**Projeto JavaDeveloper**

***(WebServices Rest)***

**Índice**

1. Introdução 3

1.1. Propósito 3

1.2. Escopo 3

1.3. Definição do Projeto 3

1.4. Referências 3

2. Arquitetura Padrão 4

2.1. Camada de Apresentação 4

2.2. Camada de Negócios 4

2.3. Camada de Persistência / Acesso a Banco de Dados 4

2.4. LOG e Mecanismos de rastreio 6

2.4.1. LOG4J 6

3. Visão Lógica 7

3.1. Pacotes 7

3.2. Padrão de Nomenclatura para Classes 7

3.3. Versionamento 7

4. Arquitetura Visual Macro do Projeto 8

5. Plataforma de Desenvolvimento 9

5.1. Versões Recomendadas 9

6. Usabilidade 10

6.1. Interface de Usuário 10

7. Testes - cenários desenvolvidos 12

7.1. Cenários para a questão 1 – Pesquisa Cep 12

7.1.1. Pesquisando um cep existente: 12

7.1.2. Pesquisando um endereço cep cadastrado: 13

7.1.3. Pesquisando um cep no formato inválido: 13

7.1.4. Tentando localizar o CEP aproximado 14

7.2. Cenários para a questão 2 – CRUD Endereço 14

7.2.1. Exemplo Inserir Endereço 14

7.2.1.1. Inserir Endereço com sucesso 15

7.2.1.2. Inserir Endereço com CEP não localizado sucesso 16

7.2.1.3. Inserir Endereço com CEP não localizado sucesso 16

7.2.2. Exemplo Alterar Endereço 17

7.2.2.1. Alterar Endereço com sucesso 17

7.2.2.2. Alterar Endereço passando um id inexistente 18

7.2.3. Pesquisa de Endereço 18

7.2.3.1. Pesquisando um Endereço com sucesso 19

7.2.3.2. Pesquisando um Endereço inexistente 19

7.2.4. Exemplo Deletar Endereço 20

7.2.4.1. Deletar Endereço com sucesso 20

7.2.4.2. Deletar um Endereço inexistente 21

7.3. Cenários para a questão 3 – Implementação Stream e sua busca da primeira ocorrencia de um caracter que não se repete 22

7.4. Cenários para a questão 4 – Oque ocorre no processo de protocolo http entre Client e Server ao solicitar uma url 24

# Introdução

## Propósito

Esse documento fornece uma visão arquitetural compreensiva do sistema, usando-se de visões que descrevem diferentes aspectos da arquitetura, além de descrever as decisões arquiteturais significativas que foram tomadas para o sistema e apresenta os testes realizados.

## Escopo

Este documento apresenta definições de arquitetura para o sistema JavaDeveloper. Deverá ser um guia de referência para desenvolvedores, projetistas e analistas do projeto, bem como, ser uma das bases de conhecimento deste projeto.

Este documento detalha os testes realizados no sistema e os diversos cenários existentes.

## Definição do Projeto

Projeto constitui em efetuar consultas de CEP e a manutenção de endereços através de serviços disponibilizados por WebService Rest com comunicação através de json. O projeto também possui uma função para detectar qual a primeiro caráter de uma String que não se repete, o qual também estará disponível via Webservice Rest.

Possuo um domínio onde aproveitei para efetuar a publicação da aplicação javaDeveloper.

Seus serviços podem ser acessados na seguinte url: [www.koresystems.com.br/javaDeveloper](http://www.koresystems.com.br/javaDeveloper).

No decorrer do documento, principalmente no item de testes, será apresentado como consumir os WebService Rest.

## Referências

**Java Enterprise Edition 7 -** https://jcp.org/aboutJava/communityprocess/implementations/jsr342

**Rest - EasyRest** - http://resteasy.jboss.org/

https://jcp.org/aboutJava/communityprocess/final/jsr339/index.html

**EJB 3.1 (JSR318)** - https://jcp.org/en/egc/view?id=318

**MyBatis** - https://mybatis.github.io/mybatis-3/

# Arquitetura Padrão

## Camada de Apresentação

Entende-se como camada de apresentação, todo e qualquer artefato de código que é responsável por apresentar informações, interagir e receber entrada de dados do usuário.

A camada de apresentação poderá interagir com todos os subsistemas, portanto faz-se necessário a disponibilidade de interfaces dos subsistemas para que a camada de apresentação não fique fortemente acoplada com os componentes de negócio.

O padrão de implementação para a camada de apresentação foi “API for RESTful Web Services (**JAX-RS)**” (implementação RestEasy presente no servidor WildFly).

Os artefatos para a camada de apresentação são:

* Servlets;
* Filters;
* ExceptionHandlers;
* WebService Rest;
* Json;
* Listeners;

## Camada de Negócios

O objetivo da camada de negócios é implementar a lógica da aplicação, expondo esta lógica para a camada de apresentação ou para outras aplicações clientes remotas.

Para cumprir este objetivo, optou-se pela utilização de Enterprise Java Beans (EJB 3.1).

## Camada de Persistência / Acesso a Banco de Dados

A camada de persistência compreende o grupo de classes responsáveis em persistir dados e informações no banco de dados, como também, recuperá-los de e gerenciá-los de forma eficiente.

O acesso a base de dados, foi feito por meio de queries JDBC. **Sem utilização de frameworks de ORM**, como Hibernate, JPA e similares.

**Framework MyBatis**. Por trabalhar com queries nativas e objetos POJO, o MyBatis tem se mostrado ideal para evitar estouro de memória e sobrecarga no banco de dados.

As classes de acesso ao banco devem seguir o pattern DAO. Onde todas as consultas e instruções de insert, update e delete devem ficar concentradas. Agrupando por tabela, ou conjunto de tabelas de uma mesma funcionalidade ou regra de negócio.

Na ocasiao foi utilizado o banco de dados MySql e na aplicação sua configuração se encontra no arquivo “dbcp.javadeveloper.properties” a atual configuração está apontando para um bando de dados que está na web, deixarei o mesmo liberado por um tempo para que se desejarem possam conectar e manipular as informações, ou simplesmente utilizar uma aplicação local apontando para o mesmo sem ter a necessidade de criar um banco local.

