



UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO  
ESCOLA DE INFORMÁTICA APLICADA  
CURSO DE BACHARELADO EM SISTEMAS DE INFORMAÇÃO

ARQUITETURA E DESENVOLVIMENTO  
DE UMA APLICAÇÃO WEB DISTRIBUÍDA BASEADA EM JAVASCRIPT

Afonso Almendra Praça de Carvalho  
Orientadora: Dra. Leila Cristina Vasconcelos de Andrade

Rio de Janeiro, RJ - Brasil  
Junho de 2014

ARQUITETURA E DESENVOLVIMENTO DE UMA APLICAÇÃO WEB DISTRIBUÍDA  
BASEADA EM JAVASCRIPT

Aprovado em \_\_\_\_/\_\_\_\_\_/\_\_\_\_

BANCA EXAMINADORA

---

Dra. Leila Cristina Vasconcelos de Andrade (orientadora)

---

Dr. Mariano Pimentel

---

Dr. Luiz Amancio Machado de Sousa Junior

O autor deste Projeto autoriza a ESCOLA DE INFORMÁTICA  
APLICADA da UNIRIO a divulgá-lo, no todo ou em parte, resguardados os direitos  
autorais conforme legislação vigente.

Rio de Janeiro, \_\_\_\_ de \_\_\_\_ de \_\_\_\_.

---

Afonso Almendra Praça de Carvalho

ARQUITETURA E DESENVOLVIMENTO  
DE UMA APLICAÇÃO WEB DISTRIBUÍDA BASEADA EM JAVASCRIPT

Projeto de Graduação apresentado à Escola  
de Informática Aplicada da Universidade  
Federal do Estado do Rio de Janeiro  
(UNIRIO) para obtenção do título de  
Bacharel em Sistemas de Informação

Afonso Almendra Praça de Carvalho  
Orientadora: Dra. Leila Cristina Vasconcelos de Andrade

*"Escreva programas que façam uma tarefa bem definida e a façam bem feita.  
Escreva programas que saibam trabalhar em conjunto com outros programas,  
e que lidem com entradas e saídas de dados em forma de texto,  
porque texto é uma interface universal"*

***Douglas McIlroy***

*"A força de um sistema vem muito mais da relação entre os programas  
do que dos programas propriamente ditos."*

***Brian Kernighan and Rob Pike***

## **Agradecimentos**

Agradeço aos meus pais, Gustavo e Lana, pelo apoio e pela confiança que sempre me depositaram durante toda a vida. À minha esposa, Daniela Soria, pela paciência e pelo incentivo, fundamental, para que eu concluísse essa jornada. À minha filha, Maria Morena, pela compreensão nos momentos em que estive ausente. Agradeço também à minha orientadora, Leila Cristina Vasconcelos de Andrade, pelo incentivo desde o início do curso.

## **Resumo**

Nesse trabalho é apresentado, através da especificação de arquitetura e da implementação de um protótipo funcional, o uso de uma arquitetura de *software* distribuída e orientada a serviço, seguindo o estilo arquitetural REST, em uma aplicação *Web*. São apresentadas as tecnologias utilizadas, o estilo arquitetural e o projeto da aplicação, tanto do lado cliente como do lado servidor.

**Palavras-chave:** Arquitetura de *software*, Aplicação *Web*, API REST, JavaScript

## **Sumário**

<i>1 - Introdução</i>	<i>- 8</i>
<i>2 - Tecnologias utilizadas</i>	<i>- 10</i>
2.1 - HTTP	- 10
2.2 - HTML 5	- 10
2.3 - JavaScript	- 11
2.4 - Angular JS	- 11
2.4.1 - Funcionalidades Chave do AngularJS	- 12
2.5 - MongoDB	- 14
2.6 - Node.js	- 16
2.7 - Express	- 17
2.8 - Bootstrap	- 17
<i>3 - Arquitetura REST</i>	<i>- 18</i>
<i>4 - Proposta</i>	<i>- 22</i>
4.1 - Arquitetura	- 24
4.1.1 - Arquitetura do servidor (back end)	- 24
4.1.2 - Arquitetura dos clientes (front end)	- 25
4.1.3 - Arquitetura de implantação (deploy)	- 25
4.2 - Modelagem de dados	- 27
4.3 - Colaboração no Sistema	- 28
4.4 - Definição da estrutura de rotas da API	- 29
<i>5 - Desenvolvimento</i>	<i>-30</i>
<i>6 - Conclusão e trabalhos futuros</i>	<i>- 36</i>

## 1) Introdução

O objetivo deste trabalho é apresentar o uso de uma arquitetura distribuída e orientada a serviço, através de um protótipo de sistema *web* funcional baseado na linguagem de programação JavaScript, num projeto de pequeno porte. Na arquitetura proposta, o lado servidor, onde estão implementadas as regras de negócio e a persistência dos dados, está totalmente separado do lado cliente, que é responsável pela exibição e formatação dos dados no navegador dos usuários, bem como pela captura e tratamento das entradas de dados por parte dos mesmos. Vamos considerar que cada uma dessas partes é um subsistema do sistema proposto.

A comunicação entre os subsistemas é feita através de requisições HTTP<sup>1</sup> a serviços expostos em uma API REST<sup>2</sup> com interface bem definida e documentada. Os dados que são trocados entre os subsistemas estão em formato estruturado, de modo que cada subsistema tem a flexibilidade e a independência necessárias para ser desenvolvido e alterado sem afetar os outros subsistemas, desde que este mantenha a compatibilidade da API, definida anteriormente. Desse modo, consegue-se obter flexibilidade no desenvolvimento de todos os subsistemas, facilidade na manutenção do código e da infra-estrutura, bem como agilidade nas respostas às mudanças dos requisitos de projeto e a novas necessidades de negócio.

O motivo da escolha do tema é o fato de as organizações, hoje, terem cada vez mais sistemas independentes e que precisam se inter-comunicar, inclusive com sistemas de terceiros e parceiros de negócio, com o fim de executar uma tarefa maior. A arquitetura de *software* das corporações, atualmente, vem se caracterizando por ser composta de diversos sistemas simples, ao invés de grandes sistemas monolíticos. Esse novo formato arquitetônico evidencia a grande importância da integração entre os sistemas. Sistemas que antes eram muito grandes, complexos, de difícil escalabilidade e desenvolvimento, hoje são desenvolvidos como diversos subsistemas independentes.

Em uma arquitetura distribuída e orientada a serviço, cada sistema (ou subsistema) resolve um problema específico, e a integração dos mesmos resolve um problema mais complexo.

---

<sup>1</sup> Protocolo de comunicação utilizado para sistemas de informação de hipermídia, distribuídos e colaborativos. É a base para a comunicação de dados da Web.

<sup>2</sup> Estilo de arquitetura para construção de *web services* consistentes e coesos. O estilo de arquitetura REST é baseado em recursos e nos estados desses recursos.



É natural que o ambiente mais propício para o desenvolvimento de sistemas nesse novo paradigma, por sua natureza distribuída, seja a Internet. Dado que a linguagem mais utilizada hoje na *Web* é o JavaScript, implementado na maioria dos navegadores atuais, se deu a escolha dessa linguagem para o projeto.

Estando o JavaScript em franca expansão também no *server side*, com a implementação do Node.js<sup>3</sup>, usado hoje por grandes organizações em suas soluções *Web*, essa também foi a linguagem utilizada no lado servidor do projeto. Com o uso da mesma linguagem em todo o projeto, o mesmo ganhou coesão e houve redução na tradicional lacuna que separa os desenvolvimentos *front end* e *back end*.

A importância da integração de sistemas em rede trouxe à tona a necessidade de uma arquitetura que defina como esses sistemas devem interagir entre si e como devem definir suas interfaces. Um dos estilos arquiteturais que se propõe a resolver essa questão é o REST (Transferência de Estado Representacional). O sistema desenvolvido nesse projeto faz uso do estilo arquitetural REST para tal fim.

O protótipo funcional de sistema *Web* desenvolvido é um guia de restaurantes que, na instância exemplificada aqui, lista os restaurantes de Penedo, distrito da cidade de Itatiaia - Rio de Janeiro. O protótipo funcional está disponível na *Web* no seguinte endereço: <<http://penedorj.com.br>>.

---

<sup>3</sup> Plataforma de *software* para aplicações de servidor e aplicações de rede escaláveis e de alta performance. Node.js utiliza o motor de interpretação do Google, o V8.

## 2) Tecnologias utilizadas

### 2.1) HTTP

O *Hypertext Transfer Protocol* (HTTP), em português, Protocolo de Transferência de Hipertexto, é um protocolo de comunicação (na camada de aplicação segundo o Modelo OSI da ISO) utilizado para sistemas de informação de hipermídia, distribuídos e colaborativos. É a base para a comunicação de dados da *Web*.

Conforme consta na especificação do HTTP 1.1 (*Especificação do HTTP 1.1, 2014*), o mesmo é um protocolo *stateless* (que não guarda estado da aplicação) o que o torna, por natureza, muito escalável. Uma importante funcionalidade do HTTP é a tipagem e a negociação da representação de dados, o que permite que os sistemas que o utilizam para comunicação sejam construídos de forma independente dos dados que estão sendo transferidos. O HTTP está em uso na *Web* desde 1990.

O HTTP funciona através da transmissão de mensagens, que são sua unidade básica de comunicação. Existem dois grupos principais de mensagens: As *request messages*, que são as mensagens de requisição de um recurso, e as *response messages*, que são respostas às mensagens de requisição.

