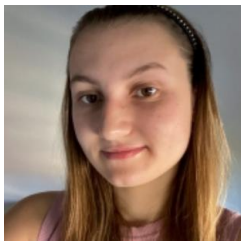
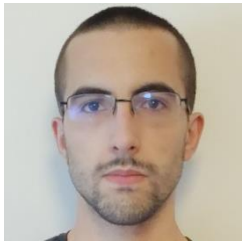
Mestrado em Engenharia Informática

Requisitos e Arquiteturas de Software

Grupo X - PL4

Ana Filipa Rodrigues Pereira PG46978
Bruno Alexandre Dias Novais de Sousa PG45577
Carolina Gil Afonso Santejo PG47102
Raquel Sofia Miranda da Costa PG47600

17 de dezembro de 2021
2021/2022

Índice

Índice de Figuras

1 INTRODUÇÃO E OBJETIVOS

Neste trabalho prático da unidade curricular de Requisitos e Arquiteturas de Software, foi pedido ao grupo que desenvolvesse uma aplicação que permitisse aos seus utilizadores apostar em eventos desportivos. Assim sendo, a primeira etapa do projeto focou-se no domínio do problema tendo sido abordados tópicos como o propósito do sistema, restrições gerais deste projeto ou identificação dos clientes, consumidores e *stakeholders*. Além disto, na fase 1, foram também levantados os requisitos funcionais e não funcionais do sistema.

Por outro lado, esta etapa do trabalho foca-se no domínio da solução. Desta forma, o objetivo desta segunda fase é planear a arquitetura da aplicação de maneira a estar de acordo com o que foi decidido na fase 1 e a ter em conta fatores relevantes como a escalabilidade ou eficiência no acesso aos dados. Para explicar a arquitetura desenvolvida pelo grupo, foram usados diversos diagramas UML que serão detalhadamente explicados ao longo deste relatório.

2 RESTRIÇÕES DA ARQUITETURA

Durante esta fase do trabalho, o grupo precisou de tomar algumas decisões relativas à arquitetura do sistema a desenvolver. Estipulou-se que, na aplicação, seriam utilizadas duas linguagens de programação, usando-se *javascript* para o desenvolvimento do *frontend* e *java* para o desenvolvimento do *backend.* Uma vez que devido ao pouco tempo disponível para criar a aplicação, esta, provavelmente, terá que ser feita em linha de comandos. No entanto, as linguagens selecionadas permitiriam também o desenvolvimento de interfaces gráficas, caso as queiramos implementar no futuro.

3 ÂMBITO E CONTEXTO DO SISTEMA

A aplicação a desenvolver está inserida num sistema bastante fechado onde as únicas entidades externas com quem interage são os utilizadores (*User* e *Bookmaker*). Será um modelo de aplicação web com dois componentes principais: *Frontend* e *Backend*. Em relação ao *Frontend* poderá ser desenvolvido inicialmente em linha de comandos e, posteriormente o objetivo é ser apresentado num browser numa página *HTML*. Quanto ao *Backend*, será constituído por dois elementos fundamentais: lógica de negócio e base de dados. Esta permitirá à aplicação a garantia de persistência dos dados.

Este assunto foi já detalhado no tópico 8 da primeira fase do projeto, onde se pode observar o respetivo diagrama de sistema para uma melhor compreensão.

4 ESTRATÉGIA DA SOLUÇÃO

Os três tópicos que de seguida se apresentam representam as principais linhas de raciocínio seguidas durante o planeamento e concretização da arquitectura do sistema. Desde o inicio da fase de planeamento da arquitectura que

META 1: DIVISÃO DE RESPONSABILIDADES ENTRE COMPONENTES DO SISTEMA

<contexto>

Pretende-se que o sistema tenha um grau de modularidade considerável. Estando garantida esta propriedade, obtém-se:

- A reestruturação da solução passa a ser um processo mais simples e menos dispendioso
- A adaptação do sistema actual para acomodar a implementação de novas funcionalidades é simplificada
- O processo de detecção e correcção de erros/*bugs* é menos dispendioso, permitindo reduzir o *downtime* da equipa de desenvolvimento, que poderá aplicar os seus esforços na optimização do sistema e até na implementação de novas funcionalidades

META 2: AGREGAR FUNCIONALIDADES EM COMPONENTES MVC

Sistemas com este grau de complexidade poder ser geralmente, decompostos em três camadas de funcionalidades principais:

- *View*: o *frontend* do sistema dedicado à interacção com os utilizadores finais.
- *Model: o backend do* sistema, responsável pela orquestração da lógica de negócio (*BL*)
- *Controller*: entidade responsável pela mediação de dados entre o a *View* e o *Model*

O *frontend* e o *backend* têm de comunicar entre si. Além disto, e prevendo que no futuro será desejável distribuir o produto através de tecnologias *web/cloud*, esta comunicação terá de se basear em estratégias e protocolos que permitam a troca de dados entre estes dois componentes que, previsivelmente, serão executados em máquinas diferentes.

Sabendo que estes componentes, de responsabilidades diametralmente opostas, têm de comunicar e partilhar dados coerentemente, parte-se para a implementação, em cada um, de componentes especialmente dedicados à intermediação destas comunicações. Em concreto, adopta-se o padrão de *design* denominado *facade*, onde são implementadas interfaces que tem a única responsabilidade de expor uma *API,* que a cada tipo de pedido/resposta HTTP faz corresponder um desencadear de acções/transformações no sistema que o/a recebeu. Estas API's comunicam com o resto dos componentes do seu sistema recorrendo, novamente, a outras interfaces especializadas e que disponibilizam sub-rotinas que trabalham sobre um dado tipo de objectos.

Depois de se definirem estes serviços, recorre-se ao controlador, que define as opções disponíveis para a concretização de pedidos e envio de respostas. Esta entidade define e expõe os *endpoints aos quais o frontend se pode ligar.* O tráfego nesta ligação *frontend-controller* consistirá apenas em sequências de pares (pedido, reposta) HTTP. De facto, a totalidade das funcionalidades disponíveis no lado do utilizador pode ser caracterizada como a mera submissão de pedidos (*GETs* e *POSTs*) para a obtenção e criação de dados.

A comunicação entre o controlador e o *backend* é, em comparação, muito mais directa. Neste caso, o controlador tem conhecimento dos serviços e funcionalidades disponíveis e desencadeia as sequências de ações necessárias para responder aos pedidos do utilizador.

META 3: SUPORTAR DESENVOLVIMENTO FUTURO DE APLICAÇÕES *CROSS-PLATFORM*

O sistema deve poder crescer de forma a suportar o desenvolvimento e integração ágeis de aplicações web/móveis que possam vir a ser utilizadas pelos *end users*.

Para providenciar este grau de suporte, optou-se pela utilização de tecnologias/frameworks web frequentemente utilizados em contextos semelhantes e que apresentam uma grande variedade de recursos de documentação e aprendizagem.

A *stack* de tecnologias utilizada é:

1. *Frontend* (interação com utilizadores por CLI/GUI): Node.js, React.js
2. *Backend* (*business logic*): Java Spring Boot
3. Base de Dados: PostgreSQL

A escolha dos *frameworks* utilizados assenta, essencialmente, na necessidade de se facilitar o desenvolvimento futuro de aplicações *cross-platform*. Além disto, pretende-se que as implementações destas aplicações sejam consideravelmente homogéneas, de forma a evitar a manutenção de várias *code-bases*. Este ponto é importante quando se prevê que o produto a desenvolver irá abranger várias plataformas. Se a equipa de desenvolvimento conseguir utilizar uma única *code-bas*e para o desenvolvimento de várias aplicações, então, quando for necessário adicionar novas funcionalidades, essas terão de ser implementadas apenas uma vez.

Por estas razões, enveredou-se pela utilização dos *frameworks* Node.js e React.js. São ferramentas com muita utilização e que disponibilizam mecanismos apropriados à criação de *software* que pode mais tarde ser adaptado a várias plataformas terminais (utilizando por exemplo React Native).

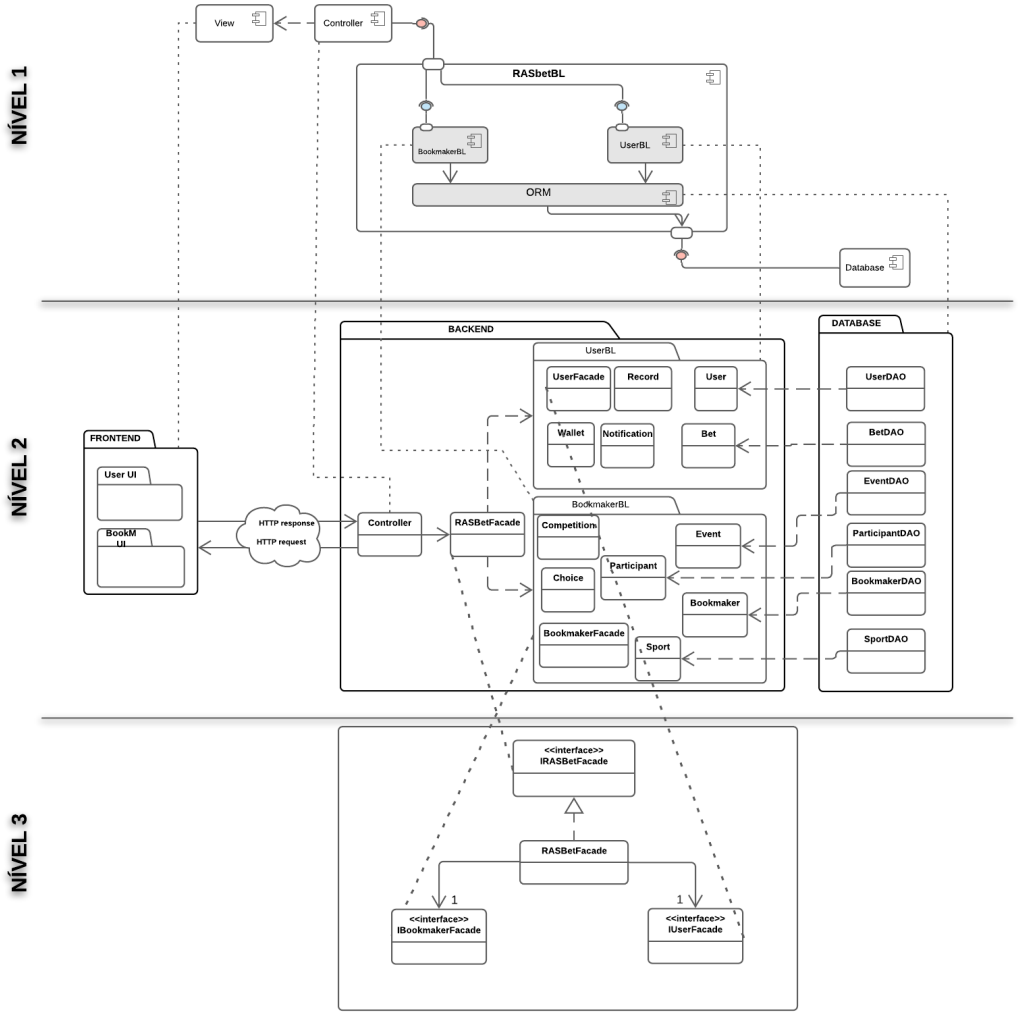
A utilização do Spring Boot facilita o processo de criação de um serviço *RESTful* através do sistema de anotações providenciado pelo *framework*, agilizando o desenvolvimento dos sistemas que são efectivamente mais importantes e relevantes à lógica de negócio.

