck* e passa esse valor como um *boolean* a um outro método de *filter* (*filterGame()*, *filterEvent()*, etc.). Este método de *filter* deduz através do valor recebido se deve filtrar ou não-filtrar a informação. Exemplificando para o caso da *checkbox* de *Games* ser *checked*, ou seja, quando a informação a mostrar deve apresentar *Games*, o método chama o método *filterGames()* do *CalendarService* sobre a *info*, que vai retornar todos os *Games* encontrados na informação que veio da *web API* e faz *push* destes *Games* todos para o *show*, ordenando-o posteriormente por data. Se a mesma *checkbox* for *unchecked*, o método irá em vez disto chamar o método *unfilterGames()* do *CalendarService* sobre o *show*, que irá remover todas as entradas *Game* deste *array*. Em qualquer um dos casos, chama *getPast()* e *getUpcomming()* no fim para atualizar também os *arrays* com a informação das outras abas.

```

1  filterGame(bool) {
2      if (bool) {
3          this.show.push.apply(this.show, this.calendarService.filterGame(this.info));
4          this.show = this.calendarService.sortByDate(this.show);
5      }
6      else this.show = this.calendarService.unfilterGame(this.show);
7      this.getUpcomming();
8      this.getPast();
9  }
10
11 gameCheckChange(e) {
12     let newState = !e.currentTarget.checked;
13     this.filterGame(newState);
14 }

```

Já a barra de pesquisa, é implementada de uma maneira menos linear. É necessário considerar, quando se faz uma pesquisa, que o utilizador pode cancelar por completo a barra de pesquisa (e a pesquisa deve ser removida), e que a cada caractere que este insere/remove, ambas as 3 abas têm de ser atualizadas com o filtro. A barra de pesquisa está ligada a um método *searchFilter()*, que é chamado cada vez que o evento da barra de pesquisa é invocado (ou seja, cada vez que o utilizador insere ou remove um caractere da pesquisa). Este método lê o valor do evento que foi invocado, e afeta a variável *searchtext* com esse valor, ou com *undefined* se a barra de pesquisa estiver vazia (para suplementar a lógica de que as abas *Upcomming* e *Past* devem filtrar o *array show* em vez do *array showFilter*, caso não exista pesquisa). Se existir texto na barra de pesquisa, o método invoca o *searchFilter()* do *CalendarService*, para filtrar o *show* por texto. Este *searchFilter()* filtra apenas por nome,

local e/ou oponente das entradas do *show*. O resultado do filtro é inserido no *showFilter*. No caso da barra de pesquisa não contiver texto, o *showFilter* passa apenas a ser igual ao *show*. É mais uma vez chamado o *getPast()* e *getUpcomming()* para atualizar a informação das outras abas. Para cobrir o facto do utilizador poder simplesmente cancelar a barra de pesquisa, existe um método *onCancel()* que afeta o *searchtext* com *undefined* e o *boolean searchBar* com *false*. Depois chama o *getPast()* e *getUpcomming()*, que mais uma vez, como o *searchtext* está a *undefined*, irá filtrar pelo *show* em vez de pelo *showFilter*. Isto faz com que o Calendário volte ao seu estado inicial quando o utilizador cancela a barra de pesquisa.

```

1 searchFilter(event) {
2   this.searchtext = event.target.value.toLowerCase() === '' ? undefined :
     event.target.value.toLowerCase();
3   if (this.searchtext) this.showFilter = this.calendarService.searchFilter(
     this.show, this.searchtext);
4   else this.showFilter = this.show;
5   this.getUpcomming();
6   this.getPast();
7 }
8
9 onCancel() {
10  this.showSearchbar = false;
11  this.searchtext = undefined;
12  this.getUpcomming();
13  this.getPast();
14  this.getDisplayInfo();
15 }

```

### 4.2.3 *Athletes/Staff*

Ao contrário dos restantes componentes, o *endpoint* de *Athletes* e *Staff* é apresentado utilizando uma listagem diferente. Foi decidido que, dado que uma lista de atletas e *staff* apresenta informação que é, visualmente, mais facilmente perceptível utilizando as fotografias dos perfís, optá-mos por fazer uma grelha com *ion-cards*. Um *ion-card* serve essencialmente como uma peça visual que agrupa a informação mais importante sobre um perfil, e serve de ponto de entrada para informação mais detalhada. Podemos utilizar uma colecção de *ion-cards* para agrupar toda a informação que é mais relevante numa lista de atletas sem obrigar o utilizador a entrar para dentro de um perfil. Cada cartão tem o nome e imagem do atleta, as posições em que joga, e botões que sirvam para modificar a informação daquele atleta (*edit* e *delete*), assim como um botão que redireciona o utilizador para o *endpoint* das estatísticas. Isto dá ao utilizador um panorama geral dos atletas e das suas posições, com possibilidade de alterar e remover atletas da lista sem necessitar de visualizar perfís.

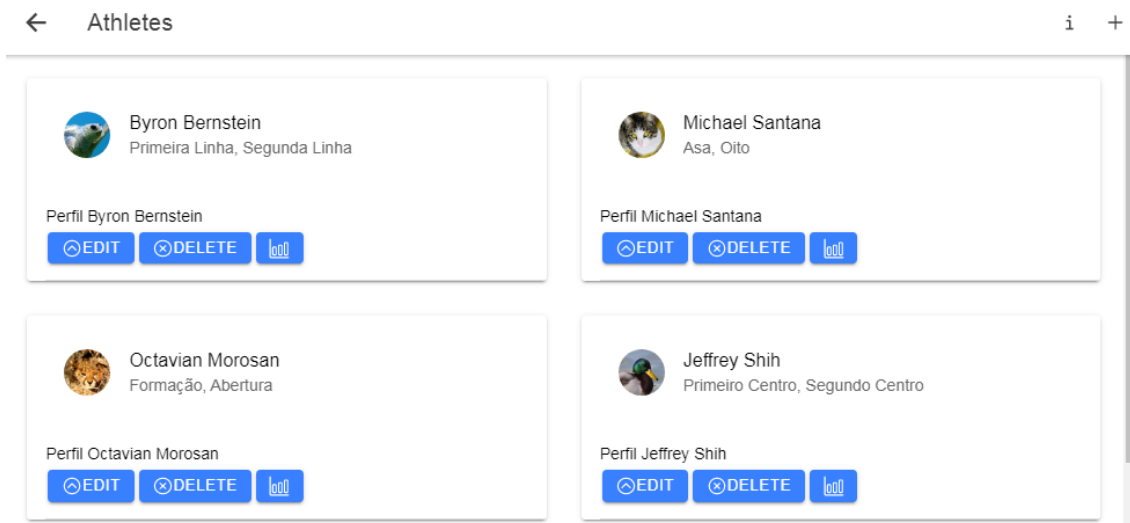


Figura 4.16: Figura do *endpoint* da lista de *Athletes*.

No cabeçalho deste *endpoint*, são também observáveis o *back button* no canto superior esquerdo, que redireciona para a página anterior, o *info button* (**não implementado**) e o *add button*, que redireciona para o formulário de atleta, discutido na sub-seção 4.2.8.

#### 4.2.4 *Profile*

O *endpoint* de *Profile* é também diferenciado dos outros *endpoints* da aplicação cliente, por não mostrar uma lista de entidades, mas sim todos os dados de uma entidade específica. Foi decidido, como referido na sub-seção 4.1.6, usar-se uma *mat-grid-list* para implementar os perfis de atletas e equipa técnica, pois a *mat-grid-list* é um grelha que facilita a organização de colunas e linhas, sendo possível expandir umas e encolher outras sem comprometer visualmente a grelha.

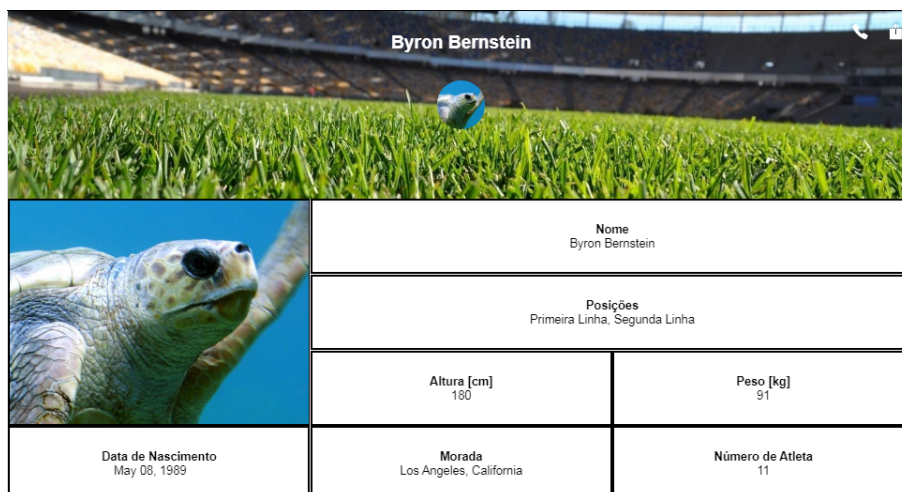


Figura 4.17: Figura do *endpoint* do *Profile* de um *Athlete*.

