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 sitio, 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

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 pratica 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

aspetos chaves do funcionamento de uma equipa desportiva, auxiliando desde a face tática do desporto (aspetos como a constituição da equipa, o plano tático, a otimização de temáticas de treino), assim como a face menos técnica de uma equipa (aspetos como a organização de treinos e jogos, a facilidade de acesso a informação pertinente, entre outros).

Pretende-se aglomerar todos estes aspetos numa única aplicação, que ofereça aos utilizadores, sendo eles atletas ou equipa técnica, uma plataforma onde possam observar os dados 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 **Formulação do Problema**
Formulação e Contextualização do Problema, Especificações Funcionais e Arquitetura da Solução.

Capítulo 3 **Aplicação Servidora**
Aspetos relacionados com a aplicação servidora, como abordagens, metodologias e detalhes de implementação.

Capítulo 4 **Aplicação Cliente**
Aspetos relacionados com a aplicação cliente, como abordagens, metodologias e detalhes de implementação.

Capítulo 5 **Testes**
Testes executados sobre as diversas vertentes do projeto.

Capítulo 6 **Conclusões**
Recapitulação das observações e conclusões importantes.

Capítulo 2

Formulação do Problema

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

2.1 Formulação

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

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

A hipótese de resposta a esta pergunta foi adquirida da noção pessoal dos estudantes, como indivíduos com ligações interpessoais com o desporto, e das ideias que resultaram do diálogo com as duas equipas referidas neste documento. Após diversas reuniões com foco na recolha de ideias, foi atingida a hipótese de resposta 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 espeficiações 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 auxilio gráfico.

2.3 Arquitetura da Solução

Esta secção explicita a arquitetura da nossa solução.

A nossa aplicação irá ser dividida 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 auxilio 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 de *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 estes 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 sim, 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-secção como camada do Modelo, representa o modelo de dados. É aqui que encontramos todas as Entidades que estruturam o modelo de dados, assim como as relações entre elas.

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.

```
@Entity
@Table (name = "event")
@Data
 public class Event {

   @Id
   @GeneratedValue (strategy = GenerationType.AUTO)
   private Long id;

   @Column
   private String name;

   @Column
   private String description;

   @Column
   private Date date;

   @Column
   private String local;

   @Column
   @OneToMany (mappedBy = "events")
   private List<Profile> profiles;
}
```

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-secção seguinte é explicado como é que se interage com estas entidades.

3.1.2 *Repository*

A camada *Repository*, referida nesta sub-secção como camada de Repositório, representa os repositórios para cada entidade. Outra das características chave do JPA é a habilidade de criar interfaces de repositórios associadas às entidades do modelo, permitindo gerar persistência na base de dados. Quando a aplicação interage com os repositórios (através da chamada a métodos de *querries*), 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.

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

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 *querries* (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.

```java
@Component
public class EventBusiness {
  @Autowired
  EventRepository eventRepository;

  public Iterable<Event> findAllEvents(){
     return eventRepository.findAll();
  }

  public Long postEvent(Event event){
   return eventRepository.save(event).getId();
  }

  public Event findEventById(Long id){
   return eventRepository.findById(id)
   .orElseThrow(()-> new ResourceNotFoundException("Event", "Id", id));
  }

  public Long updateEvent(Event event)
   eventRepository.findById(event.getId())
   .orElseThrow(()-> new ResourceNotFoundException("Event", "Id", event.
   getId()));
   return eventRepository.save(event).getId();
  }

  public void deleteEvent(Event event){
   eventRepository.findById(event.getId())
   .orElseThrow(() -> new ResourceNotFoundException("Event", "Id", event.
   getId()));
   eventRepository.delete(event);
  }
}
```

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*.

