FUNDAÇÃO DE ENSINO “EURÍPIDES SOARES DA ROCHA”
CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA – UNIVEM
CURSO DE DE BACHARELADO EM SISTEMAS DE INFORMAÇÃO

PAULO HENRIQUE CARDOSO DE OLIVEIRA

DESENVOLVIMENTO DE UM GERADOR DE API REST SEGUINDO OS PRINCIPAIS PADRÕES DA ARQUITETURA

MARÍLIA
2014

PAULO HENRIQUE CARDOSO DE OLIVEIRA

DESENVOLVIMENTO DE UM GERADOR DE API REST SEGUINDO OS PRINCIPAIS PADRÕES DA ARQUITETURA

Trabalho de Curso apresentado ao Curso de Bacharelado em Sistemas de Informação da Fundação de Ensino "Eurípides Soares da Rocha", mantenedora do Centro Universitário Eurípides de Marília – UNIVEM, como requisito parcial para obtenção do grau de Bacharel em Sistemas de Informação.

Orientador:
Prof. Ricardo José Sabatine

MARÍLIA
2014

Oliveira, Paulo Henrique Cardoso de

Desenvolvimento de um gerador de API REST seguindo os principais padrões da arquitetura / Paulo Henrique Cardoso de Oliveira; orientador: Ricardo José Sabatine. Marília, SP: [s.n.], 2014.

88 f.

Trabalho de Curso (Graduação em Sistemas de Informação) - Curso de Sistemas de Informação, Fundação de Ensino "Eurípides Soares da Rocha", mantenedora do Centro Universitário Eurípides de Marília –UNIVEM, Marília, 2014.

1. API 2. REST 3. Web Service

CDD: 005.2

CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

TRABALHO DE CONCLUSÃO DE CURSO – AVALIAÇÃO FINAL

Paulo Henrique Cardoso de Oliveira

DESENVOLVIMENTO DE UM GERADOR DE API REST
SEGUINDO OS PRINCIPAIS PADRÕES DA ARQUITETURA

Banca examinadora da monografia apresentada ao Curso de Bacharelado em Sistemas de Informação do UNIVEM/F.E.E.S.R., para obtenção do Título de Bacharel em Sistemas de Informação.

Nota: 10 (Dez)

Orientador: Ricardo José Sabatine

1°. Examinador: Fabio Lucio Meira

2°. Examinador: Paulo Rogerio de Mello Cardoso

Marília, 02 de dezembro de 2014.

Dedico este trabalho a minha esposa
Aline por todo amor, carinho e compreensão.
Aos meus familiares,
amigos e professores pelo incentivo.

AGRADECIMENTOS

À Deus, sempre presente na minha vida em todos os aspectos. Por ter me dado a sabedoria, a coragem e a força necessária para realizar este sonho.

À minha esposa Aline pela paciência e compreensão, por estar do meu lado sempre e me apoiar durante toda essa caminhada.

Aos meus pais Paulo e Joeliza, minhas irmãs Ana Carolina, Denise e Daniela, por todo amor, carinho e dedicação.

Ao meu orientador Ricardo Sabatine, pela paciência, dedicação, confiança e todo auxílio necessário para a conclusão desse trabalho.

Aos professores do curso de Sistemas de Informação, do qual tive o prazer de ser aluno durante esta jornada, que foram de imensurável importância em meu processo de formação acadêmica.

"A mente que se abre a uma nova ideia

jamais volta a seu tamanho original"

Albert Einstein

OLIVEIRA, Paulo Henrique Cardoso. **Desenvolvimento de um Gerador de Api Rest seguindo os principais padrões da arquitetura**. 2014. 88 f. Trabalho de Curso (Bacharelado em Sistemas de Informação) – Centro Universitário Eurípides de Marília, Fundação de Ensino "Eurípides Soares da Rocha", Marília, 2014.

RESUMO

Com o crescimento e adoção de novos estilos arquiteturais como SOA (*Service Oriented Architecture*) e computação em nuvem para o desenvolvimento de sistemas, o conceito de API REST está em evidência. O REST (*Representational State Transfer*) é um estilo de arquitetura de software para sistemas distribuídos hipermídia como a *World Wide Web*. Algumas características do REST são: arquitetura Cliente/Servidor, *Stateless*, Interface Uniforme, *cache*, *Code-on-demand*. Porém, REST é muito amplo e não define nenhuma tecnologia. Uma forma de obter um *Web Service* baseado em REST é denominada de RESTful, este baseia-se no protocolo HTTP para definir um conjunto de operações (*GET*, *POST*, *PUT*, *DELETE*), disponibilizando os recursos por meio da *web* usando URIs. O objetivo deste trabalho é a implementação de uma aplicação para a criação automática e personalizada de um serviço de *web* RESTful utilizando as boas práticas a partir de uma base de dados. Os resultados desse trabalho foram uma aplicação *web* para criar e gerenciar API's e um *Web Service* para acessar as API's via arquitetura REST.

Palavras-chave: Web Services, API, REST, RESTful, HTTP.

OLIVEIRA, Paulo Henrique Cardoso. **Desenvolvimento de um Gerador de Api Rest seguindo os principais padrões da arquitetura**. 2014. 88 f. Trabalho de Curso (Bacharelado em Sistemas de Informação) – Centro Universitário Eurípides de Marília, Fundação de Ensino "Eurípides Soares da Rocha", Marília, 2014.

ABSTRACT

With the growth and adoption of new architectural styles such as SOA (Service Oriented Architecture), and cloud computing for the development of systems, the concept of REST API is in evidence. The REST (Representational State Transfer) is a software architectural style for distributed hypermedia systems like the World Wide Web REST Some of the features are:. Client / Server architectures, Stateless, Interface Uniform, cache, Code-on-demand. However, REST is very broad and does not define any technology. One way to get a REST-based Web Service is called RESTful, this is based on the HTTP protocol to define a set of operations (GET, POST, PUT, DELETE), providing the resources from the web using URIs. The objective of this work is the implementation of an application for automatic and personalized creation of a RESTful web service using good practice from a database. The findings were a web application to create and manage APIs and a Web service to access the API's via REST architecture.

Keywords: Web Services, API, REST, RESTful, HTTP.

LISTA DE ILUSTRAÇÕES

Figura 1: Mensagem SOAP embutida na solicitação HTTP. 28
Figura 2: WSDL document structure. 29
Figura 3: interface orientada a recursos e interface orientada a serviços. 35
Figura 4: HATEOAS. 40
Figura 5: Arquitetura da Aplicação. 43
Figura 6: Diagrama entidade relacional da base de dados da aplicação. 44
Figura 7: Diagrama de atividade - Gerenciar API. 46
Figura 8: Diagrama de sequência, mapeamento da base de dados. 53
Figura 9: Processamento principal de uma requisição na API. 56
Figura 10: Processando uma requisição na API. 59
Figura 11: Processo de leitura de dados na API. 61
Figura 12: Resultado paginado. 63
Figura 13: Processo de inserção de dados na API. 64
Figura 14: Erro na validação. 65
Figura 15: Processo de alteração de dados na API. 65
Figura 16: Processo de remoção de dados na API. 66
Figura 17: Banco de dados CEP. 69
Figura 18: Visão CEP. 69
Figura 19: (A) Login; (B) Cadastro. 70
Figura 20: (A) Alterando configurações do perfil; (B) Token de segurança. 70
Figura 21: Criando um API. 71
Figura 22: Compartilhando uma API. 72
Figura 23: Configurando o banco de dados da API. 72
Figura 24: Gerenciando os recursos da API. 73
Figura 25: Alterando um Recurso. 74
Figura 26: Gerenciando os campos de um recurso. 75
Figura 27: Lista de validadores no campo de um recurso. 75
Figura 28: Adicionando um validador. 76
Figura 29: Opções do campo. 76

Figura 30: Lista de API's. 77
Figura 31: Pesquisando uma API Pública. 78
Figura 32: (A) Documentação Introdução e autenticação; (B) Documentação erros. 78
Figura 33: Documentação dos recursos. 79
Figura 34: Leitura de uma coleção dados na API; 80
Figura 35: Leitura de dados com identificador único. 81
Figura 36: Erro na inserção de um Recurso; 82
Figura 37: Sucesso na inserção de um Recurso. 83
Figura 38: Alteração de dados na API. 84
Figura 39: Exclusão de dados na API. 85

LISTA DE TABELAS

Tabela 1: Métodos HTTP Idempotente e Seguro.....18
Tabela 2: CRUD em REST.....36
Tabela 3: Comparativo das ferramentas 41
Tabela 4: Lista de permissões da API.....47

LISTA DE ABREVIATURAS E SIGLAS

AJAX: Asynchronous Javascript and XML

API: Application Programming Interface

CRUD: Create, Read, Update and Delete

CSS: Cascading Style Sheets

CSV: Comma-separated Values

DNS: Domain Name System

FTP: File Transfer Protocol

HATEOAS: Hypermedia As The Engine Of Application State

HTML: HyperText Markup Language

HTTP: HyperText Transfer Protocol

HTTPS: Hypertext Transfer Protocol Secure

IIOP: Internet Inter-Orb Protocol

IP: Internet Protocol

JSON: JavaScript Object Notation

MVC: Model–view–controller

NASSL: Network Application Service Specification

ORM: Object-relational Mapping

PHP: Hypertext Preprocessor

REST: Representational State Transfer

RFC: Request for Comments

RMI: Remote Method Invocation

SAAS: Software as a Service

SDL: Service Description Language

SGBD: Sistema Gerenciador de Banco de Dados

SMTP: Simple Mail Transfer Protocol

SOA: Service Oriented Architecture

SOAP: Simple Object Access Protocol

SQL: Structured Query Language

SSL: Secure Socket Layer

UDDI: Description Discovery and Integration

UML: Unified Modeling Language

URI: Uniform Resource Identifier

URL: Uniform Resource Locator

XML: eXtensible Markup Language

WADL: Web Application Description Language

WSDL: Web Services Description Language

SUMÁRIO

INTRODUÇÃO 16
Objetivo 17
Estrutura do trabalho 17

1. WEB SERVICES 18
1.1. Conceitos importantes para criação de Web Services 18
1.1.1. O protocolo HTTP 18
1.1.1.1. Idempotência 19
1.1.1.2. Segurança dos métodos 19
1.1.1.3. Métodos de Requisição 20
1.1.1.4. Cabeçalhos HTTP 21
1.1.1.5. Códigos de status 21
1.1.1.6. Passagem de parâmetros 23
1.1.1.7. Formato de Requisição 24
1.1.1.8. Formato de Resposta 25
1.1.2. API 25
1.1.3. Autenticação, Autorização e Segurança 26
1.1.3.1. Authorization Basic 26
1.1.3.2. HTTPS 27
1.2. Tecnologias para a construção de Web Services 27
1.2.1. Arquitetura WS* 27
1.2.1.1. SOAP 28
1.2.1.2. WSDL 29
1.2.1.3. UDDI 30
1.2.2. REST 30
1.3. Considerações finais 31

2. WEB SERVICES BASEADOS EM REST 32
2.1. REST e HTTP 32
2.2. Estilo de arquitetura REST 33
2.2.1. Cliente-Servidor 33
2.2.2. Stateless (Comunicação sem estado) 33
2.2.3. Cache 34
2.2.4. Sistemas em Camadas 34
2.2.5. Interface Uniforme 35
2.2.6. Code-On-Demand 35
2.3. Semânticas de recursos 35
2.4. CRUD em REST 36
2.5. Representação de Dados 37
2.5.1. Media Types 38
2.5.2. XML 38
2.5.3. JSON 39
2.6. HATEOAS 40

2.7. Considerações finais 40

3. FERRAMENTA PARA GERAR API REST 41
3.1. Arquitetura da aplicação....42
3.2. Funcionalidades da aplicação....45
3.3. Tecnologias utilizadas 51
3.3.1. Back-end....51
3.3.2. Front-end 52
3.4. Implementação da aplicação 52
3.4.1. Mapeamento da base de dados 52
3.4.2. Processamento principal de uma requisição na API....55
3.4.3. Processando recursos com identificadores únicos 58
3.4.4. Leitura de dados na API 60
3.4.5. Inserção de dados na API 63
3.4.6. Alteração de dados na API 65
3.4.7. Remoção de dados na API....66
3.5. Considerações finais 67

4. RESULTADOS 68
4.1. Criando e gerenciando uma conta no sistema....69
4.2. Criando e gerenciando uma API 71
4.3. Pesquisando uma API Pública....77
4.4. Visualizando a documentação de uma API 78
4.5. Consumindo os recursos de uma API 79
4.6.Considerações finais 85

CONCLUSÃO....86

REFERÊNCIAS 87

INTRODUÇÃO

A necessidade de integração entre sistemas criados em diferentes linguagens é encontrada cada vez mais nas grandes empresas. Com a crescente popularidade no uso de *smartphones* e uma tendência de crescimento cada vez maior de tecnologias móveis surgiu a necessidade de disponibilizar uma comunicação entre os servidores web e os dispositivos móveis.

A fim de sanar questões como estas, a tecnologia dos Web Services foi criada, permitindo assim, disponibilizar formas de integrar sistemas distintos, modularizar serviços e capacitar a integração e consumo de informações.

Existem muitos protocolos e padrões designados para o desenvolvimento de *Web Services*. Alguns exemplos são: *WS-Notification*, *WS-Security*, WSDL (*Web Services Description Language*) e SOAP (*Simple Object Access Protocol*), esses padrões juntos são chamados de pilha *WS-**, também conhecidos como "*Big Web Services*", eles são chamados assim devido a grande complexidade para implementá-los (RICHARDSON; RUBY, 2007).

Como a maioria desses *Web Services* tradicionais utilizam o protocolo HTTP para a comunicação e troca de mensagens, pode-se adotar um padrão que utiliza 100% de tudo o que esse protocolo oferece, nascendo assim um dos padrões mais utilizado atualmente, o REST (*Representational State Transfer*) foi criado por Roy Fielding (um dos criadores do protocolo HTTP) e foi elaborado com a finalidade de simplificar a criação de *Web Services*. Trabalha usando o protocolo padrão da web, o HTTP, aproveitando todos os recursos que o mesmo oferece, como interface uniforme e o princípio do endereçamento.

Seguindo a arquitetura REST pode-se disponibilizar uma API facilmente. API (*Application Programming Interface*) é um conjunto de padrões de programação que permitem a construção de aplicativos e a sua utilização de maneira não tão evidente para os usuários (CIRIACO, 2009).

Uma das soluções estudadas foi o SOAP. Porém, as mais representativas são as baseadas no estilo arquitetural REST pois possuem uma interface simples e bem definida.

Objetivo

Com a necessidades ampla de integrar sistemas em um espaço de tempo cada vez mais curto, surgiu a necessidade de uma ferramenta que agilizasse esse processo e ao mesmo tempo fosse eficiente, segura, e com uma interface amigável para o usuário.

Sendo assim, o mercado necessita de uma ferramenta que acelere a migração e que seja de baixo custo para ser implementada em sistemas já existentes ou, que poderão ser criados utilizando as facilidades da solução proposta.

O objetivo deste trabalho é apresentar a proposta, implementação e os resultados obtidos de um sistema gerador de API para a criação automática e personalizada de *Web Services* no padrão RESTful a partir da leitura dos metadados de um banco de dados relacional, que foi desenvolvido utilizando as boas práticas do estilo REST.

Estrutura do trabalho

O texto deste trabalho está organizado em cinco capítulos, dispostos a seguir:

O primeiro capítulo, descreve a revisão bibliográfica de todo o processo de criação de um *Web Service,* o funcionamento do HTTP que é considerado o principal protocolo para criação de *Web Services*. Também é apresentado os tipos de *Web Services*.

O segundo capítulo, apresenta as principais características do estilo de arquitetura REST.

O terceiro capítulo, mostra como foi feita a divisão da arquitetura da aplicação, uma explicação das principais funcionalidades contidas no sistema, quais tecnologias foram utilizadas no desenvolvimento e como foi realizada a construção das principais funcionalidades da aplicação.

O quarto capítulo, exibe os resultados do trabalho desenvolvido, juntamente com um exemplo completo de criação, gerenciamento e o consumo dos recursos de uma API criada no sistema gerenciador de API.

O quinto capítulo, descreve a conclusão de todo o trabalho desenvolvido e como o trabalho poderá ser expandido no futuro.

1. *WEB SERVICES*

Um *Web Service* é um sistema de software desenvolvido para suportar interoperabilidade entre máquinas em uma rede, o qual pode apresentar uma interface que o descreve (WSDL, WADL). Outros sistemas interagem com os *Web Services* por meio de mensagens SOAP, normalmente usando HTTP com serialização XML em conjunto com outros padrões relacionados a *Web* (W3C, 2004).

A tecnologia dos *Web Services* possibilita que diferentes aplicações interajam entre si e sistemas desenvolvidos em plataformas distintas se tornem compatíveis. Os *Web Services* são componentes que permitem que aplicações enviem e recebam dados em formatos variados. Cada aplicação pode ter a sua própria "linguagem", que é traduzida para uma linguagem universal, como é o caso dos formatos XML e JSON (MORO; DORNELES; REBONATTO, 2011).

A característica fundamental dos *Web Services* diz respeito à possibilidade de utilização de diferentes formas de transmissão de dados pela rede. Logo, a arquitetura de *Web Services* pode trabalhar com protocolos, tais como HTTP, SMTP, FTP, RMI/IIOP ou protocolos de mensagem proprietários (W3C, 2004).

As primeiras versões das especificações de *Web services* ofereciam apenas o HTTP como meio de transporte de dados (mensagens SOAP XML) e comunicação entre clientes e servidores, sendo ainda o protocolo mais utilizado atualmente (MORO; DORNELES; REBONATTO, 2011).

1.1. Conceitos importantes para criação de *Web Services*

Para se criar um *Web Service* de qualidade é necessário antes conhecer alguns conceitos importantes, os quais serão apresentados a seguir.

1.1.1. O protocolo HTTP

O HTTP é um protocolo de comunicação para distribuição de objetos de hipermídia referenciados por uma URL. Este é o principal protocolo da Internet. A função do HTTP é

bastante simples: realizar requisições e receber respostas entre um cliente e servidor (EMER, 2014).

A transmissão de informações entre um emissor e um receptor caracteriza uma comunicação. E da mesma forma que uma comunicação interpessoal, a cordialidade é essencial. O protocolo HTTP estabelece um cabeçalho para suas requisições e respostas. O cabeçalho de uma mensagem são informações complementares que são de uso do cliente e servidor. Através das informações passadas pelo cabeçalho, que são de uso exclusivo do servidor e cliente, é possível informar em uma requisição a preferência de idiomas, as codificações e os formatos de conteúdo para uma resposta. O cabeçalho da resposta, por sua vez, contém o código de status, codificação, formato e tempo de expiração da conteúdo (EMER, 2014).

1.1.1.1. Idempotência

A idempotência de um método é relativa às modificações que são realizadas em informações do lado do servidor. Trata-se do efeito que uma mesma requisição tem do lado do servidor, se a mesma requisição, realizada múltiplas vezes, provoca alterações no lado do servidor como se fosse uma única, então esta é considerada idempotente (SAUDATE, 2013).

1.1.1.2. Segurança dos métodos

Os métodos são considerados seguros se não provocarem quaisquer alterações nos dados contidos. Em relação a estas duas características, a seguinte distribuição dos métodos HTTP é feita conforme a Tabela 1:

Tabela 1: Métodos HTTP Idempotente e Seguro.

Método HTTP	Idempotente	Seguro
GET	Sim	Sim
POST	Não	Não
PUT	Sim	Não
DELETE	Sim	Não

Fonte: SAUDATE (2013).

1.1.1.3. Métodos de Requisição

O protocolo HTTP foi criado no projeto *World Wide Web* e designado para operar essencialmente com documentos de hipertexto. Na época, havia somente o método *GET* para requisitar as páginas, mas isto foi antes de qualquer especificação do protocolo (EMER, 2014).

O HTTP 1.1, que constitui a especificação consolidada mais recente, define oficialmente nove métodos que são informados no cabeçalho da requisição, cada um possui uma finalidade diferente (EMER, 2014).

Os principais métodos são:

- ***GET***

Solicita um objeto para o servidor. Objetos são entidades que o servidor conhece e que o cliente esteja interessado em manipular..

- ***POST***

Escreve um objeto no servidor sem respeitar a propriedade da idempotência. Requisições deste tipo, quando repetidas, podem gerar resultados diferentes no servidor. O uso comum deste método é para a criação de objetos no servidor.

- ***PUT***

Escreve um objeto no servidor de maneira a respeitar a propriedade da idempotência. Em linhas gerais, este tipo de requisição pode ser realizada mais de uma vez e o resultado no servidor será o mesmo. Geralmente estas requisições carregam consigo um identificador único para o objeto e portanto são mais usadas para alterá-lo.

- ***DELETE***

Método utilizado para remover um recurso no servidor.

- ***HEAD***

Igual ao método *GET*, porém ao invés de retornar o *response body* retorna apenas o *response code* e os cabeçalhos associados à requisição.

- ***OPTIONS***

O método é utilizado para saber quais métodos podem ser utilizados em determinado recurso.

1.1.1.4. Cabeçalhos HTTP

Os cabeçalhos, em HTTP, são utilizados para trafegar todo o tipo de metainformação a respeito das requisições. Vários destes cabeçalhos são padronizados; no entanto, eles são facilmente extensíveis para comportar qualquer particularidade que uma aplicação possa requerer nesse sentido (SAUDATE, 2013).

Alguns dos cabeçalhos mais utilizados são:

- ***Host*:** mostra qual foi o DNS utilizado para chegar a este servidor;
- ***User-Agent:*** fornece informações sobre o meio utilizado para acessar este endereço;
- ***Accept*:** realiza negociação com o servidor a respeito do conteúdo aceito;
- ***Accept-Language*:** negocia com o servidor qual o idioma a ser utilizado na resposta;
- ***Accept-Encoding*:** negocia com o servidor qual a codificação a ser utilizada na resposta;
- ***Connection*:** ajusta o tipo de conexão com o servidor (persistente ou não);
- ***Content-Type*:** indica o tipo e subtipo do conteúdo da mensagem.

1.1.1.5. Códigos de status

Saudate (2013, p. 66) diz que toda requisição que é enviada para o servidor retorna um código de status. Esses códigos são divididos em cinco famílias: 1xx, 2xx, 3xx, 4xx e 5xx, sendo:

- **1xx:** Informacionais
- **2xx:** Códigos de sucesso
- 3**xx:** Códigos de redirecionamento
- 4**xx:** Erros causados pelo cliente
- 5**xx:** Erros originados no servidor

Alguns dos códigos mais importantes e mais utilizados são:

- **200 – *OK***

Indica que a operação indicada teve sucesso.

- **201 – *Created***

Indica que o recurso desejado foi criado com sucesso. Deve retornar um cabeçalho *Location*, que deve conter a URL onde o recurso recém-criado está disponível.

- **204 – *No Content***

Usualmente enviado em resposta a uma requisição *PUT*, *POST* ou *DELETE*, onde o servidor pode recusar-se a enviar conteúdo.

- **301 – *Moved Permanently***

Significa que o recurso solicitado foi movido permanentemente. Uma resposta com o código 301 deve conter um cabeçalho *Location* com a URL completa (ou seja, com descrição de protocolo e servidor) de onde o recurso está atualmente.

- **304 – *Not Modified***

É utilizado, principalmente, em requisições *GET* condicionais, quando o cliente deseja ver a resposta apenas se ela tiver sido alterada em relação a uma requisição anterior.

- **307 – *Temporary Redirect***

Similar ao 301, mas indica que o redirecionamento é temporário, não permanente.