5 “BUILDING BLOCK VIEW”

A arquitetura do sistema começa o seu desenvolvimento a partir da identificação de possíveis subsistemas. Esta estratégia permite uma maior organização, uma vez que tanto as classes e os seus respetivos métodos serão agrupados de acordo com o subsistema onde melhor se adequam. Assim sendo foram identificados **dois subsistemas**: um relativo ao utilizador e a todas as entidades relacionadas com ele, e o segundo é relativo ao *bookmaker* e aos eventos/jogos.

De seguida, procedeu-se à modelação dos requisitos levantados na etapa anterior deste projeto. Para tal, foi elaborado um **diagrama de componentes** que descreve os componentes do sistema e a dependência entre eles, permitindo assim, identificar em cada nível o que é necessário para construir o sistema. Além disso, foi criado também um **diagrama de packages**, de modo a facilitar a gestão do número crescente de classes, e ainda a identificar o papel de uma classe no sistema tal como as dependências entre as diversas classes presentes. Isto permite à equipa descrever informação importante para a evolução do sistema. E, por último, foram elaborados três **diagramas de classes**, sendo que um deles, é uma generalização dos outros dois de modo a ser possível explicar como é que os dois subsistemas se relacionam. Cada um dos dois restantes diagramas de classes representa um respetivo subsistema.

Assim sendo, podemos concluir que existem 4 níveis de abstração:



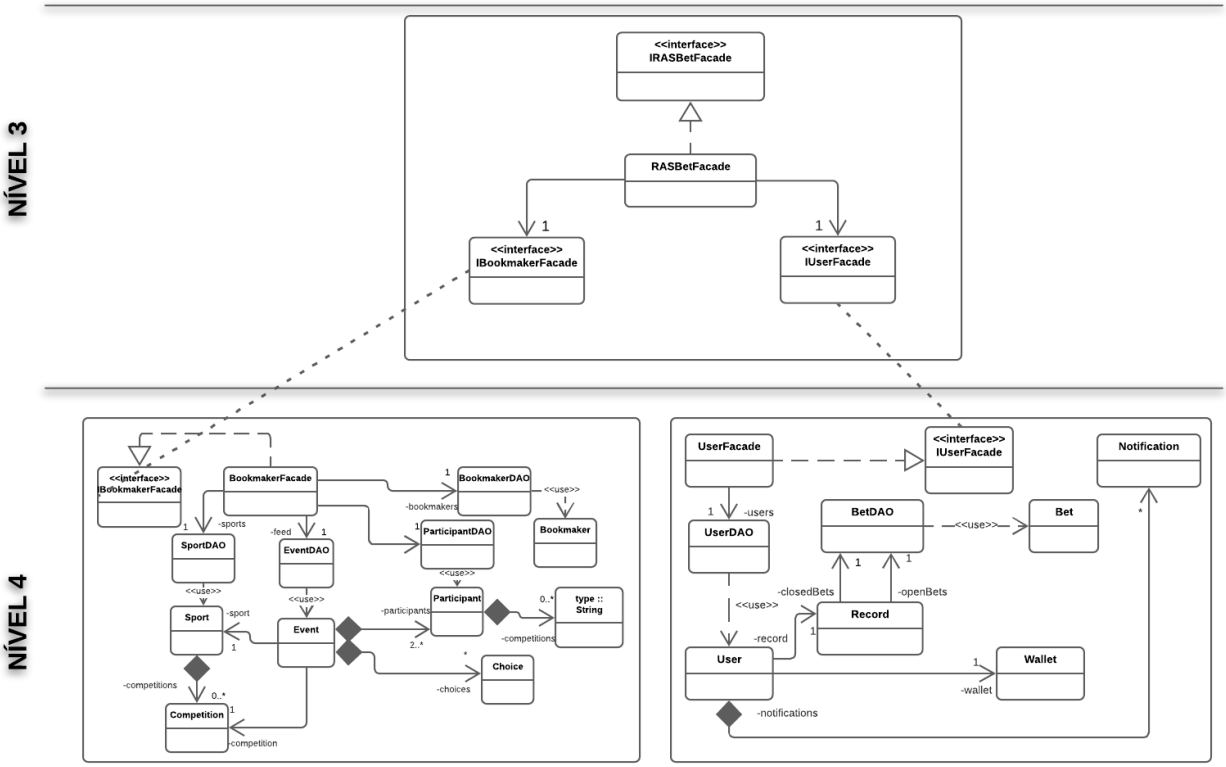


Figura 1 - Níveis de abstração

NÍVEL 1 – DIAGRAMA DE COMPONENTES

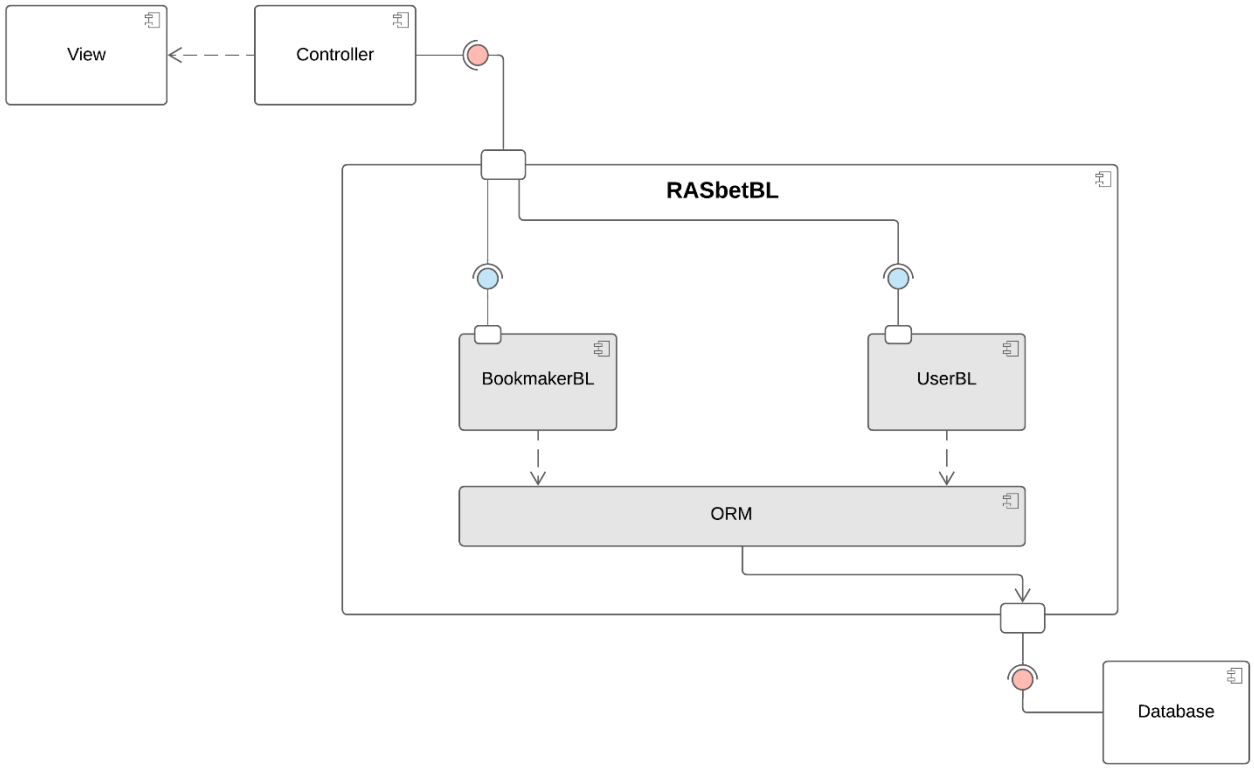


Figura 2 - Vista geral dos componentes integrantes do sistema

Neste diagrama estão representados todos os componentes que integram o sistema, e as relações entre os mesmos.

O componente ***RASbetBL*** trata-se da lógica de negócio da solução, e inclui os dois subsistemas que foram identificados anteriormente: ***BookmakerBL*** e ***UserBL***. O *BookmakerBL* abrange todas as classes relacionadas com o "bookmaker" e os eventos, já o *UserBL* retrata todas aquelas que estão relacionadas com o utilizador.

É apresentado também o componente ***ORM*** que possui todas as classes do tipo *DAO*. Estas classes são responsáveis por obter os dados que estão guardados em memória persistente, neste caso, numa base de dados.

