Melhores práticas sobre utilização de

REST e APIs na Celepar

# 

# 

# 

Sumário

[Autor](#_nh1olokqjh6)

[Colaboradores](#_25odjcz30t)

[Revisores](#_3zxzcwpnlbpq)

[Quem deve ler essa documentação](#_i0rv33yyweaf)

[Motivação](#_lre6hopn88ob)

[Introdução](#_6w1mliw7fycb)

[O que é REST?](#_ttr3kg4ztx7n)

[Interface uniforme](#_iwz4vjp6qoh7)

[Baseada em Recursos](#_e7jsh257pa5s)

[Manipulação de recursos por meio de representações](#_quzihd4gqqaz)

[Mensagens Auto-descritivas](#_espcndboi7og)

[Hipermídia como o Mecanismo de Estado da Aplicação (HATEOAS)](#_xv8nwqjxe66e)

[Inexistência de estado(Stateless)](#_o8ju4ta126tj)

[Controle de Cache](#_7vghbmu8qkc0)

[Cliente/Servidor](#_qef6dvp1dxeh)

[Camadas](#_gzkmcjq467xq)

[Código sob demanda](#_9dzkljmh0k9v)

[Organização das APIs](#_r1812pxfr54p)

[Domínios](#_v0rl58a7xwf1)

[Sistema ou módulo](#_232nd2yq4e4p)

[Versionamento](#_ou383yn5u1be)

[Considerações sobre versionamento no cabeçalho](#_ebxrmbh0hg29)

[Recursos](#_6n4ywcb9ws4)

[Identificadores](#_6swoq9j6l9sl)

[Identificadores com URNs](#_d1rgsdw9p33v)

[Verbos HTTP RFC 2616](#_br63iv4laezl)

[Verbos WebDAV RFC 4918](#_tic5kfbew2wh)

[Parâmetros de acordo com verbos](#_jo013sc7x4q)

[GET](#_1u8p1xljl6)

[DELETE e PUT](#_ifexwaaeo5f6)

[POST e PUT](#_nurl3to82mva)

[Considerações com relação ao uso dos verbos e identificadores](#_3u8wm535td7q)

[Formatos](#_9zvx8rsbg61l)

[Exceções](#_icqs0jilv4z7)

[Considerações sobre o uso do formato na URI](#_fgleitfa3gqd)

[snake\_case vs camelCase](#_p4nylss7nlqh)

[Códigos de resposta HTTP](#_6p7olrd7njg9)

[Mensagens de sucesso 2xx](#_k6xgin39q3m6)

[Mensagens de redirecionamento 3xx](#_gzq77ql2jxst)

[Mensagens de erro no cliente 4xx](#_q50w3nb6pqjj)

[Mensagens de erro no servidor 5xx](#_wac3bwtxt2rb)

[Catálogo](#_6kv943oyxwnp)

[Swagger](#_1h1np28r19mi)

[Padrões existentes](#_wgq5tvnb4g51)

[Recomendações com relação a segurança](#_l0ic52v5qwv4)

[Exemplos de implementações](#_ss53r3u7d14p)

[Recursos Adicionais](#_rcfpasaw1g5r)

[Livros](#_9iq9edyvy018)

[Websites](#_to5da72bvrc)

[Apresentações](#_6gfp98wbd5ae)

# Autor

André Alexandre Avila (COSIN-C3).

# Colaboradores

Essa documentação contou com o apoio de:

Daniel Elias de Liz (COSIN-C3)

Tarso Dutra B de Queiroz (COTAP)

# Revisores

<<Alguém se habilita>>

# Quem deve ler essa documentação

Este documento de melhores práticas é destinado a desenvolvedores que estejam interessados em criar serviços Web RESTful que proporcionam alta confiabilidade e consistência em várias tecnologias. Seguindo estas orientações, os serviços estarão bem posicionados e aderentes aos padrões de mercado.

# Motivação

Diversos tipos de arquitetura (SOA, ROA, WOA, ESB, MicroServices, APIs, etc) podem ser aplicados, ou estão relacionados ao REST. De fato, a organização e a administração fica muito mais fácil quando há um padrão definido. Sem isso, é provável que o esforço de adaptação entre cliente e servidor seja difícil e até traumático. Essa documentação segue dois princípios básicos, facilidade de desenvolvimento e facilidade de uso. Lembrando que web services são feitos para máquinas, APIs são feitas para humanos.

