





# APIs RESTful

**Definições e Orientações para uso na CAIXA**

	<b>APIs RESTful – Definições e Orientações para uso na CAIXA</b>	Página 1 / 23
		Data Emissão 16/06/2020


## Histórico da Revisão

<b>Data</b>	<b>Versão</b>	<b>Descrição</b>	<b>Autor</b>
22/03/2017	0.1	Criação do documento	ARQDIGITAL
03/04/2017	0.2	Revisão	ARQDIGITAL
06/04/2017	0.3	Revisão	ARQDIGITAL
18/04/2017	1.0	Versão Final	ARQDIGITAL
28/12/2017	1.1	Revisão, inclusão e detalhamento dos padrões	César Lima Piau
26/10/2018	2.0	Reformulação do documento	GEARQ
16/06/2020	2.1	Inclusão boas práticas, revisões e atualizações	César Lima Piau

	<b>APIs RESTful – Definições e Orientações para uso na CAIXA</b>	Página 2 / 23
		Data Emissão 16/06/2020

## Índice Analítico

1	INTRODUÇÃO .....	3
1.1	APRESENTAÇÃO E OBJETIVO DO PADRÃO .....	3
1.2	PÚBLICO ALVO .....	3
2	INTRODUÇÃO .....	4
2.1	O PADRÃO REST .....	4
3	O PADRÃO REST NA CAIXA .....	5
3.1	URL PADRONIZADA E INTELIGÍVEL .....	5
3.1.1	Domínio de negócio .....	5
3.1.2	Recursos e Coleções .....	6
3.1.3	Versão .....	7
3.2	MODELO CANÔNICO E APIS CANÔNICAS .....	7
3.3	USO CORRETO DOS METODOS HTTP .....	8
3.3.1	Casos de exceção .....	8
3.3.2	APIs REST vs APIs RPC .....	9
3.4	USO DE QUERY STRINGS .....	12
3.4.1	Filtros .....	12
3.4.2	Conteúdo parcial .....	12
3.4.3	Paginação .....	13
3.4.4	Ordenação .....	13
3.5	TIPO DE RETORNO DE DADOS .....	14
3.6	CÓDIGOS DE STATUS HTTP .....	14
3.7	LIMITES DE REQUISIÇÕES .....	15
3.8	USO DE CACHE .....	15
3.9	HYPERMEDIA .....	16
4	BOAS PRÁTICAS DE GOVERNANÇA DE APIS .....	17
4.1	AMBIENTE DE SANDBOX .....	17
4.2	ENDPOINT DE HEALTH CHECK .....	17
5	SEGURANÇA .....	18
5.1	CORS (CROSS-ORIGIN RESOURCE SHARING) .....	18
5.2	UTILIZAÇÃO DE SSL .....	18
5.3	CONTROLE DE ACESSO .....	18
6	DOCUMENTAÇÃO DAS APIS .....	20
7	REFERÊNCIAS .....	22

	APIs RESTful – Definições e Orientações para uso na CAIXA	Página 3 / 23
		Data Emissão 16/06/2020

## 1 INTRODUÇÃO

### 1.1 APRESENTAÇÃO E OBJETIVO DO PADRÃO

Este documento fornece orientações e definições para desenvolvimento e consumo de Web APIs da CAIXA.

O objetivo deste documento é nivelar o conhecimento sobre o que são APIs REST, apresentar as definições para desenvolvimento e consumo na CAIXA, fomentar a qualidade, consistência e manutenibilidade das APIs, bem como agilizar o desenvolvimento e pacificar discussões quanto ao uso de padrões e melhores práticas.

O desenvolvimento de APIs na CAIXA tem como objetivo fornecer uma forma simplificada de publicação e consumo de web services, alcançando assim uma grande variedade de clientes em diversas tecnologias.

### 1.2 PÚBLICO ALVO

Desenvolvedores e Consumidores de APIs.

## 2 INTRODUÇÃO

Uma Web API é um tipo de serviço (web service) que é exposto utilizando basicamente o protocolo HTTP.

Portanto, o protocolo HTTP é o alicerce das Web APIs, o que significa que durante o design de uma API, devemos lidar com todos os aspectos do protocolo.

O padrão arquitetural que utiliza o protocolo HTTP para o desenvolvimento de Web APIs é chamado de REST (Representational State Transfer) e as APIs desenvolvidas utilizando esse padrão arquitetural são chamadas de APIs REST.

O termo RESTful é relacionado à aderência da API ao padrão REST.

A seguir serão descritos os principais padrões e melhores práticas que envolvem o desenvolvimento de APIs REST.

### 2.1 O PADRÃO REST

A Representational State Transfer (REST), em português Transferência de Estado Representacional, é um estilo de arquitetura que define um conjunto de restrições e propriedades baseados em HTTP.

Em web services compatíveis com REST, como as APIs RESTful, os componentes do protocolo HTTP (URL, métodos, códigos de status e cabeçalhos) são usados de forma semântica para acessar e manipular recursos expostos na Web.

