



UNIVERSIDADE FEDERAL DO CEARÁ
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA DE TELEINFORMÁTICA
CURSO DE GRADUAÇÃO EM ENGENHARIA DE TELEINFORMÁTICA

iQuizzer: Integrando aplicações web e dispositivos móveis em um ambiente para criação e execução de quizzes

Autor

Tiago Augusto da Silva Bencardino

Orientador

Prof. Dr. José Marques Soares

FORTALEZA – CEARÁ
22 DE FEVEREIRO DE 2013



UNIVERSIDADE FEDERAL DO CEARÁ
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA DE TELEINFORMÁTICA
CURSO DE GRADUAÇÃO EM ENGENHARIA DE TELEINFORMÁTICA

iQuizzer: Integrando aplicações web e dispositivos móveis em um ambiente para criação e execução de quizzes

Autor

Tiago Augusto da Silva Bencardino

Orientador

Prof. Dr. José Marques Soares

*Projeto Final de curso submetido à
Coordenação do programa de Graduação
em Engenharia de Teleinformática da
Universidade Federal do Ceará como parte
dos requisitos para obtenção do grau de
Engenheiro de Teleinformática.*

FORTALEZA – CEARÁ
22 DE FEVEREIRO DE 2013

TIAGO AUGUSTO DA SILVA BENCARDINO

**iQuizzer: Integrando aplicações web e dispositivos móveis em um ambiente
para criação e execução de quizzes**

Esta Monografia foi julgada adequada para a obtenção do diploma de Engenheiro do Curso de Graduação em Engenharia de Teleinformática da Universidade Federal do Ceará.

Tiago Augusto da Silva Bencardino

Banca Examinadora:

Prof. Dr. José Marques Soares (Orientador)
Universidade Federal do Ceará - UFC

Prof. Msc. Alexandre Moreira de Moraes
Universidade Federal do Ceará - UFC

Prof. Dr. Danielo Gonçalves Gomes
Universidade Federal do Ceará - UFC

Fortaleza, 22 de Fevereiro de 2013

Resumo

Com o crescente desenvolvimento da mobilidade e da ubiquidade, um grande número de aplicações Web têm sido desenvolvidas, sendo as plataformas móveis *iOS* e *Android* algumas das tecnologias de destaque à época em que este trabalho foi escrito. Paralelamente, muitos recursos distribuídos se conjugam através de técnicas e organizações específicas, como as arquiteturas orientadas a serviços. Para exemplificar, algumas redes sociais populares, como *Twitter* e *Foursquare*, possuem interfaces de comunicação baseadas no uso de *REST* para exportar funcionalidades, em forma de serviços para outras aplicações.

Este trabalho tem por objetivo apresentar um estudo em que se mostra o desenvolvimento de uma aplicação Web que integra aplicações *iOS* e *Android* a um aplicativo web *RESTful*. Para realizar o estudo, é criada uma pequena rede social de criação de quizzes, utilizando *Ruby on Rails* e um serviço de PaaS, como o *Heroku*.

Palavras-chave: iOS, Android, Ruby on Rails, REST, Computação em Nuvem

Abstract

With the increasing development of mobility and ubiquity, a large number of web applications have been developed, where *iOS* and *Android* mobile platforms were ones of the most featured technologies when this paper was written. Meanwhile, many distributed resources are combined using techniques and specific organizations, such as service-oriented architectures. To illustrate, some popular social networks like *Twitter* and *Foursquare*, have communication interfaces based on the use of *REST* to export functionalities, as services to other applications.

This paper aims to present a study which shows the development of a web application that integrates *iOS* and *Android* applications to a *RESTful* web application. To conduct the study, it created a small social network to create quizzes using *Ruby on Rails* and a service PaaS like *Heroku*.

Keywords: iOS, Android, Ruby on Rails, REST, Cloud Computing.

“Cada sonho que você deixa pra trás é um pedaço do seu futuro que deixa de existir.”

Steve Jobs

Agradecimentos

Primeiramente, gostaria de agradecer a Deus, que sempre me acompanhou e me guiou em todos os momentos.

A meus pais, Antonio e Monica, por todos os ensinamentos e incentivos essenciais à formação de meu caráter. Às minhas irmãs, Tayanne e Isabella, pela companhia e descontração no dia-a-dia.

Ao professor Alexandre Sobral, pelos ensinamentos iniciais em lógica de programação. Ao professor Danielo Gomes, pelos ensinamentos na disciplina de computação móvel e pela oportunidade de docência como monitor desta disciplina. Ao professor, orientador e amigo José Marques Soares, pelos ensinamentos essenciais no desenvolvimento deste trabalho e conselhos de toda natureza. A todos os professores do departamento de Engenharia de Teleinformática, que de alguma forma contribuíram para minha formação acadêmica.

Ao GREat e colegas de trabalho, pela troca de experiências acadêmicas e profissionais.

À empresa Xseed Software, representada por seus diretores Ronaldo Brandão e Luiz Miranda, por toda a formação profissional e oportunidades de pesquisas durante os últimos três anos. A todos os colaboradores, em especial Marcos Sousa, Christiane Girão, Pedro Rolim, Felipe Brayner e Cleiton Uchoa, por todo o compartilhamento de conhecimento.

Aos amigos de curso Lucas Maia, Pedro Gabriel, José Felipe, Felipe Barbosa, Paulo Torres, Ronaldo Milfont, Mairton Junior, Marciel Barros, Jefferson Figueiredo, Tiago Gomes e Suelen Moura, dentre tantos outros, pelos dias de estresse e companhia nas noites viradas de estudo.

À namorada e amiga especial Luciana Borges, pelo carinho, apoio e compreensão nos momentos mais difíceis. Aos amigos da vida Indira Gurgel, Julio Higa, Filipe Alirio, Paulo Ernesto, Vinicius Ferreira, Renata Araujo, Leonnardo Rabello e Amanda Craveiro, dentre tantos outros, que sempre me acompanharam com suas amizades.

Lista de Acrônimos

CRUD	<i>Create, read, update e delete</i>
CSS	<i>Cascading Style Sheets</i>
DAO	<i>Data Access Object</i>
DRY	<i>Don't repeat yourself</i>
GUI	<i>Graphic User Interface</i>
HTML	<i>HyperText Markup Language</i>
HTTP	<i>Hypertext Transfer Protocol</i>
JSON	<i>JavaScript Object Notation</i>
JVM	<i>Java Virtual Machine</i>
IDE	<i>Integrated Development Environment</i>
MVC	<i>Model View Controller</i>
NFC	<i>Near Field Communication</i>
NIST	<i>National Institute of Standards and Technology</i>
OHA	<i>Open Handset Alliance</i>
PaaS	<i>Platform as a Service</i>
REST	<i>Representational State Transfer</i>
SGBD	<i>Sistema de Gerenciamento de Banco de Dados</i>
SOAP	<i>Simple Object Access Protocol</i>
SSH	<i>Secure Shell</i>
TDD	<i>Test Driven Development</i>

URL	<i>Uniform Resource Locator</i>
XML	<i>Extensible Markup Language</i>
YAML	<i>YAML Ain't Markup Language</i>
YARV	<i>Yet Another Ruby VM</i>

Sumário

Lista de Figuras	x
1 Introdução	1
1.1 Contextualização	1
1.2 Objetivos	2
1.2.1 Objetivos Gerais	2
1.2.2 Objetivos Específicos	2
1.3 Metodologia utilizada	3
1.4 Estrutura da monografia	3
2 Fundamentação teórica e ferramentas	5
2.1 Serviços PaaS	5
2.1.1 Definição	5
2.1.2 Heroku	6
2.2 REST	6
2.2.1 RESTful	7
2.3 JSON	7
2.3.1 Introdução	7
2.3.2 Definição	8
2.3.3 comparação com XML	8
2.3.4 Estrutura	9
2.4 Ruby on Rails	10
2.4.1 Ruby	10
2.4.2 Rails	10
2.5 Smartphones	12
2.6 iOS	13
2.6.1 Visão Geral	13
2.6.2 Objective-c	14

2.6.3	Ciclo de vida	14
2.7	Android	15
2.7.1	Visão Geral	15
2.7.2	Java	15
2.7.3	Ciclo de vida	16
2.8	Resumo do Capítulo	17
3	Especificação da aplicação	18
3.1	Visão geral	18
3.2	Requisitos	19
3.2.1	Requisitos funcionais	19
3.2.2	Requisitos não-funcionais	19
3.3	Casos de uso	20
3.4	Diagrama de classes	20
4	Implementação <i>web</i>	22
4.1	Ambientação	22
4.2	Criação de aplicação	23
4.3	Estrutura	23
4.4	Banco de dados	24
4.4.1	PostgreSQL	24
4.4.2	Modelos	25
4.4.3	Relacionamentos	25
4.5	Rotas e REST	27
4.6	Autenticação	28
4.7	<i>Front-End</i>	31
4.7.1	<i>Bootstrap</i>	33
4.8	<i>Deploy</i>	34
5	Comparativo Android x iOS	36
5.1	Ambientação	36
5.1.1	iOS	36
5.1.2	Android	36
5.2	Mostrando textos	37
5.2.1	iOS	37
5.2.2	Android	37
5.3	Inserindo textos	37
5.3.1	iOS	37
5.3.2	Android	38
5.4	Botões e eventos	38

5.4.1	iOS	38
5.4.2	Android	39
5.5	Criando listas	40
5.5.1	iOS	40
5.5.2	Android	42
5.6	Acesso a dados	43
5.6.1	SQLite	43
5.6.2	Preferências	45
5.7	Parser JSON	46
5.7.1	iOS	46
5.7.2	Android	47
5.8	Conexão HTTP	48
5.8.1	iOS	49
5.8.2	Android	49
6	Conclusão	51
6.1	Contribuições	51
6.2	Limitações	51
6.3	Trabalhos futuros	52
	Referências Bibliográficas	53

Lista de Figuras

2.1	<i>Marketshare</i> Q3 2012	13
3.1	Diagrama de casos de uso para “criador” e “jogador”	20
3.2	Diagrama de entidades	21
4.1	Tela mostrada quando o usuario tenta acessar recurso e nao está logado .	30
4.2	Edição de dados do usuário	30
4.3	Lembrar senha do usuário	30
4.4	<i>Login</i> no <i>mobile</i> (iOS)	31
4.5	Exemplo de controle <i>Bootstrap</i>	33
4.6	Tela desktop extendida	33
4.7	Tela <i>mobile</i> com menu suprimido	34
4.8	Tela <i>mobile</i> com menu extendido	34
5.1	Criando ligação entre interface builder e o código	39
5.2	Criando IBAction e associando evento	39
5.3	Ferramenta gráfica para manipulação do banco de dados utilizando o <i>xcdatamodel</i>	43

Introdução

Este capítulo visa apresentar o contexto no qual o trabalho realizado está inserido, assim como definir seus objetivos e justificar seus propósitos. Na Seção 1.1, é apresentada uma contextualização para a solução gerada. Os objetivos geral e específicos são indicados na Seção 1.2. Por fim, a Seção 1.3 realiza uma breve descrição da metodologia empregada e a Seção 1.4 descreve o formato no qual esta monografia está organizada.

1.1 Contextualização

A computação em nuvem propõe que recursos de *hardware* e *software* sejam providos como serviço pela internet. Sob essa visão, sistemas como *web services* podem ser hospedados sobre uma infraestrutura de terceiros, de modo a diminuir custos operacionais.

Web service é uma solução utilizada para realizar a comunicação entre sistemas distintos, que podem utilizar diferentes tecnologias de implementação e estar sob as mais diversas plataformas. Assim, aplicações feitas por equipes diferentes, em diferentes contextos, podem se comunicar e executar ações ou consumir recursos.

A função de um *web service* é possibilitar que os recursos de uma aplicação de *software* estejam disponíveis na rede de uma forma padronizada, ou seja, sobre um protocolo em que cliente e servidor entendam a mensagem.

Existem algumas formas padronizadas de implementar serviços, sendo as mais comuns as soluções em *Simple Object Access Protocol* ([SOAP](#)) e *Representational State Transfer* ([REST](#)). [REST](#) é um padrão de *web service* mais comum em computação móvel.

A computação móvel pode ser representada como um novo paradigma computacional que permite a usuários desse ambiente terem acesso a serviços independentemente de sua localização podendo, inclusive, estar em movimento [1]. Originalmente, telefones celulares possuíam apenas a capacidade de realizar a comunicação por voz. Com o passar do tempo,

algumas funcionalidades extras foram sendo implementadas e, atualmente, os dispositivos possuem um amplo poder de processamento e comunicação. Tais aparelhos receberam a nomenclatura comercial de *smartphone*.

Smartphones (ou telefones inteligentes) são celulares com funcionalidades avançadas, que podem ser estendidas através de programas adicionados ao seu sistema operacional. Em sistemas operacionais como *iOS* e *Android*, existem uma vasta gama de aplicativos que podem ser baixados em lojas virtuais, como *App Store* e *Google Play*, respectivamente. As categorias líderes em número de aplicativos baixados são jogos e sociais.

Rede social é uma estrutura composta por pessoas ou organizações conectadas, de modo a partilhar valores e objetivos comuns. Existem diversos tipos de redes, como as de relacionamento (*Facebook*, *Orkut*, *Twitter*), redes profissionais (*LinkedIn*), redes comunitárias (*Cromaz*), entre outros.

Em redes de relacionamentos como *Facebook* e *Orkut*, surgiram alguns jogos que possibilitam a interação de pessoas, aproveitando características de integração social. Tais jogos recebem a denominação de *social network gaming*.

1.2 Objetivos

Esta seção visa descrever os objetivos deste trabalho através de um panorama geral de seus propósitos e da descrição de pontos específicos que devem ser atendidos.

1.2.1 Objetivos Gerais

O principal objetivo desse trabalho é a criação de uma plataforma *web*, hospedada mediante serviço de computação em nuvem, que interaja com duas aplicações *mobile* (em *iOS* e em *Android*), através de um serviço de *web service* utilizando [REST](#).

A aplicação *web* permitirá a criação de questionários e enquetes, os quais denominamos de *quizzes*, bem como o acompanhamento das respostas marcadas. Por sua vez, as aplicações móveis em *iOS* e *Android* permitirão que os usuários possam baixar os *quizzes* de seu interesse e jogá-los, de forma competitiva ou apenas responder, a depender do modo de jogo inerente ao *quiz* em questão.

A depender do modo como cada *quiz* for criado, será creditada uma pontuação por jogo. Além disso, será creditada uma pontuação referente à criação de cada *quiz* e à adição de perguntas.

1.2.2 Objetivos Específicos

Os objetivos específicos deste trabalho são enumerados a seguir:

- i. Análise das tecnologias existentes para implementação dos aplicativos propostos;
- ii. Modelagem de uma interface gráfica de fácil utilização, para sistemas *web* e *mobile*;
- iii. Implementação do sistema *web* e hospedagem em um serviço de computação em nuvem utilizando *Platform as a Service* (PaaS);
- iv. Apresentar o *Ruby on Rails* como *framework web* que oferece alta produtividade no desenvolvimento, fácil manutenção e rápida interação com aplicativos móveis;
- v. Apresentar a implementação dos clientes móveis da aplicação em *iOS* e *Android*;
- vi. Comparar algumas funcionalidades e implementações do desenvolvimento para *iOS* e *Android*;
- vii. Fornecer documentação de referência para projetos de arquitetura similar.

