CENTRO PAULA SOUZA
GOVERNO DO ESTADO DE SÃO PAULO

Faculdade de Tecnologia de Americana

Curso de Bacharelado em Análise de Sistemas e Tecnologia da Informação

DESENVOLVIMENTO DE APIS BASEADAS EM REST PARA INTEGRAÇÃO E CONSTRUÇÃO DE APLICAÇÕES

Marcos Vinícius Souza Campos

Americana, SP

2013

CENTRO PAULA SOUZA

GOVERNO DO ESTADO DE SÃO PAULO

Faculdade de Tecnologia de Americana

Curso de Bacharelado em Análise de Sistemas e Tecnologia da Informação

DESENVOLVIMENTO DE APIS BASEADAS EM REST PARA INTEGRAÇÃO E CONSTRUÇÃO DE APLICAÇÕES

Marcos Vinícius Souza Campos

marcos.campos@sensedia.com

Trabalho de Conclusão de Curso desenvolvido em cumprimento à exigência curricular do Curso Bacharelado em Análise de Sistemas e Tecnologia da Informação, sob a orientação do Prof. Alberto Martins Júnior.

Área: Programação Web, Arquitetura de Software.

Americana, SP

2013

AGRADECIMENTOS

Primeiramente devo agradecer a Deus por ter me dado força e motivação para conseguir seguir sempre em frente.

Em seguida tenho pleno agradecimento aos meus familiares e principalmente a minha mãe, por ter confiado em mim e ter entendido minhas razões, dando a oportunidade para que eu tivesse total dedicação a obra realizada.

Agradeço também ao Prof. Alberto Martins Júnior por ter disponibilizado seu tempo para me orientar, pelo voto de confiança em me aceitar como orientando e pelas palavras de incentivo que me dera durante o desenvolvimento, pois me motivaram nos momentos em que perdia a confiança.

Gostaria de agradecer também a todos os colegas da empresa Sensedia pelo surgimento da oportunidade de trabalhar em projetos que foram importantes, não só pelo conhecimento adquirido em um assunto novo, mas por possibilitar a definição do tema e conteúdo desta obra.

Por fim, agradeço a todos os amigos e amigas que caminharam junto a mim não só através do desenvolvimento do trabalho, mas por todos os quatro anos que passamos juntos durante o período de faculdade, sem a ajuda de todos vocês eu não chegaria até este ponto.

“Nas grandes batalhas da vida o primeiro passo para a vitória é o desejo de vencer” – Mahatma Gandhi

RESUMO

Dentro do mundo interconectado, é necessária uma maneira de promover a comunicação entre diversos tipos de sistemas baseados nas mais diversas arquiteturas. APIs surgiram como uma maneira de facilitar a comunicação entre esses sistemas, abrindo portas para um novo mercado de desenvolvimento e integração de aplicações e plataformas.

Porém, uma API deve promover uma interface bem definida para garantir que a interação entre as aplicações e sistemas seja íntegra. Neste contexto temos a aplicação da arquitetura REST em conjunto com outras tecnologias para sistemas distribuidos como HTTP e JSON, para construção de APIs que promovem comunicação e troca de dados maneira simples e rápida.

Entretanto, o conhecimento sobre como utilizar a arquitetura REST é pouco difundido atualmente. A maioria do que é aplicado esta baseado na cultura popular, tornado REST um estilo arquitetural com diferentes vertentes, problematizando a implementação de APIs que se baseam em seus estilos.

Uma forma de resolver este problema é estabelecendo uma série de principios e boas práticas de implementação, de forma a tornar a construção de APIs um processo facil e ágil.

Esta obra trata dos conceitos relacionados aos princípios que mostram a importância da produção de APIs para o mercado de TI, as tecnologias envolvidas na arquitetura REST, os princípios para a construção de APIs. Tendo o objetivo a promoção destas abordagens como novos métodos de desenvolvimento de sistemas e aplicativos distríbuidos, mostrando as capacidades que uma API pode fornecer.

Palavras Chave: Integração de Sistemas, Desenvolvimento de aplicações, Novas Tendências de TI.

ABSTRACT

In a world interconnected, it makes necessary to find a way to promote connections through different kinds of systems and architectures. APIs emerged as a way to make easer this kind of communications, opening doors to new development markets and the integrations between applications and platforms. However, an API should promote a well defined interface to ensure that the integration between applications and systems can be on a straightaway. In this context, we have an architecture application called REST, that working with other distributed system technologies as HTTP and JSON might build APIs that provide simple and fast ways of communication and data's trade.

Unfortunately, the 'how to use' knowledge of the REST architecture isn't broadcasted enough. The majority of what is used or applied of this architecture is based on the common sense, making the REST a multifaceted architecture, making hard the APIs implementation's process that are based on this kind of styles.

Trying to solve this problem, makes necessary the set up of some principles and implementation's good ways of thinking to make the API building processes easier and efficient.

This work is going to talk about some principles related with the building processes of APIs and their necessity to the IT market. To complement this, the technologies related with the REST architecture are going to be explained as well, with the objective of approach new development methods to systems and distributed applications, always trying to show the capacity of an API based on REST architecture can provide.

Keywords: Systems Integration, Application Development, IT New Trends.

LISTA DE ABREVIATURAS E SIGLAS

ASCII	*American Standard Code for Information Interchange*
API	*Application Programming Interface*
CRUD	*Create Read Update Delete*
DOM	*Document Object Model*
EC2	*Elastic Compute Cloud*
ESPN	*Entertainment and Sports Programming Network*
HATEOS	*Hypermedia as the Engine of Application State*
HTML	*HyperText Markup Language*
HTTP	*HyperText Transfer Protocol*
IBM	*International Business Machines*
IETF	*Internet Engineering Task Force*
IP	*Internet Protocol*
JSON	*JavaScript Object Notation*
MTA	*Metropolitan Transit Authority*
NSA	*National Security Agency*
REST	*Representational State Transfer*
RPC	*Remote Protocol*
S3	*Simple Storage Service*
SaaS	*Software as a Service*
SAX	*Simple API for XML*
SOAP	*Simple Object Access Protocol*

SSL *Secure Socket Layer*

TCP *Transmission Control Protocol*

UDDI *Universal Description Discovery and Integration Services*

URI *Uniforme Resource Identifier*

URL *Universal Resource Locator*

WSDL *Web Service Description Language*

XML *eXtensible Markup Language*

LISTA DE TABELAS

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Desde o surgimento da internet entre as decadas de 1970 e 1980, o homem moderno descobriu novas maneiras de se comunicar através da transferência rápida de informações pela rede.

Porém, o mundo interconectado se tornou necessidade primária desde que a tecnologia avançou para o nível móvel. Atualmente a maior parte das pessoas faz uso de tecnologias como *Smartphones*, *Tablets* e outros dispositivos capazes de se conectar a internet. Segundo pesquisas de Allavareduarte (2012), no ano de 2012 o número de pessoas que usufruem de dispositivos móveis para acessar a internet é de 56,2% da população mundial, provando que os usuários estão percebendo as vantagens de conseguir se conectar de qualquer lugar.

Mas a motivação principal do uso de dispositivos deste tipo é a alta quantidade de aplicações existentes que garantem e facilitam o acesso a internet e seus conteúdos. Os mais procurados são aplicativos de redes sociais como o Facebook e Twitter, aplicativos para mobile *banking*, *downloads* de podcasts online, vídeos *streaming* como Youtube e Netflix entre outros.

Sendo assim, temos a visualização de um potêncial mercado de TI com novas fronteiras a partir destas aplicações. Contudo, o desafio está em atingir um número maior de clientes através das aplicações construídas, produzir aplicações que atendam a uma infinidade de plataformas disponíveis no mercado atual e garantir que o transporte de dados realizado por estas aplicações seja feito de maneira íntegra.

Tentando solucionar esta questão surgiu o conceito de APIs, elas têm o poder de encapsular as capacidades de fornecimento de dados providos por serviços de uma empresa em interfaces de programação bem definidas disponíveis via Internet.

Este conceito impulsionou o mercado de desenvolvimento de novas aplicações, principalmente para o público móvel. Porque estas interfaces disponibilizam informações de alto valor significativo de maneira fácil e rápida, utilizando tecnologias de programação web inovadoras, como é o caso da arquitetura REST e do padrão JSON.

Outro motivo interessante para o uso de APIs é o fato de que a maioria das aplicações desenvolvidas era feita por pequenas empresas ou programadores amadores que tinham ideias inovadoras sobre como utilizar os serviços providos por outras empresas.

Entretanto, o surgimento de conceitos novos pode ser problemático a partir do fato de que arquiteturas de desenvolvimento como REST não possuem padrões definidos de implementação e podem envolver uma série de outros conceitos. Portanto, de nada adianta fornecer uma API com serviços de alto valor agregado a seus clientes, sem que estas interfaces sejam desenvolvidas de maneira a facilitar a sua compreensão e uso.

Assim, esta obra tem o objetivo principal de relevar a importância sobre o uso de APIs no ambiente de TI atual, apresentar o que envolve a arquitetura REST e demostrar como obter uma API com uma interface bem desenvolvida utilizando boas práticas de implementação.

2 JUSTIFICATIVA

Atualmente, com a existência do conceito de *Cloud Computing*[1] e o crescimento da tecnologia móvel, houve a grande necessidade de tornar a conexão e interação entre diversos sistemas algo mais simplificado, pois a maioria dos dispositivos eletrônicos existentes tem a capacidade de se conectar a internet.

No ano de 2008, o número de dispositivos conectados (que engloba desde computadores pessoais, até aparelhos baseados na computação móvel) era maior que o número de pessoas no planeta Terra.

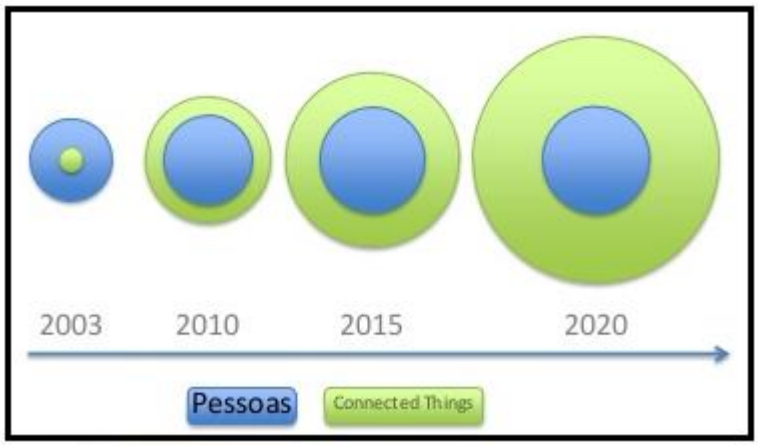


Figura 1 - Número de pessoas existente X Coisas conectadas à Internet
Fonte: Adaptado de BACILI (2012)

Com o objetivo de facilitar e agilizar a comunicação, surgiu o conceito de programação de APIs dentro do ambiente Web, onde as APIs possuem serviços que representam funções de negócio e podem ser expostas na internet para que programadores possam construir aplicações utilizando estes serviços.

O detalhe principal para a utilização de APIs dentro do contexto citado, é que o desenvolvedor não precisa necessariamente saber os detalhes da implementação do serviço. Segundo Franklin e Coustan (2009), essa abordagem deixa uma liberdade para que os programadores usem as funções de um sistema sem ter conhecimento direto de todos os processos envolvidos para execução do serviço, apenas manipulando os resultados de tal execução.

[1] Segundo Alecrim (2008), Cloud Computing "[...] se refere, essencialmente, à ideia de utilizarmos, em qualquer lugar e independente de plataforma, as mais variadas aplicações por meio da internet com a mesma facilidade de tê-las instaladas em nossos próprios computadores".

Conforme Jacobson et al (2011, p.6), API são serviços que por estarem dentro da internet ficam disponíveis 24 horas por dia, 7 dias por semana durante um periodo de 365 dias no ano, ou seja, os usuários podem acessar suas funcionalidades a qualquer momento e a API deve ser projetada para suportar uma grande demanda de usuários.

Contudo, a demanda por APIs não vem só da necessidade de integração entre sistemas. Para grandes corporações como Google, Twitter e Facebook, a transformação de serviços em um conjunto de APIs é uma oportunidade de negócios. Jacobson et al (2011, p.7) explica que uma API pode ser usada internamente para direcionar o desenvolvimento e gerenciamento de websites, aplicações móveis e outros produtos. Porém, quando seus serviços são disponibilizados para acesso público, as APIs servem como porta de entrada para a criação de novos sistemas e soluções tecnológicas que tem compatibilidade com inúmeros sistemas operacionais.

De acordo com Bacili (2012), existem seis motivos principais que levam a uma empresa a adotar a estratégia de APIs:

- Possibilita criação de novos modelos de negócio e ampliação de canais de comunicação;
- Torna a marca da empresa algo mais sólido e consolida os serviços como uma plataforma de desenvolvimento, aumentando o crescimento da própria empresa;
- Maior velocidade no desenvolvimento de aplicativos, a utilização APIs no desenvolvimento de aplicações minimiza em até 75% o tempo gasto para finalização dos projetos;
- Estabilização de uma estratégia para desenvolvimento para o mercado de aplicações móveis;
- Redução de custos operacionais quando a API é usada para melhoria dos processos internos da empresa, através de padronizações dos formatos que permitem a integração dos sistemas por eles utilizados;
- Maior inovação na criação de produtos e serviços, tanto interna quanto externamente.

O desenvolvimento de uma API para estes fins deve seguir alguns conceitos e práticas que incentivaram novos desenvolvedores a utilizarem os serviços desenvolvidos por uma empresa. Seguindo esse principio, temos a relevância da arquitetura REST dentro deste processo.

A utilização de REST é valida, pois quando focamos na produção de APIs para integração de sistemas, REST funciona exatamente como a ferramenta que irá promover a comunicação entre estes sistemas. Segundo Fielding (2000, p.76), REST é um híbrido de diversos conceitos relacionados à comunicação de sistemas distribuídos que são unidos para providenciar uma interface de programação única. Podemos citar alguns conceitos utilizados, como estilo de comunicação Cliente-Servidor via troca de mensagens por *request-response* e a utilização do protocolo HTTP para a troca, formatação e interligação dos sistemas integrados.

A vantagem em utilizar REST no desenvolvimento de API está no fato de que todos estes conceitos são conhecidos pela maioria dos programadores. Por usar apenas o protocolo HTTP para troca de mensagens, REST também dá liberdade de escolha da linguagem de programação no qual a API vai ser implementada.

Mulloy (2012, p.3) explica que REST, por ser apenas um estilo arquitetural, permite uma maior flexibilidade no desenvolvimento de uma API, porque possui uma série de teorias que garantem que a API vai retornar o máximo de resultados para quem consumir os serviços disponibilizados.

A Figura 2 mostra o uso cada vez mais frequente de REST na construção de APIs para integração de sistemas em comparação a arquiteturas mais antigas como SOAP e XML-RPC que estão ficando obsoletas, principalmente, por exigirem uma complexidade maior em sua utilização.

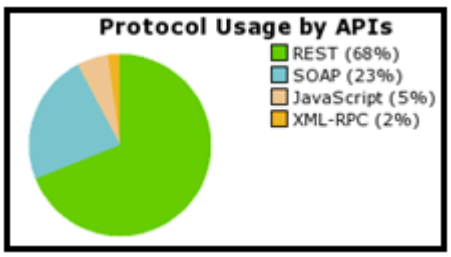


Figura 2 - Protocolos utilizados em APIs

Fonte: Adaptado de PROGRAMMABLEWEB (2013)

3 OBJETIVOS

O objetivo deste trabalho é mostrar a importância das APIs e sua utilização no mundo de TI atual, além de demonstrar quais são as maneiras de produzir uma API baseadas na arquitetura REST.

Para tal, será feita uma apresentação das tecnologias envolvidas dentro do conceito de APIs com o objetivo de facilitar a compreensão sobre seu funcionamento.

Também serão apresentadas formas e boas práticas no desenvolvimento de APIs com foco na arquitetura REST, mostrando maneiras de como usar estes princípios, buscando uma maneira de estabelecer um padrão no desenvolvimento de serviços web.

Por fim, será demostrado de forma prática, a utilização de uma API REST implementada na linguagem Java para Web, analisando como as boas práticas apresentadas foram utilizadas com o objetivo de provar os conceitos abordados.

4 METODOLOGIA

De acordo com pesquisas realizadas nos livros de Andrade (2009) temos a utilização das seguintes metodologias para o desenvolvimento deste trabalho:

- Pesquisa Básica: Agrega a apresentação de novos conhecimentos através do desenvolvimento científico, no qual será utilizada para apresentação dos conceitos introduzidos por este trabalho.
- Pesquisa Exploratória: Busca proporcionar familiaridade com o problema, com o intuito de torná-lo explícito ou a construir hipóteses. Envolve levantamento bibliográfico, entrevistas com pessoas que possuam experiências práticas com o assunto/problema pesquisado, análise de exemplos que estimulem a compreensão.
- Pesquisa Bibliográfica: Envolve a coleta de material a partir da leitura de diversos materiáis já publicados, como artigos, livros e outros trabalhos relacionados com os objetivos de servir como base teórica principal do projeto desenvolvido.

5 ELEMENTOS DE SISTEMAS WEB

Com o objetivo de melhorar a compreensão das técnicas usadas no restante do projeto, essa sessão tem como prioridade mostrar de maneira sumária alguns fundamentos usados tanto na arquitetura REST quanto nos conceitos de APIs para Web. Para tal abordaremos as tecnologias utilizadas no desenvolvimento de APIs dentro do ambiente Web, onde serão apresentados os fundamentos das tecnologias envolvidas para o desenvolvimento de APIs utilizando essa abordagem como JSON, HTTP, XML entre outras.

5.1 PROTOCOLOS E ARQUITETURAS DE COMUNICAÇÃO

O principal protocolo utilizado para transporte de dados é o HTTP em conjunto com o TCP/IP.

O HTTP é um protocolo de rede para a camada de aplicação Web, permite a transferência de dados entre redes de computadores e faz a comunicação entre estes computadores via troca de mensagens no modelo HTTP.

A forma como os recursos são adquiridos e transportados entre clientes e servidores, seguem o padrão *request-response*. Para exemplificar como o padrão funciona vamos supor que o recurso requerido por um cliente seja uma página Web presente em um site. Neste caso a comunicação seguiria da seguinte forma:

> “Quando um usuário requisita uma página Web (por exemplo, clica sobre um *hiperlink*), o browser envia ao servidor mensagens de requisição HTTP para os objetos da página. O servidor recebe as requisições e responde com mensagens de requisição HTTP que contêm os objetos” (KUROSE, 2006, p.69).

A Figura 3 ilustra melhor como a comunicação entre as partes ocorre dentro do protocolo:

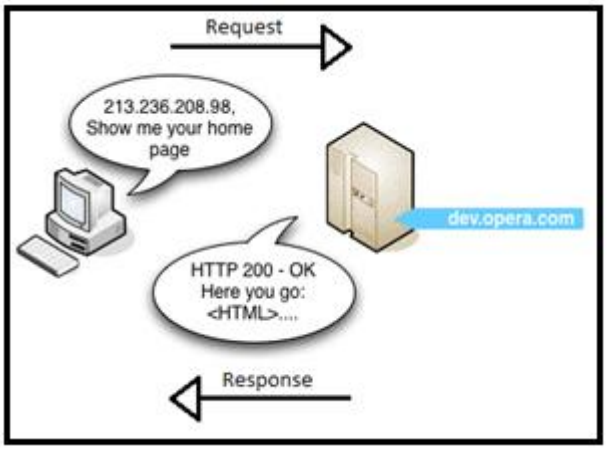


Figura 3 - Esquema de Comunicação padrão *request-reponse*

Fonte: Adaptado de LANE (2008)

Este estilo de comunicação é predominante da arquitetura cliente-servidor. Nesta arquitetura, segundo Kurose (2006, p.70), existem dois componentes principais, um hospedeiro chamado servidor, que providência todos os recursos e serviços e tem a função de responder todos os pedidos realizados por outros hospedeiros, chamados clientes.

A arquitetura cliente-servidor também define que os servidores devem ter um endereço fixo por onde os clientes possam se conectar. Esse endereço é chamado de endereço IP e a maneira como um cliente se conecta a um servidor é por um recurso chamado URL.

Para Lane (2008), toda a requisição que pode ser feita a um servidor é realizada ao se digitar seu endereço no browser, este endereço é chamado de URL.

Portanto, uma URL pode ser definida como a representação textual do endereço IP de um servidor, o que significa que para realizar a conexão ao servidor, o browser não utiliza o texto digitado em sua barra de endereços, como por exemplo http://www.google.com, e sim o endereço IP que está associado ao endereço textual passado.

A Figura 4 mostra um endereço de rede mapeado a um endereço IP.

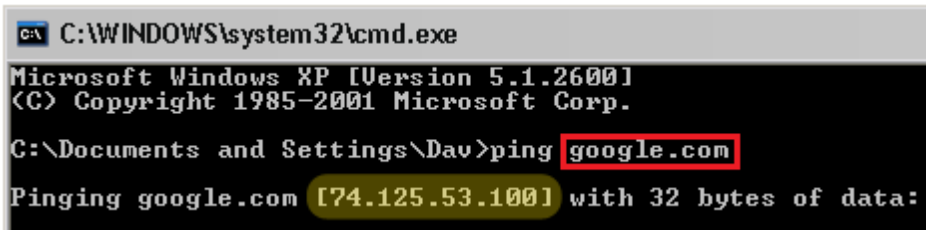


Figura 4 - Exemplo de associação entre URL e endereço IP

Fonte: Adaptado de TOTHEPC (2008)

Outro elemento importante na comunicação via internet são as mensagens enviadas entre clientes e servidores. Existem dois tipos de mensagens básicas: as mensagens de requisição e as mensagens de resposta.

- **Mensagens de Requisição**

  Tanto as mensagens de requisição quanto as mensagens de respostas são formatas no estilo ASCII e são padronizadas segundo normas do RFC 2616. Na Figura 5, nota-se um exemplo de mensagem de requisição e depois é explicado os elementos que a formam.

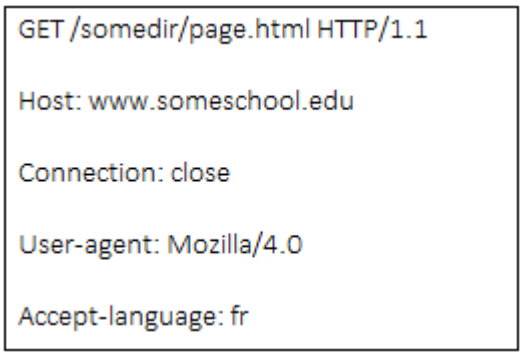


Figura 5 - Exemplo de mensagem de requisição HTTP

Fonte: Adaptado de KUROSE (2006, p.72-73)

Kurose (2006, p.73) destaca os elementos principais contidos em uma mensagem de requisição:

> “[..] a primeira linha de uma mensagem de requisição HTTP é denominada linha de requisição; as subseqüentes são denominadas linhas de cabeçalho. A linha de requisição tem três campos: o campo do método, o da URL e o da versão do HTTP.” (KUROSE, 2006, p.73)

Detaca-se que o primeiro campo é o mais importante dentro da arquitetura REST, pois ele contém informações sobre o método utilizado e onde adquirir os elementos solicitados. A Figura 5 mostra o emprego do método GET usado para adquirir informações do servidor, onde no nosso exemplo ele requisita o elemento page.html, presente no diretório /somedir do servidor www.someschool.edu.