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

```
@RestController ()
@RequestMapping ("/event")
public class EventController {
    private static final Logger logger = LoggerFactory.getLogger(
     RugbyApplication.class);

    @Autowired
    EventBusiness eventBusiness;

    @RequestMapping ("event/all")
    public Iterable<Event> findAllEvents(){
     logger.info("On method GET event/all");
     return eventBusiness.findAllEvents();
    }

    @GetMapping ("/findById/{id}")
    public Event findEventById(@PathVariable Long id){
     logger.info("On method GET event/findById/{id} with id: "+ id);
     return eventBusiness.findEventById(id);
    }

    @PostMapping ("/post")
    public Long postEvent(@RequestBody Event event){
     logger.info("On method POST event/post");
     return eventBusiness.postEvent(event);
    }

    @PutMapping ("/update")
    public Long putEvent(@RequestBody Event event){
     logger.info("On method PUT event/update");
     return eventBusiness.updateEvent(event);
    }

    @DeleteMapping ("/delete")
    public ResponseEntity<?> deleteStats(@RequestBody Event event){
     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 deste 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 *Pop-over* que existem na *Ionic Framework* (explicadas em detalhe na sub-secção 4.1.4), foi tomada como regra de decisão gerar estas estruturas num *Component* separado do *Component* 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 que 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 sitio.

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 expecíficas 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/httpservices*) 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 dispõe tambem de um serviço próprio(agrupados no *package app/componentservices*) que contem todos os métodos que envolvem a algoritmia de regras de negócio adicionais dessas entidades. Podemos observar no troço de código seguinte, como exemplo, o serviço *AthleteGameStatsService*, que contem 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

Interpolation Corresponde a ligar uma expressão que é calculada quando o *template* é gerado a um elemento *HTML* dentro desse *template*.
A sintaxe deste *binding* é {{*expression*}}.

Property Corresponde a ligar uma propriedade do *Component* a um elemento *HTML* do *template*. É uma ligação denominada *Source-to-View*, que altera dinâmicamente o valor do elemento para o valor da propriedade, mesmo quando o valor da propriedade é alterado em *run-time*.
A sintaxe deste *binding* é [*target*]="*expression*".

Event Corresponde a ligar um evento de um elemento *HTML* a uma declaração do *Component*. É uma ligação denominada *View-to-Source*, que conecta um evento na vista (*click* de um botão, *check*/*uncheck* de uma *checkbox*) a uma declaração (podendo ser um método do *Component*).
A sintaxe deste *binding* é (*target*)="*statement*".

Two-Way 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 binding é [(*target*)]="*expression*".

O *Angular* também contem 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.atribute-name*]=" *expressio*

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*, alternaveis 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 expecífico 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,
substitundo 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 apropridados, 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 automáticamente 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 *AthleteGameStats*).

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 algumas, segmentos com abas para os diversos tipos de informação a mostrar (calendário com *Upcomming*, *All* e *Past*, estatísticas com *Stats* ou *Graph*).

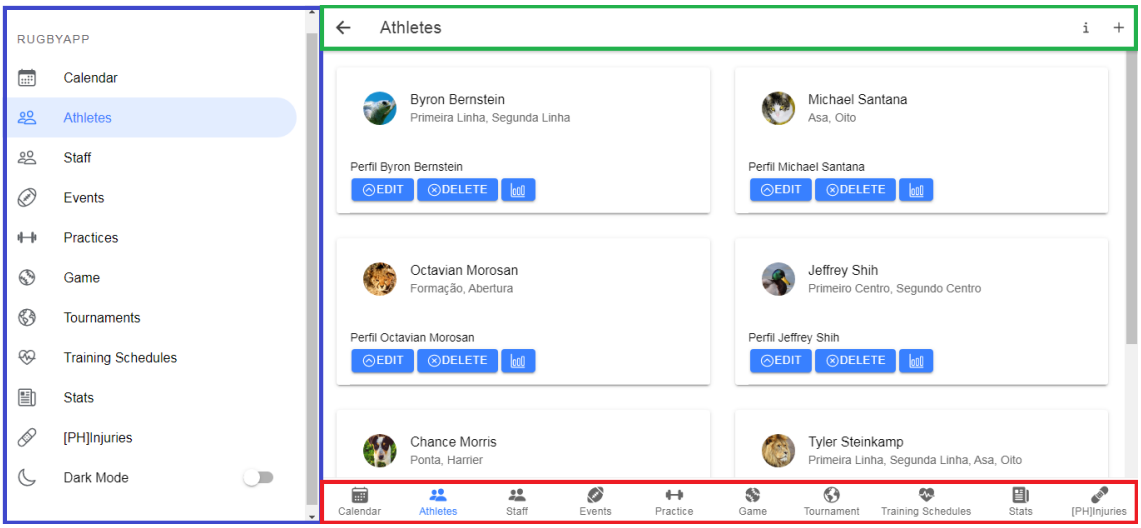

Figura 4.1: Figura da vista inicial da aplicação. Observa-se o *ion-split-pane* a azul, o *ion-tabs* a vermelho e o *ion-heasder* a verde.

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 requirir 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.

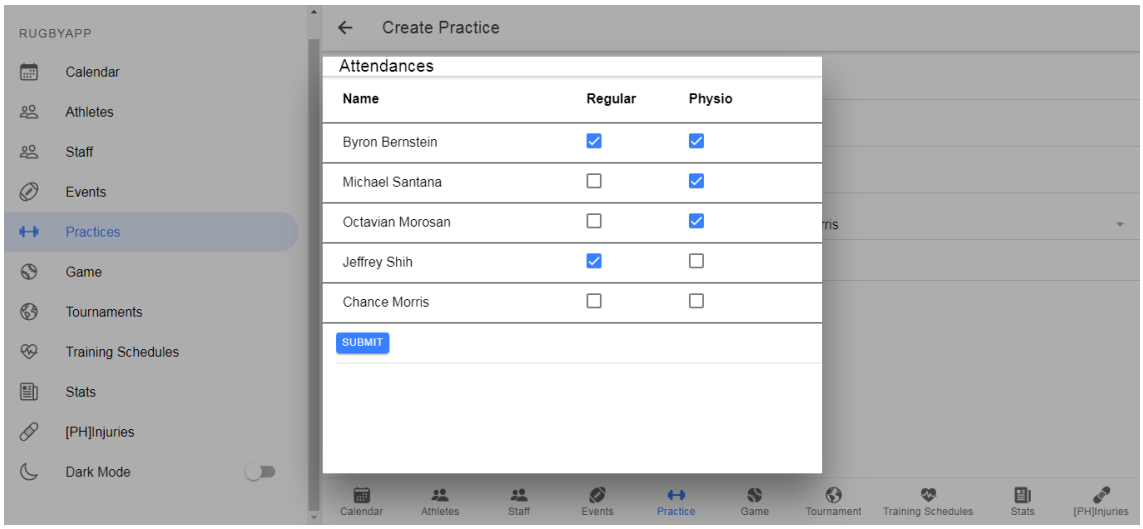

Figura 4.2: Figura do *Modal* do *endpoint* do formulário de *Practice*.

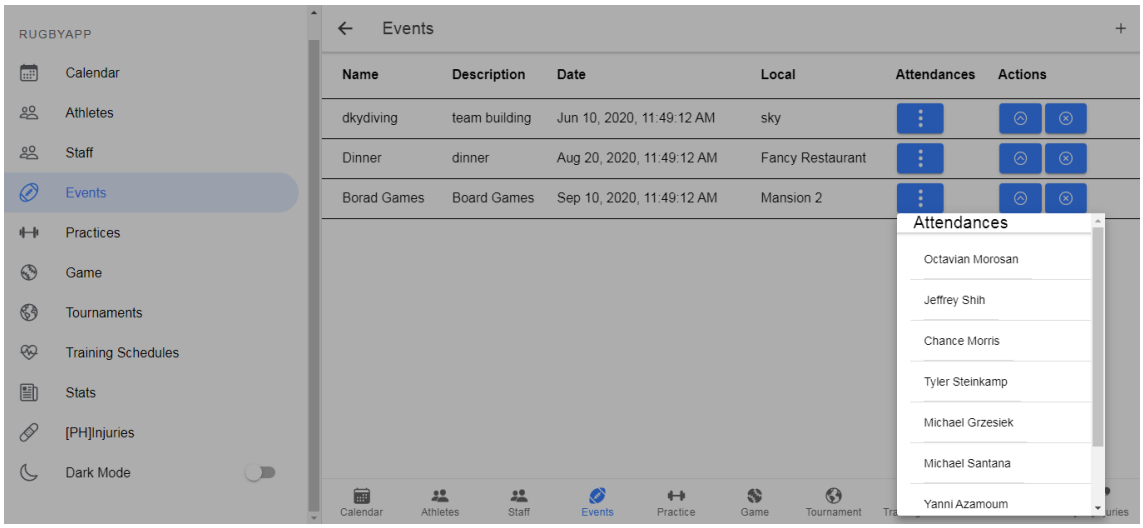

Figura 4.3: Figura do *Popover* do *endpoint Event*.

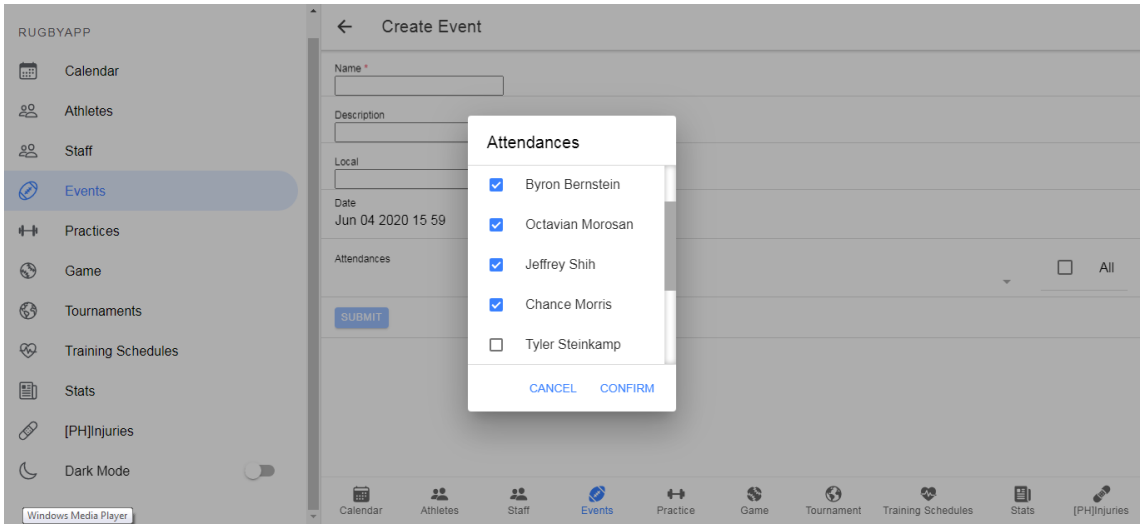

Figura 4.4: Figura do *Select* do *endpoint* do formulário de *Event*.

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*

