FACULDADE DE TECNOLOGIA DE AMERICANA
Curso de Análise e Desenvolvimento de Sistemas

Gustavo Vinícius Fernandes

DESENVOLVIMENTO DE UMA APLICAÇÃO MULTIPLATAFORMA EM METEOR.JS COMO SOLUÇÃO PARA PORTABILIDADE DE CÓDIGO

Americana, SP
2016

FACULDADE DE TECNOLOGIA DE AMERICANA

Curso de Análise e Desenvolvimento de Sistemas

Gustavo Vinícius Fernandes

DESENVOLVIMENTO DE UMA APLICAÇÃO MULTIPLATAFORMA EM METEOR.JS COMO SOLUÇÃO PARA PORTABILIDADE DE CÓDIGO

Trabalho de conclusão de curso desenvolvido em cumprimento à exigência curricular do curso de Análise e Desenvolvimento de Sistemas, sob orientação do Prof. Me. Diógenes de Oliveira.

Área de concentração: Desenvolvimento de sistemas.

Americana, SP

2016

FICHA CATALOGRÁFICA – Biblioteca Fatec Americana - CEETEPS
Dados Internacionais de Catalogação-na-fonte

F399d FERNANDES, Gustavo Vinícius
Desenvolvimento de uma aplicação multiplataforma em Meteor.js como solução para portabilidade de código. / Gustavo Vinícius Fernandes. – Americana: 2016.
52f.

Monografia (Curso de Tecnologia em Análise e Desenvolvimento de Sistemas). - - Faculdade de Tecnologia de Americana – Centro Estadual de Educação Tecnológica Paula Souza.
Orientador: Prof. Ms. Diógenes de Oliveira

1. Desenvolvimento de software I.OLIVEIRA, Diógenes de II.Centro Estadual de Educação Tecnológica Paula Souza – Faculdade de Tecnologia de Americana.

CDU:681.3.05

GUSTAVO VINÍCIUS FERNANDES

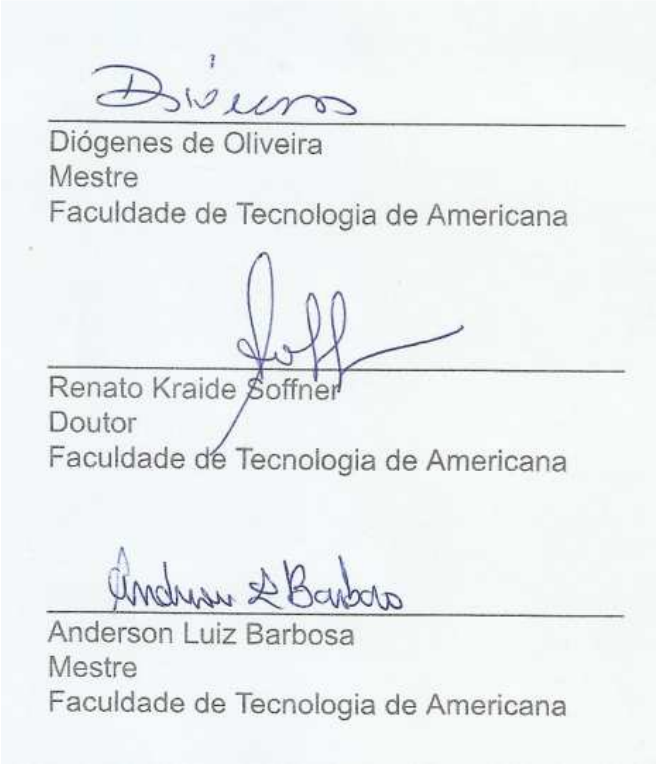
DESENVOLVIMENTO DE UMA APLICAÇÃO MULTIPLATAFORMA EM METEOR.JS COMO SOLUÇÃO PARA PORTABILIDADE DE CÓDIGO

Trabalho de graduação apresentado como exigência parcial para obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas pelo CEETEPS/Faculdade de Tecnologia - FATEC/Americana.

Área de concentração: Desenvolvimento de sistemas.

Americana, 6 de dezembro de 2016

Banca Examinadora:



Diógenes de Oliveira
Mestre
Faculdade de Tecnologia de Americana

Renato Kraide Soffner
Doutor
Faculdade de Tecnologia de Americana

Anderson Luiz Barbosa
Mestre
Faculdade de Tecnologia de Americana

DEDICATÓRIA

Aos meus pais, amigos, e professor orientador, que me deram força e apoio durante toda a minha caminhada rumo à minha formação.

RESUMO

O presente texto conceitua o desenvolvimento de uma aplicação multiplataforma em um *framework* JavaScript chamado Meteor.js. Este, por sua vez, trabalha integralmente em todas as camadas da aplicação, desde a renderização até a comunicação com o banco de dados, permitindo que o desenvolvedor utilize somente uma linguagem de programação em todo o processo de desenvolvimento.

Além disso, Meteor.js possui duas características marcantes. Uma delas é a capacidade de reutilização de código-fonte. Para isso, este *framework* utiliza um recurso chamado Cordova, o qual também é desenvolvido na mesma linguagem. Com isso, a mesma aplicação torna-se adaptável a qualquer ambiente de execução.

A segunda característica é a capacidade de entrega e atualização de dados. Isso significa que qualquer alteração, tanto no código como nos dados pertinentes a um usuário, serão refletidos automaticamente em toda a estrutura até a camada de visualização. Dessa forma, a informação é entregue aos usuários de forma mais rápida e confiável.

Por fim, a aplicação proposta neste trabalho é a realização de um jogo que utiliza os recursos diferenciais deste *framework*. Os usuários podem jogar esta aplicação tanto sozinhos quanto contra oponentes reais, mesmo que estes estejam em ambientes diferentes. Como um exemplo, um usuário de uma plataforma móvel pode jogar contra um outro usuário que esteja utilizando um navegador. Sendo assim, esta proposta pode demonstrar o potencial do Meteor.js como uma solução para aplicações multiplataforma.

Palavras-Chave: Meteor.js; JavaScript; multiplataforma.

ABSTRACT

The present text conceptualizes the development of a multiplatform application in a JavaScript framework called Meteor.js. It literally works full-stack, from the rendering layer to the database communication layer, allowing the developer to use only one programming language throughout the whole development process.

Furthermore, Meteor.js has two main characteristics. The first one is the capacity of reusing the same source code. In order to do that, this framework utilizes a tool called Cordova, which is also developed using JavaScript. Therefore, the same application turns itself adaptable into any execution environment.

The second main feature is the capacity of delivering and updating data on the fly. That means that any user data or source code modification will be reflected throughout the whole structure of the framework until the visualization, automatically. In this way, the information is delivered to the corresponding users in a fast and reliable manner.

Finally, the purpose of this work is to develop a game that uses the features provided by this framework. The users may play this application alone or against a real opponent, even if they are in different platforms. As an example, a user that is using a mobile platform can play against a user that is running the same application in the web browser. Thus, this software can demonstrate the full potential of Meteor.js as a solution for multiplatform applications.

***Keywords*:** *Meteor.js; JavaScript; multiplatform*.

SUMÁRIO

LISTA DE SIGLAS

API	Application Programming Interface
AVD	Android Virtual Device
CSS	Cascade Style Sheet
DER	Diagrama Entidade-Relacionamento
HTML	HyperText Markup Language
IDE	Integrated Development Environment
JDK	Java Development Kit
JS	JavaScript
JSON	JavaScript Object Notation
MER	Modelo Entidade-Relacionamento
MIT	Massachusetts Institute of Technology
NoSQL	No Structured Query Language
PHP	Personal Home Page
SQL	Structured Query Language
UML	Unified Modeling Language
URL	Uniform Resource Locator

LISTA DE FIGURAS

1 INTRODUÇÃO

A cada dia, a tecnologia aumenta a necessidade de melhor desempenho e capacidade de adaptação, principalmente nos aplicativos *web* e *mobile*. De fato, a usabilidade e praticidade encontradas nessas aplicações enaltecem a preferência dos usuários. Por outro lado, o desenvolvimento destas soluções digitais pode ser dificultado quando há a necessidade de portabilidade, a qual muitas vezes acompanha a alteração da lógica e do código-fonte da aplicação. A razão pela qual estas mudanças são fundamentais é que plataformas diferentes podem possuir linguagens e arquiteturas distintas, requisitando conhecimento especializado para cada tipo de sistema operacional.

É comum definir a portabilidade como a solução para aplicações multiplataforma, porém a mesma solução pode acarretar novos problemas e dificuldades durante o processo de criação da aplicação. Com efeito, a variedade de ambientes de desenvolvimento *web* e *mobile* abre um leque de possibilidades para linguagens de programação que podem ser utilizadas. Sendo assim, a facilidade de reutilização do código-fonte, sem ou com mínima alteração do mesmo, irá definir a linguagem como multiplataforma. Por outro lado, a implementação do *software*, o qual deverá ser construido a partir de uma destas linguagens, em plataformas distintas, definirá o seu nível de portabilidade. Dessa forma, percebe-se que a capacidade de implementação do mesmo aplicativo em plataformas diferentes não está relacionada à sua facilidade e centralização de desenvolvimento em uma mesma linguagem de programação.

Hoje em dia, cada ambiente de desenvolvimento adota linguagens de programação diferentes para a implementação do código-fonte. Na Internet, por exemplo, PHP, ASP.NET e Python são utilizadas frequentemente. Em ambiente móveis, as linguagens variam de acordo com a plataforma utilizada (Android faz uso de Java, iOS faz uso de Swift, Windows Phone faz uso de C#, entre outros). Verificando a abrangência de opções, a seguinte pergunta é ressaltada: Como aplicar a portabilidade em aplicativos *web* e *mobile*, utilizando o mesmo código-fonte?

Todas essas plataformas partilham um recurso em comum, o qual permite que outra linguagem de programação seja utilizada: a linguagem JavaScript. Por sua natureza, JavaScript é muito utilizado para o aumento da interatividade de páginas *web* com os usuários, onde o código é totalmente executado no cliente. Entretanto,

com a expansão da *web*, o uso de *frameworks* específicos em JavaScript pode tornar esta linguagem mais poderosa e versátil tanto para o desenvolvimento de código executado no servidor quanto para aplicativos *mobile* em plataformas distintas.

Sendo assim, o objetivo deste trabalho é desenvolver uma aplicação multiplataforma em JavaScript quando atrelado ao *framework* Meteor.js, mostrando a capacidade de portabilidade de código dessa linguagem. Nos próximos capítulos, será abordado os conceitos, a forma de trabalhos e os recursos disponíveis neste *framework*, além de explicar como acontece a integração do Meteor.js quando há o formato multiplataforma da mesma aplicação. Por fim, os últimos capítulos serão destinados à explicação das etapas do processo de desenvolvimento da aplicação multiplataforma.

2 METEOR.JS

Segundo a documentação oficial (METEOR DOCS, 2016), Meteor.js é um conjunto de ferramentas (*framework*) que permite que programadores utilizem JavaScript em todas as camadas de desenvolvimento de uma aplicação *web*. Em outras palavras, Meteor.js apresenta bibliotecas para desenvolvimento *frontend* e *backend*, as quais são combinadas para a criação de aplicações confiáveis e robustas (VOGELSTELLER, 2015, p.8). De fato, Meteor.js ainda é uma tecnologia bem recente, lançada em 2012, e que tenta conter e executar toda a lógica de uma aplicação apenas utilizando JavaScript. No entanto, Meteor.js não pode ser confundido com outros *frameworks* da mesma linguagem, como jQuery e Angularjs, pois estes não providenciam uma estrutura abrangente categorizadas como *full stack*, similar ao *framework* Meteor.js (VOGELSTELLER, 2015, p.8).

2.1 FILOSOFIA

Meteor.js está sob a licença MIT (Massachusetts Institute of Technology), a qual caracteriza-o como um *framework* de código aberto. Seus repositórios estão localizados no GitHub e abertos à comunidade para a resolução de problemas e implementação de novas funcionalidades ao módulo principal do *framework* (METEOR, 2016). Além disso, Meteor.js possui nativamente outros projetos obrigatórios, os quais ajudam a providenciar a característica de *full stack* que o *framework* disponibiliza. Estes projetos são divididos em (METEOR, 2016):

- Bibliotecas: rotinas que auxiliam na renderização e atualização automática das informações contidas nas páginas;
- Ferramentas: programas que auxiliam na compilação e minimização de código;
- Padronizações: protocolos de comunicação e *websockets*;
- Serviços: utilizados para *download* de pacotes adicionais.

Além dos projetos nativos de Meteor.js, há também a existência de pacotes adicionais. De fato, Meteor.js disponibiliza uma plataforma online chamada *Atmospherejs* onde desenvolvedores podem consultar pacotes já existentes que fazem o trabalho de, por exemplo, autenticação, criação de rotas, testes de *software*, entre outros. O *download* destes pacotes é totalmente gratuito, e para realizá-lo, basta

apenas a execução de um comando. Logo após, Meteor.js irá adicionar o pacote junto aos demais pacotes já existentes na aplicação.