Segue a configuração:

url/banco = 184.107.51.101:3306/koresy\_javaDeveloper

username=koresy\_javaDevel

password=123789

Se preferir criar o banco de dados localmente. Altere as informações do arquivo “dbcp.javadeveloper.properties” para o banco de dados desejado e usar o seguinte script para criar o banco:

create database javaDeveloper;

CREATE TABLE cep

( id INT(11) NOT NULL COMMENT 'id unico'

, cep VARCHAR(8) NOT NULL COMMENT 'cep'

, rua VARCHAR(255) NOT NULL COMMENT 'nome da rua, avenida, locacao'

, cidade VARCHAR(64) NOT NULL COMMENT 'cidade'

, estado VARCHAR(64) NOT NULL COMMENT 'estado'

, bairro VARCHAR(64) NULL COMMENT 'bairro'

, CONSTRAINT PRIMARY KEY ( id )

, CONSTRAINT CS\_UK UNIQUE KEY (cep) );

CREATE TABLE endereco

( id INT(11) NOT NULL COMMENT 'id unico'

, numero VARCHAR(8) NOT NULL COMMENT 'numero do local, podendo conter letra'

, complemento VARCHAR(512) NULL COMMENT 'complemento'

, cepId INT(11) NOT NULL COMMENT 'cepId fk'

, CONSTRAINT PRIMARY KEY ( id ) );

CREATE INDEX cepId1 ON cep ( cepId ASC);

Insert into cep (cep,rua,cidade,estado,bairro) values ('12345678','RUA DE TESTE','Sao Paulo','Sao Paulo','JD DO MAR');

Insert into cep (cep,rua,cidade,estado,bairro) values ('04158050','Doutor Rosalvo de Sales','Sao Paulo','Sao Paulo',null);

Insert into cep (cep,rua,cidade,estado,bairro) values ('00158000','Rua das Figueiras','Santo Andre','Sao Paulo',’Figueiras’);

Insert into cep (cep,rua,cidade,estado,bairro) values ('98765432','RUA DE TESTE','Rio de Janeiro','Rio de Janeiro','Barra');

Insert into cep (cep,rua,cidade,estado,bairro) values ('99900000','RUA TESTE 0','Rio de Janeiro','Rio de Janeiro','Ipanema');

## LOG e Mecanismos de rastreio

### LOG4J

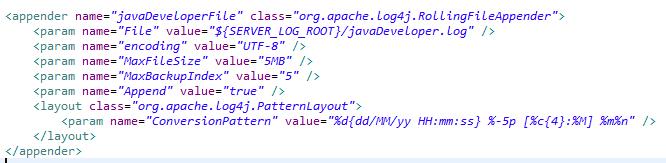
O framework de LOG utilizado é o Log4j.

O sistema deverá “logar” todas as operações, nas camadas de Serviço, Negócio e Acesso a Dados (DAO) da seguinte maneira:

**DEBUG**: Utilizar o nível DEBUG em todas as classes de Serviço, Negócio e DAO. Logar os dados de entrada e o resultado do serviço.

**ERROR**: Utilizar o nível ERROR quando a mensagem de log estiver dentro de um tratamento de exceção.

As mensagens de log, devem ser direcionadas a um arquivo a parte, separado do SystemOut.log. Conforme especificado abaixo:

Código 8 – Exemplo de RollingFileAppender  


Esse *appender*, criará o arquivo **javaDeveloper.log**, na pasta definida pela variável de ambiente  **${SERVER\_LOG\_ROOT}** que deve estar configurada no servidor de aplicação.

O arquivo terá um tamanho máximo de 5Mb, e o servidor guardará apenas os últimos 5 arquivos.

# Visão Lógica

Esta visão fornece detalhes do modelo de Domínio, design lógico do projeto, bem como as divisões em subsistemas e pacotes. Para cada subsistema será mostrada sua interface externa definindo os serviços que serão oferecidos por este sistema.

## Pacotes

A nomenclatura dos pacotes deverá ser da seguinte forma:

**br.com.javadeveloper.**<pacote> , onde:

* <pacote> - divisão de pacotes de acordo com a camada de negócios (dao, service, business, web, client, dto, enum, exception, utils, helper, factory)

## Padrão de Nomenclatura para Classes

A aplicação desenvolvida deve seguir um padrão de nomenclatura para classes de acordo com o pattern, ou camada a qual pertencem. O quadro abaixo descreve esse padrão.

|  |  |
| --- | --- |
| Classes de WebService Rest | <nomeClasse>Rest |
| Classes auxiliares camada WEB (Helper) | <nomeClasse>Helper |
| Classes de Business Delegate | <nomeClasse>Delegate |
| Classes de Facade | <nomeClasse>Facade |
| Interfaces de Negócio / Serviço | <nomeInterface>Service |
| Classes de implementação de interfaces de Negócio / Serviço | <nomeClasse>ServiceImpl |
| Classes de Acesso ao BD (DAO) | <nomeClasse>DAO |
| Classes de Transferência de Dados (DTOs) | <nomeClasse>DTO |
| Classes de Exceção | <nomeClasse>Exception |
| Classes de Enum | <nomeClasse>Enum |
| Classes de Filter | <nomeClasse>Filter |
| Classes de Comparator | <nomeClasse>Comparator |
| Classes de Handling | <nomeClasse>Handler |
| Classes Utilitárias | <nomeClasse>Util |

\* Outros patterns que não tenham sido mencionados na tabela acima, devem seguir o mesmo padrão de nomenclatura. Identificando no sufixo da classe, qual o pattern que implementam.

