API’s REST

Padrões, Boas Práticas e Instruções

[Versão do documento 4](#_Toc77831264)

[Objetivos 5](#_Toc77831265)

[Observações sobre especificidades por tecnologia 6](#_Toc77831266)

[Verbos por ação 7](#_Toc77831267)

[GET 7](#_Toc77831268)

[POST 7](#_Toc77831269)

[PUT 7](#_Toc77831270)

[PATCH 8](#_Toc77831271)

[PUT ou PATCH? 8](#_Toc77831272)

[DELETE 9](#_Toc77831273)

[HEAD e OPTIONS 9](#_Toc77831274)

[Respostas HTTP (*HTTP* *responses*) 10](#_Toc77831275)

[Http Status Codes 10](#_Toc77831276)

[Respostas de sucesso (2XX) 10](#_Toc77831277)

[200 – Success 10](#_Toc77831278)

[201 - Created 10](#_Toc77831279)

[202 – Accepted 11](#_Toc77831280)

[204 – No Content 11](#_Toc77831281)

[Mensagens de redirecionamento (3XX) 11](#_Toc77831282)

[302 – Found 11](#_Toc77831283)

[Respostas de erro do cliente (4XX) 11](#_Toc77831284)

[400 - Bad Request 12](#_Toc77831285)

[401 – Unauthorized 12](#_Toc77831286)

[403 – Forbidden 12](#_Toc77831287)

[404 – Not found 12](#_Toc77831288)

[409 – Conflict 12](#_Toc77831289)

[410 – Gone 13](#_Toc77831290)

[422 - Unprocessable Entity 13](#_Toc77831291)

[Respostas de erro do servidor (5XX) 13](#_Toc77831292)

[500 – Internal server error 13](#_Toc77831293)

[510 – Not implemented 14](#_Toc77831294)

[Corpo da resposta 14](#_Toc77831295)

[Nomenclatura 14](#_Toc77831296)

[Autenticação vs. Autorização 16](#_Toc77831297)

[Documentação do LoginService 16](#_Toc77831298)

[Autenticação por JWT 16](#_Toc77831299)

[Autorização 17](#_Toc77831300)

[Padrões assíncronos 17](#_Toc77831301)

[Rotas 19](#_Toc77831302)

[.NET 19](#_Toc77831303)

[Templates x attribute routing 19](#_Toc77831304)

[Combinações de templates REST vs. templates com actions vs. attribute routing 19](#_Toc77831305)

[Usar ou não um literal no template? 20](#_Toc77831306)

[Attribute Routing 20](#_Toc77831307)

[Padrões de tipo no retorno dos métodos 21](#_Toc77831308)

[.NET 21](#_Toc77831309)

[IHttpActionResult x HttpResponseMessage x classes de domínio 21](#_Toc77831310)

[Padrões de parâmetros 23](#_Toc77831311)

[.NET 23](#_Toc77831312)

[Tratamento de tipos específicos 24](#_Toc77831313)

[Datas, datas com horas, timezones 24](#_Toc77831314)

[Tratamento de erros/validações 25](#_Toc77831315)

[Versionamento de APIs 25](#_Toc77831316)

[Regras para criação ou não de novas versões 26](#_Toc77831317)

[Diretrizes para manter versões compatíveis do código 27](#_Toc77831318)

[.NET 27](#_Toc77831319)

[Controller 27](#_Toc77831320)

[Camada de negócio e dados 29](#_Toc77831321)

[DTO’s, Projeções e Entidades 29](#_Toc77831322)

[Mapeadores de DTO 30](#_Toc77831323)

[Como definir o fim do suporte à uma determinada versão da API na evolução do produto 30](#_Toc77831324)

[Documentação 31](#_Toc77831325)

[.NET 31](#_Toc77831326)

[PHP 32](#_Toc77831327)

[Configuração geral 34](#_Toc77831328)

[Filtros/restrições globais 34](#_Toc77831329)

[Obrigatoriedade para TLS/HTTPS 34](#_Toc77831330)

[Nível da API 34](#_Toc77831331)

[Referências 35](#_Toc77831332)

[Discussões/Pendências 36](#_Toc77831333)