2.2 PRINCÍPIOS

De acordo com Robinson, Gray e Titarenco (2015, p. 27), sete princípios definem exatamente as características fundamentais do *framework* Meteor.js: *Data on the wire*, *One Language*, *Database everywhere*, *Latency compensation*, *Full stack reactivity*, *Embrace the ecosystem*, e *Simplicity equals productivity*.

O primeiro, *Data on the wire*, reflete o porquê transmitir HTML formatado pela rede quando podemos apenas transmitir os dados e deixar o cliente renderizá-los. Essa filosofia agrega a ideia de que a lógica de certos procedimentos do sistema deve ser deixada para o próprio cliente, fazendo com que o servidor tenha um tempo de resposta menor e uma maior disponibilidade de serviço para as requisições (Robinson, Gray e Titarenco, 2015, p. 27).

One Language, por sua vez, carrega a ideia de que Meteor.js é um *framework frontend* e *backend*, o que significa que ele utiliza a mesma linguagem (JavaScript) para o desenvolvimento de ambos os lados. Em outras palavras, não há necessidade de trocar de linguagens para que haja uma comunicação entre servidor e cliente. “Trocar de linguagem pode aumentar o carregamento cognitivo, o qual é difícil de ser gerenciado por desenvolvedores iniciantes e veteranos” (Robinson, Gray e Titarenco, 2015, p. 28).

O conceito de *Database everywhere*, por sua vez, critica técnicas utilizadas para a programação no banco de dados. Assim como manter duas linguagens diferentes, usar métodos diferentes para conectar a aplicação com o banco de dados pode tornar necessário a criação de mais código-fonte. Neste caso, “igualar o acesso ao banco de dados pode simplificar e acelerar o desenvolvimento” do código (Robinson, Gray e Titarenco, 2015, p. 28).

Latency compensation explora a velocidade necessária dos dias atuais. De fato, usuários querem respostas rápidas em suas requisições. Porém, uma latência alta pode ser o fator que fará usuários desistirem ou persistirem nas transações e nas interações com a aplicação. Para isso, Meteor.js possui funções que simulam transações em tempo real enquanto a execução verdadeira ainda é executada, pareando os dados do cliente e servidor em *background* (Robinson, Gray e Titarenco, 2015, p. 28).

Full stack reactivity reflete sobre a reatividade da informação. Fazer com que os dados cheguem à todos os clientes interessados pode ser difícil quando não há um gerenciamento bem detalhado nas camadas de desenvolvimento da aplicação. “Reatividade torna as coisas mais simples” (Robinson, Gray e Titarenco, 2015, p. 29). A partir da modificação de um ou mais dados nos clientes, todos os outros interessados relacionados àquele dado receberão a atualização automaticamente.

Embrace the ecosystem aponta a reutilização de código. Como Meteor.js utiliza Node.js como sua base para execução, a ideia de colaboratividade, integrando partes existentes à sua composição nativa em vez de trocar e modificar códigos já criados, enaltece o aumento da produtividade na criação do código-fonte (Robinson, Gray e Titarenco, 2015, p. 29).

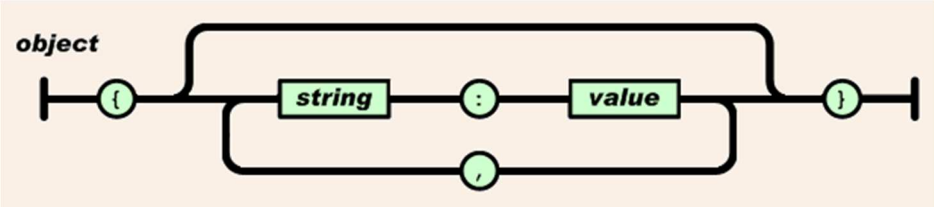
Por fim, *Simplicity equals productivity* mostra que a simplicidade é um sinônimo da produtividade. Sendo assim, Meteor.js acredita que sua facilidade de uso seja a razão ideal para o engajamento no desenvolvimento de aplicações (Robinson, Gray e Titarenco, 2015, p. 29).

2.3 ARMAZENAMENTO E ATUALIZAÇÃO DE DADOS

De acordo com Vogelsteller (2015, p. 44), MongoDB é o único banco de dados cem por cento suportado por aplicações criadas com o *framework* Meteor.js. MongoDB é um sistema gerenciador de banco de dados, de código-fonte aberto, que é baseado em uma estrutura chamada NoSQL. Robinson, Gray e Titarenco (2015, p. 73) explicam que banco de dados como MySQL e PostgreSQL utilizam uma estrutura de tabelas que facilita o relacionamento dos dados. “Estes bancos de dados relacionais são similares a uma planilha com várias abas, onde cada tipo de dado é armazenado na linha e coluna correspondente” (ROBINSON, GRAY e TITARENCO, 2015, p. 73). Entretanto, estruturas NoSQL são não-relacionais, o que significa que este tipo de banco de dados utiliza uma cadeia de encaixes que facilita relacionamentos entre os dados. No caso do MongoDB, este formato específico é o JSON.

Há duas formas distintas de se apresentar dados neste formato. O primeiro é chamado de objeto, onde se tem chaves e valores unidos pelo símbolo : (dois pontos), sendo que cada par deste conjunto é seguido pelo símbolo , (vírgula), e todo o conjunto é envolvido pelos símbolos { e } (chaves de abertura e fechamento), como mostra a Figura 1.

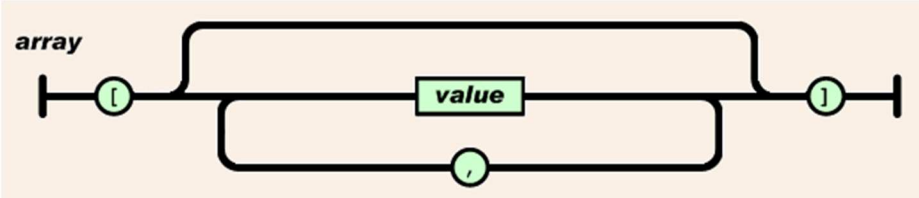
Figura 1: Representação de um objeto em JSON



Fonte: Documentação oficial do JSON [1]

O segundo é chamado vetor, onde um conjunto de valores são separados pelo símbolo , (vírgula) e envolvidos pelos símbolos [e] (colchetes de abertura e fechamento), como demonstrado na Figura 2.

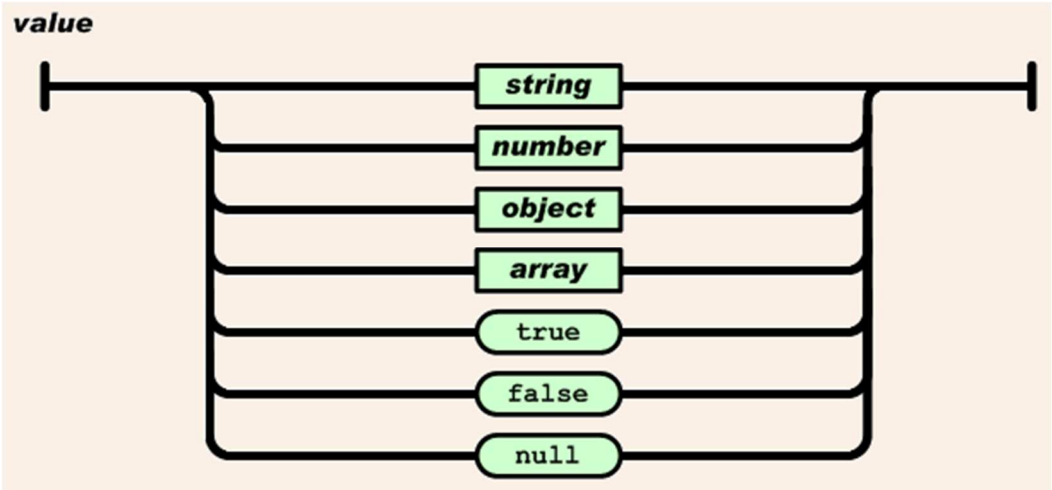
Figura 2: Representação de um vetor em JSON



Fonte: Documentação oficial do JSON [1]

Para finalizar, cada valor pode variar de acordo com um grupo de variáveis, “podendo ser uma cadeia de caracteres, um número, *true*, *false*, null, um objeto ou uma array”, como ilustrado na Figura 3.

Figura 3: Opções de valores em um JSON



Fonte: Documentação oficial do JSON [2]

[1] Disponível em <http://www.json.org/json-pt.html> Acesso em: 17 abr. 2014.
[2] Disponível em <http://www.json.org/json-pt.html> Acesso em: 17 abr. 2014.

MongoDB, em específico, usa uma estrutura de dados chamada *Collection*, a qual faz uso de documentos no formato JSON que possuem identificadores únicos (VOGELSTELLER, 2015, p. 44).

Um arquivo armazenado neste formato possui “uma formatação leve de troca de dados. Para seres humanos, é fácil de ler e escrever. Para máquinas, é facil de interpretar e gerar” (JSON.ORG, 2016). Sendo assim, há a possibilidade de aninhamento que pode criar uma cadeia de valores dentro de um mesmo valor, como demonstrado na Figura 4.

Figura 4: Exemplo de uma *Collection* no formato JSON

```
{
  "_id": "W7sBzpBbov48rR7jW",
  "myName": "My Document Name",
  "someProperty": 123456,
  "aNestedProperty": {
    "anotherOne": "With another string"
  }
}
```

Fonte: Vogelsteller (2015, p. 44).

No entanto, este formato também influencia na desnormalização de dados, que é uma técnica onde o mesmo dado, ou conjunto de dados, é armazenado em vários lugares. Apesar deste aspecto colaborar com o aumento da redundância, ele também permite manipular e recuperar dados relacionados com uma única operação. Além disso, caso seja necessário, MongoDB também permite o uso de técnicas de normalização de dados, a qual acontece quando apenas uma referência do dado, ou conjunto de dados, é armazenada em vários locais (MONGODB DOCS, 2016).

Por fim, MongoDB é iniciado tanto no servidor quanto no cliente da aplicação, o qual receberá uma cópia do banco de dados chamada Minimongo. Segundo Vogelsteller (2015, p. 61), o Minimongo é uma instância que roda completamente na memória do cliente e tem funções originais reduzidas. Esta abordagem introduzida pelo Meteor.js é uma das características do *database everywhere*, que é um conceito de que a aplicação do banco de dados deve ser executada em ambos os ambientes para manter uma conexão e atualização de dados constantemente.

2.3.1 *Publication/Subscription*

Outra abordagem é adotada pelo Meteor.js, chamada *publication/subscription*. De acordo com Vogelsteller (2015, p. 61), toda vez que um usuário conecta-se ao servidor, Meteor.js realiza o *download* dos dados pertinentes e armazena-os no cliente, podendo mandá-los para a visualização, caso seja necessário. Assim sendo, quando um cliente atualiza um ou mais dados, todos os demais clientes que também possuem o mesmo conjunto de dados terão suas informações locais atualizadas no Minimongo automaticamente, podendo ser mostradas na visualização em tempo real.

2.4 PARTICULARIDADES

Um dos recursos particulares deste *framework* é uma técnica chamada *hot code pushes*. Vogelsteller (2015, p.88) explica que o *hot code push* acontece quando há uma mudança nos arquivos de código-fonte da aplicação e enviada para os clientes automaticamente. Sendo assim, quando o Meteor.js executado no cliente verifica a modificação enviada pelo servidor, ele recarrega a página sem perder os dados já existentes em formulários e sessões. Isto é, o usuário não percebe que houve um recarregamento e pode continuar a interagir com a aplicação normalmente.

Outra característica singular sobre o Meteor.js é a reatividade. De acordo com Robinson, Gray e Titarenco (2015, p. 44), “Reatividade é a habilidade que o *template* tem de detectar e refletir modificações dos dados subjacentes automaticamente”. Em outras palavras, a visualização da página sempre irá mostrar os dados atualizados pertinentes à ela. Com isso, a confiabilidade e consistência da apresentação das informações são realçadas com as atualizações constantes apresentadas nesta técnica. Além disso, é válido afirmar que as abordagens de armazenamento e atualização de dados citadas na seção 2.3, como *database everywhere* e *publication/subscription*, podem ser consideradas extensões desta técnica, pois ao atualizar o banco de dados, a visualização se encarregará de assegurar que as novas informações serão mostradas aos usuários pertinentes.

2.5 INSTALAÇÃO

Como Meteor.js é um *framework* com raízes no Node.js, ele precisa ser instalado no servidor para ter seu funcionamento completo. Sucintamente, Node.js é um motor JavaScript que tem o intuito de “construir aplicações de rede rápidas e

escaláveis [...], ideal para aplicações em tempo real com troca intensa de dados através de dispositivos distribuídos” (MOREIRA, 2013).

Atualmente, Meteor.js é suportado nos três principais sistemas operacionais no mercado: Windows, Linux e Mac (VOGELSTELLER, 2015, p.10). Segundo a documentação oficial (METEOR DOCS, 2016), para a instalação do *framework* nas plataformas Linux e Mac, apenas a execução de um comando no terminal é necessária. Em plataformas Windows, a instalação acontece através de um instalador oficial que pode ser baixado diretamente do site do Meteor.js.