O perfil de um atleta organiza assim, numa grelha 3x4, toda a informação do atleta. Também se observa, no cabeçalho do perfil, uma imagem de fundo (**irá ser alterada no futuro**), e os botões de *Contact*, com o e-mail e o número de telefone do atleta para fins de entrar em contacto com este, assim como um *Share* (**não implementado**).

#### 4.2.5 *AthleteStats*

O *endpoint* de estatísticas de atleta é também diferenciado dos restantes *endpoints* com tabelas por conter uma tabela com maior dimensão, dada a carga de dados que tem de apresentar. Neste *endpoint* em particular, o *split-pane* é escondido manualmente pela aplicação, para conseguir manter a tabela toda no ecrán.

```

1  ionViewDidEnter() {
2      this.menuController.enable(false, 'main-menu');
3  }
4
5  ionViewDidLeave() {
6      this.menuController.enable(true, 'main-menu');
7  }

```



Para entender melhor estes métodos, é explicado em seguida o *Ionic Lifecycle* de cada 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 nossas variáveis.
<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.

É possível, através deste *Lifecycle*, definir quando é que alguns comportamentos são executados. Neste caso, o *split-pane* é escondido no gancho *ionViewDidEnter* e volta a ser mostrado no gancho *ionViewDidLeave*.

Como esta tabela contem um elevado número de colunas, foram usadas siglas para o nome de cada coluna.

Game	E	F	RC	YC	TO	Tr	M	PT	Ta H	Ta M	Ta %	CK H	CK M	CK %	GK H	GK M	GK %	DK H	DK M	DK %	LO H	LO M	LO %	OK H	OK M	OK %
5/29 1:33 vs Team1	8	7	8	7	8	7	8	7	7	8	46.67	7	8	46.67	8	9	47.06	8	7	53.33	8	7	53.33	7	8	46.67
5/15 1:33 vs Galacticos	1	9	0	0	6	4	5	80	5	5	50	5	3	62.5	8	7	53.33	2	3	40	6	8	42.86	8	3	72.73
<b>Total</b>	<b>9</b>	<b>16</b>	<b>8</b>	<b>7</b>	<b>14</b>	<b>11</b>	<b>13</b>	<b>87</b>	<b>12</b>	<b>13</b>	<b>48</b>	<b>12</b>	<b>11</b>	<b>52.17</b>	<b>16</b>	<b>16</b>	<b>50</b>	<b>10</b>	<b>10</b>	<b>50</b>	<b>14</b>	<b>15</b>	<b>48.28</b>	<b>15</b>	<b>11</b>	<b>57.69</b>

Figura 4.18: Figura da tabela do *endpoint AthleteStats*.

No canto superior direito do cabeçalho da página, encontramos o botão *Subtitles*, que abre um menu lateral com a legenda da tabela.

The screenshot shows a mobile application interface. At the top, there's a header with a back arrow, a 'STATS' tab, and a 'GRAPH' tab. Below this is a table with columns for various statistics. A 'Subtitles' menu is open on the right, listing items like 'E - Errors', 'F - Fouls', 'RC - Red Cards', etc. At the bottom, there's a navigation bar with icons for Calendar, Athletes, Staff, Events, Practice, Game, Tournament, Training Schedules, Stats, and [PH]Injuries.

Game	E	F	RC	YC	TO	Tr	M	PT	Ta H	Ta M	Ta %	CK	HCK	MCK	CK %	GK H	GK M	GK %	DK H	DK M	DK %
5/29 1:33 vs Team1	8	7	8	7	8	7	8	7	7	8	46.67	7	8	46.67	8	9	47.06	8	7	53.33	
5/15 1:33 vs Galacticos	1	9	0	0	6	4	5	80	5	5	50	5	3	62.5	8	7	53.33	2	3	40	
<b>Total</b>	<b>9</b>	<b>16</b>	<b>8</b>	<b>7</b>	<b>14</b>	<b>11</b>	<b>13</b>	<b>87</b>	<b>12</b>	<b>13</b>	<b>48</b>	<b>12</b>	<b>11</b>	<b>52.17</b>	<b>16</b>	<b>16</b>	<b>50</b>	<b>10</b>	<b>10</b>	<b>50</b>	

Figura 4.19: Figura da tabela do *endpoint AthleteStats* com o menu da legenda aberto.

Existe também neste *endpoint* uma aba de *Graphs* (**não-implementada**), que irá futuramente conter informação gráfica das estatísticas.

#### 4.2.6 *Events/Practices/Games/Tournaments/TrainingSchedules*

Estes cinco *endpoints* são todos idênticos na sua estrutura, pois são os *endpoints* que apresentam as tabelas das diferentes entidades. Como foi mencionado na sub-seção 4.1.5, estes *endpoints* implementam todos uma *mat-table* ordenável com a informação das entidades. Dependendo do contexto das entidades, existem botões em cada linha com um *Popover* com a lista de participantes/presenças/convocados, e uma coluna de Acções, onde estão os botões de *Edit* e *Delete*.

The screenshot shows the 'Games' endpoint in the application. It features a table with columns: Date, Local, Opponent, Score, Comment, Call, and Actions. A 'Popover' menu is open over the 'Actions' column, showing a list of 'Attendances' with names like Byron Bernstein, Michael Santana, etc. The left sidebar shows navigation options, and the bottom bar has the same navigation icons as Figure 4.19.

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

Figura 4.20: Figura da tabela do *endpoint Games*, com o *Popover* aberto a mostrar os Convocados.

O endpoint de *Games* apresenta duas diferenças distintas para todos os outros: o botão de *ActiveRoster* e dois botões de *Add*.

Existem dois botões de *Add* porque, como a entidade *Opponent* é uma entidade fraca de *Game* e não existe contextualmente sozinha com o seu próprio *endpoint*, foi tomado como regra de decisão inserir o *endpoint* com o formulário de adicionar um oponente no *endpoint* dos jogos.

O *ActiveRoster* serve para definir os titulares no contexto de um jogo. O botão de *Add Active Roster* leva o utilizador para um formulário específico, onde são apresentadas as posições que os jogadores de Rugby podem cumprir, e cada posição abre um *ion-select* com uma lista filtrada de atletas que jogam naquela posição.

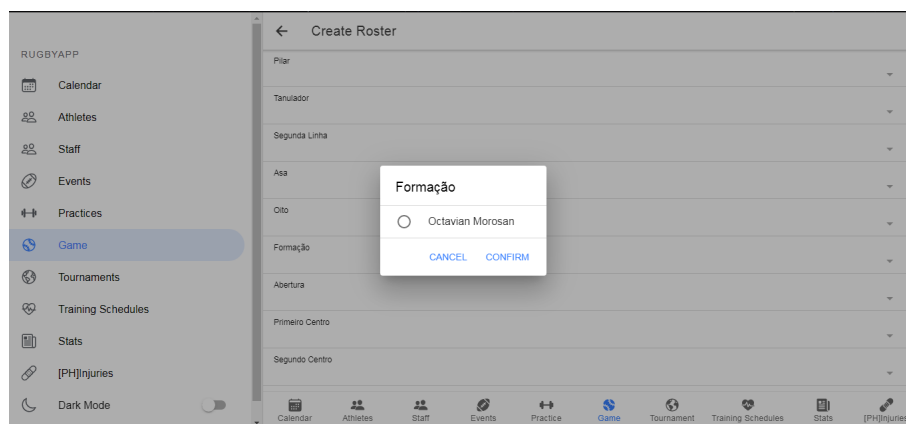


Figura 4.21: Figura da tabela do *endpoint ActiveRoster*, com a posição "Formação" selecionada, e o *ion-select* a mostrar a lista filtrada de atletas que jogam naquela posição.

Este formulário é implementado da seguinte maneira

```

1 <form (ngSubmit)="processRoster()">
2   <div *ngFor="let position of positions">
3     <ion-item>
4       <ion-label position="stacked"> {{position.name}} </ion-label>
5       <ion-select [(ngModel)]="selected" name="selected"
6         cancelText="Cancel" okText="Confirm" (ionChange)="onChange(position.id)">
7         <ng-template [ngIf]="athletes != undefined">
8           <ion-select-option
9             *ngFor="let athlete of athleteService.getAthletesByPosition(athletes,
10               position.name)"
11               [value]="athlete">
12               {{athlete.profile.name}}
13             </ion-select-option>
14           </ng-template>
15         </ion-select>
16       </ion-item>
17     <ion-item>
18       <ion-button type="submit" class="ion-no-margin">Submit</ion-button>
19     </ion-item>
20 </form>

```

RUGBYAPP

- Calendar
- Athletes
- Staff
- Events**
- Practices
- Game
- Tournaments
- Training Schedules
- Stats
- [PH]Injuries
- Dark Mode

Name	Description	Date	Local	Attendances	Actions
dkydiving	team building	Jun 10, 2020, 11:49:12 AM	sky		
Dinner	dinner	Aug 20, 2020, 11:49:12 AM	Fancy Restaurant		
Borad Games	Board Games	Sep 10, 2020, 11:49:12 AM	Mansion 2		

Calendar Athletes Staff **Events** Practice Game Tournament Training Schedules Stats [PH]Injuries

Figura 4.22: Figura da tabela do *endpoint Events*.