# Introdução

O termo REST (REpresentational State Transfer)[[1]](#footnote-0) se originou numa tese de doutorado de um dos criadores do **HTTP** (Roy Thomas Fielding). É um paradigma arquitetônico, muito utilizado para o desenvolvimento de serviços web. Sua implementação se dá utilizando principalmente os verbos HTTP (POST, GET, PUT e DELETE), cabeçalhos e URIs para definir recursos, métodos e ações.

Atualmente estamos vendo uma explosão na quantidade de serviços para a web, já existiam diversas formas de comunicação e outras foram criadas para comportar a demanda por um mundo conectado. Quanto mais o tempo passa, vemos que o desenvolvimento de APIs tem uma visão mais acessível do que era esperada no passado, assim o REST vem se tornando o padrão de fato para empresas, pessoas e ferramentas na parte de serviços.

Existem inúmeros recursos sobre as melhores práticas para a criação de serviços Web RESTful (consulte a seção Recursos no final deste documento). Muitos dos recursos disponíveis são conflitantes, dependendo da data que eles foram escritos. Além disso, a leitura e compreensão de vários livros sobre o assunto, a fim de implementar novos serviços pode ser complexa. A fim de facilitar a absorção rápida e compreensão de conceitos RESTful, sem exigir a leitura de pelo menos três a cinco livros sobre o assunto, este guia é feito para acelerar o processo de condensação do REST, com melhores práticas, convenções e apenas os principais pontos sem muita discussão.

REST é mais uma coleção de restrições do que é um conjunto de normas. Além das suas mais abrangentes seis restrições, nada é definido. Há claro, "melhores práticas" e padrões de fato, mas tudo está em constante evolução.

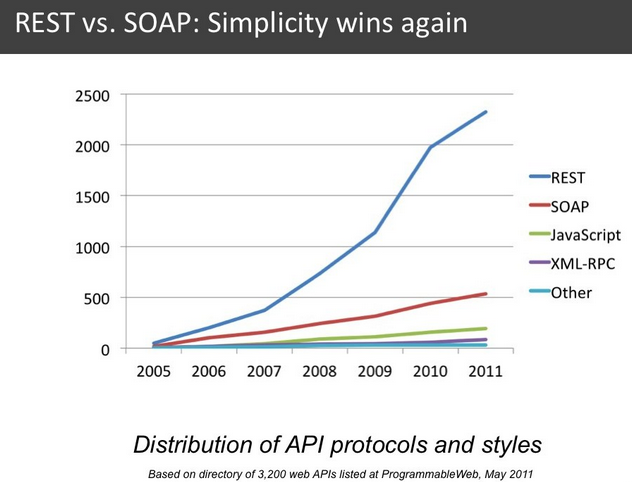
Este documento reúne informações disponíveis em outras fontes, adaptando-os com a experiência adquirida através do uso, discussões e pesquisas.

Ainda há um debate considerável sobre se REST é melhor do que o SOAP (e vice-versa) e talvez, ainda há razões para criar serviços SOAP. Mas não se preocupe, este documento não vai gastar muito tempo discutindo méritos relativos. Em vez disso, como a tecnologia e a indústria seguem em frente, vamos continuar com o pressuposto de que alavancar REST é a melhor prática atual para a criação de serviços Web.

A primeira seção apresenta uma visão geral do que é o REST, as suas limitações e que o torna único. Em seguida, fornecemos algumas dicas e conceitos de serviços REST. As secões posteriores descrevem como será organizado o catálogo de APIs, Swagger e sobre padrões possíveis de serem implementados.

Recomendamos também acompanhar o crescimento da internet e seus padrões[[2]](#footnote-1), afinal, na era da “internet das coisas” tudo está em constante evolução.

Numa análise das crescentes tecnologias utilizadas na comunicação entre os serviços web, podemos observar a adoção do REST.



Fonte: <http://www.infoq.com/news/2011/06/Is-REST-Successful>

# 

# 

# O que é REST?

A arquitetura REST descreve seis restrições que foram originalmente descritas por Roy Fielding em sua tese de doutorado. Elas definem a base do estilo RESTful.