## Versionamento

O padrão de versionamento utilizado foi via GitHub e o mesmo está disponível em: <https://github.com/rdmardegam/javaDeveloper>

# Arquitetura Visual Macro do Projeto

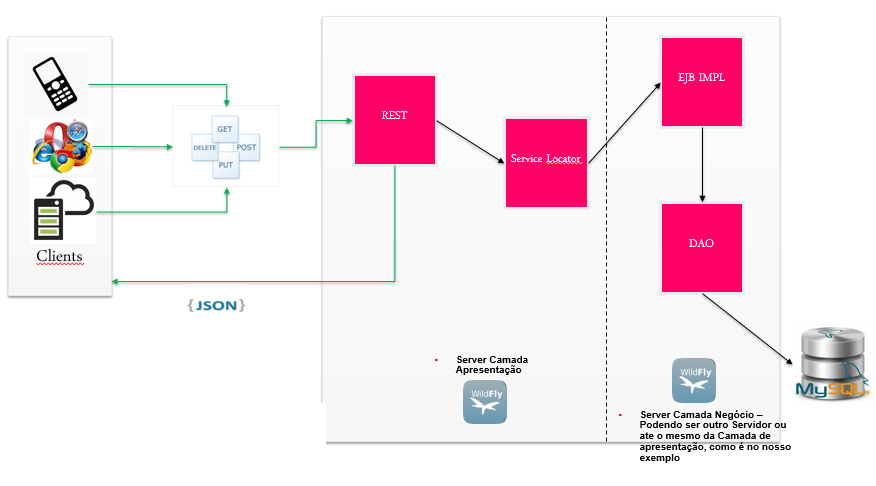
Neste projeto ficou definido que existiria uma integração de sistema, pois o serviço de consulta CEP já existia e o serviço Rest iria consumi-lo. Neste caso, simulei que seria disponibilizado um EJB Remote em que os serviços Rest o consumiriam.

No projeto foi utilizado a seguinte arquitetura:

\* Um WebService Rest com a implementação “EasyRest” irá receber as solicitações através de “Post”, “Get”, “Put” ou “Delete”.

\* O Rest utilizará o ServiceLocator para localizar o Ejb que o mesmo precisa invocar para que a solicitação seja atendida. Esse ejb poderia estar em outro local, outro servidor, mas no nosso exemplo está na mesma aplicação.

\* O ejb por sua vez solicitado, irá controlar a transação e executar as regras de negócio necessárias, chamando a camada de persistência DAO, onde foi utilizado o framework MyBatis para persistência de dados em um banco Mysql.



# Plataforma de Desenvolvimento

A plataforma de desenvolvimento da aplicação foi baseada no servidor de aplicações “Wildfly-8.2.0.Final” e bancos de dados MySql.

A IDE utilizada para desenvolvimento foi o Eclipse Mars juntamente com plugin Jboss Tools para possibilitar a inclusão do servidor de aplicação WildFly 8.X.

## Versões Recomendadas

As dependências do sistema devem seguir as especificações listadas no quadro a seguir:

|  |  |
| --- | --- |
| Plataforma | JavaSE 1.7.0\_25  JavaEE 7 |
| IDE | Eclipse Mars |
| Interface de Usuário | WebService Rest |
| Interface de Negócios | EJB 3.1 |
| Acesso a Dados | JDBC  MyBatis 3.3.0 |
| Log framework | Log4J 1.2.17 |
| Servidor de Aplicações | WildFly 8.2.0 Final |
| Banco de Dados | MySql |

# Usabilidade

## Interface de Usuário

O client que está tentando obter ou manipular as informações via Webservice Rest deve respeitar as definições de métodos do tipo GET, POST, PUT e DELETE estipulados por ele.

Por definição o projeto deve trabalhar apenas com a interface de retorno Json retornado da chamada do Webservice Rest. Para isso foi criado 2 padrões, um de erro e um de sucesso.

1. Para erros encontrados ou não previsto:

{

sucess: false

status: 400

codigoErro:1

mensagem: "A RUA DEVE SER INFORMADA CORRETAMENTE"

link: "javaDeveloper/endereco/method/"

mensagemDesenvolvedor: "O atributo rua é obrigatório e deve ser informado com até 128 caracteres."

}

Onde:

|  |  |
| --- | --- |
| sucess | Ao ocorrer um resultado não espera, retorna false |
| status | Retornado o http status que ajuda a identificar o problema |
| codigoErro | Código de erro da aplicação, no projeto foi utilizado um código fake |
| mensagem | Mensagem de erro que poderia ser possivelmente apresentada ao usuário |
| link | Poderia ser um link de ajuda ou referente ao método chamado. |
| mensagemDesenvolvedor | Mensagem de erro para ajudar o desenvolvedor a identificar o problema, em tempo de desenvolvimento pode exibir até o stack trace. |

2. Para chamadas efetuadas com sucesso:

{

sucess: true

data:

{

id: 2

cep: "04158050"

rua: "Doutor Rosalvo de Sales32"

bairro: null

cidade: "Sao Paulo"

estado: "Sao Paulo"

}

}

Ou apenas informativo

{

sucess: true

mensagem: "Endereço Cadastrado com Sucesso"

}

Onde:

|  |  |
| --- | --- |
| sucess | Ao ocorrer um resultado satisfatório, retorna true |
| data | Data sempre irá retornar a informação desejada, podendo ser uma lista ou coleção de informações, apenas um objeto ou valores singulares. Quando o retorno for simplesmente uma operação efetuada com sucesso sem a necessidade de retorno, esse atributo não será retornado. |
| mensagem | Mensagem amigável com o resultado da operação que foi realizada com sucesso. |

# Testes - cenários desenvolvidos

Os cenários apresentados a seguir foram executados utilizando a aplicação publicada na web. Para realizar os testes foi utilizado um app do Chrome chamado Advanced Rest Client que permite a chamada de Webservices Rest e seu resultado foi apresentado em forma de imagem nos testes abaixo:

## Cenários para a questão 1 – Pesquisa Cep

Por convenção as pesquisas devem ser feitas por um método http “GET” e como o GET não aceita header para receber um json, para que fosse possível passar um json, como a questão 1 estipula, solicitei que fosse passado um parâmetro para a chamada contendo um valor no formato json,

Exemplo de uso:

**Method**: **GET**

**Chamada**:

[http://www.koresystems.com.br/javaDeveloper/rest/endereco/buscarCep/{"cep":"04158050"}](http://www.koresystems.com.br/javaDeveloper/rest/endereco/buscarCep/%7b%22cep%22:%2204158050%22%7d)

Exemplo Result Json para a operação:

{

sucess: true

data:

{

id: 2

cep: "04158050"

rua: "Doutor Rosalvo de Sales32"

bairro: null

cidade: "Sao Paulo"

estado: "Sao Paulo"