RUGBYAPP

- Calendar
- Athletes
- Staff
- Events
- Practices**
- Game
- Tournaments
- Training Schedules
- Stats
- [PH]Injuries
- Dark Mode

Date	Local	Comment	Attendances	Actions
Jun 27, 2020, 12:00:00 AM	Field 1	normal gear		
May 13, 2020, 7:30:00 PM	Track1	running gear		

Calendar Athletes Staff Events **Practice** Game Tournament Training Schedules Stats [PH]Injuries

Figura 4.23: Figura da tabela do *endpoint Practices*.

RUGBYAPP

Calendar

Athletes

Staff

Events

Practices

Game

**Tournaments**

Training Schedules

Stats

[PH]Injuries

Dark Mode

← Tournaments +

Name	Date	Local	Classification	Comment	Games	Actions
Rui Miguel Marques Garcia	May 27, 2020, 3:00:00 AM	Lisbon	3			⋮ ⌂ ⌂
Torneio das Caldas	Jun 10, 2020, 11:49:12 AM	somewhere	1	we won		⋮ ⌂ ⌂

Calendar Athletes Staff Events Practice Game Tournament Training Schedules Stats [PH]Injuries

Figura 4.24: Figura da tabela do *endpoint Tournaments*.

RUGBYAPP

Calendar

Athletes

Staff

Events

Practices

Game

Tournaments

**Training Schedules**

Stats

[PH]Injuries

Dark Mode

← Training Schedules +

Day	Description	Link	Targets	Actions
Jun 27, 2020	Gym 1	example.pt	⋮	⌂ ⌂
May 20, 2020	Gym 2	example.com	⋮	⌂ ⌂

Calendar Athletes Staff Events Practice Game Tournament Training Schedules Stats [PH]Injuries

Figura 4.25: Figura da tabela do *endpoint TrainingSchedules*.

#### 4.2.7 GameStats

O *endpoint* de *GameStats* é o endpoint principal das estatísticas de jogo. Enquanto que o *AthleteStats* mostra as estatísticas associadas a um jogador, o *GameStats* mostra e deixa adicionar estatísticas associadas a um jogador num jogo.

A primeira pagina deste *endpoint* apresenta a lista de jogos, similarmente à lista de *Athletes* ou *Staff*, através duma grelha de *ion-cards*. A diferença principal é que cada *ion-card* encaminha para o menu de estatísticas de jogo.

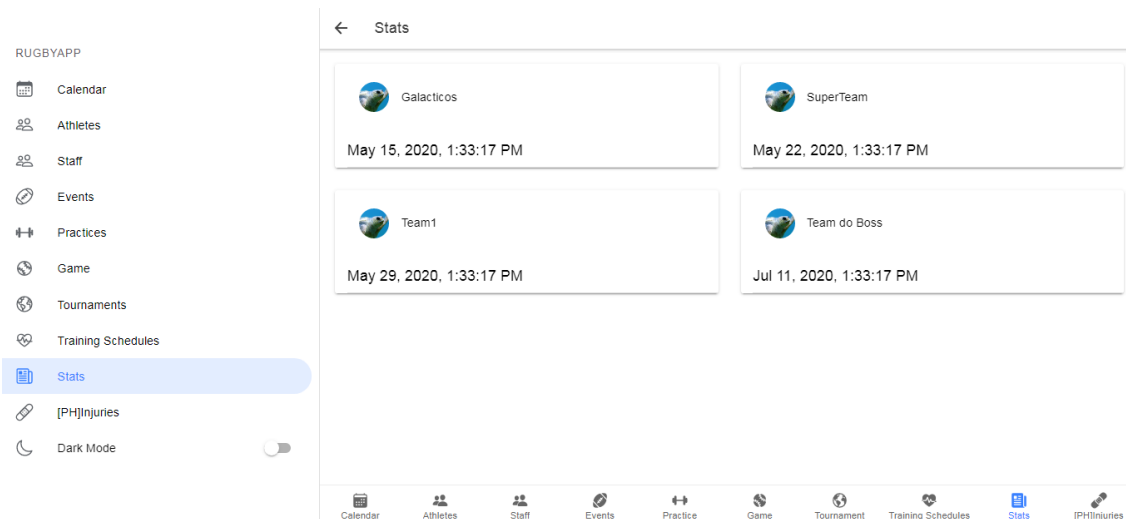


Figura 4.26: Figura da lista de *Games* do *endpoint GameStats*.

O menu de estatísticas de jogo apresenta uma lista de atletas associados ao jogo em contexto (**futuramente irá só apresentar os titulares**) e permite, para cada atleta, adicionar estatísticas.

O utilizador pode clicar em qualquer um dos atletas na lista, e este atleta é apresentado na zona de *selected* como o Atleta Ativo. Todas as estatísticas do menu são atualizadas com o valor das estatísticas deste atleta, e é possível alterá-las. As estatísticas são alteradas através de botões que incrementam o valor da estatística. No caso de estatísticas *Hit* e *Miss*, existem dois botões para incrementar cada um. No meio de cada item de cada estatística está o nome da estatística e o valor (no caso de *Hit* e *Miss* mostra ambos), e no caso também de *Hit* e *Miss*, encontra-se do lado esquerdo a percentagem de sucesso. Cada estatística tem também um botão de *Undo* (com as *Hit* e *Miss* a terem dois, um para *Undo* de um *Hit*, outro para *Undo* de um *Miss*). Estes botões existem para decrementar a estatística, em caso de erro humano.

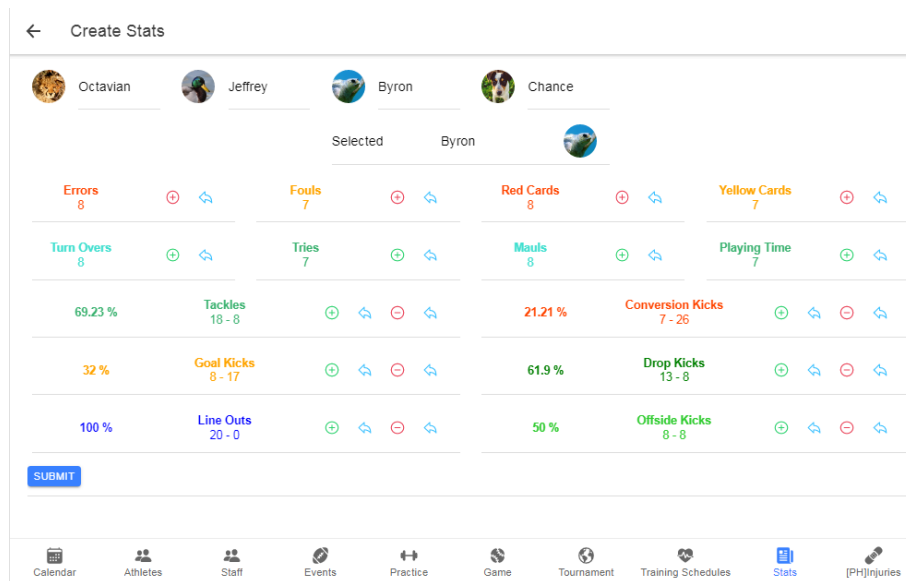


Figura 4.27: Figura do menu de estatísticas com o atleta "Byron" selecionado.

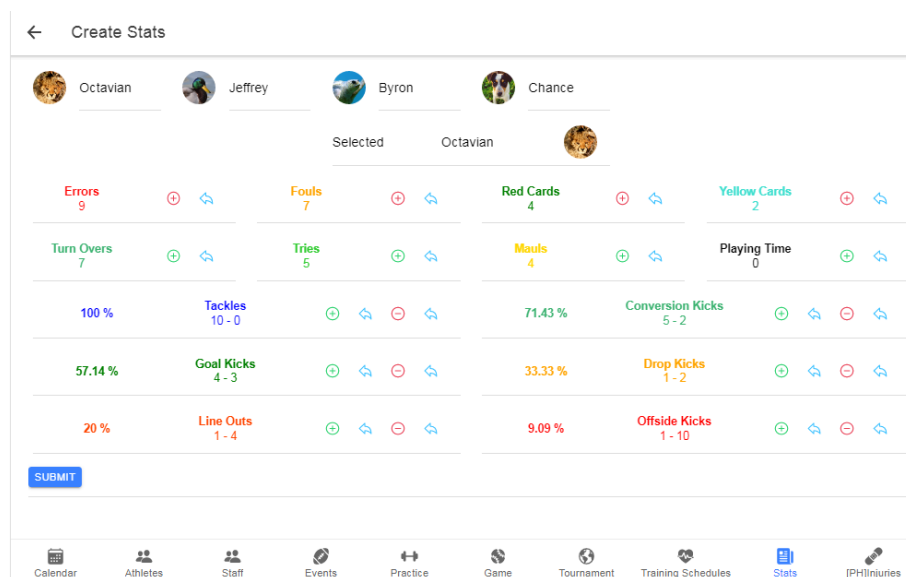


Figura 4.28: Figura do menu de estatísticas com o atleta "Octavian" selecionado.