Cada subsistema implementa a sua respetiva interface, e por sua vez o sistema *RASbetBL* irá importar essas mesmas interfaces, além de implementar também a sua própria interface. O componente ***Controller***, por sua vez, irá importar a interface do *RASbetBL*, de modo a conseguir aceder às suas funcionalidades. O componente ***View*** representa o "frontend" da aplicação, o *controller* irá estar em comunicação com o mesmo, uma vez que ele dita aquilo que a "view" deverá apresentar.

NÍVEL 2 – DIAGRAMAS DE PACKAGES

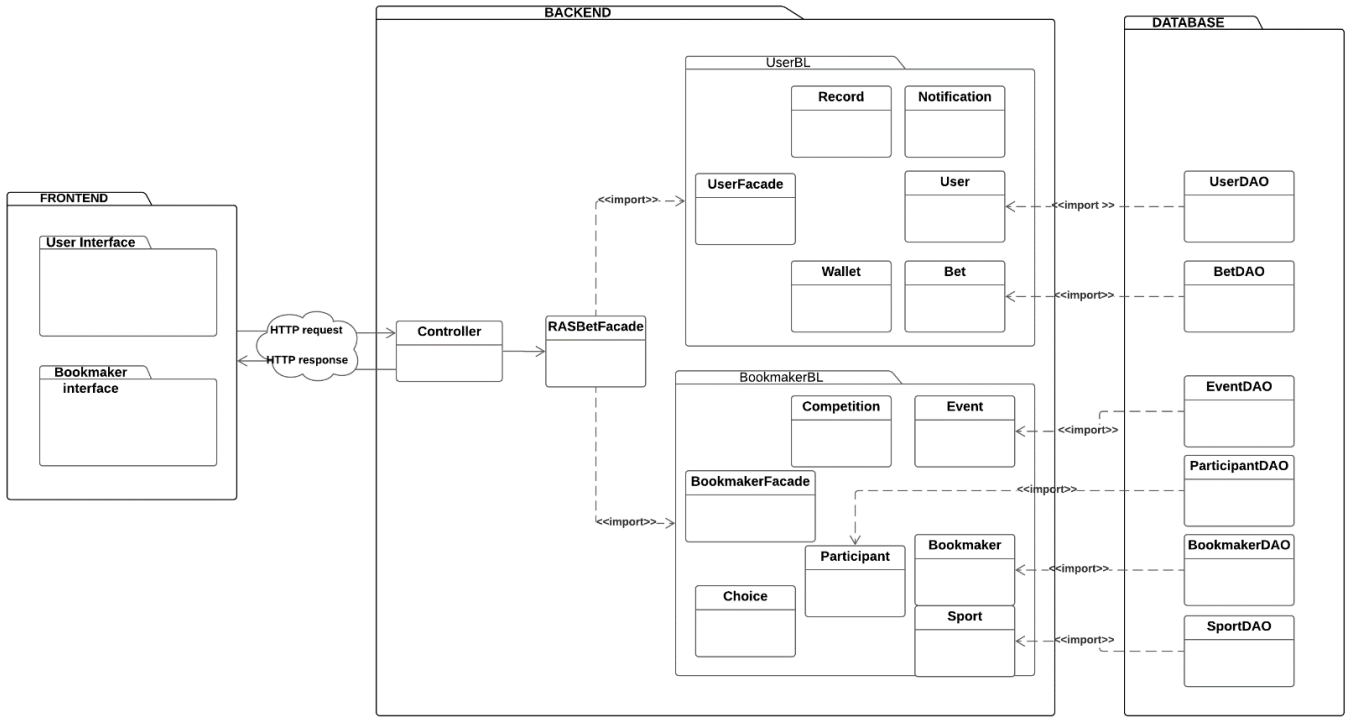


Figura 3 - Diagrama das packaes associadas à modelação da arquitectura

Este diagrama tem como objetivo representar as dependências entre os vários packages.

Começando pelo package ***frontend***, é possível observar que este contém outros dois packages, um deles responsável pela interface do utilizador e outro pela interface do bookmaker. A comunicação entre o *fronted* e o *backend* será feita através de pedidos HTTP REST, tal como está representado na figura. É importante mencionar que esta comunicação com o *backend* da solução é feita através da classe *Controller*. Esta classe por sua vez tem acesso a todos os métodos presentes na *RASBetFacade*, uma vez que importa a interface desta mesma classe.

A ***RASBetFacade***, de forma a não existirem demasiadas dependências entre classes e permitir o encapsulamento dos dados, importa os packages representativos de cada subsistema, isto é, o ***UserBL*** e o ***BookmakerBL***. Dentro deles, estão presentes as classes que cada um deles inclui. Algumas destas classes serão importadas por aquelas que se encontram no package ***Database***. As classes DAO permitem salvaguardar os objetos das classes a que correspondem, mantendo assim, a persistência dos dados.

Concluindo, existem três packages principais:

- ***Frontend:*** É responsável por armazenar todas os objetos que fazem parte da interface gráfica.
- ***Backend:*** É responsável pela lógica de negócio.
- ***Database:*** Contém todas as classes necessárias à preservação de dados.

NÍVEL 3 – DIAGRAMA DE CLASSES GERAL

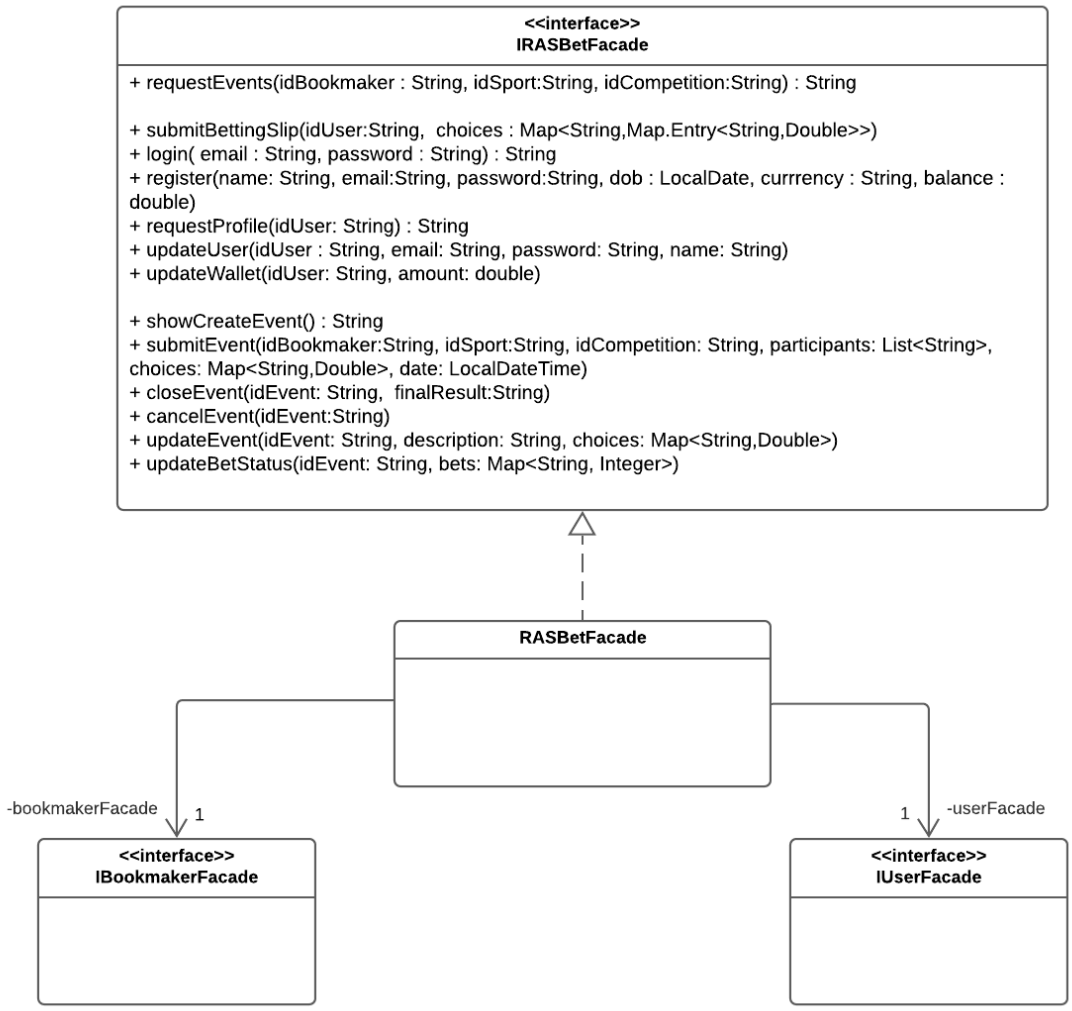


Figura 4 - Diagrama de classes relativo à interação entre os principais subsistemas

Este diagrama de classes trata-se de uma visão geral, isto é, uma visão mais abrangente de como os dois subsistemas estão relacionados, e como são tratadas as dependências entre as classes pertencentes a cada um deles.

Primeiramente, é na interface ***IRASBetFacade*** onde se encontram os métodos principais do sistema, que representam as funcionalidades da aplicação a desenvolver. Estes métodos irão chamar "sub-métodos" que estão definidos nas interfaces de cada subsistema, isto é, ***IBookmakerFacade*** e ***IUserFacade***. Deste modo, diminui-se a dependência entre classes, e dá-se prioridade ao encapsulamento de dados.

Assim sendo, tal como é possível observar através do diagrama, a classe ***RASBetFacade*** irá conter um objeto do ***facade*** do subsistema *BookmakerBL* e outro do ***facade*** do *UserBL*.