- **400 – *Bad Request***

É uma resposta genérica para qualquer tipo de erro de processamento cuja responsabilidade é do cliente do serviço.

- **401 – *Unauthorized***

Utilizado quando o cliente está tentando realizar uma operação sem ter fornecido dados de autenticação ou a autenticação fornecida for inválida.

- **403 – *Forbidden***

Utilizado quando o cliente está tentando realizar uma operação sem ter a devida autorização.

- **404 – *Not Found***

Utilizado quando o recurso solicitado não existe.

- **405 – *Method Not Allowed***

Utilizado quando o método HTTP informado não é suportado pela URL. Deve incluir um cabeçalho *Allow* na resposta, contendo a listagem dos métodos suportados separados por “,".

- **409 – *Conflict***

Utilizado quando há conflitos entre dois recursos. Comumente utilizado em resposta a criações de conteúdos que tenham restrições de dados únicos – por exemplo, criação de um usuário no sistema utilizando um login já existente. Se for causado pela existência de outro recurso (como no caso citado), a resposta deve conter um cabeçalho *Location*, explicitando a localização do recurso que é a fonte do conflito.

- **415 – *Unsupported Media Type***

Utilizado em resposta a clientes que solicitam um tipo de dados que não é suportado, por exemplo, solicitar JSON quando o único formato de dados suportado é XML.

- **500 – *Internal Server Error***

É uma resposta de erro genérica, utilizada quando nenhuma outra se aplica.

- **503 – *Service Unavailable***

Indica que o servidor está atendendo requisições, mas o serviço em questão não está funcionando corretamente. Pode incluir um cabeçalho *Retry-After*, dizendo ao cliente quando ele deveria tentar submeter a requisição novamente.

1.1.1.6. Passagem de parâmetros

Os métodos HTTP suportam parâmetros sob duas formas: os chamados *query parameters* e *body parameters*.

Segundo Saudate (2013, p. 52) os *query parameters* são inseridos a partir do sinal de interrogação. Este sinal indica para o protocolo HTTP que, dali em diante, serão utilizados *query parameters*. Esses são inseridos com formato <chave>=<valor>. Caso mais de um parâmetro seja necessário, os pares são separados com um &. Por exemplo, a requisição para o serviço do Google poderia, ser enviada da seguinte maneira:

GET /?q=HTTP&oq=HTTP HTTP/1.1

Host: www.google.com.br”

Quando há a necessidade de fornecer dados complexos, é possível fazê-lo pelo corpo da requisição. Algo como:

POST /clientes HTTP/1.1

Host: *localhost*:8080

Content-Type: text/xml

Content-Length: 93

<cliente>

<nome>Paulo</nome>

<dataNascimento>2000-01-01</dataNascimento>

</cliente>

1.1.1.7. Formato de Requisição

O protocolo HTTP foi desenvolvido de maneira a ser o mais flexível possível para comportar diversas necessidades diferentes. Em linhas gerais, este protocolo segue o seguinte formato de requisições:

<método> <URL> HTTP/<versão>

<Cabeçalhos - Sempre vários, um em cada linha>

<corpo da requisição>

Por exemplo, para realizar uma requisição ao site wikipedia.org/wiki/REST, pode ser enviado para o servidor algo semelhante ao seguinte:

GET /wiki/REST HTTP/1.1

Host: en.wikipedia.org

Accept: text/html

Accept-Encoding: gzip

Com isso, o método *GET* foi utilizado para solicitar o conteúdo da URL /wiki/REST. Além disso, o protocolo HTTP versão 1.1 foi utilizado. Note que o *host*, ou seja, o servidor responsável por fornecer os dados, é passado em um cabeçalho à parte. No caso, o *host* escolhido foi en.wikipedia.org. Além disso, outros dois cabeçalhos, *Accept* e *Accept-Encoding*, foram fornecidos. O primeiro tem a função de informar ao servidor qual o tipo de

informação será aceita, no caso *text/html*, o segundo informa ao servidor que tipo de codificação do conteúdo será aceita na resposta (SAUDATE, 2013).

1.1.1.8. Formato de Resposta

A resposta para as requisições segue o seguinte formato geral:

HTTP/<versão> <código de status> <descrição do código>

<cabeçalhos>

<conteúdo da resposta>

Por exemplo, a resposta para a requisição ao site wikipedia.org/wiki/REST foi semelhante à seguinte:

HTTP/1.1 200 OK

Content-Type: text/html; charset=UTF-8

Content-Length: 20209

Content-Encoding: gzip

<!DOCTYPE html>

<html lang="en" dir="ltr" class="client-nojs">

<head>

<title>Representational state transfer - Wikipedia, the free encyclopedia</title>...

Note que o código 200 indicou que a requisição foi bem-sucedida, e o cabeçalho *Content-Length* trouxe o tamanho da resposta. Além disso, o cabeçalho *Content-Type* indica que a resposta é, de fato, HTML.

1.1.2. API

Na grande maioria das vezes uma API se comunica com diversos outros códigos interligando diversas funções em um aplicativo. Um exemplo de uma API muito utilizada é a

API dos sistemas operacionais que possuem diversos métodos e se comunicam com diversos processos do sistema operacional.

Já no contexto de desenvolvimento *web*, uma API é um conjunto definido de mensagens de requisição e resposta HTTP, geralmente expressado nos formatos XML ou JSON. Ainda que esse termo seja um sinônimo para *Web Service*, a chamada *Web* 2.0 está aos poucos depreciando o modelo de serviços SOAP para a técnica REST (MEDEIROS, 2012).

1.1.3. Autenticação, Autorização e Segurança

Nas aplicações web que utilizam segurança, geralmente é necessário fornecer mecanismos para que o cliente possa se identificar. A partir desta identificação, o servidor checa se as credenciais fornecidas são válidas e quais direitos estas credenciais dão a ele. Estes procedimentos recebem o nome de autenticação (ou seja, o cliente fornecendo credenciais válidas) e autorização (o servidor checando quais direitos o cliente tem) (SAUDATE, 2013).

1.1.3.1. Authorization Basic

O protocolo HTTP oferece um mecanismo para autenticação: o *basic*. A autenticação *basic* é realmente muito simples; consiste em passar o cabeçalho HTTP *Authorization* obedecendo ao seguinte algoritmo:

Basic Base64({#usuário}:{#senha}).

Considerando um usuário paulo e senha henrique, o cabeçalho passaria a ter o seguinte valor:

Authorization: *Basic* cGF1bG86aGVucmlxdWU=.

Note que este sistema não apresenta muita segurança, caso um atacante intercepte esta requisição, o algoritmo *Base64* é facilmente reversível, levando aos dados originais. Desta forma, a maior parte dos sistemas que utilizam autenticação *basic*, utilizam também HTTPS, para que o atacante não descubra o usuário e senha que estão sendo trafegados (SAUDATE, 2013).

1.1.3.2. HTTPS

O protocolo HTTP, disponibiliza um mecanismo padrão de proteção contra o problema da interceptação do dados,.

A solução mais usada para trafegar as informações de maneira segura é o uso do HTTPS (*HyperText Transfer Protocol over Secure Sockets Layer*), e baseia-se na aplicação de uma camada de segurança com SSL (*Secure Socket Layer*) sobre o HTTP. Esta camada utiliza certificados digitais para encriptar as comunicações e garantir a autenticidade do servidor e, opcionalmente, do cliente (SAUDATE, 2013).

1.2. Tecnologias para a construção de *Web Services*

A seguir será apresentado os principais padrões para a construção de *Web Services*.

1.2.1. Arquitetura WS*

No final da década de 90, muitas pessoas estavam desenvolvendo sistemas baseados em HTTP e XML. Cada um com sua própria maneira de implementar segurança, confiabilidade, gerenciamento de transações. Com isso, muitos problemas começaram a surgir: incompatibilidades entre sistemas, dificuldade na manutenção e em compensação, a necessidade de se criar uma maneira comum de realizar estas tarefas. Estes fatos impulsionaram o surgimento dos padrões WS-*. Hoje mantidos pela W3C e OASIS (MORO; DORNELES; REBONATTO, 2011).

Atualmente, a arquitetura WS-* é composta por mais de 20 especificações, sendo as especificações base deste conjunto a SOAP, WSDL e UDDI. Além dos mencionados, existem também os padrões *WS-Notification, WS-Addressing, WS-Transfer, WS-Policy, WS- Security, WS-Trust, WS-ReliableMessaging, WS-Transaction, WS-AtomicTransaction, WS-I Basic Profile* entre outros. Uma das principais reclamações em relação a esse formato diz respeito ao grande número de especificações que o torna complexo e burocrático, difícil de ser dominado por uma pessoa só. A seguir serão explicadas as 3 principais especificações (MORO; DORNELES; REBONATTO, 2011).

1.2.1.1. SOAP

No padrão WS-*, as mensagens trocadas entre servidor e cliente devem ser armazenadas em envelopes SOAP. Este protocolo de comunicação dita um formato de envio de mensagens entre aplicações, o qual é descentralizado e distribuído, ou seja, qualquer plataforma de comunicação pode ser utilizada, seja ela proprietária ou não (MORO; DORNELES; REBONATTO, 2011).

Este protocolo define uma estrutura para interoperabilidade entre sistemas através de mensagens em formato XML. Um envelope SOAP é um documento XML, o formato deste documento é definido por um XML *schema*, que, por sua vez, faz uso de XML *namespaces* para garantir extensibilidade. Um documento SOAP é basicamente constituído de um elemento raiz envelope, o qual identifica o arquivo XML como sendo uma mensagem SOAP. Um elemento *Header* que possui dados de cabeçalho. Um elemento *Body* responsável por armazenar informações de chamadas e respostas e um elemento *Fault* utilizado para manter informações e status de possíveis erros. A Figura 1 exemplifica a estrutura de um documento SOAP (MORO; DORNELES; REBONATTO, 2011).

Figura 1: Mensagem SOAP embutida na solicitação HTTP.

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
   <SOAP-ENV:Header>
       <t:Transaction
           xmlns:t="some-URI"
           SOAP-ENV:mustUnderstand="1">
               5
       </t:Transaction>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body>
       <m:GetLastTradePrice xmlns:m="Some-URI">
           <symbol>DEF</symbol>
       </m:GetLastTradePrice>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Fonte: W3C (2000).

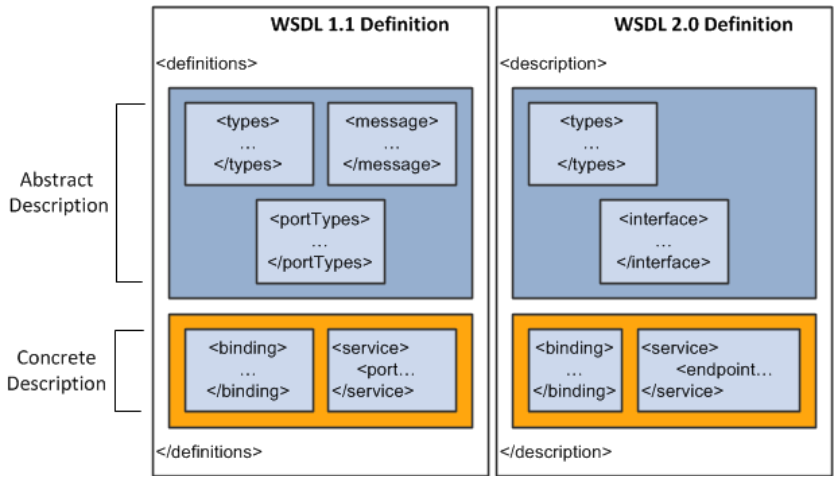
1.2.1.2. WSDL

Quando um *Web Service* é criado, geralmente se espera que outras pessoas o utilizem. Para que isso seja possível, o cliente do serviço deve conhecer quais informações devem ser enviadas ao serviço, quais informações são retornadas por ele e onde encontrá-lo. Ter um formato padronizado para essas informações torna o consumo de um serviço muito mais simples. Foi para este fim que a WSDL (*Web Service Description Language*) foi criada.

A linguagem WSDL foi criada no ano 2000, originada da combinação de duas outras linguagens: NASSL (*Network Application Service Specification*) da IBM e SDL (*Service Description Language*) da Microsoft. No ano seguinte a versão 1.1 foi enviada como nota para W3C e em 2007 tornou-se recomendação em sua versão 2.0. A linguagem WSDL é considerada um vocabulário XML utilizado para descrever e localizar *Web Services*. Permite que desenvolvedores de serviços disponibilizem informações importantes para a utilização dos mesmos. É altamente adaptável e extensível, o que permite a descrição de serviços que se comunicam em diferentes meios, tais como SOAP, RMI/IIOP (MORO; DORNELES; REBONATTO, 2011).

O documento WSDL é dividido logicamente em duas partes, as descrições abstratas e as concretas, ilustrado pela Figura 2. A parte abstrata descreve a capacidade do *Web Service*, como mensagens recebidas e enviadas pelo mesmo. Na parte concreta são descritos os elementos utilizados para vincular fisicamente o cliente ao serviço (MORO; DORNELES; REBONATTO, 2011).

Figura 2: WSDL document structure.



Fonte: Understanding Providing WSDL Documents (2014).

Na versão 2.0 da especificação WSDL, algumas mudanças foram realizadas em relação à versão 1.1. A principal mudança foi a retirada do elemento *message*. Sem este elemento, a referência ao tipo de dado enviado ou recebido em uma mensagem fica no elemento *operation*, o qual referencia diretamente um tipo de dado no elemento *type*. Outras mudanças na estrutura da linguagem dizem respeito aos novos nomes dos elementos: *definitions* para *description*; *portType* para *interface* e; *port* para *endpoint* (MORO; DORNELES; REBONATTO, 2011).

1.2.1.3. UDDI

UDDI (*Description Discovery and Integration*) é um protocolo que disponibiliza métodos e padrões para publicação e localização de informações sobre *Web Services*, funcionando como um repositório com informações úteis para a utilização destes serviços. Em um cenário onde uma organização utiliza somente seus próprios *Web Services* para a execução de diferentes processos, ou compartilha serviços entre poucas companhias, o gerenciamento destes não envolve muita complexidade. Em alguns anos, a gama de parceiros de uma organização pode crescer e mais serviços em comum serão utilizados pelas mesmas. No processo de uma venda via comércio eletrônico, onde empresas utilizam não somente os seus serviços, mas o de outras para realizar uma transação bancária ou a atualização de um estoque, a necessidade de um registro de serviços universal, padronizado, se torna fundamental para a busca e cadastro dos mesmos (MORO; DORNELES; REBONATTO, 2011).

1.2.2. REST

REST (*Representational State Transfer*) é um estilo de arquitetura para sistemas de hipermídia distribuídos. Esse termo foi utilizado pela primeira vez por Roy Fielding (um dos criadores do protocolo HTTP), em sua tese de doutorado publicada no ano 2000. Assim, é perceptível que o protocolo REST é guiado pelo que seriam as boas práticas de uso de HTTP (SAUDATE, 2013, p. 26):

- Uso adequado dos métodos HTTP;
- Uso adequado de URLs;
- Uso de códigos de status padronizados para representação de sucessos ou falhas;
- Uso adequado de cabeçalhos HTTP;
- Interligações entre vários recursos diferentes.

1.3. Considerações finais

Neste capítulo foi demostrado o funcionamento do HTTP que é considerado o principal protocolo para criação de *Web Services*. Também foi apresentado os tipos de *Web Services*, entre os tipos apresentados o escolhido para o desenvolvimento deste trabalho foi o REST, já que REST é baseado nos conceitos do protocolo HTTP.

No entanto, REST não é tão fácil quanto simplesmente utilizar HTTP, existem regras que devem ser seguidas para se realizar o uso efetivo deste protocolo. A intenção do próximo capítulo é apresentar quais são estes conceitos e as técnicas que foram utilizadas para desenvolvimento da solução proposta no capítulo 3.

2. *WEB SERVICES* BASEADOS EM REST

Como dito anteriormente, REST é baseado nos conceitos do protocolo HTTP. Além disso, este protocolo é a base da *web*, sendo que a própria navegação nesta pode ser encarada como um uso de REST.

O termo é geralmente usado para descrever qualquer interface que transmita dados de um domínio específico sobre HTTP sem uma camada adicional de mensagem como SOAP ou *session tracking* via *cookies* HTTP. Estes dois conceitos podem entrar tanto em conflito como em sobreposição. É possível desenvolver um sistema de *software* de acordo com as restrições impostas pelo estilo arquitetural REST sem usar HTTP e sem interagir com a *Web*. Também se torna possível projetar interfaces HTTP + XML que não condizem com os princípios REST (FIELDING, 2000).

2.1. REST e HTTP

Segundo Saudate (2013, p. 95):

> REST é fortemente ligado ao HTTP, a ponto de não poder ser executado com sucesso em outros protocolos.

O protocolo, na realidade, não é uma restrição em REST, na teoria. Na prática, o HTTP é o único protocolo 100% compatível conhecido. Vale destacar que o HTTPS não é uma variação do HTTP, mas apenas a adição de uma camada extra de segurança, o que mantém o HTTPS na mesma categoria que o HTTP, assim como qualquer outro protocolo que seja utilizado sobre o HTTP, como *SPDY* (SAUDATE, 2013).

As possíveis causas para esta "preferência" podem ser apontadas: o autor do HTTP também é o autor de REST e, além disso, o protocolo HTTP é um dos mais utilizados no mundo (sendo que a *web*, de maneira geral, é fornecida por este protocolo). Estes fatos promoveriam uma aceitação grande e rápida absorção de REST pela comunidade de desenvolvedores (SAUDATE, 2013).

Ele é baseado em diversos princípios que fizeram com que a própria web fosse um sucesso. Estes princípios são:

- URLs bem definidas para recursos;
- Utilização dos métodos HTTP de acordo com seus propósitos;
- Utilização de *media types* efetiva;
- Utilização de *headers* HTTP de maneira efetiva;
- Utilização de códigos de *status* HTTP;
- Utilização de Hipermídia como motor de estado da aplicação.

2.2. Estilo de arquitetura REST

Um estilo de arquitetura é um conjunto coordenado de restrições arquiteturais que restringe as funções/características dos elementos da arquitetura e permite relações entre esses elementos dentro de qualquer arquitetura que está de acordo com esse estilo (FIELDING, 2000).

2.2.1. Cliente-Servidor

O estilo Cliente/Servidor visa separar as responsabilidades e é muito utilizado em aplicações baseadas em rede, visa principalmente separar a interface do usuário do armazenamento de dados, facilitando assim a responsabilidade do servidor. No estilo cliente/ servidor têm-se um Servidor responsável por oferecer um conjunto de serviços que serão invocados por aplicações clientes, o servidor disponibiliza seus recursos para ser consumido por algum cliente independente da plataforma ou da linguagem escrita (FIELDING, 2000).

2.2.2. *Stateless* (Comunicação sem estado)

Existe uma restrição às interações entre o cliente e o servidor, a comunicação será *client-stateless-server* e isto significa que para cada requisição será enviada toda informação necessária para o entendimento da requisição e não poderá re-utilizar nenhum contexto armazenado no servidor. Toda requisição deve ser auto-suficiente e deve manter o estado da sessão no cliente (RONDON, 2010).

Este conceito trás alguns benefícios, tais como:

- **Visibilidade:**

O pacote de requisição de dados contém todas as informações necessárias para responder a solicitação, diminuindo o trabalho do servidor.

- **Confiabilidade:**

É melhorada porque facilita a tarefa de recuperação de falhas.

- **Escalabilidade:**

Não há necessidade de manter o estado das solicitações, permitindo que o servidor se livre dos recursos alocados rapidamente e ainda simplificando a implementação.

2.2.3. Cache

Outro elemento muito importante na arquitetura REST e o *cache* no cliente. Para melhorar o desempenho de rede deve-se utilizar este mecanismo para eliminar parcialmente as interações, melhorando a eficiência, escalabilidade e desempenho pelo usuário (RONDON, 2010).

A vantagem de se utilizar cache é a eliminação parcial ou total de algumas interações entre cliente e servidor, o que melhora a eficiência (menos tráfego de rede), escalabilidade (menos processamento) e performance , já que o servidor fica menos carregado.

O armazenamento dos resultados em *cache* é baseado em cabeçalhos padronizados do protocolo HTTP, o cabeçalho mais utilizado é o *Cache-Control*.

2.2.4. Sistemas em Camadas

A utilização de camadas tem como finalidade melhorar a escalabilidade do processo; isto significa que é permitido uma arquitetura composta por camadas hierárquicas por condicionar o comportamento de componentes de modo que cada componente não pode "ver" além da camada imediata com as quais estão interagindo. A implementação de *cache* por

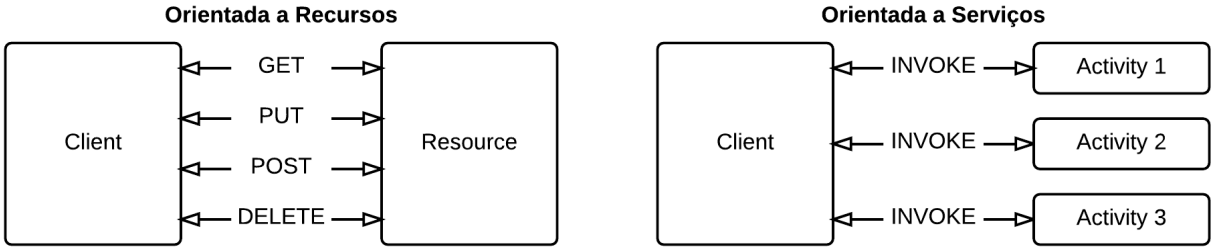
domínio é uma estratégia para quem for aproveitar desta característica do REST. (RONDON, 2010).

2.2.5. Interface Uniforme

A característica central de REST é sua ênfase em uma interface uniforme entre os componentes (cliente, servidor) . É um estilo que define quatro princípios fundamentais como a identificação de recursos, manipulação de recursos através de representações, mensagens auto descritivas e recursos hipermídia (FIELDING, 2000).

A Figura 3 ilustra as interfaces dos *Web Services* nas abordagens RESTful (orientada a recursos) e WS-* (orientada a serviços).

Figura 3: interface orientada a recursos e interface orientada a serviços.



Fonte: MORO; DORNELES; REBONATTO (2011).

2.2.6. Code-On-Demand

Código sob demanda é um paradigma de sistemas distribuídos que define a possibilidade e as técnicas de mover um código existente no servidor para a execução no cliente, um exemplo disso são os códigos *JavaScript*. Porém, o seu uso é opcional (FIELDING, 2000).

2.3. Semânticas de recursos

Serviços REST são baseados nos chamados recursos, que são entidades bem definidas em sistemas. Um recurso é algo que pode ser encontrado em um computador e representado por uma sequência de bits, que são entidades bem definidas em sistemas, que

possuem identificadores (URI's) e endereços (URL's) próprios, por exemplo, um documento, uma imagem, um registro de uma tabela ou o resultado da execução de um algoritmo (RICHARDSON, 2007).

Os recursos são representados por URI's. Dependendo do método utilizado para invocá-lo e dos dados na requisição HTTP, este URI terá um funcionamento diferenciado.

Para invocar um serviço REST identificado por URI's, utiliza-se o formato genérico de mensagem HTTP. Caso seja feita uma solicitação a um serviço de exclusão (*DELETE*) ou listagem de produto (*GET*), apenas a requisição para o identificador do serviço é necessário, pois o método HTTP já indica qual a operação a ser realizada e o recurso solicitado (MORO; DORNELES; REBONATTO, 2011).