1.3 Metodologia utilizada

Primeiramente, uma revisão bibliográfica referente às áreas de computação em nuvem, *web services* e computação móvel foi realizada com o intuito de familiarização com o estado da arte (no que concerne às tecnologias atuais que seguem esse modelo). Em seguida, deu-se início a uma escolha de ferramentas e tecnologias que seriam utilizadas para a implementação do *software*. Após a escolha, a codificação da parte *mobile* em *iOS* e *web* em *Rails* foi feita, sendo implementada posteriormente em *Android*. Nas duas últimas etapas posteriores, testes e melhorias se deram de forma concomitante. A última etapa é reservada a melhorias e modificações demandadas por usuários, conforme o produto for sendo utilizado quando lançado no mercado.

1.4 Estrutura da monografia

Esta monografia está organizada em seis capítulos, incluindo-se a introdução aqui apresentada.

O capítulo 2 aborda os conceitos gerais que serão utilizados por todo o trabalho, como computação em nuvem, *web service* e tecnologias correlatas, computação móvel e os trabalhos relacionados que constituíram o embasamento teórico deste trabalho.

No capítulo 3, a aplicação desenvolvida é abordada através da apresentação das ferramentas utilizadas para sua criação e descrição dos requisitos, funcionais e não-funcionais, e das camadas que compõem a arquitetura da aplicação.

O capítulo 4 apresenta a tecnologia *web Ruby on Rails* e mostra como algumas funcionalidades foram implementadas.

O capítulo 5 traça um comparativo entre as funcionalidades que foram implementadas para as plataformas *iOS* e *Android*.

Por fim, o capítulo 6 apresenta considerações finais acerca do trabalho realizado.

Fundamentação teórica e ferramentas

2.1 Serviços PaaS

2.1.1 Definição

Computação em nuvem refere-se a capacidade de processamento e armazenamento na internet. Neste paradigma, é necessário apenas um cliente, com componentes de entrada e saída de dados, para realizar o acesso aos recursos de um computador, através de serviços. Um dos serviços definidos é o [PaaS](#).

De acordo com o *National Institute of Standards and Technology* ([NIST](#)), o [PaaS](#) é “a capacidade fornecida ao consumidor de publicar aplicações usando linguagem de programação, bibliotecas e serviços suportados pelo provedor”. Com o fornecimento de tal serviço, o consumidor não precisa se preocupar com o controle de certos serviços de infraestrutura, como a rede, servidores, sistemas operacionais, armazenamento, ou seja, cria-se uma camada de abstração de serviços de infraestrutura.

Os serviços [PaaS](#) possuem as seguintes características [2]:

- ▶ Serviços para desenvolver, testar, publicar, hospedar e manter aplicações de forma integrada;
- ▶ Arquitetura *multitenancy*, onde uma única instância do *software* executa em um servidor, servindo à múltiplas organizações clientes (*tenants*).
- ▶ Construção garantindo escalabilidade, incluindo balanceamento de carga (*load balancing*) e replicação de dados para recuperação de falhas (*failover*);
- ▶ Integração com *web services* e banco de dados através de padrões comuns;

- ▶ Suporte para desenvolvimento em equipes, podendo ter ferramentas de planejamento de projetos e de comunicação;
- ▶ Ferramentas para gerenciamento dos custos.

Assim, sistemas [PaaS](#) são úteis para desenvolvedores individuais e *startups*¹, pois fornecem facilidade de publicação sem os custos e complexidades de *hardwares* e *softwares* inerentes a uma aplicação *web* comum [3].

Os principais provedores de serviços [PaaS](#) são: Google App Engine, Windows Azure e Heroku.

2.1.2 Heroku

O *Heroku* é uma plataforma *cloud* de serviços [PaaS](#) montado sobre o *Amazon EC2*, existente desde junho de 2007. Possui suporte para as seguintes linguagens: Ruby, Java, Node.JS, Scala, Clojure, Python e PHP. Ao contrário de seus concorrentes Google App Engine e Windows Azure, o Heroku fornece suporte a Ruby nativo², tendo sido por isso escolhido para este projeto.

Inicialmente, o *Heroku* foi desenvolvido com suporte exclusivo para a linguagem Ruby. Em julho de 2011, Matz Matsumoto, criador do Ruby, entrou para a empresa como arquiteto-chefe e, nesse mesmo mês, passou a dar suporte também para Node.js e Clojure. Em setembro de 2011, a rede social *Facebook* fez uma parceria com o *Heroku* a fim de facilitar a publicação de aplicativos para sua própria plataforma [4]. Em poucos passos, é possível criar uma aplicação no *Facebook* e no *Heroku*, simultaneamente.

O *Heroku* usa uma unidade de máquina virtual chamada “Dyno” com quatro núcleos e até 512MB de RAM, sobre o sistema operacional Ubuntu.

2.2 REST

O [REST](#) é uma arquitetura de engenharia de *software* para sistemas hipermídia distribuídos sobre a *world wide web*. O termo foi definido em uma tese de doutorado por Roy Fielding [5].

De modo geral, o [REST](#) é uma interface de comunicação onde há um provedor de serviços e um consumidor. Tal interface pode ser descrita utilizando *Extensible Markup Language* ([XML](#)), *Hypertext Transfer Protocol* ([HTTP](#)), *YAML Ain't Markup Language*

¹Termo utilizado para designar modelo de negócios repetível e escalável, em um ambiente de extrema incerteza. Normalmente, projetos ou empresas *startups* estão associadas a áreas de tecnologia.

²O Google App Engine fornece suporte ao JRuby, uma implementação Ruby sobre a máquina virtual do Java.

([YAML](#)), *JavaScript Object Notation* ([JSON](#)) ou até mesmo texto puro, de modo a não utilizar trocas de mensagens complexas como o [SOAP](#).

O [REST](#) possui alguns princípios, a saber:

- ▶ Modelo provedor/consumidor *stateless* (sem estado): cada mensagem [HTTP](#) trocada possui todas as informações necessárias para a comunicação, ou seja, nenhuma das partes necessita gravar estado da comunicação. Em sistemas *web*, é comum o uso de *cookies*³ para manter o estado da sessão entre requisições sucessivas. Já em sistemas *mobile*, é comum a utilização de um *token* de autenticação com a mesma finalidade.
- ▶ Operações [HTTP](#): de modo a diminuir o tráfego de dados, são utilizados métodos [HTTP](#), definidos no RFC 2616 [6], para acessar os recursos de informação. As operações mais utilizadas são o GET, PUT, POST e DELETE. Em sistemas [REST](#), é comum combinar tais métodos com operações de *Create, read, update e delete* ([CRUD](#)), que faz persistência de dados em um determinado recurso ou entidade.
- ▶ Identificação de recursos: as URLs identificam cada uma das entidades e seus elementos, ficando a cargo da operação [HTTP](#) definir a ação a ser feita com cada um dos recursos ou elementos.
- ▶ Uso de hipermídia: as trocas de mensagem de comunicação são feitas utilizando, no corpo da mensagem [HTTP](#), uma linguagem de marcação, conforme já citado anteriormente. Porém, não há uma restrição geral quanto ao uso, podendo ser usada linguagens próprias (texto puro).

2.2.1 RESTful

Em uma arquitetura julgada como *RESTful*, o método desejado é informado dentro do método [HTTP](#), contido no *header* do mesmo. Além disso, o escopo da informação é colocado na própria *Uniform Resource Locator* ([URL](#)). Por definição, uma aplicação deixa de ser *RESTful* caso o método [HTTP](#) não combine com o método da informação, ou seja, com a funcionalidade esperada para aquela estrutura de dados [7].

2.3 JSON

2.3.1 Introdução

[JSON](#) é um padrão aberto de texto para representar estruturas de dados, de forma inteligível para humanos. Sua origem é a linguagem JavaScript, e seu formato está descrito

³ *Cookie* é um grupo de dados trocado entre o navegador e o servidor, colocado num arquivo de texto criado no computador do cliente. Sua função principal é a de manter a persistência de sessões [HTTP](#).

no RFC 4627 [8].

2.3.2 Definição

O **JSON** é um dos formatos mais usados na serialização e transmissão de dados estruturados pela internet, ao lado do **XML** e **YAML**, sendo muito usado em sistemas orientados a serviço e/ou *web services*. Muitas linguagens e *frameworks*, como o “Foundation” (*iOS*) e o “Android”, dão suporte para esse padrão, através de *parsers*⁴ para construção e consumo.

De acordo com [7], “é muito mais fácil para um *browser* lidar com uma estrutura JavaScript oriunda de uma estrutura **JSON** do que a partir de um documento **XML**”. Ainda de acordo com a fonte, cada navegador oferece uma interface JavaScript diferente para seus *parsers* **XML**, enquanto um objeto **JSON**, que por definição é um objeto JavaScript, será interpretado da mesma maneira em qualquer interpretador JavaScript. O **JSON** é uma alternativa mais leve para serialização de dados do que o **XML**, definido pelo “XML Schema”.

2.3.3 comparação com XML

JSON e **XML** são dois formatos de manipulação de informações que podem ser usados com o mesmo objetivo, mas possuem implementações e aplicações distintas.

No artigo *Comparison of json and xml data interchange formats: A case study* [9], é feito um estudo considerando a hipótese de que não há diferença em relação ao tempo de transmissão e os recursos utilizados entre **JSON** e **XML**. A fim de realizar o estudo, foi criado um ambiente operacional consistindo de uma aplicação cliente/servidor em Java, onde o servidor escuta uma porta e o cliente se conecta.

No referido teste, foram utilizadas as seguintes métricas: número de objetos enviados, tempo total para enviar o número de objetos, tempo médio de transmissão, uso da CPU pelo usuário, uso da CPU pelo sistema e o uso de memória.

De acordo com a conclusão do artigo, codificação **JSON** é, em geral, mais rápida e consome menos recursos que a codificação **XML**, o que nega a hipótese de igualdade de escolha entre as duas tecnologias. Ou seja, em um ambiente onde é necessário um tempo de resposta rápido e os recursos são limitados, como sistemas móveis, é preferível utilizar **JSON**.

⁴Em português, *parsers* são analisadores sintáticos que analisam uma sequência de entrada para determinar sua estrutura gramatical segundo uma determinada gramática formal.

2.3.4 Estrutura

De acordo com W3resource [10], o [JSON](#) suporta duas grandes estruturas de informação: coleção de pares chave/valor e listas ordenadas de valores. Ambas estruturas são também suportadas pela maioria das linguagens de programação modernas, o que reforça a ideia de ser uma boa escolha de linguagem para transmissão de informações.

O [JSON](#) possui alguns tipos de dados, a saber:

- ▶ **Objetos:** um objeto começa e termina com “{” e “}”, contendo um número de pares chave/valor. A separação entre uma chave e um valor é feita com o caractere “:”, e a separação entre pares é feita com “,”. O valor de um par pode ser qualquer estrutura [JSON](#).
- ▶ **Arrays:** Um array começa e termina com “[” e “]”. Entre eles, são adicionados certo número de valores, separados por “,”.
- ▶ **Valores:** os valores podem ser string, número, objeto, array, valor booleano ou nulo.

Exemplo de código [JSON](#):

```
{
  "firstName": "Bidhan",
  "lastName": "Chatterjee",
  "age": 40,
  "address": {
    "streetAddress": "144 J B Hazra Road",
    "city": "Burdwan",
    "state": "Paschimbanga",
    "postalCode": "713102"
  },
  "phoneNumber": [
    {
      "type": "personal",
      "number": "09832209761"
    },
    {
      "type": "fax",
      "number": "91-342-2567692"
    }
  ]
}
```

2.4 Ruby on Rails

2.4.1 Ruby

Ruby é uma linguagem de programação orientada a objetos, com tipagem forte e dinâmica, criada por Yukihiro Matsumoto (Matz) e lançada em 1995 [11].

Segundo a Caelum [12], uma de suas principais características é sua expressividade, ou seja, a facilidade de ser lida e entendida, o que facilitaria o desenvolvimento de sistemas escritos por ela.

O livro *Learning Rails 3* [13] lista algumas das características mais importantes do Ruby, a saber:

- ▶ É uma linguagem interpretada. Em consequência, um sistema em Ruby pode se tornar um pouco mais lento, porém, é notável o ganho em flexibilidade;
- ▶ Possui uma sintaxe de linguagem flexível, fazendo com que a curva de aprendizado seja menor em relação a outras linguagens. Um exemplo clássico é a não-necessidade (opcional) de escrever parênteses ao redor dos parâmetros de um método. Entretanto, podem surgir erros de difícil depuração por não colocar parênteses em algumas situações ambíguas;
- ▶ Possui uma tipagem dinâmica, ou seja, não é necessário especificar o tipo de informação que será guardado em cada variável. Isso torna a abordagem bem mais flexível, tornando as operações dependentes do próprio contexto. Entretanto, isso também permite a ocorrência de comportamentos inesperados e de difícil depuração;
- ▶ Suporte a blocos e *closures*⁵.

Atualmente, Ruby encontra-se entre as linguagens de programação mais populares, ocupando a posição 11º no índice Tiobe⁶, liderados pelas linguagens C e Java. Grande parte desse sucesso deve-se ao *framework Rails*, implementado como solução *web* utilizando Ruby.

2.4.2 Rails

O *Ruby on Rails*, também chamado *Rails* ou *RoR*, é um *framework* de desenvolvimento *web* de código aberto que tem como premissa aumentar a velocidade e a

⁵Em uma *closure*, uma função é declarada dentro do corpo de outra, e a função interior pode referenciar variáveis locais da função exterior.

⁶O índice TIOBE mede, mensalmente, a popularidade de uma determinada linguagem de programação, baseado no número de engenheiros qualificados, cursos, vendedores e buscas nos principais motores de busca. Pode ser encontrado em <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>

facilidade no desenvolvimento de aplicações *web* orientados a banco de dados. Foi lançado oficialmente em julho de 2004 pelo seu criador David H. Hansson, estando atualmente na versão 3.2.11, com a 4ª versão em desenvolvimento [14].