```
constructor(private eventService: EventService, private popoverController:
    PopoverController, private alertController: AlertController) { }

/* . . .*/

async createPopover(profiles: Profile[], ev) {
  const popover = await this.popoverController.create({
    component: EventPopoverComponent,
    componentProps: { profiles },
    event: ev
  });
  return await popover.present();
}
```

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.

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

```
<ion-header>
  <ion-title>Attendances</ion-title>
</ion-header>

<ion-content>
  <ion-list>
    <ion-item *ngFor="let profile of this.navParams.get('profiles')">
      <ng-template [ngIf]="profile.athlete" [ngIfElse]="staff">
        <ion-item detail="false" (click)="close()" routerLink="/app/athlete/
    athlete-profile/{{profile.id}}">
          <ion-label>
            <h3>{{profile.name}}</h3>
          </ion-label>
        </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 perfíl 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.

4.2 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 contem um

Capítulo 5

Testes

Este capítulo aborda os testes executados no projeto.

5.1 Aplicação Servidor

Nesta fase do projeto, 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:

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

Postman API cliente para gerar pedidos HTTP.

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

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

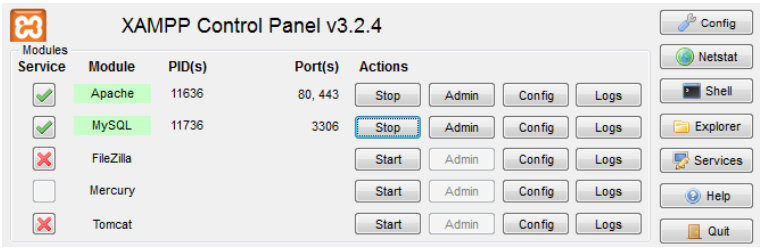

Figura 5.1: Figura do painel de controlo do *XAMMP*.

As seguintes sub-secções 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-secção 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.