2.4. CRUD em REST

Para interagir com as URL's, os métodos HTTP são utilizados. A regra de ouro para esta interação é que as URL's são substantivos, e métodos HTTP são verbos. Isto quer dizer que os métodos HTTP são os responsáveis por provocar alterações nos recursos identificados pelas URL's (SAUDATE, 2013).

Estas modificações são padronizadas, de maneira que:

- ***GET:*** recupera os dados identificados pela URL;
- ***POST:*** cria um novo recurso;
- ***PUT:*** atualiza um recurso;
- **DELETE:** apaga um recurso.

Sistemas que seguem esse padrão também são referenciados como "RESTful".

Os quatro métodos principais podem ser diretamente relacionados a operações de bancos de dados. Assim, para recuperar o usuário número 1 do banco de dados, basta utilizar o método *GET* em conjunto com a URL /*user*/1; para criar um novo usuário, basta utilizar o método *POST* sobre a URL /*user* (o identificador será criado pelo banco de dados); para atualizar este usuário, utilize o método *PUT* sobre a URL /*user*/1 e, finalmente, para apagar o usuário, utilize o método *DELETE* sobre a URL /*user*/1. A Tabela 2 ajuda a entender esse conceito.

Tabela 2: CRUD em REST.

CRUD	SQL	HTTP	URI	Definição
Create	*Insert*	*POST*	*/user*	Cria um recurso, a URL é definida pelo servidor
Retrieve	*Select*	*GET*	*/user*	Recupera a representação de muitos recursos
Retrieve	*Select*	*GET*	*/user/{id}*	Recupera a representação de um único recurso
Update	*Update*	*PUT*	*/user/{id}*	Sobrescreve/altera um recurso
Delete	*Delete*	*DELETE*	*/user/{id}*	Remove um recurso

Fonte: o autor.

2.5. Representação de Dados

Quando uma aplicação é dividida em recursos, conforme pregado pelo estilo REST, as necessidades do usuário de um serviço podem ser sanadas de uma forma mais dinâmica. O usuário pode construir uma URI e acessá-la para que o serviço desejado seja alcançado. Trabalhando com a idéia de recursos torna-se possível oferecer representações em diferentes formatos e linguagens (MORO; DORNELES; REBONATTO, 2011).

Um recurso é uma fonte de representações e uma representação são apenas dados sobre o estado do recurso corrente. A maioria dos recursos na *Web* são itens de dados próprios como uma lista de produtos, logo, uma representação óbvia de um recurso são seus dados em si. O servidor pode então oferecer uma lista de produtos como um documento XML, uma página *Web* ou em formato JSON (MORO; DORNELES; REBONATTO, 2011).

As informações no cabeçalho HTTP podem tanto originar do *browser* do cliente como também serem adicionadas na criação da requisição, pelo cliente consumidor do serviço. O campo *Accept-Language* do cabeçalho HTTP pode ser usado pelo servidor para identificar em que formato o cliente espera o retorno da requisição ou o campo *Content-Type* para fins de identificação do formato solicitado (MORO; DORNELES; REBONATTO, 2011).

2.5.1. Media Types

Os *Media Types* são utilizados para que o cliente saiba como trabalhar com o resultado (e não com tentativa e erro, por exemplo), dessa maneira, os *Media Types* são formas padronizadas de descrever uma determinada informação.

A forma de utilizar os *Media Types* são divididos em tipos e subtipos, e acrescidos de parâmetros (se houver). São compostos com o seguinte formato: tipo/subtipo. Se houver parâmetros, o ; (ponto e vírgula) será utilizado para delimitar a área dos parâmetros. Portanto, um exemplo de *Media Type* seria *text/xml*; charset="utf-8" (para descrever um XML cuja codificação seja UTF-8) (SAUDATE, 2013).

Em serviços REST, vários tipos diferentes de *Media Types* são utilizados. As maneiras mais comuns de representar dados estruturados, em serviços REST, são via XML e JSON, que são representados pelos *Media Types application/xml* e *application/json*, respectivamente. O XML também pode ser representado por *text/xml*, desde que possa ser considerado legível por humanos (SAUDATE, 2013).

2.5.2. XML

XML é uma sigla que significa *eXtensible Markup Language*, ou linguagem de marcação extensível. Por ter esta natureza extensível, consegue-se expressar grande parte das informações utilizando este formato (SAUDATE, 2013).

As estruturas mais básicas em um XML são as *tags* e os atributos, de forma que um XML simples tem o seguinte formato:

<*tag* atributo="valor">conteúdo da *tag*</*tag*>.

XML é uma linguagem bastante semelhante a HTML (*HyperText Markup Language*), porém, com suas próprias particularidades. Por exemplo, todo arquivo XML tem um e apenas um elemento raiz (assim como HTML), com a diferença de que este elemento raiz é flexível o bastante para ter qualquer nome. Por exemplo, para representar vários produtos, é preciso encapsulá-los em um único elemento raiz (no exemplo a seguir, produtos):

```
<produtos>
    <produto id="1">
```

```
        <nome>Produto 1</nome>
    </produto>
    <produto id="2">
        <nome>Produto 2</nome>
    </produto>
</produtos>
```

2.5.3. JSON

JSON é uma sigla para *JavaScript Object Notation*. É uma linguagem de marcação criada por Douglas Crockford e descrito na RFC (*Request for Comments*) 4627, e serve como uma contrapartida a XML. Tem por principal motivação o tamanho reduzido em relação a XML, e acaba tendo uso mais propício em cenários onde largura de banda (ou seja, quantidade de dados que pode ser transmitida em um determinado intervalo de tempo) é um recurso crítico (SAUDATE, 2013).

A marcação JSON segue o modelo:

- Um objeto contém zero ou mais membros;
- Um membro contém zero ou mais pares e zero ou mais membros;
- Um par contém uma chave e um valor;
- Um membro também pode ser um *array*.

O formato desta definição é o seguinte:

```
{"nome do objeto" : {"nome do par" : "valor do par"}}
```

Caso seja uma listagem, o formato é o seguinte:

```
{"nome do objeto" : [
        {"nome do elemento" : "valor"},
        {"nome do elemento" : "valor"}
    ]
}
```

2.6. HATEOAS

Uma das técnicas mencionadas por Roy Fielding é o uso de HATEOAS (*Hypermedia As The Engine Of Application State*). Trata-se de algo que o desenvolvedor *web* já conhece, porém como o termo *link*.

Toda vez que uma página *web* é acessada, além do texto da página, diversos links para outros recursos são carregados. HATEOAS considera estes *links* da mesma forma, através da referência a ações que podem ser tomadas a partir da entidade atual. Na Figura 4 é ilustrado um XML que possui uma compra e diversos itens da compra, porém existe um elemento pagamento o qual é um link para o pagamento associado a essa compra (SAUDATE, 2013).

Figura 4: HATEOAS.

```
1 <compra id="123">
2     <item>
3         <cerveja id="1">Stella Artois</cerveja>
4         <quantidade>1</quantidade>
5     </item>
6     <link rel="pagamento" href="/pagamento/123" />
7 </compra>
```

Fonte: SAUDATE (2013).

2.7. Considerações finais

Neste capítulo, foram apresentado as principais características do estilo de arquitetura REST. Por ser um estilo de arquitetura, a especificação de REST não define quais as tecnologias e como deve ser uma implementação de um *Web Service* baseado em REST, define somente característica que esse *Web Service* deve apresentar.

Com base nisso, muitas variações foram apresentadas, como RESTful, que define o uso do protocolo HTTP para transporte de dados, o uso de recursos, o uso dos métodos do HTTP para determinar as ações sobre esses recursos e a independência na representação de dados. Esses conhecimentos foram importantes para a criação do sistema gerador de API que será apresentado a seguir.

3. FERRAMENTA PARA GERAR API REST

Com a necessidade ampla de integrar sistemas em um espaço de tempo cada vez mais curto, surgiu a necessidade de uma ferramenta que agilizasse esse processo e ao mesmo tempo fosse eficiente, segura e com uma interface amigável para o usuário.

Outras soluções foram estudadas como SOAP, entre outros. Porém, as mais representativas são as baseadas no estilo arquitetural REST pois possui uma interface simples e bem definida. Como inúmeras aplicações foram e ainda são desenvolvidas sem levar em consideração os conceitos de integração de serviço, a migração pode ser algo difícil e com alto custo. Existem algumas ferramentas porém, elas apresentam limitações que são superadas com este trabalho.

As principais ferramentas são:

- **Slashdb**

Disponibiliza um banco de dados para aplicações web, móvel e empresarial gerando automaticamente uma API REST/HTTP. O conteúdo do banco de dados torna-se acessível para a leitura e escrita em vários formatos de dados padrão, como HTML, XML, JSON e CSV para o máximo de interoperabilidade (SlashDB, 2014).

- **SqlREST**

Expõe um banco de dados como *Web Service* usando HTTP e XML, os recursos podem ser consultados, removidos e alterados (Online REST Web Service Demo, 2014).

- **RestSQL**

É uma camada de acesso a dados para clientes HTTP. O RestSQL é um framework de persistência ou um intermediário para a clássica arquitetura de três camadas: cliente, servidor e banco de dados da aplicação Ele também pode ser incorporado em qualquer camada intermediária como uma biblioteca Java (restSQL Overview, 2014).

A Tabela 3 apresenta um comparativo entre este trabalho e as principais ferramentas do mercado.

Tabela 3: Comparativo das ferramentas.

	Aplicação proposta	**Slashdb**	**SqlREST**	**RestSQL**
Leitura de dados	SIM	SIM	SIM	SIM
Escrita de dados	SIM	SIM	SIM	SIM
Múltiplos bancos de dados	SIM	SIM	SIM	SIM
Software como serviço	SIM	NÃO	NÃO	NÃO
Interface web amigável	SIM	SIM	NÃO	NÃO
Formatos de dados: XML e JSON	SIM	SIM	NÃO	NÃO
Outros formatos de dados: HTML e CSV	NÃO	SIM	NÃO	NÃO
Validadores nas requisições de entrada	SIM	NÃO	NÃO	NÃO
API's públicas	SIM	NÃO	NÃO	NÃO
Cache na leitura de dados	SIM	NÃO	NÃO	NÃO
Autenticação por usuário	SIM	SIM	NÃO	SIM
Paginação dos dados	SIM	NÃO	NÃO	NÃO
Documentação gerada automaticamente	SIM	NÃO	NÃO	NÃO

Fonte: o autor.

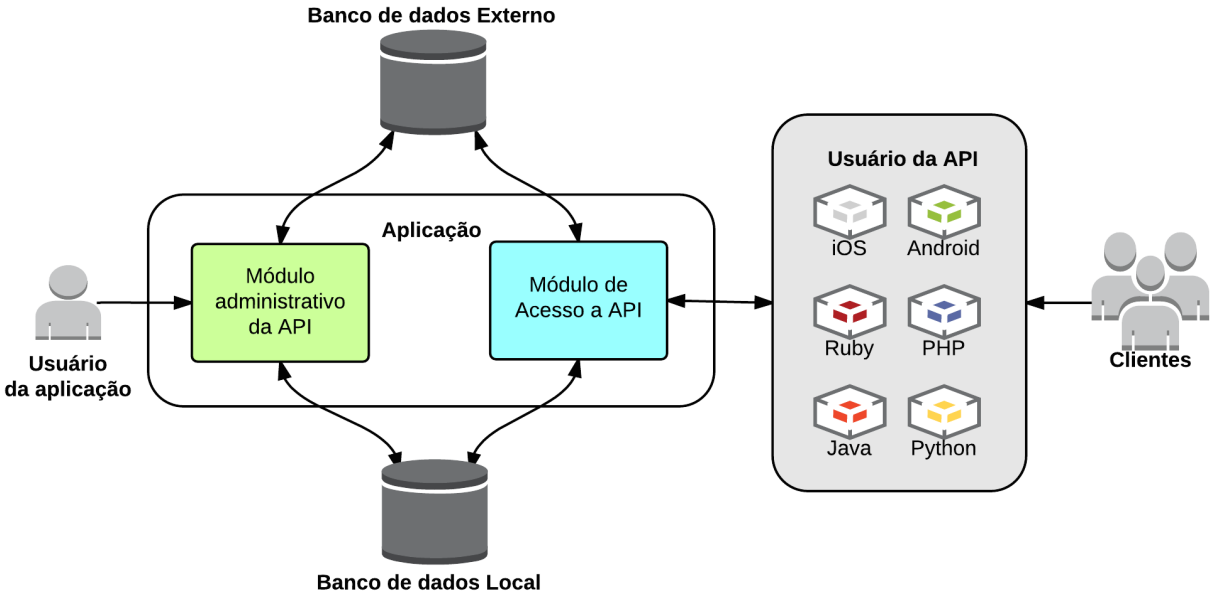
Para facilitar a geração de *Web Services* é necessário ter ferramentas que automatizem os processos que foram citados acima. Sendo assim, o mercado necessita de uma ferramenta que acelere a migração de dados e que seja de baixo custo para ser implementada em sistemas já existentes ou, que podem ser criados utilizando as facilidades da solução desenvolvida.

Neste trabalho foi desenvolvida uma ferramenta para gerar API's REST. Ela permite gerenciar e publicar *Web Services* de uma forma simples e rápida, deixando os complicados detalhes de implementação escondidos do desenvolvedor.

3.1. Arquitetura da aplicação

A arquitetura da aplicação foi projetada em dois módulos principais, que podem ser vistos na Figura 5 e logo depois é apresentada a definição de cada item da Figura.

Figura 5: Arquitetura da Aplicação.



Fonte: o autor.

- **Módulo administrativo da API**

Neste módulo os usuários da aplicação podem criar e gerenciar suas API's, definir permissões de acesso e gerenciamento em suas API's por outros usuários, definir a conexão com o banco de dados que o sistema deverá mapear para gerar os recursos, escolher quais recursos estarão disponíveis e quais ações (ler, criar, alterar e apagar) poderão ser executadas em cada recurso, podendo ainda adicionar validadores de entrada nos campos de cada recurso.

Esse módulo também é responsável por realizar a leitura dos metadados do banco de dados externo e salvar essas informações de forma tratada no banco de dados local da aplicação para que os dados do banco de dados externo possam ser acessados em arquitetura REST no módulo de acesso a API.

- **Módulo de acesso a API**

Módulo responsável por disponibilizar o banco de dados externo em formato REST para os usuários da API, aplicando todas as regras definidas no Módulo administrativo da API.

- **Banco de dados Externo**

O banco de dados que o usuário gostaria de disponibilizar em formato REST.

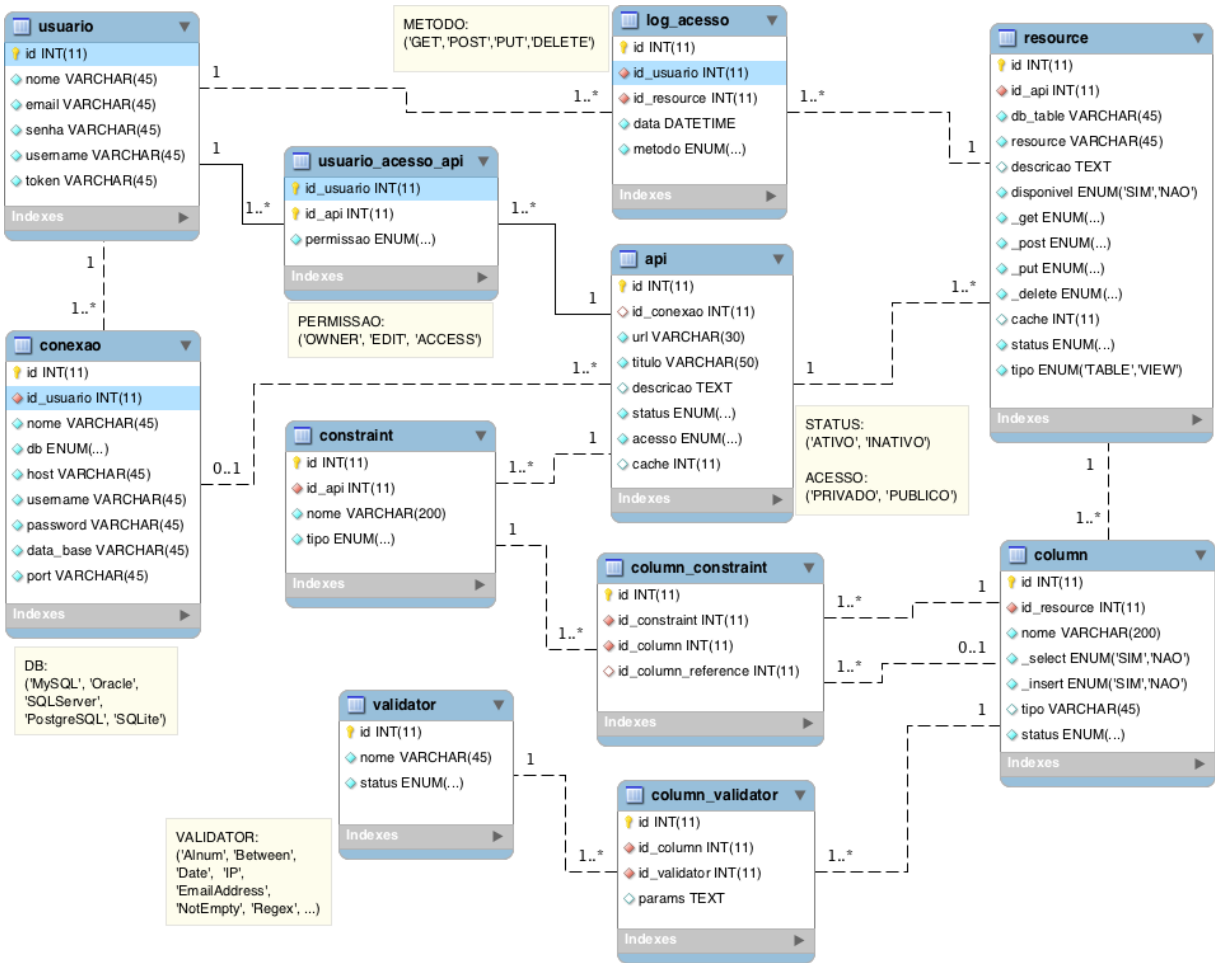
Os seguintes SGBD's são suportados na aplicação: MySQL, PostgreSQL, Oracle, SQLite, SQLServer e IBM DB2.

Para o bom funcionamento da aplicação é indispensável que o usuário fornecido para acessar o banco de dados externo tenha permissão para leitura dos metadados e privilégios para as ações de *SELECT*, *INSERT*, *UPDATE* e *DELETE* no banco de dados. Além é claro de liberar acesso externo ao banco de dados informado, que pode ser feito liberando o IP da aplicação no servidor onde o banco de dados externo se encontra.

- **Banco de dados Local**

É o banco de dados da aplicação, onde as informações sobre os usuários e os metadados do banco de dados externo são salvos de forma tratada, para mais tarde serem acessados os dados da base externa. A Figura 6 ilustra o diagrama entidade relacional da base de dados da aplicação.

Figura 6: Diagrama entidade relacional da base de dados da aplicação.



Fonte: o autor.

- **Usuário da aplicação**

É o usuário que tem um banco de dados e gostaria de disponibilizar em formato REST, ele vai utilizar o módulo administrativo da API para fazer todo o gerenciamento de sua API.

- **Usuário da API**

É o usuário que possui uma conta de usuário da aplicação mas gostaria de consumir a API disponibilizada por ele ou por outro usuário da aplicação via arquitetura REST; para conseguir realizar o consumo da API ele necessita ter permissão de acesso ou edição na mesma. O usuário da API pode ser considerado qualquer cliente HTTP 1.1 que consiga realizar requisições; na Figura 5 é ilustrado um exemplo de linguagens de programação e sistemas operacionais diferentes consumindo a mesma API.

Para realizar a autenticação na API além do e-mail e senha informados no cadastro o usuário precisará informar o *token* de segurança; o *token* é um código único associado ao usuário e gerado automaticamente pelo sistema, a qualquer momento o usuário poderá solicitar a geração de um novo *token*. Os passos para realizar a autenticação e como consumir a API serão explicados na seção 3.4.

3.2. Funcionalidades da aplicação

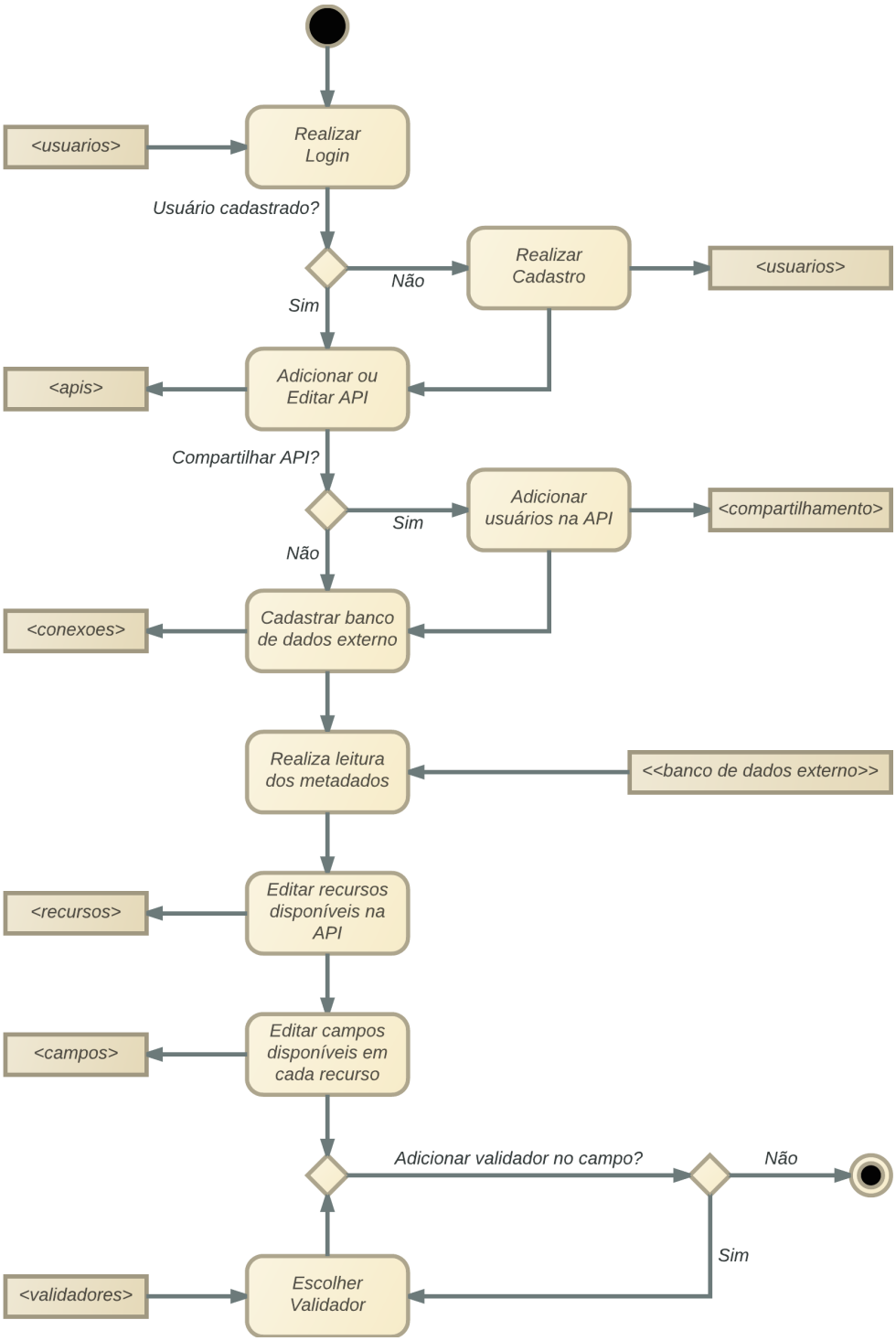
A aplicação foi projetada para funcionar como SAAS (*Software as a service*), que é um programa de software disponibilizado como um serviço compartilhado na nuvem e liberado como um “produto". O modelo de entrega SAAS é usado normalmente para fazer um serviço reutilizável na nuvem e amplamente disponível (muitas vezes no mercado) para uma gama de consumidores variados. O usuário cria sua conta e as informações criadas por ele no sistema são mantidas separadas dos outros usuários (ERL, PUTTINI, MAHMOOD, 2013, p. 66).