[Pensar em orientar que uma atualização completa de CRUD seja precedida de um GET e seja validada a concorrência otimista 37](#_Toc77831334)

# 

# Versão do documento

|  |  |  |  |
| --- | --- | --- | --- |
| Versão | Data | Autor | Alterações |
| 1 | ? | Márcio Fonseca | Versão inicial |
| 2 | 11/6/2021 | Gustavo Regal | Removidas menções a OutSystems.  Removidas várias seções incompletas.  Alteradas definições sobre HTTP Status Codes.  Alterada definição sobre corpo padrão de resposta. |
| 3 | 22/7/2021 | Gustavo Regal | Adicionada seção sobre autenticação e autorização. |

# Objetivos

Este documento, bem como as iniciativas de padronização, tem como objetivos:

* Seguir melhores práticas na criação de APIs escaláveis, limpas e de fácil manutenção;
* Evitar padrões que, mesmo "funcionais", levam a problemas futuros de evolução dos produtos, causem problemas de entendimento, dificultem o uso e, principalmente, que possam causar impactos em atualizações.

# Observações sobre especificidades por tecnologia

Para facilitar a leitura e o acesso às informações contidas neste documento, cada tópico que possuir alguma especialização conforme a tecnologia será dividido em novos tópicos. Desta forma a documentação permite uma leitura semelhante para todos os times e, quando houver alguma diferenciação, basta seguir a leitura apenas do subtópico da tecnologia que estiver sendo utilizada pelo desenvolvedor. Um exemplo pode ser visto abaixo:

Nomenclatura

* Descrição geral das regras sobre nomenclatura.
* .NET
* Descrição específica para implementações em .NET.
* PHP
* Descrição específica para o caso do PHP.

Nesse sentido, o fluxo de leitura para um desenvolvedor do .NET seria de ler o tópico principal e, após ele, o tópico específico de .NET, enquanto que um desenvolvedor do GVcollege ou outro time que utilize PHP puro leria o tópico principal e o de PHP. Por fim, alguns casos podem possuir uma especificação compartilhada, como no exemplo abaixo:

Documentação

* Sem regras gerais.
* .NET e PHP
* Regras de documentação compartilhadas entre as duas tecnologias.

# Verbos por ação

É necessário termos alguns padrões para uso das rotas e padronização das requests para API. A abordagem mais completa e menos sensível é uma combinação das três possibilidades de uso de rotas na web api:

* Usar padrão REST para o CRUD sempre, ou seja, não incluir nomes de actions nas requests, reservar as actions para exceções e não para o padrão;
* Demais padrões complexos como associações da entidadeOuDto/Detalhes usar attribute routing.

## GET

Usado para todos os cenários comuns de busca por recursos.

## POST

Reservar preferencialmente para o Incluir do CRUD.

Recomenda-se usar o padrão correto no retorno da inclusão, o HTTP Code 201 (created) e não 200, bem como incluir um cabeçalho HTTP Location, como na especificação do protocolo HTTP. As controllers de projeto [ASP.NET](http://asp.net/) já tem helpers para esse caso, como nos exemplos dados na seção de padrões de retorno. Alguns exemplos de helpers são o Created e o CreatedAtRoute.

## PUT

Usado para atualizações de recursos/entidades.

O identificador do recurso/entidade deve ser passado sempre na URL como parâmetro.

O retorno da request não deve conter o recurso/entidade no body depois de processada no servidor.

Definimos não seguir essa parte da especificação e retornar **404 - Not Found:**

Algumas considerações da [RFC7231](https://tools.ietf.org/html/rfc7231#section-4.3.4)

*A especificação trata o PUT com sentido de CreateOrUpdate, ou seja se o recurso especificado não existir ele deveria ser criado e o retorno deveria ser o mesmo do Create, 201 - Created com Location header incluindo a url para acessar o recurso, porém como o identificador na grande maioria dos casos é dinâmico na criação, seria estranho criar o recurso com outro id. Neste caso o cliente da api pode cometer erros de associação do dado pelo id que ele forneceu e não pelo real do recurso.*

## PATCH

Usado para atualizações **PARCIAIS** de recursos/entidades.