O *Rails* é um *framework full-stack*, ou em português, arcabouço de desenvolvimento. Isso significa que, com ele, é possível desenvolver a aplicação por completo, desde os *layouts* das páginas, à manutenção do banco de dados. Além disso, o *Rails* enfatiza o uso de alguns padrões de engenharia de *software*, a saber:

- ▶ *Active record*: padrão de projeto para armazenamento de dados em banco de dados relacionais. A interface de um certo objeto deve incluir funções de **CRUD**, como inserir, atualizar, apagar e algumas funções de consulta. Cada tabela de um banco de dados é “envelopada” em uma classe, sendo cada instância dessa classe um registro (tupla) único na tabela. Esse conceito foi definido por Martin Fowler em [15].
- ▶ Convenção sobre configuração: modelo de desenvolvimento de *software* que busca diminuir o número de decisões que os desenvolvedores precisam tomar, ou seja, o desenvolvedor não precisa definir aspectos convencionais da aplicação. Em *Rails*, é fácil perceber esse padrão na escolha dos nomes das tabelas: se um modelo chama-se “Usuario”, a tabela correspondente se chamará “Usuarios” e, se existir uma relação entre usuários e contas (m:m), a nova tabela será denominada, automaticamente, *usuarios_contas*.
- ▶ *Don't repeat yourself* (**DRY**): O objetivo principal é reduzir a repetição de informação de qualquer tipo. Esse conceito incentiva o bom uso da reutilização de código, que é também uma das principais vantagens da orientação a objetos.
- ▶ *Model View Controller* (**MVC**): esse padrão de arquitetura de *software* separa a aplicação em três camadas: uma contendo a lógica da aplicação e regra de negócios, chamada *model*; uma contendo a entrada e saída de dados com o usuário, chamada *view*; uma interligando ambas, de maneira a manipular dados da *view* para o *model* entender e vice-versa, chamada *controller*. O principal objetivo dessa arquitetura é a reusabilidade de código e a separação de conceitos [16].

De acordo com a apostila da Caelum [12], a estrutura sobre a qual o *Rails* é feito permite que as funcionalidades de um sistema possam ser implementadas de maneira incremental, por conta dos padrões e conceitos supracitados. Por consequência, isso tornaria o Rails uma boa escolha para projetos e empresas que adotam metodologias ágeis no desenvolvimento da aplicação.

O Ruby é uma linguagem interpretada. Antes de se tornar popular, existia apenas um interpretador disponível, escrito em C pelo próprio criador da linguagem. Hoje em

dia, o interpretador mais conhecido é o 1.9 ou *Yet Another Ruby VM* ([YARV](#)), para a versão mais atualizada e estável (Ruby 1.9.3.).

Existem outros interpretadores Ruby famosos, como:

- ▶ JRuby: implementação alternativa que permite usar a *Java Virtual Machine* ([JVM](#)) do Java para interpretar código Ruby. Uma de suas principais vantagens é a interoperabilidade com código Java existente, além de aproveitar as vantagens já maduras do java: *garbage collector*⁷, *threads* nativas, etc.
- ▶ IronRuby: Implementação .Net da linguagem, mantido pela própria Microsoft.
- ▶ Rubinius: Traz idéias de máquinas virtuais do SmallTalk e é implementada em C/C++.

2.5 Smartphones

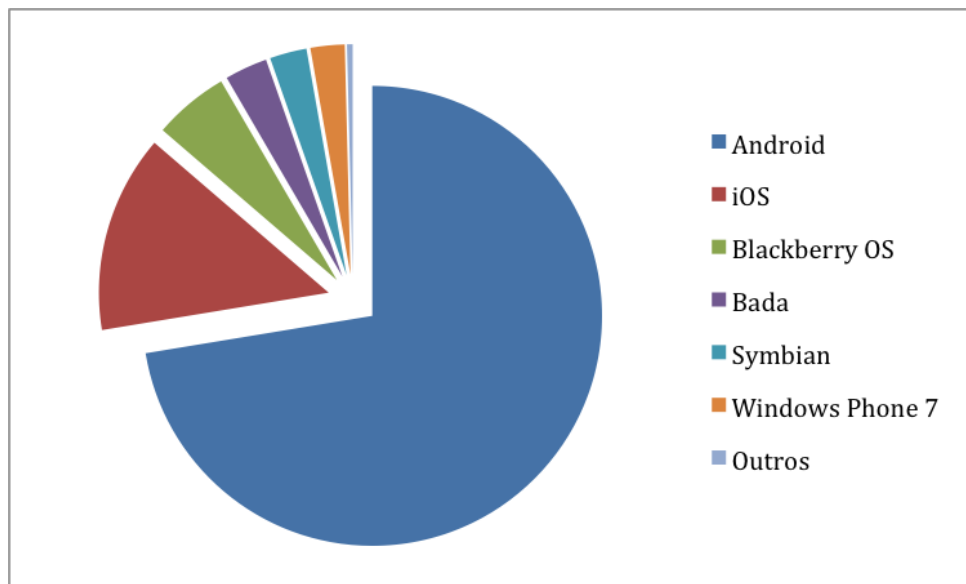
O mercado de *smartphones* tem crescido vertiginosamente nos últimos anos. Estima-se que no início de 2012 o número de celulares inteligentes tenha atingido a marca de 1 bilhão de unidades vendidas e, segundo projeções, esse número deve dobrar em 2015 [[17](#)].

Embora o número de *smartphones* seja expressivo, ele ainda é pequeno se comparado ao número de pessoas que possuem um aparelho celular. Isso significa que ainda há muito espaço para crescimento, em particular em mercados emergentes como a China, Índia e Brasil.

Usualmente, um *smartphone* possui alguns recursos de ponta, como câmera, bom reprodutor de mídia, bom processamento gráfico para jogos, bluetooth, GPS, acesso a internet via wi-fi e 3G/4G, *Near Field Communication* ([NFC](#)), um bom sistema operacional, entre outros. Atualmente, os sistemas operacionais mais populares para *smartphones* são: *Android*, *iOS* (Apple), *Blackberry OS* (RIM), *Bada* (Samsung), *Symbian* e *Windows Phone 7* (Microsoft). A distribuição do mercado, no Q3⁸ de 2012, pode ser vista no seguinte gráfico:

⁷*garbage collector* é um processo usado para a automação do gerenciamento de memória

⁸Q3 faz referência ao trimestre entre julho e setembro. As denominações Q1, Q2, Q3 e Q4, para os trimestres do ano, são comuns em relatórios financeiros.

Figura 2.1: *Marketshare* Q3 2012

Fonte: NewWin [18]

Notadamente, o *Android* e o *iOS* são as plataformas mais populares. O grande diferencial entre o *marketshare*⁹ desses dois sistemas é o segmento de mercado: enquanto o *Android* atua em todos os segmentos, desde celulares *low-end* aos celulares de ponta, o *iOS* atua somente com celulares de ponta, chamados *high-end*. Outro fato a considerar é que, nesse gráfico, não são contemplados outros dispositivos, como reprodutores de música e *tablets*.

2.6 iOS

2.6.1 Visão Geral

O *iOS* é um sistema operacional para dispositivos móveis, lançado pela Apple em 2007. Inicialmente, foi desenvolvido para o iPhone, sendo posteriormente aproveitado nos dispositivos iPod Touch, iPad e Apple TV. Ele é um sistema operacional licenciado para funcionar apenas em *hardware* produzido pela Apple, otimizado para a arquitetura de processadores ARM.

Sua entrada de dados é feita de forma direta, através de multi-toques. Esses toques podem ser desde o toque, similar a um clique do mouse, até balançar o aparelho, de modo a utilizar seu acelerômetro. Todos os controles de entrada de dados são controlados pela *Graphic User Interface* (GUI) “Cocoa Touch”.

O livro “Use a cabeça - iPhone” [19] cita que o iPhone revolucionou a maneira de ver um celular: ele é, atualmente, uma plataforma de jogos, um organizador pessoal, um

⁹Termo que designa a fatia de mercado que o objeto de estudo possui em relação a seus concorrentes.

browser(navegador) completo e um telefone. Muito de seu sucesso deve-se ao êxito da loja virtual “App Store”, de mídias e aplicativos, que abriu oportunidade para desenvolvedores independentes competirem em escala mundial com grandes empresas de *software*.

2.6.2 Objective-c

A programação nativa em *iOS* utiliza uma linguagem de programação chamada Objective-C.

O Objective-C é uma linguagem de programação reflexiva e orientada a objeto, com origens no SmallTalk e no C. Foi criada no início da década de 80 por Brad Cox e Tom Love, mas somente se tornou popular quando foi licenciada pela NeXT, de Steve Jobs, em 1988. Atualmente é a principal linguagem utilizada para desenvolvimento para Mac OS X.

Como o Objective-C foi construído sobre C, qualquer código C pode ser compilado com um compilador Objective-C. Por definição, é uma camada sobre o C que aceita orientação a objetos, através de mensagens.

Em linguagens com *message parsing*, métodos não são chamados por objetos, mas sim mensagens são enviadas ao objeto. Essa diferença implica em como o código referenciado pelo método ou nome da mensagem é executado. Neste caso, o “alvo” da mensagem é resolvido em tempo de execução, com o objeto receptor interpretando a mensagem.

2.6.3 Ciclo de vida

O ciclo de vida constitui uma sequência de eventos entre o início e a finalização da aplicação. Um aplicativo *iOS* começa quando o usuário toca o ícone do mesmo na tela inicial do dispositivo. Feito isso, o sistema operacional inicia alguns procedimentos de renderização e chama a função principal (*main.m*) do aplicativo.

Uma vez iniciado, o comando da execução passa a ser do UIKit, *framework* de controle do *iOS*, que carrega a interface gráfica e lê o ciclo (*loop*) de eventos. Durante o *loop*, o UIKit delega cada evento a seu respectivo objeto e responde aos comandos emitidos pelo aplicativo. Quando o usuário realiza uma ação que causa um evento de saída, o UIKit notifica a aplicação e inicia o processo de saída.

2.7 Android

2.7.1 Visão Geral

O *Android* é a proposta da Google para ocupar o segmento de mercado de sistemas operacionais para plataformas móveis. Consiste em um sistema baseado no sistema operacional Linux, com diversas aplicações já instaladas, além de um ambiente de desenvolvimento forte e flexível.

De acordo com o *Google Android*, de Ricardo Lecheta [20], o *Android* causou um grande impacto quando foi anunciado, em especial, pelas empresas que estavam por trás de seu desenvolvimento: Google, Motorola, LG, Samsung, Sony, entre muitas outras. A esse grupo de empresas, denominado *Open Handset Alliance* (OHA), coube a padronização de uma plataforma de código aberto e livre para celulares, com o objetivo de atender a uma demanda de mercado.

Um dos pontos fortes do *Android* é seu sistema flexível: é fácil integrar aplicações nativas com a sua aplicação, ou até mesmo substituir algumas dessas aplicações nativas pela sua própria. Isso gera um grande apelo para empresas de telefonia, que podem usar dessa personalização para lançarem suas próprias versões de aparelhos *Android* personalizados.

O grande foco do *Android* é a interação entre aplicativos: agenda, mapas, contatos são facilmente alcançáveis por qualquer aplicação. Essa funcionalidade é realizada por um recurso chamado *intent*.

Outro ponto forte do *Android* é que seu sistema operacional é baseado no Linux (baseado no kernel 2.6), ou seja, ele mesmo se encarrega de gerenciar a memória em uso e os processos. Isso faz com que seja possível rodar mais de uma aplicação (genuinamente) ao mesmo tempo, fazendo com que outros aplicativos rodem em segundo plano durante outros serviços, como ao atender um telefonema ou acessar a internet.

O que é dito como vantagem também é apontado, fatalmente, como um problema: uma vez que aplicativos podem rodar em segundo plano, aplicativos maliciosos também podem ser rodados em segundo plano. Durante muito tempo, aplicativos desse tipo podiam ser encontrados para download no Google Play, havendo hoje uma melhor seleção dos aplicativos que de fato estão sendo disponibilizados na loja.

2.7.2 Java

A linguagem nativa de programação para *Android* é o Java, utilizando o *framework Android* criado pela OHA.

O Java é uma linguagem de propósito geral, concorrente e orientada a objetos. Sua primeira versão foi lançada em 1995 pela Sun Microsystems e, atualmente, encontra-se em sua sétima versão, sendo mantida pela Oracle.

Atualmente, é uma das linguagens de programação mais populares do mundo, ocupando o 2º lugar no índice TIOBE. Devido a isso, há um grande número de desenvolvedores que possuem o pré-requisito básico para iniciar programação para *Android*: o domínio da linguagem.

O grande sucesso do Java é normalmente creditado a sua capacidade de funcionar nos mais diversos ambientes, desde micro-sistemas como cartões de crédito a grandes plataformas *web*. Isso é devido à implementação de sua máquina virtual, que é capaz de rodar nas mais diversas plataformas.

2.7.3 Ciclo de vida

O ciclo de vida de uma aplicação *Android* é controlado por uma *Activity*, a qual também gerencia a interface com o usuário, recebe requisições, realiza o tratamento e processa.

Toda *Activity* possui os seguintes métodos de controle, a saber:

- ▶ `onCreate()`: é o primeiro método a ser executado em uma *Activity*. Usualmente é o método responsável por carregar os layouts XML e inicializar atributos de classe e outros serviços.
- ▶ `onStart()`: é chamado imediatamente chamado após o `onCreate()`. Diferentemente deste, é chamado toda vez que a *Activity* volta a ter foco após um período em *background*.
- ▶ `onResume()`: Assim como o `onStart()`, é chamado no início da *Activity* e, também, quando a mesma volta a ter foco. A diferença entre ambos é que o `onStart()` só é invocado quando a *Activity* não está mais visível, enquanto o `onResume()` é chamado toda vez que retorna o foco.
- ▶ `onPause()`: é a primeira função a ser chamada quando a *Activity* perde o foco.
- ▶ `onStop()`: é chamado quando uma *Activity* é substituída por outra *Activity*.
- ▶ `onDestroy()`: é o último método a ser executado. Quando o `onDestroy()` é executado, a *Activity* é considerada “morta” e fica pronta para ser removida pelo *Garbage Collector*.
- ▶ `onRestart()`: Chamado quando a *Activity* sai do estado de “stop”; após sua execução, é chamado o método `onStart()`.

2.8 Resumo do Capítulo

Neste capítulo foram abordadas os três principais paradigmas necessários para o desenvolvimento deste projeto: computação em nuvem, *web services* e computação móvel. Tais paradigmas foram aprofundados em um nível o qual fornecessem ao leitor os pré-requisitos para a compreensão do restante do projeto, o qual será apresentado nos capítulos posteriores.

Especificação da aplicação

3.1 Visão geral

Com o intuito de demonstrar a aplicação em estudo com as áreas apresentadas nos capítulos 1 e 2, foi idealizado um aplicativo com módulos *Web* e *Mobile*. Um *quiz* é uma modalidade de jogo onde os jogadores têm por objetivo responder determinadas questões corretamente. Usualmente, *quizzes* são usados em educação e entretenimento para medir grau de conhecimento e habilidades, como pensamento rápido e associação mental. Em modalidades mais competitivas, os *quizzes* podem ter pontuações e, para competições em grupo, pode ter um vencedor (o participante com maior pontuação, por exemplo).

O sistema é formado por dois grandes módulos:

- ▶ Sistema *web*: nessa parte da aplicação, usuários podem gerenciar suas contas (metadados) e criar *quizzes*. No momento da criação, tal usuário é denominado “criador”, e recebe uma pontuação por participação nesse processo. Nesse sistema, também é possível consultar os resultados dos jogos (ou partida) realizada sobre cada um dos *quizzes*;
- ▶ Sistema *mobile*: nessa parte da aplicação, que está sobre as plataformas *Android* e *iOS* de forma nativa, o usuário (agora denominado “jogador”), pode baixar os *quizzes* de seu interesse para seu dispositivo e jogar, enviando seu resultado no final para ser consultado posteriormente por um criador.