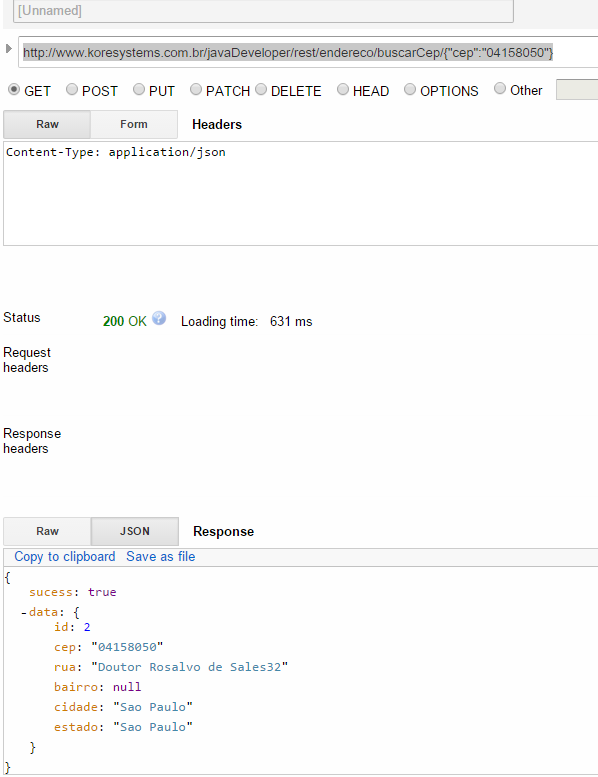
}

}

### Pesquisando um cep existente:

**Method**: **GET**

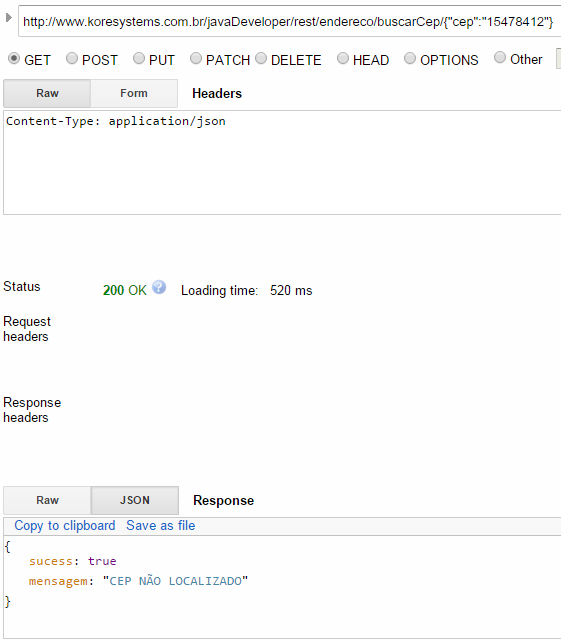
**Chamada**: [http://www.koresystems.com.br/javaDeveloper/rest/endereco/buscarCep/{"cep":"04158050"}](http://www.koresystems.com.br/javaDeveloper/rest/endereco/buscarCep/%7b%22cep%22:%2204158050%22%7d)



### Pesquisando um endereço cep cadastrado:

**Method**: **GET**

**Chamada**: [http://www.koresystems.com.br/javaDeveloper/rest/endereco/buscarCep/{"cep":"15478412"}](http://www.koresystems.com.br/javaDeveloper/rest/endereco/buscarCep/%7b%22cep%22:%2215478412%22%7d)

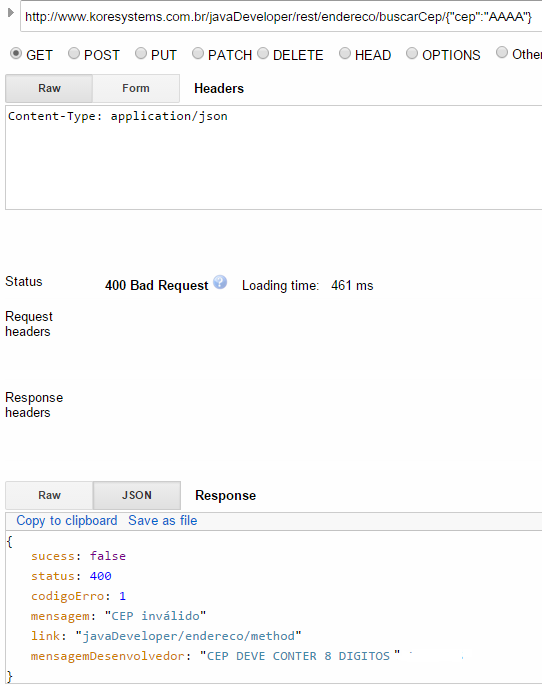


\*\*OBS: Neste caso foi retornado um sucess true, pois o CEP foi pesquisado com sucesso, porem o cep nao foi localizado. Dependendo do sistema podemos ter outra abordagem e retornar um status diferente de 200, como um (404 NOT FOUND). Como é um projeto exemplo, foi deixado o status 200 de retorno.

### Pesquisando um cep no formato inválido:

**Method**: **GET**

**Chamada**: [http://www.koresystems.com.br/javaDeveloper/rest/endereco/buscarCep/{"cep":"AAAA"}](http://www.koresystems.com.br/javaDeveloper/rest/endereco/buscarCep/%7b%22cep%22:%22AAAA%22%7d)