Cenários mais comuns aplicado a ações de processos como:

* Aprovações;
* Cancelamentos;
* Mudanças de estado quaisquer que não impliquem o processamento total do recurso/entidade.
* Inativação em entidades com conceito de Ativo/Inativo;

Considerar ambiguidade com entidades com exclusão lógica combinada.

Se possível padronizar o nome da Action para “AlternarAtivacao” ou outro nome que contemple as duas ações conforme o parâmetro de ativo.

## PUT ou PATCH?

Ambos têm basicamente o mesmo significado semântico, sendo o PATCH indicado para atualizações parciais, então para evitar vários padrões daremos a seguinte recomendação de padrão:

Deixaremos o PUT para os casos de Edição de recursos e PATCH para as alterações parciais como rotinas de troca de estado ou evolução via processo e não via CRUD simples.

## DELETE

Usado sobretudo no cenário de exclusão nos CRUDs.

[RFC 7231 - 4.3.5. DELETE](https://tools.ietf.org/html/rfc7231#section-4.3.5)

A especificação acima prevê 3 possibilidades de retorno para uma ação de delete:

**204 - No Content**: Quando a ação não tiver conteúdo na response.

Mais adequado na maioria dos cenários.

200 - Ok: Se a response contiver algum conteúdo.

202 - Accepted: Quando a requisição foi aceita e será processada, seja por ser assíncrona ou por ser processada/analisada em fila ou mesmo analisada por alçada para aprovação.

## HEAD e OPTIONS

Neste momento não serão usados, apesar de termos cenários onde hajam benefícios.

# 

# Respostas HTTP (*HTTP* *responses*)

Este tópico cobre alguns cenários padronizados para que tenhamos como diretriz as RFCs do [Internet Engineering Task Force](https://ietf.org), na sua maioria quanto a casos de sucesso, códigos HTTP, etc...

## Http Status Codes

Este tópico é bem longo e sempre seguiu uma especificação formal dada ainda pela antiga RFC2616. Sua evolução ficou mais estável nas especificações [RFC7230](https://tools.ietf.org/html/rfc7230), [RFC7231](https://tools.ietf.org/html/rfc7231),[RFC7232](https://tools.ietf.org/html/rfc7232) e depois houve segmentações específicas do protocolo http/1.1 para [cache/RFC7234](https://tools.ietf.org/html/rfc7234), [segurança/RFC7235](https://tools.ietf.org/html/rfc7235), etc..

### Respostas de sucesso (2XX)

São os status da classe 2XX, que sempre indicam que a requisição teve sucesso.

* GET: o recurso foi buscado e transmitido no corpo da mensagem.
* PUT ou POST: o resultado da ação é transmitido no corpo da mensagem.
* DELETE: a exclusão teve sucesso.

#### 200 – Success

A requisição foi processada com sucesso **e o retorno (body) contém dados**.

Pode ser usado em GET, PUT, POST, DELETE, ...

#### 201 - Created

A requisição foi processada com sucesso **e uma entidade foi criada**.

Só faz sentido para POST.

Incluir no header da response um link para o item recém criado, como manda a especificação. O link deve ser passado no header *Location*. Verifique na seção de [Helpers](#_belstsiue5ps) se a tecnologia/framework utilizado já não tem um helper facilitando isso.

#### 202 – Accepted

A requisição foi **recebida** com sucesso **e será processada posteriormente.**

Usado em comunicação assíncrona. O cliente não espera o processamento. Apenas é notificado de que ele será iniciado.

Faz sentido para POST, PUT, DELETE.

#### 204 – No Content

A requisição foi processada com sucesso **e o retorno (body) não contém dados**.

Pode ser usado em POST, PUT, DELETE, ...

Para GET, não faz sentido. Se não há dados a retornar, considerar 404 ou 410.

### Mensagens de redirecionamento (3XX)

#### 302 – Found

Significava originalmente que o recurso solicitado foi temporariamente movido para outro endereço. Na prática, porém, é utilizado de uma forma geral para forçar o redirecionamento do cliente para outra URL.

O header da resposta deve conter o campo *Location* indicando a URL a ser consultada.

Navegadores obedecem automaticamente ao 302, sem envolvimento do código Javascript.