Para ilustrar as funcionalidades da aplicação foi utilizado o diagrama de atividade UML; esse diagrama descreve o fluxo das atividades, com suporte para o comportamento condicional e paralelo. O símbolo central desse diagrama é a atividade, que é o estado de fazer algo, por exemplo: um processo do mundo real, como digitar um carta, ou a execução de uma rotina de software, como chamar um método em uma classe. Esse diagrama é útil para

criação de conexões em *workflow* e na descrição de comportamento que tem vários processamentos em paralelo (FOWLER, SCOTT, 1999, p. 129).

Na Figura 7 será apresentado o processo de criação e gerenciamento de uma API no sistema e logo após uma explicação detalhada de cada atividade no diagrama.

Figura 7: Diagrama de atividade - Gerenciar API.



Fonte: o autor.

As seguintes funcionalidades estão disponíveis na aplicação:

1. **Realizar login (usuário):** Para o usuário ter acesso as funcionalidades do sistema, o mesmo deve se autenticar através da combinação de e-mail e senha. Caso os dados informados sejam incorretos uma mensagem de erro será exibida.
2. **Realizar cadastro (usuário):** A aplicação provê mecanismos de cadastro de usuários, sendo necessário informar as seguintes informações (nome, e-mail, *username*, senha e confirmar senha).
3. **Adicionar ou editar Api:** Após o usuário realizar o cadastro ou o login no sistema, o mesmo poderá editar ou criar suas API's, sendo necessário informar:
    a) URL: subdomínio onde a API estará disponível, caso já exista uma outra API cadastrada no sistema para o subdomínio informado uma mensagem de erro será apresentada e um novo subdomínio deverá ser informado.
    b) Titulo: Texto que os outros usuários poderão encontrar a API.
    c) Descrição: Texto informativo que será exibido na documentação da API criada.
    d) *Cache*: O usuário pode informar o tempo em segundos para que os dados dos recursos solicitados permaneça em *cache* no cliente. Caso seja informado, um cabeçalho *Cache-Control* será adicionado na resposta HTTP.
    e) Status: Define se a API está ativa ou não, o usuário pode inativar a API a qualquer momento.
    f) Acesso: O usuário pode definir a API sendo pública ou privada, com o acesso público qualquer usuário pode encontrar a API através da funcionalidade API's Públicas e se inscrever e consumir os recursos fornecidos pela API. No acesso privado somente os usuários adicionados pela funcionalidade compartilhamento terão permissão para acessar a API.
4. **Compartilhar API:** O usuário proprietário da API pode adicionar outros usuários para gerenciar a API ou somente consumir os recursos da mesma. Para adicionar um novo usuário deve-se informar o e-mail ou o *username*, e

escolher se o mesmo terá permissão para editar as informações. Com a permissão de editar o novo usuário poderá alterar as informações básicas da API, adicionar novos usuários e gerenciar os recursos, só não poderá alterar ou visualizar as informações de conexão com o banco de dados externo; as permissões podem ser vistas na Tabela 4.

Tabela 4: Lista de permissões da API.

	OWNER	***EDIT***	***ACCESS***
Editar API	Sim	Sim	Não
Editar Conexão	Sim	Não	Não
Editar Recursos	Sim	Sim	Não
Apagar API	Sim	Não	Não
Consumir da API	Sim	Sim	Sim

Fonte: o autor.

5. **Cadastrar banco de dados externo:** Após o cadastro da API, deve-se definir a fonte de dados da mesma, que serão as informações sobre a conexão com o banco de dados externo, caso a aplicação não consiga realizar a conexão uma mensagem de erro será exibida.
6. **Leitura dos metadados:** A leitura dos metadados ocorrerá após o usuário fornecer as informações sobre a fonte de dados e a aplicação ter sucesso na conexão no banco de dados externo. A partir da coleta dos metadados a aplicação criará os recursos disponíveis na API. A qualquer momento o usuário poderá executar uma nova leitura dos metadados. O fluxo completa da coleta dos metadados será apresentado na seção 3.4.1.
7. **Editar recursos disponíveis na API:** Após a leitura dos metadados uma lista dos recursos (tabelas e visões do banco de dados externo) será apresentado na tela, os usuários com permissão de editar poderão alterar as seguintes informações em cada recurso:
    a) URI do recurso: Caminho onde o recurso poderá ser consumido, por padrão será o nome da tabela que o recurso está associado, porém o usuário poderá definir um nome diferente, se o valor informado estiver sendo usado em outro recurso da API uma mensagem de erro será

apresentada e um novo valor deverá ser informado. A URL de acesso do recurso é formada pelo subdomínio da API mais a URI do recurso, ficando no seguinte formato: http://*subdominio-da-api*.exemple.com/ *nome-do-recurso*

b) Descrição: Texto informativo sobre o recurso que será apresentado na documentação da API criada.

c) *Cache*: O usuário pode informar o tempo em segundos para que os dados do recurso solicitado permaneça em cache no cliente; esse valor sobrescreve o valor do cache informado no cadastro da API. Caso seja informado, um cabeçalho *Cache-Control* será adicionado na resposta HTTP.

d) Disponível: Por padrão todos os recursos estarão disponíveis, mas o usuário poderá a qualquer momento tornar indisponível o recurso. Se um recurso indisponível for solicitado uma resposta HTTP com o código 404 (*Not Found*) será retornada.

e) Ações disponíveis: Para cada recurso poderá ser definido as ações que os usuários com permissão de acesso poderão executar na API. Se uma ação que estiver indisponível for requisita uma resposta HTTP com o código 405 (*Method Not Allowed*) será retornada. As ações disponíveis são:

- Leitura (*GET*): Solicitar uma lista dos dados do recurso ou somente um único recurso informando o identificador único do recurso.
- Adicionar (*POST*): Adicionar dados em um recurso.
- Alterar (*PUT*): Alterar os dados de um recurso.
- Apagar (*DELETE*): Apagar os dados de um recurso.

8. **Campos disponíveis no recurso:** Para cada campo de um recurso poderá definir se o mesmo estará disponível para as ações de leitura, escrita (Adicionar e Alterar) ou ambas.

9. **Adicionar validadores nos campos:** Em cada campo também poderá adicionar validadores de entrada, que serão executados sempre que uma ação de escrita (Adicionar ou Alterar) for executada no recurso. A configuração dos validadores será feita via interface gráfica disponibilizada pelo sistema. Para o

funcionamento dessa funcionalidade foi utilizado o componente *Zend Validator* do *Zend framework 2*. Alguns dos validadores disponíveis são:

- *Alnum*: apenas caracteres e dígitos;
- *Between*: o valor informado está entre dois valores;
- *Date*: o valor contém uma data;
- *EmailAddress*: o valor é um endereço de e-mail válido;
- *IP*: o valor é um endereço IP;
- *NotEmpty*: o valor não pode ser vazio;
- *Regex*: verifica se o valor casa com a expressão regular informada;

Além das funcionalidade de gerenciamento da API, o sistema disponibiliza outras funcionalidades:

1. **Geração automática da documentação da API:** A aplicação gera uma documentação automática para cada API criada, baseado no que foi informado na construção da mesma. As seguintes informações serão apresentadas na documentação:
   - Título e descrição da API;
   - Um exemplo de como realizar a autenticação;
   - Uma lista dos possíveis erros e o número de cada erro que podem ocorrer na requisição;
   - Os recursos disponíveis e ações disponíveis em cada recurso;
   - Exemplos de uso na linguagem PHP.
2. **API's públicas:** As API's marcadas como públicas no cadastro poderão ser encontradas por outros usuários do sistema através da funcionalidade API's públicas.
3. **Relatórios de acesso:** A aplicação gera relatórios e gráficos de acesso baseado nas informações geradas nas requisições feitas na API. Com essas informações é possível saber:
   - Qual usuário está consumindo mais recursos na API;
   - Qual recurso está sendo mais acessado;
   - Qual método está sendo mais executado nos recursos;
   - E qual o período das requisições;

3.3. Tecnologias utilizadas

Para criar a aplicação diversas tecnologias foram utilizadas, as quais foram divididas entre *back-end* e *front-end*.

3.3.1. *Back-end*

- ***Zend Framework 2***

Para criar a base MVC da aplicação foi utilizado o *Zend Framework 2*, que é um *framework* PHP *open source* 100% orientado a objetos para o desenvolvimento de aplicações *web*, ele é uma evolução do *Zend Framework 1*, um *framework* PHP de sucesso com mais de 15 milhões de *downloads* (About Zend Framework 2, 2014).

- ***Doctrine***

A conexão e o mapeamento dos objetos para persistência no banco de dados local foram realizados usando o *Doctrine*.

Doctrine que é um ORM (*object-relational-mapper*) para PHP que facilita a persistência de objetos PHP em banco de dados relacional. Ele utiliza o padrão *Data Mapper* como base, conseguindo gerar uma separação completa da lógica de negócio da persistência dos dados no banco de dados (Getting Started with Doctrine, 2014).

- ***Apache***

O servidor *web* escolhido para o projeto foi o Apache HTTP Server. O Apache é um um servidor HTTP de código aberto para os sistemas operacionais modernos. O objetivo dele é fornecer um servidor seguro, eficiente e extensível que fornece serviços HTTP em sincronia com os padrões HTTP atuais (The Apache HTTP Server Project, 2014).

- ***Mysql***

O banco de dados escolhido para gerenciar as informações da aplicação foi o MySql. Que é o sistema de banco de dados de *open source* mais popular do mundo. Os pontos fortes do MySql são: confiabilidade, facilidade de uso e velocidade superior (About MySQL, 2014).

3.3.2. *Front-end*

- ***Bootstrap***

Para criar o *layout* responsivo da aplicação e os padrões de botões e formulários do sistema foi utilizado o *Bootstrap,* que é um *framework* para se criar layout responsivo na *web.* Ele utiliza padrões de classes CSS para criar componentes reutilizáveis usando como base HTML, CSS e *Javascript* (Bootstrap front-end framework, 2014).

- ***jQuery***

jQuery é uma biblioteca JavaScript rápida, pequena e rica em recursos. Ele auxilia na manipulação de documentos, manipulação de eventos, animação e requisições AJAX (*Asynchronous Javascript and XML*) muito mais simples, com uma API fácil de usar, que funciona em uma infinidade de navegadores. Com uma combinação de versatilidade e capacidade de extensão, *jQuery* facilita a escrita de códigos *JavaScript* (jQuery, 2014).

- ***jQuery-pjax***

Pjax é um *plugin jQuery* que usa AJAX e *pushState* para oferecer uma experiência de navegação rápida. Ele funciona buscando o html no servidor via AJAX e substituindo o conteúdo de um contêiner na página com o HTML retornado no AJAX. Em seguida, ele atualiza a url atual do navegador usando *pushState* sem recarregar o *layout* da página ou quaisquer recursos (*JavaScript*, CSS), dando a aparência de um carregamento rápido e completa da página (Pjax, 2014).

3.4. Implementação da aplicação

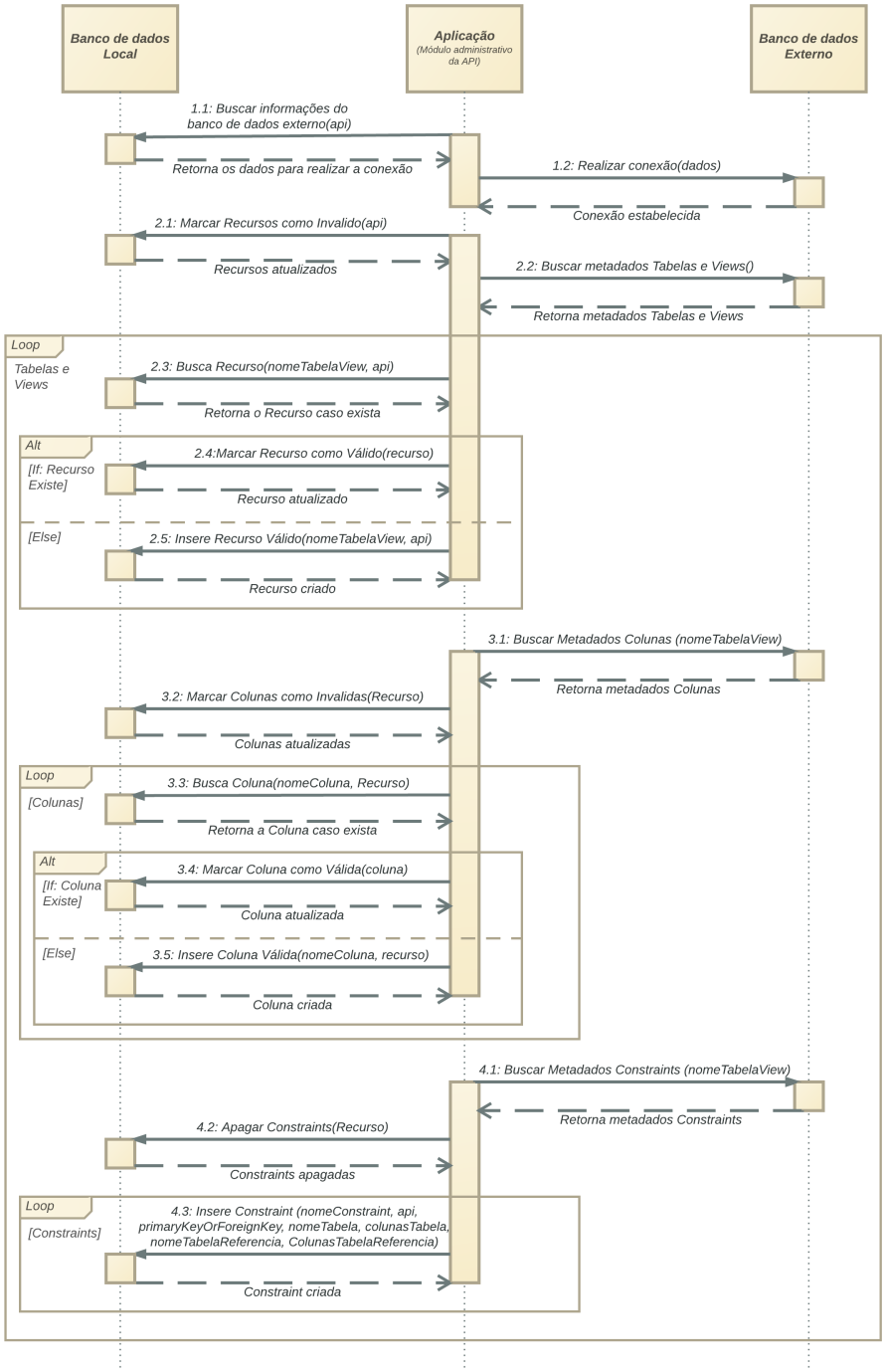
A seguir será explicado como alguns dos principais processos da aplicação foram implementados e os métodos utilizados para criar essas funcionalidades.

3.4.1. Mapeamento da base de dados

A Figura 8 ilustra o fluxo do mapeamento da base de dados e como é feita a interação entre os componentes banco de dados local, aplicação e banco de dados externo.

Para ilustrar foi utilizado o diagrama UML de sequência, esse diagrama descreve como grupo de objetos colaboram em um determinado comportamento. O diagrama de sequência consiste em um diagrama que tem o objetivo de mostrar como as mensagens entre os objetos são trocadas no decorrer do tempo para a realização de uma operação (FOWLER, SCOTT, 1999).

Figura 8: Diagrama de sequência, mapeamento da base de dados.



Fonte: o autor.

A operação de mapeamento da base de dados foi dividido em 4 pontos principais, os quais são:

1. **Realizar conexão no banco de dados externo:** A primeira ação é buscar as informações necessárias para realizar a conexão ao banco que será mapeado, essas informações são cadastradas pelo usuário da aplicação no cadastro da API. A ação seguinte é estabelecer a conexão com o banco de dados externo.
2. **Buscar metadados de tabelas e visões e cadastrar recursos:** A operação de mapeamento pode ser repetido sempre que for necessário, como por exemplo quando uma tabela for adicionada ou removida do banco.
    - Após conectado ao banco de dados externo é realizada uma consulta para obter uma lista com os nomes das *tables* (tabelas) e *views (*visões*)* contidas no banco.
    - Caso o processo de mapeamento já tenha sido feito anteriormente todos os recursos da API são marcados automaticamente como inválidos, isso é feito para o caso de uma tabela ter sido removida ou renomeada no banco de dados, com isso essa tabela aparecerá na listagem de recursos como inválida e poderá ser excluída pelo usuário administrador da API.
    - O próximo passo é percorrer a lista com os nomes das tabelas e visões; para cada uma delas é verificado se já existem na tabela de recursos do banco de dados local, caso existam é atualizado como recurso válido, do contrário um novo recurso é adicionado na API e associado a tabela ou visão do banco de dados externo.
3. **Buscar metadados de colunas e cadastrar colunas:** Para cada tabela ou visão são buscados os metadados das colunas no banco de dados externo; com esses dados obtidos os seguintes processos são realizados:
    - As colunas do recurso que contém a tabela ou visão como fonte de dados são marcados como inválidas, seguindo o mesmo conceito dos recursos, caso uma coluna tenha sido apagada ou removida do banco de dados externo o usuário administrador poderá removê-la da API.

- A lista de colunas é percorrida e para cada uma é verificado se a mesma já existe como coluna do recurso, caso exista, é marcada como válida, do contrário, uma nova coluna é adicionada ao recurso.

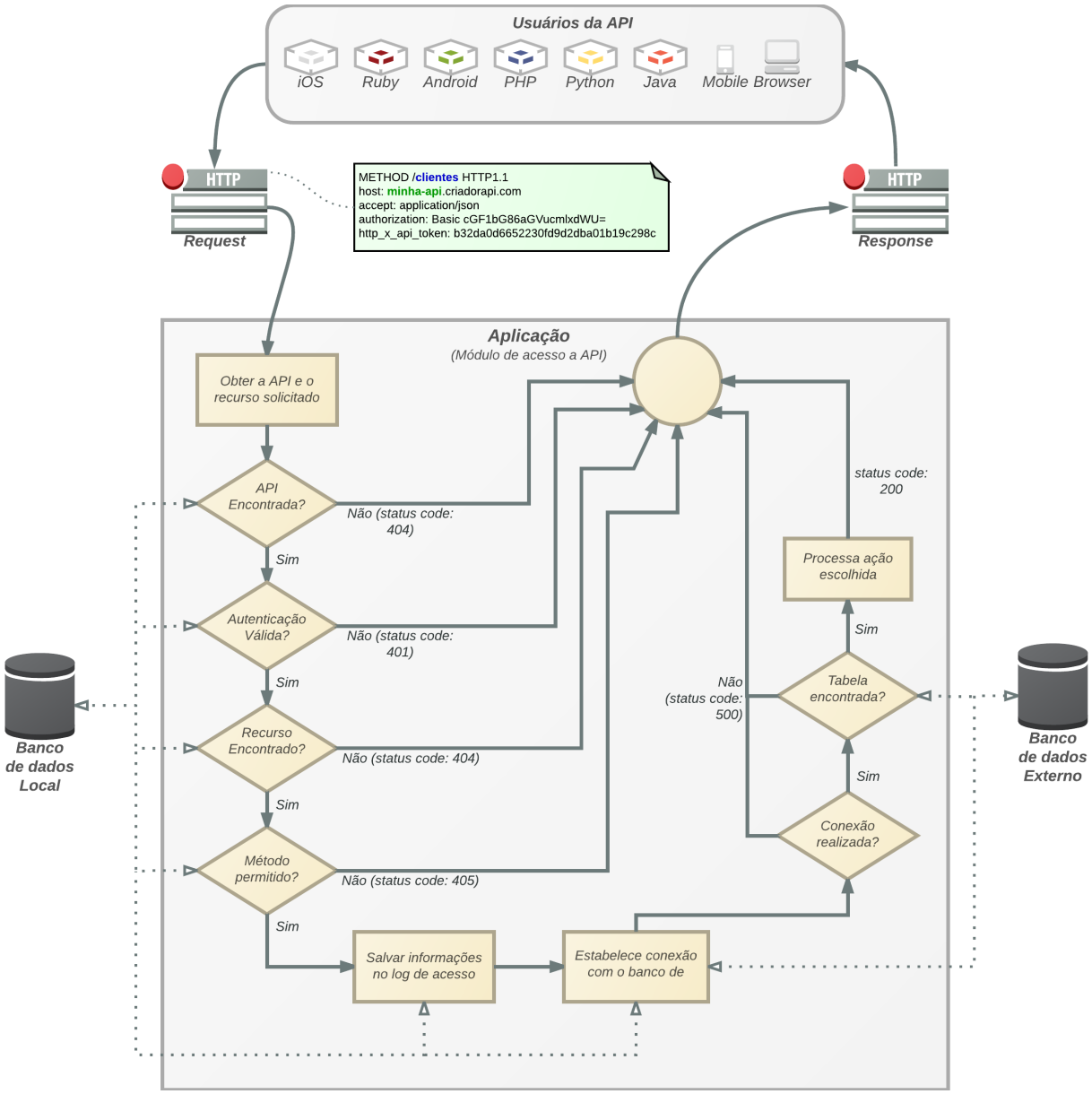
4. **Buscar metadados de *constraints* e cadastrar *constraints*:** Ainda no laço de tabelas e visões para cada uma delas são buscados os metadados das restrições (*constraints*). Os dados retornados são: nome da *constraint*, o tipo (chave primária ou chave estrangeira), colunas da tabela atual que possuem chaves, nomes das tabelas que possuem referências com essa tabela e as colunas da tabela de referência. Após obter essas informações os seguintes passos são realizados:
   - Todas as *constraints* cadastradas na API são removidas.
   - A lista de *constraints* é percorrida e inserida na tabela de *constraint* do banco de dados local com as informações necessárias.

   As informações sobre as *constraints* são importantes para saber a chave primária do recurso, que será utilizado quando um único documento do recurso for solicitado na API, mais informações na seção 3.4.3.

3.4.2. Processamento principal de uma requisição na API

A Figura 9 ilustra o processamento principal de uma requisição na API; esse processo ocorrerá em todas as requisições enviadas a API independente do método HTTP ou URL solicitada.

Figura 9: Processamento principal de uma requisição na API.



Fonte: o autor.