O protocolo HTTP é um protocolo de requisição/resposta. Um cliente envia uma requisição para o servidor na forma de um *request method*, uma *URI*, e o protocolo da versão, seguida pelo tipo de dados da mensagem, algumas meta-informações e, se necessário, um corpo de conteúdo, através de uma conexão com o servidor. O servidor responde com um código de status, que inclui a versão do protocolo utilizada, meta informações e, possivelmente, um corpo de mensagem.

Toda a comunicação entre os subsistemas do projeto apresentado aqui, se dá através de requisições HTTP, logo, essa tecnologia é uma das bases sobre as quais o sistema proposto funciona.

### 2.2) HTML 5

A linguagem de marcação de documentos da *World Wide Web* sempre foi o HTML. O HTML foi inicialmente projetado como uma linguagem que descrevesse semanticamente

documentos científicos, embora seu modelo e suas adaptações, através dos anos, tenham permitido ao mesmo ser usado para descrever inúmeros tipos de documentos.

Porém, uma área que não havia sido adequadamente coberta pelo HTML é a área das, hoje populares, aplicações *Web*. A especificação do HTML5 (*W3C - Especificação do HTML5, 2014*) tenta corrigir isso, e, ao mesmo tempo, atualizar as especificações para tratar questões levantadas nos últimos anos.

Dentre diversas novas funcionalidades, o HTML5 permite que os autores incluam dados nas marcações, para que os mesmos sejam usados por *scripts* do lado cliente ou do lado servidor, através do uso de atributos com o prefixo "data-". O HTML5 garante que estes atributos não serão alterados pelos navegadores. Essa é uma funcionalidade muito importante e amplamente utilizada pelos *frameworks* de desenvolvimento de aplicações *Web* atuais, como o AngularJS, por exemplo. O AngularJS faz uso desses dados nas marcações para diversos fins, um exemplo é o atributo "data-ng-model" para vincular uma tag HTML5 a um atributo de um objeto JavaScript.

### 2.3) JavaScript

JavaScript (popularmente conhecido como JS) é uma linguagem interpretada e orientada a objeto com funções de primeira classe, apesar de mais conhecida como a linguagem de *scripts* para páginas *Web*, tem diversos outros usos em ambientes fora do navegador, como, por exemplo, em Node.js ou em bancos de dados não relacionais, como o Apache CouchDB. Conforme documentação (*JavaScript - Mozilla Developer Network, 2014*), é uma linguagem baseada em *prototype*, multi-paradigma e dinâmica, suportando tanto programação orientada a objetos, como imperativa e funcional.

### 2.4) AngularJS

AngularJS é um *framework* MVC (ou MVW) para desenvolvimento de aplicações *front-end* em JavaScript desenvolvido e mantido pelo Google. Com uma estrutura bem definida, o AngularJS facilita a organização do código, além de fornecer controle de dependências e *two-way binding* nas *views* HTML. O AngularJS, que conta com ampla documentação (*AngularJS: API Reference, 2014*), nos permite estender o vocabulário HTML

de modo específico para cada aplicação, através da criação de novas *tags* ou novos atributos em *tags* HTML padrão. O ambiente resultante é extraordinariamente expressivo, legível e rápido para se desenvolver.

O *framework* é personalizável, permitindo seu uso da maneira mais adequada ao desenvolvimento de sua aplicação. É inteiramente extensível e funciona bem com outras bibliotecas, como, por exemplo, JQuery.

#### 2.4.1) Funcionalidades Chave do AngularJS

- ***Data Binding***

*Data-binding* é uma forma automática de atualizar a *view* (HTML) sempre que o modelo é atualizado. Da mesma forma, o modelo é atualizado quando os dados na *view* (HTML) são alterados. Essa funcionalidade nos fornece uma grande vantagem, que é não precisarmos nos preocupar com a manipulação do DOM (HTML), tendo em vista que o *framework* já cuida disso através do *Data-Binding*.

- ***Controller***

*Controllers* fornecem o comportamento aos elementos do DOM, permitindo que expressemos as lógicas da *view* de forma limpa e legível, sem a necessidade de manipulação direta do HTML, abstraindo, dessa forma, a estrutura do mesmo.

- ***Comunicação com o server-side***

O *framework* provê, nativamente, serviços que rodam em cima de XHR, permitindo chamadas AJAX ao *server-side* e utilizando *promises*<sup>4</sup>, que simplificam seu código, já que são elas que lidam com o retorno e tratamento das requisições assíncronas.

- ***Diretivas***

Diretivas são marcações em elementos do DOM (feitas como um atributo, um nome de elemento, ou mesmo uma classe CSS) que indicam ao compilador AngularJS que um comportamento específico deve ser anexado ao elemento, ou até mesmo que esse elemento e seus elementos filhos devem sofrer alguma transformação. Através do uso

---

<sup>4</sup> Uma *promise* representa o resultado de alguma operação assíncrona. Ela pode estar em três diferentes estados: pendente, resolvida, ou rejeitada.

de diretivas é possível estender a sintaxe do HTML, de forma específica, visando atender requisitos intrínsecos da aplicação a ser desenvolvida. As diretivas possibilitam a criação de componentes reutilizáveis. Um componente permite o encapsulamento de estruturas complexas de HTML, CSS, e comportamento (JavaScript), conseguindo com isso, que as funcionalidades se mantenham coesas e bem definidas.

- **Filtros**

Filtros são componentes que manipulam dados, tipicamente para exibição dos mesmos, sem alterar os dados originais. O AngularJS traz diversos filtros nativos, como, por exemplo, filtros para exibição de valores monetários e datas. Porém, cada desenvolvedor pode criar novos filtros personalizados de acordo com a necessidade da sua aplicação. Filtros são muito práticos e podem ser encadeados, como um *pipe*, onde a saída de um filtro é a entrada de seu sucessor.

- **Serviços**

Serviços são muito úteis quando determinada funcionalidade serve a diversos *controllers*. Usando um serviço, podemos definir uma funcionalidade apenas uma vez e a utilizar em diversos *controllers*, através da injeção de dependências. Um serviço é sempre um *singleton*<sup>5</sup>. Também se presta muito bem a ser um local onde podemos persistir informações de estado da aplicação, dado que os *controllers* são reiniciados a cada vez que a rota da aplicação muda.

- **Factorys**

*Factorys* podem funcionar como os modelos da aplicação. Após definirmos uma *Factory*, injetamos a mesma no *controller* ou serviço que for a utilizar, e esse pode instanciar novos objetos a partir dessa *Factory*. Uma *Factory* pode ser muito útil quando usada para retornar uma função “classe”, essa utilizada para a criação de novas instâncias da mesma. É importante lembrar que as *Factorys*, tal qual os serviços, são sempre *singletons*.

---

<sup>5</sup> Em engenharia de *software*, *singleton* é um padrão de projeto que restringe o número de instâncias de uma classe a um único objeto. Isso é útil quando exatamente um objeto é necessário para coordenar ações em diversas partes do sistema.

- **Roteamento**

O módulo *ngRoute* provê serviços e diretivas de roteamento. É esse módulo quem observa as mudanças na URL da aplicação e reage às mesmas, intanciando o devido *controller*, carregando o *template (view)* correto, e, possivelmente, resolvendo algum carregamento assíncrono antes de iniciar o *controller*.

- **Localização e Internacionalização**

Uma parte importante de qualquer aplicativo global é a localização. Através de filtros e diretivas o AngularJS fornece as ferramentas necessárias para tornar a aplicação disponível em todas as localidades.

- **Injeção de dependência**

A injeção de dependência no AngularJS permite que o desenvolvedor, declarativamente, descreva como sua aplicação é interligada (que módulo depende de que outros módulos). Injeção de dependência também é *core* no AngularJS. De modo que, se algum componente não se adequa às necessidades de seu projeto, você pode substituí-lo facilmente. A declaração explícita das dependências facilita a leitura e manutenção do código, da mesma forma que a substituição de um módulo e os testes unitários do mesmo.

- **Testabilidade**

O AngularJS foi projetado, desde o seu início, para ser testável. Isso encoraja a separação entre o comportamento e a exibição, o uso de *mocks*, e as vantagens de um mecanismo eficiente de injeção de dependências. O AngularJS também já vem com um mecanismo de teste *end-to-end*, para testes de integração entre os módulos.

## 2.5) MongoDB

MongoDB é um banco de dados *open-source* orientado a documentos. É o líder de mercado no segmento de banco de dados NoSQL. Conforme sua especificação (*MongoDB manual*, 2014), é escrito em C++ e, dentre diversas outras, contém as seguintes funcionalidades-chave:

- Esquemas de dados dinâmicos, oferecendo simplicidade e robustez
- Suporte à indexação em qualquer atributo
- Replicação e alta disponibilidade
- Capacidade de escalar horizontalmente
- Pesquisas baseadas em documentos

MongoDB foi projetado visando a facilidade de uso e capacidade de escalar conforme demandas de uso de forma automática.

Um registro no MongoDB é um documento, isto é, uma estrutura de dados contendo pares de chave e valor. Os documentos do MongoDB são similares a objetos JSON<sup>6</sup>. O valor dos campos em um documento podem conter outros documentos, *arrays*, e *arrays* de documentos.

As vantagens claras do uso de documentos são: o fato de documentos (objetos) corresponderem aos tipos de dados nativos de diversas linguagens de programação (JavaScript, por exemplo); o fato de documentos conterem *arrays* e outros documentos reduzir muito a necessidade de *joins* custosos; e o fato de um esquema dinâmico suportar um polimorfismo fluente.