NÍVEL 4 – DIAGRAMAS DE CLASSES

- BookmakerBL

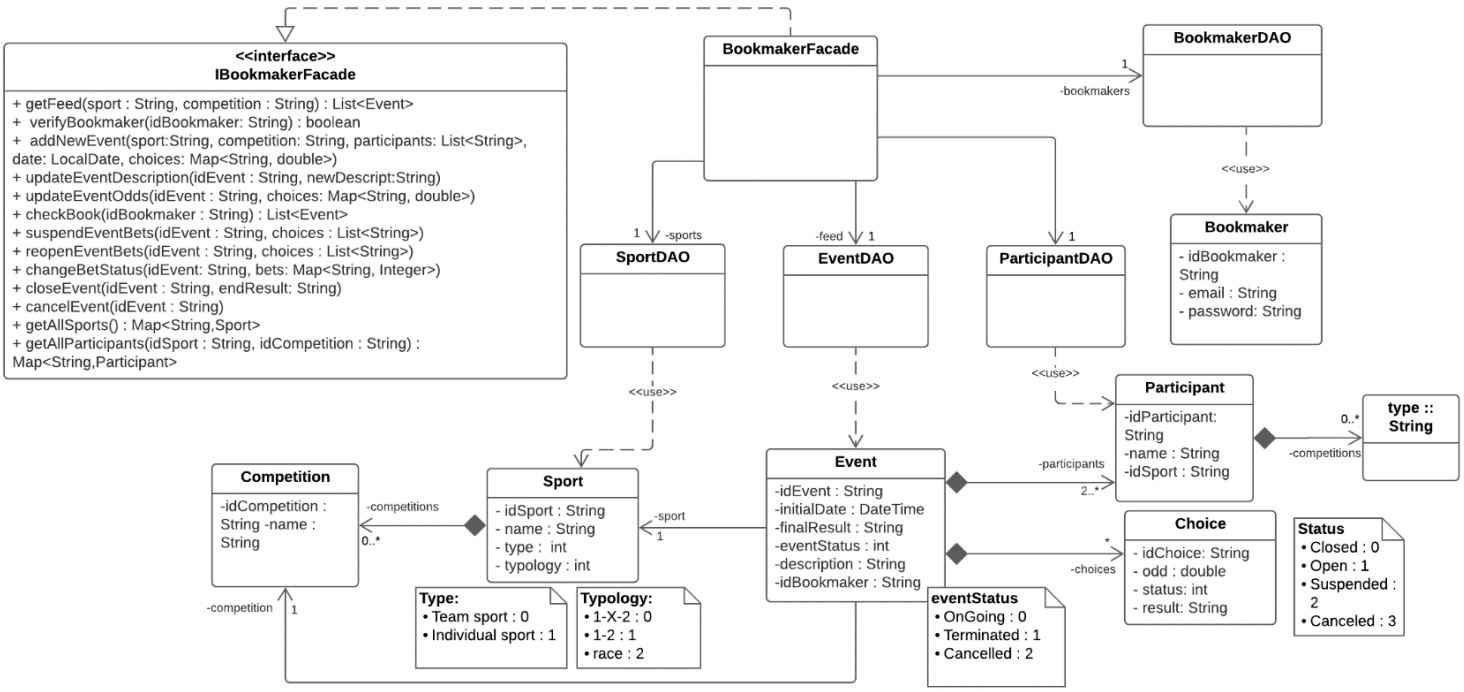


Figura 5 5 - Diagrama de classes da lógica de negócio respeitante ao Bookmaker

Este diagrama de classes representa o subsistema ***BookmakerBL***, é aqui onde estão incluídas as classes e todos os métodos relacionados com o *Bookmaker* e o gerenciamento de eventos.

A ***IBookmakerFacade*** trata-se de uma interface de alto nível que faz com que o presente subsistema seja mais fácil de ser utilizado. A classe ***BookmakerFacade*** conhece quais as classes presentes neste subsistema e irá fazer chamadas aos objetos de forma a delegar-lhes trabalho. Neste caso, irá incluir as classes *SportDAO*, *EventDAO*, *ParticipantDAO* e *BookmakerDAO*, portanto irá ter acesso a todos os objetos criados a partir dos dados relativos aos desportos, eventos, participantes e *bookmakers* que estão guardados na base de dados.

A classe ***Sport*** contém um tipo e uma tipologia, onde o tipo identifica se é um desporto de equipa ou individual, e a tipologia representa se o desporto além de vitória e derrota pode também acabar com um empate ou se é, então, um desporto de corrida. Além disso, inclui também uma lista de todas as competições ativas que esse desporto tem.

A classe ***Event***, além da data e hora de início, resultado final e estado do evento, tem também uma descrição que permite ao *bookmaker* atualizar essa mesma descrição de acordo com aquilo que está a acontecer no momento nesse evento. Além disso, contém o id do *bookmaker* que criou o evento, uma lista dos participantes, e ainda uma lista das escolhas que o utilizador irá ter disponível para apostar. Cada objeto ***Choice*** contém o resultado possível desse evento, a respetiva *odd* e ainda o estado dessa mesma *choice*. Já cada ***Participant*** além do seu nome, possui também uma lista com os id's das competições onde o mesmo participa.

A classe ***Bookmaker***, apenas contém o email e password já pré-definidos, para que o bookmaker consiga aceder à aplicação na sua página.

- UserBL

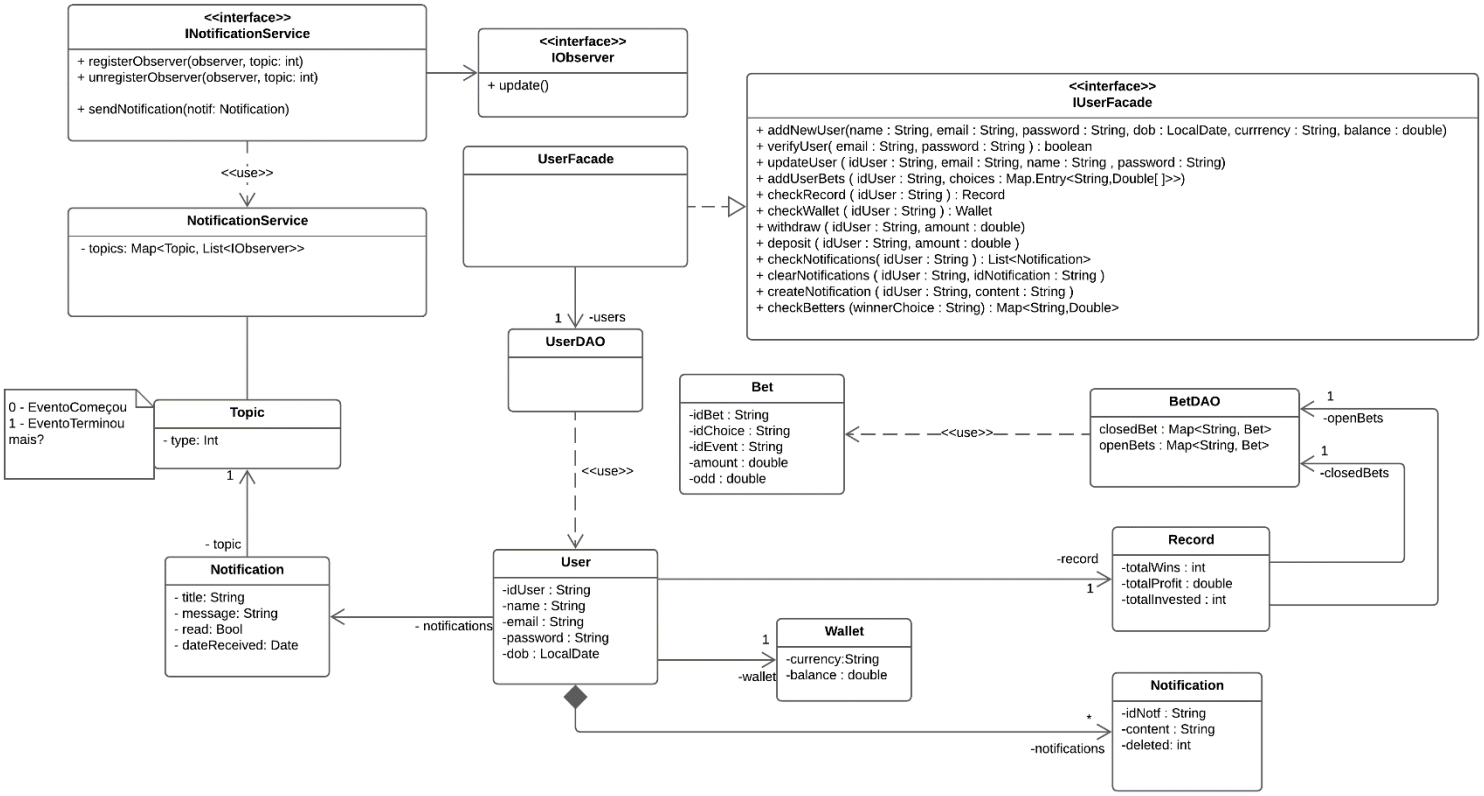


Figura 6 - Diagrama de classes relativas à BL do User

Este diagrama de classes representa o subsistema ***UserBL***, é aqui onde estão incluídas as classes e todos os métodos relacionados com o utilizador, as suas apostas realizadas e notificações.

O ***IUserFacade*** contém todos os métodos principais usados neste subsistema. A classe ***UserFacade*** possui todos os utilizadores que estão presentes na base de dados.

A classe ***User*** além das suas credenciais, contém também a carteira respetiva desse utilizador, uma lista das suas notificações, e ainda o seu Record, isto é, o seu histórico com a informação das apostas que ele fez e que ainda se encontram abertas bem como aquelas que já estão fechadas, além disso, inclui também as estatísticas desse utilizador.

Neste diagrama está também presente a especificação das entidades que constituem o serviço de notificações. Cada utilizador mantém um registo das notificações que lhe estão associadas. Cada objecto ***Notification*** existe sempre em relação a um ***Topic,*** que permite organizar o registo de *observers* que o ***NotificationService*** tem de servir. Este serviço é responsável pelo envio de notificação a todos os *observers*/utilizadores que estão registados.

6 "RUNTIME VIEW"

Para descrever de forma mais aprofundada a interação entre os diversos blocos do nosso sistema foram utilizados diagramas UML de sequência. Para isso foram escolhidas algumas funcionalidades principais que a aplicação deverá implementar.

Assim sendo, para representar a interação entre as diversas classes do sistema foram selecionados 8 métodos do *Facade* principal da lógica de negócio, aos quais será feita uma análise aprofundada acerca do comportamento dos mesmos. Esta decisão baseou-se no facto de quase todas as interações que o utilizador faz no *Frontend* desencadearem a chamada de algum método da classe *RASBetFacade*.