As seguintes ações são realizadas no processamento principal:

1. **Obter a API e o recurso solicitado:** Quando um cliente HTTP envia uma requisição, a aplicação obtém os dados da API e do recurso a partir da URL solicitada, na Figura 9 a url solicitada é http://minha-api.criadorapi.com/clientes, portando logo a API solicitada é "minha-api" e o recurso solicitado é “clientes";

2. **Verificar se a API existe:** Obtido os dados da API, verifica-se a API existe no banco de dados local, caso não exista é retornado ao cliente uma resposta HTTP com o código de status 404 (*Not Found*).
3. A**utorização e autenticação:** Primeiramente o usuário que quiser utilizar a API deverá ter permissão de acesso ou edição na API. Para realizar a autenticação, em todas as requisições deverá ser enviado 2 cabeçalhos HTTP, os quais são:
    - ***Authorization*:** esse cabeçalho segue o padrão explicado no item 1.1.3.1, o qual deve seguir o seguinte formato: "**Authorization: Basic Base64({#email}:{#senha})**". As variáveis **{#email}** e **{#senha}** devem ser substituídas respectivamente pelo e-mail e senha do usuário que está realizando a requisição.
    - ***HTTP_X_API_TOKEN***: O *token* de segurança é um código único associado ao usuário e gerado automaticamente pelo sistema, esse *token* pode ser encontrado no menu minha conta no sistema. O cabeçalho deve ser enviado no seguinte formato: "**HTTP_X_API_TOKEN: {#token}**", a variável **{#token}** deve ser substituída pelo *token* único do usuário.

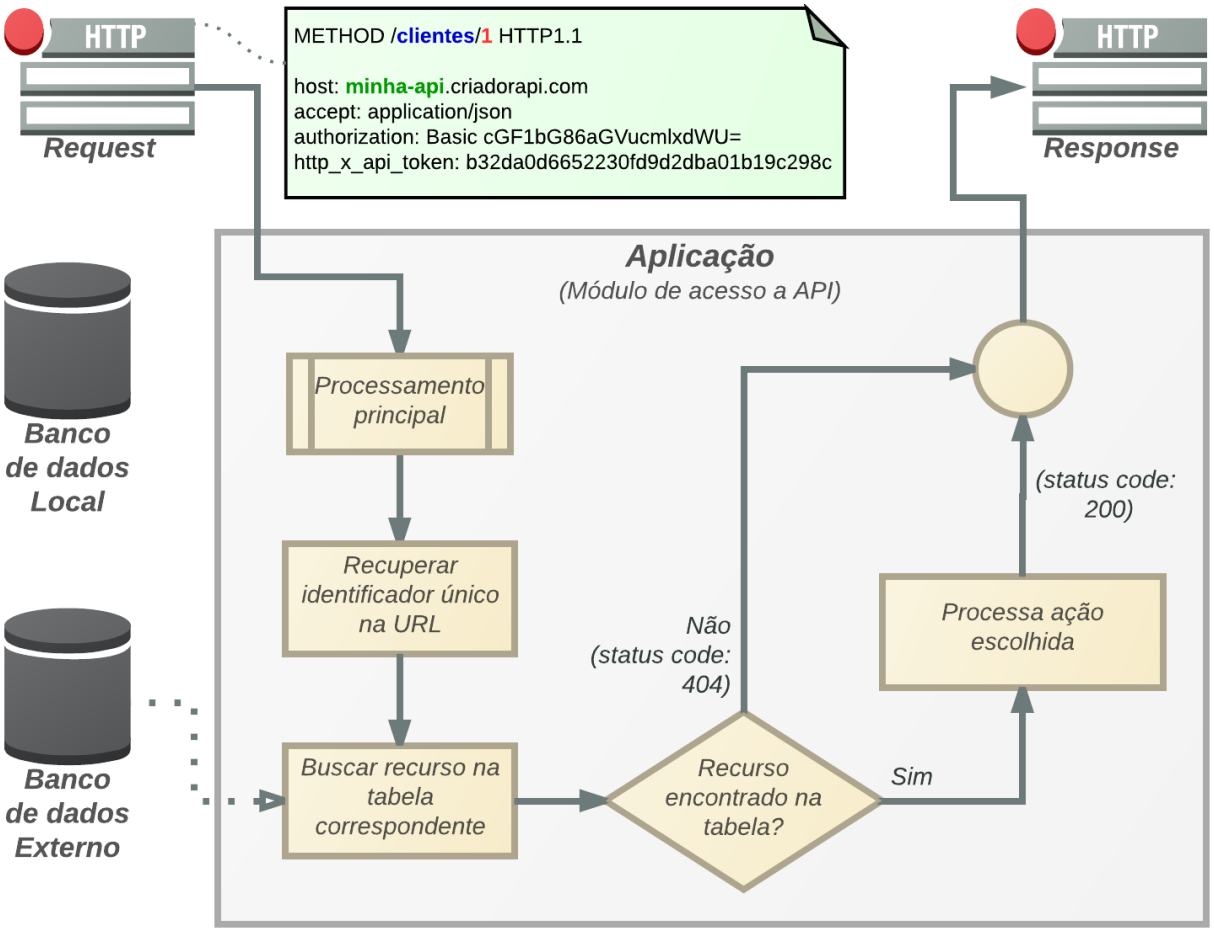
    Caso o usuário não tenha permissão para acessar a API ou alguns dos cabeçalhos enviados esteja inválido, é retornado ao cliente uma resposta HTTP com o código de status 401 (*Unauthorized*).
4. **Verificar se o recurso existe:** Depois de realizado o processo de autenticação é verificado se o recurso solicitado existe no banco de dados local, caso não exista é retornado ao cliente uma resposta HTTP com o código de status 404 (*Not Found*).
5. **Método HTTP permitido no recurso:** Quando uma requisição é enviada uma verificação é realizada para comprovar que o método HTTP (*GET*, *POST*, *PUT* ou *DELETE*) solicitado está permitido para o recurso escolhido. Caso não seja permitido é retornado ao cliente uma resposta HTTP com o código de *status* 405 (*Method not allowed*).

6. **Salvar informações no *log* de acesso:** Toda requisição que for realizada na API e que as informações estejam válidas será salva no *log* de acesso da API, essa informação é útil para gerar o relatório de acesso da API.
7. **Estabelecer conexão com o banco de dados externo:** As informações necessárias para realizar a conexão a base de dados da API solicitada são buscadas no banco de dados local, com essas informações é realizado a tentativa de conexão ao banco de dados externo. Caso a conexão não seja estabelecida é retornado ao cliente uma resposta HTTP com o código de *status* 500 (*Internal server error*).
8. **Verificar se a tabela do recurso existe:** Após realizada a conexão ao banco de dados externo é verificado se a tabela que foi mapeada para o recurso ainda existe, pode ocorrer da tabela ser removida ou renomeada, caso a tabela não seja encontrada é retornado ao cliente uma resposta HTTP com o código de *status* 500 (*Internal server error*).
9. **Processar ação escolhida:** A ação escolhida é verificada pelo método HTTP enviado na requisição, nas seções a seguir será apresentado como é feito o processo para cada ação.

3.4.3. Processando recursos com identificadores únicos

Para as ações de alterar, apagar ou recuperar um único recurso deve ser informado o identificador único na URL solicitada. Um processo é realizado para descobrir o identificador único e se o mesmo existe no recurso solicitado, a Figura 10 ajuda a entender esse processo.

Figura 10: Processando uma requisição na API.



Fonte: o autor.

1. **Processamento principal:** É o processo realizado em todos os tipos de requisições na API, mais informações estão descritas na seção 3.4.2;
2. **Recuperar identificador único na URL:** Para as ações de alterar, apagar ou recuperar um único recurso, deverá ser informado o identificador único na URL do recurso, que é a chave primária da tabela que o recurso está associado. Na maioria das vezes esse identificador único será o campo ID da tabela, porém pode ocorrer casos em que a chave primária é composta (dois ou mais campos formando a chave primária).

   Para os casos que a chave primária for simples (um único campo) poderá solicitar a url no seguinte formato: "http://minha-api.criadorapi.com/clientes/1". O valor “1" no final da url significa que o solicitante gostaria de obter o recurso cliente no qual a chave primária tenha o valor 1 na tabela.

No caso de a chave primária for composta, o seguinte formato deve ser seguido:

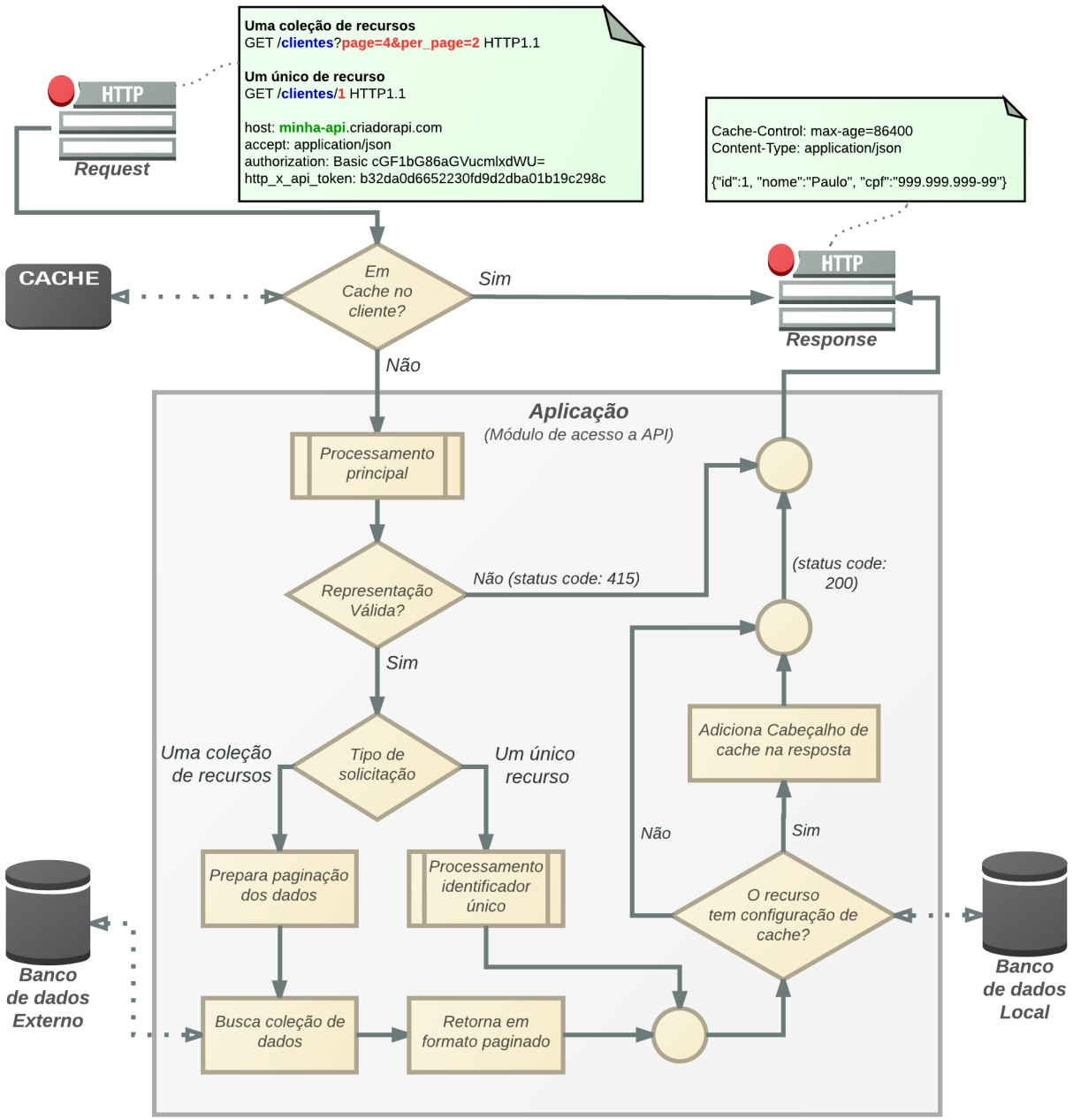
- Primeiramente deve ser criada um texto JSON seguindo o formato: {"chave1":"valor1", "chave2":"valor2", …}, com as chaves primárias e os valores dos campos;
- Em seguida codificar o texto JSON usando *Base64*.
- A URL solicitada nesse caso ficaria no seguinte formato: "http://minha-api.criadorapi.com/clientes/J2YWxvcjEiLCAiY2hhdmUyIjoidmFao0=".

3. **Buscar o recurso na tabela correspondente:** Após obter o identificar único é realizada uma consulta no banco de dados externo para retornar a informação solicitada. Caso não exista é retornado ao cliente uma resposta HTTP com o código de *status* 404 (*Not Found*).
4. **Processar ação escolhida:** A ação escolhida é verificada pelo método HTTP enviado na requisição, nas seções seguintes será apresentado como é feito o processo para cada ação.

3.4.4. Leitura de dados na API

Para as ações de leitura de um único recurso ou uma coleção de dados do recurso utiliza-se o método HTTP *GET*. A Figura 11 ilustra o processo de uma requisição de leitura de dados na API.

Figura 11: Processo de leitura de dados na API.



Fonte: o autor.

1. ***Cache*:** Caso o cliente HTTP já realizou um solicitação anteriormente para a URL do recurso, possivelmente esse recurso esteja em *cache* no próprio cliente, se estiver em *cache* e o período do *cache* estiver válido é retornado o conteúdo solicitado ao cliente sem a necessidade de consultar o servidor da aplicação;
2. **Processamento principal:** É o processo realizado em todos os tipos de requisições na API, mais informações estão descritas na seção 3.4.2;

3. **Representação válida:** O cliente deve enviar o cabeçalho *Accept* na requisição para informar em qual formato os dados devem ser retornados. Os *Media Types* suportados são: *application/json*, *application/xml* e *text/xml*. Caso seja enviado um valor inválido será retornado ao cliente uma resposta HTTP com o código de *status* 415 (U*nsupported media type*).
4. **Tipo de solicitação:** Se na URL tiver informado o identificador único significa que a resposta deve conter somente o documento solicitado, do contrário, uma coleção de dados do recurso é retornado.
5. **Processamento identificador único:** É o processo realizado em todos as requisições que possuem o identificador único na URL, mais informações estão descritas na seção 3.4.3;
6. **Prepara paginação dos dados:** Para as requisições de coleção de dados na API o retorno dos dados são paginados por padrão, são retornados 25 itens da página 1. Porém o solicitante pode alterar esses valores passando os valores em parâmetros na URL solicitada. Para alterar o numero de itens por página deve-se informar o parâmetro *per_page* e para alterar o número da página de retorno deve-se informar o parâmetro *page.*
7. **Busca coleção de dados:** Após preparar a paginação é realizada uma consulta no banco de dados externo para retornar a informação solicitada. Nesse caso, se nenhum valor for encontrado, não é retornado um código de erro, o retorno será uma coleção vazia.
8. **Retorno paginado:** Para a URL http://minha-api.criadorapi.com/clientes?page=4&per_page=2 que foi solicitada na Figura 11 o resultado seria o texto JSON da Figura12. O qual possui as informações:
    - ***first:*** o número da primeira página;
    - ***previous:*** o número da página anterior;
    - ***current:*** o número da página atual;
    - ***next:*** o número da próxima página;
    - l***ast:*** o número da última página na paginação;
    - ***_links:*** no sistema de dados paginados é retornado juntamente com o resultado os *links* para as outras páginas. Seguindo a técnica HATEOAS

do REST, e com isso facilitando a navegação entre os resultados paginados;

- ***per_page:*** o número de itens que deve ser retornado por página;
- ***total_itens:*** o número total itens;
- ***data:*** os itens da página atual;

Figura 12: Resultado paginado.


```
{
    "status": "success",
    "result": {
        "first": 1,
        "previous": 3,
        "current": 4,
        "next": 5,
        "last": 8,
        "_links": {
            "first": "http://minha-api.criadorapi.com/clientes?page=1&per_page=2",
            "previous": "http://minha-api.criadorapi.com/clientes?page=3&per_page=2",
            "current": "http://minha-api.criadorapi.com/clientes?page=4&per_page=2",
            "next": "http://minha-api.criadorapi.com/clientes?page=5&per_page=2",
            "last": "http://minha-api.criadorapi.com/clientes?page=8&per_page=2"
        },
        "per_page": 2,
        "total_itens": 15,
        "data": [
            {
                "id": 7,
                "nome": "Paulo"
            },
            {
                "id": 8,
                "nome": "Henrique"
            }
        ]
    }
}
```

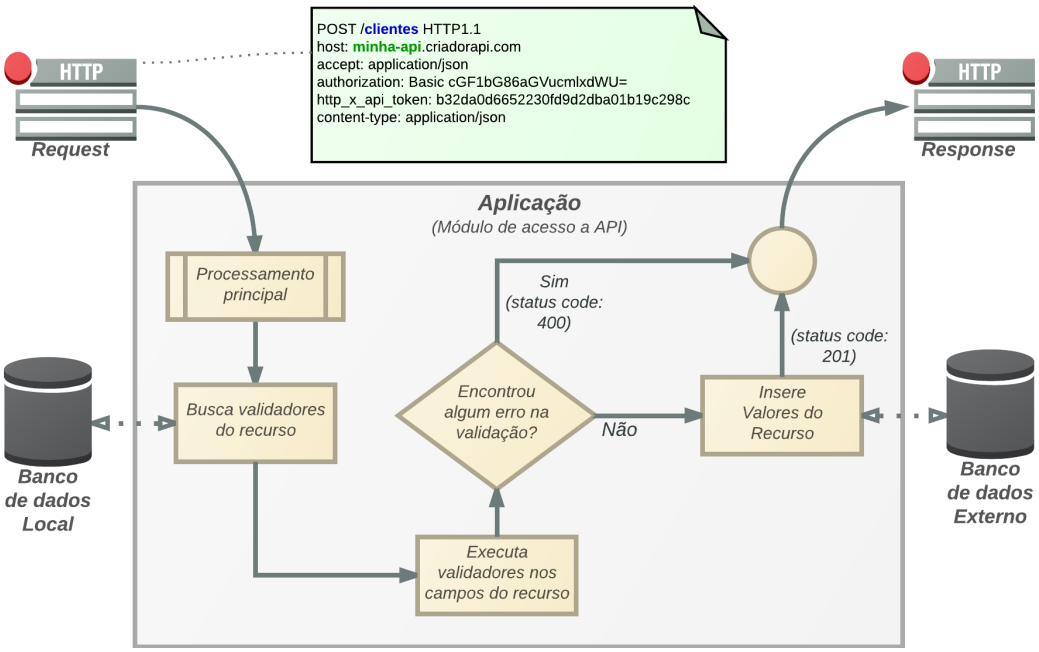

Fonte: o autor.

9. **Configuração de *cache*:** Se na configuração do recurso ou da API estiver cadastrado o período de *cache*, será adicionado o cabeçalho HTTP *Cache-Control* na resposta. Na Figura 10 é ilustrado um cabeçalho de *cache* com o valor "*max-age=86400*", isso significa que o cliente deve manter esses dados armazenados por 86400 segundos (24 horas) antes de realizar uma nova requisição.

3.4.5. Inserção de dados na API

Para a ação de inserção de dados nos recursos da API utiliza-se o método HTTP *POST*. A Figura 13 ilustra o processo de uma requisição de inserção de dados.

Figura 13: Processo de inserção de dados na API.



Fonte: o autor.

1. **Processamento principal:** é o processo realizado em todos os tipos de requisições na API, mais informações estão descritas na seção 3.4.2;
2. **Busca validadores do recurso:** é realizada uma consulta no banco de dados local para encontrar os validadores do recurso que receberá a inserção de dados.
3. **Executa os validadores:** antes de inserir os dados no recurso é feita a validação dos valores que serão inseridos, mas somente nos campos que o usuário gerenciador da API configurou validadores. Caso algum dos validadores encontre um erro nos valores que serão inseridos, é retornado ao cliente uma resposta HTTP com o código de *status* 400 (*Bad request*) e no conteúdo da resposta terá as mensagens de erro da validação executada, como pode ser visto no exemplo da Figura 14.

Figura 14: Erro na validação.


```
{
    "status": "error",
    "result": {
        "message": {
            "number": {
                "notBetweenStrict": "The input is not strictly between '100' and '1000'"
            },
            "zipcode": {
                "regexNotMatch": "The input does not match against pattern '/^\d{5}(?:[-\s]\d{4})?$/'"
            }
        },
        "code": 4001
    }
}
```

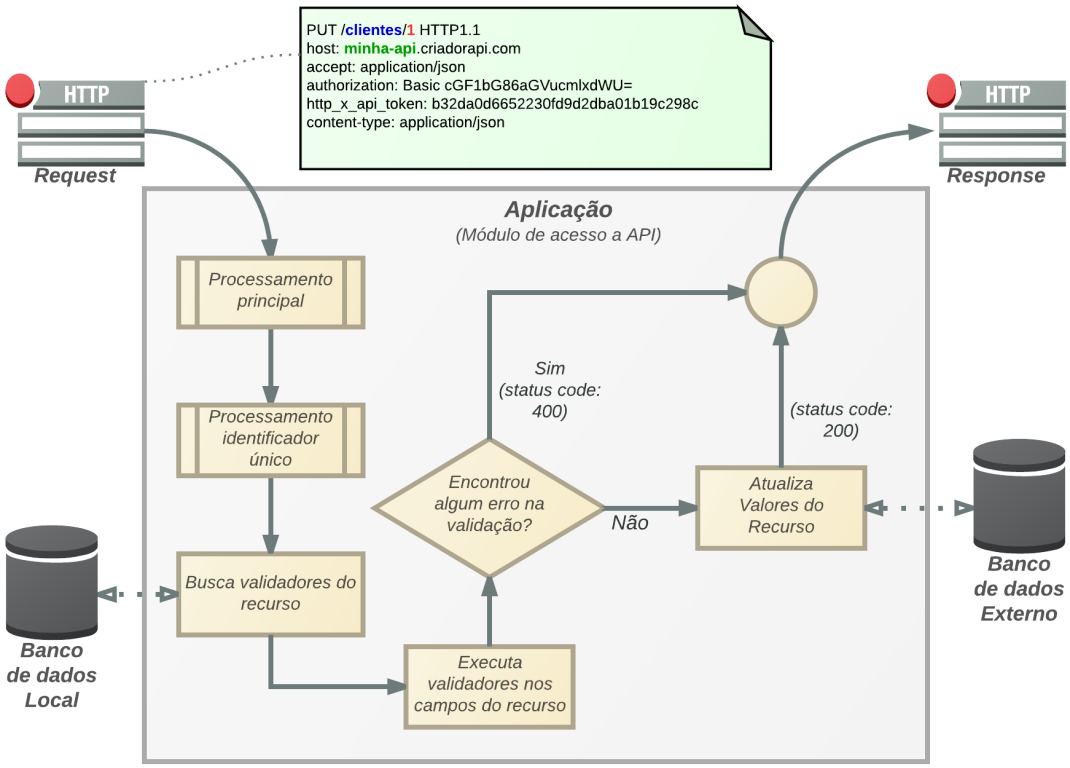

Fonte: o autor.

4. **Inserir valores:** se os valores tiverem sucesso na validação, os mesmos serão inseridos na tabela do recurso, e será retornada ao cliente uma resposta HTTP com o código de *status* 201 (*Created*).

3.4.6. Alteração de dados na API

Para a ação de alteração de dados nos recursos da API utiliza-se o método HTTP *PUT*. A Figura 15 ilustra o processo de uma requisição de alteração de dados.

Figura 15: Processo de alteração de dados na API.