Diferente dos SGBDs relacionais, no mundo NoSQL, o esquema dos dados é usualmente responsabilidade da aplicação. Se o banco ou a coleção não existem no momento em que são acessados, o MongoDB os cria automaticamente, sem forçar nenhum esquema no momento da criação. Por outro lado, o MongoDB provê aos usuários toda a flexibilidade do uso de índices – permitindo a criação de índices em campos aninhados e em *arrays*, tanto como índices esparsos, geoespaciais e de texto.

Os dados (documentos) são armazenados internamente no formato BSON (JSON binário) (*BSON - Binary JSON, 2014*), projetado para ter três características principais, que seguem abaixo:

- **Leveza**

Diminuir a sobrecarga de espaço para um tamanho mínimo é fundamental para qualquer representação de dados, especialmente quando usado sobre a rede.

---

<sup>6</sup> JSON (JavaScript Object Notation) é um formato de transferência de dados extremamente leve. É fácil tanto para os humanos como para os sistemas lerem e escreverem.

- **Transversalidade**

BSON foi projetado para ser atravessado facilmente. Essa é uma propriedade vital para seu papel como modelo de representação de dados primários no MongoDB. Essa característica permite buscas e edições rápidas nos documentos.

- **Eficiência**

Codificar dados para o formato BSON e decodificar os mesmos é uma tarefa que pode ser executada de forma muito rápida na maioria das linguagens, devido ao uso dos tipos de dados nativos da linguagem C no BSON.

## **2.6) Node.js (Plataforma de desenvolvimento de servidor)**

Node.js é uma plataforma de *software* para aplicações de servidor e aplicações de rede escaláveis e de alta performance (*Node.js platform, 2014*). As aplicações Node.js são escritas em JavaScript e podem rodar dentro do ambiente *runtime do Node.js*, tanto no Windows, como no Mac e no Linux, sem serem necessárias alterações no código fonte. Node.js utiliza o motor de interpretação do Google, o V8.

As aplicações Node.js são projetadas visando maximizar o *throughput* e a eficiência, usando operações de entrada e saída não bloqueantes, e eventos assíncronos. As aplicações Node.js rodam, cada uma, em apenas uma *thread*.

O Node.js usa, internamente, a *engine* Google V8 JavaScript para executar o código e uma grande parte de seus módulos é escrito em JavaScript. Node.js contém, nativamente, uma biblioteca HTTP de lado servidor, tornando possível rodar um servidor web sem outra ferramenta, como por exemplo o Apache ou Nginx.

Devido às características citadas acima, o Node.js se adapta perfeitamente para o uso em aplicações de tempo real como uso intensivo de dados e que rodem em dispositivos distribuídos. Com um vasto ecossistema de bibliotecas, larga comunidade *open-source*, alta performance e facilidade de *deploy*, Node.js é uma excelente alternativa para a criação de servidores para aplicativos *Web*, trazendo, ainda, a vantagem de se poder utilizar a mesma linguagem no *server side* e no *front end*.



## 2.7) Express

Express é um framework para desenvolvimento de aplicações *Web*, minimalista e flexível, que roda sobre a plataforma Node.js (*Express - web application framework for node*, 2014). Provendo um conjunto robusto de funcionalidades para a construção de aplicações *Web*, sejam elas *Spa* (Single-page Applications), aplicações com múltiplas páginas, ou aplicações híbridas.

Com diversos métodos utilitários HTTP e de *middleware* disponíveis, o Express torna fácil a criação de Web APIs amigáveis de forma rápida e robusta. O Express provê uma pequena camada de funcionalidades fundamentais para aplicações *Web*, porém, sem obscurecer as funcionalidades já disponíveis no Node.js.

## 2.8) Bootstrap

Bootstrap é um *framework front-end* de CSS para o desenvolvimento de projetos web responsivos (sites que se adaptam a diferentes tamanhos de tela, se ajustando ao dispositivo do usuário) para a *web* (*Bootstrap - Framework de CSS*, 2014). Atualmente é o *framework* mais popular na *Web* para tal fim. Ele poupa muito do trabalho de se escrever CSS, e oferece um sistema de *grid* responsivo, que se adapta aos mais diversos tamanhos de tela.

Além do poderoso sistema de *grid*, oferece uma grande quantidade de componentes, como botões, barras de navegação, ícones, componentes de paginação, dentre vários outros. Com isso, ele mune o desenvolvedor *Web*, de forma fácil e intuitiva, de muito do que esse vai precisar para construir seus protótipos e primeiras versões do seu sistema.

Além do CSS tradicional, o Bootstrap inclui o suporte aos dois pré-processadores de CSS mais populares do mercado, o Less e o Sass. Falando de responsividade no *layout*, o Bootstrap escala seu projeto para ser exibido em telefones, *tablets* ou *desktops*, a partir de uma única base de código, facilitando assim, a portabilidade de plataformas e simplificando a manutenção do código.

Um dos fatores diferenciais do Bootstrap é sua extensa e abrangente documentação, com centenas de exemplos funcionais, trechos de código, e imagens explicativas.

### 3) Arquitetura REST

REST foi um termo cunhado por Roy Thomas Fielding, em sua tese de doutorado (FIELDING, 2011), onde ele modela um estilo de arquitetura para construção de *web services* consistentes e coesos. O estilo de arquitetura REST é baseado em recursos e nos estados desses recursos.

As pesquisas em arquitetura de *software* investigam métodos para determinar a melhor forma de se particionar um sistema, como perceber e conceber seus componentes, e como esses componentes irão se identificar e se comunicar uns com os outros. Versam também sobre como um elemento de um sistema pode evoluir de forma independente, e como todos os itens acima podem ser descritos de uma maneira formal. Um estilo arquitetônico é um conjunto coordenado de restrições arquiteturais. E a implementação de um estilo bem definido é pilar de uma boa arquitetura e de um bom sistema de informação.

REST é a abreviação de Transferência de Estado Representacional. É um estilo arquitetural baseado em recursos e nas representações desses recursos. Enfatiza a escalabilidade na interação entre os componentes, a generalidade de interfaces, a implantação (*deploy*) independente dos componentes de um sistema, o uso de componentes intermediários visando a redução na latência de interações, o reforço na segurança e o encapsulamento de sistemas legados.

Quando dizemos que o estilo arquitetônico REST é baseado em recursos, isso significa que nos referimos, tipicamente, a recursos ou coisas ao invés de ações. (MASSE, 2011) Em REST, nos referimos a substantivos ao invés de verbos. Quem determina a ação que deve ser feita sobre esse recurso é o verbo HTTP utilizado na requisição (GET, POST, PUT, DELETE). Uma chamada do tipo `getAllUsers()`, em REST seria algo do tipo `GET api.domain.com/users`. Ser baseado em recurso é um conceito que caracteriza fortemente o estilo de arquitetura REST. Os recursos são identificados por URIs e são separados de suas representações. Múltiplas URIs podem se referir a um mesmo recurso, e esse recurso pode estar representado de forma diferente em diferentes URIs.

O que nos referimos como uma representação é parte ou até mesmo o todo do estado de um recurso. É o dado que será transferido entre o *Client Side* e o *Server Side*, usualmente em XML ou JSON.

Existem seis grandes restrições arquiteturais definidas no estilo REST (REST API Tutorial, 2014):

- Interface uniforme
- Independência de estado da aplicação
- Cacheabilidade
- Cliente-Servidor
- Sistema em camadas
- Código sob demanda (opcional)

A seguir, descrições mais detalhadas sobre cada uma das seis grandes restrições arquiteturais REST:

- **Interface uniforme**

A restrição de interface uniforme diz que deve ser definida uma interface entre o cliente e o servidor. Com uma interface bem definida, a arquitetura se torna mais simples e desacoplada, de modo que, cada parte da aplicação pode ser desenvolvida de forma independente. Essa interface significa o uso dos verbos HTTP, as URIs e o modo de usar os HTTP Responses (status, body).

Recursos individuais são identificados nas requisições usando URIs. Os recursos são, em si, conceitualmente separados da representação que é retornada ao cliente. Por exemplo, o servidor não retorna o seu banco de dados, mas sim algum HTML, XML ou JSON, que representa algum registro do banco, muito possivelmente de forma parcial, e com as informações estruturadas para determinado fim.

A Interface uniforme torna possível a manipulação dos recursos através de suas representações. Quando um cliente tem uma representação de um recurso, ele tem informação suficiente para modificar e até excluir esse recurso do servidor (se tiver permissão para tal).

Cada mensagem deve incluir informação que descreva, de forma suficiente, como ela deve ser processada. As mensagens devem ser autodescritivas. Por exemplo, o tipo de *parser* que deve ser usado para um tipo de mídia, deve ser especificado por um *Internet media type* (anteriormente conhecido como MIME type).

É natural que, quando falamos de Interface Uniforme, estamos falando de uma interface onde Hypermedia é o motor do estado da aplicação. Os clientes entregam o estado ao servidor através do conteúdo do body das mensagens, dos parâmetros de *query string*, dos *request headers* e da URI solicitada (identificador do recurso). Já os serviços, entregam o estado das aplicações aos clientes via o conteúdo do *body* dos *responses*, os *response codes*, e os *response headers*. Isso é tecnicamente conhecido como *hypermedia* (ou *hyperlinks* com *hypertext*).

Uma Interface Uniforme que todo serviço REST deve prover é fundamental para seu design. A Interface é o link entre o cliente e o servidor.