Quanto à instalação de plugins e pacotes adicionais criados pela comunidade, a plataforma online chamada Atmospherejs funciona como um repositório de código-fonte do próprio Meteor.js. Dessa forma, qualquer pessoa pode navegar e procurar por soluções prontas e instalá-las diretamente no projeto utilizando apenas uma linha de comando que o próprio plugin ou pacote disponibiliza.

3 INTEGRAÇÃO MULTIPLATAFORMA

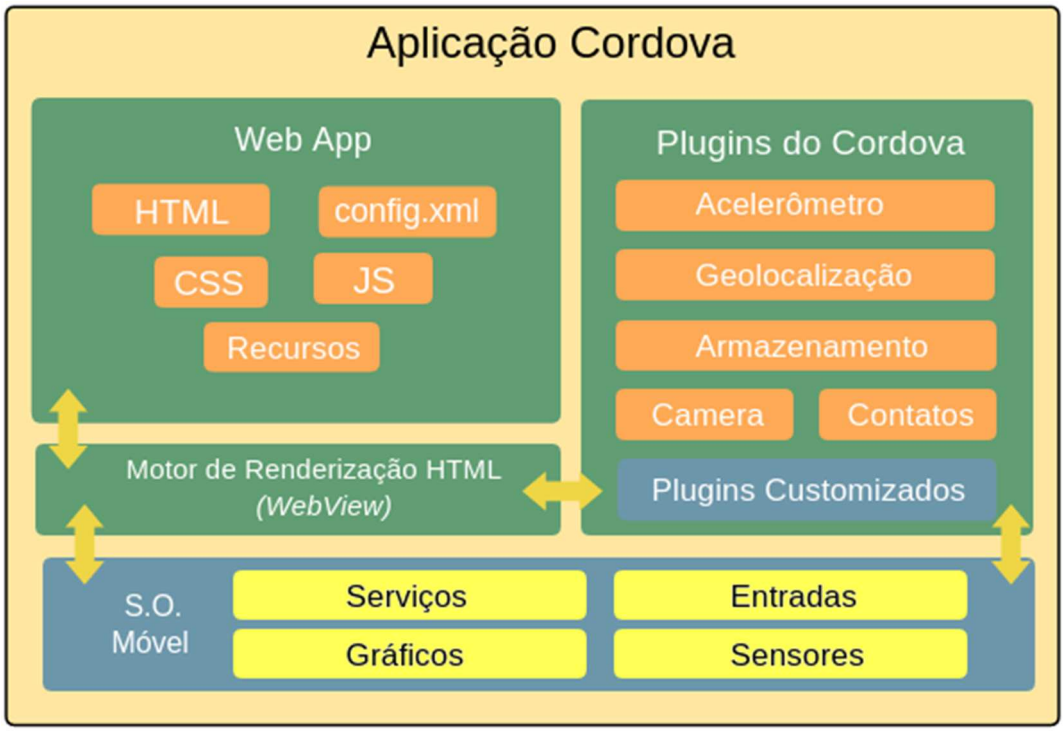
Uma das principais particularidades do Meteor.js é a capacidade de executar o mesmo código-fonte em uma plataforma diferente, sem a necessidade de modificar a programação realizada previamente. Para isso, este *framework* possui uma integração com um projeto de código aberto chamado Cordova, o qual permite que a mesma aplicação seja executada em dois sistemas operacionais móveis distintos: iOS e Android (METEOR DEVELOPERS, 2016).

Basicamente, “um aplicativo Cordova é uma aplicação *web* escrita utilizando HTML, CSS e JavaScript usualmente, mas é executada em uma *Web View* acoplada em um aplicativo nativo em vez de um navegador móvel comum” (METEOR DEVELOPERS, 2016). Em outras palavras, Cordova permite que elementos comuns na programação para *web* sejam aproveitados no desenvolvimento de um aplicativo móvel. Além disso, recursos do Meteor.js como *hot code pushes* também podem ser utilizados, fazendo com que as atualizações do aplicativo aconteçam em tempo real.

É importante destacar que, mundialmente, o Cordova também é conhecido como PhoneGap. Wargo (2012, p. 4) aponta que o projeto PhoneGap, o qual apresenta muitas similaridades com o Cordova, foi adquirido pela Adobe em 2011. No entanto, “quando a Adobe doou o código para a fundação Apache em 2012 com o intuito de assegurar um modelo de governança mais aberto, o projeto foi renomeado para Cordova” (METEOR DEVELOPERS, 2016). Atualmente, há várias distribuições diferentes deste mesmo projeto, porém todas elas apresentam a mesma base estrutural, o que facilita a integração de extensões e plugins voltados para este tipo de projeto multiplataforma.

A Figura 5 ilustra detalhadamente a arquitetura geral do Cordova, dividindo as camadas de recursos web disponíveis, renderização de conteúdo, plugins que permitem a comunicação do aplicativo híbrido com o sistema operacional móvel, e recursos nativos da plataforma.

Figura 5: Diagrama de arquitetura do Cordova



Fonte: Página da documentação do Cordova[3]

3.1 APLICATIVOS HÍBRIDOS E NATIVOS

De acordo com Kohan e Montanez (2015), aplicativos nativos são aqueles desenvolvidos nas linguagens de programação nativas das plataformas. Como exemplo, aplicativos nativos são desenvolvidos em linguagens como Java, Swift e C#, e compilados para Android, iOS e Windows Phone respectivamente. Além disso, Whinnery (2012) complementa que aplicações nativas tem total acesso aos componentes internos de *hardware* e *software* dos dispositivos, obtendo um nível de desempenho maior. No entanto, Gathol e Patel (2012, p. 9-10) alertam algumas dificuldades na criação de aplicativos deste tipo, como a necessidade de se ter equipes distintas de acordo com a plataforma, e o desafio para se obter *interfaces* similares entre plataformas distintas. Tais dificuldades podem ter impacto no aumento do custo do desenvolvimento destas aplicações.

Por outro lado, aplicativos híbridos utilizam tecnologias *web* (HTML5, CSS3 e JavaScript) e *frameworks* de apoio em um único pacote, como é o caso do Cordova no Meteor.js. O código, então, é executado dentro de um componente chamado *Web*

[3] Adaptado e traduzido pelo autor. Imagem original disponível em <https://cordova.apache.org/docs/en/latest/guide/overview/> Acesso em 10 out. 2016.

View, o qual simula um navegador responsável por renderizar o conteúdo de páginas *web* em um aplicativo nativo da plataforma e segue fielmente o comportamento de qualquer outra aplicação que não seja híbrida (WARGO, 2012, p. 7). Sendo assim, há a possibilidade do mesmo código ser executado em plataformas distintas, contanto que elas possuam este mesmo componente.

É importante ressaltar que o componente *Web View* se assemelha com o navegador nativo da plataforma, mas não pode ser considerado idêntico. De fato, um navegador não consegue utilizar outros itens de *hardware* e *software* que estão disponíveis no dispositivo. Por outro lado, o componente *Web View* pode se comunicar com os demais, uma vez que este apenas reproduz a capacidade de renderização de páginas *web* em um aplicativo nativo (WARGO, 2012, p. 7-8).

3.2 TECNOLOGIAS SIMILARES

Assim como o Cordova, outros *frameworks* se arriscam a atingir o mesmo objetivo geral: criar aplicativos multiplataforma. Dentre eles, podemos destacar os outros dois mais conhecidos no mercado: Appcelerator Titanium e Xamarin.

3.2.1 Appcelerator Titanium

Appcelerator Titanium, ou apenas Titanium, é um *framework* similar que, de certa forma, efetua o mesmo trabalho do Cordova, utilizando tecnologias *web* (WARGO, 2012, p. 20). Entretanto, há diferenças importantes entre estas duas soluções.

Segundo Whinnery (2012), Cordova e PhoneGap focam-se em criar aplicativos híbridos, enquanto Titanium concentra-se na criação de aplicações nativas da plataforma. Apesar de ambas as soluções utilizarem tecnologias *web*, Titanium apenas faz o uso da linguagem JavaScript para desenvolver suas aplicações. Já as camadas de interação com o usuário, por sua vez, são desenvolvidas em linguagens que seguem metodologias e convenções do próprio *framework*. Este fator pode dificultar o uso desta solução como um todo, uma vez que há a necessidade de aprendizado de uma linguagem completamente nova, a qual será utilizada exclusivamente pelo Titanium.

Além disso, Whinnery (2012) complementa que o Titanium é uma proposta de reuso de código em plataformas distintas, baseando-se em componentes nativos de cada uma delas para proporcionar uma melhor experiência ao usuário final. Em outras

palavras, o código-fonte desenvolvido em Titanium é traduzido e compilado para as plataformas suportadas, sendo instalado como um aplicativo nativo em vez de híbrido. Isso o difere do Cordova e PhoneGap, onde os aplicativos são interpretados em tempo real pela *Web View* (WARGO, 2012, p. 7).

No entanto, Whinnery (2012) também alerta que, como componentes e seus métodos de uso variam entre uma plataforma e outra, a portabilidade do código programado pode chegar a noventa por cento em casos otimistas, o que significa que ajustes específicos devem ocorrer, dependendo do recurso utilizado na aplicação na plataforma desejada.

3.2.2 Xamarin

De acordo com Dickson (2013, p. 7), Xamarin é um *framework* que auxilia no desenvolvimento de aplicativos multiplataforma, mas não foca todas as suas atenções à portabilidade de código. De fato, “cada implementação da mesma aplicação é desenvolvida independentemente entre as plataformas suportadas, para que as *interfaces* nativas sejam mantidas”, preservando a experiência do usuário com a plataforma (DICKSON, 2013, p. 8). Mesmo assim, Xamarin consegue reutilizar partes do código-fonte que são consideradas gerais, ou seja, independentemente da plataforma, a lógica para atingir o objetivo desejado ainda é a mesma. Estas partes do código estão relacionadas à comunicação com o servidor, regras de negócio, acesso a banco de dados, entre outros. Dickson (2013, p. 8) ainda afirma que até setenta e cinco por cento do código-fonte pode ser reutilizado. Logo após o desenvolvimento do aplicativo, o código-fonte é compilado para a plataforma desejada.

Adicionalmente, Xamarin não utiliza ferramentas *web* para o desenvolvimento de aplicativos. Em vez disso, apenas a linguagem C# é necessária, o que pode dificultar a aceitação de desenvolvedores *web* que teriam que aprender outra linguagem de programação para utilizar os recursos do *framework* (DICKSON, 2013, p. 7).

3.3 CORDOVA E METEOR.JS

De acordo com DeBergalis (2014), a versão 0.9.2 do Meteor.js foi a primeira a oferecer suporte ao Cordova, podendo-se gerar aplicativos para iOS e Android. A partir disso, com poucos comandos, é possível adicionar as funcionalidades do Cordova através de pacotes nativos do Meteor.js e acessar vários componentes do dispositivo móvel, tanto *hardware* como *software*.

Para realizar a checagem de qual ambiente o código-fonte está sendo executado, Meteor.js providencia uma variável específica que identifica se o ambiente atual de execução da aplicação é o Cordova ou não, como demonstrado na Figura 6. Sendo assim, todo o código-fonte referente a um aplicativo móvel pode ser desenvolvido dentro de uma estrutura condicional que verifica o tipo do ambiente correspondente (METEOR DOCS, 2016).

Figura 6: Verificação do ambiente de execução

```
if (Meteor.isCordova) {
  console.log("Printed only in mobile cordova apps");
}
```

Fonte: Documentação oficial do Meteor.js [4]

Além disso, funcionalidades padrões do Meteor.js como *hot code pushes* também são suportadas nos aplicativos móveis. Em outras palavras, há também a possibilidade de baixar um aplicativo em lojas virtuais como Apple Store e Google Play Store, realizar uma mudança no código-fonte no servidor e, logo após, atualizá-lo nos clientes sem a necessidade de reinicialização do aplicativo (DEBERGALIS, 2014).

A instalação do recurso Cordova em uma aplicação Meteor.js acontece através de algumas linhas de comando (METEOR DEVELOPERS, 2016), sendo elas respectivamente:

- *meteor add-platform ios*|*android*: Adiciona uma plataforma móvel à uma aplicação Meteor;
- *meteor run ios*|*android*: Constrói e executa a aplicação em uma das plataformas adicionadas.

No entanto, é necessário completar alguns pré-requisitos antes de executar estes comandos. No caso da plataforma Android, é necessário a instalação do JDK e

[4] Disponível em <http://docs.meteor.com/#/full/> Acesso em: 25 mar. 2016.

da configuração de um AVD, o qual irá simular o sistema operacional móvel. Já no caso da plataforma iOS, apenas os usuários do sistema operacional Macintosh podem realizar este procedimento. Além disso, estes também devem possuir um ambiente de desenvolvimento integrado chamado Xcode com a versão 7.2, no mínimo (METEOR DEVELOPERS, 2016).

4 DESENVOLVIMENTO DE UMA APLICAÇÃO

4.1 PROPOSTA

Para que haja uma análise mais detalhada do nível de portabilidade de código em aplicações desenvolvidas em Meteor.js, é necessária a criação de um aplicativo que possa ser executado em diferentes plataformas. Como foco, estas plataformas serão *web* e Android.