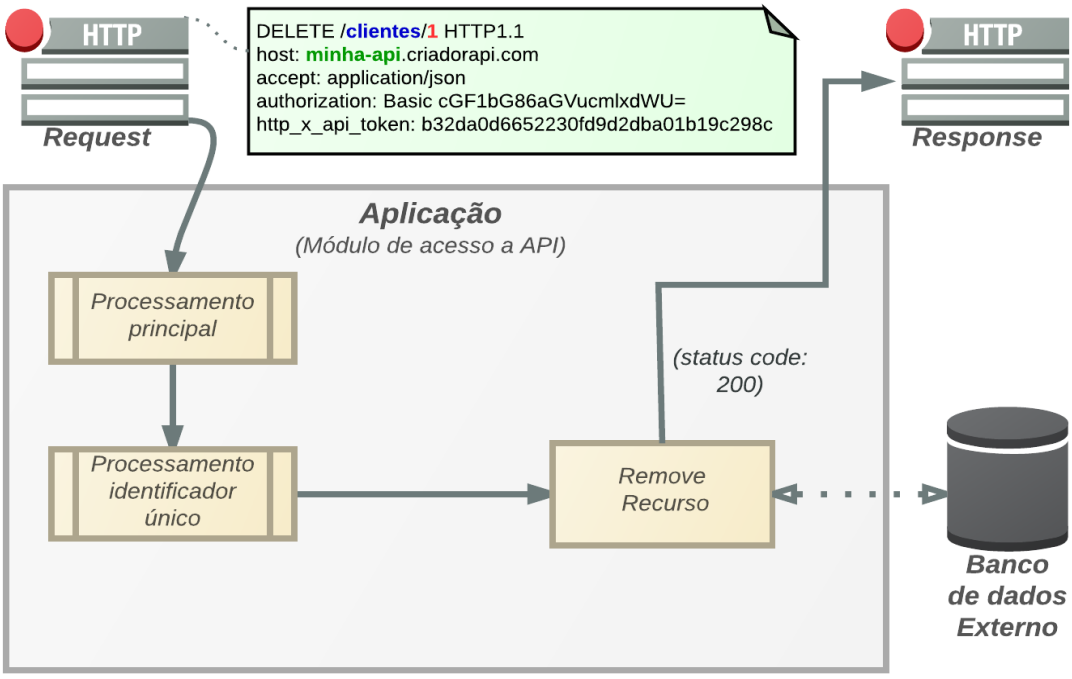
Fonte: o autor.

1. **Processamento principal:** é o processo realizado em todos os tipos de requisições na API, mais informações estão descritas na seção 3.4.2;
2. **Processamento identificador único:** É o processo realizado em todas as requisições que possuem o identificador único na URL, mais informações estão descritas na seção 3.4.3;
3. **Validação dos dados:** o processo de validação dos dados é o mesmo da inserção de dados na API, mais informações estão descritas na seção 3.4.5 item 2 e 3.
4. **Atualizar valores:** se os valores tiverem sucesso na validação, os valores do recurso solicitado serão alterados com os novos dados, e será retornado ao cliente uma resposta HTTP com o código de *status* 200 (*OK*).

3.4.5. Remoção de dados na API

Para a ação de remoção de dados nos recursos da API utiliza-se o método HTTP *DELETE*. A Figura 16 ilustra o processo de uma requisição de remoção de dados.

Figura 16: Processo de remoção de dados na API.



Fonte: o autor.

1. **Processamento principal:** é o processo realizado em todos os tipos de requisições na API, mais informações estão descritas na seção 3.4.2;
2. **Processamento identificador único:** É o processo realizado em todas as requisições que possuem o identificador único na URL, mais informações estão descritas na seção 3.4.3;
3. **Remove dados do recurso:** se houve sucesso no item 2, é removido do recurso o documento solicitado, e será retornado ao cliente uma resposta HTTP com o código de status 200 (*OK*).

3.5. Considerações finais

Neste capítulo foi apresentada a realização da divisão da arquitetura da aplicação, uma explicação das principais funcionalidades contidas no sistema, quais tecnologias foram utilizadas no desenvolvimento e como foi feito a construção das principais funcionalidades da aplicação.

No próximo capítulo serão apresentados os resultados do trabalho desenvolvido, juntamente com um exemplo completo da criação, gerenciamento e o consumo de uma API no sistema.

4. RESULTADOS

Este trabalho resultou em uma aplicação *web* para criar e gerenciar API's e um *Web Service* para acessar as API's via arquitetura REST.

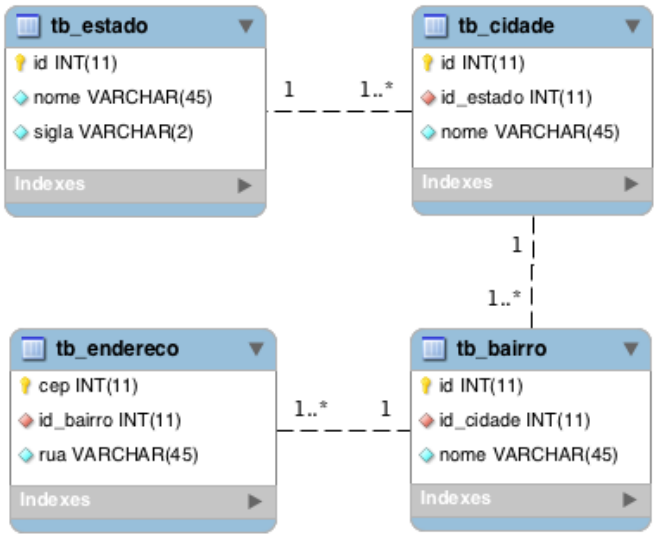
Para entender o objetivo da aplicação criada utiliza-se o ponto de vista de uma loja virtual que atende seus clientes através do envio de produtos pelo correio. Em algum momento da compra o cliente terá que informar o endereço da entrega. Nesse momento o Código de Endereçamento Postal (CEP) exerce um papel fundamental, já que ele indica o estado, a cidade, o bairro e a rua do destinatário. Somente com a posse desses elementos é que a loja virtual pode calcular o valor do frete a ser pago para o envio do produto.

Uma solução para a loja seria armazenar na sua base de dados todos os CEPs brasileiros, porém uma solução melhor seria utilizar uma API externa que recebesse uma mensagem contendo o valor de um CEP e retornasse uma outra mensagem com os dados referentes ao CEP indicado. A aplicação da loja virtual teria que enviar uma requisição com o CEP do cliente e receberia uma resposta do *Web Service* com as informações necessárias para poder calcular o frete.

Contudo, existe um outro problema: onde poderia ser encontrado um *Web Service* que disponibilize os dados de CEP ou qualquer outro assunto. É justamente nesse ponto que entra o sistema gerador de API, com ele qualquer pessoa que tiver um banco de dados poderá disponibilizar a outros ou se preferir mante-lo privado e utilizar a aplicação somente para facilitar o processo de criação de um *Web Service* REST.

Ainda seguindo o exemplo do *web service* de CEP, para exemplificar o funcionamento do sistema, neste capítulo será criado uma API para consulta de CEP, o SGBD escolhido no exemplo foi o MySql, o qual contém um banco de dados com uma estrutura de 4 tabelas interligadas (*tb_estado*, *tb_cidade*, *tb_bairro* e *tb_endereco*) que podem ser vista na Figura 17.

Figura 17: Banco de dados CEP.



Fonte: o autor.

Para cada tabela a ferramenta criará um recurso REST, porém se o cliente precisar do endereço completo (estado, cidade, bairro, rua e CEP) terá que realizar 4 requisições e isso não é viável, para solucionar o problema será criada uma visão no banco de dados para retornar as informações em uma única requisição. O código SQL que foi utilizado para a construção pode ser visto na figura 18 .

Figura 18: Visão CEP.

```
CREATE VIEW view_cep AS
SELECT `End`.`cep`, `End`.`endereco`, `End`.`id_bairro`, `Bar`.`nome` AS bairro, `Bar`.`id_cidade`,
       `Cid`.`nome` AS cidade, `Cid`.`id_estado`, `Est`.`nome` AS estado, `Est`.`sigla` AS sigla_estado
FROM `tb_endereco` AS `End`
JOIN `tb_bairro` AS `Bar` ON `Bar`.`id` = `End`.`id_bairro`
JOIN `tb_cidade` AS `Cid` ON `Cid`.`id` = `Bar`.`id_cidade`
JOIN `tb_estado` AS `Est` ON `Est`.`id` = `Cid`.`id_estado`;
```

Fonte: o autor.

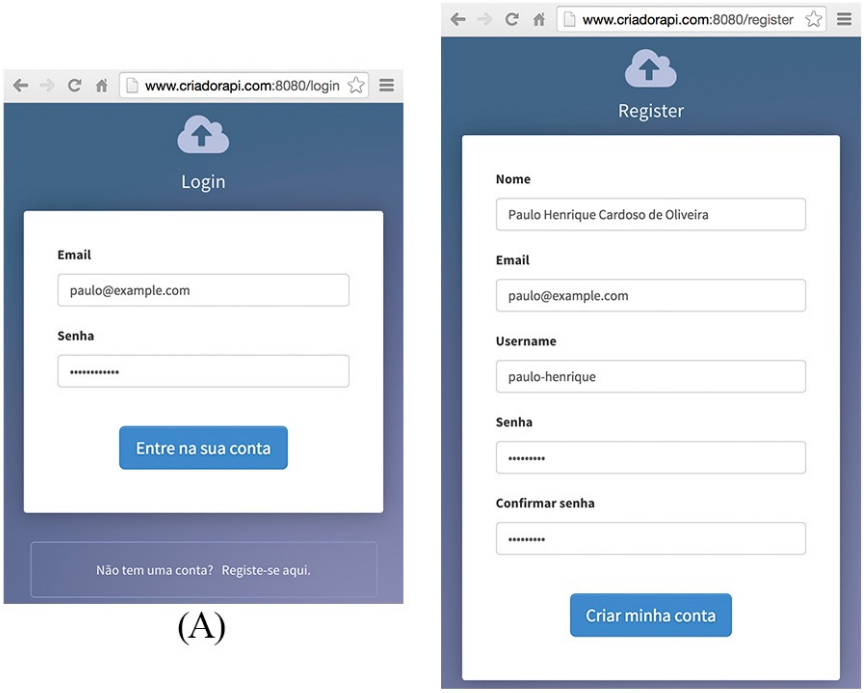
4.1. Criando e gerenciando uma conta no sistema

Login e Cadastro:

Para acessar a aplicação, caso já possua uma conta no sistema, o usuário deverá acessar a URL http://www.criadorapi.com:8080/login e realizar o login (Figura 19).

Se não possuir uma conta deverá então acessar a URL http://www.criadorapi.com:8080/register e criar uma conta informando seus dados (Figura 19).

Figura 19: (A) Login; (B) Cadastro.

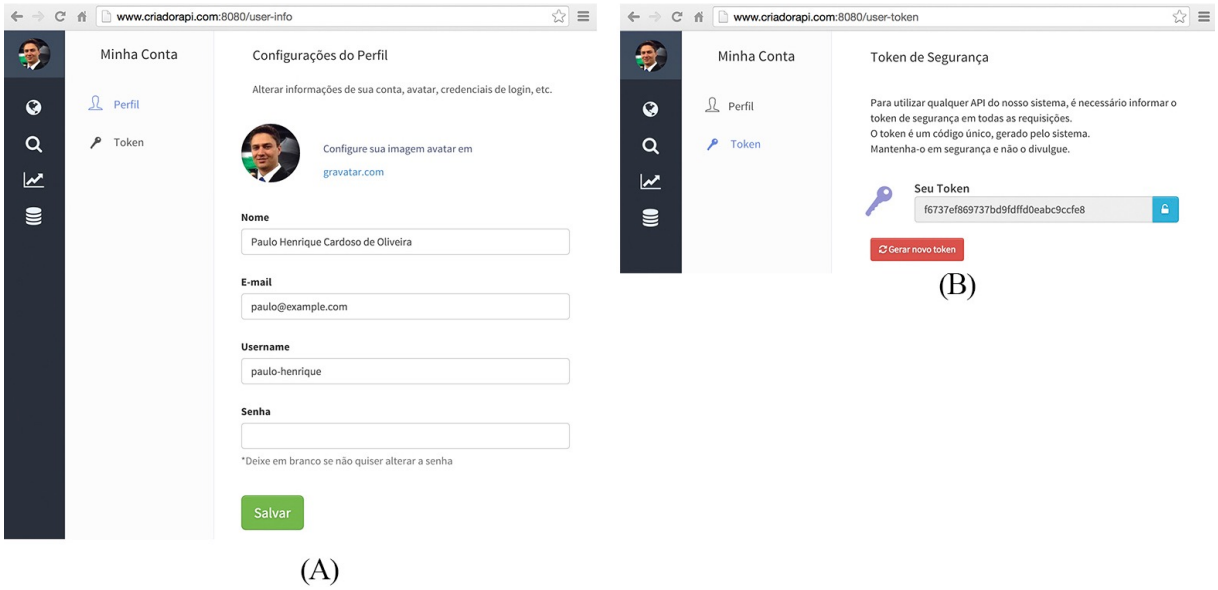


Fonte: o autor.

Editar perfil e *token*:

O usuário tem a possibilidade de alterar as configurações do seu perfil e visualizar ou gerar um novo *token* de segurança no sistema (Figura 20).

Figura 20: (A) Alterando configurações do perfil; (B) *Token* de segurança.



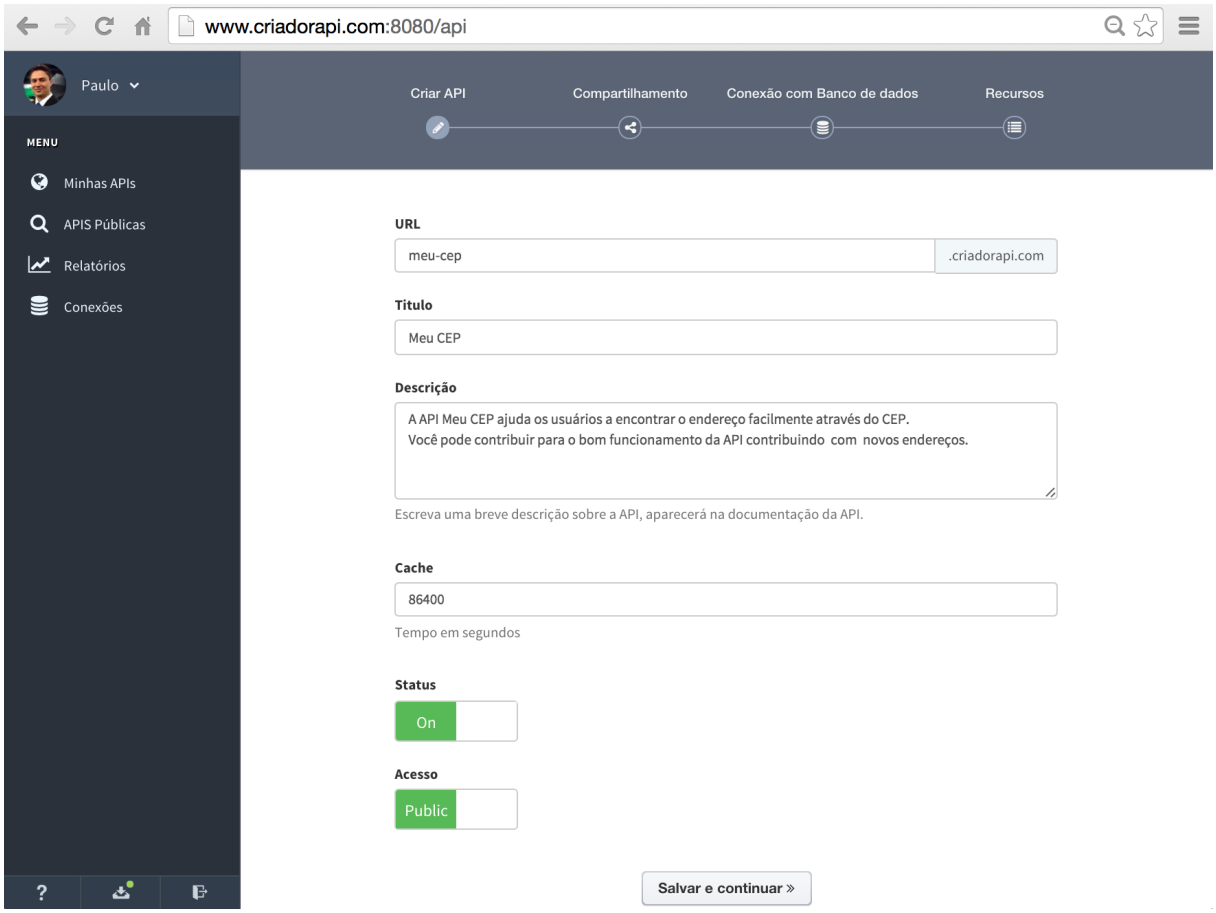
Fonte: o autor.

4.2. Criando e gerenciando uma API

Criando uma API:

Após o usuário se autenticar no sistema ele poderá criar sua API, informando os dados necessários conforme a Figura 21.

Figura 21: Criando um API.



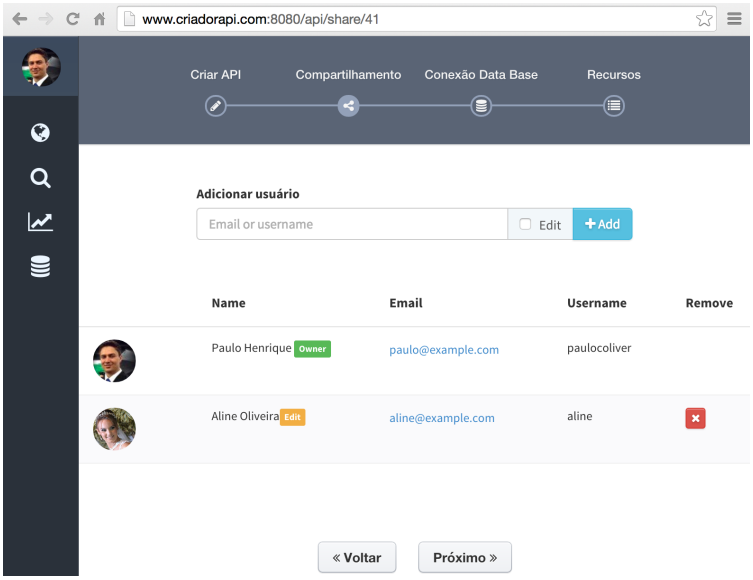
Fonte: o autor.

Compartilhando uma API:

Depois de criada, a API poderá ser compartilhada com outros usuários e definir a permissão que terão. Na Figura 22 pode ser visto que existem dois usuários, o primeiro usuário (Paulo Henrique) com permissão *Owner* possui todas as permissões, o segundo usuário (Aline Oliveira) possui permissão *Edit* e pode ajudar a gerenciar a API.

Na Figura 22 também pode ser visto como o *layout* da aplicação é exibido caso seja acessado em um *Tablet*.

Figura 22: Compartilhando uma API.

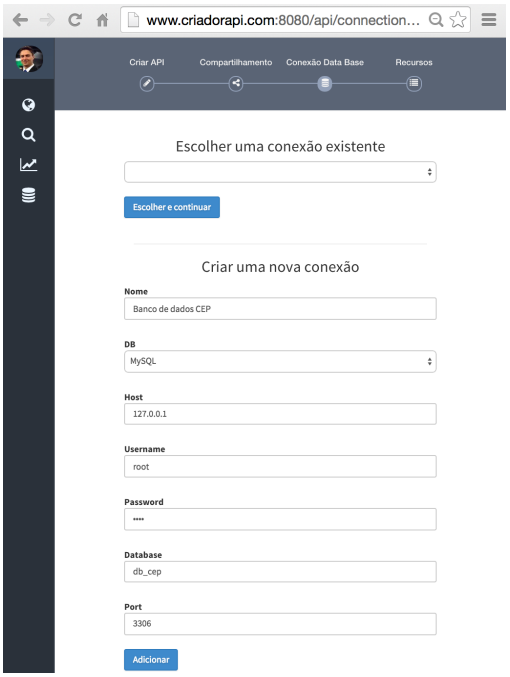


Fonte: o autor.

Configurando o banco de dados da API:

O próximo passo é configurar a fonte de dados da API, o qual pode ser visto na Figura 23, onde pode ser criada uma conexão com um banco de dados externo ou escolher uma conexão cadastrada anteriormente pelo usuário. Se a aplicação não conseguir estabelecer uma conexão com o banco de dados informado, uma mensagem de erro será apresentada na tela.

Figura 23: Configurando o banco de dados da API.

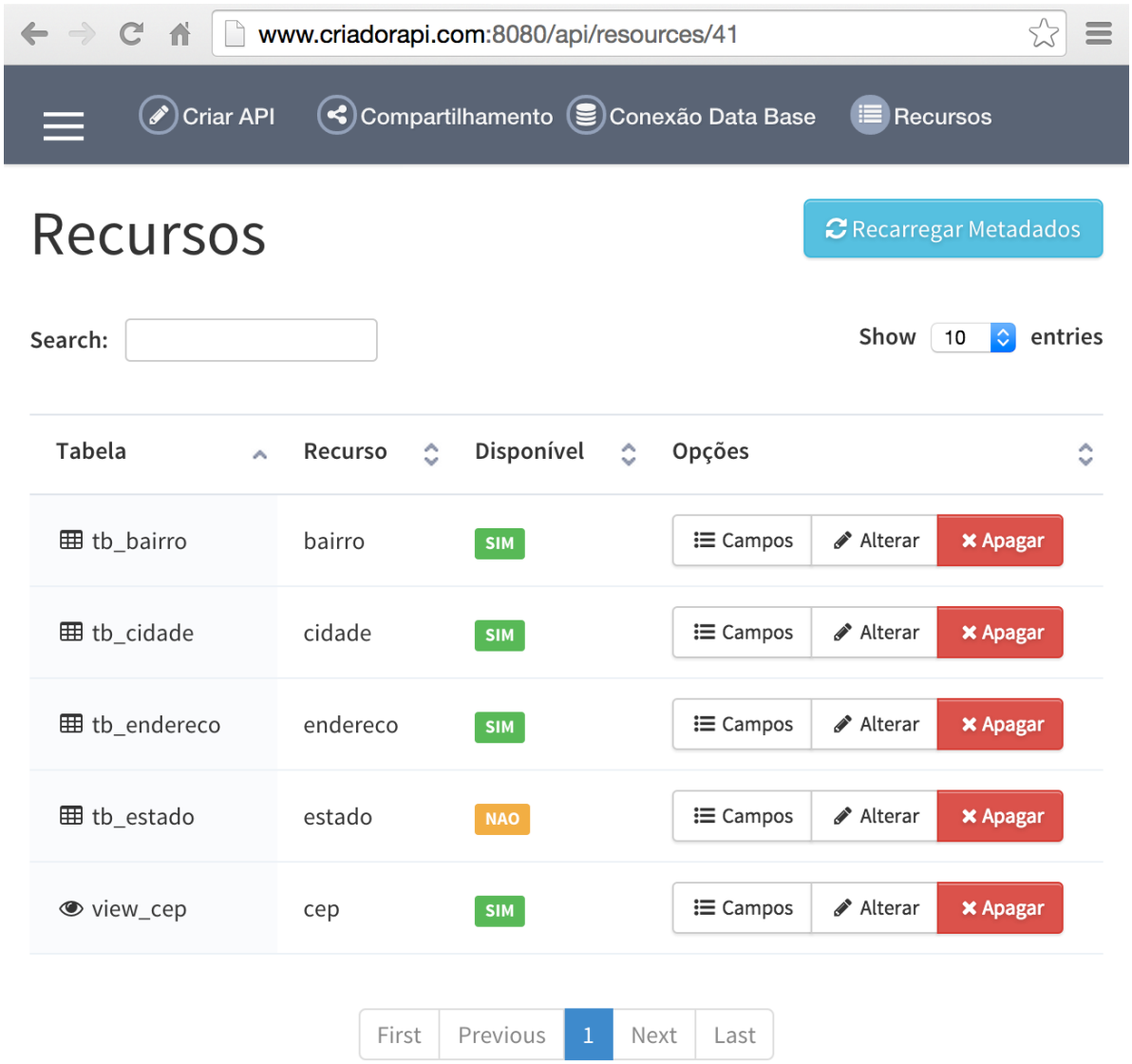


Fonte: o autor.