- **Independência de estado da aplicação**

REST é um acrônimo de Transferência de Estado Representacional, logo, independência do estado da aplicação é um conceito chave. Essencialmente, isso significa que o estado necessário para que um servidor possa responder uma requisição de um cliente está contido na própria requisição, seja como parte da URI, como um parâmetro de *query string*, como *body*, ou *headers*. A URI identifica unicamente o recurso, e o *body* contém seu estado ou mudança de estado. Depois que o servidor faz o seu processamento, o estado apropriado, ou a parte desse estado que interessa na ocasião, é retornada ao cliente via *headers*, *status* e o *response body*.

Com REST, o cliente deve incluir toda informação que o servidor precisa para processar a requisição, reenviando as informações necessárias a cada nova requisição. Independência do estado da aplicação provê grande escalabilidade, já que o servidor não mantém o estado da aplicação. Com isso, balanceadores de carga não precisam se preocupar com sessões de usuário.

- **Cacheabilidade**

Como é intrínseco a *World Wide Web*, clientes podem fazer *cache* das respostas de requisições HTTP. Porém, as *Responses* devem, implícita ou explicitamente, se autodefinir como *cacheable*, ou não, visando prevenir reuso de dados antigos ou inapropriados em respostas a futuras requisições. Quando bem gerenciado, o cache pode eliminar, parcial ou completamente, algumas interações cliente-servidor, melhorando a escalabilidade e a performance da aplicação como um todo.

- **Cliente-Servidor**

Separação de responsabilidades é o princípio por trás da restrição Cliente-Servidor. É a interface uniforme; a fronteira entre o cliente e o servidor. Essa separação de conceitos implica que, por exemplo, clientes não devem tratar de armazenamento de dado, assunto que deve permanecer interno em cada servidor. Dessa forma, a portabilidade de código cliente é melhorada. Já o servidor, não se preocupa com a interface do usuário, nem tampouco com o estado do mesmo, tornando-se os servidores, muito mais escaláveis e simples. Servidores e clientes podem ser desenvolvidos e publicados (*deploy*) de forma independente, desde que a interface de comunicação (REST) não seja alterada. Essa separação de responsabilidades é o que permite que os componentes da arquitetura se desenvolvam de forma independente, suportando, dessa forma, o requerimento de escala intrínseco à Internet.

- **Sistema em camadas**

O cliente, normalmente, não sabe quando ele está conectado diretamente ao servidor final ou a um intermediário ao longo do caminho. Servidores intermediários podem melhorar a escalabilidade de um sistema permitindo balanceamento de carga e provendo *caches* compartilhados. Camadas podem reforçar as políticas de segurança.

- **Código sob demanda (opcional)**

Servidores podem, eventualmente, estender ou especificar as funcionalidades de um cliente ao transferir código para que esse cliente o execute (através de um *plugin* ou extensão do navegador). Exemplos disso incluem componentes compilados, como Java applets e também *client-side scripts* como o JavaScript.

Implementar essas restrições e seguir o estilo de arquitetura REST irá permitir a qualquer tipo de sistema distribuído em rede (*distributed hypermedia system*) ter diversos atributos desejados, como performance, escalabilidade, simplicidade, modificabilidade, visibilidade, portabilidade e confiabilidade.

#### 4) Proposta

A proposta do projeto é definir a arquitetura, modelar e implementar um guia de restaurantes do distrito de Penedo, na cidade de Itatiaia - RJ, utilizando as tecnologias abordadas no segundo capítulo deste trabalho, e seguindo o estilo arquitetural REST, abordado no terceiro capítulo do mesmo.

Tais tecnologias foram escolhidas por serem utilizadas nas aplicações *Web* de ponta, que representam o estado da arte em termos de tecnologia e desenvolvimento na Internet.

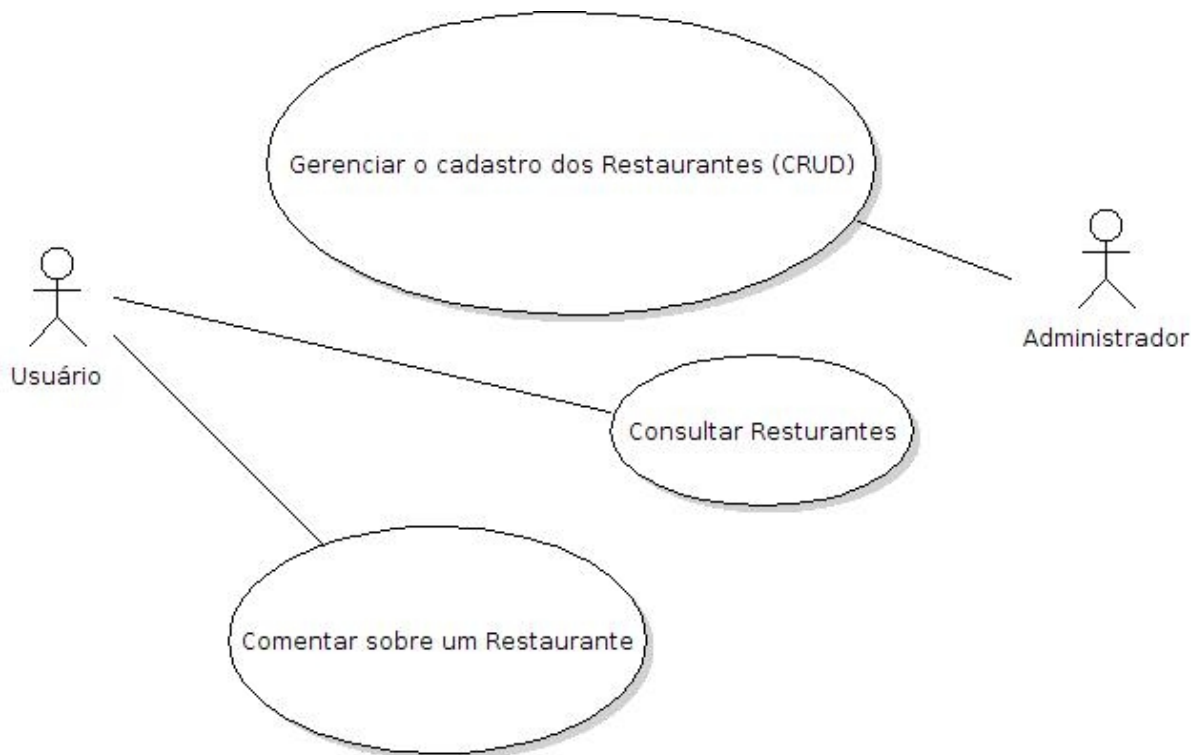
A arquitetura do sistema e, naturalmente, a comunicação entre seus componentes (subsistemas), foi, desde o início, o ponto central do projeto e guiou grande parte das decisões de tecnologia.

Foi buscada uma arquitetura flexível e que permitisse a evolução do sistema como um todo, bem como permitisse que determinado módulo ou subsistema pudesse ser trocado ou implementado de outra forma sem afetar o restante do sistema. Naturalmente se deu a decisão de desenvolver o projeto utilizando tecnologias escaláveis e que dessem suporte a esse paradigma de arquitetura distribuída, como HTTP e NodeJS.

Para atingir o objetivo de ser distribuído e escalável, o projeto é dividido em duas grande partes, o lado-servidor e o lado-cliente. A comunicação entre as partes se dá através do uso de uma API REST exposta pelo servidor.

No lado servidor foi construída uma API REST baseada em NodeJS, utilizando, para tal, o framework Express. A API REST é a responsável por acessar a camada de persistência de dados, esta implementada com o uso do banco de dados MongoDB. Também é a API quem acessa o serviço de armazenamento de arquivos (S3 da Amazon Web Services) para gravar as imagens utilizadas no sistema. (*AWS | Amazon Simple Storage Service (S3) - Online Cloud Storage, 2014*)

O lado cliente também se divide em duas partes principais, duas aplicações (subsistemas) JavaScript completamente independentes: uma que é responsável pela exibição dos dados do sistema para os clientes finais, e que será consumida pelo público em geral; e outra aplicação, de acesso restrito, específica para a administração dos dados. As duas aplicações JavaScript são projetadas para serem desenvolvidas com base nos *frameworks* AngularJS e Bootstrap. Na Figura 1, diagrama de casos de uso do sistema. Após a figura seguem as descrições, em alto nível, dos casos de uso.



**Figura 1 - Diagrama de casos de uso do sistema**

**Caso de Uso:** Gerenciar o cadastro dos Restaurantes (CRUD)

**Ator:** Administrador

**Descrição:** O administrador, através da interface administrativa da aplicação (ou de sua API), lista os restaurantes e exibe detalhes, edita, cria ou exclui determinado Restaurante.

**Caso de Uso:** Consultar Restaurantes

**Ator:** Usuário

**Descrição:** O usuário, através da interface de cliente final da aplicação, lista os restaurantes e exibe detalhes de determinado Restaurante.

**Caso de Uso:** Comentar sobre um Restaurante

**Ator:** Usuário

**Descrição:** O usuário, através da interface de cliente final da aplicação, publica um comentário sobre determinado Restaurante.

## 4.1) Arquitetura

O sistema se divide em duas partes principais, o lado-servidor e o lado-cliente, conforme mostra a Figura 2. O lado-cliente se sub-divide em duas partes, o *front-end* da administração do sistema e o *front-end* para o público (o guia em si, que será consumido pelos usuários finais), ambos consumindo a mesma API, exposta pelo lado servidor. A persistência dos dados se dá em uma instância do banco de dados MongoDB, em documentos no formato BSON, um *encode* binário do JSON. Apenas a API acessa o banco de dados. É através da API que todos os outros módulos escrevem e lêem os dados.