Este *endpoint* apresenta um esquema de cores dinâmico. Como mencionado na sub-seção 4.1.3, é possível, através de um *style binding*, alterar dinamicamente o estilo de componentes *HTML*. No serviço *StatsService* foi definido um *spectrum* (*array* de *strings* com valores de cores ordenados espectralmente), e cada valor de cada estatística altera a cor do texto do item para a cor indicada no *spectrum*. Isto permite oferecer uma vertente visual às estatísticas de um jogador, dando uma percepção realista do seu desempenho sem considerar apenas os números. **(Este esquema está implementado estáticamente. O espectro de cores pode não corresponder à realidade de cada um, dado ser um tema emocional, e**

nem todos os jogos irão ter o mesmo grau de avaliação das estatísticas. Um jogo contra um oponente de nível inferior implica um maior peso negativo nos erros, e um jogo contra um oponente de nível superior implica um menor peso negativo nos erros).

#### 4.2.8 Forms

Os formulários são a principal maneira como o utilizador interage com a aplicação quando quer adicionar informação persistente (através dos *endpoints POST* ou *PUT*). O formulário apresenta uma quantidade variável de campos com *input* para o utilizador criar qualquer uma das Entidades.

No contexto da aplicação cliente, o nosso *endpoint* para os formulários é o mesmo para métodos de adicionar ou alterar uma entidade. No caso de o formulário ser acedido pelo *update*, os campos do formulário são pré-preenchidos pelos valores atuais do objeto da entidade. Se for acedido pelo *post*, os campos encontram-se vazios. Isto concretiza-se utilizando o módulo *ActivatedRoute* do *Angular*, de onde é possível extrair variáveis do *endpoint* que origina a criação do componente. Se o *endpoint* contiver um *id*, quer dizer que foi chamado no contexto de um objeto já existente, e por isso é um *update* e vai, através da *web API* da aplicação servidora, fazer *fetch* desse objeto para pré-preencher os campos.

The screenshot displays the 'Create Staff' form within the RUGBYAPP. The form is titled 'Create Staff' and contains several input fields. The 'Name' field is pre-filled with 'Rui Miguel Marques Garcia', 'Address' with 'somewhere street', 'Mail' with 'dsdsads@dsadsd dsasd', 'Phone' with '910000000', 'Birth' with 'May 27 2020', and 'Staff Number' with '123'. The 'Type' dropdown menu is set to 'Coach'. A blue 'SUMMIT' button is located at the bottom of the form. On the left, a sidebar menu shows 'Staff' as the selected option. The bottom navigation bar features icons for various app sections: Calendar, Athletes, Staff, Events, Practice, Game, Tournament, Training Schedules, Stats, and (PH)Injuries.

Figura 4.29: Figura do formulário de um *Staff* quando acedido através do botão *Edit* da Lista de *Staff*.

São utilizados alguns componentes visuais da *IONIC API* para garantir uma melhor experiência de utilizador no que toca a formulários



*ion-select* Explicado em detalhe na sub-secção 4.1.4

*ion-datetime* Componente visual para se escolher datas. Em vez de ser escrita manualmente, um *click* num *ion-datetime* abre um menu giratório onde o utilizador pode, através de *scrolls*, escolher os valores dos diversos campos da data. Em casos que se apliquem, como em Eventos e Jogos, o *displayFormat* da data é *MMM DD AAAA HH mm* para dar a opção de escolher a hora do evento, enquanto as datas de nascimento são em formato *MMM DD AAAA*.

Também é possível, através dos *class binding* e *style binding* e *attribute binding*, atribuir comportamento visual aos campos do formulário. No contexto da nossa aplicação, o botão de *submit* só fica *clickable* se o formulário estiver corretamente preenchido. Através de um *bind* à propriedade *disabled* do botão com o objeto que representa o formulário, pode-se adicionar este comportamento ao botão, para que ele só seja *enabled* quando todos os campos do formulário estão correctamente preenchidos (e só os que são *required*).

```

1 <ion-item>
2   <ion-button [disabled]="!loginForm.form.valid" type="submit" class="ion-no-
   margin">Submit</ion-button>
3 </ion-item>

```

Também é possível atribuir estilos *html* aos campos do formulário que estão preenchidos corretamente ou incorretamente. Gerando propriedades *css* para as classes *.ng-valid* e *.ng-invalid* que são utilizadas pelo próprio *Angular* na validação de formulários, podemos ter campos bem preenchidos com uma cor verde, e os mal preenchidos com vermelho.

```

1 .ng-valid[required], .ng-valid.required {
2   border-left: 5px solid #42A948;
3 }
4
5 .ng-invalid:not(form) {
6   border-left: 5px solid #a94442;
7 }

```

Figura 4.30: Figura do formulário de um *Staff* em preenchimento.

Podemos observar na figura 4.30 todos os aspetos importantes de um formulário. Os campos que são *required* ficam a vermelho quando não estão preenchidos, os campos que têm

padrões de escrita (um número de telefone requer pelo menos 9 dígitos, um e-mail requer o padrão *text@text.com*) ficam a vermelho quando o padrão não é cumprido corretamente. O botão *submit* encontra-se *disabled* pois os campos não estão todos corretos. Ao clicar na data de nascimento, abre-se o *ion-datetime*, observável também na figura.

#### 4.2.9 [WIP] *Injuries*

O *endpoint* de lesões ainda não se encontra implementado. Este tema representa uma especificação funcional que foi considerada anteriormente pelos elementos do grupo como secundária. No entanto, após algumas reuniões recentes com os clubes, foi decidido aumentar a prioridade desta especificação. Apesar de não haver ainda implementação para este tema, já foi possível aferir, através de uma reunião com o fisioterapeuta do Belas Rugby Clube, o modelo da entidade *Injuries*.

AthleteInjury

Lesão

Situação da lesão (inicial, em recuperação, recuperado)

Data de Início (mais virado para organização)

Data provisória de retorno do atleta

Recomendações do fisio (exercícios para fazer em casa, precauções)

Lesão

Nome da Lesão

Zona Corporal (orgão, osso, musculo, etc)

Severidade (há lesões iguais com graus de severidade diferentes)

Futuramente, este conceito irá ser inserido na aplicação servidora junto das outras Entidades, e a aplicação cliente irá ter um *endpoint* com uma lista de lesões. Possivelmente o perfil de um atleta irá também conter alguma desta informação.

# Capítulo 5

## Testes

Este capítulo aborda os testes executados no projeto.

### 5.1 Aplicação Servidora

Os testes feitos à partição da aplicação servidor foram executados sem a participação da aplicação cliente.

Para a execução destes testes, foram usadas duas ferramentas distintas:

*XAMPP* free open-source cross-platform web server solution stack para criar um servidor de testes.

*Postman* API cliente para gerar pedidos HTTP.

No ficheiro *application.properties* existente na *framework Spring* foram escritas as propriedades essenciais para garantir a ligação entre a aplicação e o servidor *XAMPP*.

```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://${MYSQL_HOST:localhost}:3306/test
spring.datasource.username=root
spring.datasource.password=
```

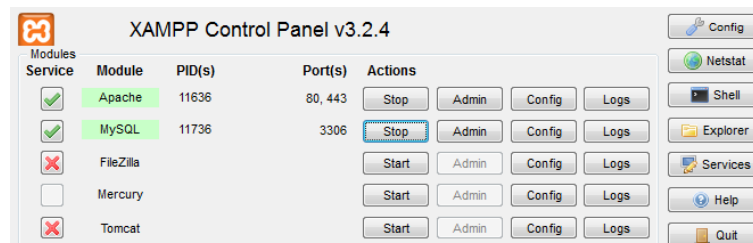


Figura 5.1: Figura do painel de controlo do *XAMPP*.

As seguintes sub-seções demonstram os vários testes aos *endpoints* da classe **Evento**.

### 5.1.1 Testes de GET

Os testes feitos aos *endpoints* de *GET* foram repetidos ao longo dos testes dos outros métodos para fins de observação de resultados, pelo que nesta sub-seção apenas se observam dois testes realizados antes da geração de informação de teste.

Teste : *GET ALL*

Endpoint : `http://localhost:8080/event/all`

Body : Nenhum

Resultado : Lista de objetos *JSON* de todos os Eventos.

Status : *200 OK*

Nota : Para efeitos de teste, foi colocado previamente um Evento para observação.