Outros tipos de web services, como web services SOAP, expõem seus próprios conjuntos arbitrários de operações.

O termo REST foi apresentado no ano de 2000 por Roy Fielding, um dos principais autores da especificação do protocolo HTTP, em uma tese de doutorado (PHD) na UC Irvine.

Você pode ler mais sobre a tese de Roy Fielding [aqui](#).

### 3 O PADRÃO REST NA CAIXA

Para o desenvolvimento de APIs REST na CAIXA, foi adotado grande parte do modelo de Roy Fielding, porém, como de praxe na web, algumas definições foram flexibilizadas ou ignoradas para ficarem mais próximas à realidade da empresa.

A seguir vamos detalhar as características que as APIs REST da CAIXA devem ter para atender aos padrões de qualidade esperados.

#### 3.1 URL PADRONIZADA E INTELIGÍVEL

A definição da URL é uma das partes mais importantes do design de uma API, pois boa parte do entendimento de como a API funciona pode ser obtido interpretando a sua URL.

Portanto, a URL deve expressar de maneira clara quais recursos estão sendo acessados e de que forma eles são manipulados.

Em uma API REST, a URL pode ser dividida em 4 componentes:

- **HOST:** É o endereço onde está hospedada a API. É importante que uma organização tenha um host dedicado para suas APIs. Exemplo:

```
http://api.caixa  
https://api.caixa.gov.br
```

- **BASE PATH:** Representa o domínio de negócio da API, permitindo que a API seja agrupada e organizada. Exemplo:

```
http://api.caixa/loterias  
https://api.caixa.gov.br/habitacao
```

- **PATHS:** Representam os recursos disponibilizados pela API. Exemplo:

```
http://api.caixa/loterias/resultados  
https://api.caixa.gov.br/habitacao/contratos
```

- **QUERY STRING:** São utilizadas para enviar parâmetros nas requisições. Exemplos:

```
http://api.caixa/loterias/resultados/mega-sena?concurso=2569  
https://api.caixa.gov.br/habitacao/contratos?cpf=00100200344
```

##### 3.1.1 Domínio de negócio

É importante que a URL tenha um domínio de negócio que permita agrupar e organizar as APIs de forma coerente. Exemplo:

```
https://api.caixa/cartao-credito/faturas  
https://api.caixa/cadastro/clientes
```

Algumas APIs podem ter sub-domínios para melhorar a organização e a compreensão da sua finalidade. Exemplo:

```
https://api.caixa/seguranca/gestao-senhas  
https://api.caixa/seguranca/criptografia-senhas
```

### 3.1.2 Recursos e Coleções

Uma API REST vai estar sempre manipulando ou consultando recursos/itens de uma coleção/lista e a URL deve representar essas coleções de recursos por meio de substantivos concretos. Por se tratar de uma coleção, os recursos devem ser representados sempre no plural.

Por exemplo, uma API que trata da consulta e manipulação de clientes, deve apresentar em sua URL a coleção **clientes** como sendo um recurso.

```
https://api.caixa/cadastro/clientes
```

O acesso um recurso de uma coleção na URL deve seguir o formato “**coleção/item**”, onde **item** é um identificador único para um determinado recurso de uma coleção.

```
.../credito-comercial/consignado/contratos/{id-contrato}  
.../cadastro/clientes/{id-cliente}
```

O conceito de coleções para uma API REST vai além da questão semântica. Ao realizar uma consulta (GET) em uma URI de coleção sem identificar um recurso específico, devemos obter uma lista de itens dessa coleção. Exemplo:

```
GET  
https://api.caixa/cadastro/clientes
```

Essa consulta deveria retornar uma lista dos clientes da coleção. Posteriormente serão abordados os mecanismos para filtrar ou organizar os resultados da consulta de uma lista.

Assim como na orientação a objetos ou em um modelo relacional de dados, os recursos podem estar relacionados a outros recursos, bem como um recurso pode ser dividido em sub-recursos.

Ainda no exemplo anterior, um determinado cliente de uma coleção pode ter associado a ele uma coleção de **endereços** ou **rendas**.

```
https://api.caixa/cadastro/clientes/{id-cliente}/enderecos/{id-endereco}  
https://api.caixa/cadastro/clientes/{id-cliente}/rendas/{id-renda}
```

O conjunto de recursos e sub-recursos de uma API é chamado de **árvore de recursos**. A árvore de recursos é uma diagrama que mostra como os recursos se relacionam e são acessados na URL. A construção dessa árvore de recursos é umas das atividades mais importantes da etapa

de design da API e deve ser feita com a devida atenção, pois impacta diretamente na qualidade da API.

A seguir temos um exemplo de árvore de recursos para clientes:

```
clientes/{id_cliente}  
    /endereços/{id_endereco}  
    /rendas/{id_renda}
```

Para nomes compostos de recursos ou atributos na URL, deve-se utilizar o padrão **spinal-case** ou, caso exista algum impedimento, utilizar o padrão **cramcase**. A sintaxe das URL deve seguir a **RFC 3986 - Uniform Resource Identifier (URI): Generic Syntax**.