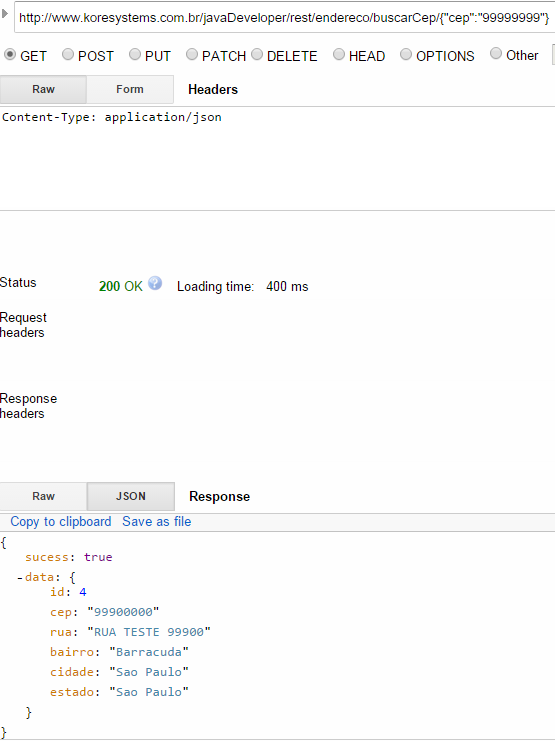


### Tentando localizar o CEP aproximado

Foi pedido na questão que fosse adicionado zeros à direita até que fosse encontrado o CEP, neste caso, existe no banco de dados o CEP de número “99900000'”, portanto, tentaremos pesquisar pelo CEP “99999999” e o sistema deve retornar a informação do CEP “99900000”:

**Method**: **GET**

**Chamada**: [http://www.koresystems.com.br/javaDeveloper/rest/endereco/buscarCep/{"cep":"99999999"}](http://www.koresystems.com.br/javaDeveloper/rest/endereco/buscarCep/%7b%22cep%22:%2299999999%22%7d)



## Cenários para a questão 2 – CRUD Endereço

A questão 2 se divide em varias chamadas ao nosso Rest para efetuar as operações de CRUD.

### Exemplo Inserir Endereço

Como os endereços vão ser controlados por id, foi permitido que fosse cadastrado o mesmo endereço varias vezes, pensando no mundo real, varias pessoas que moram na mesma resdencia poderiam cadastrar o mesmo endereço. Nesse caso como o “client” sempre é obrigado a informar o estado, cidade, cep entre outros dados, no nosso exemplo fiz com que o cep fosse sempre atualizado com a informação que o usuário esta fornecendo ao cadastrar seu endereço, no mundo real isso não deveria acontecer, mas sim, considerar sempre o endereço da pesquisa de cep e não permitir que o nome da rua, estados entre outros fosse alterados. Sendo assim, todos endereços que estão associados ao cep, terá sua informação de rua, estado entre ouros alterado.

Exemplo de uso:

**Method**: **POST**

**Chamada**:

<http://www.koresystems.com.br/javaDeveloper/rest/endereco/cadastrar>

Exemplo de Json a ser enviado ao Rest:

{

"numero": "25",

"complemento" : "Ap 443",

"cepInfo": {"cep": "99900000", "rua": "Rua TESTE 99900",

"bairro": "Barracuda","cidade": "Sao Paulo",

"estado": "Sao Paulo"

}

}

Exemplo Result Json para a operação:

{

sucess: true

mensagem: "Endereço Cadastrado com Sucesso"

}

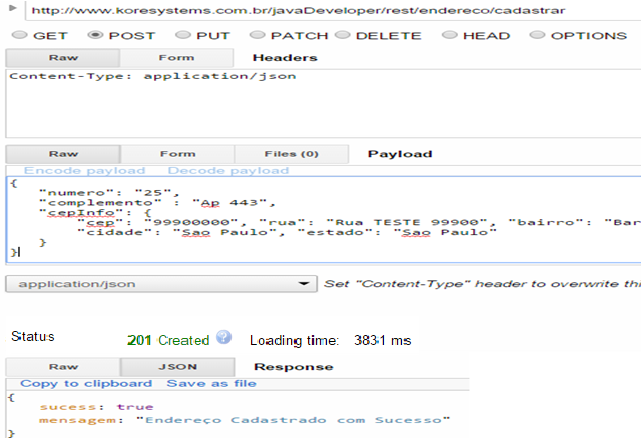
### Inserir Endereço com sucesso

**Method**: **POST**

**Chamada**: <http://www.koresystems.com.br/javaDeveloper/rest/endereco/cadastrar>

**Json Enviado**:

{ "numero": "25", "complemento" : "Ap 443", "cepInfo": { "cep": "99900000", "rua": "Rua TESTE 99900", "bairro": "Barracuda","cidade": "Sao Paulo", "estado": "Sao Paulo"}}



\*\*OBS: Aqui poderíamos ter um status “OK 200” dependendo do padrão da aplicação

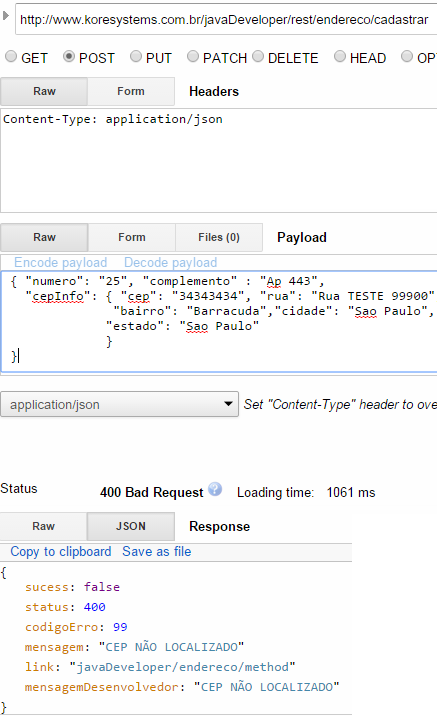
### Inserir Endereço com CEP não localizado sucesso

**Method**: **POST**

**Chamada**: <http://www.koresystems.com.br/javaDeveloper/rest/endereco/cadastrar>

**Json Enviado**:

{ "numero": "25", "complemento" : "Ap 443", "cepInfo": { "cep": "34343434", "rua": "Rua TESTE 99900", "bairro": "Barracuda","cidade": "Sao Paulo", "estado": "Sao Paulo"}}



\*\*Obs: Poderiamos retornar um status NOT FOUND 404, dependendo do padrão da aplicação