Tal sistema pode ser usado com duas finalidades básicas: entreter e realizar pesquisas. Na primeira forma, perguntas de conhecimento são criadas e jogadas casualmente, com o intuito de acertar uma pergunta. Na segunda forma, pode ser usada como uma enquete, onde o objetivo não é acertar uma determinada resposta, mas sim informar qual das opções corresponde a uma verdade do usuário.

3.2 Requisitos

3.2.1 Requisitos funcionais

Os requisitos funcionais definem ações que um sistema ou componente devem ser capazes de executar, sem levar em consideração restrições físicas. Para a aplicação iQuizzer, temos os seguintes requisitos:

- ▶ Gerenciamento de *quiz*: a aplicação deve fornecer meios para que o dono do *quiz* possa inserir, modificar ou apagar dados relativos ao *quiz*, as perguntas pertencentes a ele e as respostas pertencentes a cada pergunta;
- ▶ Sistema de pontos: a aplicação deve adicionar pontos a cada *quiz* ou pergunta inseridos por um criador de *quiz*. Além disso, deve adicionar pontos relativos a cada jogo, desde que o modo de jogo do *quiz* executado permita acumular pontos;
- ▶ Gerenciamento de resultados: a aplicação deve permitir ao criador visualizar as respostas marcadas (acertos e erros, a depender do modo de jogo), de cada jogador, ao seu *quiz*;
- ▶ Gerenciamento de conta: a aplicação deve fornecer meios para que o usuário possa criar e apagar sua conta. Além disso, deve permitir ao usuário modificar seus dados cadastrais.

3.2.2 Requisitos não-funcionais

Os requisitos não-funcionais especificam alguns fatores relacionados ao uso da aplicação, como desempenho, usabilidade, confiabilidade, entre outros. Esses requisitos podem constituir restrições aos requisitos funcionais, pois são características mínimas de um software de qualidade, ficando a cargo do desenvolvedor optar ou não por atender esses requisitos. Para a aplicação iQuizzer, temos os seguintes requisitos:

- ▶ Interatividade: a aplicação deve exigir interação do usuário, de maneira rápida e intuitiva. Atividades básicas não podem demandar muitas interações;
- ▶ Confiabilidade: a aplicação não pode apresentar erros durante a execução. O sistema móvel deve prever situações de perda de conexão e situações de estouro de memória;
- ▶ Integridade e privacidade dos dados: toda a informação trocada entre o sistema móvel e o sistema *web* deve ser mantidas de maneira íntegra. Além disso, os dados do sistema *web* só podem ser visualizados caso haja autorização para isso, ou seja, devem existir autenticação e tokens de autenticação.

3.3 Casos de uso

O diagrama de casos de uso fornece um modo de descrever o sistema e suas interações com o mundo exterior, representando uma visão de alto nível da funcionalidade do sistema mediante uma requisição do usuário.

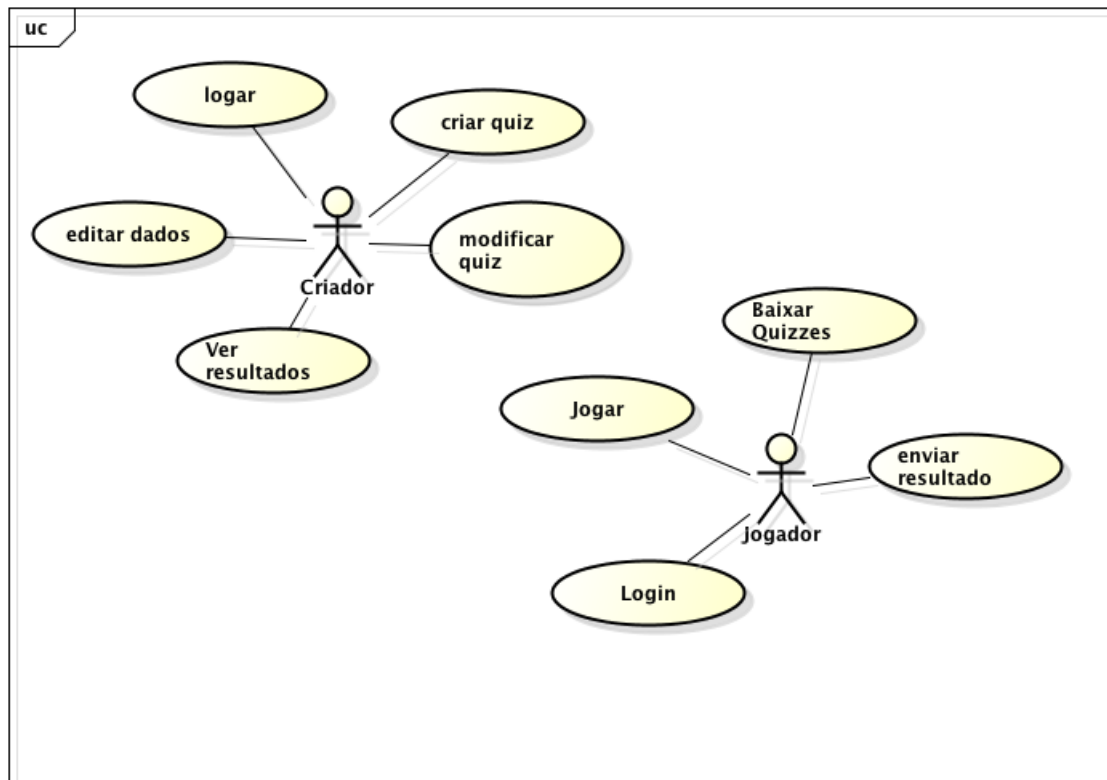


Figura 3.1: Diagrama de casos de uso para “criador” e “jogador”

3.4 Diagrama de classes

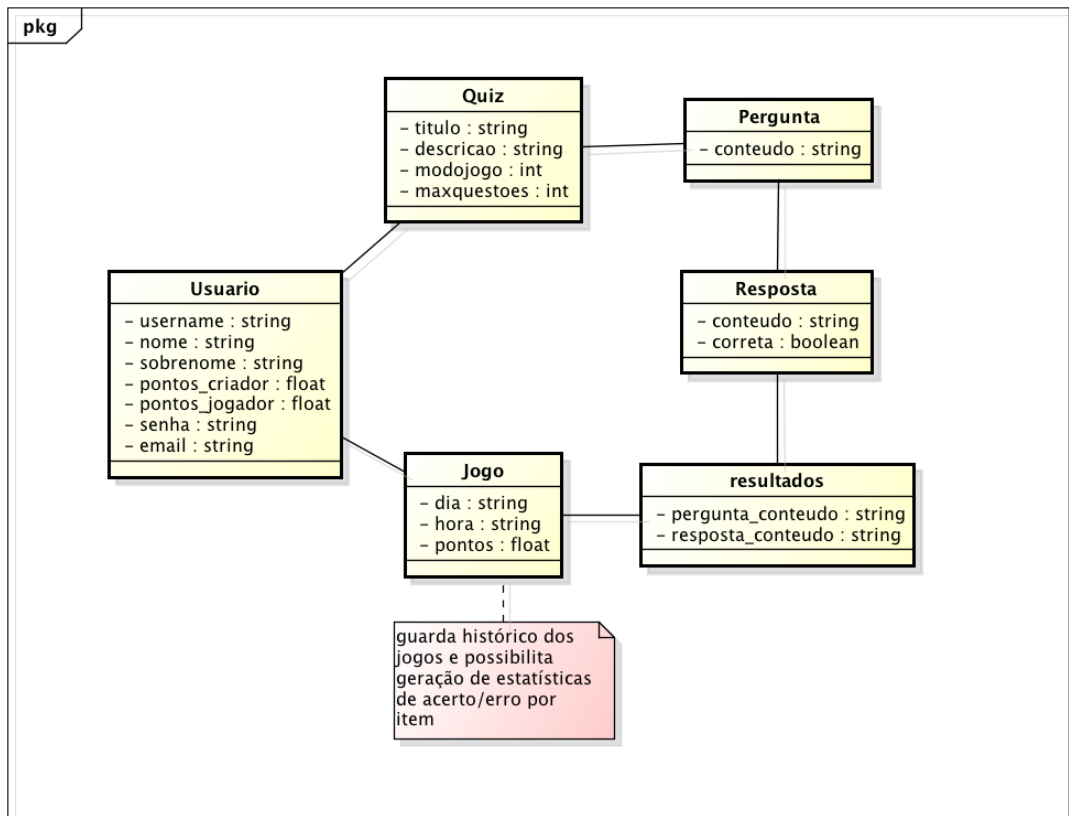
Diagrama de classes é uma representação da estrutura e relações das classes que servem de modelo para objetos. Com ela, é possível ter uma visão rápida das classes do sistema, de modo a ajudar na implementação do modelo (M do MVC) da aplicação. Além disso, é a base para construção dos diagramas de comunicação, sequência e estados.

Nesse diagrama, estão representadas as tabelas básicas para o funcionamento da aplicação iQuizzer. Existem três entidades principais, a saber:

- ▶ **Usuário:** é a classe que representa o jogador e o criador do *quiz*. Além das informações básicas, como nome e email, estão guardadas informações para login (nome de usuário, senha) e a pontuação (de criador e jogador).
- ▶ **Quiz:** é a classe que representa um *quiz*. Hierarquicamente, um *quiz* possui um número ilimitado de perguntas únicas, e uma pergunta possui um número ilimitado de respostas únicas. Cada *quiz* possui um modo de jogo, que indica se é um *quiz*

de perguntas arbitrárias, em sequência, com tempo, etc., e um número máximo de perguntas a ser jogado por partida.

- **Jogo:** essa classe representa cada uma das partidas (jogos) que foram realizadas por cada jogador no sistema móvel. Cada jogo possui uma quantidade de resultados, correspondente ao número máximo de perguntas do *quiz* jogado.



powered by Astah

Figura 3.2: Diagrama de entidades

Implementação *web*

Conforme citado no capítulo 2, utilizaremos o *framework web* chamado Ruby on Rails, implementado sobre a linguagem Rails.

4.1 Ambientação

Para a configuração do ambiente Ruby on Rails, foram utilizadas as soluções de [21], que implementou facilitadores para instalação em ambientes Windows e OS X. Porém, os aplicativos necessários podem ser instalados separadamente. Os necessários para a nossa aplicação foram:

- ▶ Ruby 1.9.3 - Interpretador da linguagem Ruby
- ▶ Rails 3.2.11 - Pacote contendo todo o *framework* Rails utilizado
- ▶ Git 1.7.10 - Controlador de versão Git. Utilizamos para o controle da versão junto ao GitHub e para fazer o *deploy*¹ no Heroku
- ▶ RVM1.16.17 - Gerenciador de versões Ruby
- ▶ Bundler 1.2.1 - Controle de dependências (Gems²)

Uma das decisões mais importantes no processo de manufatura de uma aplicação é a escolha da ferramenta ideal de desenvolvimento, que forneça agilidade e os recursos necessários para o desenvolvedor.

Por ser uma aplicação em Ruby, onde os arquivos de código e de configuração são baseados em texto puro, o *Ruby on Rails* não exige nenhuma ferramenta avançada de criação. Em outras palavras, utilizar um editor de texto simples ou uma *Integrated Development Environment* (IDE) é uma decisão pessoal.

¹ *Deploy* ou publicação é a instalação da aplicação em um servidor de aplicações.

² Gems são pacotes que contém toda informação de arquivos a serem instalados.

As IDE's mais comuns para se trabalhar com *Rails* são:

- ▶ RubyMine - IntelliJ IDEA
- ▶ Aptana Studio 3 - antes conhecida como RadRails, *plugin* para Eclipse
- ▶ Ruby in Steel- Visual Studio
- ▶ NetBeans - (até a versão 6.9)

De acordo com [12], a maioria dos desenvolvedores Ruby on Rails não utiliza nenhuma *IDE*, apenas um bom editor de texto e um terminal (ou *prompt*) de comando aberto. Algumas ferramentas de texto boas para esse propósito, junto aos sistemas operacionais que suportam, são as seguintes:

- ▶ TextMate - Mac OS X
- ▶ Sublime Text - Mac OS X, Linux, Windows
- ▶ Gedit - Mac OS X, Linux
- ▶ Notepad++ - Windows
- ▶ VI/Vim - Mac OS X, Linux, Windows

Nesse projeto foi utilizado o TextMate, no ambiente Mac OS X.

4.2 Criação de aplicação

Uma vez que o *Rails* tenha sido instalado corretamente e, automaticamente, definido como variável de ambiente no terminal, pode-se executar o comando “rails” para criação de aplicação e módulos. Pode-se, também, verificar a versão do rails através do comando:

```
$rails --version
```

A versão utilizada nesse projeto foi a 3.2.11, a última disponibilizada quando este texto foi escrito. A fim de criar a aplicação, utilizando o *Sistema de Gerenciamento de Banco de Dados (SGBD)* PostgreSQL, foi executado o seguinte comando:

```
$rails new iQuizzer -d postgresql
```

Ao executar a última linha, o *Rails* criou um diretório com alguns arquivos, que serão vistos na próxima seção.

4.3 Estrutura

Um projeto *Rails* tem uma estrutura básica composta das seguintes pastas:

- ▶ App: contém os arquivos específicos da aplicação. Conforme citado no capítulo 2, o *Rails* utiliza o padrão MVC, e dentro desse diretório a divisão é feita através dos sub-diretórios *model*, *view* e *controller*;
- ▶ Config: configurações diversas da aplicação;
- ▶ db: contém as migrações (alterações no banco de dados durante o processo de desenvolvimento), além de alguns outros arquivos relacionados ao banco de dados;
- ▶ doc: documentação do sistema;
- ▶ lib: bibliotecas;
- ▶ log: algumas informações de *log*;
- ▶ public: nessa pasta estão contidos todos os arquivos públicos que serão servidos pela *Web*;
- ▶ test: utilizado para testar a aplicação, normalmente quando se usa *Test Driven Development* (TDD)³. Neste projeto, não foi utilizado;
- ▶ Tmp: arquivos temporários de sessão e *cache*;
- ▶ Vendor: projetos dependentes (terceiros).

4.4 Banco de dados

4.4.1 PostgreSQL

Banco de dados são coleções de informações que se relacionam de forma que crie um sentido. Seu objetivo principal é representar abstratamente uma parte do mundo real, conhecida como “Universo de Discurso” [22].

Na aplicação, foi utilizado um SGBD relacional chamado PostgreSQL. Lançado em 1995, está atualmente na versão 9.1.4, sobre a licença BSD⁴.