A ideia de aplicativo a ser desenvolvido é um jogo chamado Hog. Este jogo foi uma proposta do professor John DeNero, instrutor da matéria CS61A de Estrutura e Interpretação de Programas de Computador, no ano de 2012, na Universidade Berkeley da Califórnia.

Originalmente, Hog é um jogo de dados onde dois jogadores disputam quem alcança cem pontos primeiro. Em cada turno, o jogador atual escolhe uma quantidade de até dez dados de seis lados para lançar. A pontuação do turno será igual ao resultado da soma dos dados, a qual será somada com a pontuação total do jogador. Entretanto, quando um ou mais dados resultarem 1, a pontuação do turno será de apenas 1, não importando a soma dos demais dados (DENERO, 2012).

Além disso, DeNero (2012) propõe a adição de três regras adicionais, as quais são:

- *Free Bacon:* Se um jogador escolher jogar 0 dados, a sua pontuação do turno será igual ao número da casa decimal da pontuação do oponente, mais a adição de um ponto (Ex.: Se o jogador oponente tiver 54 pontos e o jogador atual lançar 0 dados, a pontuação do jogador atual naquela rodada será 6, ou seja, 5+1);
- *Hog Tied*: Caso a soma da pontuação de ambos os jogadores termine com o número 7 (Ex.: 37, 47, 57), então, o jogador do turno atual só poderá lançar até 1 dado;
- *Hog Wild*: Caso a soma da pontuação de ambos os jogadores for um múltiplo de sete (Ex.: 7, 14, 21, 28), então, em vez dos dados do jogador do turno terem seis lados, eles terão apenas quatro lados.

Adicionalmente, há a possibilidade de escolha entre dois modos de jogo: *Singleplayer* e *Multiplayer*. O modo *Singleplayer* permite que o jogador jogue sozinho, contra uma inteligência artificial, sem a necessidade de uma conexão com a Internet caso o jogo esteja rodando em uma plataforma *mobile*. Por outro lado, o modo

Multiplayer permite que o jogador jogue contra um outro jogador em tempo real. Nesta última opção, o jogador anfitrião deverá enviar um código alfanumérico para o jogador convidado, o qual será utilizado para identificar e sincronizar os dados de ambos os jogadores durante a partida.

4.2 FERRAMENTAS

Para o desenvolvimento dessa aplicação, foi feito o uso de ferramentas que facilitem este processo:

- *Frameworks* JavaScript: Meteor.js 1.3, MongoDB 2.0 e Cordova 2.4;
- Sistemas operacionais utilizados: Windows 8, 10, e Ubuntu 16;
- IDE de desenvolvimento: PHPStorm 2016;
- Desenvolvimento de Diagramas: Astah Community 7.0.0.

Além disso, o uso de alguns pacotes terceiros do Meteor.js, os quais podem ser encontrados pela plataforma Atmospherejs, também serão primordiais no processo de construção desta aplicação.

- *twbs:bootstrap*: *Framework* responsável por ajustar o *layout* da aplicação em diferentes resoluções de tela;
- *fastclick*: Plugin que permite que os eventos disparados pelo toque em dispositivos *mobile* sejam processados mais rapidamente;
- *iron:router*: Responsável por gerenciar os caminhos de rotas disponíveis, tanto no *frontend* como no *backend*;
- *momentjs:moment*: Biblioteca responsável por melhorar a interação com datas e horas no JavaScript;
- *standard-minifier-css*: Reduz e compacta arquivos CSS para melhorar o desempenho na renderização de conteúdo;
- *standard-minifier-js*: Reduz e compacta arquivo JS para melhorar o desempenho no processamento de regras de negócio.

4.3 DIAGRAMAS UML

De acordo com Guedes (2014, p. 15), UML (*Unified Modeling Language* ou Linguagem de Modelagem Unificada) tem como objetivo facilitar a modelagem e visualização do sistema como um todo ou em parte. “Essa linguagem se tornou, nos últimos anos, a linguagem-padrão de modelagem de software adotada internacionalmente pela indústria de Engenharia de Software.”

Na UML, há diversos diagramas que podem ser utilizados para o fornecimento de visualizações gerais e específicas do sistema. De fato, “alguns diagramas enfocam o sistema de forma mais geral, apresentando uma visão externa do sistema [...] ao passo que outros oferecem uma visão de uma camada mais profunda do software” (GUEDES, 2014, p.16-17). Além disso, há a possibilidade de falhas no sistema serem encontradas a partir dos diagramas criados, facilitando a correção destes erros antes mesmo da fase de desenvolvimento (GUEDES, 2014, p. 17).

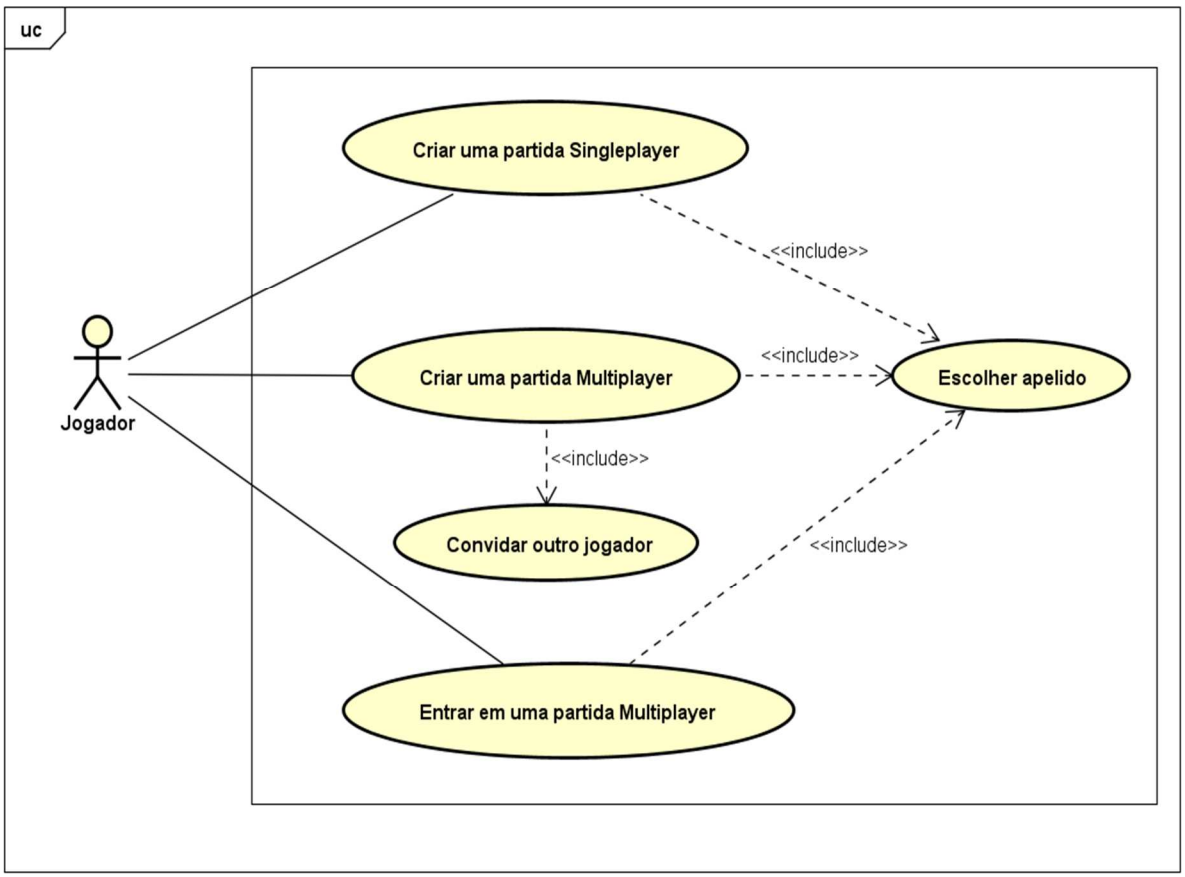
Nesta proposta de desenvolvimento foi realizado o uso de três diagramas distintos, sendo eles: Casos de Uso, Classe e Atividades, com o intuito de expor as funcionalidades do sistema em multiplas perspectivas.

4.3.1 Casos de Uso

Segundo Guedes (2014, p. 17), “o Diagrama de Casos de Uso apresenta uma linguagem simples e de fácil compreensão para que os usuários possam ter uma ideia geral de como o sistema irá se comportar”.

Para o cenário proposto, há a diagramação de dois casos de uso distintos. Em um primeiro cenário, ilustrado pela Figura 7, há as possibilidades que o jogador poderá optar antes de começar a jogar. No processo de criação da partida, o jogador poderá optar entre criar uma partida *Singleplayer*, entrar em uma partida *Multiplayer* (ou seja, ser o jogador convidado em uma partida), ou criar uma partida *Multiplayer*. Caso o jogador escolha a última opção, ele deverá convidar um outro jogador logo em seguida. Após a escolha do modo de jogo, independente da opção selecionada, o jogador deverá escolher o apelido que ele usará durante a partida.

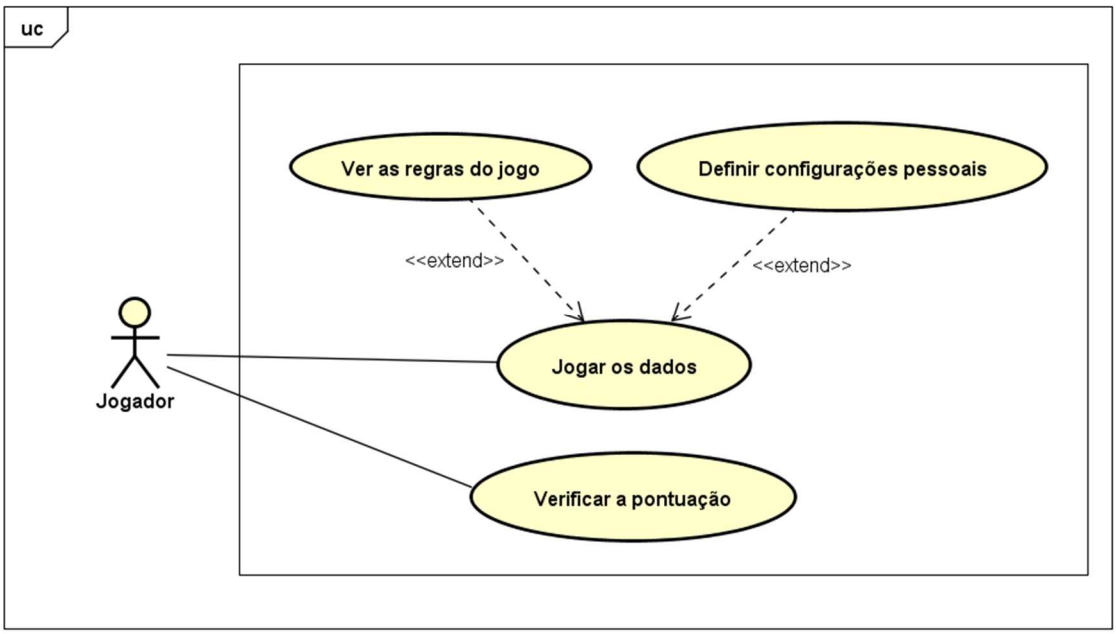
Figura 7: Diagrama de caso de uso ao criar uma partida



Fonte: elaborado pelo autor

Em um segundo cenário, as possibilidades de escolha do jogador são menores que o anterior. De fato, não há muitas etapas durante a jogada do turno, pois o objetivo é somente acrescentar mais pontos à pontuação geral. Nesta etapa, como ilustrado na Figura 8, o jogador deverá optar apenas por rolar os dados. No entanto, caso o jogador deseje, ele poderá verificar sua pontuação, verificar as regras do jogo e redefinir as configurações do idioma.