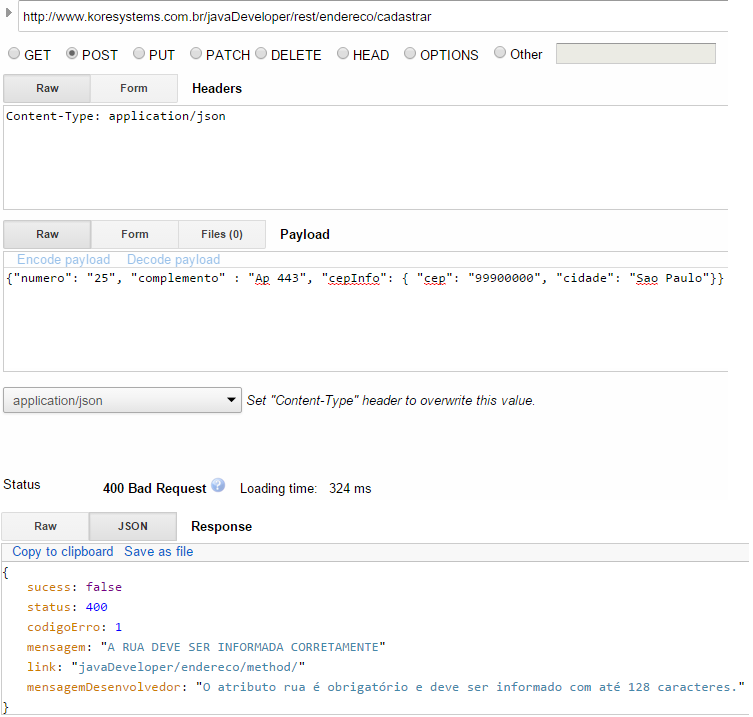
### Inserir Endereço com CEP não localizado sucesso

**Method**: **POST**

**Chamada**: <http://www.koresystems.com.br/javaDeveloper/rest/endereco/cadastrar>

**Json Enviado**:

{"numero": "25", "complemento" : "Ap 443", "cepInfo": { "cep": "99900000", "cidade": "Sao Paulo"}}



### Exemplo Alterar Endereço

Para que o endereço seja alterado, o mesmo deve existir previamente cadastrado, portanto, temos que enviar seu id para alteração.

Exemplo de uso:

**Method**: **PUT**

**Chamada**:

<http://www.koresystems.com.br/javaDeveloper/rest/endereco/atualizar>

Exemplo de Json a ser enviado ao Rest:

{

"id" : 33,

"numero": "356",

"complemento" : "Prox ao Posto XYZ",

"cepInfo": {

"cep": "99900000",

"rua": "RUA TESTE 99900",

"bairro": "Barracuda 2",

"cidade": "Sao Paulo",

"estado": "Sao Paulo"

}

}

Exemplo Result Json para a operação:

{

sucess: true

mensagem: "Endereço Alterado com Sucesso"

}

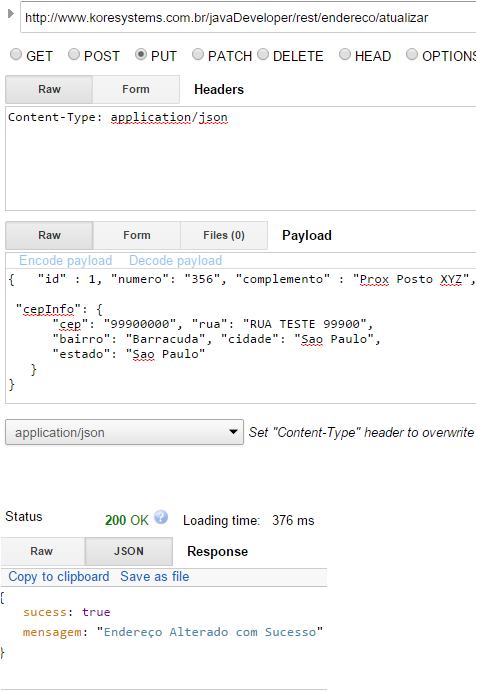
### Alterar Endereço com sucesso

**Method**: **PUT**

**Chamada**: <http://www.koresystems.com.br/javaDeveloper/rest/endereco/atualizar>

**Json Enviado**:

{ "id" : 1, "numero": "356", "complemento" : "Prox Posto XYZ", "cepInfo": {"cep": "99900000", "rua": "RUA TESTE 99900", "bairro": "Barracuda", "cidade": "Sao Paulo","estado": "Sao Paulo"}}



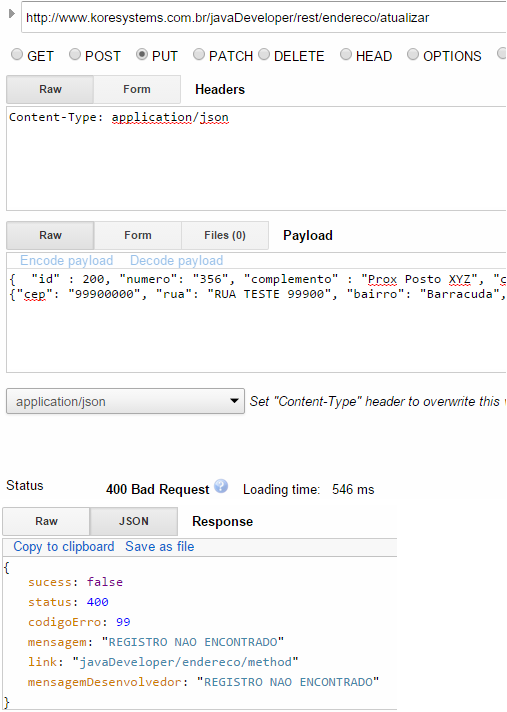
### Alterar Endereço passando um id inexistente

**Method**: **PUT**

**Chamada**: <http://www.koresystems.com.br/javaDeveloper/rest/endereco/atualizar>

**Json Enviado**:

{ "id" : 1, "numero": "356", "complemento" : "Prox Posto XYZ", "cepInfo": {"cep": "99900000", "rua": "RUA TESTE 99900", "bairro": "Barracuda", "cidade": "Sao Paulo","estado": "Sao Paulo"}}



### Pesquisa de Endereço

Outra vez utilizaremos um serviço de método GET, mas neste caso não irei esperar um json como no pesquisa cep e sim um parâmetro na própria url simplesmente com o id do endereço desejado.