A opção pela escolha do PostgreSQL deve-se ao bom suporte do Heroku a este SGBD. Entretanto, outros motivos poderiam ser colocados em prol dessa decisão, a saber:

- ▶ Independência de plataforma - roda nos principais sistemas operacionais
- ▶ Leve, podendo ser rodado em *desktops* convencionais

³Desenvolvimento orientado a testes é uma técnica de desenvolvimento de *software* onde um caso de teste é escrito para cada funcionalidade a ser implementada.

⁴BSD foi criada pela Universidade de Berkeley e é conhecida por ser uma licença de pouca restrição quando comparada a GNU. Ela permite que o *software* distribuído sobre a licença seja incorporado a produtos proprietários.

- Instalação simples
- Gratuito

4.4.2 Modelos

Conforme o diagrama de classes citado na seção 3.4, a aplicação possui algumas entidades, onde entidade é representada por uma tabela presente no banco de dados e um *active record* contido na pasta `app/models`.

O Rails possui uma forma bem simples de criar modelos, através de um comando de criação. Seja, por exemplo, a entidade “Usuário”, tal que seus campos e tipos de campos sejam passados como parâmetros de um comando de criação de modelos:

```
$rails generate model Usuario nome:string sobrenome:string  
apelido:string pontos_criador:float pontos_jogador:float  
username:string senha:string
```

A execução desse comando gera os seguintes arquivos:

- Um arquivo de migração de tabela, que cria a tabela usuário no banco de dados;
- Uma classe chamada `usuario.rb`, dentro de `app/models`, e estende de `ActiveRecord`;
- Arquivos de teste.

Após a criação do modelo, pode ser executado o seguinte comando:

```
$rake db:migrate
```

Esse comando tem a função de executar todas as migrações, ou seja, de alterar os dados no banco de dados. Caso seja necessária a modificação de um campo dessa tabela, como por exemplo, ao criar uma associação, pode ser criado um novo arquivo `migration` especificamente para isso.

4.4.3 Relacionamentos

Ao ser analisado o diagrama de classes da aplicação na seção 3.4, pode-se verificar que existem relacionamentos entre as entidades apresentadas. Tais entidades representam relações entre tabelas do banco de dados, as quais devem ser implementadas na camada de modelo da aplicação.

Para relacionar tais modelos, deve-se informar ao Rails o tipo de relacionamento existente. Isso é feito declarando o tipo de relacionamento entre modelos nos *active records*. Os tipos de relacionamentos tratados são:

- *Belongs_to*: é utilizado quando um modelo tem como atributo uma referência de outro modelo (em um relacionamento um para muitos ou um para um);

- ▶ *has_many*: associação contrária ao *belongs_to*; indica que um determinado modelo tem muitas instancias de outro modelo;
- ▶ *has_one*: similar ao *has_many*, porém com apenas uma instância (relacionamento um para um);
- ▶ *has_and_belongs_to_many*: associação muitos para muitos.

As declarações de relacionamento criam alguns métodos de acesso automaticamente. Observe o código abaixo, do modelo de *quiz* e do modelo de pergunta:

```
##-- classe Quiz
class Quiz < ActiveRecord::Base
  #modojogo: 1 - random, 2 - ordenate
  attr_accessible :titulo, :perguntas_attributes, :modojogo,
                 :maxquestoes, :descricao, :usuario_id

  has_many :perguntas
end

##--classe Pergunta
class Pergunta < ActiveRecord::Base
  attr_accessible :conteudo, :respostas_attributes

  belongs_to :quiz
  has_many :respostas
end
```

Nesses modelos, foi definido que uma pergunta pertence a um *quiz*, e que um *quiz* possui várias perguntas. Tais definições criam métodos de acesso automaticamente, como:

```
quiz.perguntas #retorna array de perguntas
pergunta.quiz #retorna objeto Quiz
```

O *Rails* não verifica automaticamente se a relação declarada funciona. Porém, de acordo com o princípio “convenção sobre configuração”, definido no capítulo 2, é esperado que existam chaves estrangeiras do tipo `<modelo>_id` nas tabelas que possuam relacionamentos do tipo *belongs_to*. Para tanto, devemos criá-las nas tabelas do banco, utilizando um objeto *migration*⁵.

Para criar um objeto *migration*, entre *quiz* e perguntas, utilizou-se, por exemplo, o comando *generate*:

```
rails generate migration AddColumnQuizIdPergunta
```

⁵ *Migrations* são objetos em Ruby que realizam manipulações no banco, através de código SQL.

Isso gerou um arquivo `migration` com o nome correspondente a “`< data_hora > add_column_quiz_id_pergunta`”, no diretório `/db/migrate`. Implementou-se, então, o `migration` com o campo `id`:

```
class AddColumnQuizIdPergunta < ActiveRecord::Migration
  def up
    add_column :Perguntas, :quiz_id, :integer
  end

  def down
    end
end
```

Ao executar o *script* de execução, `rake db : migrate`, será executado um SQL de criação da coluna no banco de dados. É recomendado que as alterações sofridas no banco de dados ao longo do projeto utilizem `migration`, uma vez que é possível registrar toda a sequência de alterações no banco e reproduzi-las em outros ambientes.

4.5 Rotas e REST

As rotas no Rails servem para transformar determinadas requisições [URL](#) em chamadas para controles particulares. Algumas dessas rotas, para recursos, são utilizadas para o desenvolvimento da aplicação utilizando [REST](#), em conformidade ao definido no capítulo 2.

Aplicações em *Rails* são ditas *RESTful*, associando o protocolo [HTTP](#) às operações de [CRUD](#). Na aplicação, têm-se as seguintes operações disponíveis:

- ▶ *Get*: retorna o valor ou representação do recurso - *select*;
- ▶ *Post*: criação de um novo recurso - *insert*;
- ▶ *Put*: altera o recurso - *update*;
- ▶ *Delete*: remove o recurso - *delete*.

Para acessar o recurso, definimos uma rota de recurso no arquivo `routes.rb`. Seja o modelo de *quiz*; define-se o seguinte recurso:

```
# routes.rb
resources :quizzes
```

Para esse recurso, o rails automaticamente cria sete rotas de acesso, a saber:

- ▶ GET /quizzes:controller => 'quizzes', :action => 'index'
- ▶ POST /quizzes:controller => 'quizzes', :action => 'create'

- ▶ GET /quizzes /new:controller => ' quizzes ', :action => 'new'
- ▶ GET /quizzes /:id:controller => ' quizzes ', :action => 'show'
- ▶ PUT /quizzes/:id:controller => 'quizzes ', :action => 'update'
- ▶ DELETE /quizzes /:id:controller => ' quizzes ', :action => 'destroy'
- ▶ GET /quizzes /:id/edit:controller => ' quizzes ', :action => 'edit'

Para todas as rotas criadas, o Rails também cria os *helpers* (view do [MVC](#)):

- ▶ `quizzes_path# => "/quizzes"`
- ▶ `new_quiz_path# => "/quizzes/new"`
- ▶ `edit_quiz_path(3)# => "/quizzes/3/edit"`

Os resultados dos métodos invocados pelas rotas, por padrão, são em *HyperText Markup Language* ([HTML](#)). Entretanto, existem facilitadores no *Rails* para que esse retorno seja em [XML](#) ou [JSON](#). Além disso, podemos passar no corpo de um [HTTP](#) uma entrada em [XML/JSON](#), de modo a ser interpretada pelo *controller*.

Na aplicação, toda a comunicação entre o *Web Service* (aplicação *Rails*) e o cliente (aplicação *mobile*) é feita através de [REST](#) utilizando [JSON](#). Isso é possível utilizando um *format*⁶, como no exemplo abaixo:

```
# GET /quizzes
# GET /quizzes.json
def index
  @quizzes = Quiz.all
  respond_to |format|
    format.html # index.html.erb
    format.json { render :json => @quizzes }
  end
end
```

O método `index` é responsável por retornar um *array* com todos os *quizzes*. Se a requisição vier por [HTML](#), ou seja, do próprio *site*, a resposta será em [HTML](#); caso contrário, a resposta será em [JSON](#). Dessa forma, percebemos que o modo como os dados são requisitados é que diferenciam a saída.

4.6 Autenticação

Conforme foi definido na seção [3.2.2](#), um dos requisitos não funcionais é a segurança, ou seja, o sistema deve ser capaz de garantir que os dados de cada usuário estejam

⁶Os objetos *respond_to* são relativos ao tipo de requisição [HTTP](#) que chega. Para cada tipo, uma variável *format* é definida.

protegidos.

A autenticação de um usuário busca verificar a identidade digital do mesmo. A partir da autenticação, o sistema autoriza o acesso ou não a determinados recursos do sistema.

Na aplicação, a autenticação é feita através de *tokens*. *Tokens* são chaves criptografadas que identificam uma sessão do usuário. Optou-se por disponibilizá-los no momento que o usuário realiza o *login*. Uma vez que o navegador (ou o aplicativo móvel nativo) obtenha o *token*, o acesso é garantido até que o mesmo expire.

A fim de reduzir o tempo de desenvolvimento, utilizou-se uma *gem*⁷ chamada *Devise*, para autenticação e gerência de contas. Essa *gem* possui doze módulos para gerência de conta, dos quais foram utilizados os seguintes, com as funções de:

- ▶ *Database authenticable*: encripta e armazena uma senha no banco de dados para validar a autenticidade de um usuário enquanto está logado;
- ▶ *Token Authenticable*: usuários permanecem logados enquanto um *token* autenticado for válido;
- ▶ *Registerable*: permite ao usuário criar contas;
- ▶ *Recoverable*: gera uma nova senha para o usuário e o manda instruções para alterar sua senha;
- ▶ *Rememberable*: gerencia e limpa *tokens* para lembrar o usuário de um *cookie* já salvo;
- ▶ *Trackable*: guarda informações de acesso, como o número de vezes que o usuário realizou *login*, o momento do último *login* e o endereço IP;
- ▶ *Validatable*: Valida a conta através de email/senha.

O *Devise* cria algumas telas (*helpers*) por padrão. As telas de confirmação, envio de email, gerência de contas, senha e sessão são geradas de maneira básica. As imagens seguintes mostram algumas dessas telas, com o *layout* modificado pelo *Bootstrap*, que será abordado na próxima seção.

⁷As *gems* são pacotes de arquivos Ruby com determinadas funcionalidades (*plug-ins*).

Login



You need to sign in or sign up before continuing.

Nome de usuário ou e-mail

Senha

☐ Lembrar senha

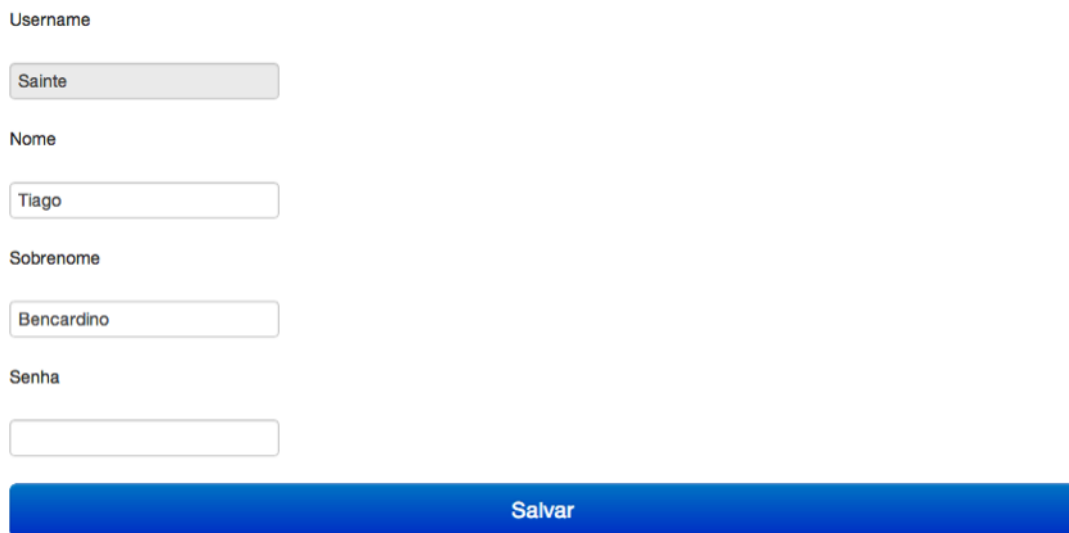
Entrar

[Cadastrar](#)

[Esqueceu sua senha?](#)

Figura 4.1: Tela mostrada quando o usuário tenta acessar recurso e não está logado

Editando Usuário



Username

Sainte

Nome

Tiago

Sobrenome

Bencardino

Senha

Salvar

Figura 4.2: Edição de dados do usuário

Esqueceu sua senha?



Email

Envie-me instruções para resetar a senha

[Logar](#)

[Cadastrar](#)

Figura 4.3: Lembrar senha do usuário

Para a validação da sessão no *mobile*, utilizamos essencialmente manipulação por *tokens*. No momento que o usuário abre a aplicação, é verificado se o *token* é válido: caso não seja, uma tela pedindo nome de usuário e senha é mostrada. Após o usuário fornecer os dados, é enviada uma requisição em [JSON](#) para /api/v1/token_controller,

que verifica se os dados estão corretos e se o usuário realmente existe. Se tudo ocorrer corretamente, um *token* é retornado da requisição; caso contrário, uma mensagem contendo o erro é retornada.



Figura 4.4: Login no mobile (iOS)

4.7 Front-End

No modelo [MVC](#), a parte de *view* (visão) é responsável pelas interações com o usuário, recebendo as entradas e renderizando a saída. Os tipos de arquivos de fronteira mais comuns são o [HTML](#), ERB e o eRuby. Nessa aplicação, utilizou-se em quase todas as páginas a extensão ERB.

Os *controllers*, por sua vez, são responsáveis por receber a ação de uma *view* e executar alguma lógica ligada a algum modelo. Basicamente, a função do *controller* é de atender às requisições entre modelo e visão, fazendo toda a tradução necessária entre as camadas.

Um *controller* pode ser criado através de um comando *generate*, como a seguir:

```
$rails generate controller quizzes
```

Esse comando irá criar uma classe chamada *quizzes_controller.rb* e uma pasta chamada *quizzes*, dentro da pasta *views*. A ligação entre as camadas é feita automaticamente, devido ao conceito de “convenção sobre configuração”.

Nesse *controller* criado, foram colocados métodos de manipulação de dados (*RESTful*). Na parte de visão, colocamos alguns arquivos para manipular *quizzes*.

Veja, a seguir, o código da página `index.html.erb`, que tem como função listar *quizzes* e navegar para tela de criação:

```
<h1>Quizzes</h1>

<table class="table_table-hover">
  <thead>
    <tr>
      <th>#</th>
      <th> Titulo </th>
      <th> Descricao </th>
      <th> Criador </th>
      <th> #Perguntas </th>
    </tr>
  </thead>
  <tbody>
    <% @quizzes.each do |quiz| %>
      <tr>
        <td><%= quiz.id %></td>
        <td><%= link_to quiz.titulo, quiz %></td>
        <td><%= quiz.descricao %></td>
        <td><%= link_to quiz.user.username, quiz.user %></td>
        <td><%= quiz.perguntas.size %></td>
      </tr>
    <% end %>
  </tbody>
</table>

<%= link_to new_quiz_path do %>
  <button class="btn btn-large btn-block btn-primary" type="button">
    Criar quiz
  </button>
<% end %>
```

Nesse código, é possível observar que existe código Ruby inserido entre os caracteres “<%” e “%>”. Esses caracteres representam um escape do **HTML** para o Ruby. Quando colocado com um sinal de igual, “<%=”, o resultado da linha executada é impressa na tela final **HTML** apresentada ao usuário.