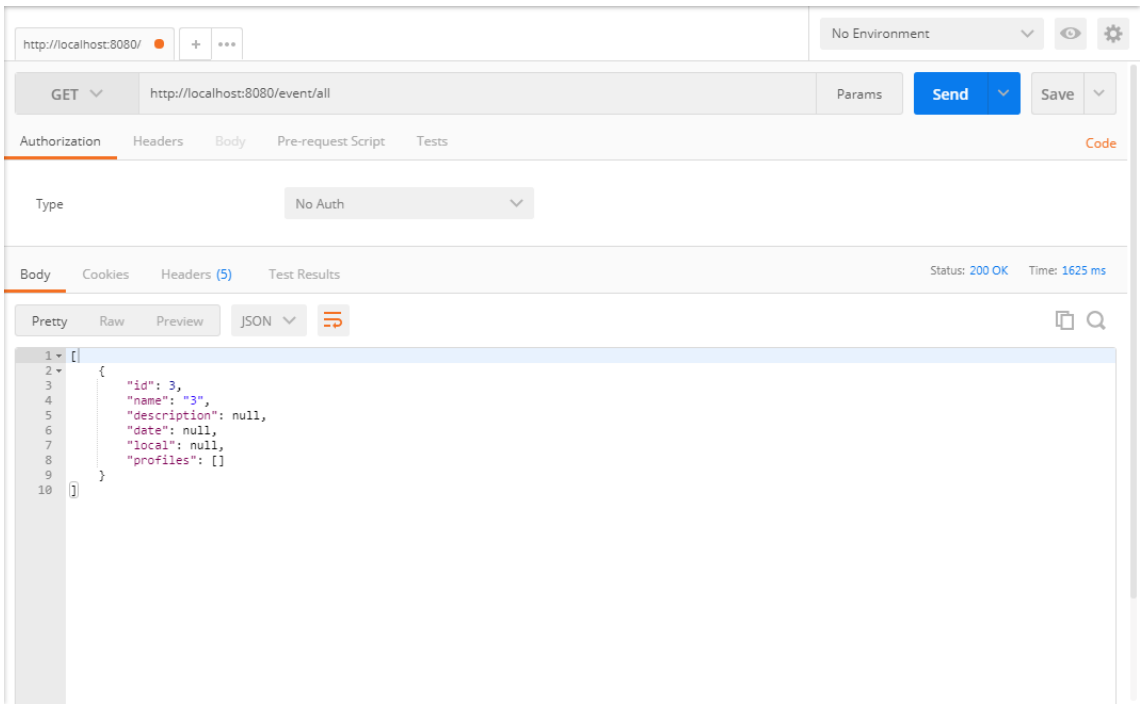

Figura 5.2: Resultado observável no *Postman*.

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.

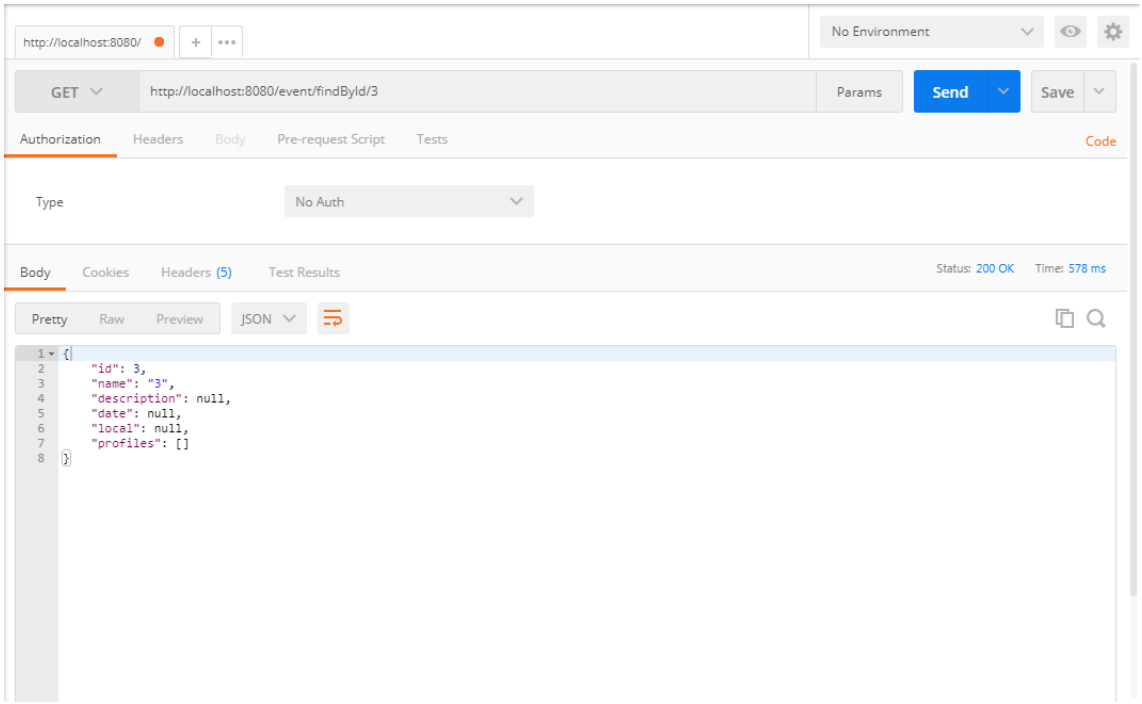

Figura 5.3: Resultado observável no *Postman*.

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.