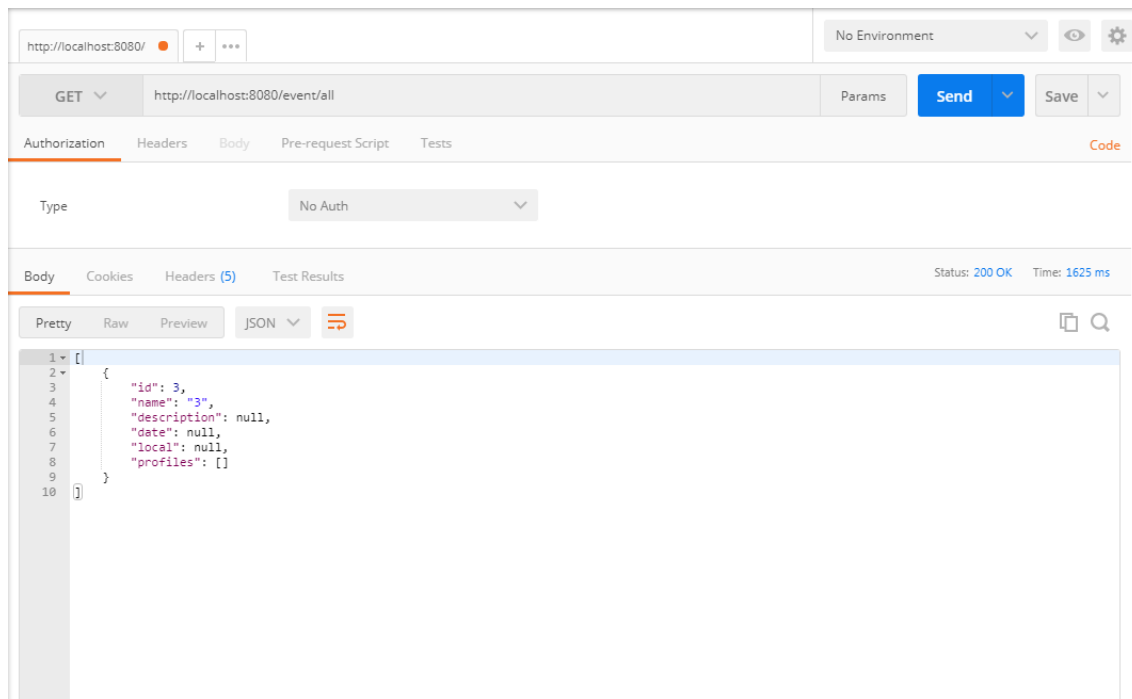


Figura 5.2: Resultado observável no *Postman*.

Teste : *GET BY EXISTENT ID*

Endpoint : `http://localhost:8080/event/findById/3`

Body : Nenhum

Resultado : Objeto *JSON* do evento com id 3.

Status : *200 OK*

Nota : Para efeitos de teste, foi colocado previamente um Evento com id 3 para observação.

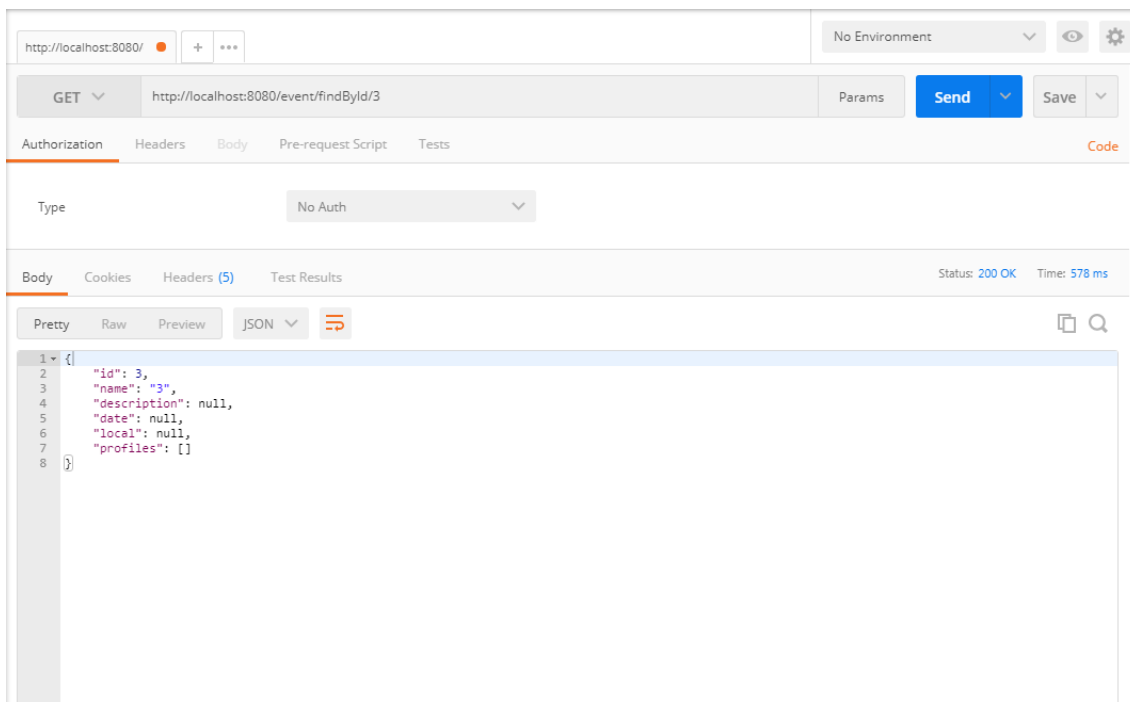


Figura 5.3: Resultado observável no *Postman*.

Teste : *GET BY NONEXISTENT ID*

Endpoint : `http://localhost:8080/event/findById/4`

Body : Nenhum

Resultado : Erro

Status : *404 NOT FOUND*

Nota : Para efeitos de teste, foi escolhido um id que não existe na base de dados.

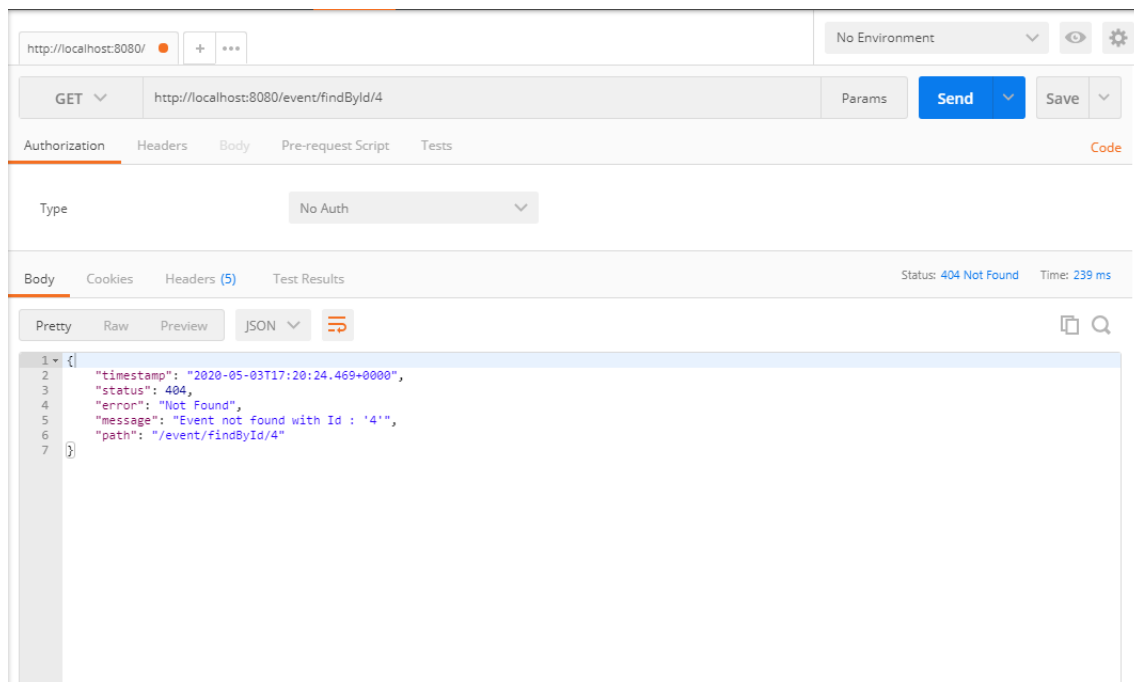


Figura 5.4: Resultado observável no *Postman*.

### 5.1.2 Testes de POST

Os testes feitos ao *endpoint* de *POST* são executados com um objecto *JSON* enviado no corpo do pedido. O *id* não é enviado juntamente com o resto da informação do objeto, pois é um campo gerado automaticamente e é retornado no corpo da resposta.

Teste : *POST*

*Endpoint* : `http://localhost:8080/event/post`

*Body* :

```
{
  "name": "TestEvent",
  "description": "Description Of TestEvent",
  "date": "2020-05-04T00:00:00.000Z",
  "local": "Portugal",
  "profiles": []
}
```

Resultado : *id* do objeto criado.