- **Mensagens de Resposta**

  A seguir é apresentado um exemplo de mensagem de resposta, item importante para interpretarmos a resposta do servidor à requisição de determinada informação.

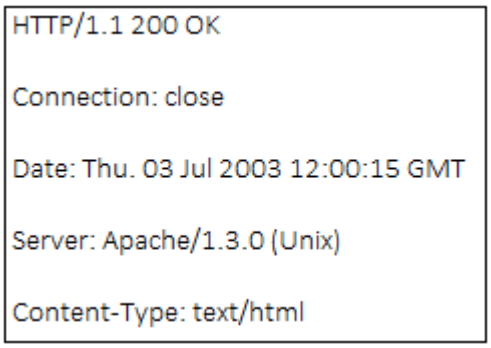


Figura 6 - Exemplo de mensagem de resposta HTTP
Fonte: Adaptado de KUROSE (2006, p.74)

Kurose (2006, p.74) define que a mensagem de resposta tem três seções: a linha de estado, seis linhas de cabeçalho e, em seguida, o corpo da entidade onde esta presente toda a informação que foi solicitada ao servidor.

Destacam-se alguns itens importantes dentro da mensagem de resposta. Em primeiro lugar, a linha de estado, onde temos três campos: o campo de versão do protocolo, que mostra exatamente qual a versão do protocolo o servidor está usando (HTTP/1.1), seguido de um código que representa se a requisição foi ou não bem sucedida (200) junto da mensagem que representa aquele estado (OK).

Outro item importante está nas linhas de cabeçalho e é chamado *Content-Type*. Ela define qual será a formatação dos dados presentes dentro do corpo da entidade, ou seja, qual é o formato dos dados que trafegam entre

cliente e servidor. Sendo que, tanto no cliente quanto no servidor os formatos configurados devem ser idênticos. Os itens a seguir tem a função de explicar a importância dos códigos de resposta e qual é a função do *Content-Type* para formatações de dados.

5.2 MÉTODOS HTTP

Dentro do protocolo HTTP existem alguns métodos que representam operações que podem ser feitas para transferência de dados. Nesta seção vamos explicar os quatro mais importantes de acordo com leituras de NSA (2012, p.4), Firmo (2012, p.2) e Richardson (2007, p.99).

GET: A proposta inicial do método GET é extrair informações sobre um recurso qualquer de acordo com a URI[2] solicitada, normalmente é equivalente ao método *Read* do CRUD[3]. As respostas do método GET vêm estruturadas no corpo da entidade de resposta e são formatadas de acordo com a representação identificada no atributo *Content-Type* presente no cabeçalho da mensagem.

POST: Este método é utilizado para criação e atualização de recursos, sendo que a atualização não é uma operação válida se estivermos trabalhando com o estilo *Pure REST,* conforme explicado no capítulo 7.2. Em REST é comumente usado para criação de recursos subordinados, ou seja, recursos individualizados que tem relação com recursos de hierarquia maior que normalmente pertencem a uma coleção, onde o cliente apenas tem que conhecer a URI pai para executar a operação de criação de um novo recurso.

PUT: Usamos este método para atualizar um recurso por completo ou criar um novo recurso caso ele não exista na coleção. Porém neste caso, o cliente passará a URI completa do recurso que será criado, onde no corpo da mensagem de requisição deve ser passada a representação do recurso com as informações que devem ser modificadas ou atualizadas.

DELETE: A proposta básica deste método é apenas apagar um recurso existente no servidor de acordo com a URI passada. O método deve responder ao

[2] Segundo Techterms.com, URIs tem a capacidade de identificar o nome ou o local onde se encontra um arquivo ou seus recursos. São similares as URLs, entretanto, fazem referência apenas a parte da URL que compõe o nome do recurso ou arquivo referênciado.

[3] De acordo com Jaccon (2010), CRUD é um conjunto de operações que tem a função de criar, recuperar, atualizar ou apagar informações de um banco de relacional

cliente com uma mensagem de sucesso ou fracasso indicando o estado da operação.

5.3 CÓDIGOS DE RESPOSTA HTTP

Para Fisher (2013), os códigos de resposta HTTP são códigos padronizados fornecidos por servidores, onde cada código auxilia na identificação de um estado ou problema ocorrido na requisição ou invocação de um recurso do servidor. Normalmente estes códigos são divididos por uma categoria de números, onde cada categoria indica um tipo de resposta do servidor.

Hall (2010, p.178), divide as categorias de códigos da seguinte maneira:

- 100-199: Códigos desta categoria são informativos, indicam que o cliente deve responder com alguma ação;
- 200-299: Valores que indicam que houve sucesso na requisição do recurso ou operação;
- 300-399: Indicam que o recurso solicitado foi movido para outro lugar, normalmente é adicionado o parâmetro *Location* no corpo da mensagem de resposta indicando qual é o novo local onde o recurso se encontra;
- 400-499: Indica que ocorreu algum erro no cliente durante a requisição ao servidor;
- 500-599: Indica que ocorreu um erro no servidor durante o processamento da requisição;

Podemos perceber que dentro das categorias podem existir inúmeros tipos de códigos de resposta, mas vamos exemplificar apenas os mais utilizados, onde, segundo Hall (2010, p.179) e Kurose (2006, p.75) os mais comuns são:

- 200 OK: Indica que a requisição foi bem sucedida e a operação requerida pelo cliente foi executada com sucesso;
- 202 *Accepted*: Indica que a requisição foi aceita pelo servidor, porém, a operação requirida por ele não foi completada;
- 400 *Bad Request*: Indica que o formato da mensagem de requisição não atende os padrões aceitos para que o servidor possa realizar uma operação;

- 401 *Unauthorized*: Significa que o cliente tentou acessar um recurso autenticado do servidor, mas não passou as credenciais corretas;
- 403 *Forbbiden*: Significa que o servidor recusou o pedido feito por um cliente, independente de autorização. Esse tipo de resposta é dado caso o recurso solicitado esteja corrompido, ou o acesso seja restrito;
- 404 *Not Found*: O código 404 é o mais famoso entre os códigos. Normalmente ocorre quando o recurso requisitado como um documento, página ou arquivo não está resente no servidor;
- 405 *Method Not Allowed*: Significa que o método HTTP (GET, POST, PUT ou DELETE) invocado por um cliente não é suportado pelo servidor.
- 415 *Unsuported Media Type*: Ocorre quando no corpo da mensagem de requisição é mandado um conteúdo em um formato diferente daquela que o servidor espera;
- 500 *Internal Server Error*: Acontece quando ocorre um erro dentro do servidor que impediu que a operação requisitada pelo cliente fosse realizada, onde não se sabe ao certo qual foi o erro causado;

Alguns destes códigos são usados no desenvolvimento de APIs, pois indicam aos programadores qual é o comportamento do servidor quando seus serviços são solicitados, possibilitando a criação de mensagens de validação de operações utilizando alguns dos tipos de códigos citados.

5.4 INTERNET MEDIA TYPES

Dentro dos protocolos de comunicação, temos que os tipos de dados que trafegam pela rede podem ser de inúmeros formatos. De acordo com Firmo (2012, p.3) estes dados podem ser textos estruturados, como arquivos no formato XML, JSON e ATOM. Páginas no formato HTML, documentos com extensões PDF ou DOC, imagens do tipo GIF ou JPEG e até arquivos de áudio e vídeo das mais diversas extensões. Cada um desses formatos que é transportado pela rede é chamado de *Media Type* e estão presentes no corpo da mensagem de resposta quando são requisitados.

Para definir esses formatos, o protocolo HTTP não considera a extensão do arquivo que será transportado, para definir qual é o *Media Type* do arquivo que vai

ser enviado entre cliente e servidor é utilizado o atributo *Content-Type* presente no cabeçalho da mensagem.

A função do *Media Type* é dizer à aplicação que irá receber a mensagem, como o conteúdo da dela deve ser manipulado.

Normalmente este atributo contém um valor que segue a seguinte estrutura: “tipo /subtipo”, onde “tipo” define qual o dado que será trafegado que pode ser *application*, *text*, *audio*, *image* entre outros e “subtipo” define qual estrutura de dados que o “tipo” irá seguir.

Abaixo, a partir dos exemplos mostrados por Wansen (2006, p.1-2) apresenta-se uma pequena lista de alguns dos *Media Types* mais utilizados:

- *application/json*: Manipula objetos JavaScript Object Notation;
- *application/javascript*: Manipula arquivos JavaScript;
- *application/octet-stream*: Geralmente manipula arquivos que não são associados a uma aplicação específica, apenas um conjunto arbitrário de binários;
- *text/html*: Retorna textos do tipo HTML;
- *text/xml*: Trabalha com textos do tipo XML;
- *text/plain*: Trabalha com dados de textos comuns;
- *application/x-www-form-urlencoded*: Manipula dados que vem de formulários de páginas HTML;

A seguir é mostrado um exemplo de requisição de um cliente feita via browser a um documento presente nos servidores do *Google Docs*.

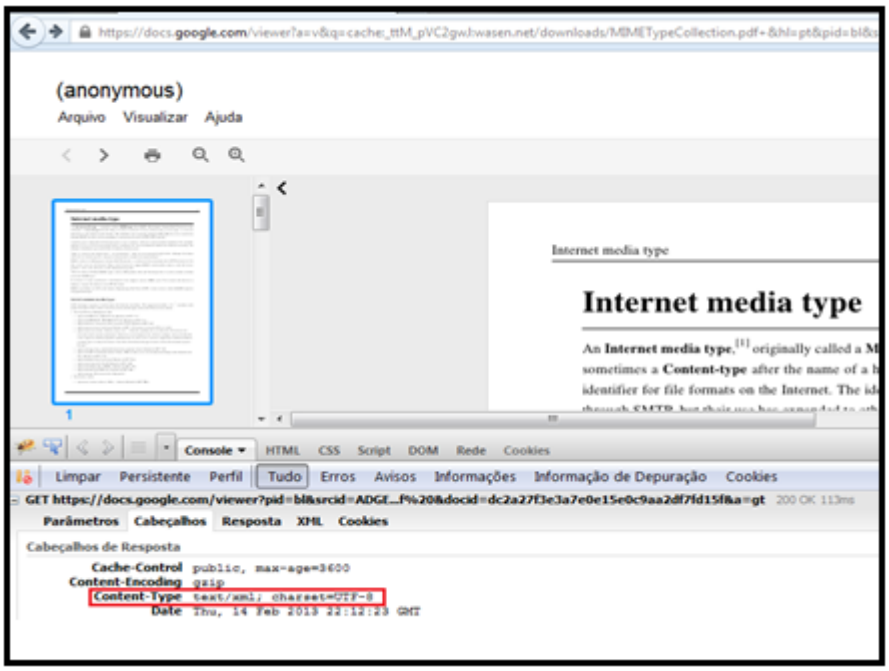


Figura 7- Content-Type de cabeçalho de mensagem de resposta

Fonte: Adaptado de WANSEN (2013)

Particularmente, para exibição do documento no browser o *Google* utiliza textos em formato XML para estruturá-los na tela.

Figura 8 - Conteúdo do corpo da mensagem

Fonte: Adaptado de WANSEN (2013)

Note que no cabeçalho de resposta produzido pelo servidor ilustrado na Figura 7, contêm o atributo *Content-Type* da mensagem com valor *text/xml*,ou seja, o tipo de dado presente no corpo da entidade é do tipo texto, e se apresenta no formato XML como exemplificado na Figura 8.

Os *Media Type* padrões mais utilizados dentro de APIs são o *application/json* e *application/xml*, sendo que a primeira opção é a mais utilizada, pois retorna objetos no formato JSON que são mais faceis de manipular.

5.5 FORMATOS ENVOLVIDOS NA CONSTRUÇÃO DE MENSAGENS

Aqui serão explicados a origem e alguns conceitos sobre os dois formatos de mensagens que são mais utilizados para comunicação via Internet, o XML e o JSON.

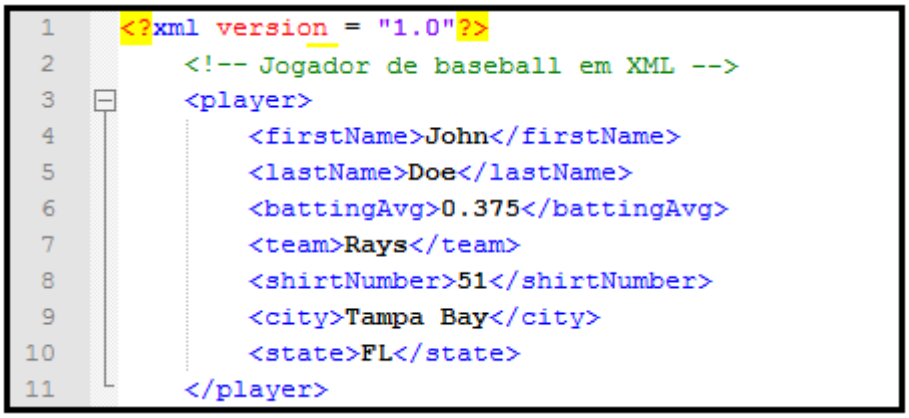
5.5.1 XML

XML é a sigla para *eXtensible Markup Language* e foi desenvolvida em 1996 pela *W3C XML Working Group*. Esta linguagem se tornou o padrão para formatação de dados utilizados para comunicação e armazenamento entre aplicações via Internet.

A linguagem XML é bastante similar a linguagem HTML, é uma linguagem de marcação de texto que utiliza o conceito de descrição de dados a partir da manipulação de *tags*. Para Hardware.com (2013), dentro da linguagem HTML, *tags* “são comandos que especificam como as diferentes partes do texto devem ser formatadas para exibição”.

Porém em XML, as *tags* são apenas utilizadas para representar algum tipo de dado, não tendo nenhuma função específica de formatação de texto para exibição. Para Deitel (2008, p.516), esta abordagem permite que os autores de um documento XML possam descrever uma série de tipos de informações diferentes, sendo que um programador pode representar em formato XML uma fórmula matemática para ser utilizada em um relatório financeiro. “A representação em XML permite que os dados descritos por ela sejam compreendidos por máquinas e seres humanos facilmente.” DEITEL (2008, p.516).

Na Figura 9 temos um exemplo de informação sobre um jogador de *baseball* representado em formato XML.



```
1   <?xml version = "1.0"?>
2       <!-- Jogador de baseball em XML -->
3       <player>
4           <firstName>John</firstName>
5           <lastName>Doe</lastName>
6           <battingAvg>0.375</battingAvg>
7           <team>Rays</team>
8           <shirtNumber>51</shirtNumber>
9           <city>Tampa Bay</city>
10          <state>FL</state>
11      </player>
```

Figura 9 - Texto estruturado em XML

Fonte: Adaptado de DEITEL (2008, p.517)

A maneira como os dados são representados dentro de um documento XML utiliza o seguinte padrão:

A primeira linha do arquivo é chamada de declaração XML. Esta linha é opcional e contém as informações sobre a versão do XML utilizado para descrição do documento, onde na Figura 9 está declarada a versão 1.0. Pode ter também a definição da codificação dos caracteres.

O restante do documento contém uma série de elementos alinhados, que podem ser divididos em atributos e conteúdo. Um elemento é definido pelas *tags* de início e fim, que na Figura 9 cita-se o exemplo da *tag<firstName>* e *</firstName>* e o conteúdo do elemento é o dado que está escrito entre as *tags*, no caso *John*.

Para manipulação de dados dentro de um arquivo XML são usadas ferramentas chamadas *XML Parsers*, que realizam a aquisição das informações presentes no documento e transmitem para o formato utilizado na aplicação, uma ferramenta muito utilizada é o *DOMParser*[4] e o *SAXParser*[5].

Mkyong (2008) acredita que o *DOMParser* basicamente leva em consideração a estrutura hierárquica de elementos montados no XML, carregando o documento todo em memória e montando seus elementos em um modelo em árvore, onde cada

[4] Mais informações sobre *DOMParsers*: http://www.mkyong.com/java/how-to-read-xml-file-in-java-dom-parser/.
[5] Mais informações sobre *SAXParsers*: http://www.mkyong.com/java/how-to-read-xml-file-in-java-sax-parser/.

elemento se torna um nó desta árvore, isto faz com que as informações presentes no documento XML sejam encontradas mais facilmente.

A Figura 10 apresenta um pequeno exemplo de como algumas informações do XML apresentado na Figura 9 podem ser representadas hierarquicamente em modelo DOM para serem utilizadas pelo *DOMParser*.

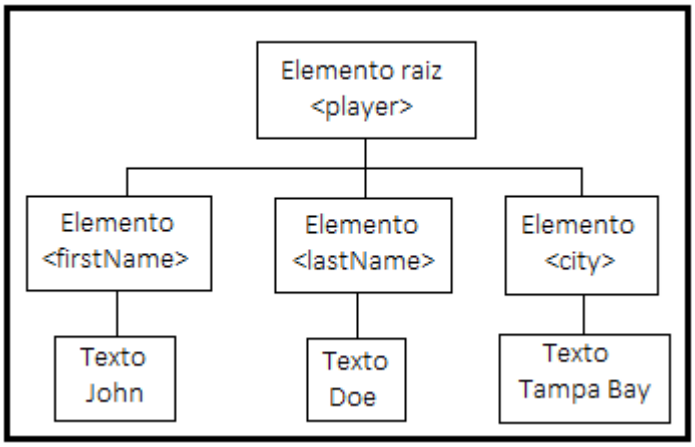


Figura 10 - Estrtura hierarquica de dados em XML

Fonte: Adaptado de XMLDOM (2013)

O *SAXParser* trabalha de maneira diferenciada, segundo Mkyong (2008), o *SAXParser* também carrega o documento XML em memória, porém não cria objetos para armazená-lo, ao invés disso ele utiliza uma função que indica a estrutura presente no XML.

Se analisarmos a fundo como cada um dos conversores é implementado, podemos perceber que a conversão de dados em XML pode se tornar complicada caso o arquivo tenha muitos dados para serem extraídos. Esse é um fator que desconsidera o XML na escolha de uma linguagem para comunicação entre aplicações, pois o processamento se torna mais demorado por causa do número de funções que ela deve executar. Além de exigir o consumo de mais memória, por carregar o documento XML a ser convertido e pela lógica de implementação ser muito ampla.

Apesar disso, o padrão XML é amplamente utilizado na construção de *Web Services* conforme apresentado no capítulo 5.6. Pois os documentos utilizados, como o WSDL para especificação do comportamento do serviço, o UDDI para

definição do local onde o serviço será acessado e o protocolo de comunicação e transporte de dados SOAP, usam documentos com marcação de texto para organização de seus dados.

5.5.2 JSON

Para a comunicação remota de sistemas, JSON vem se tornando uma estrutura de dados cada dia mais popular entre programadores no mundo todo como opção ao XML, por ser fácil de interpretar e ser compatível com uma série de plataformas e linguagens de programação.

A estrutura JSON entrou para o *ECMA Standard* em 1999, mas se popularizou graças a Douglas Crockford em 2002.

Para Diaz (2006), as razões que tornaram a estrutura JSON tão popular foram:

- JSON é facil de utilizar e entender;
- Possui uma sintaxe simples;
- JSON é rápido;
- Proporciona organização linear na forma que os dados são apresentados;
- Não causa conflito com outras liguagens que são utilizadas na implementação de um código.

JSON pode ser definido como uma interface para troca de dados baseado em texto, que permite que os dados transferidos sejam organizados como objetos. Um objeto pode ter uma série de valores e atributos que o caracterizam, cada objeto representado em JSON apresenta o seguinte formato:

```
{nomeDaPropriedade1:valor1,nomeDaPropriedade2:valor2,...}
```

Exemplo:

```
{ "player" :
        {
    "firstName": "John",
        "lastName": "Doe",
        "battingAvg": 0.375,
        "team": "Rays",
        "shirtNumber": 51,
        "city": "Tampa Bay",
    "state": "FL"
      }
}
```

Fonte: Próprio autor

No exemplo acima é mostrada a mesma representação do jogador de *baseball* apresentada na Figura 9, porém, em formato JSON. Aqui temos a apresentação de um objeto chamado "*player*", onde este objeto contém as propriedades, "*firstName*", "*lastName*", "*battingAvg*", "*team*", "*shirtNumber*", "*city*" e "*state*" que são propriedades que caracterizam o objeto "*player*". Podemos observar que a representação em JSON acaba sendo mais intuitiva do que a representação em XML.

JSON suporta apenas valores numéricos, tipo *String*, tipos booleanos como *true* ou *false* e valores nulos para representação das propriedades de um objeto.

Deitel (2008, p.444) acredita que o formato JSON providencia uma rápida maneira de manipular objetos, tanto para *Javascript* quanto para qualquer outra linguagem de programação que suportar esse formato, pois permite a extração de dados transmitidos pela Internet.

Um dos processos de extração de dados de um objeto JSON usando a linguagem Java é realizado através da utilização de um mecânismo chamado *Object Mapper*[6], que em resumo faz a transformação das informações presentes em objetos JSON para objetos em Java.

Quando comparados, nota-se uma discrepância muito grande entre a conversão de um arquivo XML e a conversão de um objeto JSON em um objeto Java. Isto acontece por que é necessária a utilização de bibliotecas para a transformação de dados JSON para Java. Estas bibliotecas fornecem as funções necessárias para a transformação das informações sem a necessidade de implementação de códigos adicionais, o que deixa o processo simplificado e ágil.

Para que a conversão seja bem sucedida, os dois objetos, tanto o objeto Java quanto o objeto JSON, devem ter seus atributos nomeados da mesma maneira, além de implementar a classe *Serializable*, que indica que o objeto pode receber informações no formato JSON, pois o que ocorre é exatamente um mapeamento entre os atributos dos dois objetos para que haja a transferência de dados entre eles, como exemplificado na Figura 11.