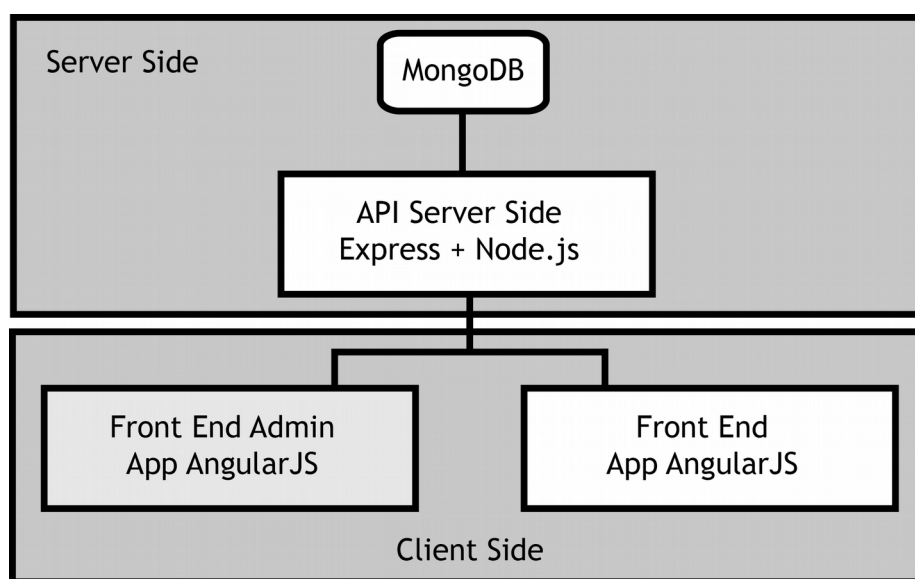


Figura 2 - Visão geral da arquitetura do sistema

### 4.1.1) Arquitetura do servidor (*back-end*)

A arquitetura de *Back End* do sistema é implementada em Node.js e exposta através de uma API REST. A API é criada com um servidor Express que acessa uma instância MongoDB (banco de dados não relacional). Essa instância do banco de dados é acessada apenas pela API, para persistência dos dados.

O servidor Express (*framework web* para Node.js) expõe rotas HTTP para criação, exclusão, consultas, e alterações dos recursos. Essas rotas seguem o estilo arquitetural REST.



A instância de MongoDB, acessada apenas pelo servidor Node.js, é quem faz a persistência dos dados em entidades no formato BSON (*Binary JSON*). Além de expor as rotas de persistência, o servidor Express também implementa autenticação, com apoio da biblioteca Passport, permitindo criação de usuários e login, sempre a partir do Facebook.

#### 4.1.2) Arquitetura dos clientes (*front end*)

As interfaces de lado cliente (administração e cliente final) do sistema são interfaces *Web*, implementadas como *Single Page Application*<sup>7</sup>, utilizando o AngularJS como *framework* de JavaScript, e o Bootstrap como *framework* de CSS, este provendo o sistema de *grid* para o *layout* e a funcionalidade de *layout* responsivo<sup>8</sup>. Os dois subsistemas de *front end* citados são executados no *browser* do cliente.

Para utilizar e manipular os dados, os subsistemas fazem requisições à API REST exposta pelo servidor *back end*. Os subsistemas de *front end*, ao usar o AngularJS, são descritos em termos de injeção de dependências. Todos os seus módulos se comunicam através do requerimento de outros módulos. Dessa forma, as dependências de cada módulo ficam muito evidentes, o que facilita a organização e facilita uma arquitetura onde cada classe tenha uma função única e bem definida.

#### 4.1.3) Arquitetura de implantação (*deploy*)

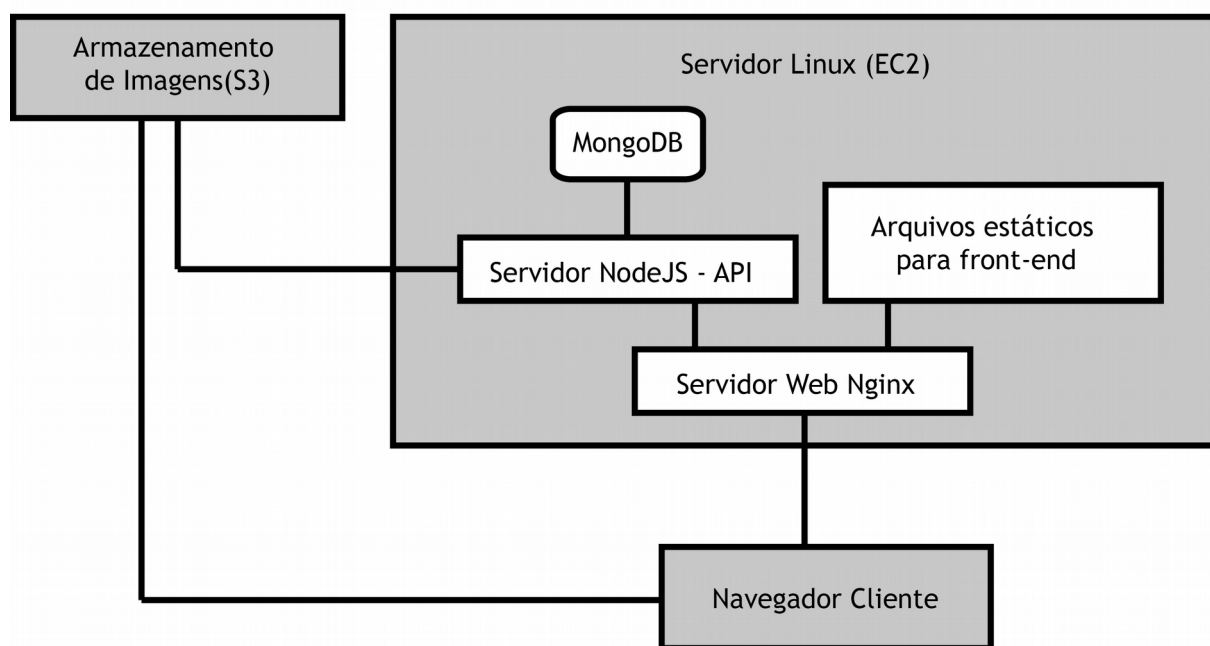
O Sistema foi publicado na internet utilizando os serviços da plataforma Amazon Web Services (AWS), que oferece infra-estrutura escalável como serviço. Na AWS foram utilizados 3 serviços principais.

O S3 é um armazenador de arquivos, que é onde ficam as imagens exibidas pelo site. O EC2 (*AWS | Amazon Elastic Compute Cloud (EC2) - Scalable Cloud Hosting, 2014*) é um serviço que oferece servidores virtuais. Lá, o sistema usa uma instância Linux, onde é executado o código de *server-side* e, onde também, é entregue o *front-end* do site (por questão de escala e redução de custos operacionais, o *server-side* e o *front-end*, que poderiam estar em ambientes completamente separados, são entregues na mesma máquina).

---

<sup>7</sup> Aplicação *Web* que é executada em uma única página, com o fim de prover uma experiência de uso mais fluida.

<sup>8</sup> *Layout* que adapta sua exibição de acordo com o ambiente (dispositivo) de visualização.



**Figura 3 - Visão geral da arquitetura de implantação**

Além de executar a API e entregar o *front end*, essa máquina Linux também executa o Banco de Dados MongoDB (o mesmo poderia estar separado). O último serviço que usamos do AWS é o Route53 (*AWS | Amazon Route 53 - Domain Name Server - DNS Service, 2014*), que é um serviço de DNS. É ele quem resolve o nome do domínio da aplicação para o IP do servidor da mesma no EC2.

Como um único servidor responde pelo *server-side* e pelo *front-end* foi necessário instalar um servidor *Web* de borda, no caso um Nginx que, a partir da URL requisitada, direciona a requisição para a aplicação correta. Se for uma requisição para o *front-end*, o próprio Nginx responde à requisição, enviando os arquivos solicitados. Já se for uma solicitação à API, ele faz um *proxy* para outra porta da máquina onde está sendo executado o servidor NodeJS, para que o mesmo possa tratar a requisição. É o servidor NodeJS quem acessa a camada de persistência de dados. Um diagrama simplificado do esquema de implantação pode ser visto na Figura 3.

O processo de implantação do sistema (*deploy*) é feito através de acesso SSH<sup>9</sup> ao servidor Linux, utilizando o GIT<sup>10</sup> para clonar e atualizar o código. GIT também é a

<sup>9</sup> SSH é um protocolo de rede criptografado que permite linha de comando remota.