### Respostas de erro do cliente (4XX)

#### 400 - Bad Request

A requisição falhou porque **o cliente não obedeceu à sintaxe esperada**.

Exemplo: parâmetro na URL com grafia incorreta ou JSON no corpo da requisição com estrutura incorreta/inesperada.

Reparar na diferença para o 422.

#### 401 – Unauthorized

A requisição **não está autenticada**.

Exemplos: usuário não fez login, portanto não tem cookie de sessão; token ausente; token presente porém inválido.

Reparar na diferença para o 403.

#### 403 – Forbidden

A requisição está autenticada porém **a operação não é permitida**.

Exemplo: usuário tem perfil que não lhe permite aquela transação.

Reparar na diferença para o 401.

#### 404 – Not found

O recurso requisitado **não foi encontrado**.

Pode ser retornado quando foi consultada uma rota que não existe. Ou uma rota válida, porém não existe a entidade com o ID especificado, por exemplo.

Reparar na diferença para o 410.

#### 409 – Conflict

Sintaxe e semântica estão OK, porém **a requisição encontrou algum conflito**. Típico de quando alguma **regra de negócio** não foi obedecida.

É uma opção para os casos de tentativa de inclusão de registro duplicado, por exemplo, ou registro que viola alguma consistência.

Reparar na diferença para o 422 (são facilmente confundidos).

#### 410 – Gone

O recurso requisitado **não foi encontrado, embora previamente estivesse presente**.

Parecido com o 404, porém deixa claro que a entidade um dia existiu. Foi excluída, não está mais disponível.

#### 422 - Unprocessable Entity

A requisição falhou porque **o cliente não obedeceu à semântica esperada**.

Indica que o servidor entende o tipo de conteúdo da entidade da requisição, e a sintaxe da requisição está correta, mas não foi possível processar as instruções presentes.

Reparar na diferença para o 400 e para o 409.

### Respostas de erro do servidor (5XX)

#### 500 – Internal server error

O servidor encontrou um **erro inesperado**, um erro com o qual não sabe lidar.

Típico de exceção tratada mas sem saída de contorno, sem *fallback*. O IIS e outros servidores web também costumam devolver erro 500 quando a aplicação não tratou alguma exceção.

#### 510 – Not implemented

A operação ainda não foi implementada.

É útil quando o serviço ainda está em desenvolvimento. Criam-se todas as rotas inicialmente sem implementação. Depois, implementa-se uma a uma.

## Corpo da resposta

Quando possível, o padrão é suportar retornos em formato preferencialmente JSON, ou então XML, alterando conforme o header de Accept recebido. O encoding a ser utilizado deve ser o UTF-8, visto que algumas tecnologias não possuem suporte direto ao ISO 8859-1.

Não existe estrutura (schemas) de resposta padrão.

Sugere apenas o seguinte:

* Procure manter o corpo da resposta o mais enxuto possível. Quanto menor o payload, melhor desempenho da comunicação via rede. Também se observa o princípio de “ocultação de informação”, ou seja, não exponha mais dados do que o necessário.
* Evite incluir no corpo campos como “status”, ou seja, flags que indiquem o sucesso sim/não, ou qualquer campo que esteja indicando um dado que já esteja implícito através do status HTTP retornado. Exemplo: se o status é 200, então é sucesso. Se o status é 400, então é falha.
* Procure seguir regras de nomenclatura dos campos conforme [Wiki - Nomenclatura de entidades e campos](https://dev.azure.com/gvdasa/NAP/_wiki/wikis/NAP.wiki/3806/Nomenclatura-de-entidades-e-campos).

# Nomenclatura

Procure manter um padrão consistente para as URL´s dos recursos disponibilizados na api.

Parâmetros de métodos:

Usar sempre **camelCase**;

Demais nomes de objetos/entidades, etc:

Usar sempre **PascalCase**;

É um bom padrão usar o plural para acesso a entidades da API bem como o acesso a coleções dentro destas entidades expostas.

Padrão em nomes compostos (/ProcessosAdministrativos).