### 3.1.3 Versão

Outro componente importante da URL é a versão da API.

Dentre as diversas formas de se controlar a versão de uma API, a Caixa optou por usar o versionamento na URL, por ser a forma mais utilizada atualmente no mercado, porém, cabe ressaltar que existem outras abordagens para essa funcionalidade.

As APIs devem ser publicadas com um número de versão, pois o versionamento proporciona iterações mais rápidas, evita pedidos inválidos e mantém endpoints atualizados, além de permitir que as aplicações mais antigas consumam as primeiras versões da API e continuem funcionando corretamente.

As versões devem utilizar números inteiros prefixados com 'v', como por exemplo: v1, v2, v3. O uso de versões fracionadas deve ser evitado e a versão só deve ser incrementada se houver mudança na interface. A versão da API deve ser independente da versão do provedor do serviço, seja ele uma aplicação Java ou um fluxo no Barramento de Serviços.

Alguns projetos necessitam de versionamento a nível de recurso, podendo a API ter o mesmo recurso em versões diferente. Exemplo:

```
.../loterias/v1/resultados/mega-sena  
.../cadastro/v1/clientes  
.../v2/clientes
```

## 3.2 MODELO CANÔNICO E APIS CANÔNICAS

Os recursos de uma API precisam ter uma representação ou modelo com seus atributos e relacionamentos muito bem definidos.

Um recurso que é definido de forma global e completa dentro de uma organização de modo que seus atributos possam ser reconhecidos por qualquer aplicação possui o que chamamos de **modelo canônico**. Um recurso não pode ter mais de um modelo canônico.



Em outras palavras, ter um modelo canônico para o recurso cliente, significa que em qualquer lugar onde a entidade cliente for referenciada ela deve ter os mesmos atributos, com os mesmo nomes e os mesmos relacionamentos.

Cabe ressaltar que o modelo canônico não precisa ter uma relação direta com o modelo de dados onde o recurso é persistido.

Da mesma forma, uma **API canônica** é aquela capaz de prover o acesso aos seus recursos de forma global, contemplando todos os cenários de utilização dos seus clientes de forma versátil, sem que haja a necessidade de criação de versões “customizadas” para atender particularidades de alguns consumidores.

Por exemplo, uma API de pagamento de boleto mantida por mais de um sistema, com modelos canônicos e árvores de recursos distintos não pode ser considerada uma API canônica.

### 3.3 USO CORRETO DOS METODOS HTTP

Uma vez que as APIs na maioria dos casos representam recursos em vez de ações, devemos utilizar os verbos ou métodos HTTP para operar esses recursos. Os métodos HTTP permitem atribuir operações de CRUD (Create, Read, Update e Delete) aos recursos.

Dentro do universo de métodos HTTP, os mais utilizados para as APIs são os seguintes:

MÉTODO HTTP	DESCRIÇÃO
GET	Método utilizado para consultar uma lista ou um recurso específico, podendo encaminhar na URL da requisição filtros para recuperação dos dados.
POST	Método utilizado para incluir um recurso em uma coleção. Os dados do recurso a ser criado podem ser passados na URL como parâmetros ou no corpo da requisição em formatos como JSON, XML e outros.
PUT	Método usado para atualizar um recurso específico. O corpo da requisição contém uma representação do registro podendo estar em formatos como XML, JSON e outros.
DELETE	Método utilizado para excluir um recurso específico ou uma lista.
OPTIONS	Método utilizado para consultar ao servidor quais os métodos, cabeçalhos e origens aceitas para uma determinada API. Muito utilizado em requisições cross-origin (CORS).
HEAD	Método utilizado para consultar a disponibilidade de um recurso, sem retornar seus dados.

#### 3.3.1 Casos de exceção

Apesar do padrão REST preconizar o uso dos métodos HTTP para representar as operações, existem casos em que será necessário adequar o modelo para atender alguma necessidade específica.

Um exemplo muito comum que foge à regra é uma API de login que, além de conter um verbo na sua URL, ainda precisa utilizar o método POST para passar os dados sensíveis de autenticação dentro do corpo da requisição. Exemplo:

```
POST
https://api.caixa/login-caixa/v1/login
{
  "usuario": "x",
  "senha": "y"
}
```

Outra necessidade de utilização de métodos HTTP diferente do indicado é nos casos em que uma consulta precisa ser feita com um identificador ou filtro que contenha informações sensíveis. Nesse caso, deve-se utilizar o método POST, indicar na URL a ação que pretende fazer e passar os dados sensíveis no corpo da requisição. Exemplo:

```
POST
https://api.caixa/cartao-credito/v1/faturas/consultar
{
  "numero-cartao": "9999.9999.9999.9999",
  "mes": "01",
  "ano": "2017"
}
```

É importante que nesses casos utilizemos o verbo no infinitivo para indicar a ação: **consultar**, **cadastrar**, **excluir**, **transferir**.