As seis restrições são:

* Interface uniforme
* Inexistência de estado
* Controle de cache
* Cliente/Servidor
* Camadas
* Código sob demanda

De forma mais detalhada:

## Interface uniforme

A restrição de interface uniforme define a interface entre clientes e servidores. Ela simplifica e desacopla a arquitetura do aplicativo, permitindo que cada parte possa evoluir de forma independente. Os quatro princípios base da interface uniforme são:

##### Baseada em Recursos

Os recursos individuais são URIs utilizados nas solicitações como identificadores de recursos. Os próprios recursos devem conceitualmente separar as representações que são retornadas para o cliente. Por exemplo, o servidor não envia seu banco de dados, mas sim, um pouco de XML, JSON ou algo que representa alguns registros do banco de dados, dependendo dos detalhes da solicitação e implementação do servidor.

##### Manipulação de recursos por meio de representações

Quando um cliente tem uma representação de um recurso, incluindo quaisquer metadados anexados, o cliente tem informações suficientes para modificar ou excluir um recurso no servidor, desde que tenha permissão para fazê-lo.

##### Mensagens Auto-descritivas

Cada mensagem inclui informações suficientes para descrever como processar a mensagem. Por exemplo, qual parser utilizar, pode ser especificado por um tipo de mídia Internet (anteriormente conhecido como um tipo MIME). As respostas também indicam explicitamente qual a sua capacidade de se fazer cache.

##### Hipermídia como o Mecanismo de Estado da Aplicação (HATEOAS)

O Cliente entrega seu o estado via conteúdo(body), parâmetros query-string, cabeçalhos de solicitação e a URI solicitada (o nome do recurso). Os serviços entregam seu estado para clientes via conteúdo(body), códigos de resposta e cabeçalhos de resposta. Isso é tecnicamente conhecido como a hipermídia (ou hiperlinks dentro de hipertexto).

Além da descrição acima, HATEOAS também significa que, se necessário, as ligações devem estar contidas no conteúdo devolvido (ou cabeçalhos) para a recuperação do próprio objeto ou objetos relacionados. Em padrões existentes há uma lista de padrões que reforçam o uso do conceito HATEOAS, como JSON-HAL e JSON-LD.

A interface uniforme que o serviço REST deve fornecer é fundamental para o seu design.

## Inexistência de estado(Stateless)

Como o REST é um acrônimo para REpresentational State Transfer, não ter estado é fundamental. Essencialmente, o que isto significa é que o estado necessário para lidar com o pedido está contido dentro do próprio pedido, seja como parte da URI, parâmetros query-string, conteúdo ou cabeçalhos. A URI identifica o recurso e o corpo contém o estado (ou mudança de estado) desse recurso. Em seguida, depois que o servidor faz o seu processamento, o estado adequado, ou a identificação do estado, são enviados ao cliente via cabeçalhos, códigos de status e conteúdo.

Normalmente na web, estamos acostumados a guardar coisas dentro de um recipiente que fornece o conceito de "sessão" e que mantém o estado através de múltiplas solicitações HTTP. No REST, o cliente deve incluir todas as informações para o servidor atender a solicitação e reenviar o estado é necessário caso o mesmo necessite se estender por vários pedidos. Não manter o estado permite maior escalabilidade desde que o servidor não tem que armazenar, atualizar ou comunicar o estado da sessão. Além disso, balanceadores de carga, não precisam se preocupar com autorização de sessão em sistemas desse tipo.

Então, qual é a diferença entre estado e recurso? Considere o estado como sendo os dados que podem variar de acordo com o cliente e solicitação. Agora o recurso, por outro lado, é constante em cada cliente que o solicite.

Há casos em que não se encaixam no princípio stateless, como 3 legged OAuth, limitadores de chamadas de API, etc, no entanto, essas técnicas fazem todos os esforços para garantir que o estado da aplicação seja mínimo.

## Controle de Cache

Como nos navegadores, os clientes podem armazenar em cache as respostas. Elas portanto, devem, implícita ou explicitamente, definir como serão ou não armazenadas, para impedir que clientes reutilizem dados obsoletos ou inadequados em resposta a outros pedidos. Ter algum tipo de cache parcial ou completo, elimina algumas interações cliente-servidor, melhorando ainda mais a escalabilidade e o desempenho.

## Cliente/Servidor

A interface uniforme separa clientes de servidores. Esta separação de interesses significa que por exemplo, os clientes não estão preocupados com o armazenamento de dados, que permanece interno para cada servidor, de modo que a portabilidade do código do cliente é melhorada. Os servidores não estão preocupados com a interface do usuário ou o estado do usuário, de modo que os servidores podem ser mais simples e mais escaláveis. Os servidores e clientes podem também ser substituídos e desenvolvidos de forma independente, enquanto a interface não é alterada.