Gerenciando os recursos da API:

Após estabelecida a conexão com a fonte de dados, é realizado o processo de mapeamento da base de dados e na Figura 24 é possível ver o resultado do mapeamento do banco de dados da Figura 17 (tabelas) e Figura 18 (visão). Caso seja necessário realizar um novo mapeamento pode ser feito através do botão "Recarregar Metadados" (Figura 24).

Figura 24: Gerenciando os recursos da API.



Fonte: o autor.

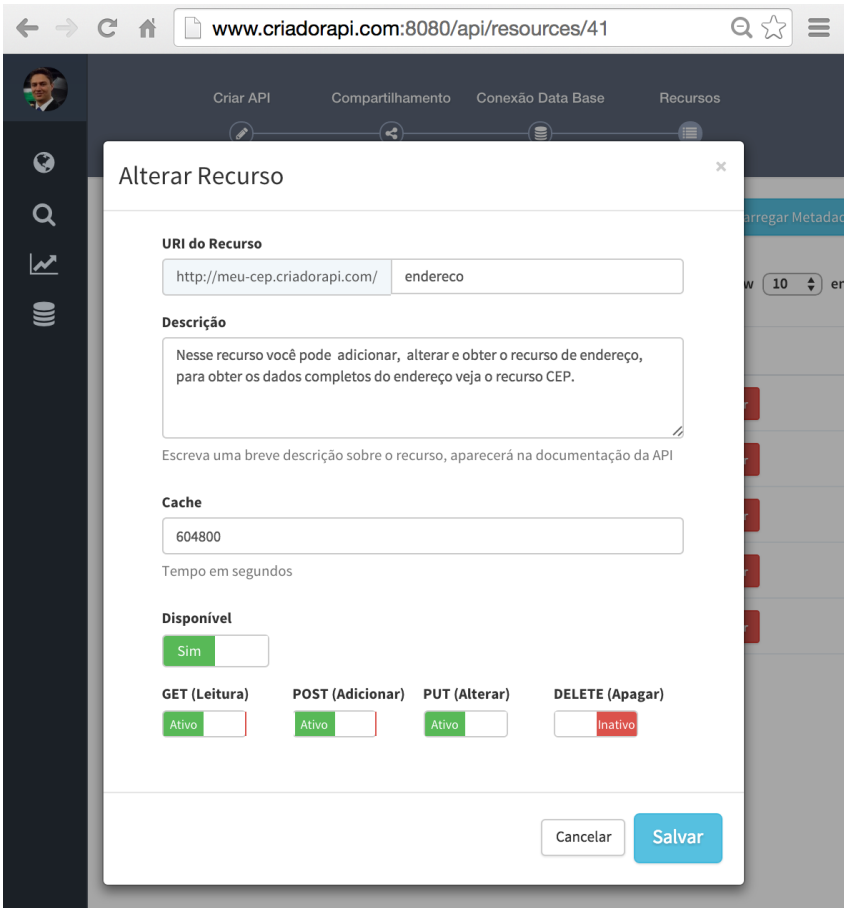
Na Figura 24 também pode ser visto como o *layout* da aplicação é exibido caso seja acessado em um *Smartphone*. As URIs de acesso dos recursos foram alterados, no caso foram removidos os prefixos *tb_* e *view_* dos recursos, isso foi feito para facilitar o uso dos recursos.

Além disso o recurso estado foi marcado como indisponível pelo proprietário da API e não poderá ser consumido.

Alterando um recurso:

Para cada recurso pode ser definido se o mesmo estará disponível e as ações que poderão ser realizados. Na Figura 25 pode ser visto que o recurso endereço está disponível e que a ação de Apagar não será permitido no mesmo, além disso foi definido um tempo de *cache* de 604800 segundos (uma semana), o qual sobrescreve o valor definido na criação da API (Figura 21).

Figura 25: Alterando um Recurso.

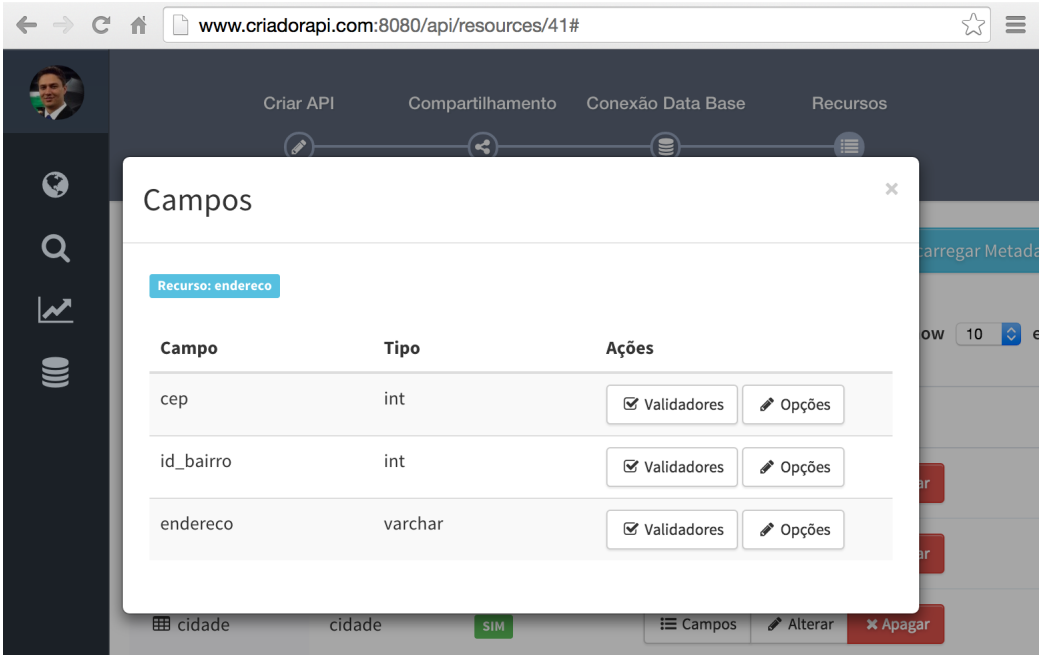


Fonte: o autor.

Gerenciando os campos de um recurso:

Na Figura 24 é possível ver que para cada recurso existe um botão para gerenciar os campos do mesmo. E na Figura 26 o recurso endereço foi escolhido para ser gerenciado, a qual exibe os campos disponíveis e o tipo do campo.

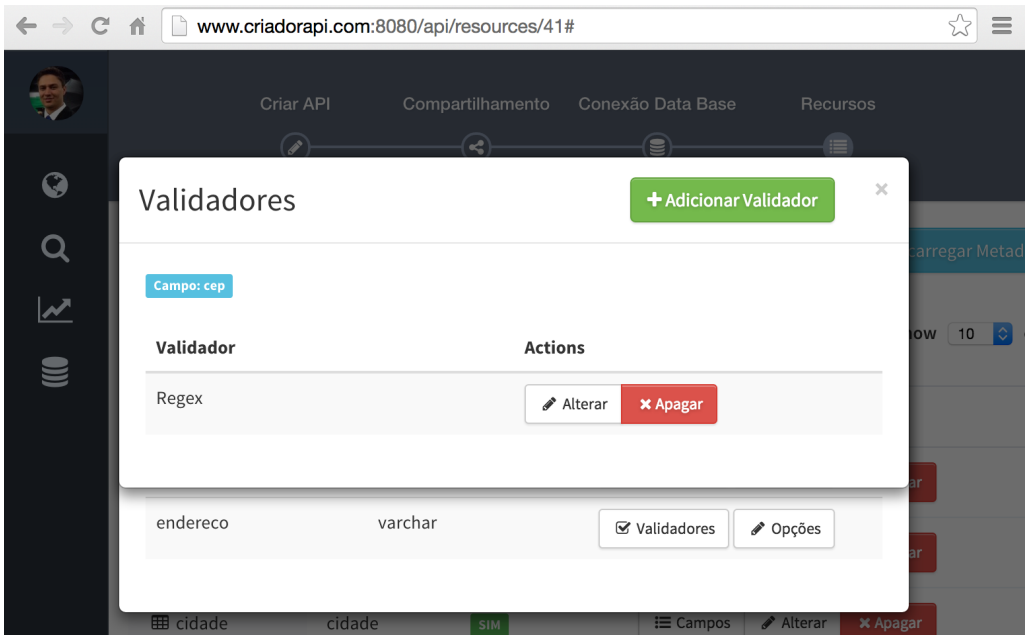
Figura 26: Gerenciando os campos de um recurso.



Fonte: o autor.

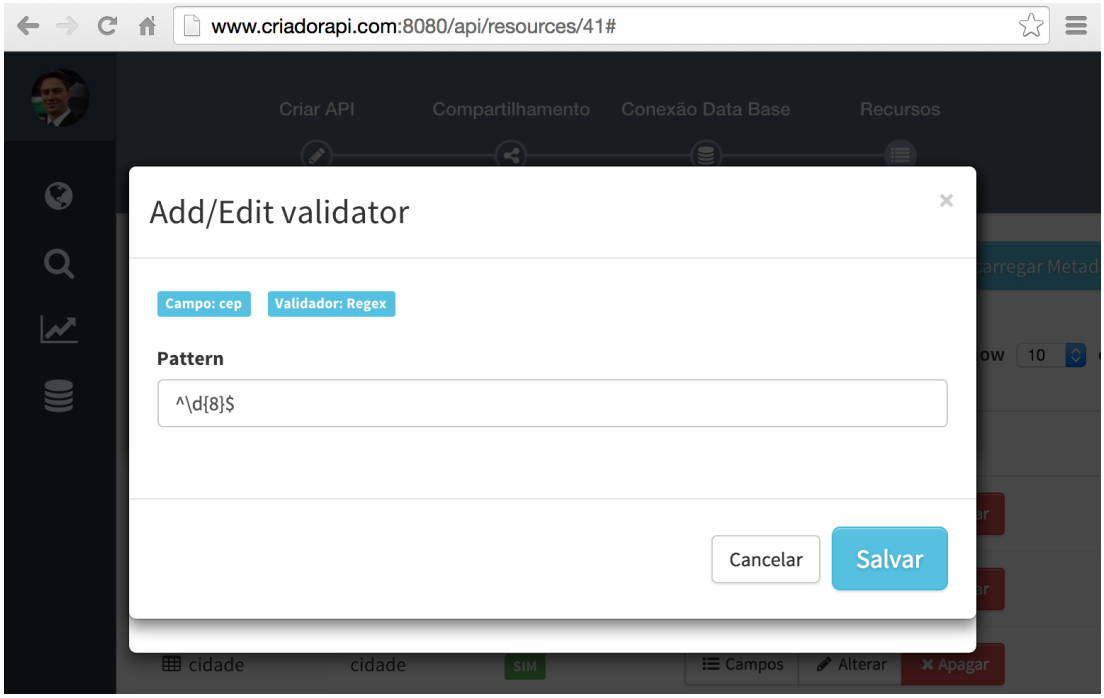
O botão Validadores (Figura 26) exibe a lista de validadores contidos no campo (Figura 27). No campo CEP foi adicionado um validador (Figura 28) de expressão regular (*Regex*) com o padrão */(^\d{8}$)/*, o qual valida se o valor informado é um número de 8 dígitos.

Figura 27: Lista de validadores no campo de um recurso.



Fonte: o autor.

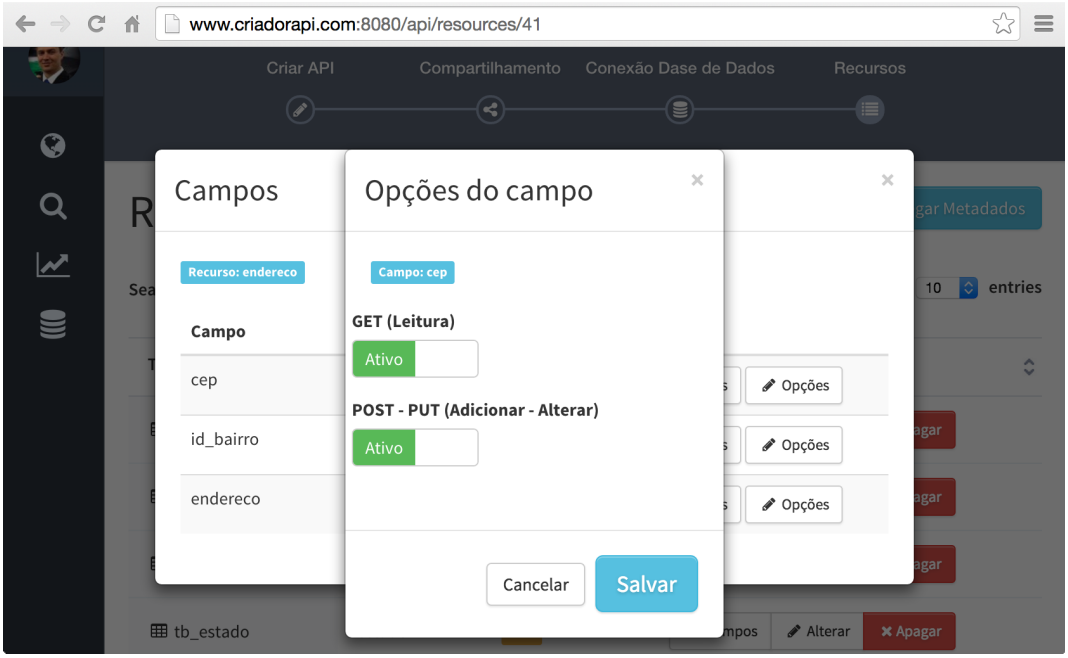
Figura 28: Adicionando um validador.



Fonte: o autor.

Ainda na figura (Figura 26) pode-se ver o botão opções, com o qual é possível definir se o campo estará disponível para requisições de leitura (*GET*) ou requisições escrita (*POST* e *PUT*) (Figura 29).

Figura 29: Opções do campo.



Fonte: o autor.

Lista de API's:

Depois de criada, a API aparece na listagem de API's do usuário (Figura 30), onde o mesmo poderá alterar, apagar ou acessar a documentação de suas API's.

Figura 30: Lista de API's.

Fonte: o autor.

4.3. Pesquisando uma API Pública

Caso a API seja criada como pública, a mesma poderá ser encontrada por outros usuários do sistema através da funcionalidade API's Públicas (Figura 31), os quais poderão visualizar a documentação para conhecer todos os detalhes da API antes de utiliza-lá. Os usuários que decidirem utilizar a API devem clicar no botão Participar, com isso a aplicação concederá permissão de acesso (*ACCESS*) na API.

Figura 31: Pesquisando uma API Pública.

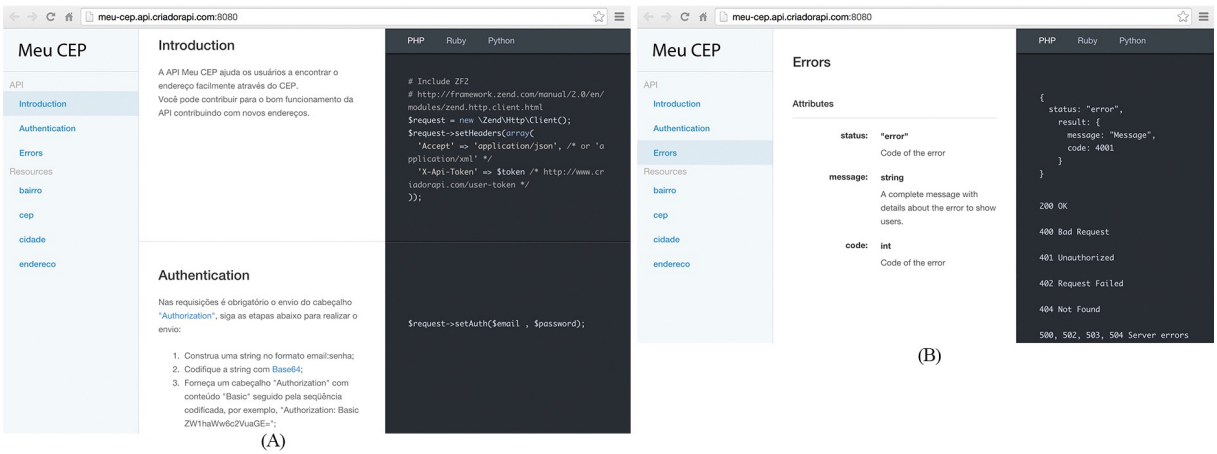
Fonte: o autor.

4.4. Visualizando a documentação de uma API

Para toda API criada no sistema será disponibilizada uma documentação, a qual é composta pelos seguintes itens: introdução, autenticação e erros (Figura 32) e pelos recursos disponíveis na API (Figura 33).

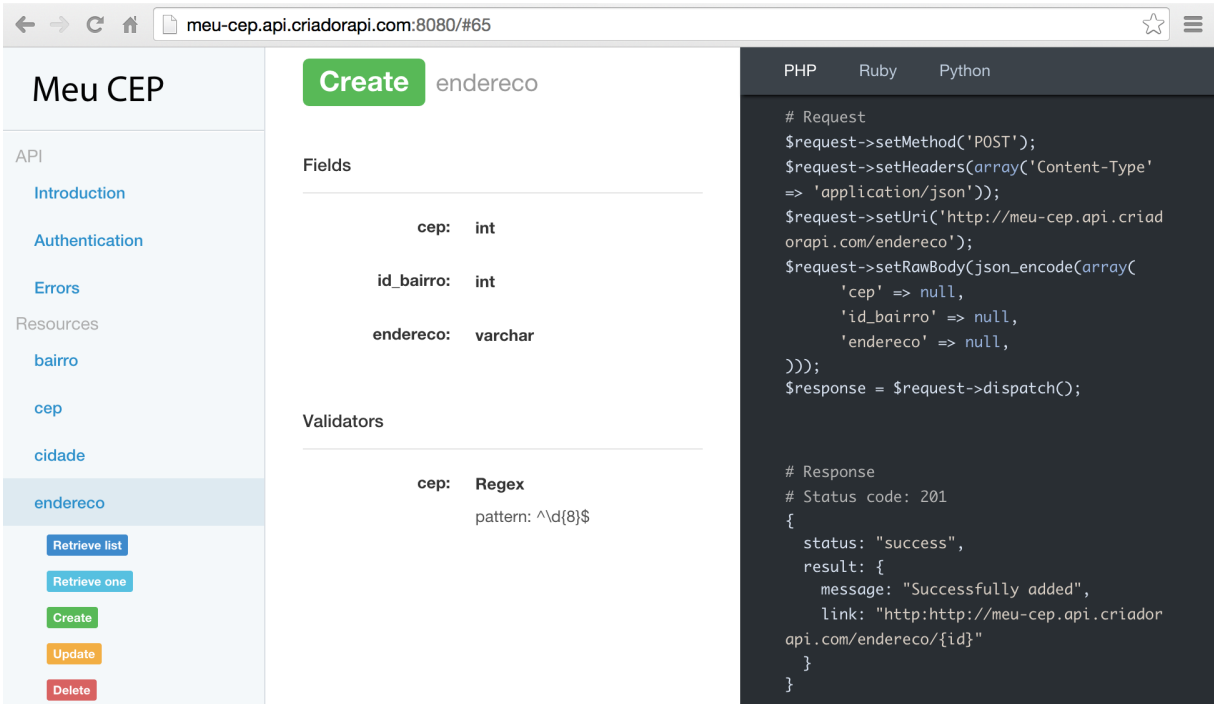
Na Figura 32 pode ser visto a documentação que foi gerada para a API criada na seção 4.3.

Figura 32: (A) Documentação Introdução e autenticação; (B) Documentação erros.



Fonte: o autor.

Figura 33: Documentação dos recursos.



Fonte: o autor.

4.5. Consumindo os recursos de uma API

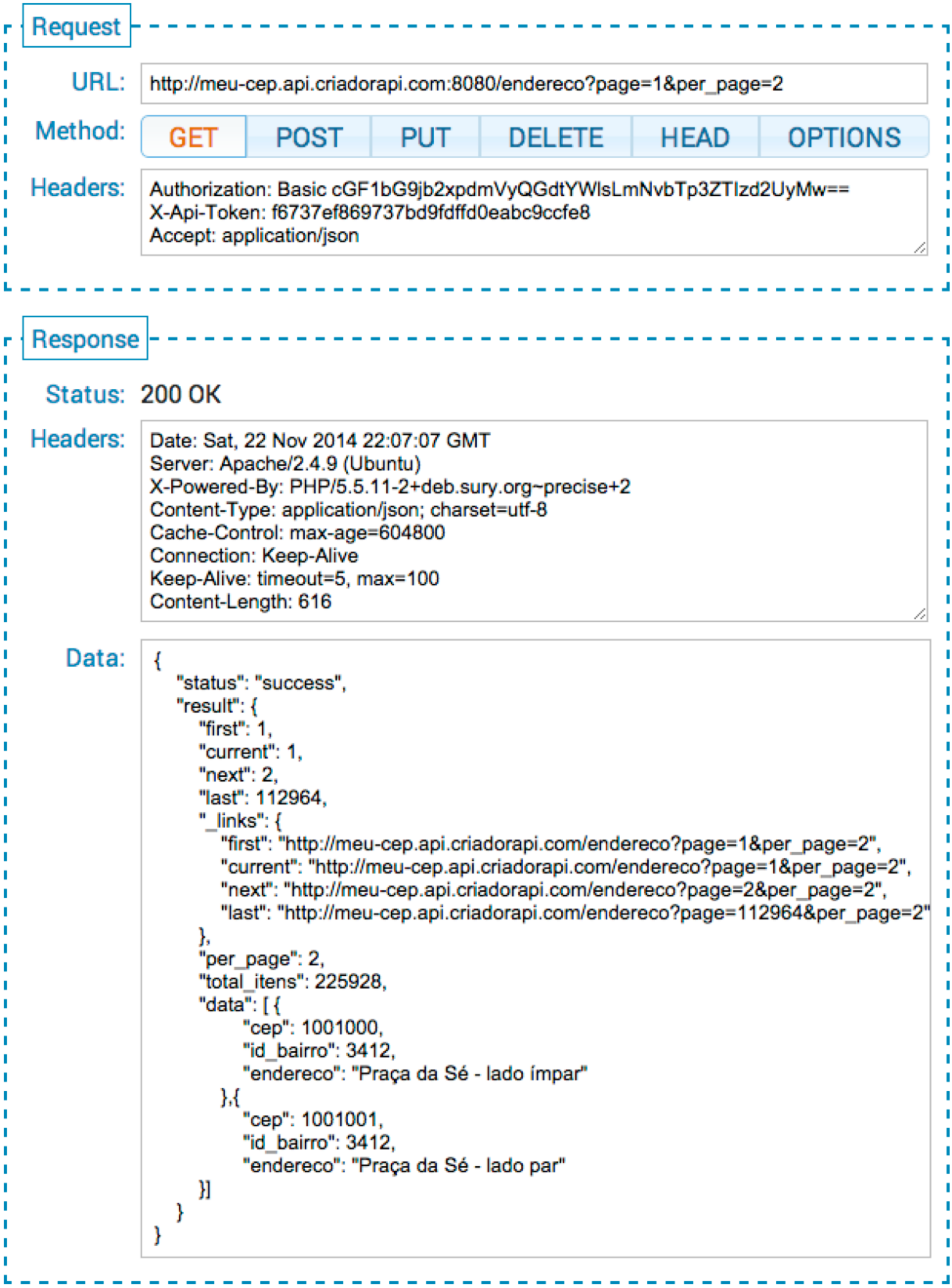
Nesta seção foram realizados exemplos práticos para demostrar a utilização da API criada na seção 4.2. A ferramenta utilizada para gerar os testes foi a extensão "*SureUtils REST API Client*" do Google Chrome.