*Status* : *200 OK*

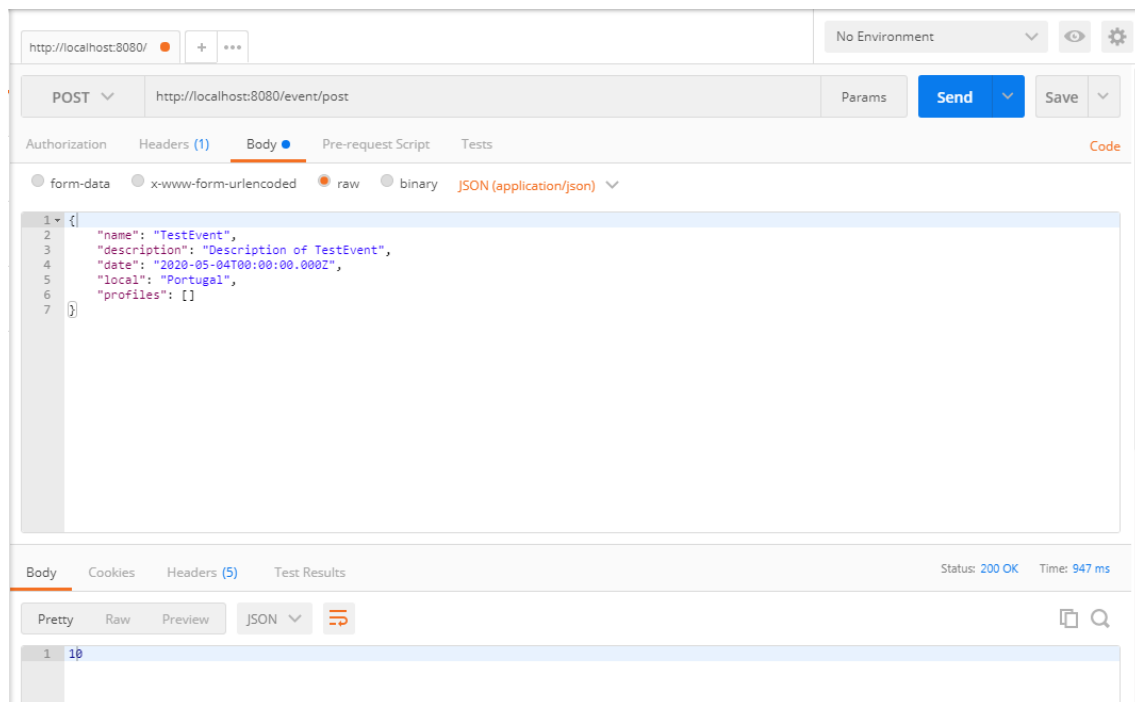


Figura 5.5: Resultado observável no *Postman*.

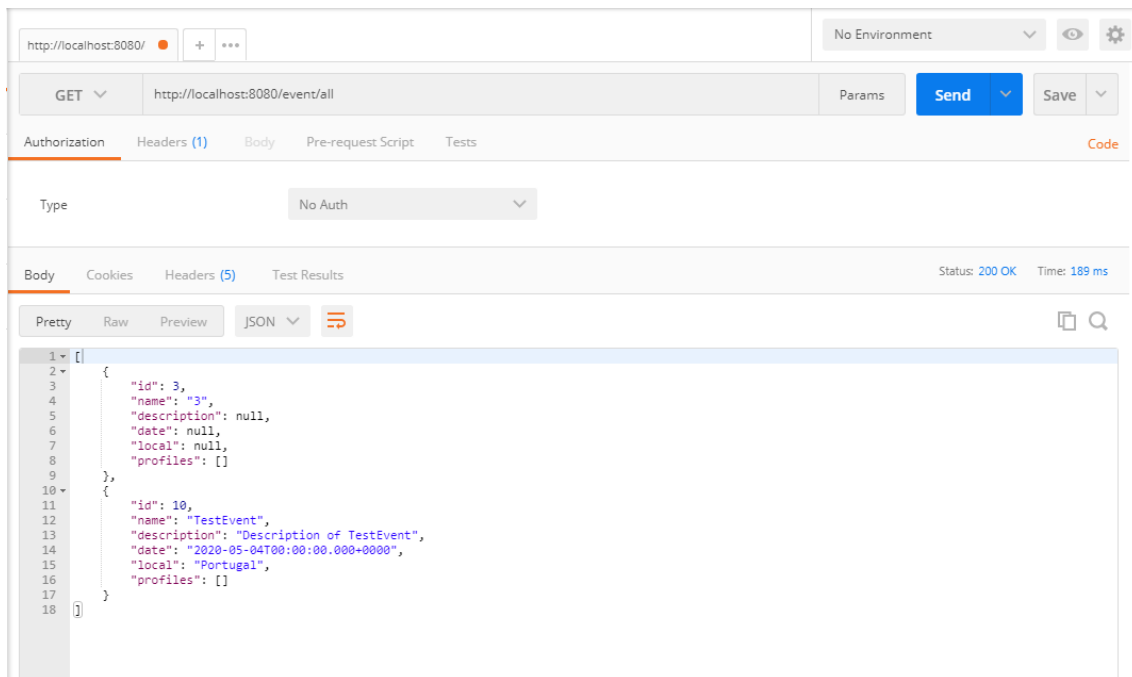


Figura 5.6: Resultado de um *GET ALL* depois do *POST* observável no *Postman*.



### 5.1.3 Testes de UPDATE

Os testes feitos ao *endpoint* de *UPDATE* são executados com um objecto *JSON* enviado no corpo do pedido com o id do objeto a ser alterado, assim como os campos a alterar. O id continua a ser retornado no corpo da resposta.

Teste : *UPDATE EXISTENT OBJECT*

Endpoint : `http://localhost:8080/event/update`

Body :

```
{
  "id": 10,
  "name": "TestEventUpdated",
  "description": "Updated Description Of TestEvent",
  "date": "2020-05-05T00:00:00.000Z",
  "local": "Spain",
  "profiles": []
}
```

Resultado : id do objeto alterado.

Status : *200 OK*

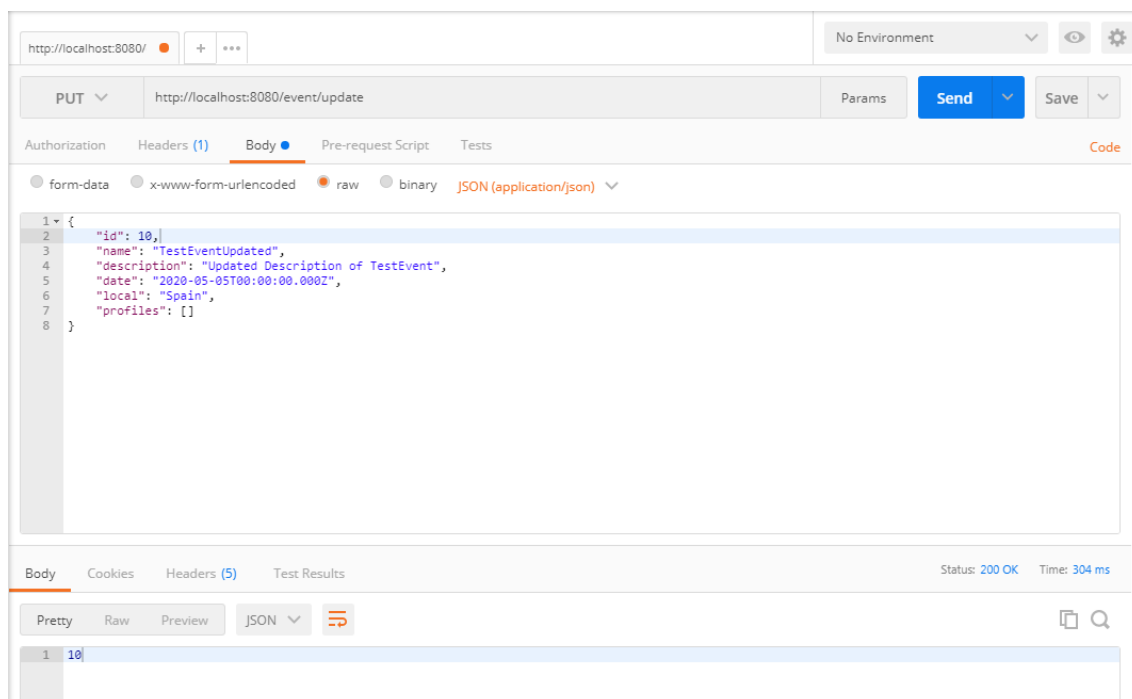


Figura 5.7: Resultado observável no *Postman*.

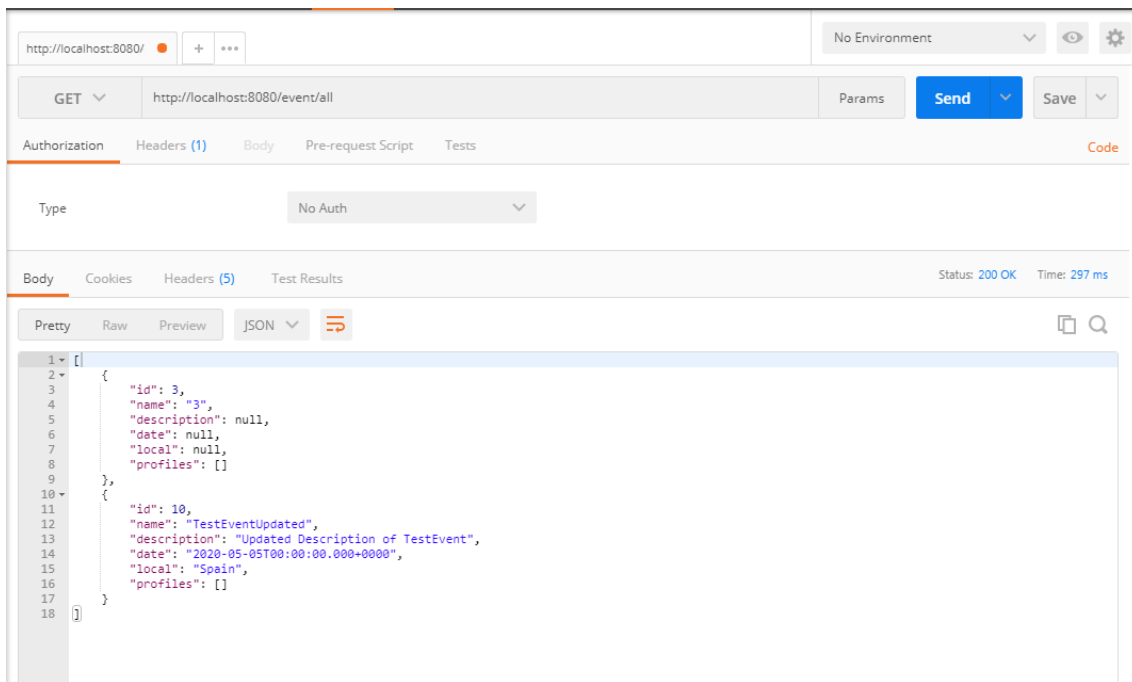


Figura 5.8: Resultado de um *GET ALL* depois do *PUT* observável no *Postman*.