**ATENÇÃO:** É necessário consultar um designer de API para verificar se seu recurso pode ser enquadrado em um caso de exceção ou se é necessário se adequar aos verbos HTTP conforme padrão.

### 3.3.2 APIs REST vs APIs RPC

Existem outros padrões de desenvolvimento de APIs além do padrão REST. Dentre esses, o mais comum é o RPC (Remote Procedure Call).

APIs RPC surgiram como uma alternativa aos Web Services SOAP, porém, sem abrir mão da orientação a operações. Elas utilizam verbos representando funções nas URLs e basicamente os métodos GET e POST.

Em um exemplo prático, uma consulta de um cliente em chamada RPC seria:

```
POST
https://api.caixa/cadastro/v1/consultarCliente
Body:
{
  "cpf": "10020030088"
}
```

A mesma função sendo chamada em uma API REST:

```
GET
https://api.caixa/cadastro/v1/clientes/10020030088
```

Ao analisar o padrão de API REST, o verbo HTTP utilizado nos mostra qual a intenção do usuário e a URL mostra o recurso ligado a esta ação.

Para deixar mais claro as diferenças entre os padrões vamos analisar mais alguns exemplos:

#### API RPC

##### POST

`https://api.caixa/cadastro/v1/incluirCliente`

Body:

```
{
  "cpf": "10020030088",
  "nome": "João",
  "data-nascimento": "1990-01-01"
}
```

#### API REST

##### POST

`https://api.caixa/cadastro/v1/clientes`

Body:

```
{
  "cpf": "10020030088",
  "nome": "João",
  "data-nascimento": "1990-01-01"
}
```

#### API RPC

##### POST

`https://api.caixa/cadastro/v1/alterarCliente`

Body:

```
{
  "cpf": "10020030088",
  "nome": "João da Silva",
  "data-nascimento": "1990-01-01"
}
```

#### API REST

##### PUT

`https://api.caixa/cadastro/v1/clientes/10020030088`

Body:

```
{
  "nome": "João",
  "data-nascimento": "1990-01-01"
}
```

Analisando um cenário onde aplicamos o modelo CRUD para clientes, teríamos as seguintes URLs para as API RPC e REST:

#### API RPC

##### POST

<https://api.caixa/cadastro/v1/incluirCliente>

##### POST/GET

<https://api.caixa/cadastro/v1/consultarCliente>

##### POST

<https://api.caixa/cadastro/v1/alterarCliente>

##### POST/GET

<https://api.caixa/cadastro/v1/excluirCliente>

##### POST/GET

<https://api.caixa/cadastro/v1/listarClientes>

#### API REST

<https://api.caixa/cadastro/v1/clientes>

(incluir:POST, listar:GET)

<https://api.caixa/cadastro/v1/clientes/00100200344>

(consultar:GET, alterar:PUT, excluir:DELETE)

O padrão REST define uma forma intuitiva de interação com a coleção de clientes, permitindo que tenhamos apenas 2 URLs para atender as 5 operações propostas. Além disso, o desenvolvedor não precisa saber quais nomes foram dados para as operações, pois o uso dos métodos HTTP em conjunto com as coleções torna-se intuitivo na medida que conhecemos o padrão.

Uma API REST ainda permite que o desenvolvedor navegue pela árvore de recursos explorando sub-recursos de forma igualmente intuitiva. Exemplo:

#### API RPC

##### POST

<https://api.caixa/cadastro/v1/incluirTelefoneCliente>

##### POST/GET

<https://api.caixa/cadastro/v1/consultarTelefoneCliente>

##### POST

<https://api.caixa/cadastro/v1/alterarTelefoneCliente>

##### POST/GET

<https://api.caixa/cadastro/v1/excluirTelefoneCliente>

##### POST/GET

<https://api.caixa/cadastro/v1/listarTelefoneCliente>

#### API REST

<https://api.caixa/cadastro/v1/clientes/00100200344/telefones>

(incluir:POST, listar:GET)

<https://api.caixa/cadastro/v1/clientes/00100200344/telefones/001>

(consultar:GET, alterar:PUT, excluir:DELETE)

Aqui podemos notar, o padrão REST é muito mais simples, intuitivo e inteligível, reduzindo a curva de aprendizado dos desenvolvedores que vão consumir o serviço.

Essa simplicidade e a grande popularidade do padrão REST no mercado, fez com que a CAIXA adotasse esse padrão para a exposição e consumo de APIs.

### 3.4 USO DE QUERY STRINGS

As query strings são componentes importantes das APIs, permitindo a inserção de parâmetros que podem ser utilizados em diferentes situações.

É caracterizada pelo sinal de interrogação ao final da URI e o uso de & para adicionar mais parâmetros à requisição:

```
https://api.caixa/{domínio-de-  
negócio}/{versão}/{coleção}?parametro1={valor1}&parametro2={valor2}
```

Query strings podem ter diversas funções em uma API REST, como veremos a seguir.