Figura 8: Diagrama de caso de uso ao jogar um turno



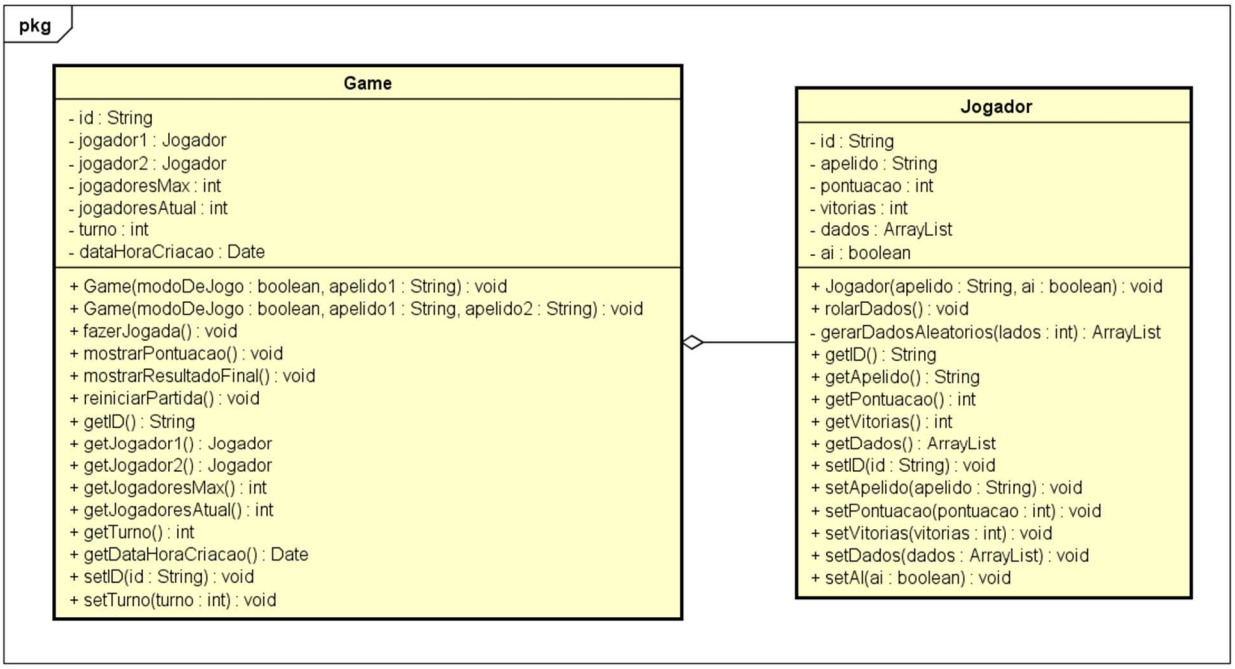
Fonte: elaborado pelo autor

4.3.2 Classe

Este tipo de diagrama tem como objetivo expor as classes, suas ligações, atributos e métodos utilizados no sistema, “servindo de apoio para a maioria dos outros diagramas” (GUEDES, 2014, p. 18-19). No caso proposto, apenas um diagrama será necessário para expor a ligação dos elementos que irão compor o jogo como um todo.

No diagrama da Figura 9, apenas duas classes são necessárias para o total gerenciamento do jogo: Game e Jogador. Em um primeiro momento, a classe Game tem o dever de gerenciar as jogadas, partidas e turnos do jogo. De fato, ela também é responsável por intermediar a comunicação dos jogadores em uma partida *Multiplayer*, onde o processo de espera durante uma jogada e outra dependerá exclusivamente da interação de jogadores reais com o jogo propriamente dito. Por outro lado, a classe Jogador tem como objetivo conter e gerenciar atributos exclusivos de cada jogador, como apelido, pontuação atual, número de vitórias, entre outros. Além disso, em caso de partidas *Singleplayer*, essa mesma classe também será capaz de gerenciar jogadas automáticas, simulando um jogador real.

Figura 9: Diagrama de classe da aplicação



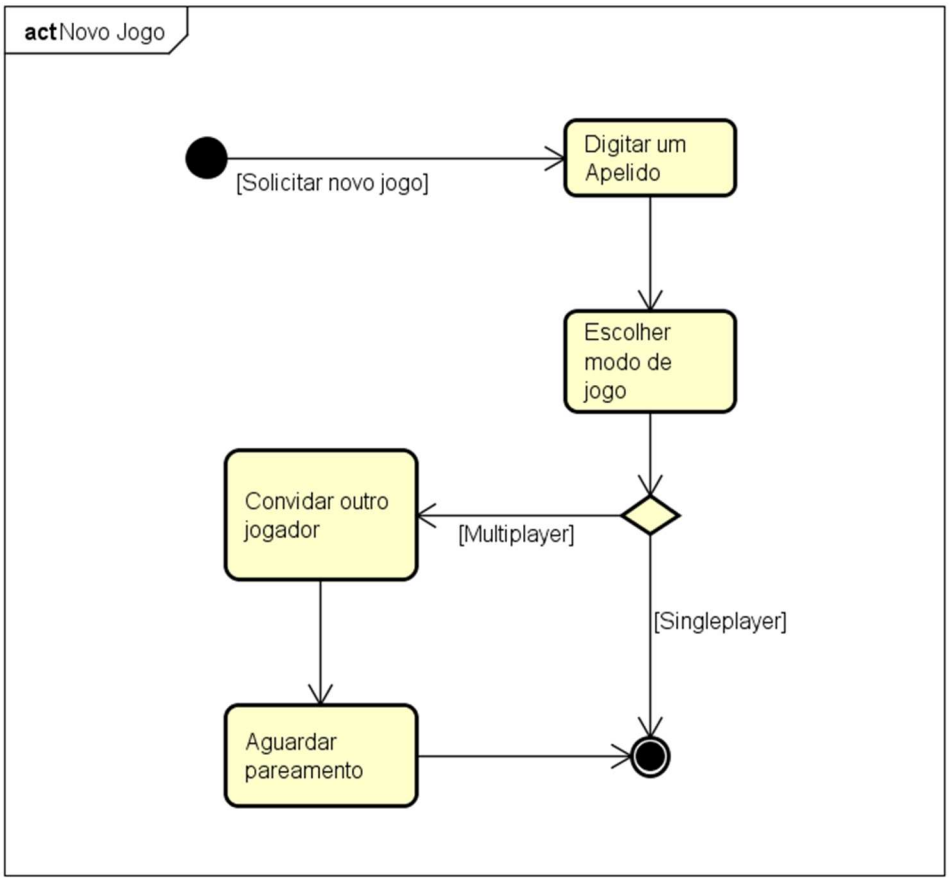
Fonte: elaborado pelo autor

4.3.3 Atividade

De acordo com Guedes (2014, p. 24), o objetivo do Diagrama de Atividade é expor todas as etapas do caminho a ser percorrido para a resolução de determinado processo dentro de um sistema. Para o cenário proposto, três diagramas ajudam a descrever melhor o fluxo de atividades realizadas pelo jogo.

O primeiro diagrama, ilustrado na Figura 10, representa a criação de uma nova partida. Neste sentido, quando há a solicitação de novo jogo feita por um jogador, o primeiro processo realizado é a escolha do seu apelido, seguida pela escolha da modalidade de partida. A partir disso, as opções se dividem. Caso o jogador decida jogar em modo *Singleplayer*, a atividade é finalizada instantaneamente. Por outro lado, se o modo *Multiplayer* for selecionado, o jogador deverá enviar o código alfanumérico de pareamento para um jogador convidado e aguardar pela conexão definitiva.

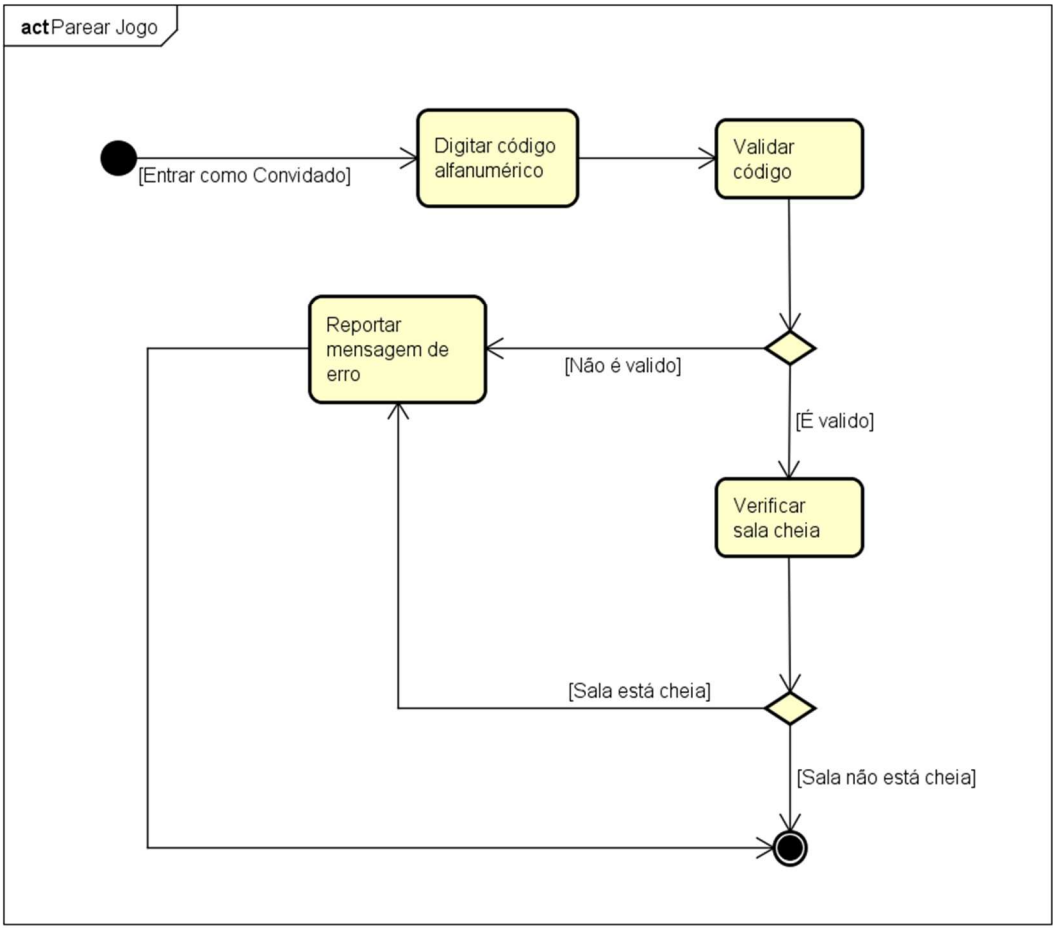
Figura 10: Diagrama de atividade de uma nova partida



Fonte: elaborado pelo autor

Após o recebimento do código alfanumérico enviado pelo jogador anfitrião, o jogador convidado deverá entrar na partida de acordo com a atividade ilustrada na Figura 11. Em um primeiro momento, o jogador deverá digitar o código alfanumérico que lhe foi enviado na tela inicial do jogo. Após a inserção, ocorrerá a validação deste código. Caso inválido, uma mensagem de erro será reportada para o jogador convidado. Do contrário, haverá uma segunda verificação, a fim de analisar se a partida referente ao código digitado possui apenas o jogador anfitrião. Esta segunda verificação é importante para assegurar que cada partida tenha apenas dois jogadores. Se esta análise apontar que a partida já possui um jogador convidado, então uma mensagem de erro será reportada. Caso contrário, o pareamento será efetuado e a partida terá início.

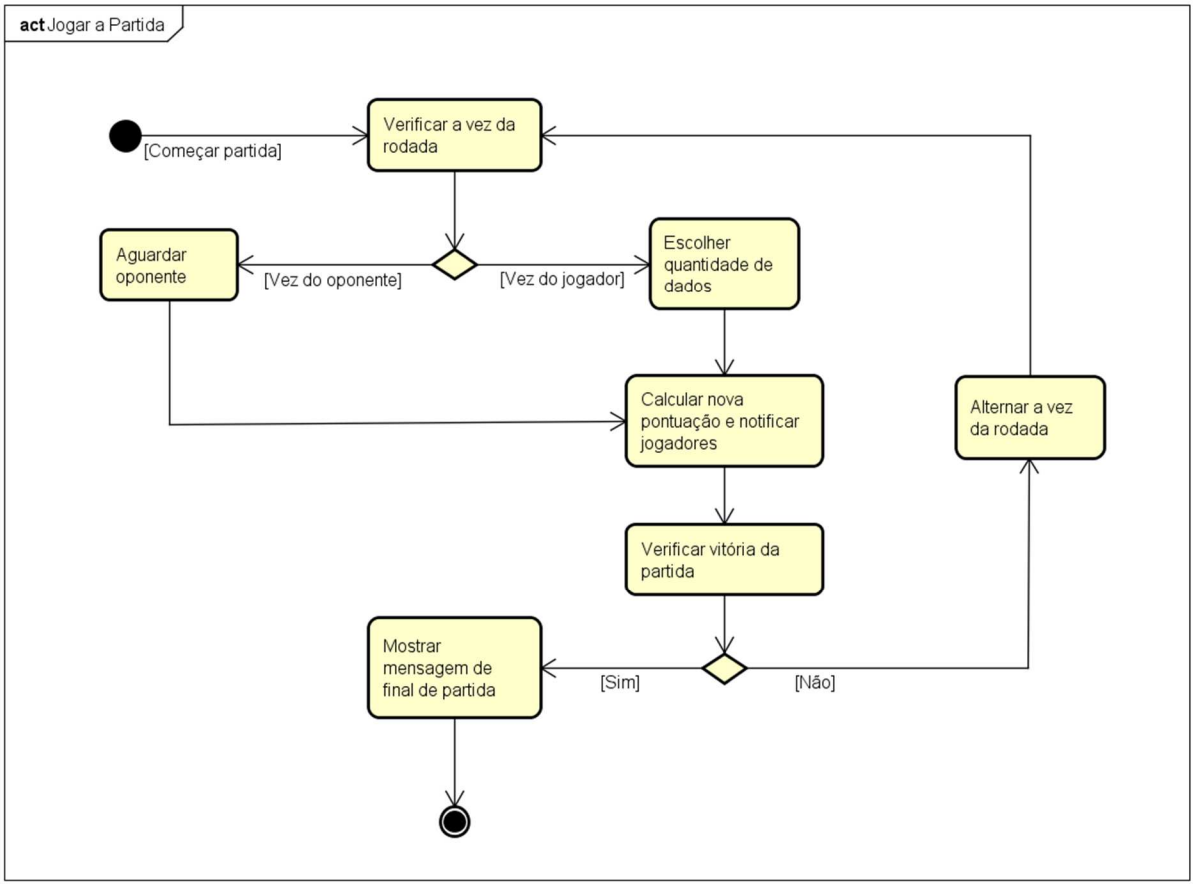
Figura 11: Diagrama de atividade de uma sincronização *multiplayer*



Fonte: elaborado pelo autor

Por último, ilustrada na Figura 12, a atividade para jogar a partida se assemelha com uma repetição. De início, verifica-se o dono do turno da rodada. Caso o turno seja do oponente, o jogador atual deverá aguardar até que a rodada seja concluída. Do contrário, este deverá escolher a quantidade de dados que serão lançados. Após esta etapa, há a contagem de pontos de ambos os jogadores, os quais são notificados com a nova pontuação. Por fim, há a verificação de um jogador vitorioso com base nos pontos totais. Caso haja, ambos recebem uma mensagem que notifica o fim da partida. Do contrário, alterna-se o dono da rodada e o processo se inicia novamente.