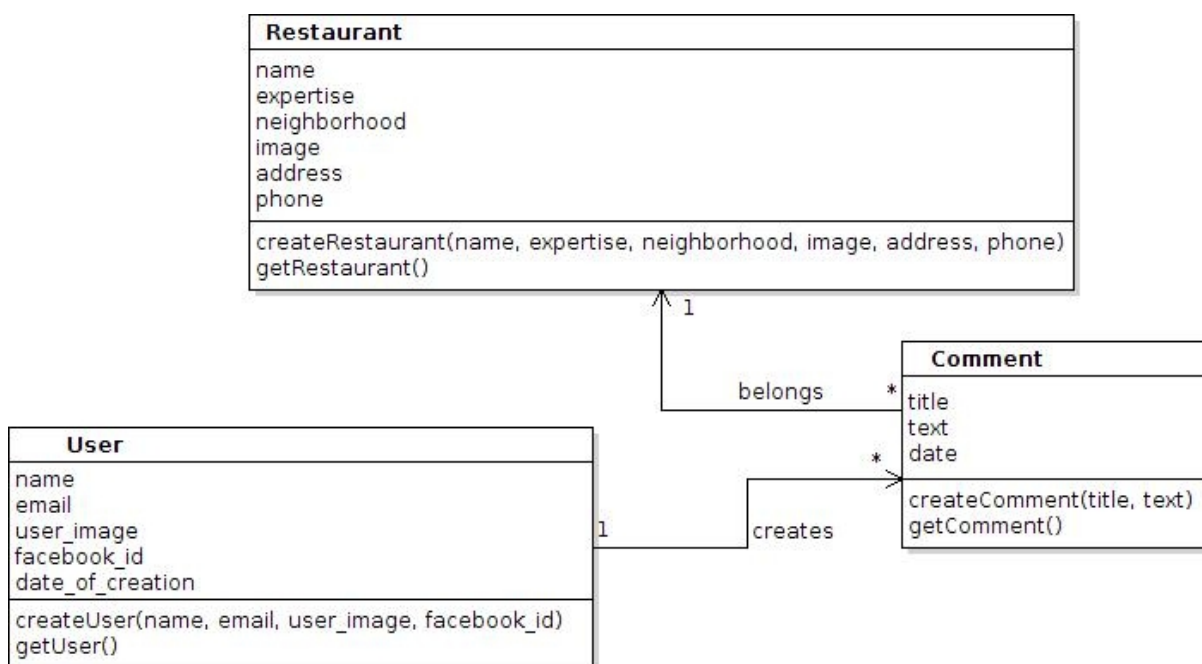
<sup>10</sup> Sistema de controle de versão distribuído.

ferramenta de versionamento da base de código utilizada em todo o projeto. O código fonte do projeto está disponível na *web* nos seguintes endereços:

- *front end* em: <<https://github.com/afonso-praca/restaurant-guide-ui>>
- *back end* em: <<https://github.com/afonso-praca/restaurant-guide-api>>

## 4.2) Modelagem de dados

O sistema proposto possui um modelo de dados bastante simples, composto por 3 entidades. São elas, o Usuário, que é criado a partir do login com uma conta do Facebook; o Restaurante, que é a entidade principal do sistema; e os Comentários, que os Usuários fazem nos Restaurantes. A Figura 4 mostra o diagrama de classes que representa essa modelagem.



**Figura 4 – Diagrama de classes**

### **4.3) Colaboração no Sistema**

No sistema proposto, uma vez que o modelo dos dados é bastante simples, uma parte fundamental do mesmo é a colaboração dos usuários, que no protótipo implementado se dá através dos comentários nos restaurantes.

Diversas outras ideias surgiram sobre possíveis usos da colaboração no sistema, porém, devido ao prazo para o desenvolvimento do projeto, as únicas implementadas foram os comentários e um mecanismo simples de recomendação de restaurantes.

De acordo com Pimentel e colaboradores (Pimentel, 2011), sistemas colaborativos são sistemas que dão suporte ao trabalho em grupo. No protótipo desenvolvido, esse trabalho é a criação de uma lista de comentários sobre os restaurantes, com o fim de auxiliar o usuário do sistema na escolha de um restaurante. O comentário só pode ser feito por um usuário autenticado, e a autenticação dos mesmos é feita através de sua conta no Facebook.

A opção do Facebook como autenticador foi uma decisão que, além de resolver a questão pontual de autenticar o usuário sem a necessidade da criação de senha, nos possibilita a futura implementação de muitas outras funcionalidades colaborativas, como, por exemplo, mostrar restaurantes que amigos comentaram, ou mesmo sugerir restaurantes para amigos. Essas ideias podem ser implementadas em trabalhos futuros.

Além do mecanismo de comentários, existe também um mecanismo simples de recomendação, que dentro do detalhe de um restaurante, lista outros restaurantes com a mesma especialidade, ordenados de acordo com o número de comentários (considerados aqui como uma avaliação, onde os mais comentados são os mais bem avaliados) que cada restaurante possui.

#### 4.4) Definição da estrutura de rotas da API

A tabela abaixo mostra as rotas da API REST, com os respectivos verbos HTTP que cada rota aceita e a descrição das ações que cada combinação de rota e verbo gera. O sistema lida com apenas três recursos: o restaurante, o comentário, e o usuário.

ROTA	VERBO HTTP	DESCRIÇÃO
/api/restaurants	GET	Retorna a lista de restaurantes.
/api/restaurants	POST	Adiciona novo restaurante na lista.
/api/restaurants/:id	GET	Retorna os detalhes de um restaurante e a lista de restaurantes similares recomendados.
/api/restaurants/:id	PUT	Atualiza os dados de um restaurante da lista.
/api/restaurants/:id	DELETE	Remove um restaurante da lista.
/api/restaurants/comments	POST	Cria um novo comentário.
/api/auth/facebook	GET	Abre a tela para <i>login</i> no facebook.
/api/auth/success	GET	Rota chamada pelo Facebook, após o mesmo autenticar o usuário. No retorno dessa rota volta um objeto de <i>user</i> do Facebook. Se ele já existir na base, o <i>login é permitido</i> , senão cria-se o usuário na base local e permite-se o <i>login</i> .
/api/auth	DELETE	Rota que faz <i>logout</i> do usuário no sistema

## 5) Desenvolvimento

O desenvolvimento do projeto se deu seguindo as seguintes etapas. A primeira delas foi, após a definição da arquitetura e do modelo de dados, a confecção de alguns *wireframes* (esboços em HTML) das páginas que a aplicação deveria exibir, todas elas baseadas no Bootstrap, o mais popular *framework* de CSS da *Web*, já abordado no capítulo 2. Na Figura 5, um dos primeiros protótipos HTML do *front end* para usuários finais do sistema. Como usual, essas telas evoluíram e mudaram muito no decorrer do projeto, não reduzindo a importância das mesmas. Foi muito importante esse primeiro momento de prototipagem, pois foi nele que o produto começou a tomar forma e diversas questões e ideias surgiram.

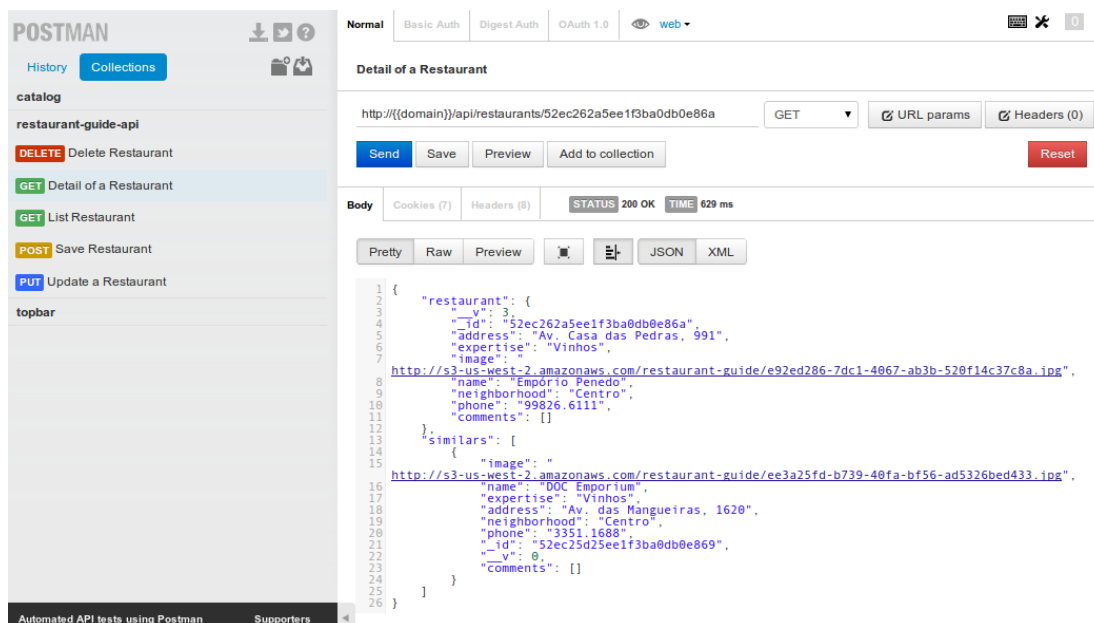


**Figura 5 - protótipo HTML do *front end* para usuários finais do sistema**

Paralelo ao desenvolvimento dos *wireframes*, foi iniciado o desenvolvimento da API REST *server side*, essa que viria a fornecer os dados para o *front end*, para que o mesmo se tornasse dinâmico e não exibisse dados fixos, como os primeiros protótipos fizeram. Como uma terceira frente, foi feita também a modelagem do banco de dados MongoDB, baseado no modelo de dados definido na arquitetura.