Para facilitar a compreensão dos diagramas, o grupo definiu que, para funcionalidades mais complexas se deveria representar o seu comportamento de uma forma mais generalizada. Pelo contrário, para os métodos mais simples optou-se por uma representação mais aprofundada, uma vez que nesses casos será vantajoso apresentar a informação de uma forma mais detalhada.

ATUALIZAR ESTADO DA APOSTA

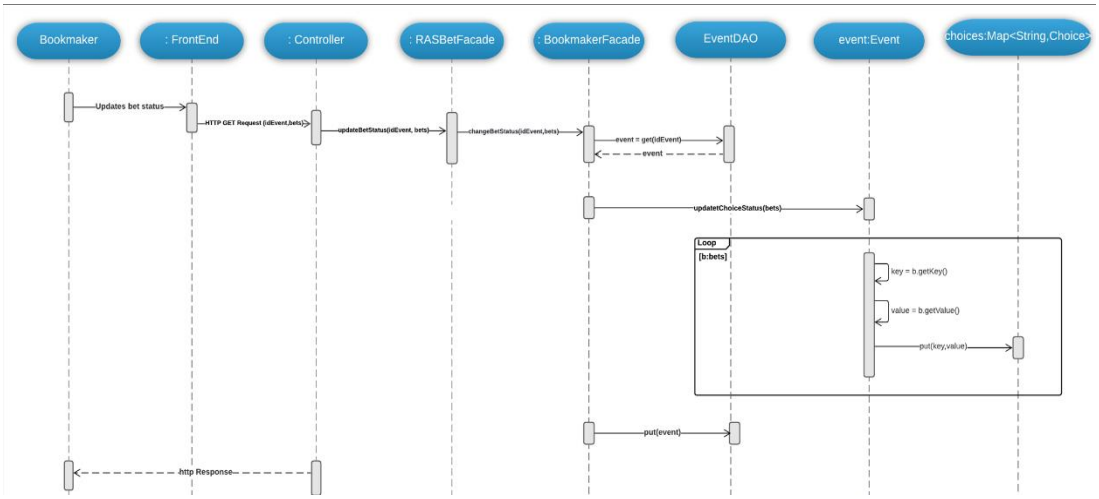


Figura 78 - Diagrama de sequência relativo à actualização do estado de uma posta

Uma das funcionalidades fundamentais do *bookmaker* nesta aplicação é a possibilidade de alterar o estado das *choices*. Desta forma, quando o *bookmaker* o desejar fazer terá que especificar não só o *id* das *choices* que quer alterar e os seus novos estados, como também o evento no qual essas *choices* foram feitas. Assim sendo, esta informação é colocada num *http request* que por sua vez é enviado ao *Controller*.

Ao receber este pedido, o *Controller* desencadeia o processo de atualização de estado das *choices*. Em primeiro lugar é necessário ir à base de dados e, através do *id*, obter o objeto evento correspondente. É de realçar que além do id do evento é passado ao método invocado

do *BookmakerFacade* uma estrutura *hashmap* com o *id* das *choices* e o novo estado que lhes queremos atribuir. Além disto, o próprio objeto evento contém uma *hashmap* que associa o *id* ao respetivo objeto *choice*. Desta forma, basta percorrer num *loop* todas as *map entries* do *hashmap q*ue contem o novo estado para cada *choice*, sendo que por cada iteração desse *loop* basta , a partir do id, ir à estrutura *hashmap* presente no objeto evento e obter o objeto choice no qual se atualiza o seu estado. Posto isto, coloca-se novamente no hashmap do evento a choice já alterada. Após o fim do *loop*, atualiza-se o objeto evento na base de dados.

Por fim, este processo termina com o *Controller* a enviar um *http response*.

CANCELAR EVENTO

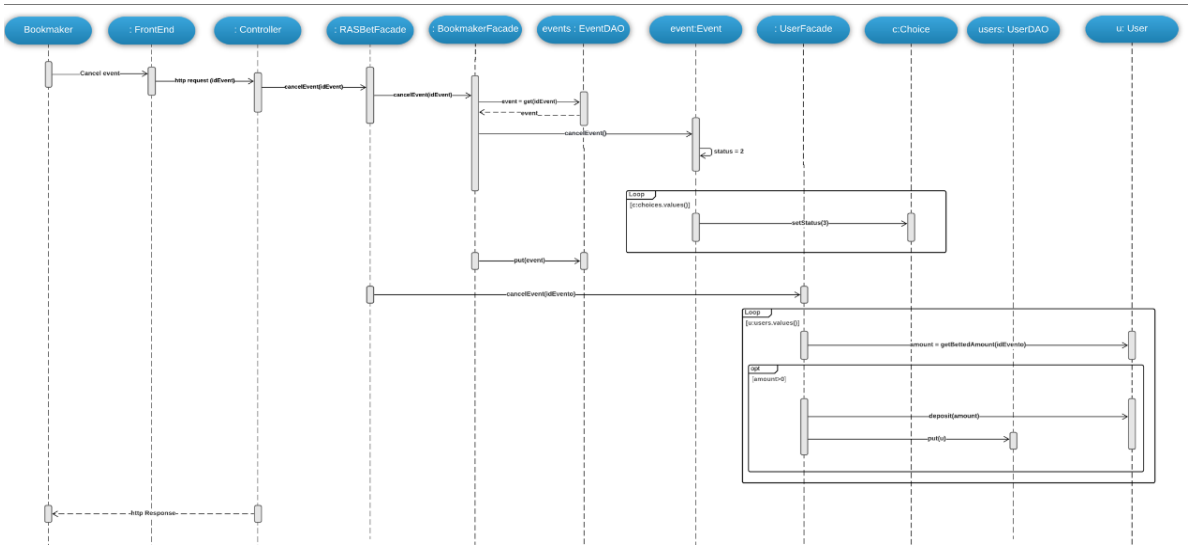


Figura 9 8 - Diagrama de sequência associado ao processo de cancelamento de um evento

Uma outra funcionalidae do *bookmaker* nesta aplicação é a possibilidade de cancelar um evento. Quando o *bookmaker* o desejar fazer terá que especificar o *id* do evento que pretende cancelar. Assim sendo, esta informação é colocada num *http request* que por sua vez é enviado ao *Controller.* Ao receber este pedido, o *Controller* desencadeia o processo de cancelamento do evento.

Em primeiro lugar é necessário aceder à base de dados e obter o objeto evento a partir do seu *id*. Após isto, atualiza-se estado desse evento para cancelado (status = 2) e através de um *loop*, atualizam-se também para o estado de canceladas, as *choices* desse evento. Posto isto, atualiza-se o objeto evento na base de dados.

Por outro lado, é ainda preciso devolver o dinheiro apostado aos utilizadores que apostaram no jogo que foi cancelado. Desta maneira, utiliza-se um *loop* para percorrer os utilizadores do sistema e, em cada iteração, verifica-se o valor que o *user* apostou no evento cancelado. Se o valor for zero então ele não apostou e nenhuma ação é tomada, caso esse valor seja superior a zero então o utilizador é apostar e é depositado na sua conta a quantia monetária apostada. É

importante realçar que cada vez que um depósito é feito na carteira de um *user*, este tem que ser atualizado na base de dados.

Por fim, este processo termina com o Controller a enviar um *http response*.

SUBMISSÃO DE UM EVENTO

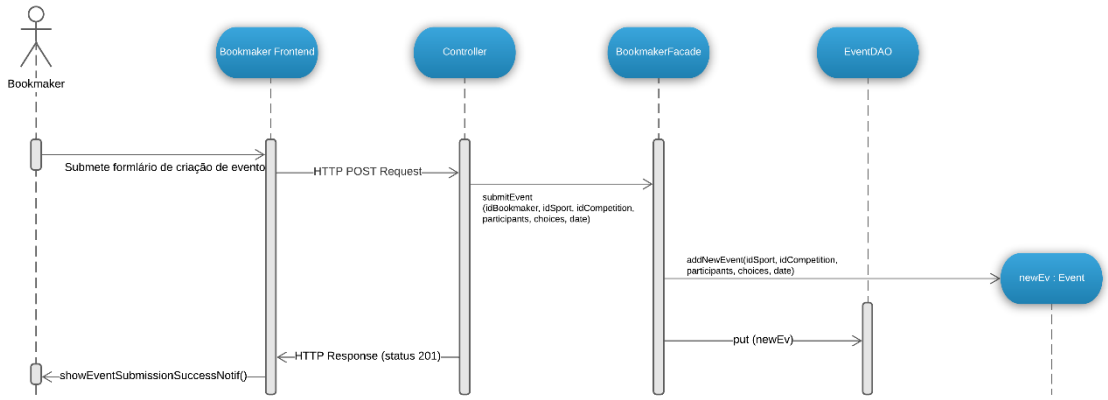


Figura 910 - Diagrama de sequência de submissão de um evento

O processo de submissão de um novo evento desportivo é iniciado por um utilizador *Bookmaker*, que submete o formulário que contem os dados constituintes do evento. O registo deste novo evento começa com a realização de um pedido *POST* HTTP com o *URI* `/api/event/sub`. O corpo deste pedido contém todos os dados necessários à criação de um novo evento:

- *idBookmaker*: identificador do *bookmaker*
- *idSport*: identificador do desporto associado ao evento
- *idCompetition*: identificador da competição/liga em que se dá o evento
- *participants*: identificadores dos participantes da partida
- *choices*: obecto que contém as diferentes configurações de apostas possíveis definidas pelo *bookmaker* para esse evento
- *date*: a data de início do evento

Ao receber este pedido HTTP, o *backend* procede à inserção desse novo evento na base de dados. Estando terminado este processo de inserção, o controlador *Controller* envia uma reposta HTTP de confirmação do sucesso ao *frontend* associado ao *bookmaker,* a quem é apresentada uma mensagem informativa acerca do sucesso da sua operação.