Exemplo de uso:

**Method**: **GET**

**Chamada**:

<http://www.koresystems.com.br/javaDeveloper/rest/endereco/buscarEndereco/1>

Exemplo Result Json para a operação:

{ sucess: true

data:

{ id: 1

numero: "356"

complemento: "Prox Posto XYZ"

cepInfo:

{ id: null

cep: "99900000"

rua: "RUA TESTE 99900"

bairro: "Barracuda"

cidade: "Sao Paulo"

estado: "Sao Paulo"

}

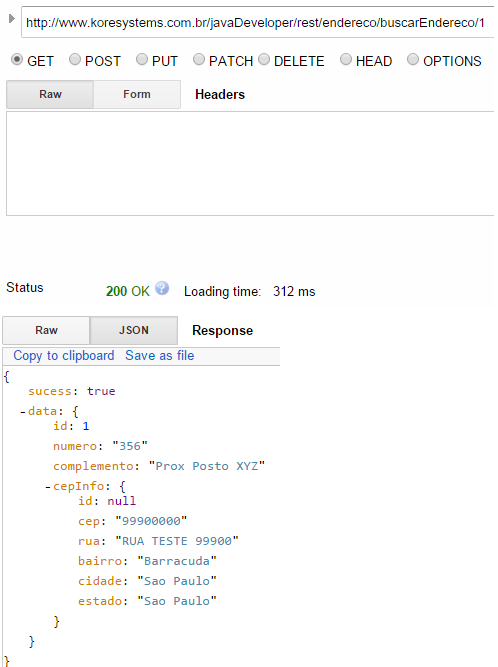
}

}

### Pesquisando um Endereço com sucesso

**Method**: **GET**

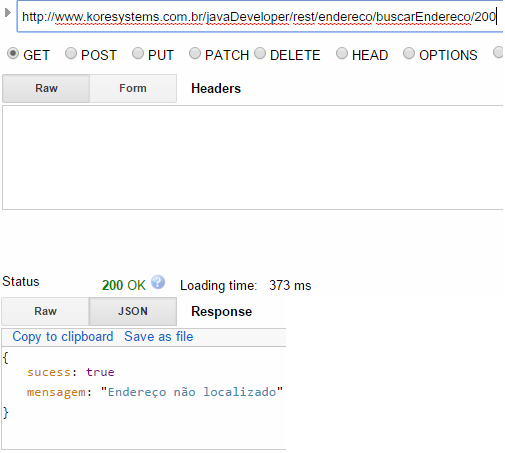
**Chamada**:<http://www.koresystems.com.br/javaDeveloper/rest/endereco/buscarEndereco/1>



### Pesquisando um Endereço inexistente

**Method**: **GET**

**Chamada**: <http://www.koresystems.com.br/javaDeveloper/rest/endereco/buscarEndereco/200>



\*\*Obs: Dependendo do padrão da aplicação poderia ser status diferente de 200 para informar que não encontrou o item pesquisado. Neste exemplo a pesquisa foi realizada com sucesso e simplesmente não existem dados para retornar.

### Exemplo Deletar Endereço

Devemos informar o id do endereço que gostaríamos de apagar. Mesmo que o endereço seja apago, a informação do cep é mantida e permanece no sistema.

Exemplo de uso:

**Method**: **DELETE**

**Chamada**:

<http://www.koresystems.com.br/javaDeveloper/rest/endereco/deletar>

Exemplo de Json a ser enviado ao Rest:

{

"id" :8

}

Exemplo Result Json para a operação:

{

sucess: true

mensagem: "Endereço Deletado com Sucesso"