## Camadas

Um cliente não pode normalmente dizer se ele está conectado diretamente ao servidor final, ou a um intermediário ao longo do caminho. Servidores intermediários pode melhorar a escalabilidade do sistema, permitindo o balanceamento de carga e proporcionando caches compartilhados. Camadas também podem aplicar políticas de segurança.

## Código sob demanda

Os servidores são capazes de estender temporariamente ou personalizar a funcionalidade de um cliente, transferindo lógica para ele, onde o mesmo pode interpretar. Exemplos disso podem incluir componentes como códigos javascript ou applets Java.

Cumprindo essas restrições e em conformidade, portanto, com o estilo arquitetonico REST, permitirá qualquer tipo de sistema distribuído, ter propriedades emergentes desejáveis, tais como desempenho, escalabilidade, simplicidade, modificabilidade, visibilidade, portabilidade e confiabilidade.

NOTA: A única restrição opcional da arquitetura REST é código sob demanda. Se um serviço viole qualquer outra restrição, não pode estritamente ser referido como RESTful.

Ser REST é definido como RESTful (isto é igual a REST + sufixo FULL).

# Organização das APIs

Seguindo a ordem: domínios, sistema ou módulo, versionamento, recursos, identificadores, verbos, formatos e códigos de resposta.

## Domínios

As APIs devem estar em domínios específicos (exclusivamente ou como atalho), sendo <subdomínio>.api.pr.gov.br (caso seja de uso público) e <subdomínio>.api.eparana.parana (no caso de APIs para uso interno), o **subdomínio** é referente ao “dono” da informação, sendo a principal categoria.

Exemplos de domínio:

sefa.api.pr.gov.br, detran.api.pr.gov.br, seae.api.pr.gov.br, sesa.api.pr.gov.br.

sefa.api.eparana.parana, detran.api.eparana.parana, seae.api.eparana.parana,sesa.api.eparana.parana.

## Sistema ou módulo

Descrever o sistema ou o objeto principal da API, deve-se usar nomes que o usuário tenha associação com sua vida e não siglas de sistema.

Exemplos: /**icms, /contribuinte,** /**ipva**, /**habilitacao, /parcelamento,** etc.

**NÃO** usar por exemplo /**sgxpto, /sgpep, /tapa, /sifu**

## Versionamento

O versionamento é separado em 2 níveis, indicado com “v” na frente e separado por um ponto. Deve haver também uma abreviação de versão, por exemplo /v1/ aponta para o último lançamento estável entre a versão **/v1.0/** e **/v1.99/**.  
Exemplos:

exemplos:

/icms/**v1.0**

/ipva/**v1.6**

/ipva/**v1** (apontando para a v1.6)

##### Considerações sobre versionamento no cabeçalho

Existem materiais que recomendam o uso de versionamento no cabeçalho Accept, outros a indicação na URI. Foi optado pela URI pois, facilita os testes e deixa mais prático o uso da API. Mas salientamos que qualquer uma dessas formas não afeta nenhuma restrição do padrão REST.

## Recursos

Após a versão, deve-se indicar o Recurso principal a ser invocado, para simplificar o processo deve-se pensar no plural, afinal, o recurso contém um conjunto, ou no caso de subrecursos, podem ser enviados na sequência da identificação do recurso principal.

Exemplos:

/habilitacao/v1.0/**condutores**

/contribuintes/v1.0/**estabelecimentos**/12345/**socios**.

## Identificadores

Se há o envio de algum identificador, o mesmo deve ser inserido junto na URI, ou no caso de subobjetos pode-se enviar 2 ou mais identificadores.

Exemplos:

/habilitacao/v1.0/condutores/**1**(no caso o número 1 poderia ser a CNH por exemplo)

/icms/v1.0/estabelecimentos/**1**/socios/**2**(Neste caso é obtidos os dados do sócio 2 da empresa 1)

## 

## 

## Identificadores com URN

Quando um serviço pode receber a entrada de diferentes tipos de identificadores há a possibilidade de uso de URN[[3]](#footnote-2) para especificar o tipo de dados informado, por exemplo urn:cpf:999999999 pode indicar um CPF, urn:cnpj:9999999999 para um CNPJ. URNs podem ser hierárquicas e indicar mais coisas do que simples campos, como uma especialização por exemplo urn:pr:cadicms:99999999 pode indicar uma inscrição estadual do Paraná já que o número de inscrição estadual tem regras diferentes em cada estado.