ENCERRAMENTO/CONCLUSÃO DE UM EVENTO

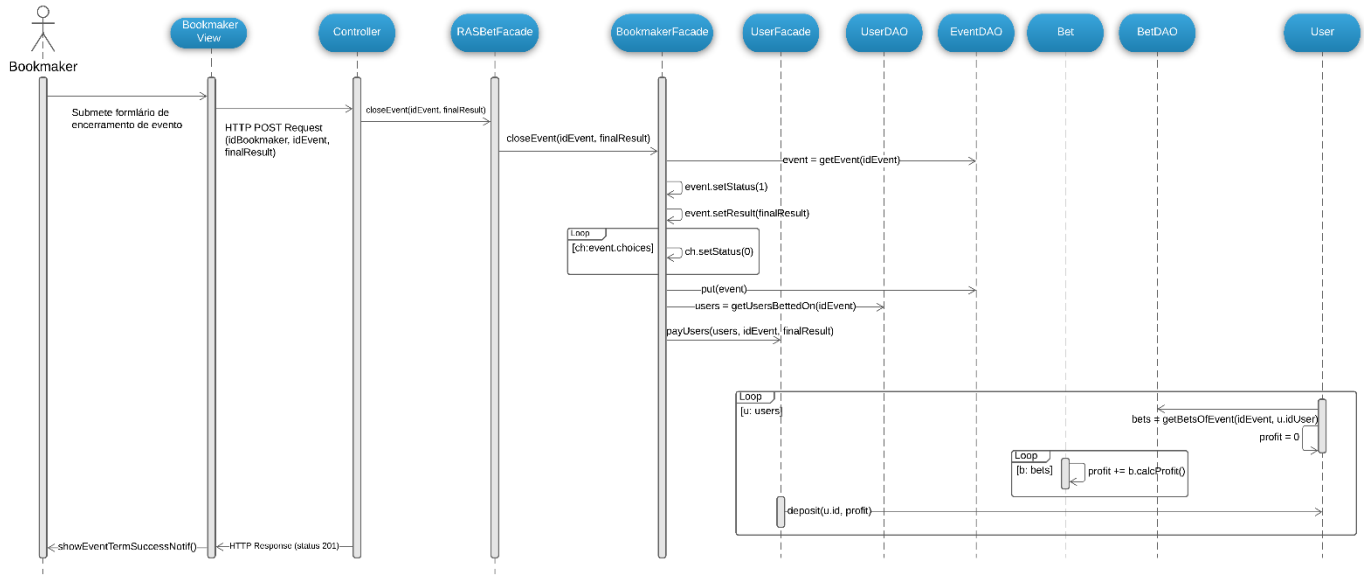


Figura 10 - Diagrama de sequência da conclusão de um evento

Para que se possa encerrar um evento, é primeiro necessário que o *bookmaker* responsável especifique o resultado final do jogo. Esta informação é embebida no HTTP POST *request* que é enviado ao controlador do *backend*, à qual se juntam os identificadores do *bookmaker* e do evento em questão.

Ao receber este pedido, o *Controller* desencadeia o processo de encerramento do evento:

1. Obter uma cópia do evento através do seu id: `event = getEvent(idEvent)`
2. Definir o estado desse evento como "Terminado": `event.setStatus(1)`
3. Definir o resultado do evento de acordo com o parâmetro `finalResult` especificado pelo *bookmaker*: `event.setResult(finalResult)`
4. Itera-se sobre a lista de todas as `choices` associadas a esse evento, e em cada uma é alterado o estado para `Closed`: `ch.setStatus(0)`
5. O evento é novamente submetido na base de dados, contendo agora a informação relativa ao seu encerramento: `put(event)`

Estando o estado do evento completamente actualizado, procede-se à creditação dos lucros associados ao resultado das apostas que cada utilizador realizou sobre esse evento:

1. Obtém-se, para cada utilizador 'u', as apostas que este realizou sobre esse evento: `bets = getBetsOfEvent(idEvent, u.idUser)`
2. Para cada uma dessas apostas, calcula-se o lucro: `profit += b.calcProfit()`
3. Credita-se o lucro total das apostas nesse evento: `deposit(u.id, profit)`

SUBMISSÃO DO BOLETIM DE APOSTAS

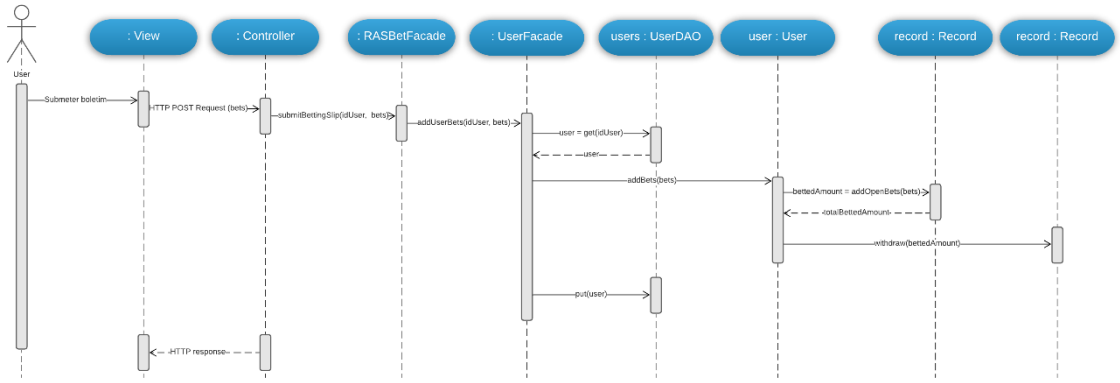


Figura 1112 - Diagrama de sequência da submissão de um boletim de apostas

Após o utilizador selecionar o conjunto de apostas que pretende efetuar, e selecionar a opção de submissão do boletim, é feito um pedido ao *Controller* e consequentemente à camada de negócio para que sejam guardados os dados dessa mesma submissão.

Desta forma, é invocado o método *submitBettingSlip* com o id do utilizador que fez a submissão e respetivas apostas que ele efetuou na forma de um *Map<String,Map.Entry<String,Double[ ]>>.* Esta estrutura de dados à qual é dado o nome de *bets,* associa cada chave que é o id da *Choice* com um valor que, por sua vez, é um par constituído pelo id do evento (*Event*) e um array de tamanho 2 onde o primeiro elemento é o montante apostado e o segundo a *Odd* da respetiva *Choice*. Com esta informação será possível criar os novos objetos da classe *Bet* correspondentes a cada aposta efetuada pelo utilizador. Estes objetos serão guardados no histórico (*Record*) do utilizador na lista de apostas abertas (*openBets*).

ATUALIZAÇÃO DO SALDO DA CARTEIRA DO UTILIZADOR

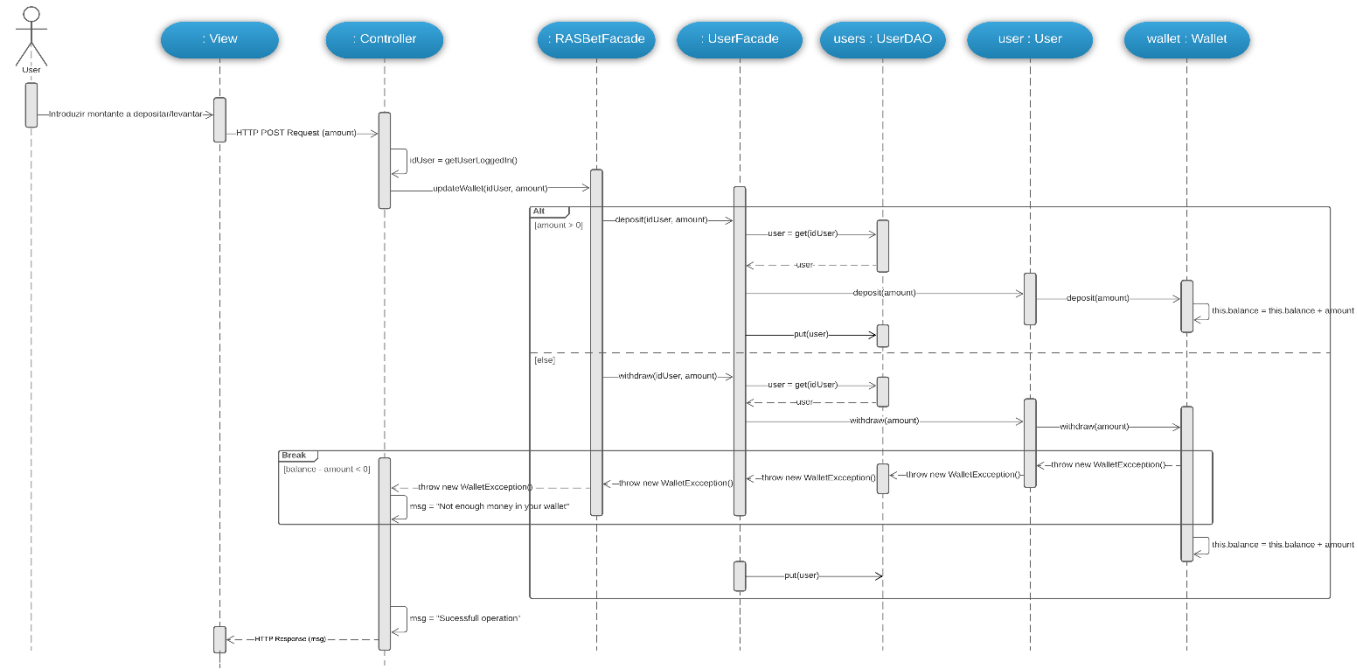


Figura 12 13- Diagrama de sequência do processo de atualização do saldo de um utilizador

Para levantar ou depositar dinheiro, o utilizador indica o montante respetivo e o *Frontend* irá fazer o devido pedido ao *Backend.* O método *updateWallet* da classe *RASBetFacade* irá fazer todo o processo de atualização da carteira do utilizador.

Assim sendo, tanto para levantar como para depositar, será necessário, em primeiro lugar obter o utilizador que efetua a transação e por fim atualizar o saldo (*balance*) da sua carteira (*Wallet*). No caso do levantamento pode haver a exceção de o utilizador não ter dinheiro suficiente.