Relacionamentos

Evitar mais de um nível de detalhe para evitar problemas de manutenção e rotas

Ex: /Estabelecimentos/1/Funcionarios/5

# Autenticação vs. Autorização

São 2 conceitos frequentemente confundidos.

* Autenticação: identificar quem está fazendo a requisição (qual usuário ou *client*);
* Autorização: dado que o requisitante está autenticado, verificar se ele tem permissão para executar aquela operação.

É importantíssimo que as API’s REST implementem ambas as preocupações.

Para ambas, indica-se como primeira opção basear-se no JWT emitido pelo LoginService.

## Documentação do LoginService

[LoginService - Wiki](https://dev.azure.com/gvdasa/LoginService/_wiki/wikis/LoginService.wiki) nova

[Sistema de Autenticação - Wiki Azure DevOps](https://gvdasa.visualstudio.com/GVcentris/_wiki/wikis/GVcentris.wiki/63/Sistema-de-Autentica%C3%A7%C3%A3o) (página antiga mantida pelo GVcentris)

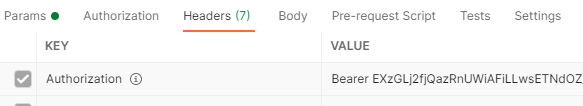
## Autenticação por JWT

O JWT pode ser recebido via *header* ou via *cookie*.

Para API’s que atendam exclusivamente a uma aplicação web (como um *frontend SPA*), recomenda-se trafegar o token como um *cookie* com as *flags* *Secure* + *HttpOnly*, pois isto aumenta os níveis de segurança.

* *Secure*: o navegador só via enviar este cookie em conexões HTTPS;
* *HttpOnly*: o navegador não vai expor esse cookie a códigos Javascript.

Para API’s que estarão atendendo a diferentes clients, é melhor optar por trafegar o token no *header Authorization Bearer*, como no exemplo a seguir no Postman:



## Autorização

O JWT padrão LoginService, a partir da versão 4, conterá *claims* que facilitarão a autenticação e também a autorização.

A primeira *claim*, chamada ***sub*** (em versões anteriores esta *claim* se chamava ***uid***), indica o id do usuário no LoginService. Qualquer aplicação de posse do token pode, portanto, identificá-lo. Necessitando de dados como nome, e-mail, ..., do usuário, a aplicação pode fazer uma requisição à API do LoginService ou a alguma outra API que contenha cadastro de pessoas (como as do GVcollege e GVcentris).

Em versão ainda a definir, o LoginService emitirá tokens com uma *claim* objeto, chamada ***prf***, cuja intenção é expor quais são os perfis atribuídos àquele usuário por estabelecimento.

Exemplo:



# Padrões assíncronos

Sempre que possível se sua API e seu backend suportam padrões assíncronos, dê preferência para eles. Basicamente temos somente vantagens:

* Diminuição da contenção de recursos em operações mais demoradas;
* A diminuição da contenção contribui diretamente para a escalabilidade da aplicação quando atendida por um pool de threads de tamanho fixo (padrão .NET), pois ao encontrar uma diretiva de execução assíncrona o .NET libera o thread para processar outra requisição web no servidor enquanto aguarda o retorno da operação assíncrona.

O retorno das requests deve ser **202, Accepted.**

O conteúdo da response deve indicar onde/como o usuário poderá ver o resultado final quando processado pelo servidor, por exemplo, processos que terminam no serviço de tarefas assíncronas que podem ser acompanhados com a devida permissão.

Observação: Cuide para não confundir com padrões de processamento paralelo, tipicamente não suportado por frameworks de ORM como Entity Framework.

# Rotas

## .NET

### Templates x attribute routing

Normalmente o uso de attribute routing é mais recomendado para exceções do que padrões. Sua principal vantagem é construir rotas complexas como no caso de associações/relacionamentos de entidades não representados com templates globais. A principal desvantagem é que se espalham as chamadas *"magic strings"* que sempre acabam gerando problemas nas refatorações ou ao se fazer mudanças globais acabam requerendo ajustes em todas as controllers.

### Combinações de templates REST vs. templates com actions vs. attribute routing

Essas combinações globais ou somadas a attribute routing causam alguns problemas no roteamento, necessitando que o desenvolvedor siga alguns passos para não quebrar suas rotas ao incluir novas actions na controller:

* Aqui se obtém a maior flexibilidade e padronização, se respeitadas algumas regras e também é onde temos mais problemas com rotas.