Leitura dados na API:

A seguir será demostrado o funcionamento de requisições com o método HTTP *GET* na API. A Figura 34 ilustra a requisição ao recurso "endereco", o qual retorna uma coleção de dados paginados. Na Figura 35 a requisição é feita no recurso "cep" buscando o documento 17522710.

Em ambos os casos pode-se ver a presença do cabeçalho *Cache-Control* na resposta o qual indica o tempo em segundos que o cliente deve armazenar as informações. Outros dois cabeçalhos importantes que estão presentes são o *Accept* na requisição e o *Content-Type* na resposta, que indica que o cliente solicitou o conteúdo da resposta em formato JSON.

Figura 34: Leitura de uma coleção dados na API;



Fonte: o autor.

Figura 35: Leitura de dados com identificador único.

Fonte: o autor.

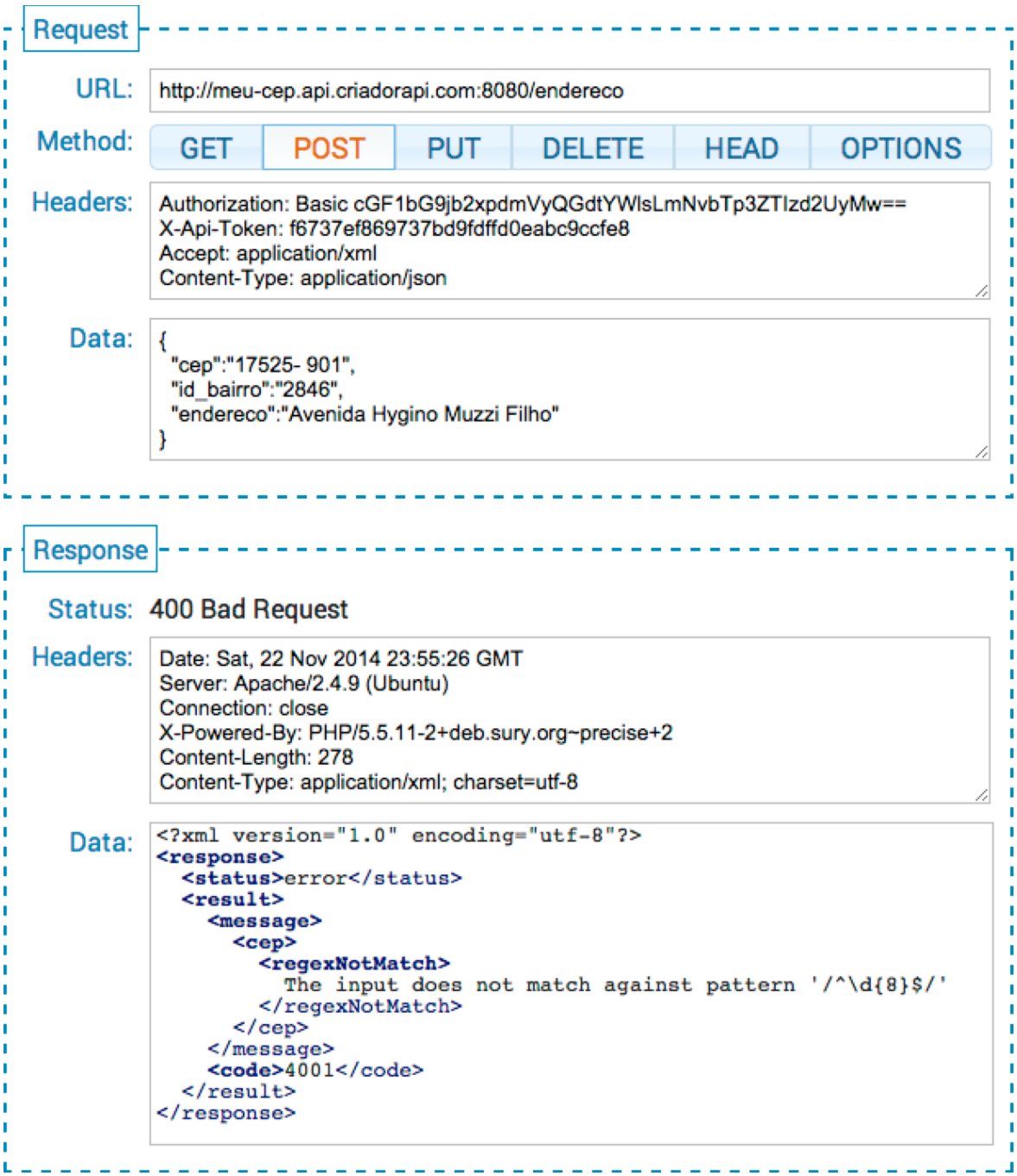
Inserção de dados na API:

A seguir será demostrado o funcionamento de requisições com o método HTTP *POST* na API. A Figura 36 ilustra uma tentativa de inclusão de dados no recurso “endereco”, porém o campo CEP contém um validador *Regex* que verifica que o valor informado está inválido, causando um erro 400 (*Bad Request*). Na Figura 37 o campo CEP está com o valor

correto, com isso os dados são inseridos com sucesso no recurso e é retornado o *status code* 201 (*Created*).

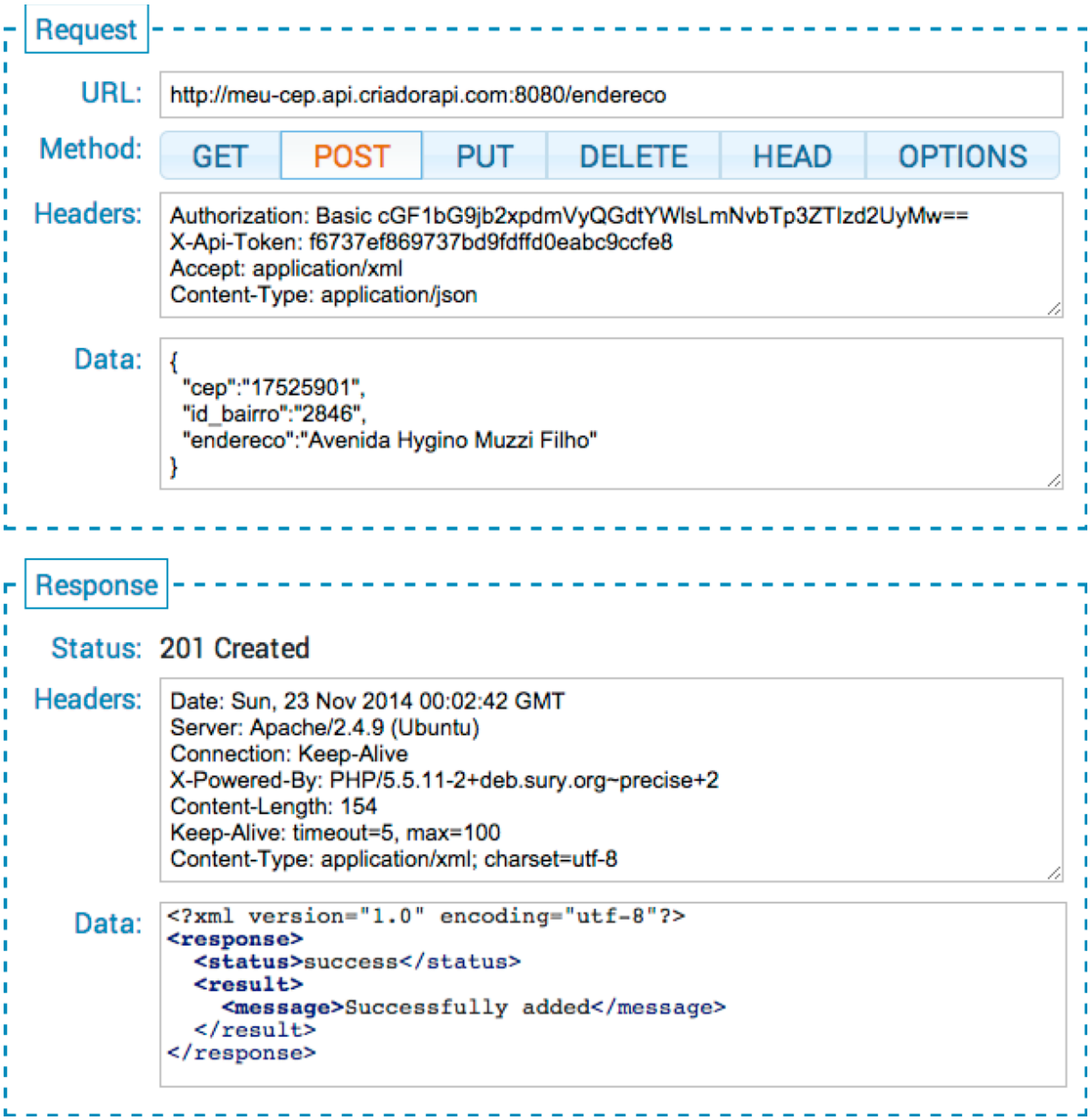
Em ambos os casos pode-se ver através do cabeçalho *Content-Type* da requisição que os dados são enviados no formato JSON, porém a resposta é aguardada no formato XML que pode ser visto através da presença do cabeçalho *Accept* na requisição e o do *Content-Type* da resposta.

Figura 36: Erro na inserção de um Recurso;



Fonte: o autor.

Figura 37: Sucesso na inserção de um Recurso.

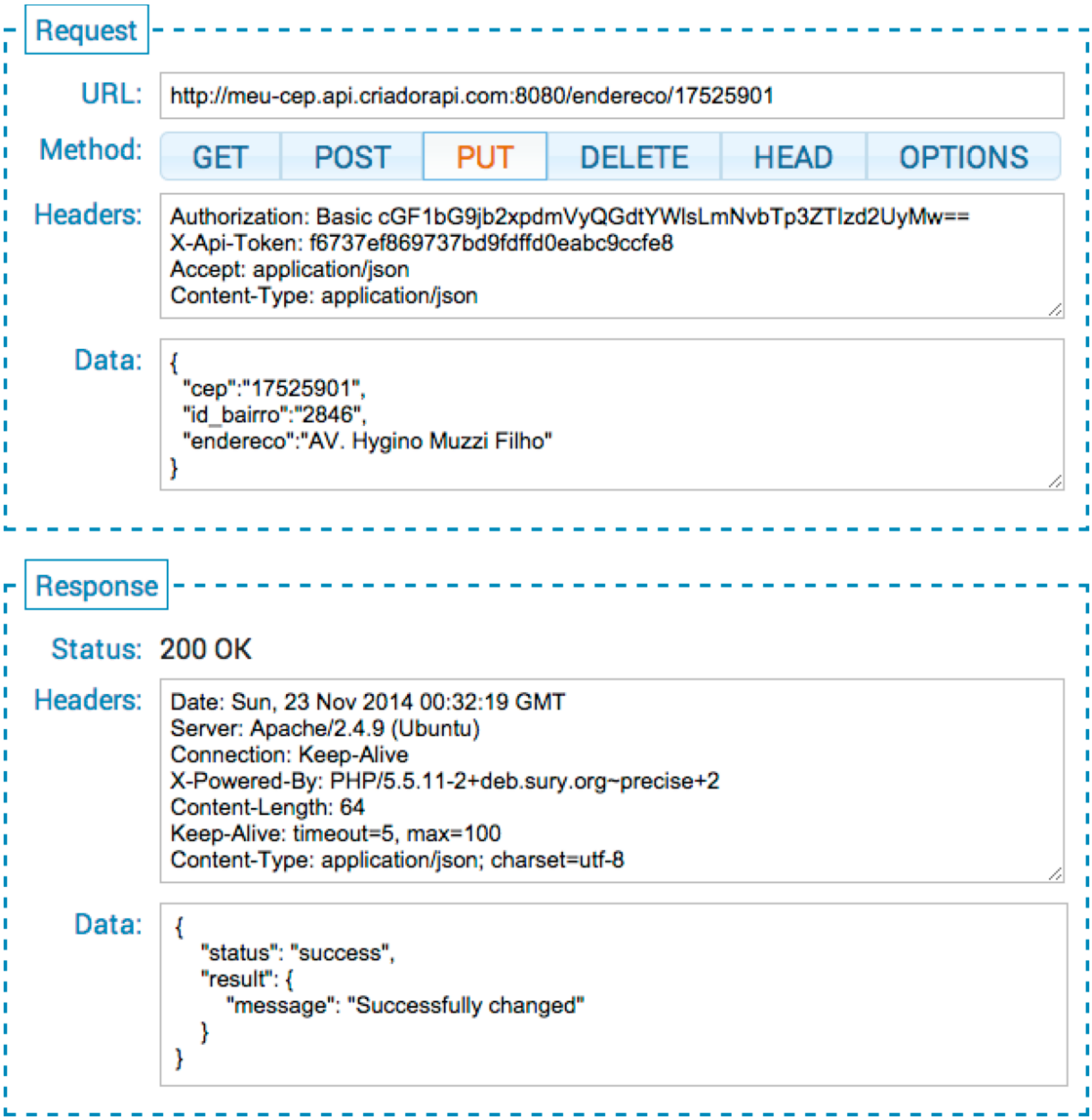


Fonte: o autor.

Alteração de dados na API:

A Figura 38 ilustra o funcionamento de requisições com o método HTTP *PUT* na API. No exemplo da figura a tentativa de alteração dos dados do recurso endereço obteve sucesso, com isso os dados foram alterados e foi retornado o *status code* 200 (*OK*).

Figura 38: Alteração de dados na API.

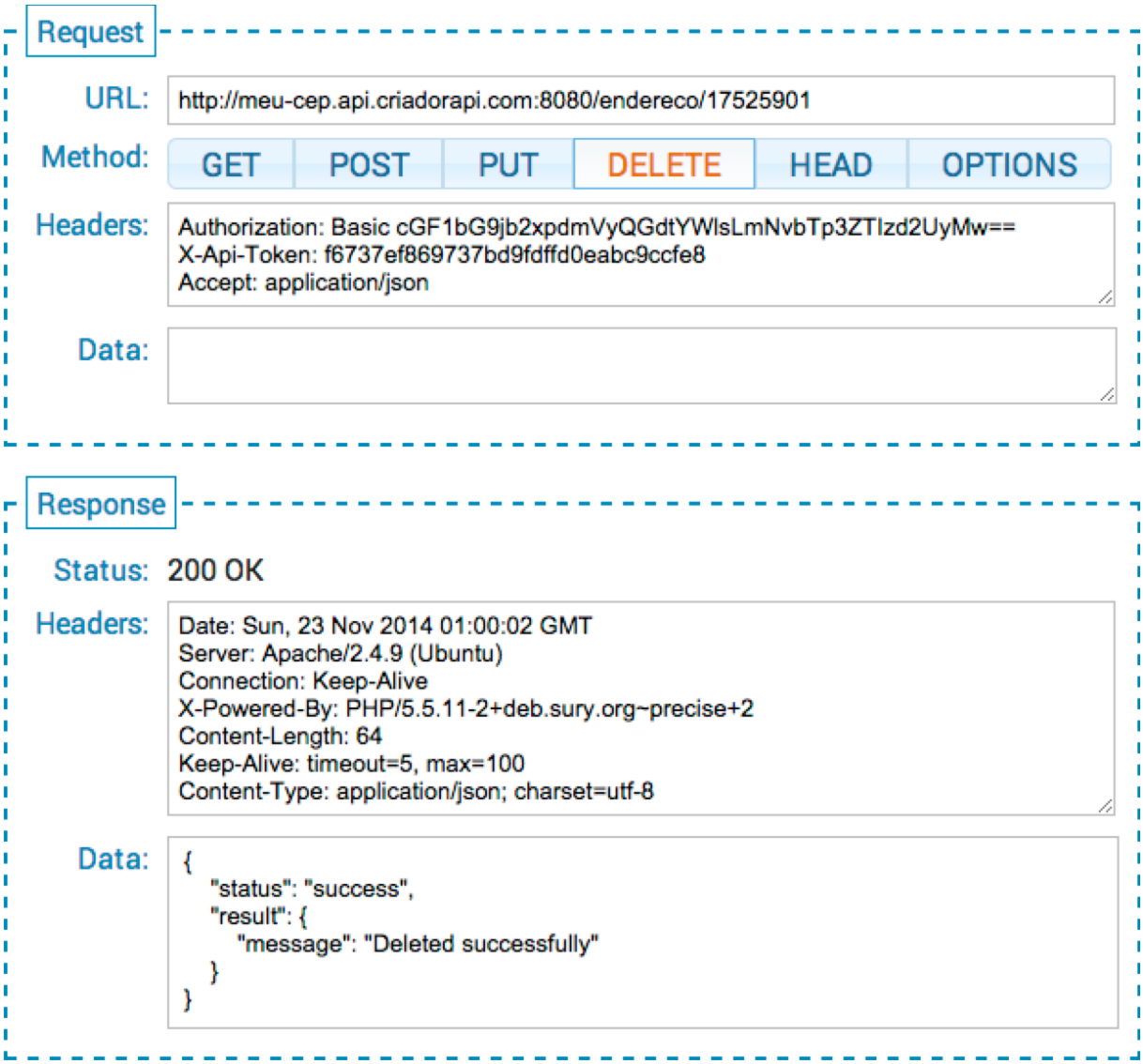


Fonte: o autor.

Exclusão de dados na API:

A Figura 39 ilustra o funcionamento de requisições com o método HTTP *DELETE* na API. No exemplo da figura a tentativa de exclusão dos dados do recurso endereço obteve sucesso, com isso os dados foram apagados e foi retornado o *status code* 200 (*OK*).

Figura 39: Exclusão de dados na API.



Fonte: o autor.

4.6.Considerações finais

Neste capítulo foi apresentados os resultados do trabalho desenvolvido, juntamente com um exemplo completo de criação, gerenciamento e o consumo dos recursos de uma API criada no sistema gerenciador de API.

CONCLUSÃO

Neste trabalho foi desenvolvida uma aplicação para a criação automática e gerenciável de *Web Service*s na arquitetura REST a partir de uma base de dados relacional. Os objetivos desse trabalho foram atingidos, os quais eram: desenvolver uma aplicação web para criar e gerenciar API's e um *Web Service* para acessar as API's via arquitetura REST.

A criação desse sistema possibilita que a geração de API's seja realizada em um período de tempo mais curto, com baixo custo para ser implementado, facilidade na manutenção e segurança das informações trafegadas.

Esta experiência proporcionou aprendizado e ampliação de habilidades técnicas e cognitivas relacionadas as seguintes áreas: desenvolvimento *web*, arquitetura REST, *Web Services*.

Este trabalho poderá receber novas funcionalidades para deixar o software mais completo. Das funcionalidades futuras, foi possível identificar:

- Gerar remuneração: Adicionar a opção para os donos das API's cobrarem pelo uso de seus dados, a partir de *n* requisições na API o serviço é interrompido e só voltará a funcionar através da adição de créditos;
- Filtros nas requisições de leitura de dados: Adicionar uma opção para ser realizado filtro das informações nas requisições de leitura de dados, o qual poderia ser feito através de *query parameters* na URL. Com isso diminuiria o consumo desnecessário de dados;
- Duplicar API's: Para facilitar a criação de versões da mesma API.

Esta ferramenta poderá ser disponibilizada na internet para testes funcionais e, após avaliações dos usuários e aprimoramentos do sistema, será liberada para uso efetivo.

REFERÊNCIAS

About MySQL. Disponível em: <http://www.mysql.com/about>. Acesso em: 10 novembro 2014.

About Zend Framework 2. Disponível em: <http://framework.zend.com/about>. Acesso em: 10 novembro 2014.

Bootstrap front-end framework. Disponível em: <http://getbootstrap.com>. Acesso em: 10 novembro 2014.

CIRIACO, Douglas. **O que é API**: 24 de março de 2009. Disponível em: <http://www.tecmundo.com.br/programacao/1807-o-que-e-api-.htm>. Acesso em: 10 maio 2014.

EMER, Jean Carlo. **O grande desencontro do HTTP com o HTML**: 6 de janeiro de 2014. Disponível em: <http://tableless.com.br/o-grande-desencontro-http-com-o-html/>. Acesso em: 10 agosto 2014.

ERL, T; PUTTINI, R; MAHMOOD, Z. **Cloud Computing: Concepts, Technology & Architecture.** Prentice Hall, 2013.

FIELDING, Roy Thomas. **Architectural Styles and the Design of Network-based Software Architectures**. UNIVERSITY OF CALIFORNIA, IRVINE, 2000.

FOWLER, Martin; SCOTT, Kendall. **UML Distilled: A Brief Guide to the Standard Object Modeling Language (2nd Edition).** Addison-Wesley, 1999.

Getting Started with Doctrine. Disponível em: <http://doctrine-orm.readthedocs.org/en/latest/tutorials/getting-started.html >. Acesso em: 10 novembro 2014.

jQuery. Disponível em: <http://jquery.com>. Acesso em: 10 novembro 2014.

MEDEIROS, Higor. **Application Programming Interface: Desenvolvendo APIs de Software**: 12 de março de 2012. Disponível em: < http://www.devmedia.com.br/application-programming-interface-desenvolvendo-apis-de-software/30548/>. Acesso em: 24 agosto 2014.

MORO, Tharcis D.; DORNELES, C.; REBONATTO, M. T. **Web services WS-* versus Web Services REST**. Instituto de Ciências Exatas e Geociências, Universidade de Passo Fundo (UPF), 2011.

Online REST Web Service Demo. Disponível em: <http://www.predic8.com/rest-demo.htm>. Acesso em: 10 dezembro 2014.

Pjax. Disponível em: <http://pjax.herokuapp.com>. Acesso em: 10 novembro 2014.

restSQL Overview. Disponível em: <http://restsql.org/doc/Overview.html>. Acesso em: 10 dezembro 2014.

RICHARDSON, Leonard; RUBY, Sam. **RESTful Web Services.** O'Reilly, 2007.

RONDON, Thiago. **Arquitetura REST e o Serviço Web 'RESTful'.** 2010. Disponível em: <http://sao-paulo.pm.org/artigo/2010/RESTful>. Acessado em: maio de 2014.

SAUDATE, Alexandre. **REST: Construa API's inteligentes de maneira simples.** Casa do código, 2013.

SAUDATE, Alexandre. **SOA Aplicado: Integrando com web services e além.** Casa do código, 2013.

SlashDB. Disponível em: <http://demo.slashdb.com/index.html>. Acesso em: 10 dezembro 2014.

STANDARD Validation Classes Zend Framework 2. Disponível em: <http://framework.zend.com/manual/2.0/en/modules/zend.validator.html>. Acesso em: 10 novembro 2014.

The Apache HTTP Server Project. Disponível em: <http://httpd.apache.org>. Acesso em: 10 novembro 2014.

Understanding Providing WSDL Documents. Disponível em: <http://docs.oracle.com/cd/E38689_01/pt853pbr0/eng/pt/tibr/concept_UnderstandingProvidingWSDLDocuments-076201.html>. Acesso em: 7 dezembro 2014.

W3C. **Simple Object Access Protocol (SOAP) 1.1**: 08 de Maio de 2000. Disponível em: <http://www.w3.org/TR/2000/NOTE-SOAP-20000508/>. Acesso em: 20 julho 2014.

W3C. **Web Services Architecture**: 11 de fevereiro de 2004. Disponível em: <http://www.w3.org/TR/ws-arch/ >. Acesso em: 20 julho 2014.