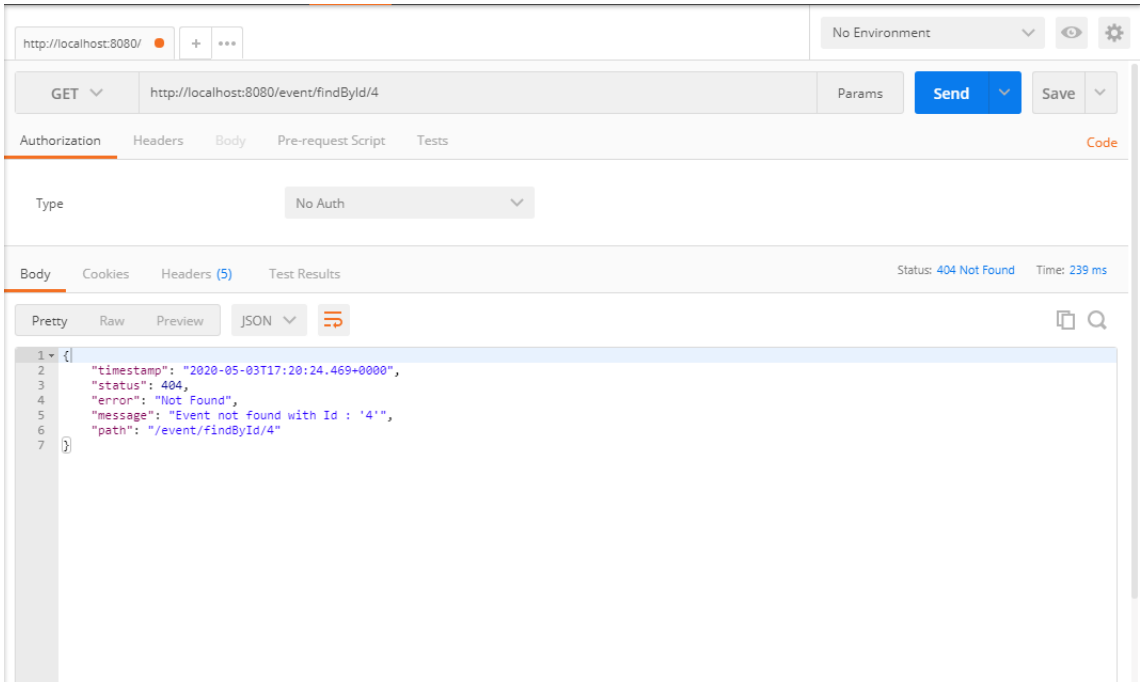

Figura 5.4: Resultado observável no *Postman*.

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*

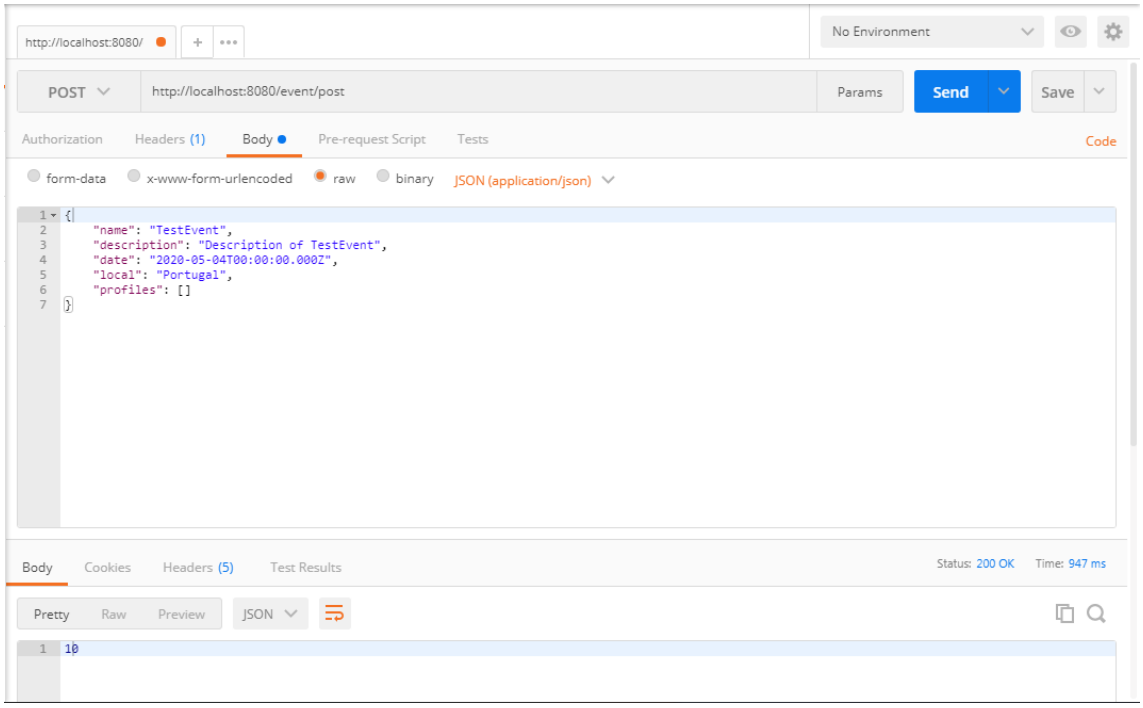

Figura 5.5: Resultado observável no *Postman*.

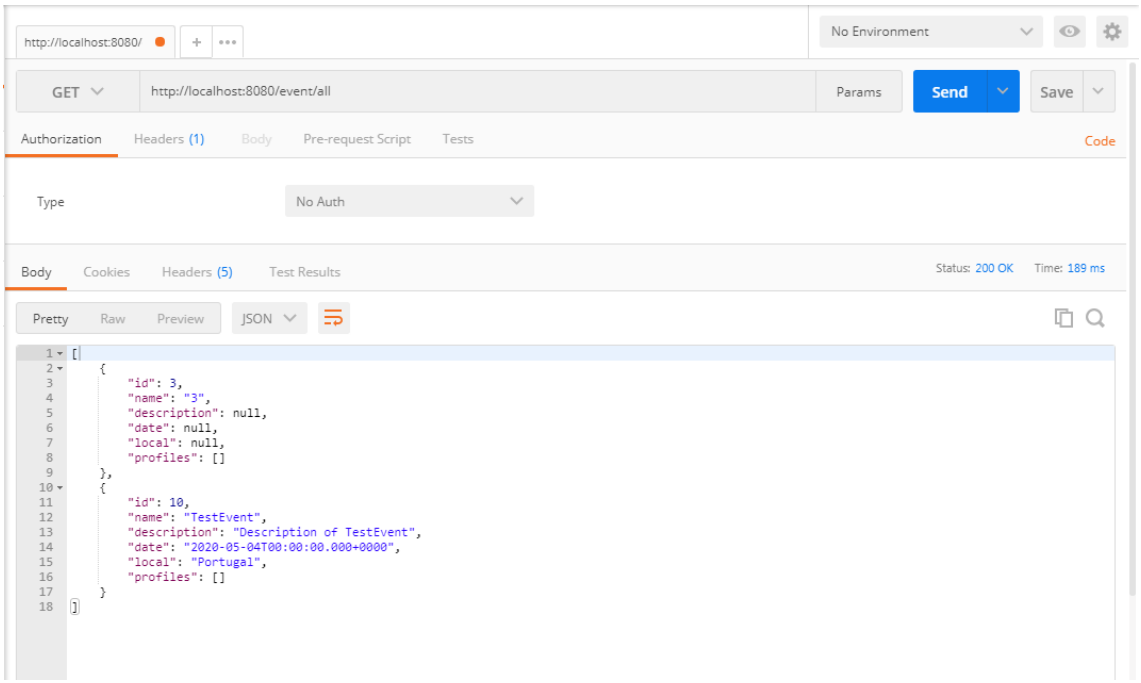

Figura 5.6: Resultado de um *GET ALL* depois do *POST* observável no *Postman*.