Como contornar essa situação:

Se sua controller tiver exceções ao padrão REST definido aqui e você precisar mais actions com o mesmo verbo HTTP siga os seguintes passos:

* Inclua um atributo [RoutePrefix] na controller incluindo o literal do template de rota se tiver definido na configuração da API (geralmente o literal é api/);
* Inclua um atributo **[Route("")]** em cada action do REST que não tiver parâmetro;
* Inclua um atributo **[Route("{param}")]** em cada action do REST com parâmetros, se tiver ambiguidade com outra rota da controller. Ex: Obter/ObterTodos, ambos com HttpGet;
* Inclua em cada action que for exceção ao padrão um atributo [Route("TokenDoFinalDaRota")] o nome que você quer para o final da rota. Ex: Cancelar, Estornar, Ativar, etc..

Quais as vantagens de manter um padrão REST para o CRUD com verbos HTTP x route ao invés de simplesmente usar os nomes das actions?

* Se você resolver alterar toda a sua API para assíncrona, por exemplo, não vai precisar ajustar as rotas, nem deixar a nomenclatura inadequada depois da mudança;
* Se você estabelecer novos padrões de consumo da API, como em um front-end específico, você tem menos impacto de refatoração nesse front-end se seguir estes passos.

#### Usar ou não um literal no template?

* É uma boa prática para evitar impacto com necessidades futuras, como a inclusão de um projeto MVC;
* O template de projetos de Web API do Visual Studio também usa um literal como api/\* para todas as rotas da API e assim podemos ter acesso a rotas no MVC sem colisão;
* A única desvantagem pequena é no uso de attribute routing usar um RoutePrefix tendo que repetir o mesmo literal do template geral em cada controller.

### Attribute Routing

Para mais informações veja [Attribute Routing in ASP.NET Web API 2](https://docs.microsoft.com/en-us/aspnet/web-api/overview/web-api-routing-and-actions/attribute-routing-in-web-api-2)

# Padrões de tipo no retorno dos métodos

Seção que visa a padronização dos possíveis retornos das actions conforme cada tecnologia.

## .NET

### IHttpActionResult x HttpResponseMessage x classes de domínio

A versão atual da Web API suporta várias formas de retorno como as citadas acima. Cada uma tem vantagens e desvantagens, mas no geral a indicada para uso atual pela Microsoft é:

IHttpActionResult e a variação assíncrona Task<IHttpActionResult>.

Vantagens:

* Seguir uma orientação da Microsoft e não usar as formas mais antigas como HttpResponseMessage;
* Você não precisa construir a response do método e pode usar helpers da classe base da controller da API:

return Ok();

return Ok<T>();

return BadRequest();

return BadRequest(modelState);

return BadRequest(mensagem);

return NotFound();

return Conflict();

return StatusCode(System.Net.HttpStatusCode.NoContent);

return CreatedAtRoute("DefaultApi", new { id = Entidade.Id }, EntidadeOuDto);

return Created("Location", T);

Para mais informações sobre retorno de conteúdo em Web Api veja: [Action Results in Web API 2](https://docs.microsoft.com/en-us/aspnet/web-api/overview/getting-started-with-aspnet-web-api/action-results)

# Padrões de parâmetros

Tipos simples devem ser passados na URL e tipos complexos devem ser passados no body da requisição.

## .NET

Para receber parâmetros complexos deve ser utilizado o atributo *FromBody*.

# 

# Tratamento de tipos específicos

## Datas, datas com horas, timezones

De maneira geral deve-se procurar sempre trabalhar com UTC, tanto para datas e horas recebidas quanto para as retornadas pela API.

# Tratamento de erros/validações

Sempre conduzir, no cliente, o tratamento dos retornos em primeiro lugar conforme o status HTTP recebido.

Dependendo do status, seguir um fluxo de tratamento diferente.

Exemplo: para uma determinada operação, o corpo de respostas 200 pode ter um *schema*, já o corpo da resposta para status 400 pode ter outro *schema*.