É importante observar que todos os atributos de classe do *controller*, representados por um @, como em @quizzes, estão sobre o mesmo escopo.

4.7.1 Bootstrap

“Elegante, intuitivo e poderoso *framework* de *front-end* para desenvolvimento *web* rápido e fácil” - com essa descrição, o *Bootstrap* é apresentado em seu *site* oficial [23]. Lançado em Agosto de 2011 pelo Twitter, é atualmente o projeto mais popular do GitHub [24].

O *Bootstrap* é uma coleção livre de ferramentas para desenvolver *sites* e aplicações *web*. Basicamente, contém *templates HTML*, JavaScript e *Cascading Style Sheets (CSS)* para tipografia, formas, botões, gráficos, navegações e etc. Na aplicação, utilizamos o *Bootstrap* para personalização de botões, tabelas, divisão da página em duas seções e menu.



Figura 4.5: Exemplo de controle *Bootstrap*

Uma das funcionalidades mais aclamadas do *Bootstrap* é a adequação da tela para resoluções menores: quando colocado em telas de pouca largura, tipicamente em *smartphones*, a página se adequa para ser apresentada em um *layout* mais amigável para dispositivos móveis, como abaixo:

iQuizzer	Quizzes...	Quizzes	Usuários	Resultados	Sainte ▾
----------	------------	---------	----------	------------	----------

Usuários				Sobre	
#	Apelido	Pontos criador	Pontos jogador	Esse aplicativo é parte de uma plataforma experimental utilizando REST em aplicativos móveis iOS/Android, para trabalho de conclusão de curso em Engenharia de Teleinformática da Universidade Federal do Ceará.	
1	Sainte	300	280		
4	ronaldo		0		
2	tsainte				

Figura 4.6: Tela desktop estendida



Figura 4.7: Tela *mobile* com menu suprimido



Figura 4.8: Tela *mobile* com menu extendido

4.8 Deploy

Para que uma aplicação *Rails* execute, é necessário um servidor de aplicação. Em ambiente de desenvolvimento, utilizou-se o Webrick, que é um servidor já embutido em todas as aplicações. Para executá-lo, basta abrir o terminal, navegar até o diretório raiz da aplicação e executar o seguinte comando:

```
rails server
```

Caso uma porta diferente da porta padrão seja necessária (3000), pode ser passado o parâmetro `-p xxxx`, onde `xxxx` é a porta desejada.

O ambiente de produção deste projeto está hospedado como serviço no Heroku, sobre a instância “Celadon Cedar”. Para ativar, foi necessário fazer o cadastro, instalar o *Heroku Toolbelt*, efetuar *login* via terminal e fazer *deploy*. Mais detalhes podem ser encontrados no QuickStart do Heroku [25].

A submissão de arquivos para o servidor do Heroku é feita através do controlador de versão Git. Para submeter, deve-se realizar o *commit*⁸ do código e realizar um *git push*⁹ para o servidor. Caso seja necessário executar algum comando, como *rake* de *migrations*, usa-se o *run*, conforme a seguir:

```
heroku run rake db:migrate
```

⁸O termo “commit” refere-se à ideia de fazer permanentes um conjunto de mudanças experimentais. Em um controlador de versão, é uma ação onde as alterações nos códigos são salvas, com uma descrição e colocadas em uma determinada versão.

⁹O *git push* é um comando do Git para o envio de uma versão do projeto para um servidor git.

Comparativo Android x iOS

5.1 Ambientação

5.1.1 iOS

Para implementação nativa de aplicações, utilizamos a [IDE](#) gratuita chamada *Xcode*. Desenvolvida pela Apple Inc., funciona apenas sobre o sistema operacional *Mac OS X*. Por padrão, já vem com suporte ao Objective-C, linguagem de programação utilizada para desenvolvimento de aplicativos nativos no *Mac OS X* e no *iOS*.

Atualmente, o *Xcode* está na versão 4.5.2 e pode ser baixado via Mac App Store, de forma gratuita para usuários do *Mac OS X Lion* e *Mac OS X Mountain Lion*, disponível em [\[26\]](#).

Para desenvolvimento *iOS*, é necessário possuir o *iOS SDK*, que pode ser baixado internamente a ferramenta *Xcode*. Entretanto, para realização de testes no dispositivo, é necessária a posse da licença de desenvolvimento.

5.1.2 Android

Para o desenvolvimento de aplicações nativas em *Android*, necessita-se de um ambiente que possua *Android SDK* instalado. É possível, por exemplo, gerar aplicativos utilizando apenas um editor de texto, como Notepad ou Gedit. Entretanto, a Google recomenda o uso do Eclipse com ADT Plugin, disponível em [\[27\]](#).

5.2 Mostrando textos

5.2.1 iOS

Para exibir textos sem interação direta com o usuário, utilizamos uma instância da classe `UILabel`. Essa classe possui algumas propriedades para modificação de aspectos, a saber:

- ▶ `text`: o texto mostrado no campo;
- ▶ `textColor`: cor do texto;
- ▶ `numberOfLines`: número máximo de linhas suportadas;
- ▶ `font`: fonte utilizada pelo texto, instância de `UIFont`.

5.2.2 Android

De maneira similar ao *iOS*, o *Android* possui um controle específico para mostrar textos na tela, chamado `TextView`. Um `TextView` pode ser definido no *layout XML* ou no próprio Java. Assim como no *iOS*, possui uma propriedade do tipo `String` chamada “`text`”, a qual representa o texto apresentado.

Uma diferença importante entre o `UILabel` e o `TextView` é que o último aceita interações com o usuário; podemos colocar eventos de interação com o usuário. Essa funcionalidade costuma ser usada com textos que representam *links*.

5.3 Inserindo textos

5.3.1 iOS

Para entrada de textos, utilizamos uma instância de `UITextField`. A manipulação do texto é feita através do disparo de uma ação para um *target* quando o usuário pressiona o botão *return* do teclado.

Essa classe normalmente é associada a um `UITextFieldDelegate`, o qual fornece métodos adicionais de decisão.

Quando o usuário toca em um `textfield`, esse controle torna-se o *first responder* e invoca o aparecimento do teclado para o sistema. O teclado deve ser configurado, pelo desenvolvedor, para desaparecer quando o botão de *return* for pressionado. Isso deve ser feito através da mensagem `resignFirstResponder`, a ser enviada para o `textfield`.

Para que o `textfield` não fique “escondido” na tela, embaixo do teclado, cabe ao desenvolvedor mover tela de maneira conveniente, de forma a aparecer o conteúdo.

A aparência do teclado pode ser configurada utilizando o protocolo `UITextInputTraits`. Existem, entretanto, alguns tipos de teclado a serem definidos por padrão, como o *ASCII*, *Number*, *Url*, *Email*, entre outros.

5.3.2 Android

Para a entrada de texto do usuário, o Android fornece o controle chamado `EditText`. Assim como o `UITextField`, esse controle possui uma propriedade chamada *text*, que representa o texto mostrado no controle. Esse texto pode ser tanto a entrada do usuário como também um texto configurado via programação pelo aplicativo.

Para cada `EditText`, podemos definir o tipo de teclado que será exibido ao interagir com o usuário, como teclados de telefone, números e de letras.

O `EditText` possui duas grandes diferenças em relação ao `UITextField`, a saber:

- ▶ O teclado retorna automaticamente quando terminada a edição;
- ▶ Ao entrar em modo de edição, o `EditText` faz com que a tela do aplicativo desloque-se, caso seja necessário para aparecer o conteúdo do controle.

5.4 Botões e eventos

5.4.1 iOS

Um botão é representado por uma instância da classe `UIButton`. Os botões interceptam eventos de toque e enviam mensagens para um *target*(alvo) pré-definido quando tocados. Métodos para configurar os *targets* e ações são herdados de `UIControl`. Como atributos, possuem título, imagem e outras propriedades de aparência.

Os botões respondem a algumas ações, como *touch drag*, *touch down*, *touch up*, entre outros. Para associar um método a um evento, criamos uma `IBAction` (método) e, ao evento, indicamos tal `IBAction`. Isso pode ser feito facilmente no *Interface Builder/Xcode*, “ligando” o botão ao código correspondente (*file’s owner* desse botão), conforme os passos abaixo:

- i. Arrastar, com o botão direito, o botão ao código do *file’s owner*

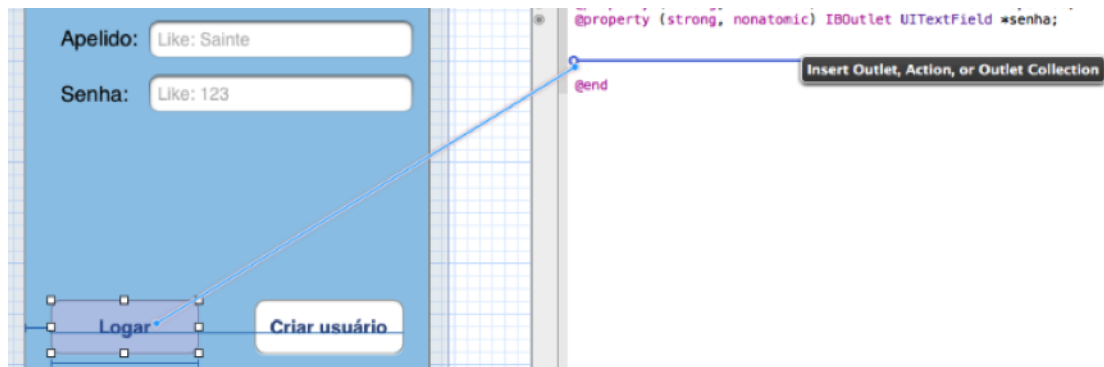


Figura 5.1: Criando ligação entre interface builder e o código

ii. Nomear um IBAction e associar um evento

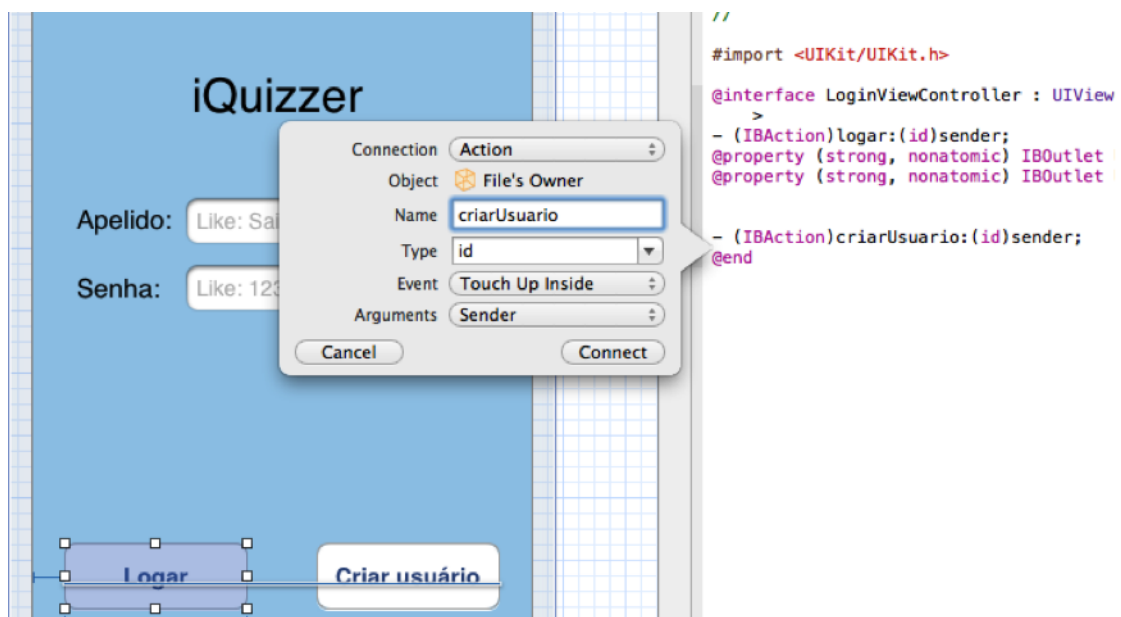


Figura 5.2: Criando IBAction e associando evento

5.4.2 Android

Os botões no *Android* pertencem à classe `Button`. Existem duas formas de inserir eventos nos botões, a saber:

- ▶ via `Listener Java`: podemos implementar o `OnClickListener` na `Activity` ou em alguma classe anônima dentro da `Activity`, de modo que ao disparar o evento de *click*, o método `onClick()` da interface seja chamado.
- ▶ via propriedade `onClick` no `LayoutXML`: os botões possuem uma propriedade chamada `onClick`, que recebem uma `String` como valor. Esse valor é o nome do método da `Activity` onde o *layout* contendo esse botão foi inflado¹. Neste

¹Para que um *layout XML* ser colocado dentro de outro *layout XML*, é necessário “inflar”, utilizando a classe `LayoutInflater`.

projeto, foi utilizado somente `onClick` como propriedade [XML](#), uma vez que toda a parte de *layout* foi implementada via [XML](#).

5.5 Criando listas

5.5.1 iOS

Listas em *iOS* são feitas utilizando instâncias de `UITableView`. Essa classe, por sua vez, estende de `UIScrollView`, que habilita a rolagem vertical (e apenas a vertical). Internamente, cada linha (célula) é representada por um objeto de `UITableViewCell`, que são totalmente configuráveis.

Esse controle normalmente é associado a um `UINavigationController`: quando uma célula é tocada, é feito um *push* de um novo `UIViewController`, detalhando a célula.

Table views possuem dois estilos, a saber: `UITableViewStylePlain` e `UITableViewStyleGrouped`. Uma vez criado o controle, não é possível mudar o estilo. No estilo `Plain`, as seções de *header/footer* flutuam nas bordas do conteúdo. Nesse estilo, pode haver um índice variando de A - Z, que facilita a navegação vertical. No estilo `Grouped`, uma cor padrão é definida para o fundo da *view* e das células. Nesse estilo, podem ser criadas várias listas, de formas agrupadas. Nesse caso, não podem ter índice.

Muitos métodos utilizam o objeto do tipo `NSIndexPath` como parâmetro e retornam valores. Esse objeto representa o índice da linha atual e da seção atual.

Um objeto do tipo *Table View* necessita de um objeto que atue como fonte de dados (*data source*) e um objeto que atue como *delegate*. Normalmente, utilizamos os protocolos `UITableViewDataSource` e `UITableViewDelegate` no `UIController` no qual o *table view* esteja inserido. O *data source* fornece informação que o `UITableView` precisa para construir tabelas e o *delegate* fornece as células e executa alguns outros métodos de manipulação.