## Verbos HTTP[[4]](#footnote-3) RFC 2616

A utilização dos verbos HTTP deve respeitar o seu devido uso:

**GET** - Obtenção do recurso.

**POST** -Criação do recurso.

**PUT** - Atualização do recurso caso exista, ou a criação do recurso com o id especificado.

**DELETE** - Exclusão do recurso.

**OPTIONS** - Utilização para definir verbos autorizados e CORS.

**PATCH -** Alteração parcial de um recurso, por exemplo alteração de apenas 1 campo de um recurso já existente.

**HEAD** - Meta informação, tamanho do arquivo, etc. No nosso caso não iremos definir um padrão.

Material relacionado:

<http://restpatterns.org/HTTP_Methods>,

<http://www.restapitutorial.com/lessons/httpmethods.html>

## Verbos WebDAV RFC 4918

Não vamos abordar o uso de verbos WebDAV, pois normalmente os clientes HTTP seguem o padrão RFC 2616.

## 

## Parâmetros de acordo com verbos

Para parâmetros adicionais, devemos enviar da seguinte forma:

##### **GET**

Os parâmetros podem ser enviados por query string ou na URI com o identificador.

Parâmetros que não pertecem ao recurso a ser consultado, recomendamos o uso do parâmetro com \_(underscore) na frente, indicando de forma clara que não pertence ao recurso. Exemplos \_fields, \_limit, \_offset, \_page,

Exemplos:

/servico/v1.0/usuarios**?nome=Jose&sobrenome=Silva** (Exemplo de busca)

/servico/v1.0/usuarios**?nome=Jose&\_fields=nome,cpf,email&\_limit=10** (Exemplo de busca)

/servico/v1/usuarios/1

##### DELETE e PUT

É necessário enviar o identificador do recurso e no caso do método PUT, além do identificador deve-se enviar dados no corpo da requisição, melhor explicado no tópico a seguir.

Exemplos: **DELETE** /servico/v1.0/usuarios/**1**

##### POST e PUT

No corpo da requisição deve-se passar o formato de acordo com o cabeçalho “Content-Type” definido, exemplos form-urlencoded, multipart-formdata, application/json ou application/xml.

##### Considerações com relação ao uso dos verbos e identificadores

O desenvolvedor deve tomar cuidado com relação ao uso dos verbos, dividindo a responsabilidade entre eles, por exemplo, o verbo GET é responsável ela pesquisa e obtenção da informação (caso inclua o identificador). Outros verbos são responsáveis, pela inclusão (POST), alteração (PUT) e exclusão (DELETE). Para efetuar os procedimentos de busca e exclusão por exemplo, é necessário efetuar duas chamadas REST, uma GET para obter o identificador do recurso e outra DELETE para a exclusão.

|  |  |  |
| --- | --- | --- |
| **Verbo** | **Coleção inteira(Ex. /usuarios)** | **Item específico (ex. /usuarios/{id})** |
| **GET** | Status 200, Retorna toda coleção, com paginação caso possível. | Retorna item específico, caso não exista retorna Status 404. |
| **PUT** | Status 404\* ou 405\*\*, Não é correto alterar toda a coleção. | Status 200\* ou 204\*\* e caso não exista cria o recurso. |
| **POST** | Status 201, O cabeçalho Location deve retornar a URI do recurso com o identificador  Ex. Location: usuarios/1 | Status 404\* ou 405\*\*. |
| **DELETE** | Status 404\* ou 405\*\*, não é correto excluir toda coleção. | Status 200 e caso não exista retorna Status 404. |

\* Há implementações que consideram um recurso a dupla (verbo + caminho(URI)), neste caso para facilitar a implementação o status de retorno é 404 (Ex. Micro frameworks REST Sinatra style).

\*\* Também há implementações que consideram primeiramente o caminho para depois chegar ao verbo, normalmente neste caso o retorno é status 405(JAX-RS).

# Formatos

Para facilitar a comunicação cliente/servidor, uma prática muito usada é a negociação de conteúdo. Nela se define via cabeçalhos qual formato se está enviando (**Content-Type**) e qual se deseja receber (**Accept**). Também recomendamos o uso do **JSON** (menor e mais simples) e **XML** (tem maior interoperabilidade).