A decisão de implementar a API e toda a camada *server side* em Node.js foi tomada tanto pela praticidade e rapidez de desenvolvimento que a linguagem oferece, como também pela ativa comunidade *open source* que colabora com a mesma. Durante o desenvolvimento da API, em Node.js, foi preciso usar uma ferramenta para testar a mesma. Para tal, foi usado um aplicativo para o navegador Google Chrome chamado Postman, um cliente para APIs

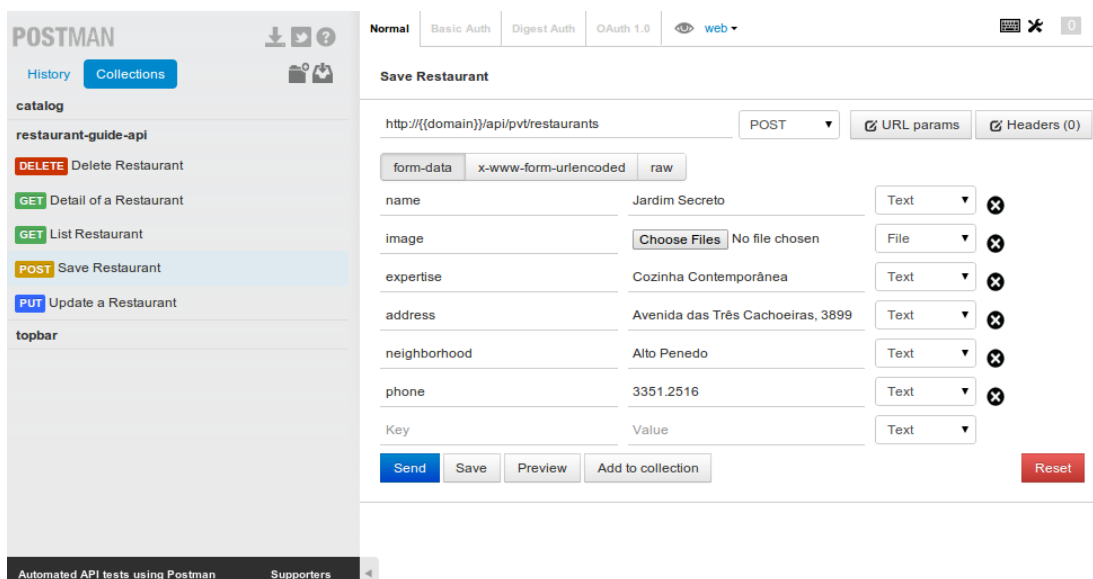
REST. No Postman pode-se criar uma coleção de requisições HTTP, e testar as mesmas. Nele foram criadas chamadas para cada uma das rotas da API. Na Figura 6, uma tela do Postman.



**Figura 6 - Tela do PostMan, REST Client.**

**Nesse exemplo testando a rota de detalhes de um restaurante.**

Devido a impossibilidade de tempo hábil, apenas o subsistema de *front end* destinado ao público teve seu protótipo desenvolvido. Porém, pela flexibilidade que a arquitetura forneceu, foi possível contornar a falta do subsistema de administração dos dados, através do uso de um sistema de terceiros (o Postman), que é um cliente para APIs REST. Foi o Postman quem escreveu os dados dos restaurantes, invocando diretamente as rotas da API. Na Figura 7, uma tela do Postman que cadastra os dados na API. O Postman, no projeto, representou o papel da interface administrativa do sistema. De qualquer forma, a falta da interface de administração é uma lacuna a ser preenchida e, a construção da mesma, é sugerida como um trabalho futuro.



**Figura 7 - Tela do PostMan, cliente para APIs REST que foi utilizado tanto para testar as rotas da API como para cadastrar os dados na mesma, fazendo as vezes de uma interface administrativa.**

Com os protótipos das telas prontos, a API REST funcionando, com as rotas expostas, escrevendo e lendo no banco de dados, a próxima etapa foi a implementação do *front end* do subsistema para os usuários finais, em AngularJS. Os HTMLs dos protótipos foram convertidos em *views* no projeto AngularJS, e alterados com a adição dos atributos especiais para a funcionalidade de *data-binding*. Foram criados também, nessa etapa, os códigos JavaScript de acesso à API e de controle do subsistema. Durante a codificação em AngularJS os protótipos sofreram diversas melhorias e alterações. Na Figura 8, a página inicial do subsistema de *front-end* para o cliente final já implementado em AngularJS, visitada a partir de um *desktop*. A mesma interface é mostrada posteriormente, na Figura 9, porém, acessada a partir de um dispositivo móvel. As duas imagens mostram a funcionalidade de *layout* responsivo. Com uma única base de código, a interface se adapta a diferentes dispositivos.



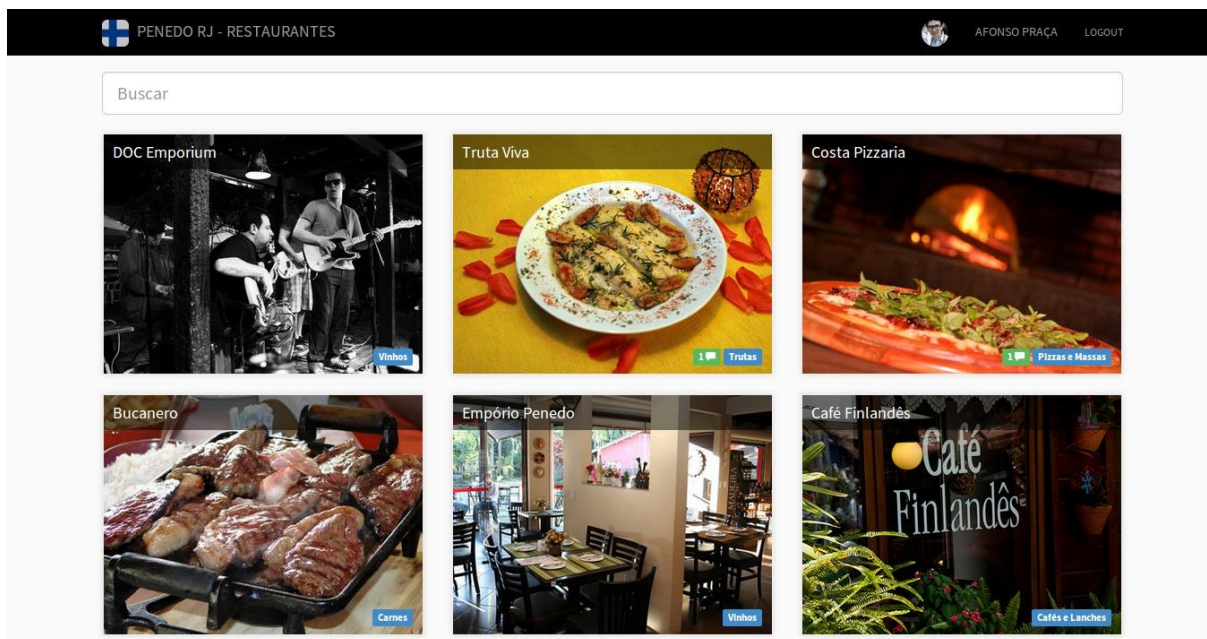


Figura 8 - Página inicial do módulo *front-end* para o cliente final, acessada a partir de um *desktop*

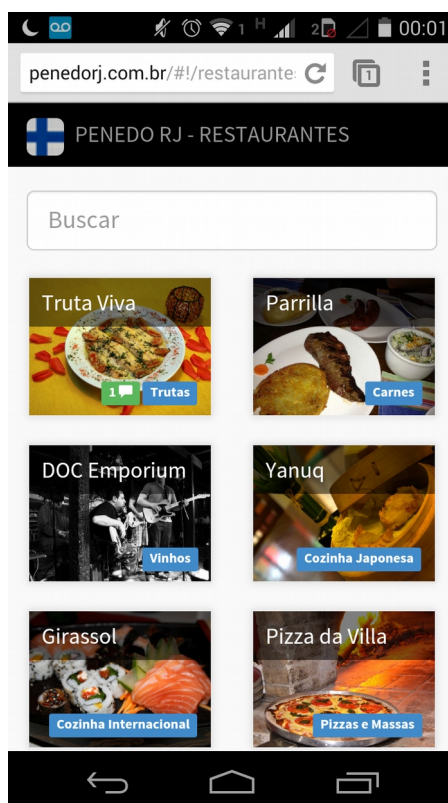
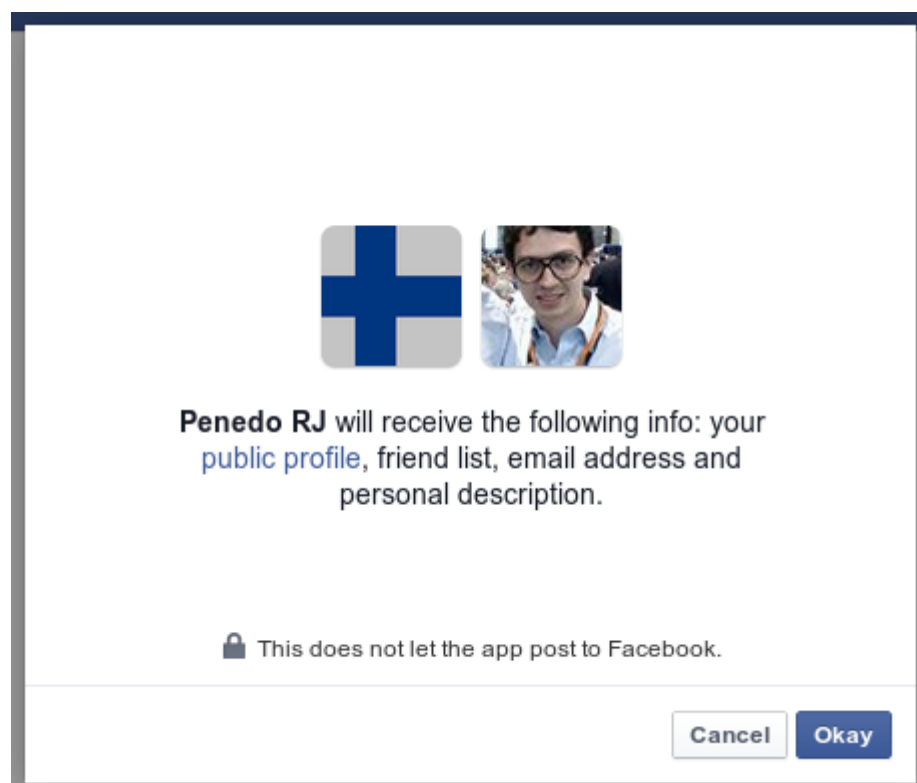