#### 3.4.1 Filtros

Em APIs de consulta, os filtros possibilitam que apenas os registros necessários para a aplicação consumidora sejam trafegados.

É possível filtrar uma lista por vários atributos ao mesmo tempo e filtrar mais de um valor para um atributo. Exemplo:

```
http://api.caixa/informacoes/v1/unidades?uf=DF,MG&penhor=sim
```

#### 3.4.2 Conteúdo parcial

A função de trazer conteúdo parcial é uma das mais importantes de uma query string, pois permite ao cliente de nossa API limitar o tráfego de informação apenas ao que ele precisa.

Nesse caso é comum o uso de uma variável chamada “fields” nas APIs de mercado. Aqui na CAIXA, decidimos por usar essa mesma variável, em sua tradução para o português: *campos*.

```
https://api.caixa/cadastro/v1/clientes/id_cliente?campos=nome,idade
```

Nos casos em que um recurso possui muitos atributos, devemos permitir a consulta de conteúdo parcial. Exemplo:

```
GET  
https://api.caixa/corporativo/v1/unidades/5404?campos=nome,sigla,uf  
  
HTTP 200 OK  
{  
  "nome": "ARQUITETURAS DE TI",  
  "sigla": "GEARQ",  
  "uf": "DF"  
}
```

Em alguns casos, podemos prover conteúdo parcial com base em grupos de atributos. Essa abordagem diminui a complexidade de implementação da API. Exemplo:

```
GET
https://api.caixa/cadastro/v1/clientes/11122233344?campos=dados-basicos

HTTP 200 OK
{
  "dados-basicos":
  {
    "nome": "João da Silva",
    "data-nascimento": "25-12-1980",
    "nome-mae": "Maria da Silva"
  }
}
```

### 3.4.3 Paginação

A paginação de resultados é uma opção quando o volume de registros de uma consulta for muito grande.

Para a paginação, deve-se utilizar os parâmetros **offset** e **limit**, onde offset representa o índice do primeiro elemento da lista e limit a quantidade de elementos a serem retornados na lista.

A resposta da API para uma lista paginada poderá utilizar o status HTTP **206 – Partial Content**. É importante que a resposta também contenha referências para a página anterior e a próxima página. Exemplo:


```
GET
http://api.caixa/informacoes/v1/unidades?limit=3

HTTP 206 OK
{
  "unidades":
  [
    {"Codigo": "0002", "nome": "PLANALTO, DF"},
    {"Codigo": "0003", "nome": "AEROPORTO PRESIDENTE JK, DF"},
    {"Codigo": "0004", "nome": "BERNARDO SAYAO, DF"}
  ],
  "paginacao":
  {
    "count": 100,
    "next": "http://api.caixa/v1/unidades?offset=3,limit=3"
  }
}
```

### 3.4.4 Ordenação

Deve-se evitar o uso de ordenação no servidor e dar preferência, sempre que possível, para que a ordenação dos resultados seja feita no cliente.

Caso seja necessária a implementação de ordenação de resultados no servidor, deve-se usar os parâmetros **sort** e **desc**, onde sort contém o nome do atributo usado para ordenar a lista e desc indica que a ordem será descendente ou decrescente (por padrão a ordem é ascendente ou crescente). Exemplo:

	APIs RESTful – Definições e Orientações para uso na CAIXA	Página 14 / 23
		Data Emissão 16/06/2020

<https://api.caixa/cadastro/v1/clientes?sort=nome>

### 3.5 TIPO DE RETORNO DE DADOS

Uma API pode trocar mensagens em qualquer formato suportado pelo protocolo HTTP, porém o formato JSON é o formato mais utilizado por ser mais leve, menos verboso e mais aderente ao padrão de documentação.

Na CAIXA, o formato padrão é o JSON, mas por questões de compatibilidade, uma API pode ser capaz de responder em mais de um formato (como o XML).

Para definir o tipo de dados que a API deverá retornar a partir de uma requisição, o cliente deverá passar o cabeçalho **Accept** na requisição com o tipo de retorno que o mesmo deseja (**application/xml** ou **text/xml** para o tipo de retorno XML ou **application/json** para o tipo de retorno JSON). Caso o tipo de dado informado pelo cliente não seja suportado pela API, deve-se retornar o código de status HTTP **415 - Unsupported Media Type** ou **400 – Bad Request** e se o cabeçalho não for informado deve-se retornar o formato JSON por padrão.

Na resposta da requisição é necessário informar no cabeçalho **Content-Type** o tipo da resposta que a API está enviando.

### 3.6 CÓDIGOS DE STATUS HTTP

As respostas devem incluir, além da mensagem, o código de status HTTP de acordo com o tipo de resposta.

O uso correto de códigos de status nas respostas ajuda os desenvolvedores de aplicações, portanto, evite utilizar o status 200 para todos os tipos de resposta. Por outro lado, muitos códigos de status podem aumentar a complexidade da API desnecessariamente.