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*

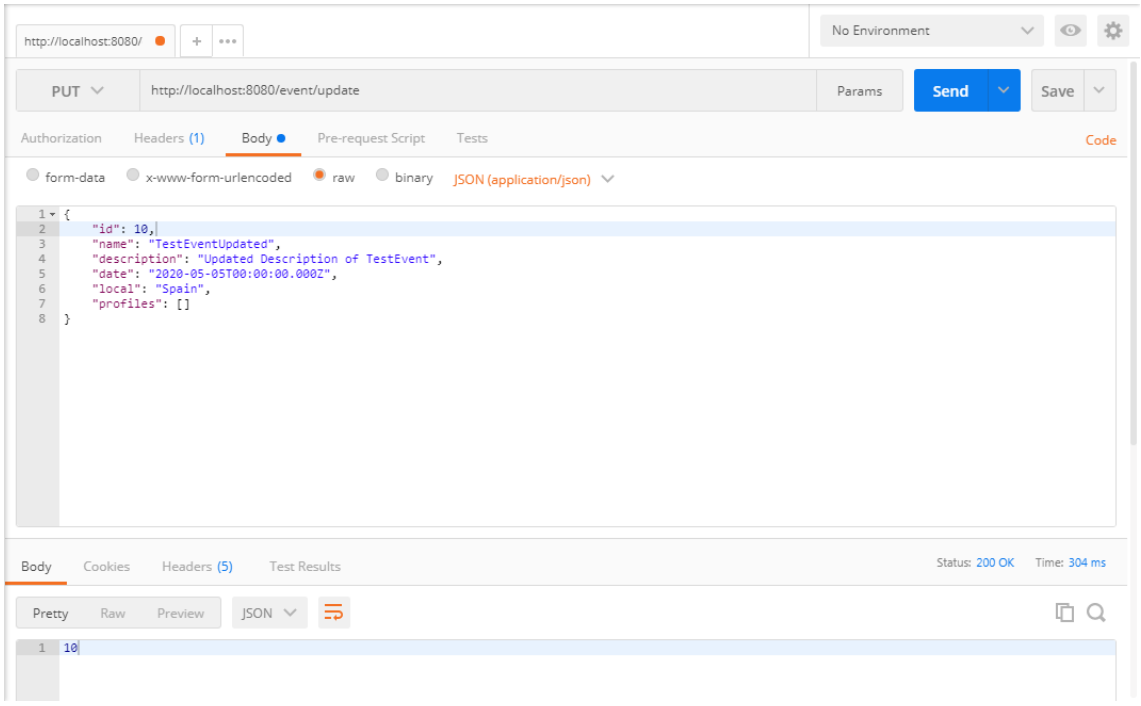

Figura 5.7: Resultado observável no *Postman*.

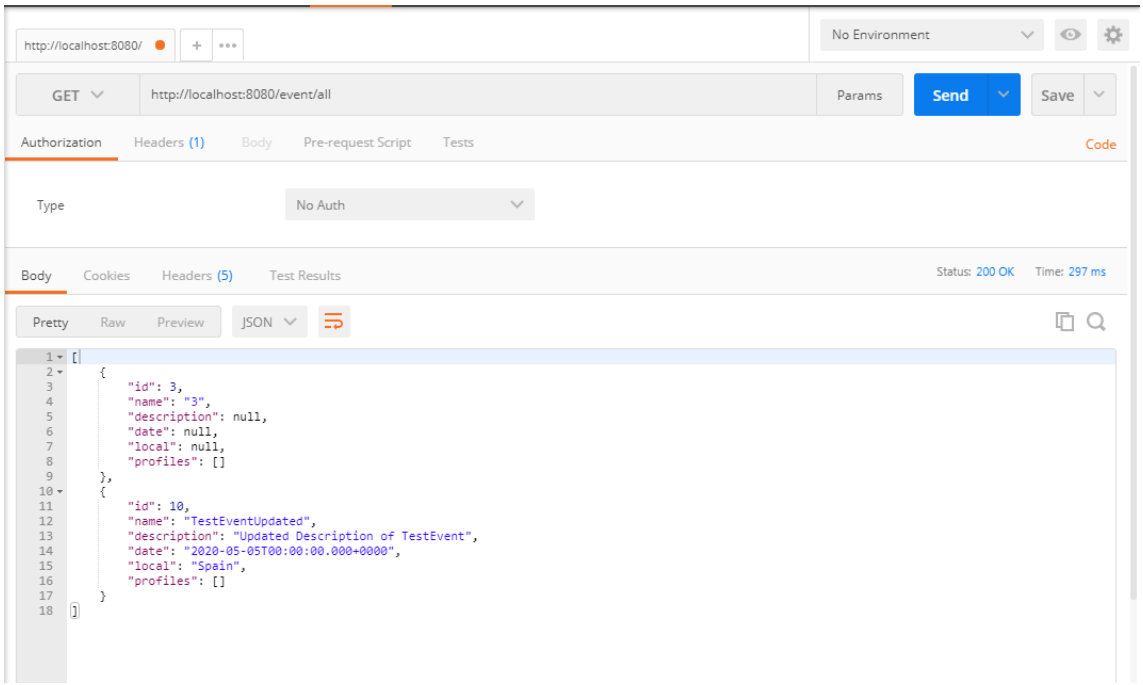

Figura 5.8: Resultado de um *GET ALL* depois do *PUT* observável no *Postman*.