Figura 9 - Página inicial do sistema, exibida em um dispositivo móvel

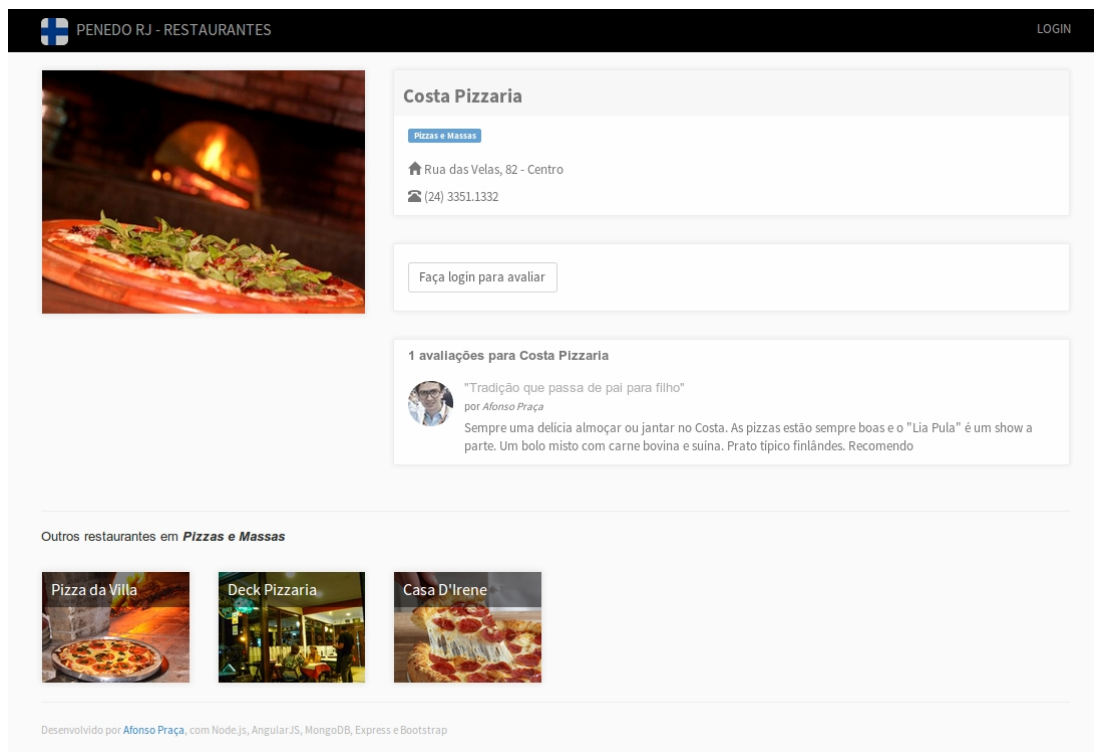
Durante o desenvolvimento da API, uma das funcionalidades implementadas foi o *login* do usuário a partir da conta do mesmo no Facebook. A opção de a única forma de autenticação no sistema ser essa, se deve ao fato do grande interesse na colaboração que o sistema tem e em como o usuário se logar com o Facebook pode colaborar com isso, conforme citado no item 4.3 (Colaboração no Sistema). Na Figura 10, detalhe da tela de *login* com o Facebook.



**Figura 10 - Página que pede autorização ao Facebook para autenticação**

Com a funcionalidade de *login* implementada e o subsistema já funcionando em AngularJS, a próxima etapa foi a implementação da área de comentários. Nela há um formulário para submissão de um comentário e a lista dos comentários já submetidos, ordenados em forma cronológica, do mais atual para o mais antigo. Na Figura 11, a página de detalhe do restaurante que, além da lista de comentários, conta também com a lista de restaurantes similares, onde são exibidos outros restaurantes da mesma categoria do restaurante em questão ordenados de forma decrescente de acordo com o número de

comentários que cada um possui. Na Figura 12, detalhe do formulário de submissão de comentário, disponibilizado apenas para usuários cadastrados e autenticados.



**Figura 11 - Página de detalhe de um restaurante.**


**Título do comentário**

**Comentário**

Adicione aqui seu comentário

Avaliar

**1 avaliações para Truta Viva**



"Viva a Truta!"  
por *Daniela Soria*

Falando em comer trutas em Penedo, não há nenhum igual. Truta com classe. Sem falar na natureza.

**Figura 12 - Detalhe do formulário de comentários.**

## 6) Conclusão e trabalhos futuros

Nesse trabalho foi apresentado, através do projeto e implementação de um protótipo funcional, a arquitetura e o desenvolvimento de uma aplicação *Web* distribuída baseada em JavaScript. No protótipo desenvolvido, o *back end* foi implementado em Node.js, a comunicação entre o *front end* e o *back end* se deu através de serviços expostos pelo *back end* em uma API REST, e o *front end* foi implementado usando os *frameworks* AngularJS e Bootstrap.

Apresentou-se ainda, brevemente, as tecnologias utilizadas e as principais características de uma API REST. Em seguida, foi exposta a arquitetura da solução proposta, englobando a arquitetura do *back end*, do *front end*, a modelagem dos dados e a arquitetura de implantação (*deploy*). O protótipo funcional desenvolvido está implantado (publicado) na infra-estrutura da AWS, um provedor de infra-estrutura como serviço, e pode ser acessado através do seguinte endereço: <<http://penedorj.com.br/>>.

O protótipo desenvolvido foi um guia de restaurantes. Com modelo de dados simples, a colaboração é parte fundamental do mesmo, se dando através de comentários dos usuários nos restaurantes. O protótipo de sistema é dividido em 3 subsistemas: um *back end* que expõe uma API REST; e dois clientes *front end*, um que é o guia, para o usuário final, e outro que é a interface de administração dos dados (esse não foi implementado devido a impossibilidade de tempo hábil para tal). O código fonte dos dois subsistemas implementados encontram-se disponíveis nos seguintes endereços: <<https://github.com/afonso-praca/restaurant-guide-ui>> e <<https://github.com/afonso-praca/restaurant-guide-api>>.

Levando-se em conta que os sistemas corporativos são distribuídos, compostos de diversos subsistemas, e as integrações entre sistemas são usuais e importantes para os negócios, a principal conclusão é que o estilo arquitetural REST se torna uma boa opção para o desenvolvimento de sistemas, devido a capacidade que ele traz consigo de construir sistemas escaláveis e flexíveis, capazes de se adaptar a mudanças de requisitos e de tecnologias.

A construção do subsistema *front end* de administração (do sistema proposto) é uma sugestão de trabalho futuro, bem como a implementação de um mecanismo de avaliação dos restaurantes, onde o usuário possa atribuir notas aos mesmos.

## Referências Bibliográficas

AngularJS: API Reference

Disponível em: <<http://docs.angularjs.org/api>>

Acesso em: junho de 2014

AWS | Amazon Elastic Compute Cloud (EC2) - Scalable Cloud Hosting

Disponível em: <<http://aws.amazon.com/ec2/>>

Acesso em: junho de 2014

AWS | Amazon Route 53 - Domain Name Server - DNS Service

Disponível em: <<http://aws.amazon.com/route53/>>

Acesso em: junho de 2014

AWS | Amazon Simple Storage Service (S3) - Online Cloud Storage

Disponível em: <<http://aws.amazon.com/s3/>>

Acesso em: junho de 2014

Bootstrap - Framework de CSS

Disponível em: <<http://getbootstrap.com/>>

Acesso em: junho de 2014

BSON - Binary JSON

Disponível em: <<http://bsonspec.org>>

Acesso em: junho de 2014

Especificação do HTTP/1.1

Disponível em: <<http://www.w3.org/Protocols/rfc2616/rfc2616.html>>

Acesso em: maio de 2014

Express - web application framework for node

Disponível em: <<http://expressjs.com/>>

Acesso em: junho de 2014

FIELDING, ROY THOMAS. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine, 2000.

JavaScript - Mozilla Developer Network

Disponível em: <<https://developer.mozilla.org/en/docs/Web/JavaScript>>

Acesso em: junho de 2014

MASSE, MARK. REST API Design Rulebook. Estados Unidos: O'Reilly Media, Outubro de 2011. 116 páginas.

MongoDB manual

Disponível em: <<http://docs.mongodb.org/manual/>>

Acesso em: junho de 2014

Node.js platform

Disponível em: <<http://nodejs.org/>>

Acesso em: junho de 2014

Pimentel, M., Fuks, H. “Sistemas Colaborativos”. Elsevier : Rio de Janeiro, 2011.

REST API Tutorial

Disponível em: <<http://www.restapitutorial.com/>>

Acesso em: junho de 2014

W3C - Especificação do HTML5

Disponível em: <<http://www.w3.org/TR/html5/>>

Acesso em: junho de 2014