Em respostas com erro, sempre complemente o status com mensagens de retorno explicativas. Exemplo:

```
HTTP 400 Bad Request
{
  "msg_erro": "o formato de mensagem solicitado não é suportado"
}
```

Utilize os três grupos de códigos de resposta para indicar: sucesso (2xx), falha devido a algum problema do lado do cliente (4xx) ou falha devido a algum problema do lado do servidor (5xx). A seguir são listados os códigos HTTP mais comuns:

CÓDIGO DE STATUS*	DESCRIÇÃO
<b>Sucesso</b>	
200 – OK	Código de sucesso geral. Pode ser utilizado em qualquer situação bem sucedida.
201 – Created	Indica que um recurso foi criado. Usado como resposta para uma requisição POST.

204 – No Content	A requisição teve sucesso, mas não existe mensagem de resposta. Geralmente usado após uma requisição de DELETE.
206 – Partial Content	A resposta está incompleta. Usada com lista de recursos paginada.
<b>Erro do Cliente</b>	
400 – Bad Request	Código de erro geral, causado por uma requisição feita de maneira indevida.
401 – Unauthorized	Cliente/usuário não reconhecido. Utilizado quando a requisição para uma API protegida não contém uma credencial ou contém uma credencial inválida.
403 – Forbidden	O cliente/usuário informado não possui os privilégios necessários para acessar o recurso.
404 – Not Found	O recurso solicitado não existe.
415 – Unsupported Media Type	O tipo de dado solicitado não é suportado pela API.
429 – Too Many Requests	O usuário atingiu o limite de requisições.
<b>Erro do Servidor</b>	
500 – Internal Server Error	A requisição está formatada corretamente, mas um problema ocorreu no servidor.

\*Nomes e códigos seguem a RFC 7231 do IETF (<https://tools.ietf.org/html/rfc7231>)

Os códigos de status as serem utilizados devem ser definidos durante o design da API de acordo com a necessidade e as características das aplicações consumidoras.

### 3.7 LIMITES DE REQUISIÇÕES

As informações sobre os limites de requisições e contagem total definidas para uma API ou cliente devem estar disponíveis para os consumidores/clientes.

Caso um cliente extrapole o limite de requisições definido para uma determinada API, o servidor deve retornar o status HTTP **429 - Too Many Requests**. O servidor poderá retornar no header (**Retry-After**) o número de segundos que se deve esperar até realizar a próxima requisição.

### 3.8 USO DE CACHE

Dependendo do tipo de requisição e da periodicidade da atualização dos dados, podemos utilizar soluções de cache para evitar requisições desnecessárias ao backend ou ao banco de dados, diminuindo, assim, o tempo de resposta da API.

O cache deve ser utilizado apenas em requisições de consulta e o tempo de validade do cache dependerá da validade da informação no backend. A chave de registro do cache (identificador único da requisição) deve ser a URL de requisição, contendo todos os seus elementos (host, path e query string).

Algumas soluções de API Management possuem recurso de cache nativo, o que diminui o volume de consultas ao backend quando ativado.



### 3.9 HYPERMEDIA


A API poderá oferecer links de hypermedia para facilitar a descoberta de recursos e operações por parte dos clientes. Cada chamada para a API pode retornar no corpo da mensagem de resposta todos os possíveis estados da aplicação a partir do estado atual.

HATEOAS (Hypermedia as the Engine of Application State) é o estilo arquitetural que define como implementar o uso de links em APIs REST.

Para implementar, links pode-se utilizar as anotações de Link da **RFC5988**. Exemplo:

```
GET
https://api.caixa/cadastro/v1/clientes/002

HTTP 200 OK
{
  "cliente":
  {
    "id": "002",
    "firstname": " Ana"
  },
  "links":
  [
    {
      "href": "https://api.caixa/cadastro/v1/clientes/002/enderecos",
      "rel": "endereços"
    }
  ]
}
```

	APIs RESTful – Definições e Orientações para uso na CAIXA	Página 17 / 23
		Data Emissão 16/06/2020

## 4 BOAS PRÁTICAS DE GOVERNANÇA DE APIS

### 4.1 AMBIENTE DE SANDBOX

Tão importante quanto o ambiente de produção de uma API é o ambiente de não produção (sandbox), pois é nesse ambiente que todos os desenvolvedores de aplicações consumidoras vão conectar suas aplicações durante as fases de implementação.

O ambiente de sandbox é caminho crítico do desenvolvimento das aplicações e, portanto, precisa ter as seguintes características:

- Possuir alta disponibilidade, de forma que as equipes possam desenvolver, testar e homologar suas aplicações a qualquer momento.
- Ser consistente, ou seja, as características de uma determinada versão de uma API em sandbox (árvore de recursos, mensagens, códigos de status etc.) devem ser as mesmas no ambiente de produção, de forma que os desenvolvedores tenham segurança durante o processo de implementação e implantação.

Dessa forma, não é recomendada a publicação de uma API de sandbox a partir do seu ambiente de desenvolvimento, pois trata-se de um ambiente sujeito a indisponibilidade e a inconsistência.