[6] Processo de conversão de objetos JSON para Java: http://www.mkyong.com/java/how-to-convert-java-object-to-from-json-jackson/

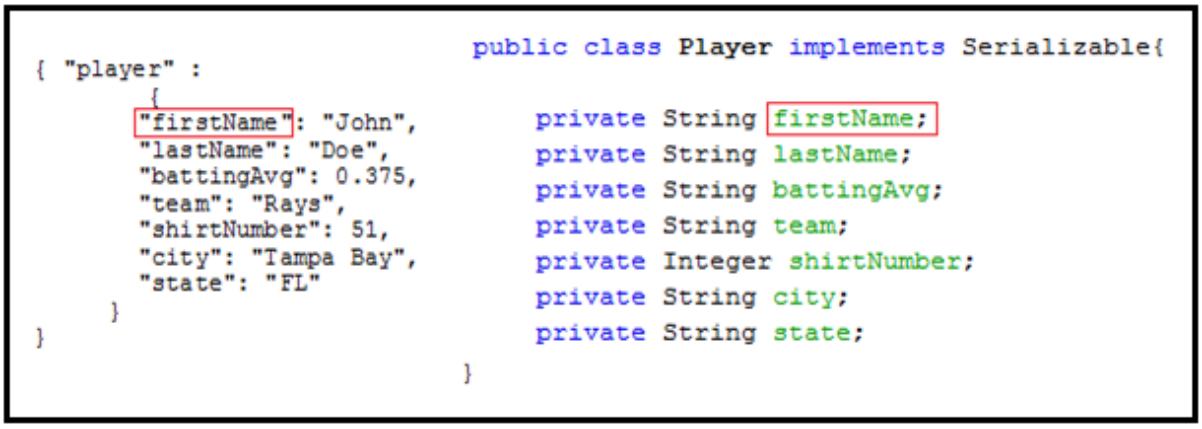


Figura 11 - Mapeamento de dados entre objetos JSON e JAVA

Fonte: Próprio autor

Portanto, analisa-se que JSON é uma opção muito mais viável para transferência de dados, pois requer muito menos recursos para transporte e transformação de informações, sendo uma das melhores opções para troca de dados pela rede.

5.6 WEB SERVICES

Quando citado sobre os conceitos envolvidos no desenvolvimento de APIs para o ambiente Web, é importante expor o conceito de *Web Services*.

De acordo com Webopedia (2013), o termo *Web Services* descreve uma maneira padronizada de integrar aplicações baseadas na Web. Os primeiros *Web Services* desenvolvidos eram construídos usando mecanismos como XML, SOAP, WSDL e UDDI sobre um protocolo de Internet. Sendo, XML usado para definição dos dados transportados, SOAP usado como protocolo de transferência de dados, WSDL tem a função de descrever o comportamento dos serviços que serão executados e UDDI faz uma listagem dos serviços que estão disponíveis.

Assim, *Web Services* possibilitam que diferentes aplicações de diferentes fontes possam se comunicar através dos serviços disponibilizados. Facilitando a troca de informações entre sistemas isolados baseados em diversas linguagens de programação, para que eles possam funcionar de forma conjunta através da comunicação destes sistemas.

Na Figura 12 podemos observar um exemplo desta abordagem, onde temos que cada componente (Hotel, Passagem e Locadora) representa um serviço disponível que pode ser baseado em uma plataforma diferente. Através de *Web*

Services cada um destes componentes consegue enviar informações para o *Web Server* independente da linguagem de implementação de cada um deles, facilitando a integração entre os sistemas envolvidos.

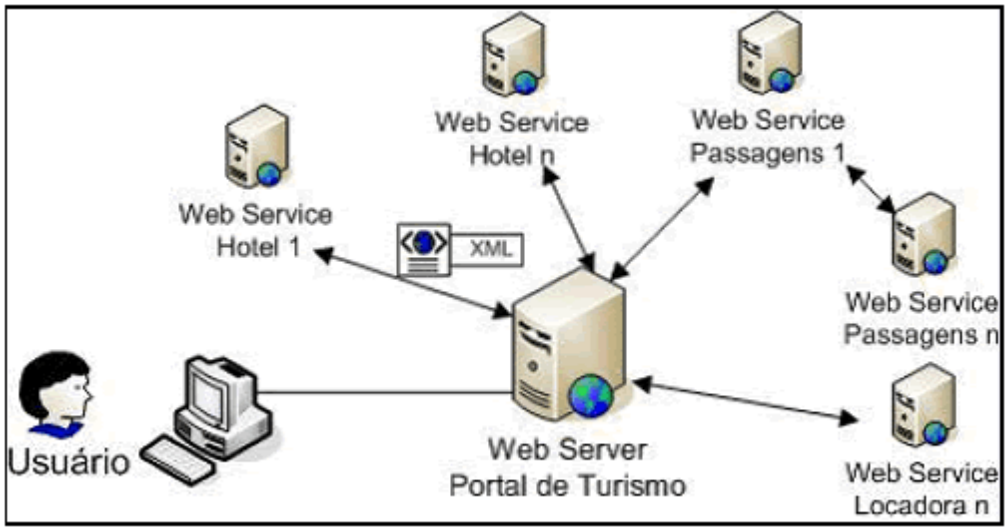


Figura 12 – Webservices possibilitando acesso a sistemas de diferentes plataformas

Fonte: IMASTERS (2013)

A arquitetura de uma *Web Service* é baseada em três componentes: provedor dos serviços, consumidor dos serviços e o registro dos serviços defende UFSC (2012). Onde em resumo, o provedor dos serviços disponibiliza um conjunto de serviços e operações através da identificação por URIs e distribuição de artefatos XML para os clientes, o consumidor dos serviços é definido como o cliente que irá utilizar um serviço disponível e o registro dos serviços é o estabelecimento do local onde os serviços estão concentrados.

APIs REST são consideradas *Web Services* exatamente por se basearem nos três componentes citados acima. Também possuem a capacidade de expor serviços através de um protocolo de comunicação bem definido, utilizam a mesma arquitetura de componentes para disponibilização dos serviços e não necessitam de uma interface visual para realizarem comunicação entre sistemas. Entretanto, um serviço REST presente em APIs (também chamado de *RESTful service*) utiliza outros mecânimos para transporte e integração de dados, porquê não se baseia nas ferramentas WSDL, UDDI e SOAP.

6 APPLICATION PROGRAMING INTERFACE

Neste capítulo serão expostos alguns aspectos relacionados às APIs, com o objetivo de aumentar a compreensão de como esta nova abordagem surgiu no mercado de TI e quais suas características.

6.1 HISTÓRIA DAS APIS MODERNAS

As APIs surgiram em meados de 1980, quando este conceito ainda era usado para integração de hardware e software. Porém, o conceito de APIs para o ambiente Web tem uma jornada mais curta, surgiram há apenas 10 anos atrás.

Segundo a Historyofapis (2013), a primeira API Web foi apresentada pela *Salesforce* em 7 de Fevereiro de 2000, como uma proposta a automatização e integração de serviços comerciais, introduzindo também o conceito de SaaS (Software como Serviço).

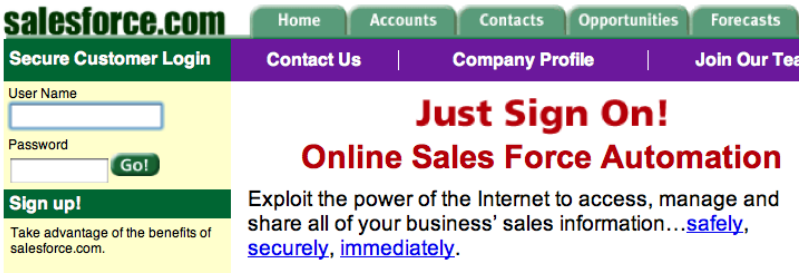


Figura 13 - API Web da *Salesforce*

Fonte: HISTORYOFAPIS (2013)

Mais tarde, em Novembro de 2000 o site de vendas online *eBay* lança o *eBay Application Programming Interface* junto com o *eBay Developer Program* que funcionava como um portal onde o usuário ou desenvolvedor poderia ver quais eram os serviços disponíveis pela API do *eBay*. Estes dois sistemas só podiam ser acessados por um conjunto específico de parceiros de negócio e desenvolvedores credenciados.

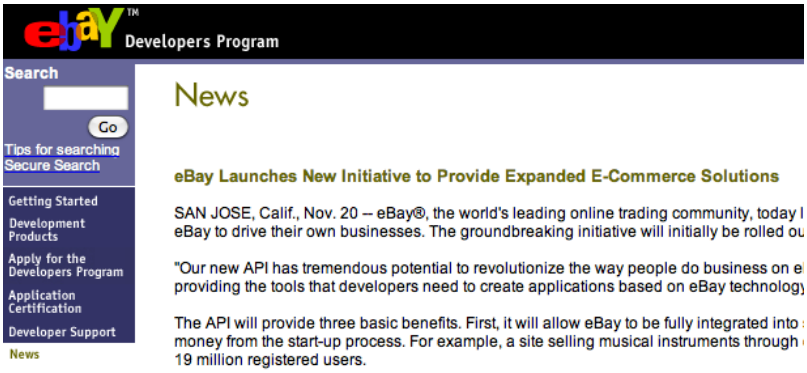


Figura 14 – eBay Developers Program

Fonte: HISTORYOFAPIS (2013)

Na época do lançamento a *eBay* fez a seguinte declaração:

> "Nossa nova API tem um tremendo potencial para revolucionar a forma como as pessoas fazem negócios no *eBay*, aumentando a quantidade de negócios realizados no site, pois irá fornecer as ferramentas que os desenvolvedores precisam para criar aplicações baseadas na tecnologia *eBay*, acreditamos que o *eBay* acabará por ser firmemente reconhecida entre os sites existentes como o futuro dos empreendimentos *e-commerce*." EBAYAPIHISTORY (2013) – Adaptado pelo autor

A API lançada pelo *eBay* foi responsável por um número considerável de aplicativos produzidos para comércio eletrônico, pois focou na padronização de como os sistemas desenvolvidos eram integrados, o que facilitava o trabalho dos desenvolvedores a partir do ecossistema de serviços disponíveis pelo *eBay*.

Em 2004 o conceito de APIs foi trazido para o contexto das redes sociais, a primeira empresa a entrar no mundo das APIs foi o site de compartilhamento de fotos Flickr. Ela lançou uma API que utilizava os conceitos da arquitetura REST para compartilhamento de dados, esta abordagem ajudou o Flickr a ser a plataforma de desenvolvimento escolhida pela maioria dos blogeiros e fãs de redes sociais por permitir que fotos fossem facilmente transportadas do Flickr para outros sites.

No periodo do ano de 2006, o Facebook e o Twitter lançaram suas APIs como plataforma de desenvolvimento, permitido que sejam acessados *posts*, comentários, fotos, entre outros recursos disponíveis por estas redes. A plataforma de desenvolvimento do Facebook baseava-se na arquitetura REST e trabalhava com informações do tipo XML, já a estrutura da API apresentada pelo Twitter tinha o

diferencial de também trabalhar com JSON e apresentou o conceito de autenticação para acesso aos serviços chamado *OAuth*.

Em Junho de 2006 a Google resolveu entrar no mercado com o lançamento do *Google Maps API*. O lançamento da API ocorreu a apenas 6 meses após o lançamento da aplicação Google Maps, em resposta direta ao número de APIs que surgiam no mercado dia após dia.

O sucesso da API lançada pela Google foi tão grande que logo surgiram aplicações construídas a partir dela, como o *ChicagoCrime.org*. Este site é um sistema utilizado para mapear as áreas de ocorrência de crimes dentro da cidade de Chicago nos Estados Unidos, no qual utiliza os recursos disponiveis pelo Google Maps para saber onde os crimes ocorreram.

A Figura 15 mostra a exibição de roubos a propriedades ocorridos em uma área de Chicago entre o periodo de 27 de Julho de 2006 a 4 de Agosto de 2006.

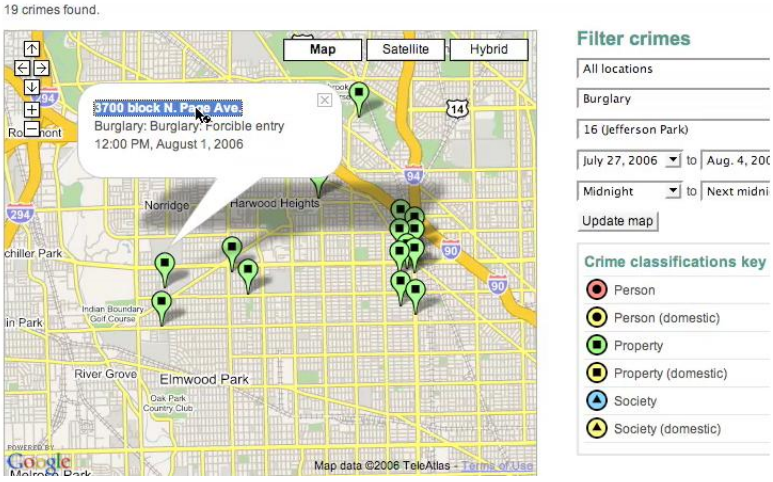


Figura 15 – Pesquisa por ocorrência de crimes utilizando *ChicagoCrime.org*

Fonte: Adaptado pelo Autor (2013)

Em Agosto de 2006 a Amazon lançou uma API para ser utilizada para *Cloud Computing* chamada *Amazon EC2*, que era de acordo com a AWS (2013) “um serviço da Web que fornece uma capacidade de computação redimensionável na nuvem. É projetado para tornar a escalabilidade computacional de nível de web mais fácil para desenvolvedores”, permitindo armazenamento de dados usando os conceitos de *Cloud Computing* da mesma maneira que o *Amazon Simple Storage System* (*Amazon S3*).

E finalmente em Março de 2009, as APIs começaram a ter um direcionamento para as tecnologias móveis com o lançamento do Foursquare. Este aplicativo é uma

rede social que permite que o utilizador possa indicar em que local ele se encontra e procurar onde seus contatos estão. Sua API proporcionou o suporte para vários tipos de sistemas operacionais móveis como o iOS, Android, Windows Mobile, Blackberry e Symbian.

6.2 DEFINIÇÃO

Segundo Jacobson et al (2011), a definição de API segue duas ramificações, uma técnica e uma de negócios.

A definição técnica para API é: um sistema que proporciona uma maneira para que duas aplicações computacionais possam se comunicar entre si. Usam um meio de comunicação comum (normalmente a Internet) para troca de informações, proporcionando uma linguagem na qual ambos os sistemas conectados possam entender. Ou seja, uma API tem a função de promover uma interface de comunicação entre diferentes sistemas através do fornecimento de serviços estabelecidos por padrões ou regras de programação definidos pelo fornecedor da API.

Já para a área de negócios, uma API é uma ponte que liga desenvolvedores e usuários aos serviços providos por uma empresa. Estes serviços são disponibilizados via Internet por um provedor, possibilitando o desenvolvimento de aplicativos a partir destes serviços. Um exemplo deste tipo de abordagem foi adotado por grandes empresas do mercado como Facebook e Twitter.

Portanto, para realizar essa ligação, uma API funciona como um contrato que fornece um pacto entre desenvolvedor e provedor da API, no qual este contrato dita as regras de como as aplicações devem ser integradas e como os serviços irão se comportar. Jacobson et al (2011, p.5) cita que existem algumas responsabilidades que tanto o provedor quanto o consumidor dos serviços de uma API devem seguir para que o contrato funcione com integridade:

- O provedor da API deve descrever exatamente como sua API funciona através de uma documentação dos serviços disponíveis;
- Deve descrever quando as funcionalidades estarão disponíveis e quando elas sofrerão alterações;

- Deve estabelecer restrições técnicas dentro da API, como por exemplo, definição de taxas de utilização, que restrigem a quantidade de acessos que determinada aplicação pode fazer a uma API;
- Devem também adicionar restrições de negócios, como limitações de marca e tipos de uso;
- O desenvolvedor então, se compromete a utilizar a API na maneira como ela foi descrita seguindo as regras impostas pelo provedor da API.

Seguindo os conceitos apresentados, podemos dividir as APIs em duas categorias: aquelas que são abertas para qualquer tipo de público (APIs Públicas) e APIs que são utilizadas apenas nos processos internos de uma empresa (APIs Privadas).

6.3 QUEM UTILIZA

Para Jacobson et al (2011, p.6), existem 3 tipos de entidades que utilizam uma API de forma distinta:

- Provedor da API (*API Provider*): É a empresa ou organização que disponibiliza um conjunto de serviços dentro de uma API, é também quem estabelece os contratos de utilização;
- Desenvolvedores (*Developers*): Pode ser classificado nesta categoria qualquer tipo de pessoa que tenha interesse direto em consumir uma API para alguma finalidade específica. Normalmente o público alvo de uma API são pessoas que tem a capacidade de criar aplicações ou soluções tecnológicas a partir dos serviços fornecidos por uma API;
- Usuários Finais (*Endusers*): Esta categoria contém o público que utiliza uma aplicação final criada pelos Desenvolvedores. Tem a importância de ditar como as APIs e as aplicações são projetadas, tanto para o provedor da API quanto para desenvolvedores. Esta categoria é importante, pois é a partir da necessidade dos usuários finais que uma empresa sabe quais os serviços devem ser expostos, e que tipo de aplicação deve ser lançado para atender essas necessidades.

6.4 IMPORTÂNCIA DAS APIS

No design e desenvolvimento de APIs, não é importante conhecer apenas os conceitos teóricos que envolvem APIs, mas principalmente, conhecer quando uma

API é necessária. A partir da leitura da obra de Jacobson et al (2011, p.15 – 19), existem alguns motivos que encadeiam o desenvolvimento de uma API qualquer.

Dentro de grandes corporações, uma das principais motivações para o design de uma API é o mercado de dispositivos móveis. O objetivo desta abordagem é a construção de uma única aplicação que deve ser compatível com vários dispositivos, assim, funcionando nos principais sistemas operacionais móveis disponíveis no mercado, como o Android, iOS e Windows Phone.

Uma API será desenvolvida neste caso para evitar que a mesma funcionalidade tenha que ser projetada de diversas maneiras, para que venha a ser compatível com cada um dos sistemas operacionais citados. Tendo em vista que cada sistema tem uma arquitetura diferenciada, uma API pode ajudar as companhias a criarem aplicações que suportem múltiplos sistemas sem a necessidade de grandes alterações arquiteturais.

Outro motivo importante para o design de uma API é a necessidade de um parceiro de negócio da empresa que pode, por ventura, precisar de uma API que facilite integrações entre aplicações ou promova a melhora da arquitetura de um sistema.

Ao citar a integração de sistemas, podemos utilizar o exemplo da empresa IBM que possue um conjunto de APIs como o *Websphere Iron Cloud* que auxilia na integração de entre sistemas conectados na nuvem. Contudo, quando citamos a melhora na arquitetura de sistemas, temos o exemplo da empresa NPR. A API projetada foi usada para realizar a migração de dados entre bancos da *Oracle*, que apresentava falhas, para um *cluster* do tipo MySQL melhorando o acesso as informações presentes em seu ambiente.

Uma API providencia dados e serviços de maneira flexível, portanto, algumas empresas como a MTA (*Metropolitan Transit Authority*), que gerencia todo o transporte público de Nova Iorque, viu nas APIs uma oportunidade de expor todas as informações relacionadas a rotas e itinerários de suas linhas de metrô em arquivos do *Google Transit*[7], fazendo com que diversos desenvolvedores da Google criassem aplicações a partir destas informações.Um exemplo de aplicação é o *Bus New York City*, nele é possivel, pelo celular, pesquisar as rotas de determinada linha de

[7] Mais Informações: http://www.google.com.br/intl/pt-BR/landing/transit/#dmy

ônibus, quais são suas paradas e seu itinerário completo, trazendo maior comodidade aos usuários de transporte público.

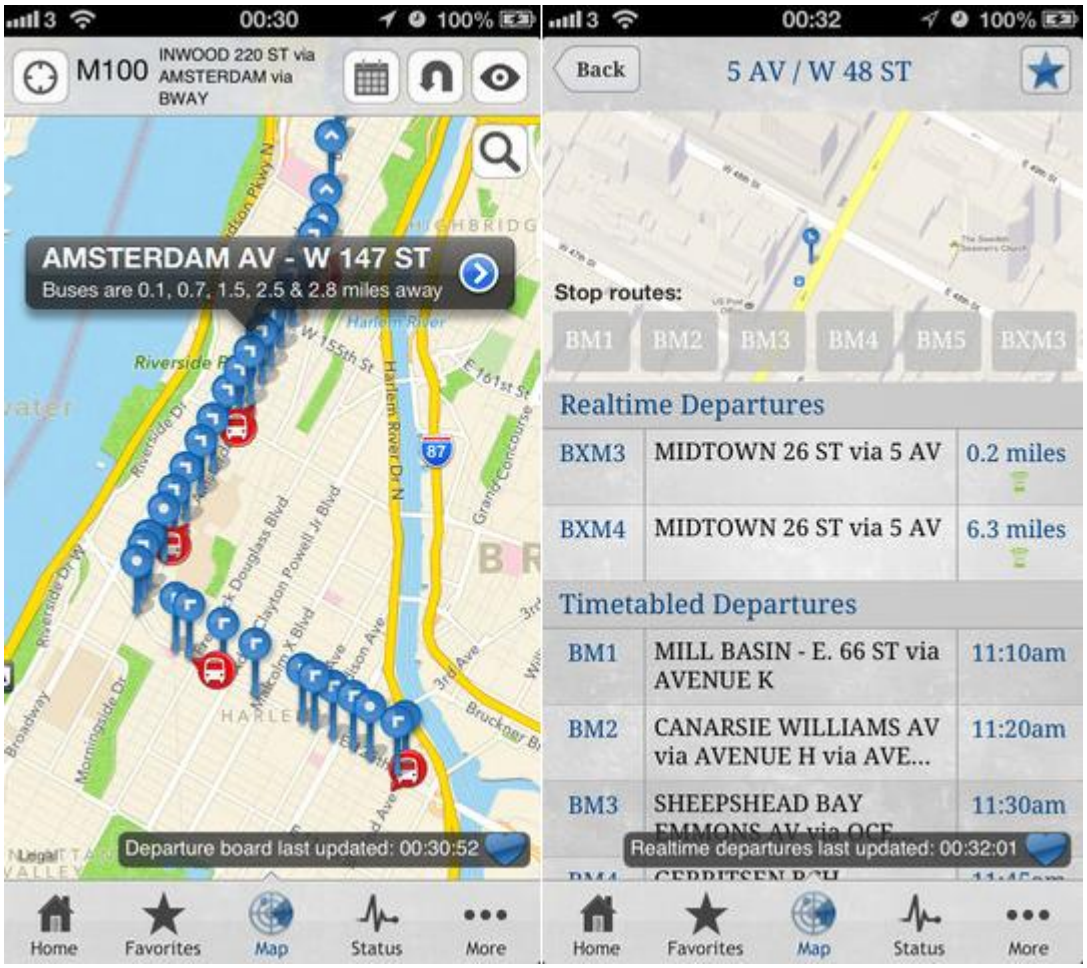


Figura 16 – Exemplo de funcionamento do aplicativo *Bus New York City* para iPhone
Fonte: ITUNES (2013)

6.5 TIPOS DE APIS

Nós podemos definir dois tipos distintos de APIs, públicas e privadas, onde neste contexto consideramos as APIs privadas as mais importantes, pois a maioria das grandes empresas como Twitter, Facebook e Netflix, que expõem grande parte dos seus serviços para desenvolvedores em geral, tendem a utilizar estas mesmas APIs para direcionamento da construção de seus produtos e definição de seus processos internos.

Na seção a seguir será explicado mais detalhadamente os conceitos sobre APIs públicas e privadas, além de definir sua importância.