}

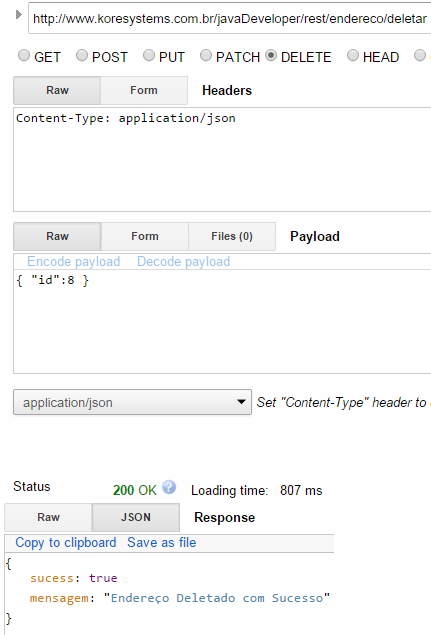
### Deletar Endereço com sucesso

**Method**: **PUT**

**Chamada**: <http://www.koresystems.com.br/javaDeveloper/rest/endereco/deletar>

**Json Enviado**:

{ "id" : 8 }



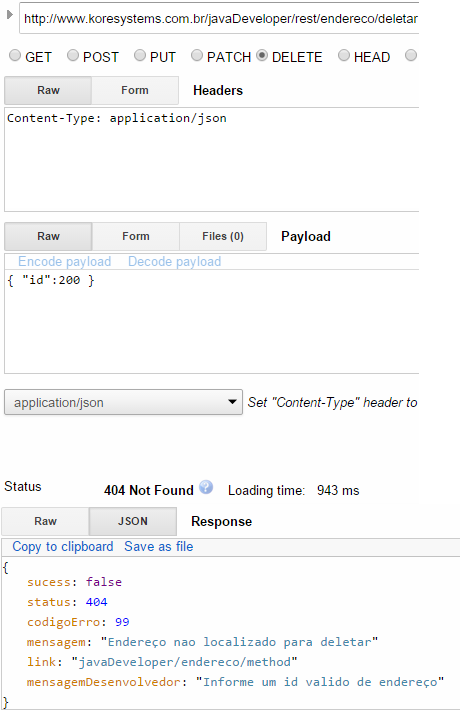
### Deletar um Endereço inexistente

**Method**: **PUT**

**Chamada**: <http://www.koresystems.com.br/javaDeveloper/rest/endereco/deletar>

**Json Enviado**:

{ "id" : 200 }

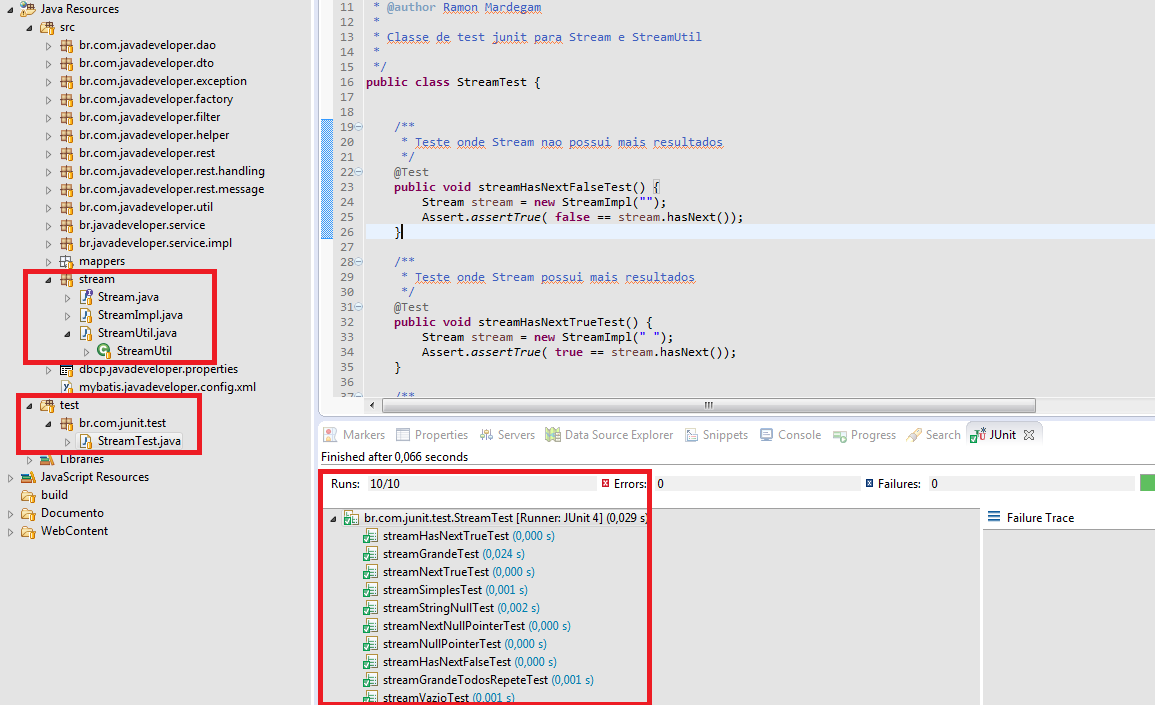


## Cenários para a questão 3 – Implementação Stream e sua busca da primeira ocorrencia de um caracter que não se repete

A questão 3 se dividia em criar uma implementação para a interface Stream para leitura de caracter por caracter de uma String e tentar localizar a primeira caracter que não se repete da melhor maneira possível.

No projeto disponibilizado temos a interface “Stream”, sua implementação “StremUtil” e também a classe “StreamUtil” que contem o método public static char “fistChart(Stream stream)” responsável por efetuar a regra de localização da primeira palavra que não se repete.

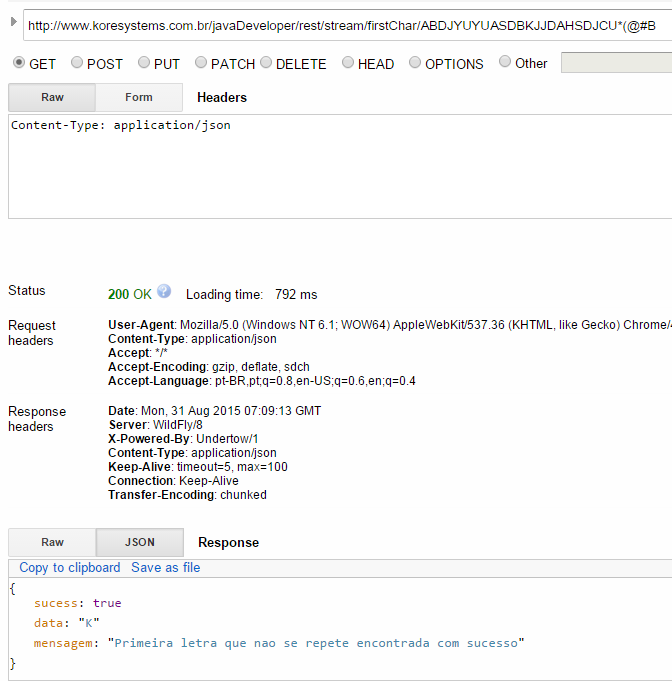
Efetuei alguns testes com difersos cenários através do junit e os disponibilizei jutamente com o codigo, o mesmo está localizado em test.br.com.junit.test.StreamTest.java



Também foi criado um WebService Rest que pode ser chamado para utilizar esse método:

**Method**: **GET**

**Chamada**: [http://www.koresystems.com.br/javaDeveloper/rest/stream/firstChar/ABDJYUYUASDBKJJDAHSDJCU\*(@#B](http://www.koresystems.com.br/javaDeveloper/rest/stream/firstChar/ABDJYUYUASDBKJJDAHSDJCU*(@#B)

 \*Trouxe corretamente o caracter “K” pois é o primiero caracter que não se repete no resto da stream

## Cenários para a questão 4 – Oque ocorre no processo de protocolo http entre Client e Server ao solicitar uma url

O protocolo http se baseia simplesmente em requisição e resposta, onde o client efetua uma requisição http ao servidor onde é possível enviar informações em seu corpo ou header para o servidor e o tipo de método a adotar nessa requisição como “Get”, “Post”, “Put”, “Delete” entre outros.

O servidor por sua vez, recebe essa requisição http onde podemos chamar de http request e identifica o tipo de requisição e seus dados. Assim o servidor poderá utilizar essas informações e efetuar as operações que julgar necessárias, seja efetuar um cálculo ou executar alguma regra de negocio e retorna ao client uma requisição http, onde poderíamos chamar de http response.

O response por sua vez poderia ser preenchido entre outros com dados e status de retorno e seria retornado ao solicitante, vulgo client.

O client por sua vez receberia o response para utilizar os dados retornados e identificar se a

operação solicitada foi executada com sucesso, através de seus status.