A fim de criar uma *table view* básica, precisa-se implementar, no mínimo, os seguintes métodos *data source*:

- i. Número de seções - retorna o número de seções para essa *table view*

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView;
```

- ii. Número de *rows* (linhas): retorna o número de linhas para cada uma das seções

```
- (NSInteger)tableView:(UITableView *)tableView  
  numberOfRowsInSection:(NSInteger)section;
```

iii. *cell for row at index*: define uma célula para cada *index* (par linha/seção) da tabela

```
-(UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath;
```

Além desses métodos, é comum implementar um método *delegate* que responda a cada célula tocada, como a seguir:

did select row at index: informa o par (linha/seção) que foi selecionado. A partir do *index path*, pode-se recuperar a célula selecionada dentro da *table view*

```
-(NSInteger)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath;
```

Para a tela de escolha dos *quizzes* (GameMenu), implementou-se uma *table view* simples, da seguinte maneira:

```
-(NSInteger)numberOfSectionsInTableView:(UITableView *)tableView{
    return 1;
}

-(NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section{
    //conta no array de quizzes a quantidade de elementos
    return [self.quizzes count];
}

-(UITableViewCell*)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath{
    /* reaproveita ou cria uma célula */
    UITableViewCell* cell =
    [tableView dequeueReusableCellWithIdentifier:@"cell"];

    if (cell == nil){
        cell = [[UITableViewCell alloc] init];
    }

    /* personalizacao da celula */
    Quiz* q = [self.quizzes objectAtIndex:indexPath.row];
    cell.textLabel.text = [q titulo];

    return cell;
}

/* disparado quando um dos quizzes for selecionado */
-(void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath{
    Quiz* q = [self.quizzes objectAtIndex:indexPath.row];
    GameController* gc = [[GameViewController alloc]
```

```
initWithNibName:@"GameViewController" bundle:nil];

gq.quiz = q;

//navegando entre telas
[self.navigationController pushViewController:gq animated:YES];
}
```

5.5.2 Android

As listas em *Android* são instâncias da classe widget `ListView` e, assim como no *iOS*, já possui rolagem vertical implementada. Os itens da lista são inseridos por outra classe controladora, chamada `Adapter`.

O `Adapter` é uma classe que provê acesso aos itens que contém a informação, como um array de strings. Além disso, o `Adapter` é responsável por criar o *layout* de cada célula.

Para a lista simples mostrada em `GameMenu`, foi implementado uma lista com células padrão da seguinte maneira:

```
private void carregarLista() {

    ArrayAdapter arrayAdapter = new ArrayAdapter(this,
        android.R.layout.simple_list_item_1, quizzes);
    ListView listView = (ListView) findViewById(R.id.lista_quizzes);
    listView.setAdapter(arrayAdapter);
    listView.setOnItemClickListener(this);

}
```

Nesse trecho de código, *quizzes* é um `ArrayList` de *quizzes*. `ArrayAdapter` e `ListView` são classes padrão do *Android*.

Os eventos de clique de célula estão associados a classe que implementa a interface `OnItemClickListener`. Tal interface exige a implementação do método `onClickListener()`. Foi utilizado, no *iQuizzer*, esse *listener* na classe `GameMenu`, tendo sido implementado o método da seguinte maneira:

```
@Override
public void onItemClick(AdapterView<?> adapterView, View view,
    int position, long id) {
    Quiz quiz = quizzes.get(position);
    Intent i = new Intent(getApplicationContext(), GameActivity.class);
    i.putExtra("quiz", quiz);
    startActivity(i);
}
```

}

5.6 Acesso a dados

5.6.1 SQLite

O SQLite é uma biblioteca implementada em C, de domínio público, que representa um banco de dados SQL. Tem como características fundamentais ser contido em si mesmo, livre de servidores, sem configuração e transacional, conforme descrito em [28].

Tanto o *iOS* quanto o *Android* possuem abstrações para representar objetos da tabela (*active records*), conexões e o próprio banco.

5.6.1.1 iOS

Existem duas maneiras de se trabalhar com SQLite em *iOS*: acesso nativo através da biblioteca `SQLite.h` e `Core Data Framework`.

No acesso nativo, a programação utilizada é a mesma para aplicações nativas em C, ou seja, não há particularidades entre utilizar o SQLite dentro ou fora do iOS. Já em `Core Data`, a programação é em Objective-C e existe uma camada de abstração do banco de dados.

Na aplicação, optou-se por usar o `Core Data` devido à facilidade de fazer a modelagem utilizando o *xcdatamodel*. O *xcdatamodel* é uma ferramenta facilitadora, integrada ao *Xcode*, na qual o desenvolvedor pode “desenhar” as tabelas e criar ligações, chamadas *relationships*. Cada uma das tabelas e relações possui propriedades, que são facilmente configuradas.

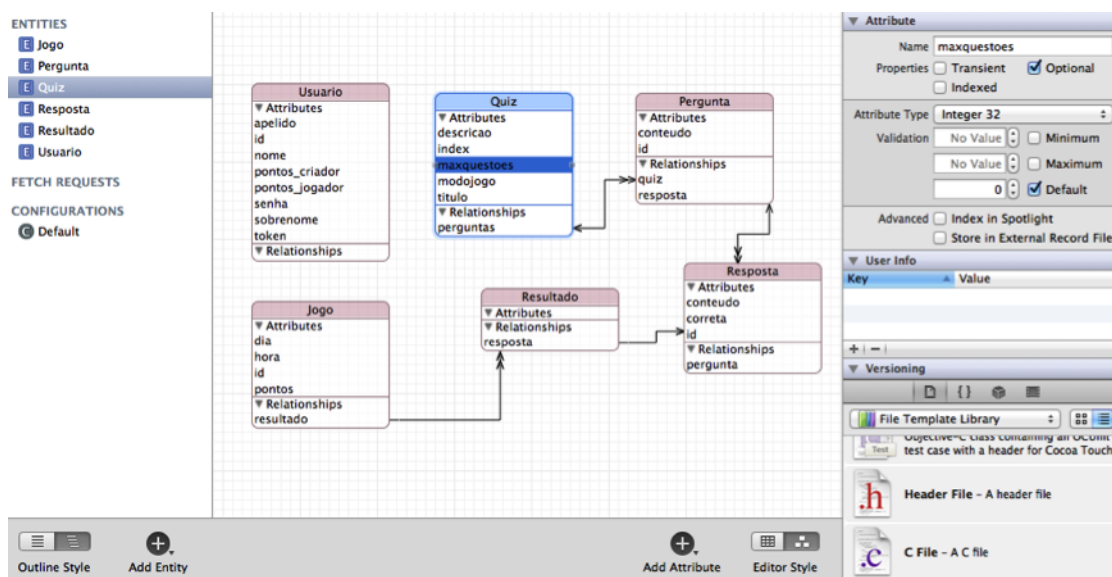


Figura 5.3: Ferramenta gráfica para manipulação do banco de dados utilizando o *xcdatamodel*

Existem três classes principais de abstração de banco de dados, a saber:

- ▶ *Managed Object Model*: é a classe que contém as definições de cada objeto (entidades, *active record*);
- ▶ *Persistent store coordinator*: é a conexão do banco; são configurados os nomes e a localização da base de dados;
- ▶ *Managed Object Context*: classe responsável pelas operações básicas de manipulação: inserir objetos, apagar objetos, etc.

No iQuizzer para *iOS*, foi criada uma classe chamada DAO e implementados os métodos de **CRUD**, como pesquisar, inserir, modificar e deletar. Para cada uma das entidades, foi criada uma classe de *Data Access Object* (**DAO**)² estendida, como QuizDAO ou PerguntaDAO, onde cada uma dessas realiza operações de **CRUD** mais específicas e voltadas para as situações que ocorrem na aplicação. Nessas classes de **DAO**, foram feitas manipulações de *managed object context* e *persistent store coordinator*.

Para cada uma das entidades, foi criado um *managed object model*, estendendo da classe nativa `NSManagedObject`. É possível criar tais classes com os atributos e relações pré-configuradas, selecionando o template “`NSManagedObject subclass`” e selecionando o *data model* desejado.

5.6.1.2 Android

O *framework* Android possui um pacote chamado `android.database.sqlite`, que fornece todas as classes necessárias para o gerenciamento do banco de dados privado a cada aplicação. Por padrão, o *Android* vêm com a versão 3.4.0 do SQLite.

Na aplicação, foram utilizadas as seguintes classes desse pacote:

- ▶ `SQLiteOpenHelper`: Essa classe possui métodos para abrir o banco, como `onCreate(SQLiteDatabase)` e `onUpgrade(SQLiteDatabase, int, int)`, que abrem, criam e atualizam o banco caso necessário.
- ▶ `SQLiteDatabase`: possui métodos para gerenciar o banco SQLite. Com essa classe, foram feitas as interações com as entidades do banco, utilizando essencialmente o método `execSQL(String)` para entrada e manutenção de tuplas.
- ▶ `Cursor`: classe de manipulação de resultados de uma consulta.

²DAO, ou em português, objeto de acesso a dados, é um padrão para persistência de dados que permite separar regras de negócio das regras de acesso a banco de dados.

5.6.2 Preferências

Existem casos onde a complexidade da informação a ser armazenada é pequena, como pares de chave-valor³. Em casos onde se tem uma informação simples a ser armazenada, utiliza-se a memória de preferências para gravar tais valores.

Basicamente, essas classes de preferências atuam como um `HashTable`, que armazena uma estrutura de chave e valor para tipos primitivos, e os valores armazenados estarão no escopo da aplicação mesmo que a aplicação seja encerrada ou o dispositivo desligado.

O funcionamento básico dessa funcionalidade é similar no *iOS* e *Android*: uma chave (`String`) é escolhida para atribuir um nome à informação a ser armazenada. Os métodos para inserir/modificar são chamados de *set*. Para recuperar o valor, invoca-se um método *get*, passando o nome atribuído à variável como parâmetro.

No aplicativo, foram utilizadas as memórias de preferências para armazenar o *token*. Quando a aplicação começa, verifica se há um valor de *token* válido armazenado. Em caso positivo, a aplicação inicia na tela de menu; caso contrário, é mostrada a tela de *login*.

5.6.2.1 iOS

No *iOS*, a classe que representa a memória de preferências é chamada de `NSUserDefaults`. Para ler ou escrever, necessita-se de uma instância de `NSUserDefaults` - na aplicação, `standardUserDefaults`. Com a instância, e de posse do valor da chave, pode-se escrever os seguintes métodos de acesso:

i. Instância padrão:

ii. Configurando valor para *token*:

```
[defaults setObject:token forKey:@"token"]; //escrita de token
```

iii. Recuperando o valor de *token*:

```
[defaults objectForKey:@"token"];
```

5.6.2.2 Android

No *Android*, a classe que representa a memória de preferências é chamada de `SharedPreferences`. De maneira similar ao *iOS*, tem-se um método para atribuir valor e outro para recuperar. Entretanto, no momento da recuperação, é passado um valor padrão a ser retornado caso não exista o valor procurado. Além disso, é preciso realizar o *commit* na memória depois de adicionar um determinado valor.

³Essa denominação é feita para associações entre um valor e uma referência (chave) a esse valor.

A atribuição e a recuperação são feitas da seguinte maneira:

i. Instância padrão:

```
SharedPreferences preferences =  
    PreferenceManager.getDefaultSharedPreferences(context);  
SharedPreferences.Editor editor = preferences.edit();
```

ii. Configurando valor para *token*:

```
editor.putString("token", token);  
editor.commit();
```

iii. Recuperando o valor de *token*:

```
String token = preferences.getString("token", "");
```

5.7 Parser JSON

A fim de manipular objetos [JSON](#), tanto na formação quanto na interpretação, existem *parsers* em forma de bibliotecas nativas incorporadas aos *frameworks* nativos do *iOS* e do *Android*. Tais bibliotecas trabalham de maneira similar, utilizando o conceito de chave e valor.

Na aplicação, todas as mensagens trocadas entre *mobile* e servidor utilizam [JSON](#), sendo de vital importância a compreensão dessa funcionalidade.

Para baixar *quizzes*, é recebido o seguinte [JSON](#) do servidor, no corpo do [HTTP](#):

```
{  
  "quiz":  
  {  
    "created_at": "2012-11-05T16:43:14Z",  
    "descricao": "Perguntas sobre fatos que ocorreram em 2012.",  
    "id": 3,  
    "maxquestoes": 5,  
    "modojogo": 1,  
    "titulo": "Retrospectiva 2012",  
    "updated_at": "2012-12-28T10:18:01Z",  
    "user_id": 1,  
    "perguntas": [.... ]  
  }  
}
```

5.7.1 iOS

A partir do *iOS 5*, pode-se utilizar a classe `NSJSONSerialization` para criar e interpretar [JSON](#). Essa classe trabalha com objetos comuns de representação de

dados, como `NSString`, `NSNumber`, `NSArray` e `NSDictionary`, não necessitando de *wrappers*⁴.

No método de baixar *quiz* do servidor, foi utilizada a seguinte conversão entre o objeto `JSON` (um `NSData` contendo o corpo do `HTTP`) e um objeto `Quiz`:

```
NSError* error;

NSDictionary* jsonObj = [NSJSONSerialization JSONObjectWithData:jsonData
options:kNilOptions error:&error];

NSDictionary* jsonQuiz = [jsonObj objectForKey:@"quiz"];

//Instanciando quiz a partir de uma entidade
Quiz* quiz = [[Quiz alloc] initWithEntity:entityDescription
insertIntoManagedObjectContext:managedContext];

quiz.titulo = [jsonQuiz objectForKey:@"titulo"];
quiz.descricao = [jsonQuiz objectForKey:@"descricao"];
quiz.maxquestoes = [jsonQuiz objectForKey:@"maxquestoes"];
quiz.modojogo = [jsonQuiz objectForKey:@"modojogo"];
quiz.index = [jsonQuiz objectForKey:@"id"];
```

Assim, foi utilizada apenas uma conversão entre o `JSON` e um `NSDictionary`. Caso a raiz do `JSON` fosse um *array*, a única modificação seria que o objeto a ser retornado da conversão seria um `NSArray`.

Para criar um objeto `JSON` a partir de um `NSDictionary` ou `NSArray`, foi utilizado o método contrário ao que utilizamos na interpretação, conforme abaixo:

```
NSArray* objects = [[NSArray alloc] initWithObjects:jogo.dia, jogo.hora,
jogo.pontos, resultados, usuario_id, nil];
NSArray* keys = [[NSArray alloc] initWithObjects:@"dia",@"hora",
@"pontos",@"resultados_attributes", @"user_id", nil];

NSMutableDictionary* jsonDict = [[NSMutableDictionary alloc]
initWithObjects:objectsforKeys:keys];