6.5.1 APIS PÚBLICAS

Segundo Iromin (2010) as APIs Públicas se tornaram uma peça importante para qualquer aplicação da Web. De acordo com o site *ProgrammableWeb.com* (2012) existem mais de 2033 API disponíveis para desenvolvimento de aplicações.

De acordo com Jacobson et al (2011 p.7), uma API pública pode ser definida em resumo, como uma API que é exposta para um conjunto de pessoas tendo pouco ou nenhuma ligação contratual entre as partes.

Este tipo de abordagem, para Iromin (2010), permite que desenvolvedores possam integrar uma série de funcionalidades que são expostas pela API em outras aplicações, um exemplo claro é a utilização de mapas para a indicação de locais em outros sites, os desenvolvedores utilizam os serviços do Google Maps para exibir estes mapas, assim não é necessário recriar uma funcionalidade já existente.

Jacobson et al (2011, p.29) também acredita que APIs públicas são usadas como estratégia de extensão de marca, ou seja, muitas empresas disponibilizam serviços para que outras aplicações sejam desenvolvidas utilizando seus recursos, possibilitando assim a criação de novos produtos. O Twitter utiliza esta abordagem permitindo que diversos desenvolvedores utilizem sua plataforma para construção de aplicações como o *TweetDeck*, que é uma aplicação *desktop* que permite acesso, envio e visualização de mensagens e perfis de outros usuários do Twitter ou Facebook através de qualquer tipo de dispositivo conectado a Internet.

Figura 17 – Tela de apresentação do TweetDeck

Fonte: http://tweetdeck.com/assets/homepage/twitter/img_screenshot_newlogo.png (2013)

A extensão de marca permite que a empresa atinja outros públicos e nichos de mercado com a construção de seus produtos, além de proporcionar maior inovação com todo o conjunto de produtos fabricados. Quando um conjunto de novos sistemas são criados baseando-se em uma API, dizemos que foi criado um ecossistema de aplicações.

A Figura 18 ilustra um exemplo de ecossistema de aplicações que foram desenvolvidas a partir da API do Twitter, no qual é mostrado, o *TweetDeck*, o *TwitPic* e o *Twitalyzer* como alguns exemplos mais conhecidos.

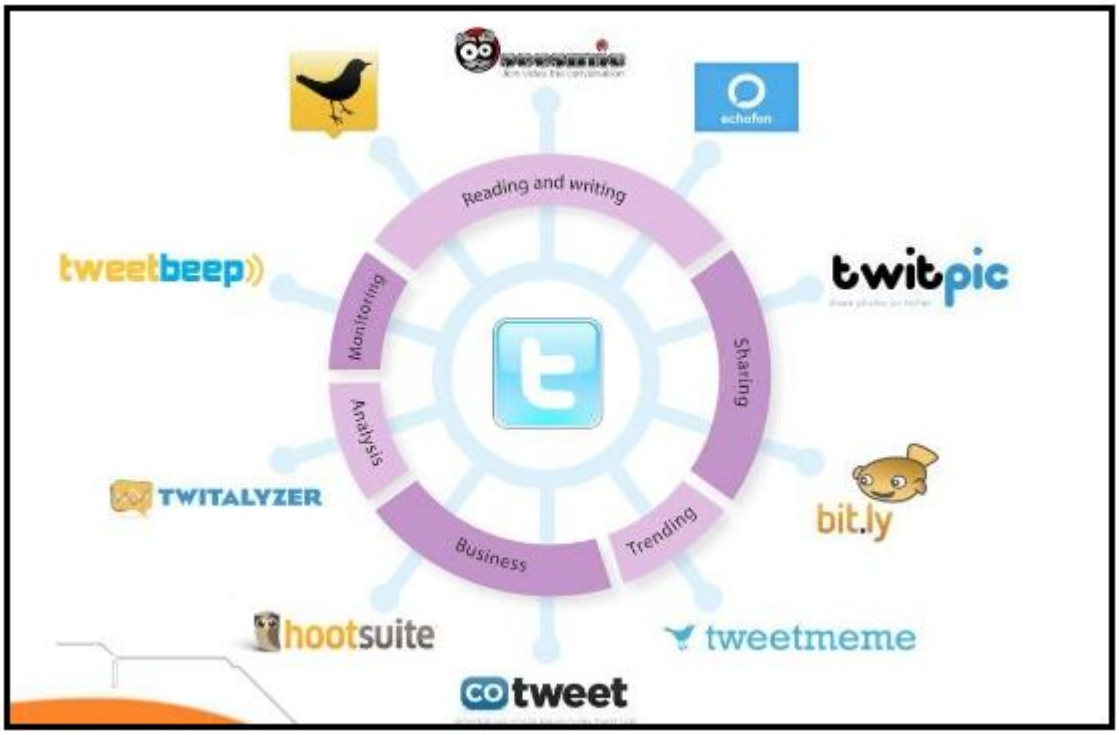


Figura 18 – Exemplo de ecosistemas de aplicações utilizando API do Twitter
Fonte: http://tweetdeck.com/assets/homepage/twitter/img_screenshot_newlogo.png (2013)

Por fim, com o crescente número de APIs públicas no mercado, existem a disponibilização de serviços para as mais diversas finalidades, sendo que podemos dividir as APIs mais conhecidas em algumas categorias a seguir:

- APIs de mídias sociais: São as APIs que permitem acesso as redes sociais como Twitter, Facebook e Google+ que realizam operações como edição de *posts*, mensagens e visualização de perfis.

- Serviços básicos: São APIs que promovem serviços como busca dinâmica de informações, como as realizadas por sites como Bing ou Google.

- Serviços de informação: Estes tipos de serviços são aqueles que têm o objetivo principal deixar o usuário ciente sobre algum tipo de informação especíica em tempo real, por exemplo, fornecer dados sobre a previsão do tempo, como é feito pelos serviços providos pela AccuWeather.

- APIs de mídias locais: Qualquer API que promove a conexão com serviços multimídia, como as APIs do Youtube que fornecem vídeos, ou os serviços do New York Times e ESPN para a uso das notícias por ela ministrados.

- APIs de sistemas *e-commerce*: Podem seguir os exemplos dos serviços disponíveis por sites como o *eBay, Amazon* ou o site nacional BuscaPé. Estes sistemas podem proporcionar o catálogo completo de seus produtos e preços ou permitir realizar operação de compras através do uso de suas APIs.

6.5.2 APIS PRIVADAS

Em contrapartida às APIs públicas, uma API privada tem a característica de ser um conjunto de funcionalidades e serviços que são utilizadas apenas dentro das corporações como auxílio na criação de seus produtos ou execução de operações.

Não existe uma definição concreta do que seja uma API privada, no entanto, temos que esse tipo de API possui algumas particularidades que o diferenciam de APIs públicas.

De acordo com Mitra (2012), este tipo de API providencia uma interface independente de liguagem que é disponibilizado através de protocolos Web e tem seu acesso limitado apenas a um conjunto específico de desenvolvedores ou organizações o que faz com que não seja comercializado para público geral, restringindo acesso as funcionalidades da API.

Mitra (2012) também explica que uma API privada possue mecanismos de segurança mais elaborados, como SSL, HTTP *Basic Authetication e OAuth2* para evitar que usuários não autorizados acessem os recursos da API.

Para Mitra (2012) e Jacobson et al (2011, p.27) a motivação principal para manter uma API privada, está no fato de que uma empresa começa a perceber que o uso de uma API pode acarretar vantagens no processo de desenvolvimento de aplicações tanto para uso interno quanto externo, porque uma API bem

desenvolvida pode reduzir custos de projeto e facilitar a manutenção e a integração de sistemas, além de acelerar os processos de produção.

Um exemplo de como uma API privada pode ajudar no processo de inovação e integração de empresas esta na obra de Jacobson et al (2011, p.33), cita o exemplo da empresa de filmes online Netflix, que usava uma API internamente, com o intuito de construir uma plataforma que visava maior suporte ao acesso do conteúdo *streaming* através de dispositivos dos mais variados fabricantes.

7 REST

Neste capitulo serão apresentados todos os conceitos e fundamentos relacionados à arquitetura REST para sistemas distribuídos.

7.1 VISÃO GERAL

REST é a sigla para *Representational State of Transfer*, e pode ser definido como um estilo arquitetural para desenvolvimento de serviços dentro de sistemas distribuídos no ambiente Web.

Essa filosofia foi introduzida por Roy T. Fielding em sua dissertação de doutorado, realizada no ano 2000. A melhor definição de REST, segundo Fielding (2000, p.76), é que este tipo de arquitetura não é um padrão e sim um estilo híbrido de outras arquiteturas de rede que são combinadas com algumas restrições adicionais definindo uma interface de conexão única.

Para uma maior definição sobre o que consiste a arquitetura REST, alguns autores como Firmo (2012) e Fredrich (2012) explicitam que a teoria sobre REST gira em torno de alguns princípios básicos da arquitetura Web, iremos citar os principais.

- **Interface Uniforme**

  Este princípio define como funciona uma interface de comunicação entre clientes e servidores dentro da arquitetura.

  A interface escolhida para o padrão REST deve ser o protocolo HTTP, ele garante que as informações que trafegam pela rede se mantenham íntegras, pois nesse protocolo, segundo Allamaraju (2010, p.3), é usado uma interface que consiste dos métodos *OPTIONS, GET, HEAD, POST, PUT, DELETE* e *TRACE*, onde cada método da interface opera com

apenas um recurso, não mudando seu comportamento, nem sua sintaxe independente da aplicação ou do recurso que esta utilizando o método.

- **Múltiplas Representações**

  Para Allamaraju (2010, p.262) “o documento que um servidor retorna a um cliente é a representação de um recurso ou serviço. A representação é o encapsulamento da informação em uma codificação especifica, seja XML, JSON ou HTML”.

  Para Friedrich (2012), o conceito de representações é aplicado em como as respostas de um serviço podem ser manipuladas. Estas respostas possuem um conjunto de informações de uma operação que foi ou que será executada. Este conjunto inclui a informação que será enviada, o serviço que será executado e a codificação em que as informações estão.

  Portanto, uma informação deve ter uma ou mais representações, nas quais clientes e servidores usam o conceito de *Media Types* conforme apresentado no capítulo 5.4.

- **Recursos *Stateless***

  Este princípio está associado diretamente com os estados em que os recursos se encontram no servidor e a interação da aplicação em relação a estes recursos.

  Segundo análise das obras de Richardson (2007, p.218) e Fielding (2000, p.47), o servidor não guarda nenhum estado em que a aplicação se encontra, por isso um serviço RESTful é considerado *Stateless*. Em outras palavras, o servidor deve considerar cada requisição dos clientes de maneira isolada de forma que cada operação em um recurso seja interpretada e entregue de maneira íntegra, sendo que cada requisição feita a um servidor deve ter todas as informações para que o servidor compreenda qual é o tipo de requisição que o cliente está fazendo e que tipo de informações ele deve retornar.

- **Sistema em camadas**

  Este princípio lida com a maneira como o sistema deve ser construído dentro dos padrões REST.
  Dentro da arquitetura REST os serviços devem ser organizados de maneira hierárquica sendo dividida em componentes, onde cada componente é coexistente entre si e possui funções únicas e bem definidas facilitando a comunicação entre as partes envolvidas.

- **Armazenamento de informações**

  Segundo Watson (2011), tendo em vista que um serviço REST é naturalmente *Stateless*, serviços programados em REST devem ter algum tipo de mecanismo que identifique se as informações de resposta de um servidor devem ou não ser armazenadas para posteriores utilizações, eliminando assim a necessidade de carregamento de informações com mesmo conteúdo em determinadas partes do sistema.

7.2 ESTILOS DE ARQUITETURA REST

Como a arquitetura REST não possui um padrão rígido que dita as regras de como ela deve funcionar e principalmente como os desenvolvedores devem utilizá-la, no decorrer do tempo, surgiram algumas sub-categorias que são usadas por programadores em geral, em que cada categoria divide o comportamento de REST de maneira diferente, para Jacobson et al (2011, p.60) existem dois tipos principais, o *Pure REST* e o *Pragmatic REST*.

Basicamente, *Pure REST* pode ser definido como o estilo que utiliza todos os conceitos estabelecidos por Fielding para formar uma API, em adição do uso do conceito HATEOAS. O princípio HATEOAS diz que, ao invés de listarmos as funcionalidades de nossa API em uma documentação fixa deve-se, no entanto, fazer com que estas funcionalidades sejam descobertas enquanto a API é utilizada, onde o cliente faz o acesso inicial através de uma URI raiz, em que esta retorna um conjunto de outras URIs que dão acesso a recursos mais específicos e assim por diante, de forma que cada operação feita pelo servidor retorna um *link* que dá mais indicações sobre outras operações. Pode ser analizado o comportamento de navegação de uma API baseada nas teorias do *Pure REST* no exemplo de Jacobson et al (2011, p.61).

Em contrapartida ao *Pure REST*, temos o estilo *Pragmatic REST* que é o mais utilizado para definir a maioria das APIs modernas dentro da Web. O estilo *Pragmatic REST* propõem a construção de APIs usando apenas alguns conceitos da arquitetura e restringem-se apenas a usar o HTTP como protocolo de comunicação e JSON para formatação de dados, além de usar URIs fixas para definir os recursos da API, impossibilitando a aplicação do HATEOAS. Pois as URIs, quando necessário, são alteradas pelos próprios gerentes da API, retirando a capacidade do servidor de moldar seus serviços de acordo com as chamadas do cliente, porém proporcionando uma manutenabilidade maior.

Esta abordagem é usada porque facilita a compreensão do funcionamento da API, tanto para quem consome quanto para quem desenvolve, além de agilizar o processo de desenvolvimento e definição da arquitetura da API, onde todas as funções são devidamente documentadas.

8 PRINCÍPIOS DE DESIGN PARA APIS REST

Para implementação de qualquer tipo de API, devem ser seguidos alguns princípios e boas práticas de origem da arquitetura REST que auxiliam o utilizador a ter sucesso ao consumir a API produzida. Esta seção tem o objetivo de mostrar estas práticas, visando ajudar o desenvolvimento de uma API a partir de um conjunto de serviços.

Claramente, uma API pode ser construída para um conjunto diversificado de públicos, para atender os mais diversos objetivos, onde os dois públicos alvo principais para o desenvolvimento de uma API segundo Jacobson et al (2011, p.54), podem ser os Desenvolvedores ou Usuários Finais.

Contudo, o objetivo desta obra é apresentar os princípios mais importantes para a produção de APIs que serão utilizadas por Desenvolvedores, porque eles são responsáveis pela produção de aplicações a partir dos serviços de uma API qualquer.

As considerações iniciais sobre as decisões que devem ser tomadas ao construir uma API para uso em desenvolvimento são citadas por Jacobson et al (2011, p.54) e Mulloy (2012, p.3 - 4):

a. Para a estrutura inicial da API, deve ser escolhida a arquitetura REST, por ela permitir uma maior flexibilidade para o desenvolvimento e aplicação de boas práticas. O estilo de arquitetura deve ter foco no tipo

Pragmatic, ela proporciona um conjunto de funcionalidades que é facil de aprender, auxilia o consumo de serviços e dá suporte maior a expansão do que outras tecnologias, aumentando a produtividade e sucesso do desenvolvedor que consome a API;

b. Com o objetivo de melhorar a comunicação entre os elementos que consomem recursos dentro da API, deve ser usado o formato JSON para constituir os dados que irão trafegar entre as partes;

c. Toda API precisa de um recurso de segurança para restringir acesso indevido a dados, principalmente em se tratando de API privadas, neste caso, deve-se optar pela arquitetura *OAuth* como componente de segurança.

Após a definição da infraestrutura que será utilizada na implementação da API,é necessário seguir um novo rumo. Segundo Allamaraju (2010, p.29), em um serviço que segue a filosofia REST, o primeiro passo que deve ser feito é a definição do modelo que identifica e classifica todos os recursos que o cliente utilizará para interagir com o servidor.

8.1 DEFININDO RECURSOS

Segundo Richardson e Ruby (2007, p.81), um recurso em REST, pode ser definido como qualquer tipo de artefato interessante a aplicação que o cliente deseja manipular, e que pode ser armazenado em um computador, seja ela um pedaço de informação, um objeto físico, um documento ou uma linha de uma tabela.

Um recurso só pode ser acessado pela URI única que o identifica. Quando definimos uma URI para recursos, devemos usar o seguinte padrão relatado pela RFC 3986 que define a seguinte sintaxe:

URI = esquema “://” autoridade “/” caminho [“?” query] [“#”fragmento]

No fragmento acima percebe-se que, “esquema” define o protocolo de rede que será utilizado, “autoridade” define qual é o endereço do servidor que detém os recursos e serviços, “caminho” indica qual o nome do recurso que será utilizado e a sessão “*query*” e “fragmento” são opcionais e dão informações adicionais que podem ser usadas, como filtros ou restrições em determinadas operações. A Figura 19 mostra o exemplo de URI montada a partir desta sintaxe indicando cada uma das partes.

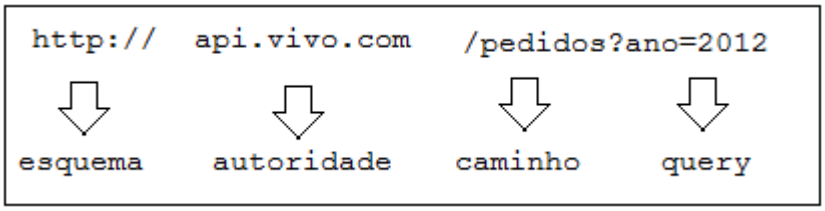


Figura 19 – URI separada em partes com indicação de seus setores na sintaxe

Fonte: Próprio autor

Porém, quando define-se as URIs de uma nova API, é importante seguir algumas regras.

A regra inicial faz parte da definição do setor “autoridade”, para Massé (2012, p.14), temos que o domínio da “autoridade” deve conter o nome do provedor da API, como apresentado no exemplo da Figura 19 (api.vivo.com). Dentro dela, temos o detentor dos serviços “vivo.com”, mais a indicação de que estes serviços estão disponiveis através de uma API, pela da inclusão do prefixo “api” ao nome do domínio.

Logo depois da definição do domínio principal, é exibido a definição dos recursos que irão constituir todos os serviços da API implementada.

Primeiramente, de acordo com Mulloy (2012, p.5), devem haver apenas duas URIs por recurso disponível, sendo uma para representar operações em coleções de objetos e outra para representar operações em uma entidade especifica, segundo exemplo abaixo:

- /pedidos: Trabalha com operações dentro de uma coleção de pedidos;
- /pedidos/123: Trabalha com operações em um pedido especifico identificado pelo *id* 123.

Cada URI,em conjunto com um dos métodos definidos pela interface uniforme de REST, retorna o registro que será manipulado na forma de representações: documentos formatados de maneira específica que detêm a informação de determinado recurso, ou executam alguma operação no servidor conforme Tabela 1.

Tabela 1 - Interação entre URIs dos recursos e métodos da interface uniforme.

Recurso	***GET***	***POST***	***PUT***	***DELETE***
/pedidos	Lista os pedidos	Criar novo pedido	Atualiza todos os pedidos*	Exclui todos os pedidos

/pedidos/564	Exibe detalhes do pedido 564	Cria um item no pedido 564	Atualiza o pedido 564**	Cancela o pedio 564
*** Normalmente não faz sentido, então retornaria o codigo 405**				
**** Se o recurso não possuísse o itens, retornaria o codigo 405**				

Fonte: FIRMO (2012, p.3)

Para Firmo (2012, p.1) o termo recurso é utilizado no lugar do termo serviço, portanto conclui-se que, os recursos construídos em uma API são na verdade a caracterização de seus serviços e sua capacidade em providenciar informações relevantes. Em outras palavras, é analisado a partir da amostra da Tabela 1.0 que, o recurso "/pedidos" na verdade é a caracterização do serviço que opera com as informações relacionadas a pedidos e engloba operações de consulta (*GET*), criação (*POST*), atualização (*PUT*) e exclusão (*DELETE*). O que nos leva a apresentar uma vantagem, uma única URI pode representar uma série de serviços diferenciados.

Percebe-se que, a sessão "caminho" de uma URI não possui em seu interior verbos que indicam operações, do tipo "salvar", "atualizar" ou "deletar", como por exemplo, "/pedidos/123/salvar". Isso impossibilitaria o uso da interface uniforme, e aumentaria o número de URIs disponíveis. Portanto, outra boa prática é nunca usar verbos que indicam operações dentro de uma URI para executar métodos HTTP no servidor.

Os nomes dos recursos de uma API são definidos a partir de objetos e entidades presentes no modelo de dados do próprio sistema, porque os dados retornados, muitas vezes vem da própria base de dados presente no servidor que providencia a API.

Uma regra bastante importante na definição de URIs de recursos, é considerar todas as entidades e objetos presentes, assim como, os relacionamentos entre eles, porque um recurso pode conter outros recursos em seu interior.

Tomemos como exemplo um serviço de aplicação de multas de trânsito, que possui uma série de entidade, entre elas: "Carro" e "Multa", em que essas possuem um relacionamento de 1 para muitos na base de dados, ou seja, 1 carro pode possuir várias multas.

Para ter acesso a todos os carros cadastrados, a URI “/carros” em conjunto com o método GET deve trazer todos os carros disponíveis em determinada representação. Entretanto, existe a possibilidade de um acesso mais específico, onde deseja-se saber quais foram as multas aplicadas e um determinado carro.

Existindo relacionamento entre a entidade “Multa” e a entidade “Carro”, deve-se construir uma URI que indentifique este relacionamento através do seguinte preenchimento em seu “caminho”: “/carros/345/multas”, em que a seção *“*345*”* representa o identificador do carro em que iremos fazer a consulta no banco de dados.

Desta maneira tem-se o retorno de todas as multas cadastradas para o carro “345”. Com as leituras das obras de, Massé (2012, p.12) e Mulloy (2012,p.9), concluir-se que, as URI devem conter representações hierárquicas, de forma que o ultimo recurso escrito no “caminho” é sempre mais específico que o anterior, e devem sempre ser separadas por um identificador entre cada recurso.

Porém, os relacionamentos entre os recursos podem se tornar um tanto complexos quando temos muitos objetos relacionados entre si. Por exemplo, a entidade “Multa” do exemplo acima, pode conter relação com a entidade “Motorista”, que por sua vez pode ter relação com a entidade “Carro”, que tem relacionamento com a entidade “Documento” e assim por diante.

Esta situação geraria uma URI com muitos recursos e tornaria a utilização da API pouco intuitiva, uma vez que o usuário teria que se lembrar de todos os relacionamentos presentes em uma entidade para recuperar uma informação específica.

Por esse motivo, relevamos a importância da utilização do setor *query,* presente na Figura 19 para inscrição de nossa URI, de maneira a conseguir maior especificidade na obtenção de informações em uma cadeia complexa de relacionamentos. Dentro dela, devem estar todos os atributos relevantes para a obtenção de informações mais específicas como demonstrado no exemplo: “/carros?cor=vermelho&ano=2002&marca=Honda”.

Acima, é verificada a construção de uma *query* que tem a função de filtrar um conjunto de carros a partir dos atributos “cor”, “ano” e “marca”. Esta abordagem pode ser feita tanto em recursos que tem a função de consulta (mapeados pelo método *GET*), quanto para recursos com a função de atualização (mapeados pelo método *PUT*).