Figura 12: Diagrama de atividade de uma partida em andamento



Fonte: elaborado pelo autor

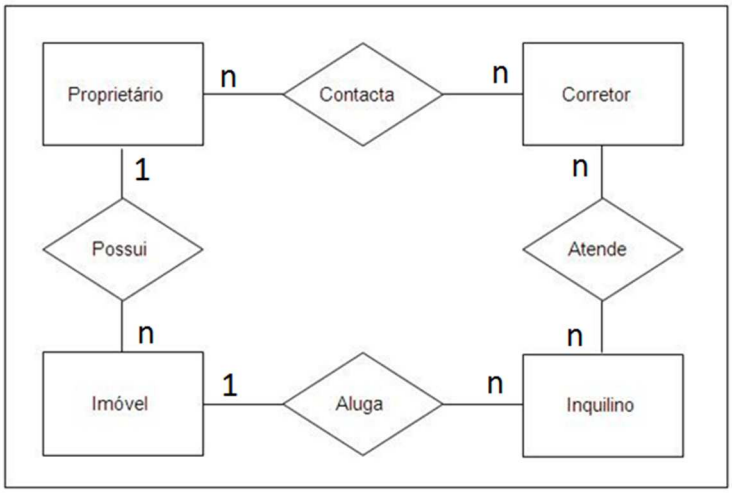
4.4 BANCO DE DADOS

Geralmente, são utilizados o MER e o DER para ilustrar e modelar a estrutura de um banco de dados relacional. De acordo com Rodrigues (2014), o MER “é um modelo conceitual [...] para descrever os objetos (entidades) envolvidos em um domínio de negócios, com suas características (atributos) e como elas se relacionam entre si (relacionamentos)”, enquanto que o DER tem como objetivo representar a proposta do MER graficamente. De fato, algumas regras específicas devem ser seguidas para a criação de um diagrama, sendo elas:

- Entidades (objetos) são representados por losangos;
- Características (atributos) são representados por elipses;
- Relacionamentos são representados por linhas, contendo uma identificação do grau de relação.

Estas regras, quando aplicadas conjuntamente à necessidade do banco de dados projetado, formam um diagrama similar ao exemplo ilustrado na Figura 13:

Figura 13: Exemplo de um diagrama para um sistema de imobiliária



Fonte: RODRIGUES, 2014 (acesso em: 27 abr. 2016).

Para o desenvolvimento da proposta do jogo, duas tabelas serão necessárias para armazenar os dados. Enquanto uma delas é responsável por manter as informações das partidas, a outra mantém os dados dos jogadores. Sendo assim, um diagrama entidade-relacionamento pode ser construido, como demonstrado na Figura 14.

Figura 14: Diagrama de entidade-relacionamento da aplicação

Fonte: elaborado pelo autor

No entanto, três motivos influenciam na desnormalização destas duas tabelas. Primeiramente, a quantidade de atributos descritos nas tabelas não será grande, além de não armazenar dados que requerem e ocupam espaços exorbitantes. Em segundo lugar, a partida tem apenas dois jogadores, o que significa que apenas os dados correspondentes aos jogadores serão redundantes. Por fim, o desempenho da manipulação de dados é maior quando o documento está em um formato desnormalizado, pois isto permite os dados relacionados sejam manipulados com apenas uma única operação de leitura ou escrita (MONGODB DOCS, 2016).

Com isso, para a representação gráfica da única tabela do banco de dados chamada Game (Figura 15), será utilizada uma formatação JSON, baseado no exemplo anterior representado na Figura 4.

Figura 15: Tabela Game

```
{
    _id: <ObjectID>,
    timeCreated: "2016-05-02T13:10:35+00:00",
    slug: 'Unique String',
    turn: 1,
    maxSubscribers: 1,
    numSubscribers: 1,
    player1: {
        name: 'Nome do Jogador',
        ready: true,
        score: 0,
        wins: 0,
        lastDices: [],
        lastScore: 0,
    }
    player2: {
        name: 'Nome do Jogador',
        ready: true,
        score: 0,
        wins: 0,
        lastDices: [],
        lastScore: 0,
    }
}
```

Fonte: elaborada pelo autor

Abaixo, os detalhes de cada campo no banco de dados ajudam a entender melhor a responsabilidade encumbida em cada um deles:

- *_id*: Este campo representa um ObjectId, que consiste em um valor hexadecimal de 12 bytes, o qual nunca se repete e haje como uma chave primária da tabela (MONGODB DOCS, 2016);
- *timeCreated*: É uma representação da data e hora de criação da partida;
- *slug*: Código alfanumérico que é gerado sempre que uma nova partida acontece ou é inserido pelo segundo jogador para efetuar o pareamento de uma partida *Multiplayer*;
- *turn*: Número inteiro (1 para o primeiro jogador e 2 para o segundo jogador) que representa o turno atual da partida;

- *maxSubscribers*: Número inteiro (1 ou 2) que representa o máximo de jogadores em uma partida (1 para *Singleplayer* e 2 para *Multiplayer*);
- *numSubscribers*: Número inteiro (1 ou 2) que representa a quantidade atual de jogadores em uma determinada partida (1 para *Singleplayer* ou *Multiplayer* enquanto não há o pareamento, e 2 para *Multiplayer* quando há o pareamento);
- *player1*: Aninhamento de sub-atributos;
    - *name*: String que contém o nome do jogador;
    - *ready*: Variável booleana que indica se o jogador está completamente pareado;
    - *score*: Número inteiro que representa a pontuação atual da partida;
    - *wins*: Número inteiro que armazena a quantidade de vitórias do jogador;
    - *lastDices*: Array que armazena os últimos dados que o jogador tirou;
    - *lastScore*: Número inteiro que representa a pontuação do jogador na rodada anterior.
- *player2*: Aninhamento que contém os mesmos sub-atributos do atributo *player1*.

4.5 CODIFICAÇÃO

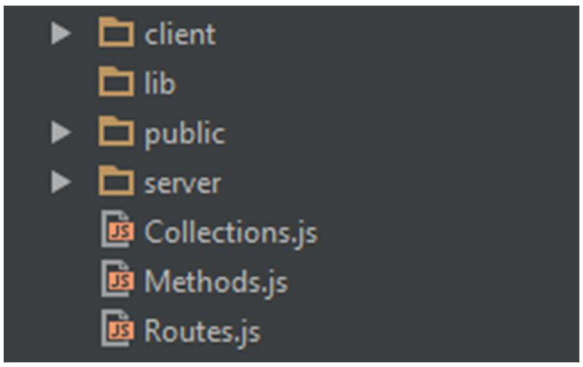
4.5.1 Estrutura de Pastas

Antes de escrever o código em si, é preciso entender onde cada arquivo deve ser colocado, criando assim, uma estrutura de pastas que permita a localização de tais arquivos. De acordo com Vogelsteller (2015, p. 12), Meteor.js possui uma grande flexibilidade quanto às estruturas de pastas desejadas para programar uma aplicação. No entanto, algumas pastas e arquivos são tratados de maneira diferente no servidor, no cliente, ou em ambos os lugares.

Segundo Vogelsteller (2015, p. 15-16), as pastas especiais *client*, *server*, *public*, *private*, *libs*, *tests* e *packages* são carregadas antes das demais pastas da aplicação por conterem arquivos importantes e preferenciais. Por outro lado, arquivos nomeados com o prefixo “*main.”* são carregados por último. O restante das pastas e arquivos é carregado seguindo a ordem de profundidade e, posteriormente, alfabética.

Desta forma, mesmo sem a obrigatoriedade de seguir uma estrutura padrão, a proposta do jogo Hog apresenta a estrutura demonstrada na Figura 16.

Figura 16: Estrutura de pastas da aplicação



Fonte: elaborado pelo autor

Cada uma das pastas possui arquivos que destinam-se a fins distintos. Na separação proposta para o jogo, a pasta *client* possui arquivos que serão carregados no *frontend* da aplicação, como *templates* em HTML, estilos CSS, disparo de eventos, interação com usuário e controle de regras. Na pasta *server*, encontram-se os arquivos de instância do banco de dados, manutenção de tabelas e regras de publicações/subscrições de dados. Na pasta *lib*, são carregados arquivos de bibliotecas que não estão relacionados ao Meteor.js em si. Por fim, a pasta *public* possui arquivos de mídia estáticos, como imagens e vídeos. Além das pastas criadas, três arquivos avulsos chamados Collection.js, Methods.js e Routes.js estão fora de todas as pastas, portanto eles são carregados tanto no servidor como no cliente.

4.5.2 Rotas

Um dos arquivos que são carregados tanto no servidor quanto no cliente é o Routes.js. Este arquivo tem como objetivo principal identificar as rotas disponíveis e acessíveis ao usuário dentro do jogo, definir os temas da camada de visualização e habilitar a subscrição e publicação correspondente dos dados pertinentes de cada partida. Segundo Vogelsteller (2015, p. 68), “rotas são as URLs de uma página *web*” que determinam o que será renderizado, e “podem ser definidas através de uma estrutura de pastas ou através da configuração do *framework* no servidor”. No caso desta aplicação, o arquivo Routes.js define as rotas habilitadas na aplicação em ambos os ambientes.

Por padrão, o *framework* Meteor.js não possui um gerenciador de rotas. Desta forma, fica a cargo do desenvolvedor escolher um pacote adicional que habilite esta opção. Vogelsteller (2015, p. 68) faz uso de uma extensão chamada iron:router, a qual foi “especificamente escrita para Meteor.js e torna fácil a configuração de rotas e a combinação destas com subscrições”. Utilizando o mesmo pacote proposto por Vogelsteller, pode-se definir a utilização de duas rotas distintas.

A primeira delas é a página inicial da aplicação, a qual irá conter algumas informações sobre o jogo e conterá as opções para a seleção do modo de jogo. Já a segunda rota tem a finalidade de subscrever o jogador a uma determinada partida. Basicamente, sempre que o jogador principal iniciar um novo jogo, um código único e aleatório será gerado. Este código, também chamado de *slug*, participará da composição total da rota na partida, pois é a partir dele que o *framework* saberá quais os dados pertinentes àquela partida.

Em caso de partidas *Multiplayer*, a rota irá indicar a renderização de *layouts* diferentes dependendo do estado da conexão inicial dos jogadores. Em outras palavras, quando um jogador convidado entrar em uma partida, a página de seleção de apelido será renderizada. Logo após a entrada do apelido do jogador convidado, a rota irá pesquisar o código único digitado pelo convidado e, em caso de sucesso, irá mostrar a página da partida. Caso a rota da partida não seja encontrada, o jogador será redirecionado para a página inicial. A Figura 17 contém o código inserido dentro do arquivo Routes.js, o qual configura o tema padrão e para rotas não encontradas, além de definir as duas rotas principais da aplicação.