Por padrão, consideramos que o formato retornado será JSON (quando o cabeçalho “Accept” não for enviado). Caso a API tenha outros formatos pode-se solicitar pelo cabeçalho “Accept”, por exemplo GET /cadastro/v1.0/usuarios/1 enviando o cabeçalho “Accept: application/xml” se recebe os dados no formato **XML**.

Caso seja necessário utilizar outros padrões, o desenvolvedor da **API** é livre para utilizá-los, apenas deve respeitar os devidos media types segundo a organização **IANA[[5]](#footnote-4).**

## Exceções

Há casos onde o serviço apenas retorna um arquivo binário, como uma imagem, PDF, ZIP, etc. Nessa situação opta-se por apenas retornar o devido formato ou armazenar dentro de um valor string usando o padrão base64.

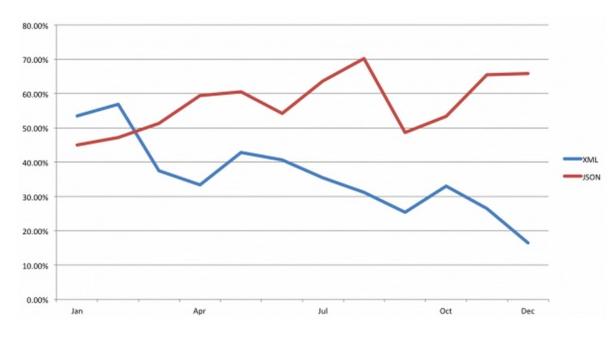
## Considerações sobre o uso do formato na URI

Há versões de APIs que suportam a solicitação do tipo a ser recebido na URI(Exemplo usuarios.json), no entanto pode haver uma interpretação equivocada quando se aprofunda nos objetos a serem usados, exemplo /**usuarios.json**/1/compras/123 ou /usuarios/1/compras/**123.json**, sendo melhor a descrição no cabeçalho “Accept”, pois acaba com essa confusão.

## snake\_case vs camelCase

Apesar das tecnologias já terem seus padrões para utilização de atributos e variáveis, recomendamos fortemente o uso do snake\_case ou under\_score, pois segundo pesquisas[[6]](#footnote-5) facilita a leitura dos atributos em até 80%. Podemos também afirmar que as maiores APIs já utilizam esse padrão.

Gráfico de novas APIs 2013



Fonte: programmableweb.com

# Códigos de resposta HTTP

No protocolo HTTP cada tipo de retorno pode ter um código de reposta específico, esses códigos facilitam o entendimento do lado cliente, de forma a efetuar algum tipo de ação ou simplesmente retornar alguma mensagem ao usuário, abordamos os principais códigos de resposta descritos na RFC 2616 e 6585:

##### Mensagens de sucesso 2xx

**200** - OK - A requisição foi concluída com sucesso.

**201** - Created - A requisição foi aceita e um recurso foi criado, deve ser acompanhada com o cabeçalho “Location” indicando a URI do recurso.

**202** - Accepted - A requisição foi aceita, mas seu processamento não foi concluído. Esse retorno é comum no caso de processamento em lote, ou fila de tarefas, onde a requisição é processada posteriormente.

**204** - No Content - A requisição foi aceita mas não precisa retornar um conteúdo ou no há conteúdo para retornar.

**206 -**  Partial Content - Informa que o conteúdo retornado é apenas uma parte do existente.

##### Mensagens de redirecionamento 3xx

**301** - Moved Permanently - O recurso não está mais presente na URI solicitada. Pode conter o cabeçalho Location indicando a URI definitiva. Pode ser usado para indicar quando uma versão se torna obsoleta, indicando no cabeçalho Location a última versão corrente.

##### Mensagens de erro no cliente 4xx

**400** - Bad Request - A solicitação está mal-formada ou com alguma informação faltante e não foi aceita

**401** - Unauthorized - Usuário não autorizado, caso o usuário se autentique a mesma chamada pode ser enviada.

**402** - Payment Required - Para enviar uma chamada ao serviço é necessário aderir a algum plano ou efetuar algum tipo de pagamento. Normalmente esse status não é usado.