De acordo com a leitura de Mulloy (2012, p.8) e Massé (2012, p.12), verifica-se alguns pontos importantes na definição de nomes de recursos em uma API:

- Devem-se usar sempre nomes que façam referência direta a objetos e entidades do nosso sistema;
- Importante a utilização de nomes curtos, com substantivos sempre no plural, como “/carros”, “/pedidos” ou ”/filmes” e de preferência padronizados na língua inglesa.
- Padronizar as URIs com letras mínusculas apenas;
- Não utilizar nomes com nível de abstração alto, ou seja, nomes que possam dar a ideia de representar muitos tipos de objetos, como “/itens”, “/ativos” ou “/coisas”;
- Sempre utilize a seguinte sintaxe para definição de relacionamentos entre recursos: “/nome-da-coleção/identificador-do-item/nome-do-subgrupo-de-objetos”, e mantenha a caracterização dos relacionamentos em no máximo três níveis. Relacionamentos com níveis acima do indicado devem ser representados através de *queries*.
- Barras verticais (“/”) no final da URI, logo após do nome do recurso, por exemplo, http://api.vivo.com/pedidos/123/, não são recomendadas. Esta abordagem pode causar certa confusão na interpretação da URI por não adicionar valor semântico e passar a impressão da existência de mais relacionamentos;
- Não incluir extensões de arquivos no final das URIs. Sabendo que URIs na verdade são *links* que fazem acesso a algum tipo de representação à arquivos, não devemos expor qual extensão a representação faz parte. Devemos deixar a função de identificar o tipo de arquivo manipulado a cargo dos atributos *Content-Type* e *Accept* dos *headers* mensagens trafegadas entre cliente e servidor.

8.2 VERSIONANDO RECURSOS

Uma API, assim como um software qualquer, deve possuir um controle de versão que indique a evolução dos recursos envolvidos, demonstrando quando houve algum tipo de alteração na funcionalidade de algum recurso ou no comportamento da API como um todo, de forma que uma API após ser

desenvolvida, nunca deva ser lançada para produção sem a indicação de sua versão atual.

Basicamente, podemos versionar os recursos de nossa API de três maneiras distintas. Versionamento pela definição de uma *QueryString*, versionamento dentro do caminho do recurso e versionamento no *Media Type* das mensagens.

8.2.1 VERSIONAMENTO EM *QUERY STRINGS*

O versionamento de recursos através do uso de QueryStrings pode ser definido como,“[...] utilização de um parâmetro específico no acesso a todos os recursos, como por exemplo http://api.vivo.com.br/{recurso}?_versao=1”.(FIRMO, 2012, p.4).

As APIs do Facebook e Netflix costumam usar esta abordagem para indicar a versão de seus recursos através da *QueryString* ?v=1.0. No exemplo, http://api.netflix.com/catalogs/titles/series/70023522?v1.5, analisa-se que, indicar a versão na *QueryString* restringe apenas a mudança de funcionalidades para um recurso em particular, uma vez que, a utilização de *queries*, na URI não é obrigatória.

Porém, induz o desenvolvedor a utilizar versões diferenciadas dos recursos da API para construir sua aplicação. Uma abordagem deste tipo pode influenciar na manutenabilidade e escalabilidade do aplicativo, pois quando as versões antigas dos recursos forem atualizadas, o desenvolvedor será obrigado a mudar cada uma das funções de sua aplicação para se adaptarem as mudanças de funcionalidade dos novos recursos.

8.2.2 VERSIONAMENTO NO CAMINHO DO RECURSO

Este princípio é o mais utilizado na construção de APIs REST e dita que a definição da versão dos recursos se dá por um prefixo adicionado antes da indicação do recurso em si e podem ser feitas de diversas maneiras.

Podemos citar como exemplo os recursos da API da *Del.icio.us* (http://api.del.icio.us/v1/posts/updates) que usa o prefixo “v1” antes da definição dos recursos da API. Já os recursos da API da *Last FM* (http://ws.audioscrobbler .com/2.0), utilizam apenas o número da versão em suas URIs e os recursos da API

do *Twilio* (https://api.twilio.com/2010-04-01), tem a tendência à usar datas para indicar a ultima versão de seus recursos.

Porém, se forem desenvolvidas APIs seguindo o estilo *Pragmatic REST*, o mais indicado segundo Mulloy (2012, p.14) é usar um prefixo simples contendo uma única letra e um número ordinal como, por exemplo, “v1”, “v2” ou “v3”. Pois defende que, a utilização de datas, ou de prefixos que indicam granularidade como “2.0”, implica ao consumidor da API que os recursos sofrem modificações com mais frequência que o habitual, o que pode acarretar maiores adaptações em funcionalidades de um aplicativo que consumir esses serviços.

8.2.3 VERSIONAMENTO NO MEDIA TYPE DAS MENSAGENS

Para este tipo de abordagem, Firmo (2012, p.5) afirma que devem ser criados *Media Types* customizados que serão inseridos nas mensagens de requisição e resposta dentro dos atributos *Accept* e *Content-Type* dos cabeçalhos das mensagens de cada um deles conforme Figura 20.

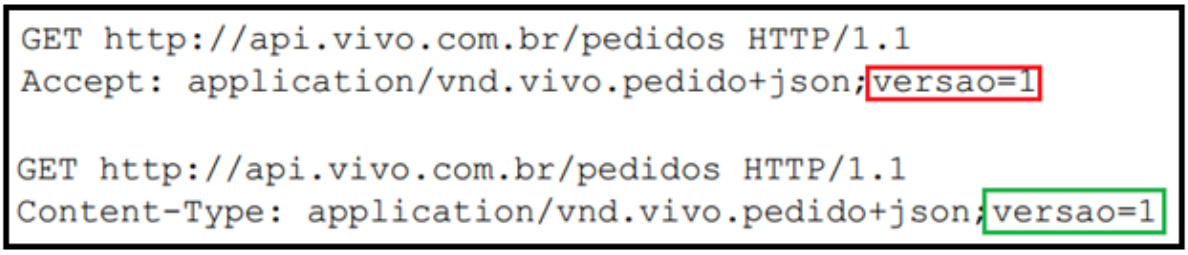


Figura 20 – Exemplo de sintaxe de cabeçalhos de mensagens de requisição (em vermelho) e mensagens de resposta (em verde) com a adição de parâmetros de versionamento.

Fonte: Próprio autor

Nota-se que a indicação da versão nas mensagens HTTP permite o versionamento direto das representações que o recurso “/pedidos” manipula para um determinado método HTTP, tanto para envio quanto para recuperação de informações do servidor, retirando estas informações do conteúdo da URI. Entretanto, esta abordagem não é indicada, pois exige que tanto cliente quanto servdor deêm suporte a estes *Media Types* customizados, o que exigiria um conjunto de configurações adicionais.

Após leituras de Firmo (2012, p.4) e Richardson (2007, p.235), podemos concluir que versionar uma API permite que clientes que consomem serviços de

uma versão não sejam afetados pelas mudanças ocorridas em certas funcionalidades, pois as atualizações estarão presentes emum conjunto de serviços mais recente, permitindo que desenvolvedores utilizem os serviços antigos enquanto preparam a migração da infraestrutura de suas aplicações para novas funções da API mais atual.

8.3 MANIPULAÇÃO DE EVENTOS DE ERRO E SUCESSO

Em se tratando de sistemas Web, temos que uma API é similar a qualquer sistema projetado para fornecer serviços através da rede, porém, com o uso frequente, ela esta sujeita a expor uma série de erros que impedem que determinada operação seja executada.

Portanto, uma API deve ser projetada de maneira a demonstrar ao usuário quando uma operação foi realizada com sucesso ou quando ocorreu algum tipo de erro, de forma que esta indicação deve ser feita da maneira mais intuitiva possível proporcionando uma verificação rápida do problema, onde o próprio usuário deve ser capaz de encontrar uma solução viável.

Segundo Mulloy (2012, p.14), a importância da projeção de mecanismos de identificação de erros em uma API REST se da pelo fato de que desenvolvedores de aplicativos, ao utilizarem um serviço qualquer, não têm acesso direto aos códigos implementados por determinado recurso de uma API, todo o processamento é implícito a eles. A única maneira de um programador aprender como uma API funciona é analisando as respostas vindas do servidor.

A indicação do estado da resposta de um servidor está no uso dos códigos de resposta HTTP como apresentado no capítulo 5.3, complementados por mensagens de erro presentes no corpo da mensagem de resposta do servidor, formatados a partir da representação escolhida.

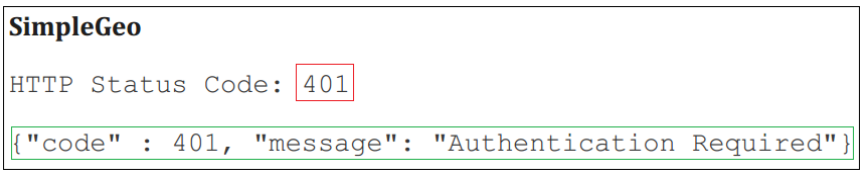


Figura 21– Exemplo de resposta do servidor representando erro com uma mensagem em formato JSON

Fonte: Adaptado de MULLOY (2012, p.10)

Na Figura 21 temos o exemplo usado pela *SimpleGeo*, no caso apresentado, ocorreu um erro de validação na autenticação do usuário, no qual é verificado que a mensagem de resposta retornada pelo servidor contém o código HTTP que representa este tipo de erro, além do objeto JSON que possui o atributo *message*, onde seu conteúdo possui um alerta que representa textualmente o erro ocorrido. Este objeto será importante para que o desenvolvedor possa ter uma ideia mais ampla sobre o evento ocorrido.

Mulloy (2012, p.11) defende que, dentre os mais de 70 códigos HTTP existentes, devemos escolher apenas os mais relevantes para representação dos eventos em nossa API levando em consideração os seguintes fatores:

- Representar quando operações foram bem sucedidas;
- Representar quando algo na aplicação que consome a API deu errado;
- Representar quando ocorreu um erro em operações no servidor.

Para cobrir os três fatores citados acima devem ser utilizados no mínimo três códigos, sendo o código 200 (*OK*) representado sucesso em operações, o código 400 (*Bad Request*) indicando erros no cliente e o código 500 (*Internal Server Error*) para representar erros ocorridos no servidor.

É recomendado o uso adicional de mais cinco códigos para aumentar o grau de detalhamento de determinadas operações, por exemplo, podemos usar o código 201 (*Created*) para indicar que operações de criação e atualização foram bem sucedidas, usamos também, como apresentado na Figura 21, o código 401 (*Unauthorized*) para caracterizar erros de autenticação.

Outros códigos que podem ser utilizados são o 403 (*Forbidden*), para barrar usúarios de acessar determinados recursos, 404 (*Not Found*) para demostrar quando o recurso acessado não existe no servidor e por fim usamos o código 304 (*Not Modified*), onde sua função é indicar que uma entidade não sofreu alterações na ocorrência de uma operação do tipo *PUT*, onde houve a requisição de atualização de determinada entidade, mas não existem diferenças nas informações enviadas para sustentar tal operação.

Lembrando que cada erro ocorrido no servidor deve, com o objetivo de auxiliar quem utiliza a API, ter no corpo da mensagem de resposta uma mensagem

que indique o evento ocorrido. Isso é importante, pois podemos perceber que alguns dos códigos HTTP apresentados possuem um nível muito grande de abstração em suas interpretações. Por exemplo, um evento que tenha retornado o código 404 apenas, não é suficiente para identificarmos qual problema ocorreu e muito menos o que deve ser alterado para que a operação obtenha sucesso.

8.4 DESIGN DE REPRESENTAÇÕES

Uma representação conforme explicado em seções anteriores e analisado pela leitura de Allamaraju (2010, p.45), são as caracterizações concretas dos recursos presentes em nossa API, ou das entidades e objetos por ela retornados.

Portanto o design das representações envolve, a definição correta dos cabeçalhos (*headers*) das mensagens de requisição e resposta e a escolha correta dos *Media Types* para envio dos dados requisitados por um cliente quando existirem.

A escolha do *Media Type* é o *application/json*, portanto, nossa discussão apresentará como os objetos JSON devem ser formatados no corpo das mensagens de resposta e analisará como os cabeçalhos, que a partir de agora chamaremos de *headers*, devem ser definidos.

8.4.1 DEFININDO O FORMATO DAS MENSAGENS JSON

Partindo do princípio de que o formato das mensagens que trafegarão entre clientes e servidores está nos padrões definidos por JSON, destaca-se que a construção dos objetos dentro deste formato deve ser bem definida.

Como explicado na seção 5.5.2, JSON suporta apenas tipos limitados de atributos. Devido a esta característica, os atributos de um objeto JSON devem seguir regras indicadas por Massé (2012, p.48). Para ilustrar melhor estas regras vamos utilizar um exemplo que mostra um objeto JSON bem formatado.

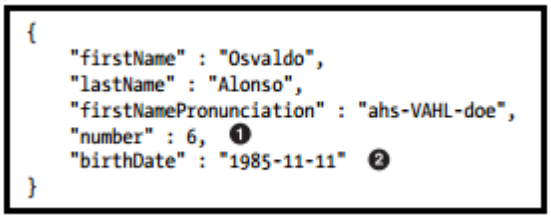


Figura 22 - Exemplo de objeto JSON bem formatado

Fonte: Adaptado de MASSÉ (2012, p.48)

Podemos destacar pela análise da Figura 22 que:

- Todos os nomes dos atributos de um objeto JSON devem ser envolvidos por aspas duplas;
- Os nomes devem ser todos representados por letras minúsculas e não devem possuir caracteres especiais;
- Atributos do tipo *String* devem possuir seu valor também envolvido por aspas duplas, como é o caso dos atributos *"firstName"* e *"lastName"*;
- Atributos que possuem representações com valores numéricos não necessitam de aspas duplas para serem definidos, podem ser inseridos de maneira direta, como no caso do atributo *"number"*;
- JSON não suporta atributos do tipo data, portanto a representação deste tipo deve ser similar aos atributos do tipo *String*, como é o caso do atributo "*birthDate*";
- Caso seja necessário que um único atributo armazene uma série de valores, podem ser usados atributos do tipo vetor[8], onde o conjunto é colocado entre colchetes e separado por virgulas, como por exemplo: "numbers" : [1, 2, 3]. No exemplo temos o armazenamento de uma série de valores do tipo inteiro dentro da variável "*numbers*" que passa a ser um vetor de números inteiros.

8.4.2 DEFININDO *HEADERS*

W3C (2012) defende que *headers* são componentes que definem os parâmetros de transação de mensagens através do protocolo HTTP. Os *headers*

[8] Segundo http://www.jsonexample.com/, Vetor ou Array é um tipo de variavél que tem a capacidade de armazenar diversos valores de um unico tipo, em JSON temos a vantagem de que o tipo da variavel é automáticamente identificado pela analise do conteúdo do vetor, não sendo necessário sua indicação.

que serão utilizados devem possuir um conjunto de atributos que irá garantir que as informações presentes no corpo da mensagem irão trafegar de maneira íntegra.

Allamaraju (2008, p.46) cita que para mensagens de resposta, os atributos principais de um *Header* são *Content-Type*, *Content-Language*, *Content-MD5*, *Content-Length*, *Content-Encoding* e *Last-Modified*. Contudo, analisando implementações de APIs REST no geral e tendo em vista que será adotado o estilo *Pragmatic REST* para construção de APIs, podemos definir que as mais importantes são:

- *Content-Type*: Define qual o *Media Type* utilizado em nossa API apontando qual formato as respostas do servidor serão escritas e como elas devem ser manipuladas pelos clientes. Deve ser estabelecido com o atributo *charset*, ele define qual a codificação de caracteres esta sendo usada na descrição da representação do objeto, onde o valor mais utilizado para esse atributo é o UTF-8;
- *Content-Language*: Especifica qual o idioma nossa representação esta escrita;
- *Last-Modified*: Indica qual periodo a representação do recurso sofreu sua última modificação em suas informações;

Abaixo é apresentado na Figura 23 um exemplo de mensagem de resposta que contém estes atributos dentro do *header*.

Figura 23 – Demonstração do uso dos atributos Content-Type, Content-Language e Last-Modified em mensagens de resposta.

Fonte: Adaptado de ALLAMARAJU (2008, p.47)

Também é importante destacar os atributos utilizados nos *headers* das mensagens de requisição. Nesse caso, o atributo mais importante e usado é o *Accept*. Este atributo, segundo W3C (2012), tem a função de indicar para o servidor

que o cliente que está chamando um serviço, está preparado para receber a representação do objeto adquirido no formato estabelecido pelo servidor, informando quais são as preferências do cliente com relação aos *Media Types* que ele deverá manipular.

A Figura 24 mostra um exemplo de atributo do tipo *Accept* escrito em um *header*, preparando o cliente para recepção de informações no formato *application/json*.

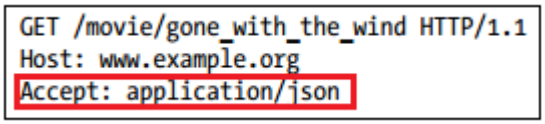


Figura 24 – Demonstração do uso do atributo Accept em mensagens de requisição
Fonte: Adaptado de ALLAMARAJU (2008, p.127)

Os atributos dos *headers* de requisição e resposta são importantes para a execução de troca de informações entre servidor e cliente por causa de um processo chamado “Negociação de Conteúdo”. Allamaraju (2008, p.123) define que a negociação de conteúdo é o processo de escolha e indicação das representações que trafegam pela rede quando existe suporte a múltiplas representações e existe a necessidade de escolha de um tipo de representação que seja compatível entre cliente e servidor.

Para que a negociação de conteúdo seja executada com sucesso é necessário sincronia das configurações do atributo *Accept* e do atributo *Content-Type*, em que cada um deles deve possuir o mesmo valor, apontando que tanto servidor quanto cliente trabalham com o mesmo tipo de informação.

Na Figura 25, analisa-se um exemplo de configuração onde tanto clientes quanto servidor estão preparados para trabalhar com dados do tipo JSON pela definição do *Media Type application/json*.

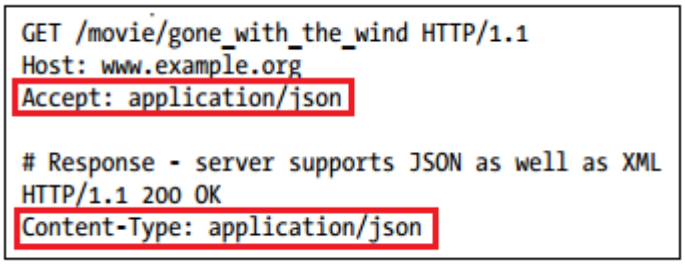


Figura 25 – Demonstração de configuração de Media Types para negociação de conteúdo entre clientes e servidores

Fonte: Adaptado de ALLAMARAJU (2008, p.127)

A configuração incorreta destes atributos pode causar erros quando a API for utilizada. A Figura 26 ilustra um exemplo em que o cliente está configurado para receber informações no formato JSON, entretanto, o servidor está configurado para formatar e retornar objetos no formato XML causando uma inconsistência nos dados trafegados. Nesta ocasião, o servidor é obrigado a retornar o erro com código 415 (*Unsupported Media Type*), acusando que tanto cliente quanto servidor não manipulam o mesmo conjunto de dados, ou alguma das partes não dá suporte para recepção do formato de dados estabelecido.

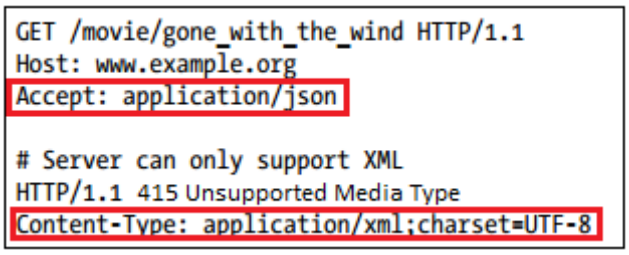


Figura 26 – Exemplo de configuração de Media Types inadequada para negociação de conteúdo entre clientes e servidores

Fonte: Adaptado de ALLAMARAJU (2008, p.127)

8.6 RESPOSTAS PARCIAS E PAGINAÇÃO DE RESULTADOS

Estabelecer um método para exibir resultados de uma consulta de maneira parcial é importante quando nossa API tem a capacidade de retornar uma quantidade enorme de informações ao cliente.

Mulloy (2012, p.16) defende que muitas das informações que são retornadas em uma requisição podem não ser úteis ao desenvolvedor na construção de uma aplicação, para solucionar este problema podemos usar o conceito de

respostas parciais como maneira de trazer apenas os campos importantes de um recurso solicitado.

Para definir uma resposta parcial, é estabelecido os campos que serão buscados através da descrição de uma *query* em nossa URI, sendo que a sintaxe da *query* deve conter um parâmetro com o nome *"fields"* para apontar que a restrição será aplicada a campos e atributos das entidades do recurso solicitado. Cada campo escolhido para ser filtrado deve ser separado por vírgulas, onde a *query* montada ao final fará referência exata dos atributos que estão presentes em uma entidade como ilustrado na Figura 27.

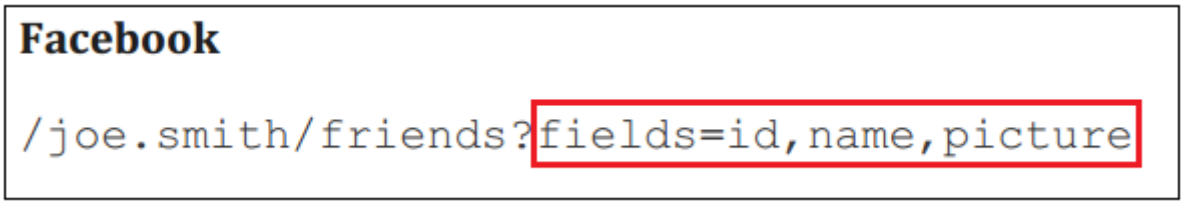


Figura 27 – Definição de *query* em URI para resposta parcial

Fonte: Adaptado de MULLOY (2012, p.16)

Na figura 27, verifica-se um exemplo de URI para respostas parciais usados pelo Facebook. A URI ilustrada tem a capacidade de trazer informações relacionadas a todos os amigos de Joe Smith. Tendo em vista que a entidade "*friends*" presente na URI pode ser a representação de outros usuários do Facebook, podemos perceber que o recurso "*/friends*" retornaria uma quantidade enorme de informações irrelevantes ao consumidor do serviço, porque cada entidade teria uma infinidade de dados relacionados a ele.

Entretanto, através da *query* (marcada em vermelho), percebe-se que é aplicada uma restrição ao recurso "*/friends*", assim as entidades retornadas pelo servidor terão apenas os campos "id", "*name*" e "*picture*", tornando o uso deste serviço mais viável, pois produzirá representações de objetos com informações reduzidas.

Firmo (2012, p.7) prefere resaltar que outra maneira de filtrar informações de uma requisição de consulta é através da paginação de resultados.