Figura 17: Código-fonte do arquivo Routes.js

```
Router.configure({
    // Layout for every other template
    layoutTemplate: 'mainLayout'
});

// Data not found Template
Router.plugin('dataNotFound', {notFoundTemplate: 'homeLayout'});

Router.map(function() {

    // Home Template
    Router.route('/', {
        name: 'Home',
        template: 'homeLayout'
    });

    // Game Routes
    Router.route('/:slug', {
        name: 'Game',
        template: 'gameLayout',
        waitOn: function() {
            return Meteor.subscribe('active-game', this.params.slug);
        },
        data: function() {
            return Game.findOne({
                slug: this.params.slug
            });
        }
    });

});
```

Fonte: elaborado pelo autor

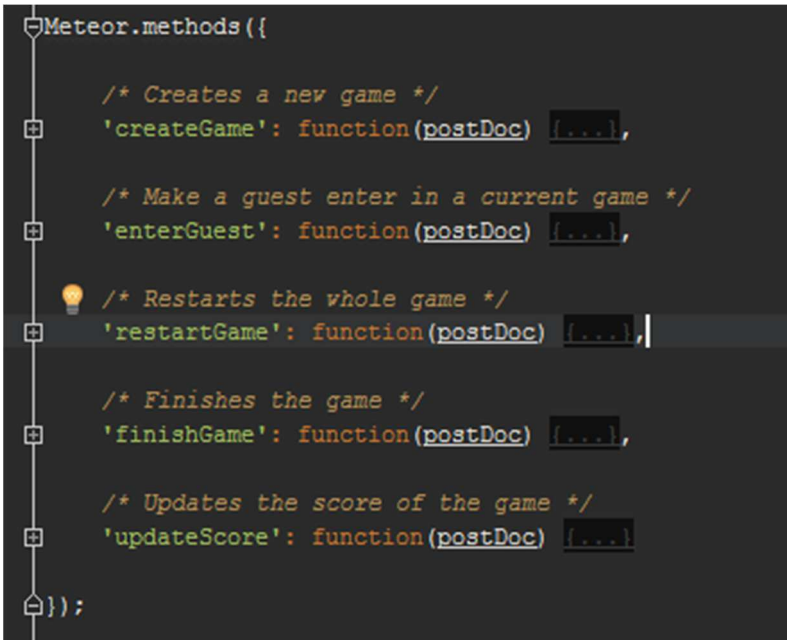
4.5.3 Métodos

Outro arquivo importante que se é carregado em ambos os ambientes é o Methods.js. Vogelsteller (2015, p. 113) explica que os “métodos são funções que podem ser chamados pelo cliente e serão executados no servidor”. De fato, a real vantagem da utilização dos métodos acontece na utilização do banco de dados. Quando um destes métodos realiza uma chamada de modificação dos dados no banco do servidor, ele automaticamente simula os resultados desta chamada no cliente, com o intuito de compensar a latência na comunicação. Caso esta chamada realizada não se concretize, as modificações efetuadas anteriormente serão removidas do cliente, voltando ao estado original (VOGELSTELLER, 2015, p. 113).

Nesta aplicação, cinco métodos serão utilizados para realizar a comunicação com o banco de dados, os quais são *createGame* (insere um novo jogo), *enterGuest* (atualiza as informações do segundo jogador, permitindo a entrada em um jogo já

criado), *restartGame* (reinicia um jogo), *finishGame* (atualiza as informações do fim de quantidades de vitórias e número total de partidas jogadas) e *updateScore* (altera a pontuação dos jogadores na partida atual), como mostra a Figura 18.

Figura 18: Escopo dos métodos definidos



Fonte: elaborado pelo autor

De fato, todos estes métodos serão utilizados no decorrer dos processos da aplicação, e serão invocados através de eventos pré-definidos em ações realizadas no cliente. Por exemplo: toda vez que um jogador escolher uma quantidade de dados a ser lançada, o método *updateScore* será invocado e se encarregará de atualizar a pontuação do jogador atual. Em casos de partidas *Multiplayer*, o adversário também deverá saber a nova pontuação deste jogador. Neste caso, após a alteração que este método realizar no banco de dados do servidor se concretizar, os dados alterados irão se propagar até a camada de visualização, alterando automaticamente as informações exibidas ao jogador adversário, que por sua vez, irá efetuar sua jogada.

4.5.4 Lógica do Jogo

Toda a lógica da regra de negócio da aplicação é executada no cliente. Desta forma, uma classe ficará encarregada por efetuar as etapas dos processos necessários para as partidas. O nome dessa classe é *GameDice*, e ela está localizada

nas subpastas *client*, *templates* e *js*. Outros arquivos também se encontram nesta pasta, como arquivos de linguagem, funções ajudantes (*helpers*) na renderização da reatividade das páginas, e manipuladores de eventos (*event handlers*) de cada *template*. Todos estes outros arquivos tem a finalidade de melhorar a experiência do usuário, porém nenhum deles aplica regras específicas da aplicação.

A Figura 19 mostra o escopo de todos os métodos que aplicam as regras do jogo. No geral, o método *rollDices* é disparado quando o jogador da rodada escolhe a quantidade de dados que ele quer rolar. Dentro deste método, é utilizado o método subsequente chamado *getDiceValue*, que irá retornar um valor aleatório de 1 a 6 (ou 1 a 4, quando é aplicado a regra *Hog Wild*) para um dado lançado. Caso o jogador esteja em uma partida *Singleplayer*, os métodos *aiTurn* e *aiDices* simularão os turnos e jogadas do oponente, escolhendo uma quantidade de dados a ser lançada e obtendo valores aleatórios dos dados, respectivamente. Por sua vez, o método *getScore* fica encarregado de atualizar a pontuação da partida após a rodada, enquanto o método *getLastScore* efetua o mesmo processo, porém para o jogador oposto ao da rodada atual. Por fim, os métodos *showResultMessage* e *showOpponentMessage* se encarregam de atualizar as visualizações dos jogadores atual e oponente com as mensagens de estado da rodada anterior (quantidade de pontos, dados lançados, regras aplicadas, entre outros).

Figura 19: Escopo dos métodos da classe *GameDice*

```
GameDice = function GameDice() {...};

/** Rolls the dices of the current player ...*/
GameDice.prototype.rollDices = function(postDoc, numberOfDices) {...};

/** Returns a dice value based on the available sides ...*/
GameDice.prototype.getDiceValue = function(sides) {...};

/** Returns the quantity of dices the A.I. rolled in its turn ...*/
GameDice.prototype.aiDices = function(max) {...};

/** Performs the A.I. turn ...*/
GameDice.prototype.aiTurn = function(postDoc) {...};

/** Returns the correct score of the round, based on the game rules ...*/
GameDice.prototype.getScore = function(postDoc, playerNumber, numberOfDices) {...};

/** Shows the last score of the opponent player ...*/
GameDice.prototype.getLastScore = function(player) {...};

/** Shows the result message of the round of the current player ...*/
GameDice.prototype.showResultMessage = function(player) {...};

/** Shows the result message of the round of the opponent player ...*/
GameDice.prototype.showOpponentMessage = function(player) {...};
```

Fonte: elaborado pelo autor

4.5.5 *Layout*

Para que o jogo possa parecer o mesmo tanto em plataformas *mobile* quanto *desktop*, há a necessidade de utilização de um *framework frontend* que possibilite a adaptação do *layout* em qualquer resolução de tela. Meteor.js providencia alguns tipos de pacotes extras que contém bibliotecas para uso destes *frameworks*. Dentre estes, o que mais se destaca é o Twitter Bootstrap. No geral, “o pacote Twitter Boostrap é um kit de ferramentas frontend para um desenvolvimento *web* mais bonito e rápido [...], providenciando componentes como tipografia, botões, tabelas, *grids* e navegação” (ROBINSON, GRAY e TITARENCO, 2015, p. 57, apud docs.meteor.com/#bootstrap, 2015). Como um dos recursos mais famosos do Twitter Bootstrap é a capacidade de fácil adaptação do *layout*, as telas da aplicação são montadas apenas uma vez. Sendo assim, dependendo da resolução do cliente, a tela será adaptada automaticamente, como mostram as Figuras 20 e 21.

Figura 20: Tela inicial da aplicação em resolução para *desktop*

Fonte: elaborado pelo autor

Figura 21: Variações da tela inicial da aplicação em resolução para *mobile*

Fonte: elaborado pelo autor

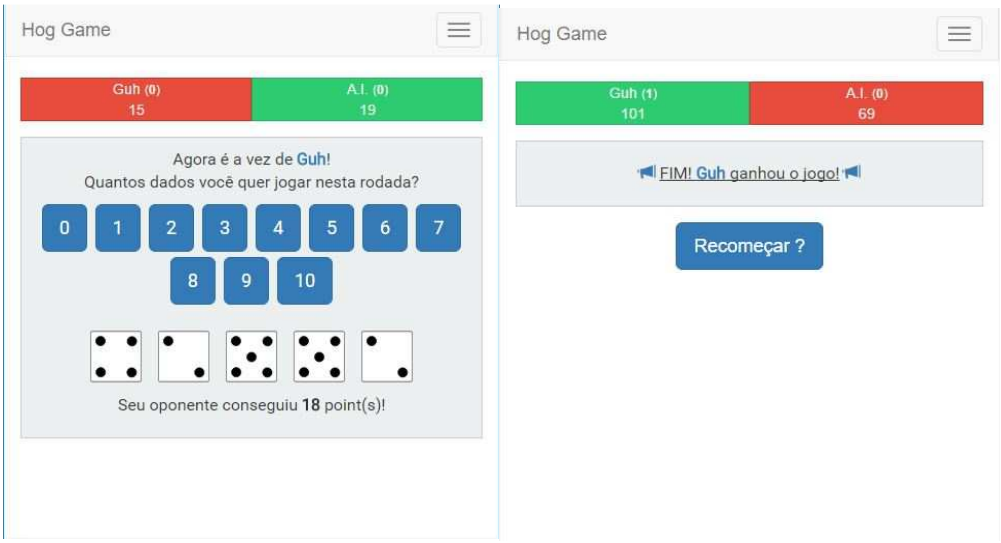
Além disso, as opções que o menu possui apresentam *templates* chamados “janelas modais”, as quais são visualizadas na mesma página que foram chamadas. Quando uma janela modal se abre, o fundo escurece e o foco da leitura se torna o conteúdo que esta janela mostrar. Outra vantagem das janelas modais são a fácil adaptação ao *layout*, ou seja, elas também são responsivas, como mostra a Figura 22.

Figura 22: Janelas modais das regras gerais e criação de um jogo

Fonte: elaborado pelo autor

Durante o jogo, duas seções irão indicar o nome, a pontuação atual e a quantidade de partidas vencidas de ambos os jogadores. Abaixo, botões serão utilizados para selecionar a quantidade de dados a ser lançada. A cada turno, abaixo dos botões, haverá uma sessão que mostrará os dados da última rodada. Por fim, mensagens que informam a situação da partida e a quantidade de pontos somadas durante o turno poderão ser visualizadas de forma direta e compacta, como é demonstrado na Figura 23.

Figura 23: Partida *multiplayer* em andamento



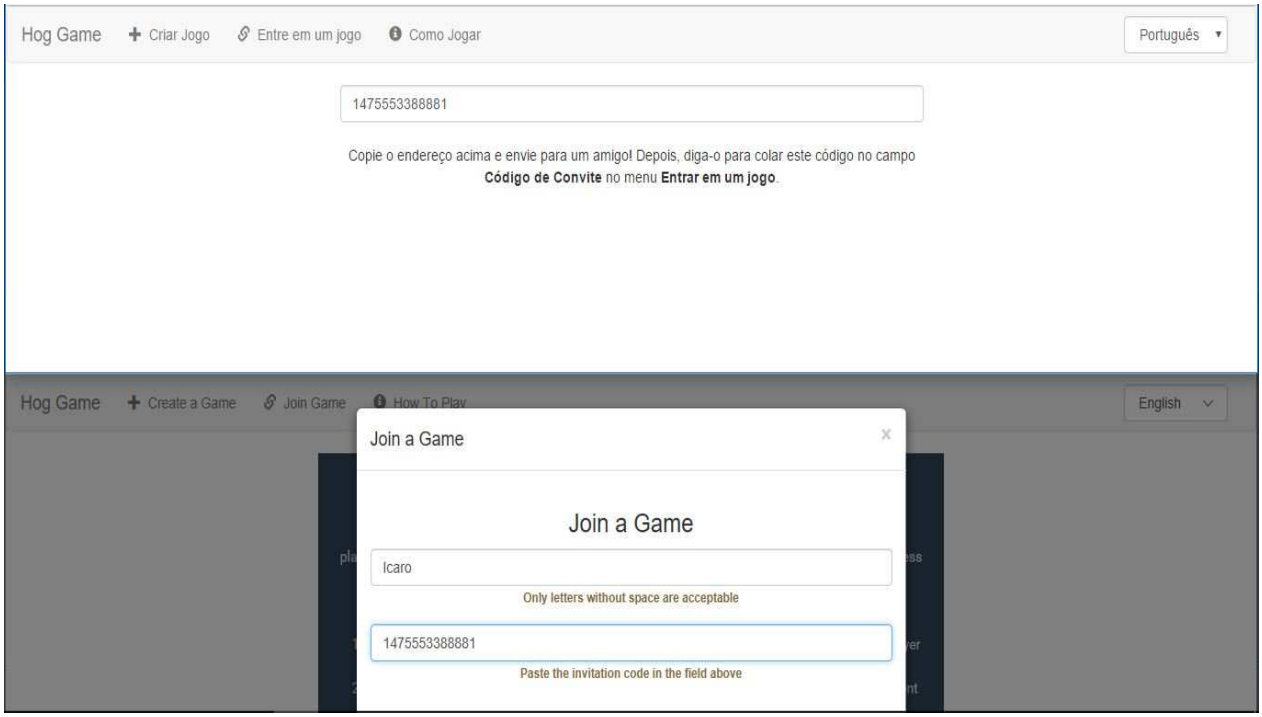
Fonte: elaborado pelo autor

Em casos de partidas *Multiplayer*, as telas dos jogadores anfitrião e oponente irão apresentar algumas diferenças durante os turnos da partida.