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*

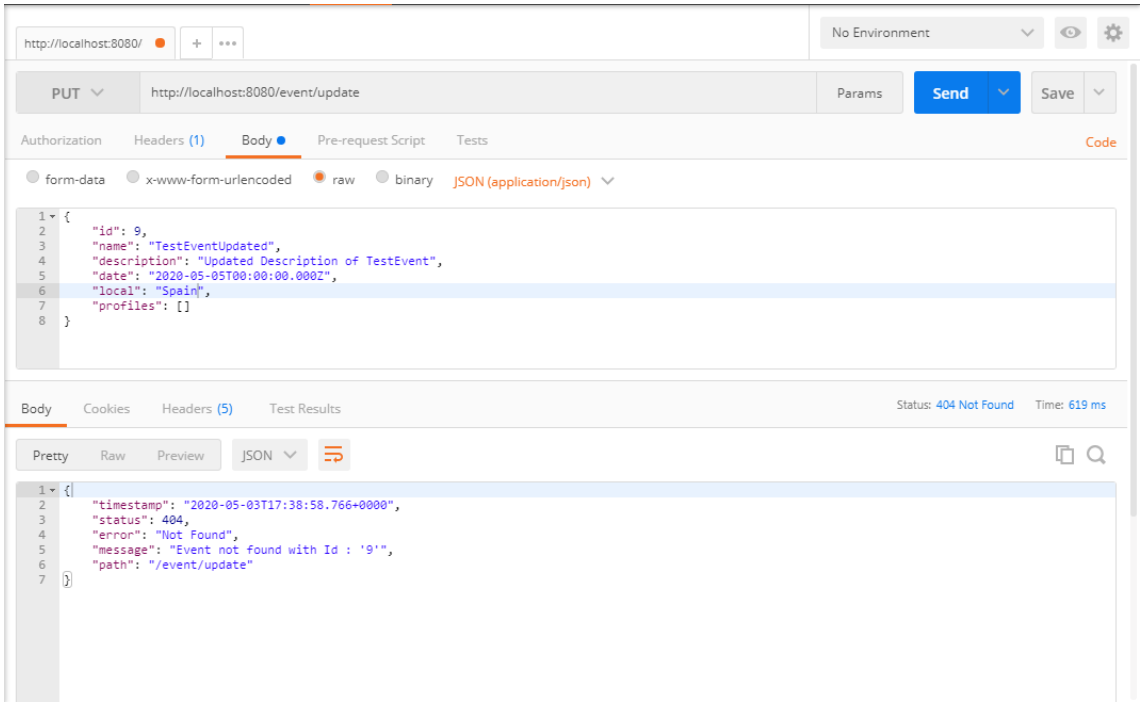

Figura 5.9: Resultado observável no *Postman*.

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*

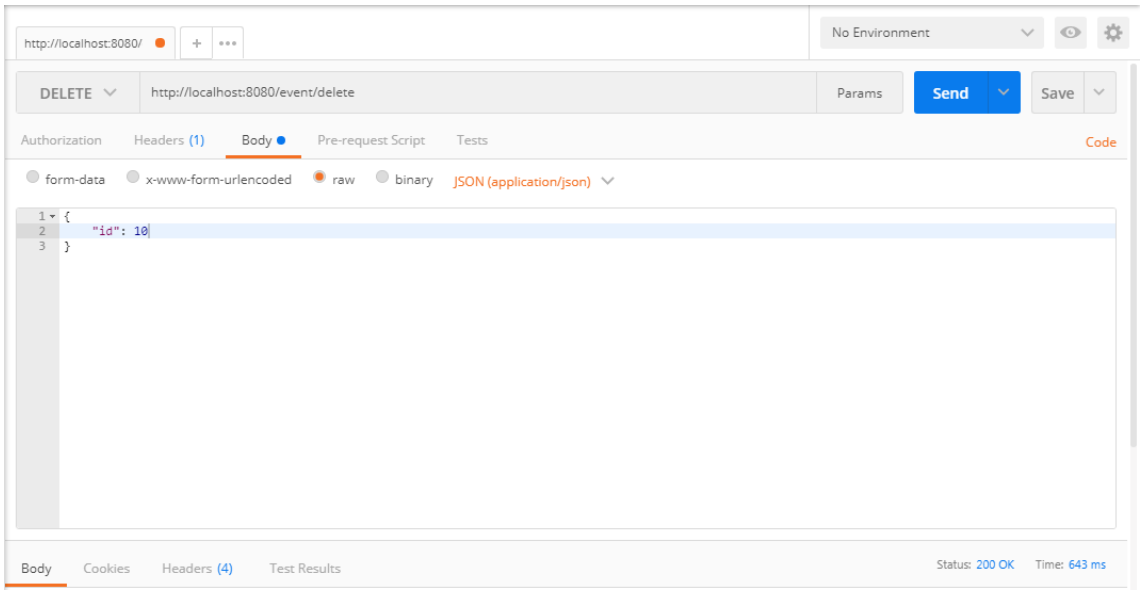

Figura 5.10: Resultado observável no *Postman*.