A abordagem de definição de paginação é parecida com a definição de respostas parciais, com a diferença dos parâmetros utilizados na *query* que são o *limit* e o *offset* ilustrados na Figura 27, onde nota-se a função dos parâmetros *limit* e *offset* a partir deste exemplo.

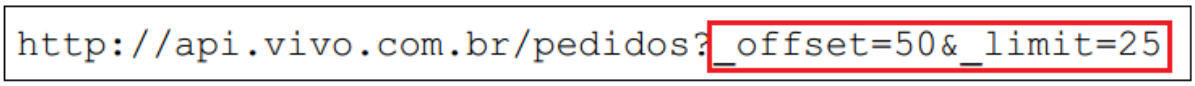


Figura 28 – Definição de *query* em URI para paginação usando parâmetro *offset* e *limit*
Fonte: Adaptado de FIRMO (2012, p.16)

Suponha que o recurso “/pedidos” depois de invocado tenha a capacidade de retornar mais de 1000 registros. Sem uma restrição de paginação, todos os 1000 registros seriam exibidos na tela tornando a interpretação dos resultados incompreensível.

Neste caso o parâmetro *limit* tem a função de filtrar a quantidade de resultados que são exibidos por página, que segundo apresentado na Figura 27 este limite é de 25 registros. Já o parâmetro *offset* tem a função de indicar a partir de que registro começa a contagem para exibição do restante de registros de nossa página, que no caso seria a partir do registro 50.

Fazendo análise da *query* apresentada por completo temos que, serão exibidos 25 registros na página a iniciar pela contagem do registro 50, definido pelo parâmetro *offset*, até o registro 75 que é o registro final definido pelo parâmetro *limit*.

Então para utilizar a sintaxe em questão como instrumento de paginação, é necessário ter em mente que ela possui uma lógica específica que não envolve indicar a página diretamente e sim se basear na quantidade de registros que serão mostrados.

Esta abordagem ainda é a mais utilizada entre desenvolvedores de APIs por que a lógica de utilização é bem conhecida por muitos programadores, o que facilita no momento da implementação de funcionalidades que retornam respostas baseadas em paginação.

Portanto temos que, a resposta parcial é usada para filtrar os tipos de campos por registro e a paginação tem a função de filtrar a quantidade de registros por página.

8.7 MECANISMOS DE SEGURANÇA PARA APIS

Manter uma API segura é um aspecto importante, assim, desenvolvedores de uma API REST devem garantir que existirá um mecanismo que restrinja acesso a usuários não autorizados evitando o consumo indevido e inadequado dos serviços disponibilizados.

Um dos métodos mais conhecidos é a utilização de *API Keys*[9]. Elas, segundo Jacobson et al (2011, p.77), são chaves únicas que provedores de uma API usam para identificar as aplicações que estão utilizando seus serviços, sem passar por um processo de autenticação rígido que os mecanismos baseados em senha providenciam, nem por processos de criptografia sofisticados, o que torna a escolha desta alternativa altamente viável, principalmente para implementação de APIs com poucos serviços.

Estas chaves constituem identificadores simples, na qual quem desejar acessar uma API deve passar este identificador através de uma *query* colocada na URI do recurso a cada requisição que for feita ao servidor, como ilustrado na Figura 28.

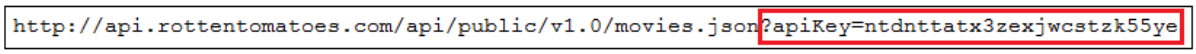


Figura 29 – Utilização de API Key (em vermelho) na identificação de aplicações que consomem serviços da API da *Rottentomatoes*

Fonte: Próprio autor

A chave de identificação é adquirida quando um potencial cliente, interessado em utilizar os serviços da API, realiza cadastro no portal do provedor da API, indicando qual será a aplicação prestes a consumir os serviços disponibilizados. Então, após cadastro, o provedor da API fornece uma chave para que o cliente possa invocar seus serviços. Aqui, conclui-se que a abordagem de autorização por *API Keys* é mais amplamente utilizada em APIs públicas onde a disponibilização de

[9] Segundo http://developer.shopping.com/kb/02n400000009R8wAAE/501400000009apNAAQ, *API Keys* são identificadores unicos gerados por sistemas por sistemas de gerênciamento de APIs, onde estes identificadores são formados por um conjunto de caracteres alfanuméricos de até 32 bits.

serviços acontece através da internet e o provedor dos serviços necessita de uma maneira de gerenciar os acessos à API.

Entretanto, se a API implementada manusear informações de alto valor significativo, o processo provido pelas *API Keys* não é o suficiente para manter as informações íntegras apenas a quem detém direito, pois não é uma ferramenta de autenticação, ela apenas restringe acesso de alguns tipos de usuários a determinados tipos de funcionalidades presentes na API através de uma verificação simples da chave passada pelo cliente.

Porém, se o objetivo buscado é o fortalecimento dos processos de auteticação visando aumentar a segurança da nossa API, é recomendado o uso do mecanismo *OAuth*[10].

Para IETF (2012), o mecanismo é basicamente um *framework* de autenticação que possibilita que aplicações de terceiros possam ter acesso restrito a serviços baseados no protocolo HTTP, orquestrando as interações de aprovação entre o proprietário dos serviços e os métodos HTTP para habilitar o acesso as funcionalidades disponíveis.

Mafra (2012) acredita que o objetivo principal do *framework OAuth* é permitir que uma aplicação possa se autenticar em outras aplicações para compartilhamento de informações, utilizando direitos de determinado usuário, porém sem a necessidade do uso e compartilhamento de senhas e nomes de usuários entre as aplicações, como ocorre com arquiteturas mais antigas como o *HTTP Basic Authentication*, mas sim usando credenciais próprias, onde o *framework* tem a capacidade de gerar um instrumento chamado *token*.

Este *token* possui propriedades similares a da *API Key*, entretanto, ao invés de indicar qual é a aplicação que consome os serviços, ela tem a função de representar o usuário que deseja ter acesso aos serviços fazendo o papel de identificador de usuários e ferramenta para o processo de autenticação usado pelo framework *OAuth*.

[10] Mais informações sobre OAuth:
http://www.railstips.org/blog/archives/2009/03/29/oauth-explained-and-what-it-is-good-for/
http://marktrapp.com/blog/2009/09/17/oauth-dummies

É importante resaltar que os mecanismos de segurança têm mais destaque no processo de gerenciamento de uma API levando em consideração o ambiente onde que será integrado, porque os processos de obtenção de *API Keys* e *tokens* de autenticação são estabelecidos por estes sistemas de gestão de APIs e devemos citá-los, pois o processo de aplicação de restrições é integrada a codificação da API em âmbito geral.

9 DESENVOLVENDO APIs

Neste capítulo apresentar-se-á uma API desenvolvida tomando como princípios alguns conceitos apresentados no capítulo 8. A arquitetura da API que será desenvolvida teve como base a construção de um conjunto simples de serviços relacionados a cadastro e consultas de veículos e multas de trânsito, onde a construção dos serviços desta API foi baseada nos tutoriais de Vogel (2012) e 308Tube (2013).

9.1 FERRAMENTAS E INSTRUMENTOS NECESSÁRIOS

Para a construção de uma API REST qualquer, são necessárias as ferramentas a seguir:

a) **Java JDK 1.6.0_22 e JRE 1.6.0_22:** São basicamente plataformas de desenvolvimento que permitiram a compilação e execução de programas baseados na linguagem Java. Foi estritamente necessário ter a plataforma Java instalada no ambiente computacional relacionado, porque ela foi responsável pela execução dos aplicativos programados utilizando sua linguagem;
b) **Eclipse Helios Java EE IDE 1.3.1.20100916-1202:** É uma plataforma de programação para ambiente de desenvolvimento *Web* baseada na linguagem Java. Fora utilizado para a inscrição e compilação dos códigos da API exemplo;
c) **Apache Tomcat 6.0.35:** Esta ferramenta é um servidor de aplicação, segundo I-Web (2008) a função dela foi providenciar mecanismos para instalação e execução de aplicativos de forma distribuída para serem acessadas no formato cliente-servidor, segundo exposto no capítulo 5.1. Dentro dela foram hospedados todos os arquivos e serviços referentes à API;

d) **Eclipselink 2.4.1:** Segundo Vogel (2008), o Eclipselink é um componente de persistência de dados que possui o objetivo de mapear objetos Java de uma aplicação e transformá-los em entidades de um banco de dados qualquer. Foi integrado ao código do projeto permitindo que fossem realizadas todas as operações referentes ao banco de dados, como retirada, exclusão e atualização de registros de maneira direta;

e) **Apache Derby 10.9.1.0:** Reponsável por agir como ambiente de construção do banco de dados que armazena as informações das entidades construídas dentro da API;

f) **Jersey RESTful Services 1.17:** Pacote de bibliotecas Java, que para Vogel (2012), disponibiliza uma série de funcionalidades para a programação de serviços REST, como um conjunto de funções para criação e manipulação de objetos JSON e mecanismos de simulação de ambientes clientes e servidores para comunicação REST utilizando protocolo HTTP.

g) **Firefox 20.0.1:** Toda API é na verdade um aplicativo Web, portanto para testar a API implementada de maneira eficiente foi usado um *browser* de internet para acesso a seus serviços;

h) **RESTClient 2.0.3:** Esta ferramenta é originalmente um complemento para o *browser* Firefox, ela tem a capacidade de realizar requisições a serviços RESTful de maneira fácil e rápida, simulando todas as operações possíveis de um recurso através da invocação dos 4 métodos HTTP e mostrando os resultados retornados por um servidor de maneira instantânea, através da formatação dos dados na representação escolhida. Este mecanismo foi utilizado com o intuito de testar as funcionalidades implementadas em nossa API.

9.2 CONSIDERAÇÕES INICIAIS

É muito importante que a leitura das próximas seções seja realizada com um conhecimento prévio sobre padrões de programação Java para Web, pois a implementação da API mostrada nesta obra utiliza padrões de programação em

camadas, padrão *singleton*[11] para instância de objetos relacionados à serviços, utilização de anotações em classes e uso de mecanismos de persistência de dados.

Também deve ser levado em consideração que o ambiente de programação da API apresentada está dentro dos padrões do sistema operacional *Windows 7Ultimate Service Pack 2 64 bits* e deve possuir as ferramentas *Eclipse Helios Java EE IDE*, *Java JDK 1.6.0_35 e JRE 1.6.0_22, Apache Derby 10.9.1.0* e *Apache Tomcat 6.0.35* devidamente instaladas e configuradas.

Todos os serviços e simulações foram feitas localmente, ou seja, os serviços implementados não têm o objetivo de serem disponibilizados a nenhum tipo de público. O objetivo principal é apenas apresentar o uso de uma API projetada segundo regras e boas práticas de implementação REST.

9.3 DEFINIÇÃO DA ARQUITETURA DE DADOS DA API

A primeira atividade que fora realizada segundo análise dos princípios de design é a definição dos recursos de nosso sistema. Como o sistema de exemplo é uma API para cadastro de veículos e multas de trânsito, é definido duas entidades principais, uma entidade para representar objetos relacionados à veículos (*Vehicle*) e outra para representar objetos relacionados à multas (*Ticket*) como ilustrado na Figura 29.

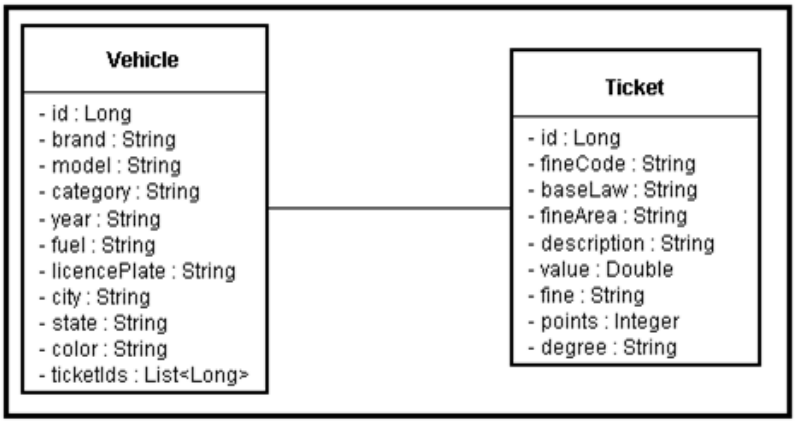


Figura 29 – Definição da arquitetura de dados para as entidades veiculos (*Vehicle*) e multas (*Ticket*)

Fonte: Próprio autor

[11] De acordo com Whatis (2005), em programação Java, *singleton* é uma classe que possui apenas uma única instância que o representa durante a execução do código.

A definição das entidades mostradas acima foi realizada via pesquisa rápida[12] sobre atributos que constituem entidades relacionadas a veículos (*Vehicle*) e multas (*Ticket*), onde o objeto *Vehicle* é constituído dos atributos: id, marca (*brand*), modelo (*model*), categoria (*category*), ano (*year*), combustível (*fuel*), placa (*licencePlate*), cidade (*city*), estado (*state*), cor (*color*) e *ticketIds* (referência para as multas cadastradas no sistemas caso o carro esteja multado).

A entidade *Ticket* contém os atributos: id, código da multa (*fineCode*), Enquadramento no CTB[13] (*baseLaw*), área de validade[14] (*fineArea*), descrição (*description*), valor (*value*), pontos (*points*), gravidade (*degree*).

Estas entidades basicamente constituem o banco de dados de nossa API, onde será armazenada todas as informações que vierem de chamadas a serviços das URIs descritas na seção a seguir e tammém constituirão as representações em formato JSON das nossas entidades.

9.4 DESCRIÇÃO DAS URIs DOS RECURSOS DA API

A partir da definição das entidades descritas no tópico anterior, aponta-se quais serão as URIs que compõem os recursos da API. Entretando, a priori, segundo princípios apresentados no capitulo 8.1, temos a definição da URL que representará o domínio, ou autoridade da API com a função de indicar o local do servidor provedor dos serviços.

Para tal, ocorre indicação do protocolo de dados utilizado no servidor como o *http*, é definido o setor autoridade da nossa URL principal como sendo *api.speedtickets.rest*[15] e em seguida a versão da API através do sulfixo *v1* no caminho do recurso seguindo regras apresentadas no capítulo 8.2.1.

[12] Lista de multas de trânsito: http://www.detran.pr.gov.br/arquivos/File/infracoesdetransito/tabeladeinfracoesepenalidades.pdf

[13] Série de regulamentos onde as leis são baseadas.

[14] Nivel de área onde a multa é aplicada.

[15] O nome da URL foi baseado no nome do projeto no momento de sua construção, seguindo padrões de implementação de serviços Jersey RS,como explicado no tutorial de 308Tube(2013), tendo função ilustrativa apenas.

Por fim, tem-se uma URL base igual à apresentada na Figura 30.

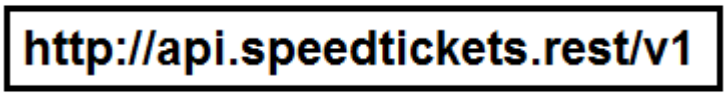


Figura 30 - URL base da API implementada

Fonte: Próprio autor

Após a definição da URL principal, foi estabelecido as URIs dos recursos utilizados para indicar os serviços da API. Segundo análise da Figura 29 e conteúdo apresentado no capítulo 8.1, temos a demonstração das seguintes URIs, segundo Tabela 2, onde mostramos a função de cada recurso mapeado para o método HTTP utilizado no serviço.

Tabela 2 - Demostração das URIs disponíveis na API construída e suas funções.

Recurso	***GET***	***POST***	***PUT***	***DELETE***
/tickets	Retorna informações sobre todas as multas cadastradas no sistema.	Cadastra uma nova multa na coleção existente.	Método não suportado, servidor retorna erro 405.	Método não suportado, servidor retorna erro 405.
/vehicles	Retorna informações sobre todos os veiculos cadastrados.	Cadastra um novo veículo na coleção existente.	Método não suportado, servidor retorna erro 405.	Método não suportado, servidor retorna erro 405.
/tickets/{id}	Exibe detalhes da multa segundo *id* passado na URI.	Método não suportado, servidor retorna erro 405.	Atualiza informações da multa de acordo com *id* passado.	Exclui uma multa da coleção dado seu *id*.
/vehicles/{id}	Exibe detalhes do veiculo segundo *id* passado na URI.	Método não suportado, servidor retorna erro 405.	Atualiza informações do veiculo de acordo com *id* passado na URI.	Exclui um veiculo da coleção dado seu *id*.
/vehicles/{id}/tickets	Exibe detalhes de todas as multas aplicadas ao veículo de *id* correspondende.	Método não suportado, servidor retorna erro 405.	Método não suportado, servidor retorna erro 405.	Método não suportado, servidor retorna erro 405.

Fonte: Próprio autor

9.5 CRITÉRIOS PARA CONTROLE DE ACESSO A RECURSOS

Na seção anterior, apresentou-se todos os recursos contidos na API projetada em questão. Porém, como apresentado no capítulo 8.7, não podemos permitir que qualquer tipo de usuário tenha acesso aos serviços providenciados.

Com o objetivo de simular uma situação onde ocorre aplicação de restrições, usamos o mecanismo de *API Keys* para identificar qual usuário consome a API, assim impedimos o acesso à serviços por usuários não autorizados e regulamos o acesso de determinado perfil de usuário à algumas funções disponíveis na API.

Para a simulação, definiram-se dois perfis de usuários meramente fictícios:

- Administrador: Este usuário possui acesso total aos recursos da API, sendo possível realizar invocações a qualquer tipo de URI disponível para qualquer um dos quatro métodos mostrados na Tabela 2.
- Visitante: Este usuário não tem direito de executar recursos com métodos do tipo POST e PUT, ou seja, ele não tem direito de inserir ou alterar informações disponíveis na API. Só sendo capaz de visualizar estas informações através da invocação de recursos mapeados para o método GET.

Assim, a API construída possui duas *API Keys* definidas para demonstração, “4dm1nk3y4uth0r1t3” para Administradores e “gu3stk3y4uth0r1t3” para Visitantes. Sendo assim, para chamada e execução satisfatória dos serviços presentes na API, fica obrigatória a passagem de umas destas *API Keys* pela *query apiKey* como ilustrado na Figura 31.

```
http://api.speedtickets.rest/v1/vehicles?apiKey=gu3stk3y4uth0r1t3
```

Figura 31 - Chamado a um serviço da API por um usuário com perfil de Visitante
Fonte: Próprio autor

Os detalhes dos resultados retornados por utilização de *API Keys* não cadastradas, inválidas, ou que possuem restrições de funcionalidades são apresentadas a seguir.

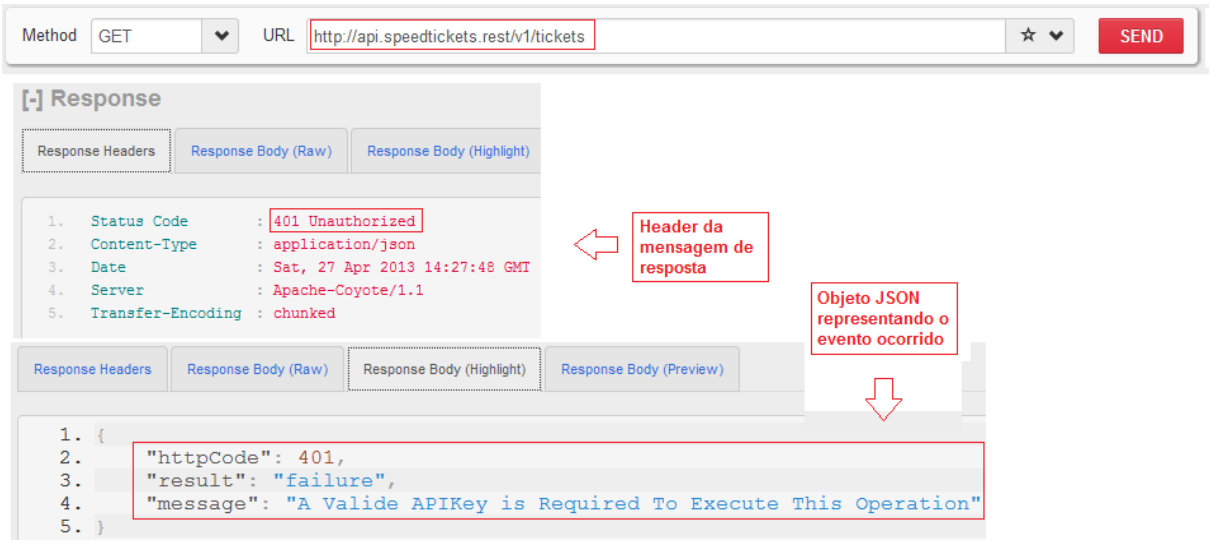


Figura 32 - Resposta do servidor para chamada a serviço sem *API Key*

Fonte: Próprio autor

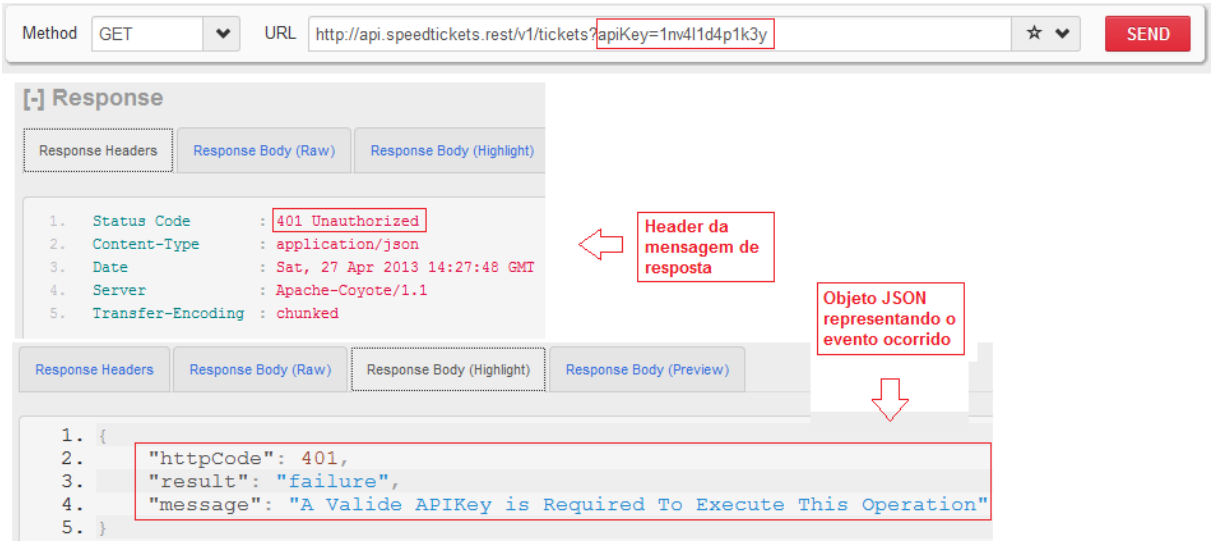


Figura 33 - Resposta do servidor para chamada de serviço com *API Key* inválida

Fonte: Próprio autor

Figura 34 - Resposta do servidor para tentativa de acesso a recurso com API Key de perfil não autorizado

Fonte: Próprio autor

Nas Figuras 32, 33 e 34 tem-se a demostração das respostas do servidor para cada um dos eventos de chamada a serviços da API, onde na Figura 32 ocorre a tentativa de chamar o recurso "*/tickets*" sem informar uma *API Key* e na Figura 33 observa-se a mesma chamada, entretanto, passando uma *API Key* diferente das definidas no início deste capítulo. Na figura 34 é demostrado um exemplo de tentativa de chamada a um serviço usando uma *API Key* que não possui permissão de acesso, no caso para o método *POST* do recurso "*/tickets*".