Na Caixa, recomendamos a utilização do ambiente de homologação (HMP) para a publicação das APIs de sandbox, por ser um ambiente consistente, mas é importante que esse ambiente se mantenha disponível.

Dependendo da criticidade da API, poderá ser criado um ambiente específico para o sandbox, utilizando, inclusive, a infraestrutura de produção.

### 4.2 ENDPOINT DE HEALTH CHECK

Uma boa prática para no desenvolvimento de APIs é a inclusão de um endpoint de **health check** que permita que, tanto aplicações consumidoras, quanto ferramentas de monitoração possam verificar a saúde da API a atuar de forma mais rápida na resolução de problemas.

É importante que o endpoint de health check implemente a verificação dos itens mais críticos da infraestrutura da API antes de responder com a situação do serviço.

O path **/health** deve ser utilizado de forma padronizada para todas as APIs, de forma a facilitar a descoberta pelas ferramentas de monitoração.

```
GET
https://api.caixa/cadastro/health
```

Alguns frameworks também utilizam os endpoints **/live** e **/ready**, que tem finalidades distintas e que podem ser utilizados em conjunto com o endpoint de health check.

## 5 SEGURANÇA

### 5.1 CORS (CROSS-ORIGIN RESOURCE SHARING)

Uma política de mesma origem (**same-origin policy**) é uma restrição que tem como objetivo impedir que aplicações Web pertencentes a um determinado domínio façam requisições para um recurso em outro domínio.

Os navegadores web possuem essa restrição, portanto, aplicações Web ou Móveis que utilizam AJAX para acessar APIs Web de outros domínios não irão funcionar a menos que as APIs implementem o padrão CORS. O CORS (Cross-Origin Resource Sharing) é um padrão W3C que permite que o servidor flexibilize o acesso aos seus recursos oriundos de outras origens.

Durante o fluxo do CORS, a aplicação encaminha uma pré-requisição para a API utilizando método HTTP OPTIONS e informando a origem, os cabeçalhos e o método HTTP da requisição original. Para tal, são utilizados os cabeçalhos HTTP **Origin**, **Access-Control-Request-Headers** e **Access-Control-Request-Method**, respectivamente.

Do outro lado, o provedor da API deve tratar a requisição e responder com as origens, cabeçalhos e métodos autorizados utilizando os cabeçalhos HTTP **Access-Control-Allow-Origin**, **Access-Control-Allow-Headers** e **Access-Control-Allow-Methods**, respectivamente.

Caso todos os aspectos da requisição original sejam autorizados, a requisição é finalmente encaminhada para o provedor da API.

No acesso às APIs que são expostas por meio de gateways de APIs, como é o caso do API Manager na Caixa, o uso de CORS não será permitido, pois o seu uso pode expor chaves de acesso e demais componentes de segurança utilizados para autorizar a aplicação cliente junto ao gateway.

### 5.2 UTILIZAÇÃO DE SSL

Ao usar SSL, que é uma comunicação criptografada, simplificam-se os esforços de autenticação sendo possível trabalhar com tokens de acesso simples ao invés de ter que assinar ou criptografar as mensagens a cada solicitação à API.

É importante que os desenvolvedores de aplicações consumidoras das APIs estejam cientes das implicações de se utilizar SSL, como por exemplo, a necessidade de importação dos certificados do servidor de API para dentro da aplicação consumidora.

### 5.3 CONTROLE DE ACESSO

Caso a API seja de acesso restrito, deve-se utilizar mecanismos de segurança para controlar o acesso aos recursos.

O protocolo a ser utilizado para a segurança das API é o **OAuth2 (Open Authentication)**, que trabalha com tokens de acesso no formato **JWT (JSON Web Token)**.

Cada requisição deve conter no cabeçalho HTTP **Authorization** um token que foi concedido ao cliente por um provedor de identidade mediante autenticação prévia. Exemplo:

```
Authorization Bearer YHD678983GHJGKUD.GDUKHDIYHDTUYTKHGA.JASTYUDAJDG
```

O provedor do recurso deve ser capaz de validar os tokens internamente (assinatura, data de expiração, emissor, aplicação cliente e usuário) ou pode submeter o token recebido para ser validado no provedor de identidade que o emitiu.


Caso o cliente faça uma requisição para um recurso protegido sem o token ou com um token inválido, o servidor deve retornar o status HTTP **401 – Unauthorized**. Caso o token seja válido, porém o usuário informado não tem acesso ao recurso, o servidor deve retornar o status HTTP **403 – Forbidden**.

Como alternativa à utilização do protocolo OAuth2, poderá ser usado o tipo Basic. Nesse caso, cada requisição deve conter no cabeçalho HTTP Authorization o usuário e a senha. O usuário e a senha devem estar entre chaves, concatenados por “:” (dois pontos) e convertidos para Base64, conforme abaixo:

```
Authorization Basic Base64({usuario}:{senha})
```

O uso de autenticação do tipo Basic não é recomendado e deve ser destinada apenas para identificação de usuário de serviço e nunca com usuário e senha de pessoas.