**403** - Forbidden - O serviço entendeu a requisição, mas se nega a atender. A autorização não importa muito nesse caso e mesmo que seja refeita será recusada. Caso seja enviada a requisição usando o método HEAD o serviço deve informar o motivo da recusa. Esse status é retornado normalmente quando o usuário não tem permissão de acesso.

**404** - Not Found - O recurso requisitado não existe. Tomar cuidado para a mensagem de erro, pois deve retornar no mesmo padrão pedido(Accept). Caso o retorno venha com o Content-Type “text/html”, há a possibilidade que o serviço não esteja funcionando.

**405** - Method not Allowed - O método requisitado para esse recurso não foi aceito. A resposta deve conter o cabeçalho “Allow” contendo os métodos com permissão.

**406 -** Not Acceptable - O recurso enviado não pode ser aceito.

**409** - Conflict - O estado do recurso(enviado) está em conflito com o atual(no servidor) e não pode ser aceito.

**413** - Request Entity Too Large - A entidade enviada na requisição é maior do que o tamanho permitido.

**415** - Unsupported Media Type - O formato de dados enviado não é suportado pelo serviço. Isso seguindo o cabeçalho Content-Type. Exemplo, o serviço suporta “application/json”, mas o usuário enviou o formato “application/xml”.

**429** - Too Many Requests - O usuário chegou no limite na quantidade de requisições disponíveis. Normalmente isso ocorre quando há no serviço uma limitação na quantidade de acessos e a mesma foi alcançada. Esse retorno é muito usado no recurso de “Rate-Limiting” para limitar a quantidade de acessos.

##### Mensagens de erro no servidor 5xx

**500** - Internal Server Error - Indica que houve um erro não esperado no serviço.

**501** - Not Implemented - O serviço especificao não foi implementado.

**502** - Bad Gateway - O servidor que repassa o acesso ao serviço(Proxy) recebeu um conteúdo inválido do servidor de destino.

**503** - Service Unavailable - Serviço indisponível por conta de alguma manutenção ou sobrecarga. Pode retornar o cabeçalho "Retry-After" para indicar quando o sistema vai retornar a funcionar.

**504** - O servidor que repassa o acesso ao serviço(Proxy) não conseguiu conectar.

**505** - HTTP Version Not Supported - a Versão HTTP usada não é suportada pelo servidor.

# Catálogo

Criação do domínio api.eparana.parana (denominado API Interna) e api.pr.gov.br (denominado API Externa) e api.eparana.parana(API interna) com a documentação das APIs.

Exemplo de catálogo: <https://developers.google.com/> e <https://developers.google.com/products/>

Cada produto pode ser separado por detentor da informação e por finalidade.

## Swagger

Após uma análise entre diversos tipos de documentação para APIs, o Swagger se mostrou o padrão mais usado, mais flexível, mais maduro, independente e com uma maior preocupação com sua evolução. Por isso o consideramos o padrão ideal para a construção de documentação e testes de APIs. O Swagger possibilita a documentação junto ao código ou separado, dessa forma, o desenvolvedor pode optar como manterá a documentação, além disso, ele suporta o uso de tokens, dados de formulário, JSON e XML.

A versão atual do Swagger é a 1.2, mas já está em discussão a versão 2.0.

Exemplos de uso do Swagger:

<https://dsp-sandman1.cloud.dreamfactory.com/swagger/>

<https://desenvolvedores.extra.com.br/api-portal/docs/apilojistav1/api-reference/swagger.html>

# Padrões existentes

Com o crescimento das APIs e principalmente do JSON, surgiram formas mais dinâmicas para representar o estado da aplicação e atender a restrição HATEOAS, além de outras caratecterísticas antes não padronizadas no JSON. Segue uma lista de padrões ainda em discussão e padrões já estabelecidos para construção de APIs com hipermídia.

**Atributos com link e relacionamentos (do mais simples ao mais complexo):**

JSON-HAL (RFC Draft\*:<http://tools.ietf.org/html/draft-kelly-json-hal>) -> mais prático

JSON-LD (W3C Recomendation: <http://www.w3.org/TR/json-ld/>)

Siren (<https://github.com/kevinswiber/siren>)

JSON-Collection (<http://amundsen.com/media-types/collection/>)

**Schema:**

JSON-Schema (<http://tools.ietf.org/html/draft-zyp-json-schema>)

**Geo:**

GeoJSON (<http://geojson.org/>, <http://en.wikipedia.org/wiki/GeoJSON>)