Nota-se que as respostas são as mesmas na Figura 32 e 33, já na Figura 34 temos a apresentação de uma resposta diferenciada onde é indicado que a *API Key* passada representa um usuário proíbido de executar tal operação.

Todas as respostas estão montadas de acordo com o conteúdo apresentado no capítulo 8.3, onde é possível verificar o código de resposta HTTP no *Response Headers* (401 *Unauthorized* ou 403 *Forbidden*), juntamente com o objeto JSON no corpo da mensagem de resposta representando o ocorrido, demonstrando como foi manipulado os eventos de erro para casos de acesso indevido ou inválido.

9.6 DEMOSTRAÇÃO DE INVOCAÇÕES À RECURSOS DA API

Esta seção tem o objetivo de explicar aspectos sobre como ocorre a transferência de dados entre o cliente consumidor dos serviços da API e qual o comportamento do servidor na produção das respostas. Também será destacado

como ocorre a negociação de conteúdo e definição de mensagens e *headers* para transporte dos dados na realização dos serviços.

Para simular as chamadas aos serviços implementados na API exemplo, utilizou-se a ferramenta *RESTClient* em conjunto com o *browser* Firefox.

Antes de realizar a primeira chamada à um recurso disponível dentro da API, deve-se configurar os *headers* das mensagens que serão transportadas, possibilitando a correta negociação de conteúdo entre o cliente invocador e o servidor provedor dos serviços, como apresentado no capítulo 8.4.2

Para tal, dentro do *RESTClient*, é necessário escolher a opção *Headers>Custom Headers*. Neste local são inseridos todos os atributos contidos nos *headers* das mensagens de requisição que serão feitas por nós. Os *headers* das mensagens de resposta são construídos automaticamente pelo servidor ao realizar o retorno da mensagem. Para uma simples simulação não é necessário mais do que o atributo *Content-Type*e *Accept* inserindo o valor *application/json* para a realização correta da troca de mensagens.

Após esta pequena configuração inicial, é invocado o serviço de obtenção de todas as multas cadastradas no sistema. Para tal deve-se inserir no campo URL o endereço do provedor da API seguida da URI "*/tickets*", inserir a *API Key* do administrador do sistema em uma *query* e escolher no campo *Method* qual o método usado por esta URI, que no caso será o método *GET* para recuperação de informações, como ilustrado na Figura 32.

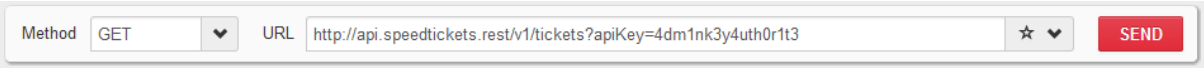


Figura 32 - Inserção de valores nos campos para realização de chamada ao recurso /tickets para consulta de todos os valores cadastrados

Fonte: Próprio autor

Quando o botão *SEND* é clicado, temos que o processo para a obtenção e retorno dos dados ocorre da seguinte maneira:

a) É enviado ao servidor um pedido de requisição para acesso ao recurso "*/tickets*";

b) O servidor começa o processamento da requisição em sua primeira camada de implementação chamada *Controller*, esta camada tem a função de direcionar a requisição para execução do método *GET*, realizar o processo de verificação da *API Key* passada na query da URI, fazer a validação dos dados passados no corpo da mensagem de requisição no caso de métodos *POST* e *PUT* e montar os objetos JSON de resposta que serão exibidos ao usuário;

c) Após a verificação do tipo de perfil do usuário passado de acordo com a *API Key*, o *Controller* faz a invocação da segunda camada, a camada de Serviços. Dentro desta camada estão os processos de busca de informações relacionados ao recurso "*/tickets*", onde dentro dela ocorre a execução de classes do tipo DAO[16] que são responsavéis por fazer o acesso ao banco de dados buscando as informações requisitadas para serem retornados para camada de Serviços;

d) Depois de recuperados pelas classes DAO, as informações sobem as camadas até retornarem a camada *Controller* onde estas informações são fatoradas em objetos JSON e retornadas ao cliente que realizou a requisição inicial produzindo o resultado ilustrado na Figura 33.

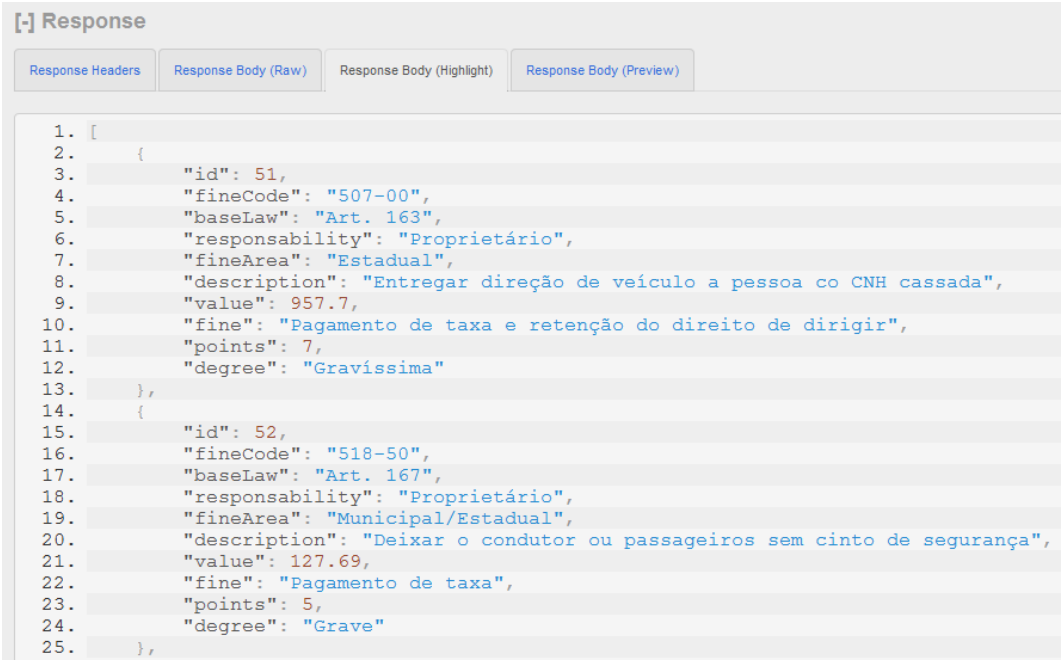


Figura 33 - Resposta à requisição feita ao recurso "/tickets" mapeado com método GET

Fonte: Próprio autor

[16]Segundo Oracle (2002), objetos DAO são usados para encapsular todos os dados relacionados a acesso a base de dados, gerenciando a conexão de dados entre aplicativo e fonte de dados.

Percebe-se que a representação da entidade *Ticket*, retornada após a requisição realizada no exemplo, possui nomes de atributos simílares aos nomes definidos na arquitetura de dados do capítulo 9.3, onde cada atributo segue os princípios indicados na seção 8.4.1 e o formato da representação segue o que foi definido nas configurações iniciais para a negociação de conteúdo de mensagens no formato JSON.

Contudo, o conteúdo mostrado na Figura 33 é apenas parte de toda a resposta produzida pelo servidor pela a chamada do recurso apresentado. Pode-se filtrar a mostragem de uma parte do conteúdo retornado utilizando ferramentas de paginação de resultados.

Para realizar a filtragem destes dados foi escolhido a abordagem de paginação com construção de *query* na URI usando os parâmetros *limit* e *offset* explicados no capítulo 8.6. Aplicando os conceitos na utilização da API, tem-se que as invocações dos serviços se dariam como ilustrado na Figura 34.

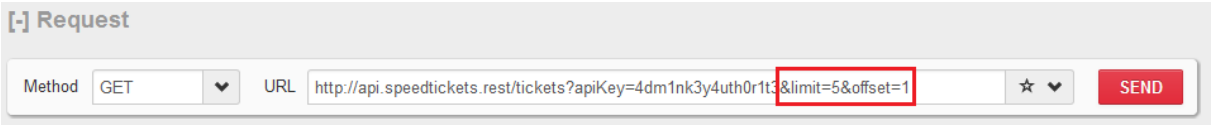


Figura 34 – Utilização de *limit* e *offset* na API implementada

Fonte: Próprio autor

No caso da Figura 34, foi montada uma query para retorno de informações com no máximo cinco registros por página, onde se iniciou a contagem dos registros a partir do registro número 1, provando a utilidade da ferramenta de páginação de resultados explicada no capítulo 8.6. Entretanto, se o identificador do registro que queremos consultar é conhecido, pode-se passá-lo na URI e obter a informação direta sobre aquela entidade.

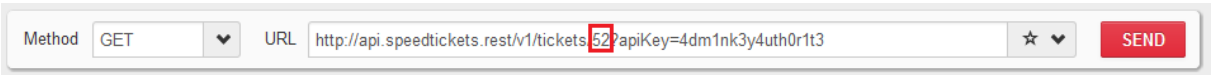


Figura 35 – Exemplo de invocação a serviço para consulta a registro com *id* conhecido

Fonte: Próprio autor

Agora, será demonstrado quais os procedimentos para a chamada à serviços mapeados para o método *POST*. Para tal, será feito uma requisição ao recurso

"*/vehicles*" para cadastrar um novo veículo no sistema. Sendo assim, é necessário além de realizar o preenchimento do campo URL, sempre passando a API *Key* de usuário administrador, escolher o método *POST* no campo *Method* e preencher o campo *Body*, inserindo em seu interior um veículo representado no formato JSON como apresentado na Figura 36.

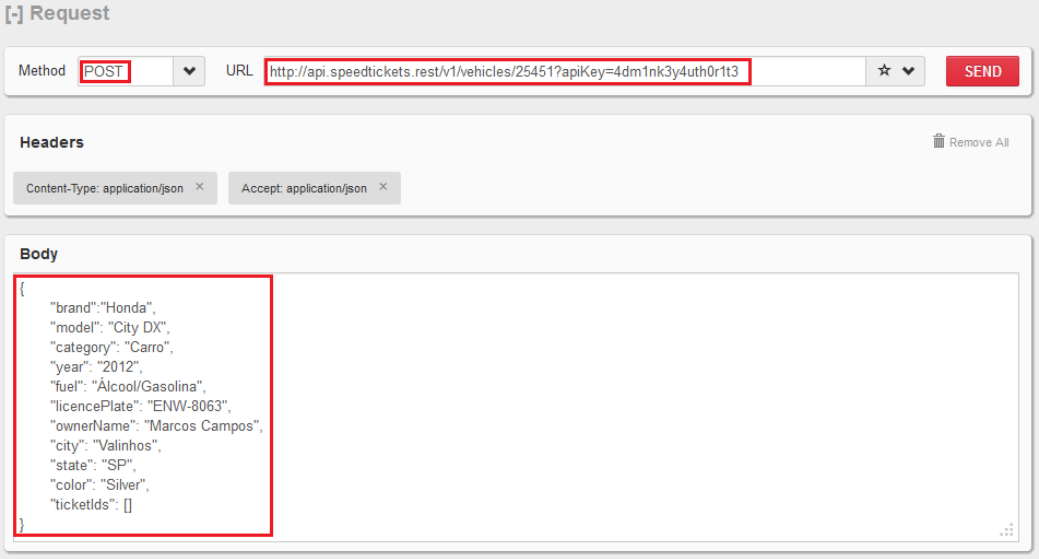


Figura 36 – Exemplo de chamada a recursos para cadastro de novo veículo

Fonte: Criado pelo próprio autor

Na Figura 36 temos a ilustração de como preencher os campos para a criação de um novo veículo via API. Analise que no campo *Body*, não é necessário a apresentação do atributo *id*, isso ocorre porque o sistema foi programado para que o atributo *id* da entidade *Vehicle* fosse gerado de maneira automática, evitando que hovesse algum tipo de erro relacionado à duplicação de chave no banco de dados.

Após o envio das informações pelo botão *SEND*, é apresentado a resposta do servidor com relação ao evento de cadastro.

Figura 37 – Resposta do servidor ao evento de cadastro

Fonte: Próprio autor

Na resposta ilustrada pela Figura 37 temos, na área *Response Headers* a apresentação dos cabeçalhos de resposta retornados pelo servidor, indicando que um registro da entidade *Vehicle* foi criado através de sua representação em JSON. Na área *Response Body*, temos a mostragem da mensagem do servidor que indica o evento ocorrido, demonstrando qual o estado da operação executada pelo servidor mostrando o código HTTP 201 (*Created*), o resultado da operação (*success*), o método HTTP executado (*POST*) e qual o recurso criado através do *link* apresentado no atributo *resource*, assim podemos acessar o recurso diretamente utilizando o método *GET* para verificar se as informações foram criadas corretamente.

Observa-se na Figura 36, que foi criado um carro sem multas, pois o atributo *ticketIds* foi inserido sem conteúdo. Caso necessário indicar que este veículo possui uma ou mais multas, pode-se usar o método *PUT* para atualizar esta entidade.

Entretanto, a forma como foi implementado a operação de atualização exige que seja passado novamente um objeto JSON que represente a entidade *Vehicle* com todos os atributos preenchidos, mas com valores diferenciados caso haja a necessidade de mudança de valores para determinados atributos. No exemplo, inseriu-se duas multas e modificou-se a cor e a cidade do veículo em questão.

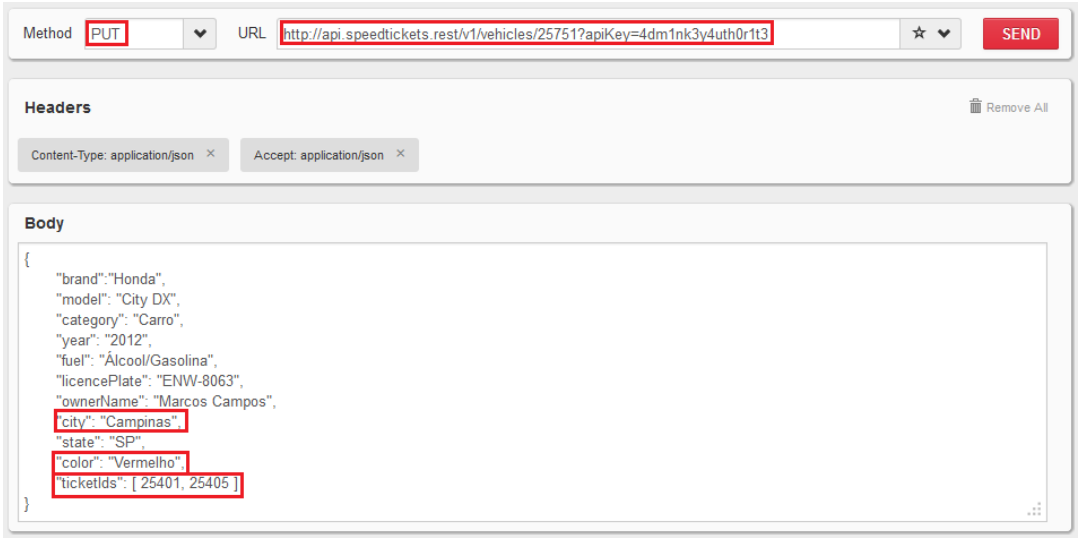


Figura 38 - Execução de serviço de atualização para atualização de veículo cadastrado

Fonte: Próprio autor

Na Figura 38, é analisado que no campo URL foi necessário passar o *id* do registro que se deseja alterar, pois o servidor irá buscar este registro no banco antes de iniciar a operação de alteração. Logo depois tem-se no campo *Body* a apresentação de todo o objeto JSON que constituía o carro cadastrado, porém com as informações de cidade, cor e multas alteradas em relação as cadastradas anteriormente. Para preenchimento do atributo *ticketIds* é necessário conhecer os *ids* das multas cadastradas no recurso "*/tickets*".

Após a execução da operação de atualização pelo servidor são apresentados os resultados ilustrados na Figura 39.

Figura 39 – Resposta do servidor ao evento de atualização

Fonte: Próprio autor

A apresentação de resultados para operações do método *PUT* ocorre da mesma maneira que operações do método *POST*, com a diferença de que o código HTTP apresentado é o 200 (OK), provando que a operação ocorreu com sucesso porém, sem a criação de novos registros pelo servidor. Recomenda-se utilizar o *link* mostrado pelo atributo *resource,* na área *Response Body,*retornado para visualizar os atributos alterados do objeto de *id* 25751.

Feito a aplicação de multas a um veículo através da API, demonstrar-se-á a chamada ao recurso "*/vehicles/{id}/tickets*" com o objetivo de obter detalhes das multas aplicadas a determinado veículo. Para isto, usou-se o veículo cadastrado com id 25751 ilustrado na Figura 38 e foi invocado seu recurso preenchendo os campos URL e *Method* como ilustrado na Figura 40.

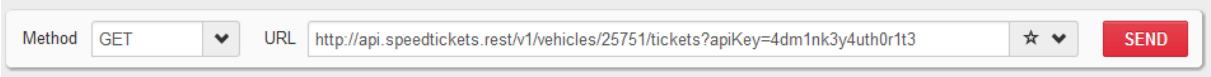


Figura 40 - Chamada a recurso /vehicles/{id}/tickets

Fonte: Próprio autor

Como ilustrado na Figura 38, é observado que o veículo de *id* 25751 possui duas multas aplicadas pelo conteúdo do atributo *ticketIds* (25401 e 25405). Desta forma, tem-se que a resposta do servidor a chamada do serviço ilustrado na Figura

40 é exatamente um objeto JSON contendo em seu interior os detalhes das multas de *id* 25401 e 25405, reforçando a função deste recurso como observado na Figura 41.

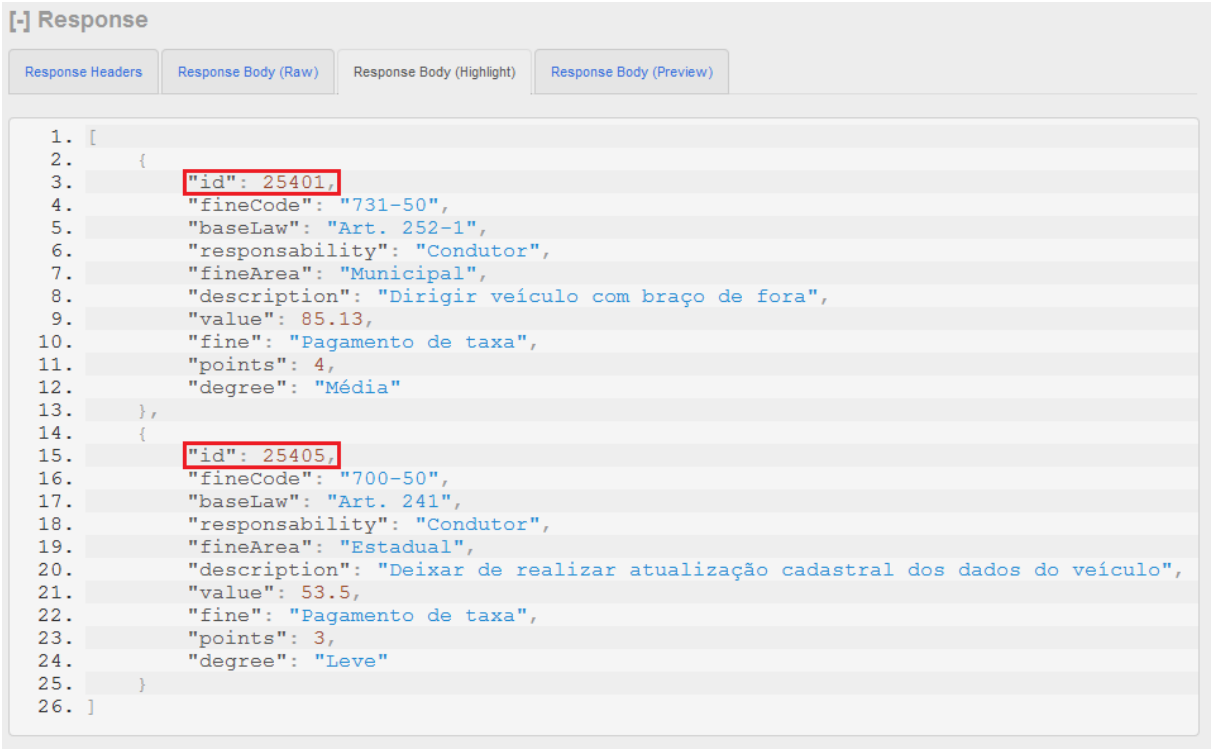


Figura 41 – Demonstração dos detalhes das multas aplicadas para o veiculo cadastrado na figura 38

Fonte: Próprio autor

Por fim, pode-se apagar um registro do servidor ao chamar qualquer serviço, mapeando-o para o método *DELETE*. Isto é feito pelo mesmo procedimento de chamada a outros métodos, onde nesta URI também é necessário conhecer o *id* da entidade ou registro que se deseja excluir, pois o método *DELETE* funciona com recursos específicos. Mostra-se um exemplo de chamada a este serviço na Figura 42.

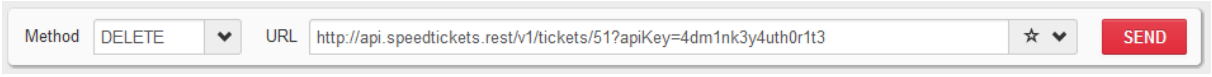


Figura 42 - Exemplo de chamada a serviço de exclusão

Fonte: Próprio autor

No exemplo mostrado pela Figura 42, verifica-se a exclusão da multa com *id* 51 ilustrada na Figura 33. Neste caso o simples prenchimento dos campos URL e

Method já são suficientes para executar a operação. Após a execução do serviço, obtêm-se a resposta do servidor representada na Figura 43.