Teste : *UPDATE NONEXISTENT OBJECT*

Endpoint : `http://localhost:8080/event/update`

Body :

```
{
  "id": 9
  "name": "TestEventUpdated",
  "description": "Updated Description Of TestEvent",
  "date": "2020-05-05T00:00:00.000Z",
  "local": "Spain",
  "profiles": []
}
```

Resultado : Erro.

Status : *404 NOT FOUND*

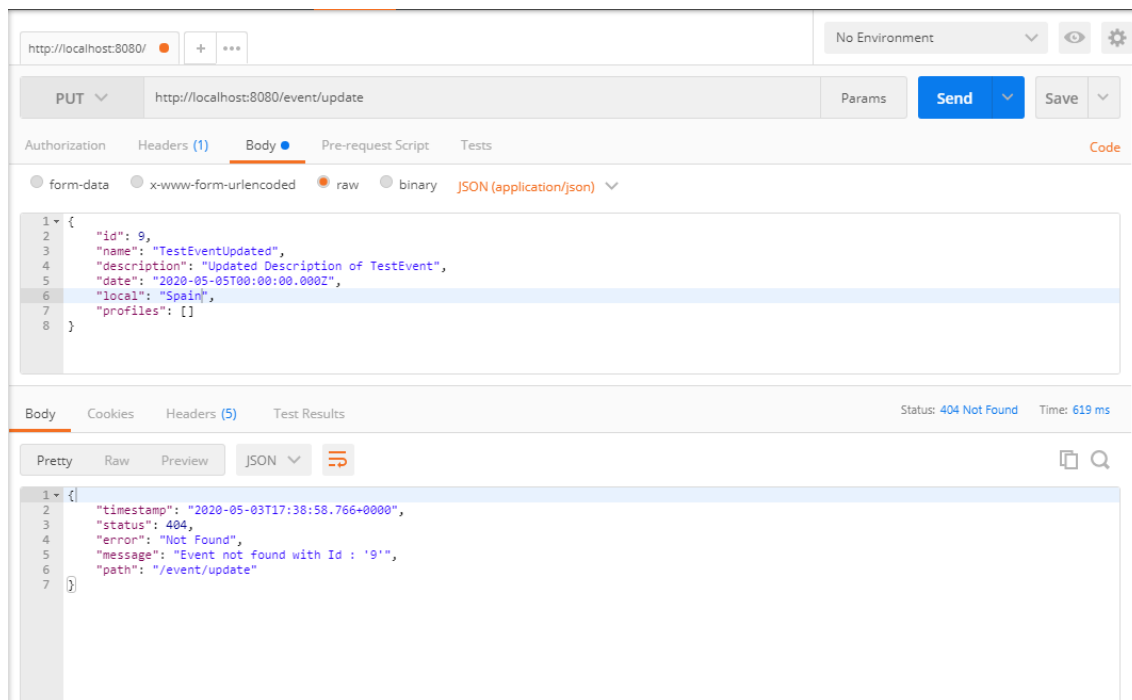


Figura 5.9: Resultado observável no *Postman*.

#### 5.1.4 Testes de DELETE

Os testes feitos ao *endpoint* de *DELETE* são executados com um objecto *JSON* enviado no corpo do pedido com o id do objeto a ser apagado.

Teste : *DELETE EXISTENT OBJECT*

Endpoint : `http://localhost:8080/event/delete`

Body : 

```
{
  "id": 10
}
```

Resultado : Nenhum

Status : *200 OK*

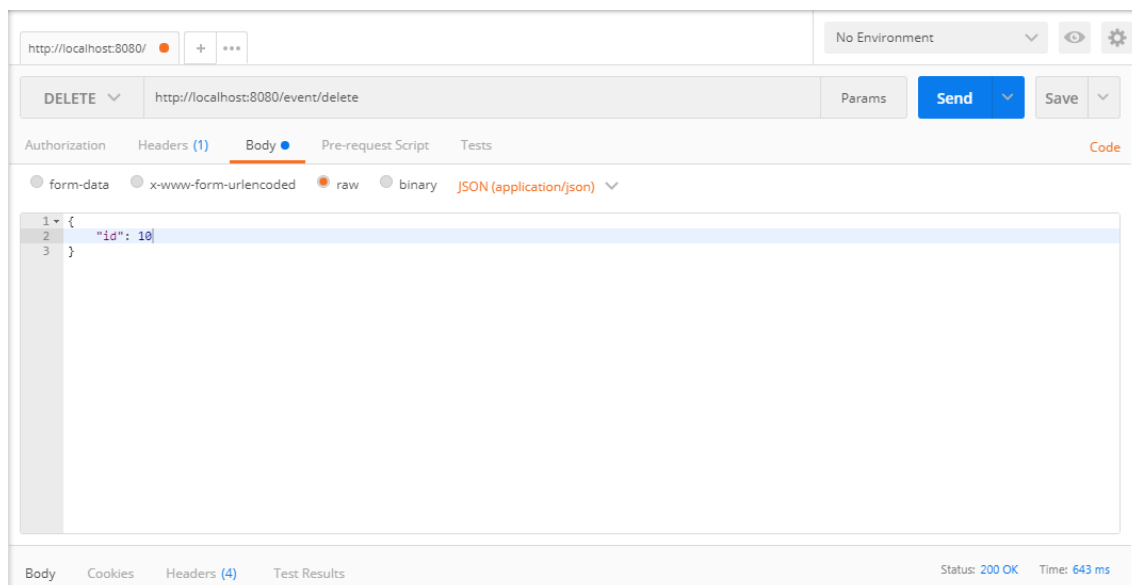


Figura 5.10: Resultado observável no *Postman*.

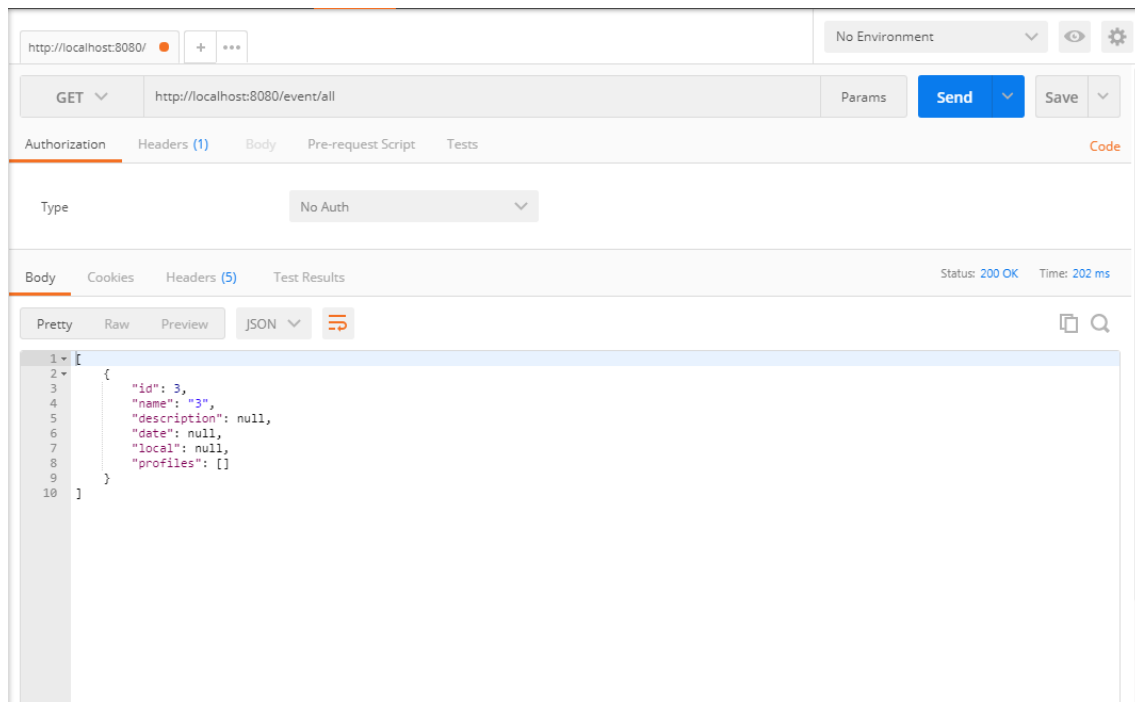


Figura 5.11: Resultado de um *GET ALL* depois do *DELETE* observável no *Postman*.