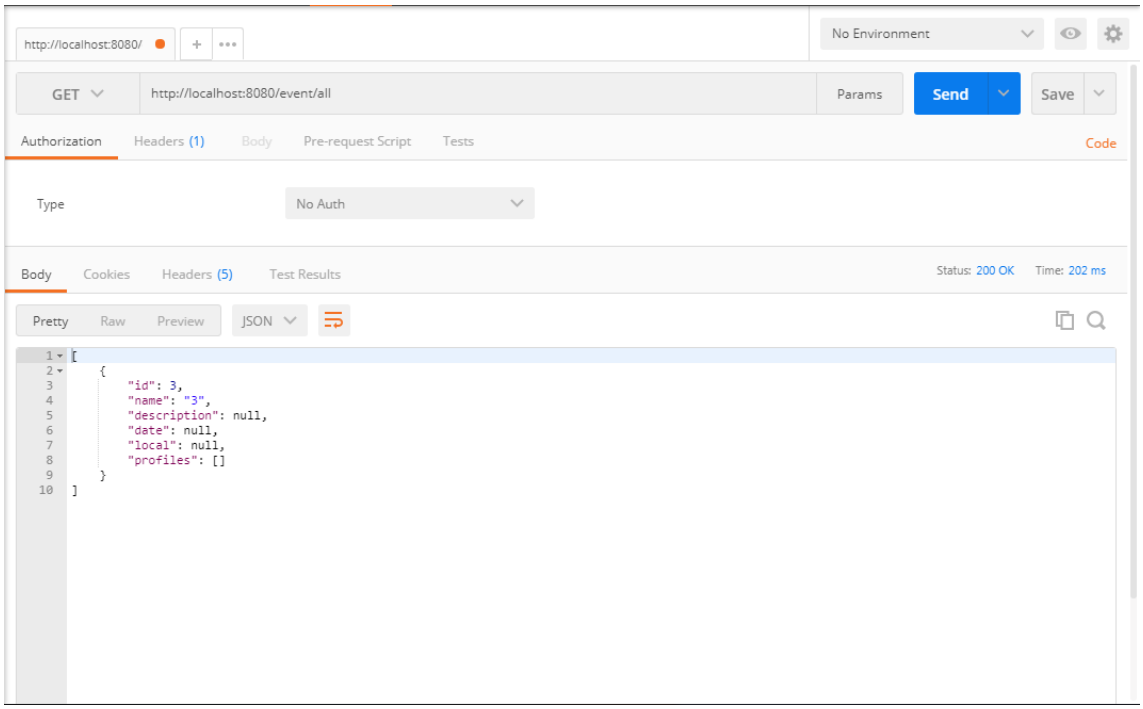

Figura 5.11: Resultado de um *GET ALL* depois do *DELETE* observável no *Postman*.

Teste : *DELETE NONEXISTENT OBJECT*

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

Body : {
”id”: 9
}

Resultado : Erro

Status : *404 NOT FOUND*

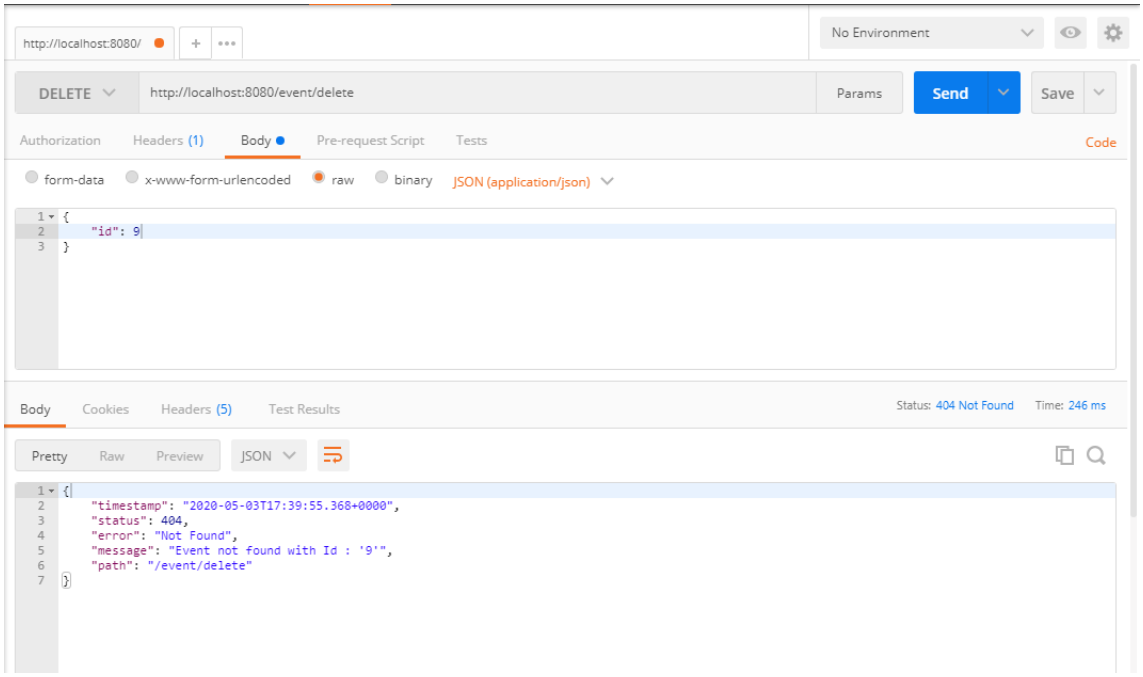

Figura 5.12: Resultado observável no *Postman*.

Capítulo 6

Conclusão

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).

Todo o trabalho feito até agora foi maioritariamente do lado da aplicação servidor. 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 apenas geradas as classes que correspondem às classes de Entidades da aplicação servidor.

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

Referências

[1] Ionic Docs. 2020. API Index - Ionic Documentation. [online] Available at: https://ionicframework.com/docs/api [Accessed 4 May 2020].

Apêndice A

Apêndice A

O Apêndice A contem a lista 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*, *ConvertionKick*,*GoalKick*,*DropKick*,*OffSide*,*LineOut*,List<*AthleteGameStats*>)
11. *Tackle* (tackleHits,tackleMiss)
12. *Mellee* (melleeHits,melleeMiss)
13. *ConvertionKick* (convertionKickHits,convertionKickMiss)

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.