MOSTRAR TODOS OS EVENTOS

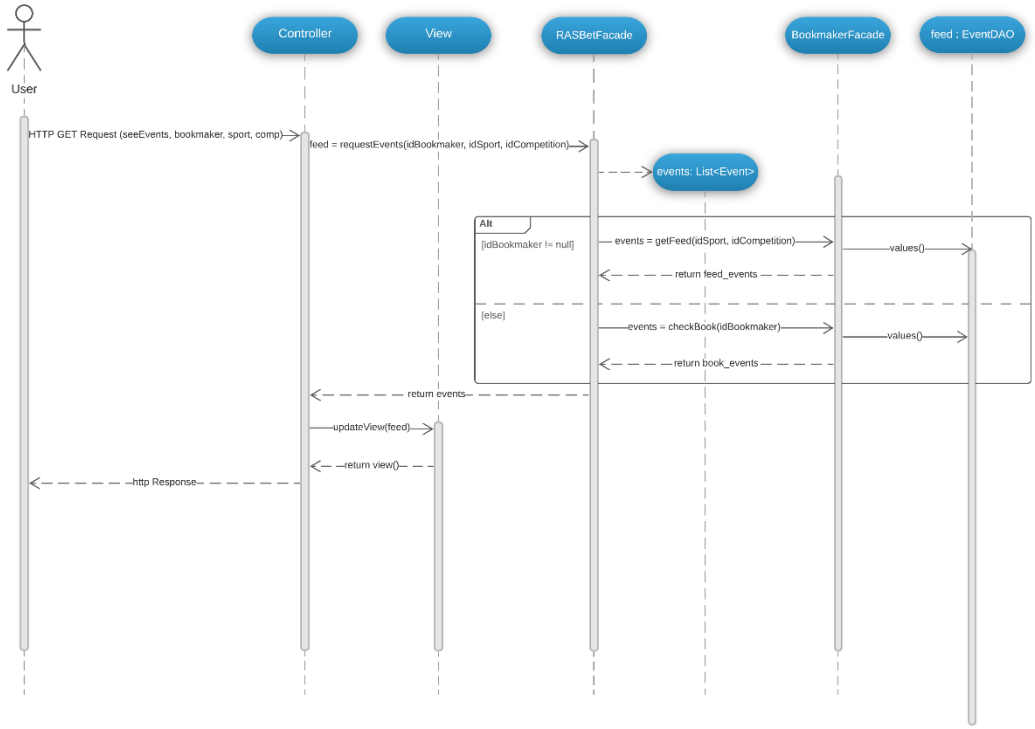


Figura 14 - Diagrama de sequência do processo de mostrar todos os eventos

Quando um utilizador acede à aplicação pela primeira vez é-lhe mostrado na página inicial uma lista de eventos nos quais pode apostar. Para obter essa lista é invocado o método *requestEvents* do RASBetFacade que obtém todos os eventos guardados na base de dados. É ainda possível efetuar uma filtragem do *Feed* por desporto e competição.

ATUALIZAR EVENTO

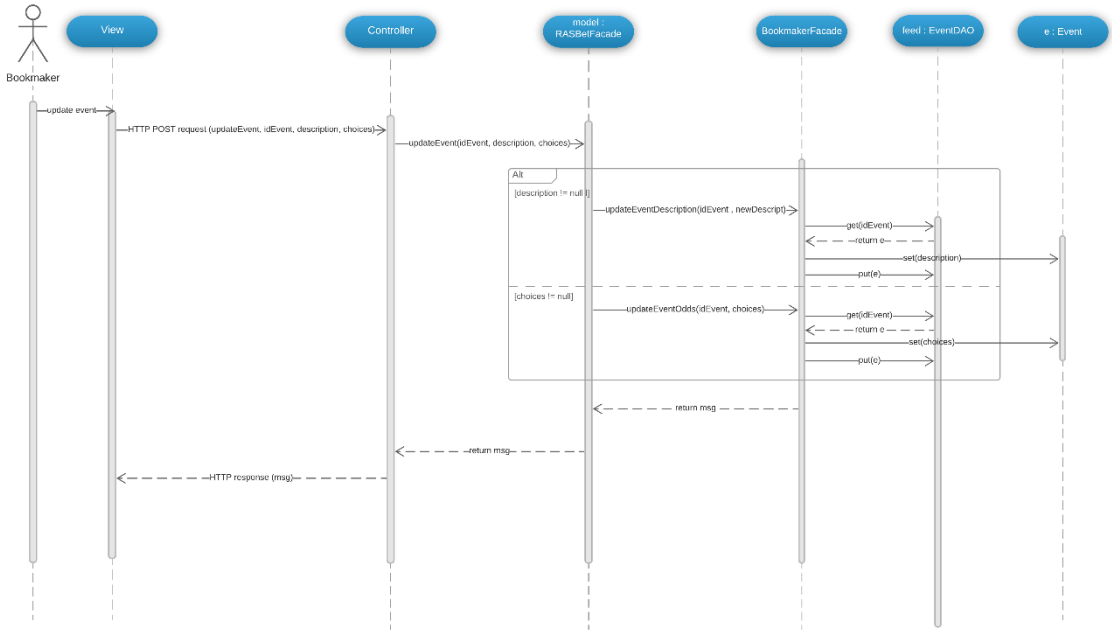


Figura 15 - Diagrama de sequência da atualização dos dados de um evento

À medida que o tempo vai passando, os eventos da aplicação podem sofrer alterações ao longo do tempo, as quais devem ser efetuadas pelo Bookmaker. Desta forma foi definido o método *updateEvent*, que permite a atualização, tanto da descrição como também das *odds* das *choices* associadas a um dado evento (*Event*).

7 DEPLOYMENT VIEW

No diagrama de *deployment*, é possível representar a estrutura física onde os vários componentes de *software* do sistema estão contidos. Assim sendo, e visto que não foi previsto um *deployment* desses componentes em serviços como a AWS ou *Google Cloud*, apenas existe um *device* que é o *laptop*. Neste nodo existem contidos dois componentes que correspondem ao *frontend* e ao *backend*. Para além disto, no *device* existem também um novo nodo que contem um componente que representa a base de dados *sql* do sistema.

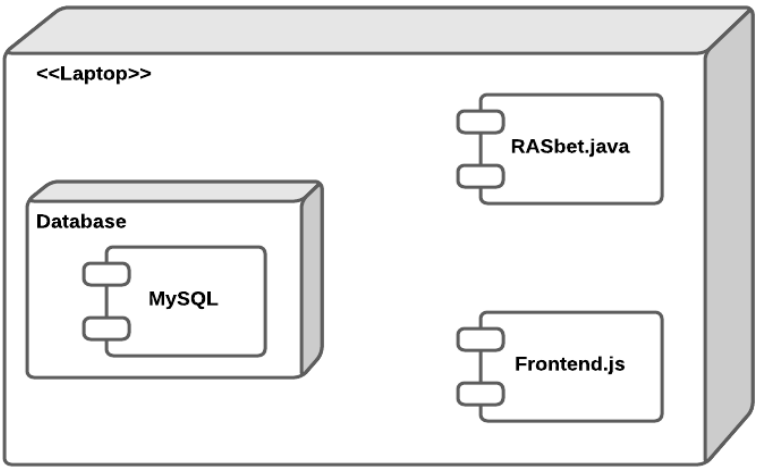


Figura - Diagrama do modelo de deployment

8 CROSS-CUTTING CONCEPTS

FRAMEWORK SPRING BOOT: ANOTAÇÕES, *DECOUPLING* E TERMINOLOGIA

A utilização de um framework como o Spring Boot é extremamente vantajosa, na medida em que o seu *workflow* consiste, essencialmente, na definição de configurações recorrendo maioritariamente a **anotações**. Estas anotações, quando estão associadas a classes ou objectos, determinam o seu comportamento e função. Este método de especificação permite **acelerar consideravelmente o desenvolvimento**.

A título de exemplo, considere-se que se pretende criar uma entidade de controlo que define uma interface RESTful para recepção e envio de *requests/responses* HTTP num URI `api/v1/event`:

```
1.  @RestController
2.  @RequestMapping(path = "api/v1/event")
3.  public class EventController
4.  {
5.      private final EventService eventService;
6.
7.      @Autowired
8.      public EventController(EventService eventService) {
9.          this.eventService = eventService;
10.     }
11.
12.     @GetMapping
13.     public List<Event> getEvents() {
14.         return eventService.getAllEvents();
15.     }
16.
17.     @PostMapping
18.     public void registerNewEvent(@RequestBody Event event) {
19.         eventService.addNewEvent(event);
20.     }
21. }
22.
```

Este excerto de código é totalmente funcional, e pode ser imediatamente integrado num projecto. No entanto, o processo de definição desta entidade de controlo passou **apenas** por:

1. definir a classe *EventController* como sendo um *RestController* (linha 1)
2. definir o URI para o qual são enviados os HTTP *requests* (linha 2)
3. definir os *endpoints* associados aos *Get* e *Post requests* (linhas 12 e 17)

Existem ainda outros mecanismos extremamente vantajosos e que podem ser obtidos através utilizando apenas anotações.