**Assinatura:**

JSON-JOSE (RFC Draft\*: <http://tools.ietf.org/html/draft-ietf-jose-json-web-signature>)

JSON-JWT JSON web token (<https://tools.ietf.org/html/draft-ietf-oauth-json-web-token>)

*\* Lembrando que Drafts ainda podem sofrer alterações.*

# 

# Recomendações com relação a segurança

Para a comunicação é altamente recomendado o uso de criptografia entre as partes e além disso é necessário algum padrão para a autenticação. Na parte de autenticação, diversas empresas se reuniram para chegar a um único padrão a fim de garantir um elo entre usuário, aplicação e serviços. Esse padrão é conhecido como OAuth. Atualmente o padrão está na versão 2.0 e incorpora a comunicação com outros padrões como SAML2 e JWT opcionalmente.

# Exemplos de implementações

Em breve.

JAVA

PHP

# Recursos Adicionais

## Livros

REST API Design Rulebook, Mark Masse, 2011, O’Reilly Media, Inc.

RESTful Web Services, Leonard Richardson and Sam Ruby, 2008, O’Reilly Media, Inc.

RESTful Web Services Cookbook, Subbu Allamaraju, 2010, O’Reilly Media, Inc.

REST in Practice: Hypermedia and Systems Architecture, Jim Webber, et al., 2010, O’Reilly Media, Inc.

APIs: A Strategy Guide, Daniel Jacobson; Greg Brail; Dan Woods, 2011, O'Reilly Media, Inc.

RESTful Web APIs, Leonard Richardson, Mike Amundsen, Sam Ruby, 2013, O’Reilly Media, Inc.

## Websites

<http://www.restapitutorial.com>

<http://restpatterns.org/>

<http://www.toddfredrich.com>

<http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm>

<http://www.json.org/>

<https://github.com/tfredrich/DateAdapterJ>

<http://openid.net/developers/specs/>

<http://oauth.net/documentation/spec>/

<http://www.json.org/JSONRequest.html>

<http://labs.omniti.com/labs/jsend>

<http://jwt.io>

<http://enable-cors.org/>

<http://www.odata.org/documentation/uri-conventions#FilterSystemQueryOption>

<http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven>

<https://developer.linkedin.com/apis>

<http://developers.facebook.com/docs/reference/api/>

<https://dev.twitter.com/docs/api>

<https://developers.google.com/>

<http://apigee.com/about/resources>

<http://momentjs.com/>

<http://www.datejs.com/>

## Apresentações

<http://www.infoq.com/presentations/api-design-advice>

<http://www.infoq.com/presentations/API-design-mistakes>

<http://www.infoq.com/presentations/API-Business-Models>

<http://www.infoq.com/presentations/api-security-federation-patterns>

<http://www.infoq.com/presentations/API-Humans>

<http://www.infoq.com/presentations/API-gov-uk>[[7]](#footnote-6)

<http://www.infoq.com/presentations/API-Government>[[8]](#footnote-7)

<http://www.infoq.com/presentations/REST-API-HATEOAS>

<http://www.infoq.com/presentations/web-api-html>

<http://www.infoq.com/presentations/Building-Hypermedia-API>

<http://www.infoq.com/presentations/Web-API-Evolution>

<http://www.infoq.com/presentations/OAuth>

<http://www.infoq.com/presentations/Techniques-for-Scaling-the-Netflix-API>

<http://www.infoq.com/presentations/BPM-with-REST>

**Apresentações em português**

<http://www.infoq.com/br/presentations/rest-arquitetura-abril>

<http://www.infoq.com/br/presentations/evoluindo-uma-arquitetura-soundcloud>

1. <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm> [↑](#footnote-ref-0)
2. <https://www.mnot.net/blog/2014/06/07/rfc2616_is_dead> e HTTP 2.0 draft <http://http2.github.io/> [↑](#footnote-ref-1)
3. <http://tools.ietf.org/html/rfc2141> [↑](#footnote-ref-2)
4. <http://www.infoq.com/br/news/2013/05/idempotent> [↑](#footnote-ref-3)
5. <http://www.iana.org/assignments/media-types/media-types.xhtml> [↑](#footnote-ref-4)
6. <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?reload=true&tp=&arnumber=5521745> [↑](#footnote-ref-5)
7. APIs no governo britânico [↑](#footnote-ref-6)
8. APIs no governo americano [↑](#footnote-ref-7)