Teste : *DELETE NONEXISTENT OBJECT*

Endpoint : `http://localhost:8080/event/delete`

Body :  

```
{  
  "id": 9  
}
```

Resultado : Erro

Status : *404 NOT FOUND*

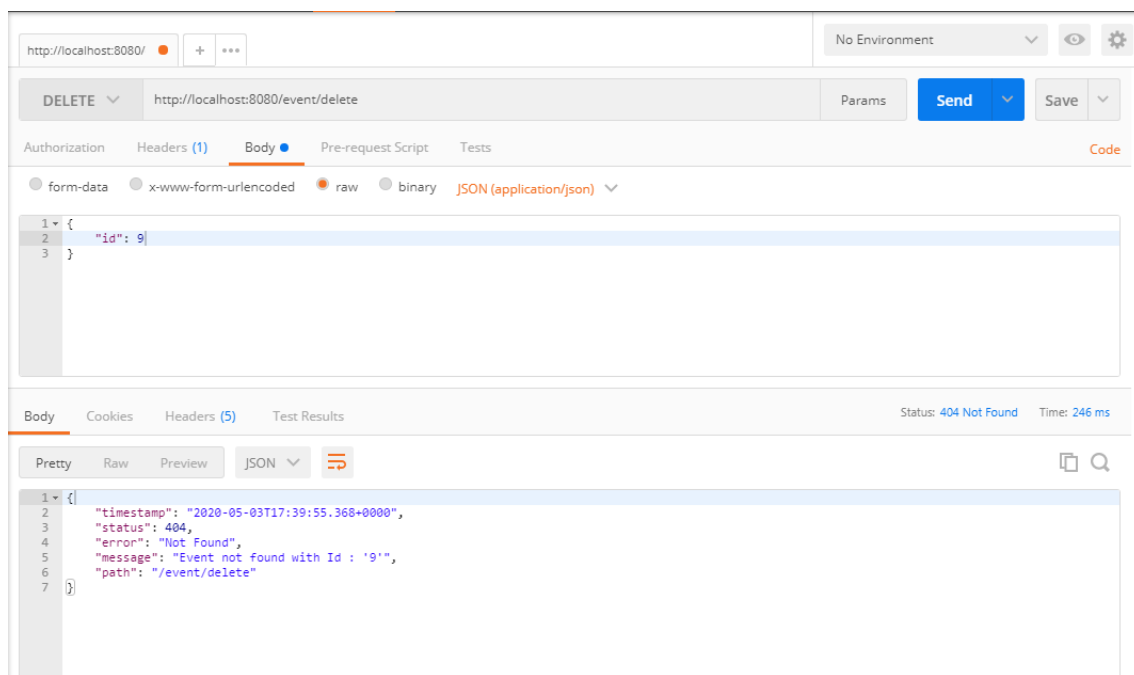


Figura 5.12: Resultado observável no *Postman*.

## 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] Angular Material. Available at: <https://material.angular.io/> [Accessed 4 June 2020].



## Apêndice A

# Apêndice A

O Apêndice A contém o diagrama de todas as entidades e as suas propriedades.

A lista de entidades representa as propriedades que são tipos primitivos pelo seu nome(*id,height,weight*), as propriedades que referem associações de um para um pelo nome da entidade a que está associada(*Profile*), e as propriedades que referem associações de um para muitos por uma lista de entidades a que está associada(*List<Event>*).

### Lista de Entidades

1. *Athlete* (*id,height,weight,athleteNumber,comment,Profile,List<Practice>,List<TrainingSchedule>,List<Game>,List<AthleteGameStats>*)
2. *AthleteGameStats* (*id,Athlete,Stats,Game*)
3. *Event* (*id,name,description,date,local,List<Profile>*)
4. *Game* (*id,date,local,comment,Opponent,List<Athlete>,List<AthleteGameStats>*)
5. *Opponent* (*id,name,photo*)
6. *Practice* (*id,date,local,comment,List<Athlete>*)
7. *Profile* (*id,name,birth,address,mail,phone,photo,List<Event>*)
8. *Staff* (*id,staffNumber,Profile,StaffType*)
9. *StaffType* (*id,name*)
10. *Stats* (*id,errors,fouls,turnOvers,yellowCards,redCards,tries,mauls,playingTime,Tackle,Mellee,ConversionKick,GoalKick,DropKick,OffSide,LineOut,List<AthleteGameStats>*)
11. *Tackle* (*tackleHits,tackleMiss*)
12. *Mellee* (*melleeHits,melleeMiss*)
13. *ConversionKick* (*conversionKickHits,conversionKickMiss*)

14. *GoalKick* (goalkickHits,goalkickMiss)
15. *DropKick* (dropKickHits,dropkickMiss)
16. *OffSideKick* (offsideHits,offsideMiss)
17. *LineOut* (lineOutHits,lineOutMiss)
18. *Tournament* (id,classification,comment)
19. *TrainingSchedule* (id,description,link,date,List<Athlete>)

Todas estas entidades foram implementadas diretamente na camada do Modelo.

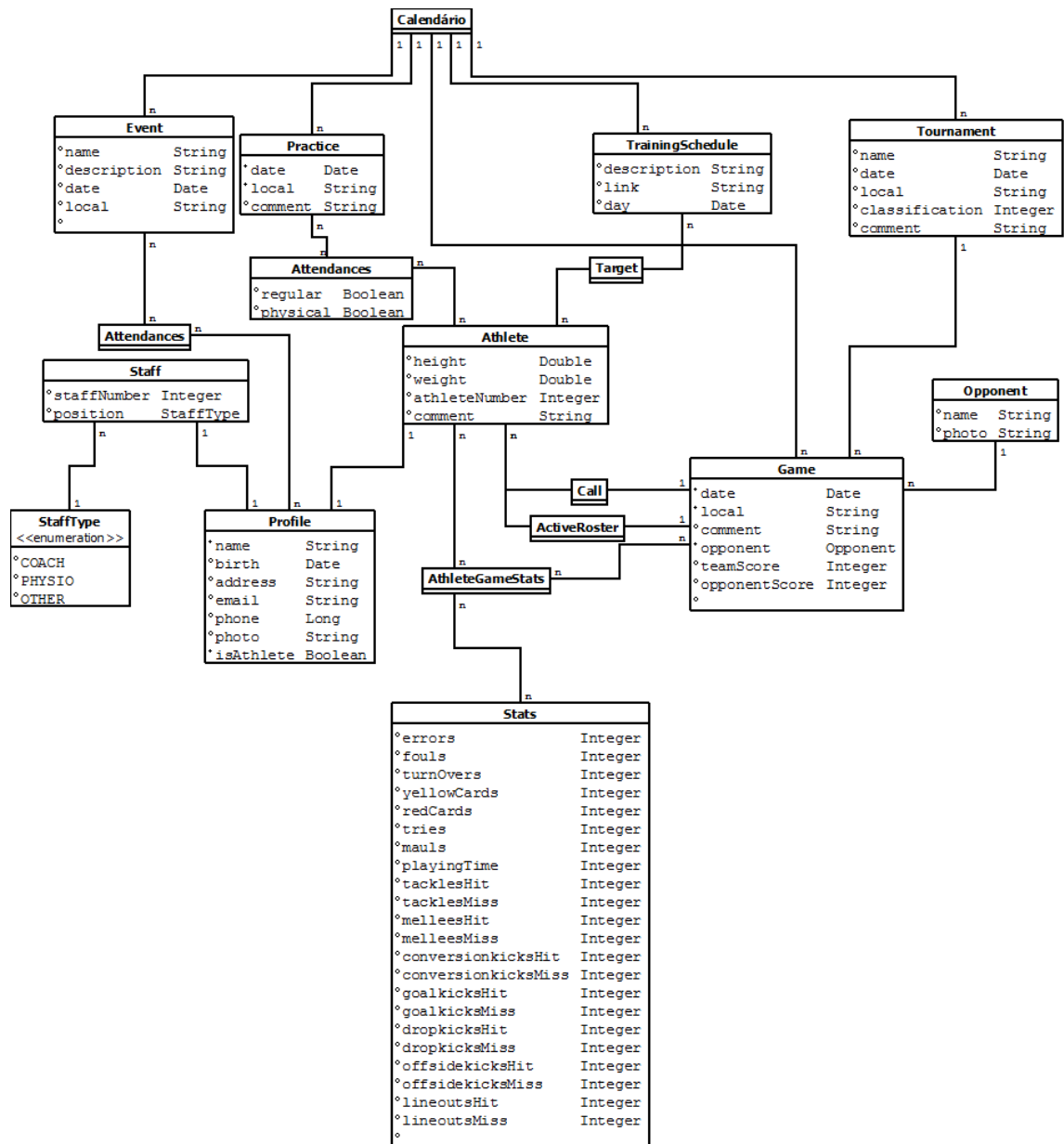


Figura A.1: Diagrama de Entidades