NSData* jsonData = [NSJSONSerialization dataWithJSONObject:jsonDict
options:kNilOptions error:nil];
```

5.7.2 Android

No *Android*, existe uma implementação da biblioteca oficial disponível em [29] interna ao *framework*. Ou seja, a maneira de se trabalhar com `JSON` é a mesma de

⁴O termo “*Wrapper*” designa classes com função de empacotamento, de modo a abstrair classes de um tipo para outro.

aplicações java que utilizam tal biblioteca.

No método de baixar *quiz* do servidor, foi utilizado a seguinte conversão entre o objeto **JSON** (uma `String` contendo o corpo do **HTTP**) e um objeto `Quiz`:

```
JSONObject jsonObj = new JSONObject(jsonData);
JSONObject jsonQuiz = jsonObj.getJSONObject("quiz");

Quiz quiz = new Quiz(jsonQuiz.getInt("id"), jsonQuiz.getString("titulo"),
    jsonQuiz.getString("descricao"),
    jsonQuiz.getInt("modojogo"), jsonQuiz.getInt("maxquestoes"));
```

Verificou-se que a maneira de interpretar **JSON** é bem parecida no *iOS* e no *Android*. A diferença notável é que, enquanto o *iOS* trabalha com objetos “nativos” como `NSDictionary` e `NSArray`, o *Android* trabalha com *wrappers* `JSONObject` e `JSONArray`.

Na criação de um **JSON**, como no método de criar jogo para enviar o resultado ao servidor, foi utilizada a seguinte conversão:

```
JSONObject jsonObject = new JSONObject();
try{
    jsonObject.put("dia", jogo.getDia());
    jsonObject.put("hora", jogo.getHora());
    jsonObject.put("pontos", jogo.getPontos());
    jsonObject.put("resultados_attributes", jsonResultados);
    jsonObject.put("user_id", usuario_id);
} catch (Exception e ){
    e.printStackTrace();
}
return jsonObject.toString();
```

Novamente, pode-se observar que a maior diferença é relativa ao *wrapper* `JSONObject` e `JSONArray`, presentes apenas na implementação *Android*.

5.8 Conexão HTTP

Conforme já citado, toda a troca de mensagens entre o servidor e a aplicação móvel é feita através do protocolo **HTTP**. Dentro do corpo da mensagem, é enviado ou recebido um **JSON**, contendo, normalmente, um recurso (como uma instância de `Quiz` ou `Pergunta`, por exemplo).

O protocolo **HTTP** possui alguns campos em seu cabeçalho que são especialmente úteis para nossa aplicação. São eles:

- *Method*: nesse campo, define-se o tipo de método **HTTP** que será executado. Esse

campo é essencialmente importante para aplicações [REST](#), uma vez que usa os mesmos (GET, POST, PUT e DELETE) para realizar as operações de [CRUD](#);

- *Content-Type*: nesse campo é definido o tipo de arquivo que estará sendo enviado. Para que a aplicação *RESTful* saiba que está sendo enviada uma requisição via *mobile* com [JSON](#) interno, utilizamos o *content-type* “JSON” para diferenciar. Caso seja uma requisição web normal, originada do navegador, o *content-type* é “text/plain”.

Existem duas formas de tratar as conexões [HTTP](#): sincronamente e assincronamente. Por uma questão de tempo e aproveitamento de código, optou-se pela maneira síncrona. Entretanto, a maneira assíncrona é, possivelmente, mais bem aceita pelo usuário, uma vez que não interrompe a execução da aplicação e, por conseguinte, será apresentada como sugestão de trabalho futuro dessa aplicação, na seção [6.3](#).

Na aplicação, criou-se uma classe chamada `WebService`, que possui métodos de comunicação com o servidor web através do protocolo [HTTP](#).

5.8.1 iOS

A implementação de requisições [HTTP](#) é feita utilizando as seguintes classes:

- `NSURL`: representa a [URL](#) que será enviada à requisição;
- `NSURLRequest`: representa a requisição [HTTP](#). Na instância dessa classe, definimos o método [HTTP](#) desejado, o *content-type* e o corpo da mensagem;
- `NSURLConnection`: representa a conexão entre o dispositivo e o servidor, possuindo uma [URL](#) e um request. Além disso, indica a classe delegate da requisição, ou seja, a classe que possui os métodos necessários para tratar as respostas dessa requisição

Na classe `WebService`, foram criados os métodos `get` e `RESTCommand`. No método `RESTCommand`, além de configurar-se a [URL](#), o cabeçalho e o corpo do [HTTP](#), define-se um *delegate* para o objeto `NSURLConnection`. Esse *delegate* é responsável por tratar a resposta da requisição, através dos métodos `didReceiveData` e `connectionDidFinishLoading`.

5.8.2 Android

Para utilizar a maneira síncrona, deve-se primeiramente modificar a política de *threads* padrão do *Android*. Isso foi feito adicionando o seguinte código no método `onCreate()` da *activity* principal:

```
StrictMode.ThreadPolicy policy =
```

```
new StrictMode.ThreadPolicy.Builder().permitAll().build();
StrictMode.setThreadPolicy(policy);
```

Para realizar requisições [HTTP](#), foram utilizadas instâncias das seguintes classes:

- ▶ [URL](#): representa a [URL](#) que será enviada à requisição;
- ▶ `HttpClient`: representa o cliente (alvo) da requisição;
- ▶ `HttpURLConnection`: representa a conexão e configura cabeçalhos [HTTP](#);
- ▶ `OutputStream` e `BufferedReader`: gerenciam os bytes enviados e recebidos durante a transmissão.

De maneira análoga ao *iOS*, foram criados os métodos `get` e `RESTCommand`, de forma a fazer a comunicação síncrona.

Conclusão

6.1 Contribuições

Este trabalho busca apresentar duas contribuições sob diferentes perspectivas: a primeira, referente ao aplicativo *iQuizzer*, mostra a concepção de uma aplicação distribuída desenvolvida em uma arquitetura que integra diferentes tecnologias; a segunda, referente à documentação de configuração e de implementação de arquitetura de aplicativo móvel utilizando *web services*, apresenta detalhes sobre a implementação, buscando, quando possível discutir diferenças entre o desenvolvimento para *iOS* e *Android*.

Os estudos foram contextualizados no desenvolvimento do aplicativo *iQuizzer*, disponibilizado nas versões *web* e *mobile*, que visa estimular a busca por conhecimento sobre assuntos diversos, tanto para quem cria um *quiz* quanto para quem joga. Além disso, a aplicação desenvolvida permite que enquetes sejam realizadas de maneira rápida, sendo os resultados avaliados remotamente em uma plataforma *web*.

A documentação feita nesta monografia fornece uma base para muitas das aplicações que estão em desenvolvimento na atualidade, utilizando a abordagem de *web service* integrada à computação móvel. Tal abordagem vem sendo utilizada por um grande número de *startups*, as quais necessitam, essencialmente, desenvolver e avaliar seus produtos de maneira rápida, mesmo que o produto ainda esteja em fase de desenvolvimento.

Todos os projetos encontram-se versionados, disponíveis no GitHub nos repositórios *iQuizzer_Rails* [30], *iQuizzer_iOS* [31] e *iQuizzer_Android* [32].

6.2 Limitações

Algumas das funcionalidades não implementadas para o *iQuizzer* representam algumas das limitações do projeto, como: a não atualização dos *quizzes* depois de

baixados, mesmo que modificados na *web*; a não adoção de conexões assíncronas; a falta de integração com as redes sociais; a falta de criptografia na troca de mensagens. Entretanto, estes aspectos podem ser relacionados como expansões deste trabalho.

No que concerne às plataformas de desenvolvimento *mobile*, existem duas limitações de versões: o aplicativo desenvolvido em *iOS* não funciona em versões inferiores ao *iOS 5.0*, devido ao uso de bibliotecas exclusivas dessa versão; de maneira similar, o aplicativo desenvolvido em *Android* não funciona em versões inferiores ao *Android 2.3 Gingerbread*.

6.3 Trabalhos futuros

A fim de lançar a aplicação *iQuizzer* no mercado, de maneira escalável e competitiva, algumas funcionalidades deveriam ser implementadas:

- ▶ Integração com as redes sociais: o módulo *web* poderia ser disponível como um aplicativo para *Facebook*, onde o controle de usuários seria feito pela própria conta do usuário na rede. Além disso, as ações do usuário, tanto na parte *web* como na parte *mobile*, podem corresponder a ações no *Facebook*, de modo a divulgar a plataforma;
- ▶ Segurança: toda a troca de mensagens pode ser feita utilizando protocolos seguros, como *Secure Shell (SSH)*. Dessa forma, poderia garantir a inviabilidade da informação trocada;
- ▶ Elementos de rede social: ações como “curtir”, “comentar” e “seguir usuário” podem ser colocadas dentro da plataforma, de modo a aumentar a experiência do usuário;
- ▶ Disponibilização do jogo para outras plataformas: a função de jogar um quiz pode ser disponibilizada na *web*, no *Windows Phone*, entre outros;
- ▶ Conexões assíncronas, de modo a otimizar a experiência do usuário em relação a usabilidade;
- ▶ Melhorias no *layout*: no estado atual, a aplicação não possui uma identidade visual, o que é de vital importância para o sucesso comercial do aplicativo;
- ▶ Notificações e controles de atualização de *quizzes*;
- ▶ Monetização, com propagandas ou venda de *quizzes*: pode-se monetizar a aplicação colocando *banners*, tanto na *web* como no *mobile*. Poderia, também, ser criado um mini-sistema de venda de *quizzes*, onde os criadores de cada *quiz* lucrassem com a compra de *quizzes* pelos jogadores.

Referências Bibliográficas

- [1] E. Nakamura, “Computação móvel: novas oportunidades e novos desafios,” Disponível via Internet, <https://portal.fucapi.br/tec/imagens/revistas/ed02_04.pdf>. Acessado em Janeiro de 2013.
- [2] Rackspace, “Understanding cloud computing stack: Saas, paas, iaas,” Disponível via Internet, <http://broadcast.rackspace.com/hosting_knowledge/whitepapers/Understanding-the-Cloud-Computing-Stack.pdf>. Acessado em Janeiro de 2013.
- [3] J. Schofield, “Google angles for business users with ‘platform as a service’,” Disponível via Internet, <<http://www.guardian.co.uk/technology/2008/apr/17/google.software>>. Acessado em Janeiro de 2013.
- [4] C. Lee, “Facebook and heroku: an even easier way to get started,” Disponível via Internet, <<https://developers.facebook.com/blog/post/558>>. Acessado em Janeiro de 2013.
- [5] R. Fielding, “Architectural styles and the design of network based software architectures,” Disponível via Internet, <http://www.cs.colorado.edu/~kena/classes/7818/f08/lectures/lecture_9_fielding_disserta.pdf>. Acessado em Janeiro de 2013.
- [6] IETF, “Hypertext transfer protocol – http/1.1,” Disponível via Internet, <<http://tools.ietf.org/html/rfc2616>>. Acessado em Janeiro de 2013.
- [7] L. R. . S. Ruby, *RESTful Web Services*. O’Reilly, 2007.
- [8] IETF, “The application/json media type for javascript object notation (json),” Disponível via Internet, <<http://www.ietf.org/rfc/rfc4627.txt>>. Acessado em Janeiro de 2013.
- [9] N. Nurseitov, “Comparison of json and xml data interchange formats: A case study,” Disponível via Internet, <<http://www.cs.montana.edu/izurieta/pubs/caine2009.pdf>>. Acessado em Janeiro de 2013.

- [10] W. Resource, “Structures of json,” Disponível via Internet, <<http://www.w3resource.com/JSON/structures.php>>. Acessado em Janeiro de 2013.
- [11] R. Language, “About,” Disponível via Internet, <<http://www.ruby-lang.org/en/about/>>. Acessado em Janeiro de 2013.
- [12] Caelum, “Desenv. Ágil para web com ruby on rails,” Disponível via Internet, <<http://www.caelum.com.br/apostila-ruby-on-rails/>>. Acessado em Janeiro de 2013.
- [13] S. S. Laurent, *Learning rails 3*. O'Reilly, 2012.
- [14] S. Pastorino, “Rails 4 in 30’,” Disponível via Internet, <<http://blog.wyeworks.com/2012/10/29/rails-4-in-30-minutes/>>. Acessado em Janeiro de 2013.
- [15] M. Fowler, *Patterns of enterprise application architecture*. Addison-Wesley, 2003.
- [16] Y. Framework, “Best mvc practices,” Disponível via Internet, <<http://www.yiiframework.com/doc/guide/1.1/en/basics.best-practices>>. Acessado em Janeiro de 2013.
- [17] F. S. Equities, “Number of smartphones around the world top 1 billion – projected to double by 2015,” Disponível via Internet, <<http://finance.yahoo.com/news/number-smartphones-around-world-top-122000896.html>>. Acessado em Janeiro de 2013.
- [18] J. Callaham, “Gartner: Microsoft increases mobile os market share in q3 2012,” Disponível via Internet, <<http://www.neowin.net/news/gartner-microsoft-increases-mobile-os-market-share-in-q3-2012>>. Acessado em Janeiro de 2013.
- [19] D. Pilone, *Use a cabeça - Desenvolvimento para iPhone*, 2007.
- [20] R. Lecheta, *Google Android*. Novatec, 2011.
- [21] R. Installer, “Rails installer,” Disponível via Internet, <<http://www.railsinstaller.org>>. Acessado em Janeiro de 2013.
- [22] E. N. Cáceres, “Banco de dados: Conceitos básicos,” Disponível via Internet, <<http://www.dct.ufms.br/~edson/bd1/bd1.pdf>>. Acessado em Janeiro de 2013.
- [23] Twitter, “Bootstrap,” Disponível via Internet, <<http://twitter.github.com/bootstrap/>>. Acessado em Janeiro de 2013.
- [24] GitHub, “Popular starred repositories,” Disponível via Internet, <<https://github.com/popular/starred>>. Acessado em Janeiro de 2013.
- [25] Heroku, “Getting started with heroku,” Disponível via Internet, <<https://devcenter.heroku.com/articles/quickstart>>. Acessado em Janeiro de 2013.

- [26] Apple, “Xcode via itunes,” Disponível via Internet, <<https://itunes.apple.com/us/app/xcode/id497799835>>. Acessado em Janeiro de 2013.
- [27] Google, “Adt plugin para eclipse,” Disponível via Internet, <<http://developer.android.com/tools/sdk/eclipse-adt.html>>. Acessado em Janeiro de 2013.
- [28] SQLite, “Sobre o sqlite,” Disponível via Internet, <<http://www.sqlite.org/about.html>>. Acessado em Janeiro de 2013.
- [29] JSON, “Json reference,” Disponível via Internet, <<http://www.json.org>>. Acessado em Janeiro de 2013., 2013.
- [30] T. Bencardino, “iquizzer - ruby on rails,” Disponível via Internet, <https://github.com/tsainte/iQuizzer_rails>. Acessado em Janeiro de 2013.
- [31] —, “iquizzer - ios,” Disponível via Internet, <https://github.com/tsainte/iQuizzer_iOS>. Acessado em Janeiro de 2013.
- [32] —, “iquizzer - android,” Disponível via Internet, <https://github.com/tsainte/iQuizzer_android>. Acessado em Janeiro de 2013.