A primeira delas é na etapa de sincronização dos dados, como demonstrado na Figura 24. Nesse sentido, quando o código de convite gerado pelo jogador anfitrião é inserido pelo jogador oponente na janela modal que lhe é mostrada quando a opção “Entrar em um jogo” do menu é selecionada, a tela e o processo de subscrição de ambos os jogadores será um pouco diferente.

Outra diferença é a tela de turnos. Enquanto um jogador aguarda o oponente efetuar o movimento, os componentes mostrados na página serão diferentes daqueles que o jogador atual do turno visualizará. De fato, esta diferença pode ser notada na Figura 25.

Figura 24: *Template* de geração de código de convite (superior) e janela modal de entrada em uma partida (inferior)



Fonte: elaborado pelo autor

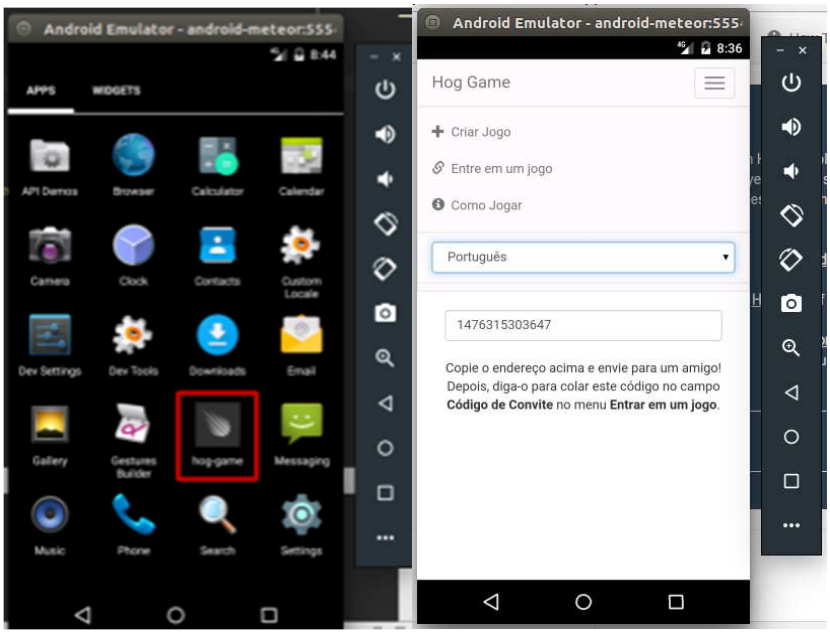
Figura 25: *Template* dos turnos dos jogadores anfitrião (superior) e oponente (convidado)

Fonte: elaborado pelo autor

4.5.6 Dispositivo Móvel

Utilizando os comandos descritos na seção 3.4, é possível gerar um aplicativo móvel a partir do mesmo código-fonte. O *layout* é responsivo em qualquer dimensão de tela, e o aplicativo é armazenado literalmente no dispositivo, o que torna possível a aplicação ser executada com ou sem conexão com a Internet, como demonstrado na Figura 26.

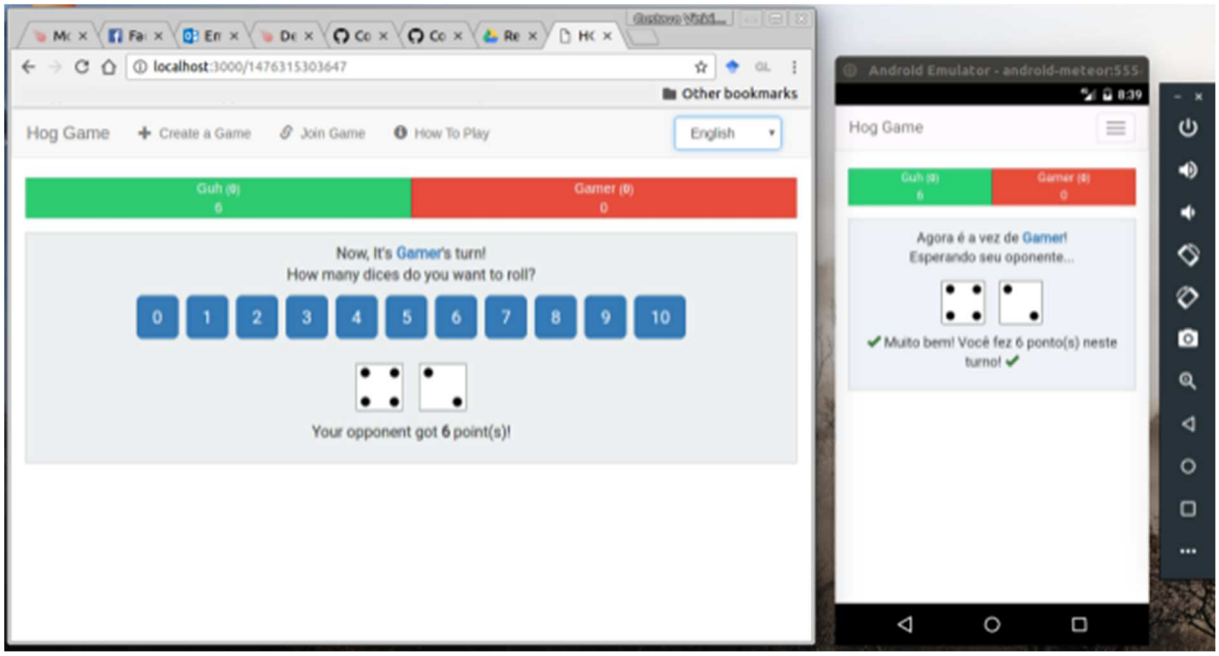
Figura 26: Aplicação rodando em um emulador Android



Fonte: elaborado pelo autor

Quando há a conexão com a Internet, o aplicativo pode criar uma partida contra qualquer outro dispositivo, bastando enviar o código de convite para o jogador oponente. A Figura 27 mostra dois jogadores diferentes jogando a mesma partida em plataformas distintas, sendo um deles no aplicativo Android e outro em um navegador *web*.

Figura 27: Dois jogadores rodando a mesma aplicação, simultaneamente, e em plataformas distintas



Fonte: elaborado pelo autor

5 CONSIDERAÇÕES FINAIS

A partir da observação dos recursos disponíveis do *framework* Meteor.js, nota-se a real capacidade de utilização de uma mesma linguagem para plataformas distintas. De fato, Meteor.js demonstra todo o potencial do JavaScript, mantendo-o desde as animações de interação do usuário no *frontend* até as regras de negócio de banco de dados no *backend*, facilitando o desenvolvimento de aplicações e mantendo sempre a mesma linguagem em todas as camadas do *framework*.

Um ponto importante a ser notado é a facilidade no relacionamento entre Meteor.js e Cordova. Como ambos utilizam a mesma linguagem de programação, portar a mesma aplicação para plataformas distintas pode se tornar fácil e intuitivo. De fato, pouca ou nenhuma modificação é necessária para realizar a portabilidade de código, pois o próprio *framework* se encarrega de distinguir a plataforma utilizada, abstraindo possíveis modificações que o usuário deveria lidar.

Outro fato a ser constatado é a capacidade de reatividade do Meteor.js. Sempre que o código é alterado, ou informações são atualizadas, todos os usuários interessados recebem os dados em tempo real, sem a necessidade da programação deste meio para tal fim. Sendo assim, apenas a aplicação em si é a principal preocupação do desenvolvedor, pois o próprio *framework* toma conta da entrega atualizada dos dados aos usuários da aplicação.

Por fim, conclui-se que o JavaScript quando atrelado a *frameworks* como Meteor.js e Cordova possui um excelente potencial para a portabilidade de código, podendo ser utilizado em diferentes tipos de aplicações. No geral, a gama de recursos disponíveis nesta linguagem facilita a integração destas funcionalidades. Mesmo que estes recursos tenham sido estruturados de formas distintas, o grau de conectividade entre eles ainda permanece nas bases de origem da linguagem.

De fato, futuros trabalhos podem apontar melhores técnicas de utilização do Meteor.js. Com a chegada de novas atualizações ao JavaScript, a tendência é que a forma de programação mude com o tempo. Por isso, outros frameworks podem vir a realizar os mesmos processos que o Meteor.js vem realizando, aperfeiçoando as camadas internas e melhorando a integração com recursos externos.

Também há de ser realizado outras tecnologias baseadas no JavaScript, pois algumas delas também podem abordar paradigmas similares ao Meteor.js. Questões

como desempenho e velocidade de comunicação devem ser estudados a fundo afim de explorar pontos prós e contras na resolução da mesma aplicação.

6 REFERÊNCIAS

DEBERGALIS, Matt. **Meteor 0.9.2: Building iOS and Android mobile apps with PhoneGap**. 2014. Disponível em: <http://info.meteor.com/blog/meteor-092-ios-android-mobile-apps-phonegap-cordova> Acesso em: 25 mar. 2016.

DEBERGALIS, Matt. **Meteor 1.0**. 2014. Disponível em: <http://info.meteor.com/blog/meteor-1-0> Acesso em: 25 mar. 2016.

DENERO, John. **Project 1: The Game of Hog**. Disponível em: <https://inst.eecs.berkeley.edu/~cs61a/fa12/projects/hog/hog.html> Acesso em: 27 mar. 2016.

DICKSON, Jared. **Xamarin Mobile Development**. 2013. 17 páginas. Bacharelado em Sistemas da Informação - Grand Valley State University. p. 6-8. Disponível em: <http://scholarworks.gvsu.edu/cgi/viewcontent.cgi?article=1167&context=cistechlib> Acesso em: 15 abr. 2016.

GHATOL, Rohit; PATEL, Yogesh. **Beginning PhoneGap**: Mobile Web Framework for JavaScript and HTML5. New York: Apress Media LLC, 2012. Disponível em: <http://sd.blackball.lv/library/Beginning_PhoneGap.pdf> Acesso em: 04 mar. 2016.

GOLDSHTEIN, Uri. **Official Angular support with angular-meteor 1.0.0**. 2015. Disponível em: <http://info.meteor.com/blog/official-angular-support-with-angular-meteor-1.0.0> Acesso em: 25 mar. 2016.

GUEDES, Gilleanes T. A. Introdução à UML. In: GUEDES, Gilleanes T. A. (Org.) **UML 2**: Guia Prático. 2 ed. São Paulo: Novatec, 2014. p. 15-32. Disponível em: <http://www.novatec.com.br/livros/uml2-2ed/capitulo9788575223857.pdf> Acesso em: 05 abr. 2016.

JSON.ORG. **Introdução ao JSON**. Disponível em: <http://www.json.org/json-pt.html> Acesso em: 17 abr. 2016.

KOHAN, Bernard; MONTANEZ, Joseph. **Native vs Hybrid / PhoneGap App Development Comparison:** A comparison of native app development (iPhone: Objective-C / Swift, Android: Java) vs hybrid / PhoneGap app development (HTML5, CSS, JavaScript). 2015. Disponível em: <http://www.comentum.com/phonegap-vs-native-app-development.html> Acesso em: 13 abr. 2016.

METEOR. **Explore the Platform**. Disponível em: <https://www.meteor.com/projects> Acesso em: 25 mar. 2016.

METEOR DEVELOPERS. **Mobile**: How to build mobile apps using Meteor's Cordova integration. Disponível em: <https://guide.meteor.com/mobile.html> Acesso em: 10 out. 2016.

METEOR DOCS. **Official Documentation for Meteor.js**. Disponível em: <http://docs.meteor.com/#/full/> Acesso em: 25 mar. 2016.

MONGODB DOCS. **Data Modeling Introduction**. 2016. Disponível em: <https://docs.mongodb.org/manual/core/data-modeling-introduction/> Acesso em 25 abr. 2016.

MOREIRA, Rafael Henrique. **O que é Node.js?**. Janeiro de 2013. Disponível em: <http://nodebr.com/o-que-e-node-js/> Acesso em: 30 mar. 2016.

ROBINSON, Josh; GRAY, Aaron; TITARENCO, David. **Introducing Meteor**: Build better apps faster with Meteor. New York: Apress Media LLC, 2015. Disponível em: <http://file.allitebooks.com/20160114/Introducing%20Meteor.pdf> Acesso em: 21 mar. 2016.

RODRIGUES, Joel. **Modelo Entidade Relacionamento (MER) e Diagrama Entidade-Relacionamento (DER)**. 2014. Disponível em: <http://www.devmedia.com.br/modelo-entidade-relacionamento-mer-e-diagrama-entidade-relacionamento-der/14332> Acesso em 27 abr. 2016.

VOGELSTELLER, Fabian. **Building Single-page Web Apps with Meteor**. Birmingham: Packt Publishing Ltd, 2015.

WARGO, John M. **PhoneGap Essentials: Building Cross-Platform Mobile Apps**. New Jersey: Pearson Education, 2012. p. 3-22.

WHINNERY, Kevin. **Comparing Titanium and PhoneGap**. 2012. Disponível em: <http://www.appcelerator.com/blog/2012/05/comparing-titanium-and-phonegap/> Acesso em: 14 abr. 2016.