Outra forma de identificação das aplicações consumidoras das APIs é por meio de API Key, as quais são fornecidas e geridas por uma solução de API Management. Geralmente as API Keys trafegam dentro do cabeçalho HTTP Apikey.

	APIs RESTful – Definições e Orientações para uso na CAIXA	Página 20 / 23
		Data Emissão 16/06/2020

## 6 DOCUMENTAÇÃO DAS APIS

A documentação é parte essencial de uma API, pois permite que usuários e desenvolvedores possam conhecer e se engajar mais facilmente com ela.

Ela deve ser fácil de encontrar, de acesso público e conter exemplos completos de utilização (requisição/resposta).

O padrão a ser adotado para documentar as APIs é o OpenAPI, mas conhecido como Swagger. Análogo ao WSDL e XSD, o padrão Swagger define uma notação para documentar APIs RESTful.

O padrão OpenAPI é mantido pelo consórcio OAI (OpenAPI Initiative), sob coordenação da Linux Foundation e atualmente está na versão 3.0.

Um documento OpenAPI pode ser escrito no formato JSON ou YAML e a sua estrutura deve conter:

- Informações básicas da API – **info**
- Localização – **host**
- Protocolo – **schemes**
- Caminho base/domínio de negócio – **basePath**
- Recursos – **paths**
- Modelos de Dados – **definitions**
- Etiquetas – **tags**

Detalhamento das especificações pode ser encontrado em <http://spec.openapis.org/oas/v3.0.3> e a sua leitura é extremamente recomendada.


Por ser um padrão adotado em amplamente pelo mercado, existem ferramentas que geram uma visão amigável e interativa de uma API a partir de um documento Swagger, além de editores que auxiliam na construção do documento.

A CAIXA disponibiliza uma ferramenta interativa para a criação dos documentos no formato OpenAPI. A ferramenta, disponível em <https://design.portalapi.caixa/dashboard>, permite a edição simultânea, o compartilhamento de documentos, a geração de código fonte e vários outros recursos.

Algumas linguagens e IDEs também permitem a geração de documentos a partir do código fonte da API. Um exemplo é a especificação JSR 311 - JAX-RS (<http://jcp.org/aboutJava/communityprocess/final/jsr311/index.html>), que permite ao desenvolvedor implementar APIs Web em Java de maneira fácil e ainda documentá-las no padrão Swagger.

A seguir temos um exemplo de documento Swagger no formato YAML:

```
Info:
  title: API de Loterias
  description: API para a aposta de loterias
  version: '1.0.0'
  termsOfService: https:\\api.caixa\\terms
  contact:
    name: Fulano de Tal
    email: apicaixa@caixa.gov.br
  host: api.caixa
  basePath: loterias/v1
  schemes:
    - https
  consumes:
    - application/json
  produces:
    - application/json
  paths:
    /mega_sena:
      post:
        summary: Incluir uma aposta de mega-sena
        consumes:
          - application/json
        produces:
          - application/json
          - application/xml
        parameters:
          - in: body
            name: body
            schema:
              $ref: "#/definitions/mega-sena"
        responses:
          '200':
            description: Aposta incluída
          '400':
            description: Parâmetros de aposta inválidos
  definitions:
    mega-sena:
      type: object
      properties:
        qtde-numeros:
          type: int
        numero:
          type: int
```

	APIs RESTful – Definições e Orientações para uso na CAIXA	Página 22 / 23
		Data Emissão 16/06/2020

## 7 REFERÊNCIAS

- Web API Design. Acesso em Março de 2017, disponível em: <https://pages.apigee.com/rs/351-WXY-166/images/ebook-2013-03-wad.pdf>
- API Facade Pattern. Acesso em Março de 2017, disponível em: <https://pages.apigee.com/rs/351-WXY-166/images/api-facade-pattern-ebook-2012-06.pdf>
- A World About Microservice Architectures and SOA. Acesso em Março de 2017, disponível em: <http://soacommunity.com/index.php/magazine/articles/236-articles-microservicesweir>
- Projeto de API RESTful – OCTO Guia de Referência Rápida. Acesso em Abril de 2017, disponível em: <http://blog.octo.com/wp-content/uploads/2015/12/RESTful-API-design-OCTO-Quick-Reference-Card-PT-2.3.pdf>
- Projetando uma API REST. Acesso em Abril de 2017, disponível em : <http://blog.octo.com/pt-br/projetando-uma-api-rest/>
- Web Link. Acesso em Abril de 2017 disponível em: <https://tools.ietf.org/html/rfc5988>
- Swagger Specification. Acesso em Abril de 2017, disponível em: <http://swagger.io/specification/>
- Editor Swagger. Acesso em Abril de 2017, disponível em: <http://editor.swagger.io>
- REST API Tutorial. Acesso em Dezembro de 2017, disponível em: <http://restapitutorial.com>