Seguindo o exemplo anterior, suponha-se agora que se pretende realizar ***caching*** dos eventos com o intuito de evitar acessos desnecessários à base de dados. Ora, este mecanismo de optimização pode ser simplesmente implementado utilizando a anotação `@Cacheable`:

```
1.  @Cacheable("events")
2.  public List<Event> getAllEvents() {
3.      return eventRepository.findAll();
4.  }
```

O funcionamento do código acima é simples. Sempre que a função `getAllEvents` é invocada, o sistema verifica primeiro se já existe uma lista de eventos em cache associada a uma key "events". Se tal de facto se verificar, então o código da linha 3 não chega a ser executado, evitando assim a realização de um acesso ao repositório de eventos (que implicaria a realização de *queries* à BD), e a **lista *cached* de eventos é devolvida como o resultado da função.**

Existe também uma preocupação relativamente ao **grau de *coupling*** que pode emergir à medida que a complexidade do sistema aumenta. O Spring Boot tem também anotações úteis que auxiliam a diminuição de relações de dependência entre objectos/sistemas. O exemplo mais comum será o padrão de **injecção de dependências**. Isto é algo que o Spring Boot providencia com necessidade mínima de configuração:

```
1.  @Autowired
2.  public EventController(EventService eventService) {
3.      this.eventService = eventService;
4.  }
```

Neste exemplo, o controlador `EventController` é inicializado com uma referência a um objecto `EventService`. No entanto, a criação do controlador é orquestrada completamente por mecanismos implementados pelo *framework*. Este é capaz de resolver as dependências que o controlador precisa no momento da sua criação e efectua as associações necessárias entre os dois objectos.

REUTILIZAÇÃO DE CÓDIGO/FUNCIONALIDADES NO *FRONTEND*

Tendo em vista suportar o desenvolvimento de outros tipos de aplicações com as quais os utilizadores finais possam interagir, decidiu-se enveredar pela utilização de uma arquitectura baseada em camadas de abstração. Esta linha de raciocínio baseia-se nas seguintes razões:

1. Pretende-se que existam várias formas de interface de interação com o utilizador (bookmaker e apostador)
2. É desejável que a implementação de uma nova interface possa tirar proveito dos mecanismos anteriormente criados
3. Manutenção de várias code-bases pode tornar-se dispendiosa à medida que a complexidade e o número de interfaces aumentam. É extremamente útil que a correção de erros num módulo possa ser propagada a todos os outros componentes que dele dependem.

Estipulou-se então a seguinte arquitectura:

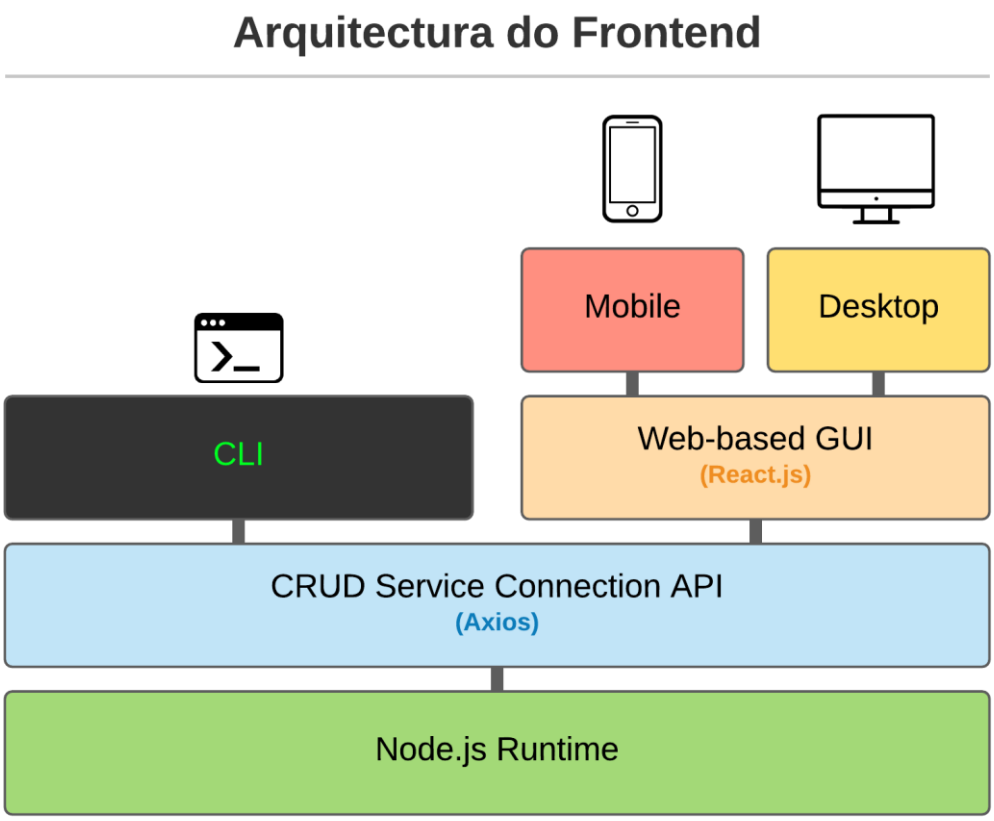


Figura 17 - Arquitectura baseada em camadas de módulos

A camada basilar é constituída apenas pelo módulo associado ao *runtime* do Node.js. A utilização deste *runtime* não é inócua. De facto, o Node.js foi propositadamente escolhido por suportar o desenvolvimento em linhas de comando e por, ao mesmo tempo, permitir a criação

de aplicações mais complexas cujo ambiente/meio de execução **não está limitada aos browsers.**

A camada seguinte será responsável pela gestão da ligação com o *backend*, pela realização de pedidos e recepção de respostas HTTP. Concretamente, irá recorrer-se a uma libraria NPM denominada *Axios*.

Finalmente, existe a camada relacionada com a implementação base das diferentes interfaces de interação. O desenvolvimento do módulo associado à interface em linha de comandos (CLI) é prioritário. No entanto, está prevista a criação de um módulo, utilizando a libraria React.js, dedicado ao desenvolvimento de interfaces gráficas (GUI's). Estando esta versão da aplicação pronta, é possível, mais tarde, realizar deployment em vários dispositivos: smartphones, desktops (em web browsers ou até em *webviews* que simulam comportamento e performance nativos).

9 DECISÕES DE DESIGN

FACADE

Uma das principais decisões que o grupo tomou foi usar o ***Facade*** como padrão de design de software neste projeto, uma vez que, o sistema com o qual se está a lidar é por vezes muito complexo e difícil de entender, já que possui um grande número de classes e também de dependências entre as mesmas. Este design permite deste modo esconder as complexidades de um sistema maior, através do uso de uma interface simplificada. Sendo assim, o *facade* é usado para definir um ponto de entrada para cada nível de subsistema

Primeiramente, foi necessário estruturar o sistema em subsistemas, ajudando desta forma a reduzir a sua complexidade. O objeto *Facade* permite minimizar a dependência existente entre os subsistemas, uma vez que fornece uma interface única que inclui as diversas funcionalidades do subsistema em questão.

Neste projeto foi então definida uma interface de alto nível que unifica as duas interfaces que representam cada subsistema existente. Deste modo, a interação com o sistema é facilitada. É de notar que a classe *Facade* tem a responsabilidade de encaminhar as chamadas aos objetos dos subsistemas corretos, enquanto as classes dos subsistemas não têm conhecimento do *Facade*, e têm de lidar com o trabalho que lhes foi atribuído pelo *Facade*.

Concluindo, como existe a necessidade de abstrair funcionalidades, a solução é providenciar uma interface que esconde detalhes internos, daí o grupo escolher o ***Facade*** como solução de design.

DATA ACCESS OBJECT

As classes DAO não só permitem a persistência dos objetos na BD, como também criam objetos a partir da informação disponível na BD, além disso também encapsulam queries SQL. Desta forma obtemos uma separação concisa entre a camada de dados e a lógica de negócio, permitindo assegurar a facilidade de manutenção do código a implementar.

10 REQUISITOS DE QUALIDADE

Para que a aplicação funcione da melhor forma possível foram definidas várias formas de garantir a sua qualidade.

Para garantir uma resposta rápida aos pedidos efetuados foram escolhidas estruturas de dados (*hashmaps*) que permitam um o menor tempo de execução possível dos algoritmos. O grupo teve o cuidado de para quase todas as funcionalidades, os algoritmos a implementar terem uma complexidade inferior a $O(n)$.

Do mesmo modo foram também evitados acessos desnecessários à base de dados, sendo que foram efetuadas, sempre que possível, todas as alterações no nível da camada da lógica de negócio.

Por último é importante referir que todos os requisitos não funcionais definidos na primeira fase do projeto foram mantidos e podem ser revistos nos tópicos 10 a 17 do relatório da mesma.

11 RISCOS E DÍVIDAS TÉCNICAS

De seguida, por ordem decrescente de risco associado (e prioridade de resolução), apresentam-se os tópicos mais pertinentes respeitantes, principalmente, à necessidade de serem implementados, no futuro, sistemas capazes de conferir e assegurar um grau de segurança, integridade e resiliência mais elevado.

ENVIO, RECEPÇÃO E ARMAZENAMENTO DE DADOS NÃO ENCRIPTADOS

A arquitectura actual ignora a necessidade de serem implementados sistemas intermédios dedicados aos processos de encriptação e desencriptação dos dados. Esta falta de segurança, evidenciada principalmente no *backend*, existe, por exemplo, sob forma de armazenamento de palavras-passe ou saldos dos utilizadores sem recorrer a qualquer tipo de encriptação ou processo de verificação de integridade.

SEGURANÇA DOS FICHEIROS DE *LOG*

A integridade e segurança dos ficheiros de *logging*, referentes, por exemplo, às transações monetárias, apostas, submissões de eventos, etc…, não estão minimamente asseguradas.

Possíveis Soluções:

1. Replicação dos ficheiros de *logging*, recorrendo a diferentes meios (físicos ou virtuais) de armazenamento.
2. Introduzir sistemas de verificação de integridade

12 GLOSSÁRIO

- *BL (Business Logic)* – Referente à lógica de negócio. Trata-se do conjunto de sistemas que determina o funcionamento dos processos de criação, transformação e armazenamento dos dados.
- *ORM (Object Relational Mapping)* – Técnica para aproximar o paradigma de desenvolvimento de aplicações orientada a objetos ao paradigma do banco de dados relacional. Providencia uma Interface configurável que determina o as funcionalidades disponíveis respeitantes aos processos mais comuns quando se pretendem realizar operações de escrita e leitura sobre uma BD.
- *DAO (Data Access Object)* – Padrão para aplicações que utilizam persistência de dados. São, normalmente, objectos mediadores da interação entre a BL e a Base de Dados, encapsulando/escondendo os detalhes internos da Base de Dados.