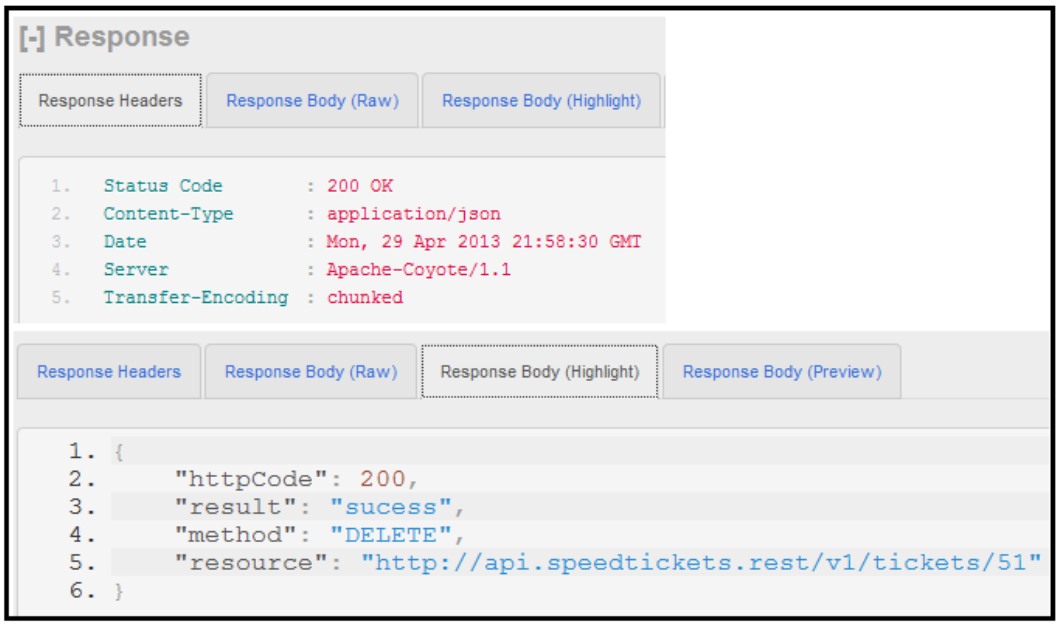


Figura 43 - Resposta do servidor ao evento de exclusão de

Fonte: Próprio autor

Assim como apresentado na explicação da Figura 39, temos uma resposta em mesmo padrão de formato para execução de operações do método *DELETE*, com mostragem de código HTTP para sucesso da operação e uma mensagem em JSON indicando o estado da operação. Porém, a única diferença neste caso é que o *link* mostrado no atributo *resource* tem apenas a função de indicar qual foi o registro deletado.

9.7 VALIDAÇÃO E APRESENTAÇÃO DE MENSAGENS DE ERRO

Demostrar-se-á neste capitulo qual é o comportamento do servidor para chamadas a serviços relacionados aos métodos *POST*, *PUT* e *DELETE* com relação a passagem de informações inválidas tanto no campo URL quanto no campo *Body*. Buscando comprovar qual a função dos códigos HTTP 400 (*Bad Request*) e 404 (*Not Found*) apresentados no capítulo 8.3

O código 400 como explicado no capítulo 5.2 é usado para representar que ocorreu o envio de informações inválidas ao servidor, neste caso, é simulado uma chamada ao recurso "*/tickets*" deixando os campos "*responsability*", "*description*", e "*degree*" em branco como mostrado na Figura 44.

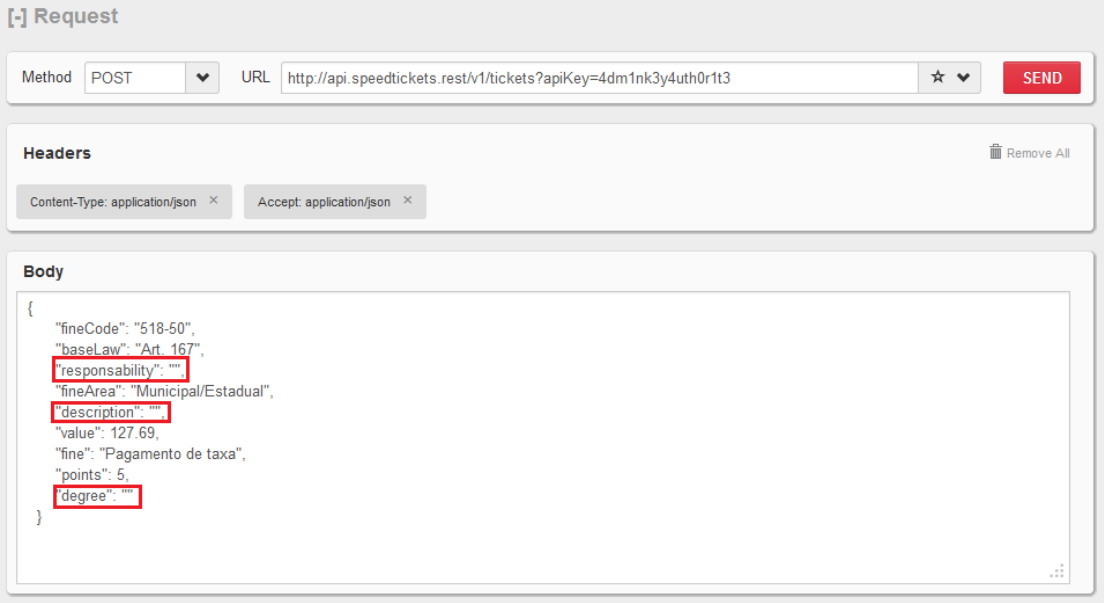


Figura 44 - Envio de mensagem ao servidor com atributos não preenchidos

Fonte: Próprio autor

Como observado, o objeto JSON que representa a entidade *Ticket* possui três atributos não preechidos segundo Figura 44. A API apresentada foi projetada para considerar que estes campos fossem obrigatórios, portanto, o servidor deve invalidar a requisição de cadastro da nova multa retornando o código de erro 400 (*Bad Request*), pois o conteúdo da mensagem de requisição não possui os requisitos necessários para que a operação seja completa, segundo ilustrado na Figura 45.

Figura 45 – Apresentação da resposta do servidor ao envio de mensagem com atributosnão preenchidos

Fonte: Próprio autor

Pode-se analisar na ilustração a apresentação de uma mensagem que possui no setor *Response Headers* a presença do código HTTP indicando o estado da requisição invalidada (400 *Bad Request*), juntamente com o retorno de uma mensagem em formato JSON que diz qual foi o erro ocorrido no setor *Response Body*. Destaca-se o atributo "*result*" com valor "*failure*", apontando o fracasso da operação e o atributo "*message*" que contém detalhes sobre o tipo de erro ocorrido com a mostragem dos campos obrigatórios que não foram preenchidos, reforçando o demonstrado no capítulo 8.3 para um evento de requisição inválida.

Agora, é simulado o evento 404 *Not Found*, em que este ocorre normalmente em chamadas às URIs que possuem mapeamento com os métodos *GET*, *PUT* e *DELETE*, onde manipulamos um registro específico com a indicação do *id* na URI. Nestes casos, as operações relacionadas a estes métodos só ocorrem se os registros com o *id* passado realmente existirem. Caso o *id* passado na URI não tenha relação com um registro existente no sistema, o servidor é obrigado a retornar um erro indicando o não encontro daquele registro para manipulação.

Para ilustrar melhor a ocorrência de um caso deste tipo, chamar-se-á o recurso "*/vehicles*" mapeado para o método *GET* com a função de consultar dados do veiculo de *id* 12345 não existente no sistema e mostrará a resposta do servidor com relação a esta requisição, segundo Figura 46.

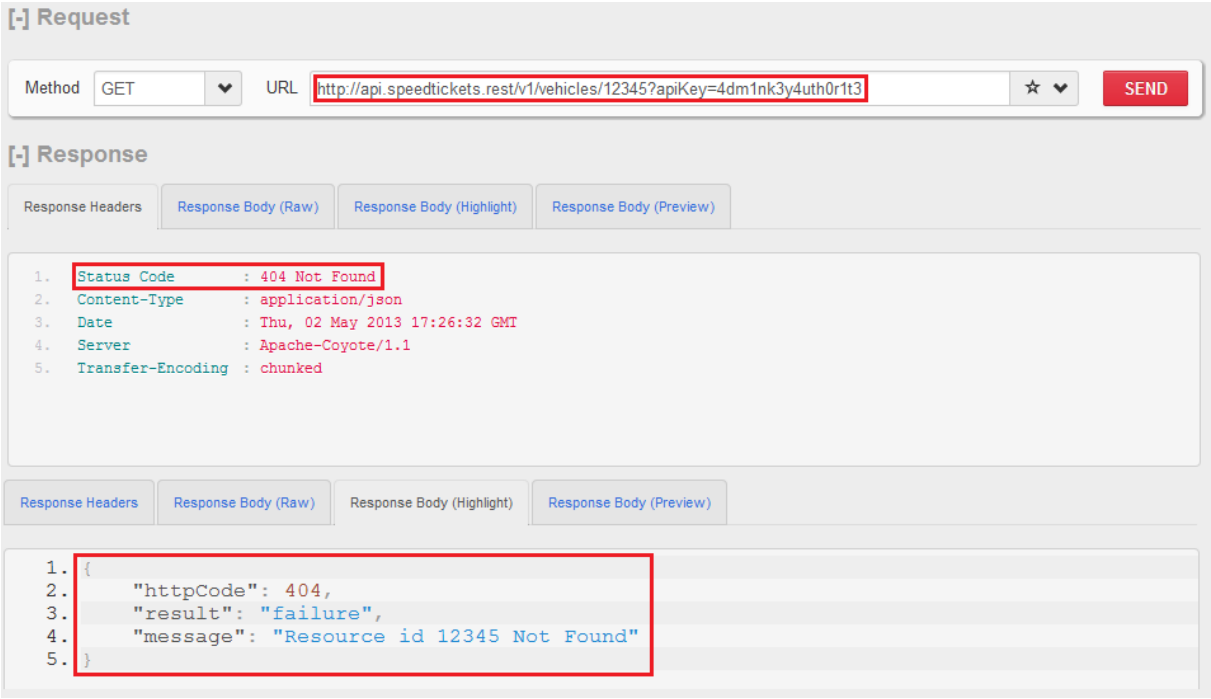


Figura 46 – Chamando serviço de consulta para veículo não cadastrado no sistema (id 12345), juntamente com a resposta do servidor monstrando o erro 404 *Not Found*

Fonte: Próprio autor

Nota-se que o formato da resposta do servidor tem o mesmo formato de outras mensagens de erro com a diferença do código HTTP mostrado no *header* (404 *Not Found*) e dos atributos presentes na mensagem JSON montada no corpo da mensagem de resposta, onde pode-se destacar como principal característica o atributo "*message*" que contém a mensagem que indica que o registro ou recurso de *id* 12345 não foi encontrado.

Para erros de processamento no servidor, onde o *status* da requisição retorna o código 500, as mensagens de erro possuem o mesmo padrão apresentado, sendo que as diferenças estão presentes nos valores dos atributos das mensagens JSON e *headers* vindos do servidor.

Esta abordagem garante uma facilidade no uso e manipulação das mensagens de erro, onde já são esperados uma quantidade de atributos fixa, apenas alterando o conteúdo presente em seu interior e garante integridade na distinção de cada evento de erro ocorrido e sua exibição ao cliente.

10 CONSIDERAÇÕES FINAIS

Com base nas exposições deste trabalho, observa-se a princípio que APIs de qualquer tipo, quando programadas através de segmentos práticos bem definidos, podem possibilitar uma quantidade enorme de capacidades baseadas em desenvolvimento de aplicativos e integração de sistemas como explicado por todo o capítulo 6.

Observa-se como grandes empresas como Twitter, Facebook e Netflix enxergaram nas APIs uma forma de aumentar suas redes de aproximação a novos clientes e busca por inovação, através da exposição de recursos e serviços próprios em uma API bem construída, onde pode-se analisar que APIs têm mais valor dentro do ambiente corporativo.

Portanto, é necessário levar em consideração que a produção de uma API deve ser guiada por um estudo aprofundado sobre processos e estratégias de negócio da empresa interessada, com o objetivo de saber qual será sua funcionalidade, qual público deseja atingir e principalmente quais são as possíveis entidades que serão manipuladas dentro da API construída.

Conclui-se que o fator de sucesso que faz das APIs uma abordagem viável de desenvolvimento de novas soluções está nas ferramentas que sustentam sua arquitetura, como os fundamentos da tecnologia Web que agregam valor as APIs, promovendo a exposição de informações através da Internet. A tecnologia JSON, que facilita a compreensão de como as informações são apresentadas ao usuário final e ao uso das boas práticas baseadas no estilo *Pragmatic REST* que facilitam a maneira como as APIs podem ser utilizadas, como apresentado no capítulo 9.

A partir das regras demonstradas por todo o capítulo 8, pode-se estabelecer novos padrões e abordagens de desenvolvimento que agilizam os processos de construção de uma API e facilitam a compreensão de quem consumir os seus serviços, pois o foco de uma API, independente do propósito, é a construção de aplicativos baseados nestas informações.

Entretanto, o desenvolvimento de sistemas deste tipo exige um conhecimento mais amplo sobre outras tecnologias não abordadas nesta obra, por que uma API

tem a mesma estrutura de um sistema baseado em arquiteturas Web, com a diferença de apresentar suas operações de maneira simples ao usuário final.

Contudo, os conceitos abordados podem abrir espaço para estabelecimento de novas regras e boas práticas de desenvolvimento de outros tipos de serviços para sistemas distribuídos, por termos nas APIs uma nova maneira de disponibilizar informações, transporte e compartilhamento de dados entre diversos dispositivos para consumo de serviços.

11 REFERÊNCIAS BIBLIOGRÁFICAS

ABNT. ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. Citação: NBR-10520/ago -2002. Rio de Janeiro: 2002.

_____. Referências: NBR-6023/ago. 2002. Rio de Janeiro: ABNT, 2002.

ALLAMARAJU, S. ***RESTful Web Services Cookbook.*** 1. ed - Gravenstein Highway North, Sebastopol, CA – O'Reilly Media, Inc. Mar. 2010.

ANDRADE, M. M. de. **Introdução à metodologia do trabalho cientifico**. 9. ed. São Paulo: Atlas, 2009.

ANDREWS. G. ***Paradigms for process interaction in distributed programs.*** ACM Computing Surveys, 23(1), Mar. 1991.

DEITEL, P.M. **Internet & World Wide Web: How to program.** 4. ed. – Upper Saddle River, NJ – Prentice Hall 2008.

FIELDING, R. T. ***Architectural styles and the design of network-based software architectures.*** 1. ed. University of California, Irvine. 2000

GILL, A. C. **Como Elaborar Projetos de Pesquisa.** 174f. 4 ed. - São Paulo, Atlas 2002.

HALL. M, BROWN. L. ***Core Servlet and JavaServer Pages.*** 2 ed. – Prentice Hall and Sun Microsystems.

JACOBSON, D. BRAIL, G. WOODS, D. ***APIs: A strategy guide.*** 1. ed – Gravenstein Highway North, Sebastopol, CA – O'Reilly Media, Inc. Dec. 2011

KUROSE, J. F. ROSS, K. W. **Redes de computadores e a Internet: Uma abordagem top-down.** 3 ed. – São Paulo : Pearson Addison Wesley, 2006 (p.67 – p.70)

MASSÉ, M. ***REST API: Design rulebook*****.** 1. ed – Gravenstein Highway North, Sebastopol, CA – O'Reilly Media, Inc. Set. 2012

NSA. ***Guidelines for implementation of REST*****.** Report # I73-015R-2011. 1 ed. – National Security Agency, 9800 Savage Rd. Suite 6704. Ft. Meade, MD 20755-6704. 2011.

RICHARDSON, L. RUBY, S. ***RESTful Web Services*****.** 1. ed - Gravenstein Highway North, Sebastopol, CA – O'Reilly Media, Inc. Mai. 2007.

308TUBE. ***Java RESTful service tutorial: Creating a REST service - Part 2*.** Disponível em: <http://www.youtube.com/watch?v=4DY46f-LZ0M> Acessado em: 10. Abr. 2013

ALLAVAREDUARTE. **Dispositivos móveis : Estatísticas (2012).** Disponível em: <http://www.avellareduarte.com.br/projeto/dispositivosMoveis/dispositivosmoveis_estatisticas2012.htm> Acessado em: 27. Abr. 2013

AWS. ***Amazon Elastic Compute Cloud (Amazon EC2)*.** Disponível em:<http://aws.amazon.com/pt/ec2/> Acessado em: 25. Fev. 2013

BACILI, K. ***Open APIs.*** Disponível em: <http://www.slideshare.net/sensedia/112012estratgia-de-apis-abertas> Acessado em: 12 Fev. 2012.

BOSWELL, W. ***What is a URL?*** Disponível em: <http://websearch.about.com/od/dailywebsearchtips/qt/dnt0526.htm> Acessado em: 07. Fev. 2013.

DIAZ. D. ***JSON for the masses.*** Disponível em: <http://www.dustindiaz.com/json-for-the-masses/> Acessado em 25 de Fevereiro de 2013

EBAYAPIHISTORY. ***History of APIs: eBay.*** Disponível em: <http://apievangelist.com/history/ebay.php> Acessado em: 25. Fev. 2013

FIRMO, F. **Padrão de implementação REST.** Disponível em: <https://docs.google.com/file/d/0BzQFvOmUvrx6ZHpNQ3k4clNiRGs/edit?pli=1> Acessado em: 22. Mar. 2013

FISCHER. T. ***HTTP Status Code*.** Disponível em: <http://pcsupport.about.com/od/termshm/g/httpstatuscode.htm> Acessado em 13. Fev. 2013

FRANKLIN.C, COUSTAN.D. ***How Operating Systems Work.*** Disponível em: <http://computer.howstuffworks.com/operating-system9.htm> Acessado em: 12 Fev. 2013

FREDRICH, T. **REST API *Tutorial*.** Disponível em: <http://www.restapitutorial.com/lessons/whatisrest.html#> Acessado em: 06Fev 2013

HARDIN. B. ***How OAuth works.*** Disponível em: <http://bretthard.in/2013/01/how-oauth-works/>. Acessado em: 8. Abr. 2013

HISTORYOFAPIS. ***History of APIs.*** Disponível em: <http://apievangelist.com/history/> Acessado em: 25. Fev. 2013

IETF. ***The OAUTH 2.0 autorization framework*.** Disponível em: <http://tools.ietf.org/html/rfc6749.> Acessado em: 8. Abr. 2013

IMASTERS. **Entendendo os Web Services**. Disponível em: <http://imasters.com.br/artigo/4245/web-services/entendendo-os-webservices/> Acessado em: 11. Abr. 2013

IROMIN. ***Public API design factors.*** Disponível em: <http://rominirani.com/2010/07/01/public-api-design-factors/>. Acessado em: 27.Fev. 2013

ITUNES. ***Bus New York City: Enhanced with MTA Bus Time & Official NYC Maps*****.** Disponível em: <https://itunes.apple.com/us/app/bus-new-york-city-enhanced/id424433686?mt=8> Acessado em: 28. Fev. 2013

I-WEB. **Servidores de Aplicações Web.** Disponível em: <www.inf.ufsc.br/~bosco/old_page/downloads/Servidores.ppt.> Acessado em: 15. Abr. 2013

JACCON. **CRUD: Definição de organização de sistema CRUD.** Disponível em: <http://jaccon.com.br/2010/03/crud-definicao-de-organizacao-de-sistema-crud.> Acessado em: 06. Mai. 2013.

LANE, J. ***How does the internet work?*** Disponível em: <http://dev.opera.com/articles/view/3-how-does-the-internet-work/> Acessado em: 08. Fev. 2013.

LEXICALSCOPE. ***How REST APIs are versioned.*** Disponível em: <http://www.lexicalscope.com/blog/2012/03/12/how-are-rest-apis-versioned/> Acessado em: 25 Mar. 2013

MAFRA. D. E. **Como funciona a autenticação Oauth.** Disponível em: <http://www.diogomafra.com.br/2010/09/como-funciona-autenticacao-oauth.html> Acessado em: 8. Abr. 2013

MITRA, R. ***Behind closed doors: The world of private APIs.***Disponível em: <http://www.layer7tech.com/blogs/index.php/behind-closed-doors-the-world-of-private-apis/> Acessado em: 05. Mar. 2013

MITRA, R. ***Considerations of private APIs.*** Disponível em: <http://www.layer7tech.com/blogs/index.php/considerations-for-private-apis/> Acessado em: 05 Mar. 2013

MKYONG.DOM. ***How to read XML file in JAVA: (DOM Parser).*** Disponível em: <http://www.mkyong.com/java/how-to-read-xml-file-in-java-dom-parser/> Acessado em: 28. Fev. 2013.

MKYONG.SAX. ***How to read XML file in Java: (SAX Parser).*** Disponível em: <http://www.mkyong.com/java/how-to-read-xml-file-in-java-sax-parser/> Acessado em: 28. Fev. 2013.

MULLOY, B. ***Web API Design***. Disponível em: <http://info.apigee.com/portals/62317/docs/web%20api.pdf> Acessado em 10. Fev. 2013.

ORACLE. ***Core J2EE paterns: Data access object*****.** Disponível em: <http://www.oracle.com/technetwork/java/dataaccessobject-138824.html>. Acessado em: 06. Mai. 2013

PROGRAMMABLEWEB. ***API Protocols.*** Disponível em:<http://www.programmableweb.com/apis> Acessado em: 27. Fev. 2013

SOBERS, R. ***Introduction to OAuth.*** Disponível em: <http://blog.varonis.com/introduction-to-oauth/> Acessado em: 9. Abr. 2013.

TECHTERMS.COM. **URI.** Disponível em: <http://www.techterms.com/definition/uri> Acessado em: 08. Mai. 2013.

TOTHEPC. ***How to find IP address of website URL.*** Disponível em: <http://www.tothepc.com/archives/find-ip-address-of-website-url/> Acessado em: 08 Fev. 2013.

UFSC. **DSOOII: Web Services.** Disponível em: <http://www.inf.ufsc.br/~frank/INE5612/Seminario2010.1/WebServices.pdf.> Acessado em: 9. Abr. 2013.

VIJAYJOSHI. ***Whats JSON.*** Disponível em: <http://www.vijayjoshi.org/2008/08/20/what-is-json-definition-and-uses-in-javascript/> Acessado em: 19. Fev. 2013.

VOGEL. Lars, ***JPA 2.0 with EclipseLink: Tutorial*****.** Disponível em:<http://www.vogella.com/articles/JavaPersistenceAPI/article.html>. Acessado em: 15. Abr. 2013.

VOGEL. Lars, ***REST with Java (JAX-RS) using Jersey: Tutorial*****.** Disponível em: <http://www.vogella.com/articles/REST/article.html>. Acessado em: 15 de Abr. 2013.

WANSEN. ***Internet Media Type.*** Disponível em: <http://wasen.net/downloads/MIMETypeCollection.pdf> Acessado em: 14 Fev. 2013.

W3C. ***Header field definitions.*** Disponível em: <http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html>. Acessado em: 3. Abr. 2013

WATSON, W. ***REST constraints: Part 5.*** Disponível em: <http://wwatson.me/2011/10/13/rest-constraints-part-5/> Acessado em: 19. Mar. 2013

WEBOPEDIA. ***Web services.*** Disponível em: <http://www.webopedia.com/TERM/W/Web_Services.html.> Acessado em: 11. Abr. 2013

WHATIS. ***Singleton.*** Disponível em: <http://whatis.techtarget.com/definition/singleton> Acessado em: 08. Mai. 2013.

XMLDOM. ***XML DOM: Tree example.*** Disponível em: <http://www.w3schools.com/dom/default.asp> Acessado em: 18. Fev. 2013.