# Versionamento de APIs

Entre as diversas opções de versionamento de APIs escolhemos a que deixa mais claro o seu uso, não cria cenários de ambiguidade ou onde a requisição inspecionada ou parcialmente exposta já deixa clara a versão, assim como não impede seu uso mesmo em tecnologias com maiores limitações como frameworks de front-end.

***O padrão escolhido é de incluir a versão da API na própria url, no componente path.***

Exemplos:

[https://MeuDominio/api/versao/Rota](https://meudominio/api/versao/Rota)

[https://outsystemsdev.gvdasa.com.br/Usuarios/api/**v1**/Grupos/{id](https://outsystemsdev.gvdasa.com.br/Usuarios/api/v1/Grupos/%7Bid)}

*Não é necessário usar versionamento x.y com a semântica de* [*Semantic Versioning 2.0*](https://semver.org/)*, pois variações no segundo octeto nunca trazem breaking changes e assim não precisaria usar o segundo octeto.*

Formas de controle descartadas

Versionamento pelo Header da requisição

Os principais inconvenientes no uso da versão em cabeçalho da requisição é a fixação/manutenção do valor, esta parte ficará oculta se apenas informada a url como em casos de suporte, imagens e traces parciais. Dificultando o auxílio ao cliente em casos de suporte.

Versionamento via query string

Os principais inconvenientes no uso da versão na query string da requisição é a fixação/manutenção a cada chamada, onde poderia estar fixa na maior parte dos casos na url base do componente para requisições ou montagem manual das requisições como no front-end.

## Regras para criação ou não de novas versões

Não é necessário criar uma nova versão quando:

Pequenas mudanças que não causam incompatibilidade com clientes da versão atual da api

Exemplos:

* Novo recurso;
* Novo parâmetro opcional em rota existente;
* Alterações de regras internas (não públicas), que não mudem o conteúdo esperado;
* Novos atributos retornados em uma requisição.
  + Esse item precisa ser avaliado, pois se um ou mais atributos começarem a aparecer nos Gets, ao atualizar os dados via api nesta versão os atributos não serão enviados, fazendo o servidor descartar dados já persistidos.

É necessário criar uma nova versão quando:

Mudanças que causam incompatibilidade ao cliente da api

* Novo parâmetro obrigatório em rota existente;
* Remoção de um recurso/endpoint;
* Mudança de comportamento interno que afete o resultado da operação.

## Diretrizes para manter versões compatíveis do código

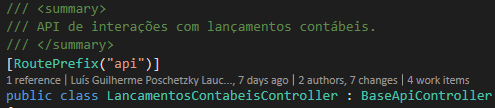
Para o versionamento das APIs e manutenção do código do back-end da aplicação optamos por seguir um cenário de manter um único projeto duplicando os métodos e actions da controller, quando necessário. Em um primeiro momento não seria necessário duplicar os objetos de dados, como DTO’s e Projeções, mas deve-se atentar ao fato de que campos novos, se não tomadas as devidas precauções, podem ser removidos ao utilizar versões antigas da API. A forma de como tratar a duplicação do código terá particularidades conforme a tecnologia, portanto a seguir serão definidos padrões para cada uma.

## .NET

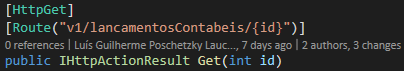
Para o .NET teremos que optar por duplicar apenas métodos nas classes de serviço e consultas, inserindo também actions de múltiplas versões nas controllers. A seguir cada cenário será melhor detalhado com exemplos práticos:

### Controller

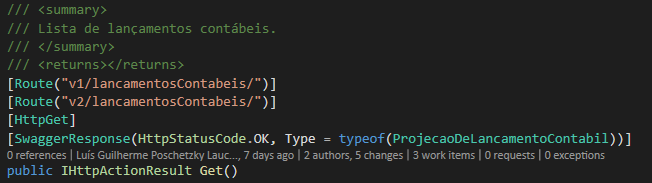
Para as controllers, como já citado anteriormente, é necessário sempre definir um RoutePrefix e utilizar o atributo Route em todas as actions de maneira que se evite conflitos de rota durante a evolução da API. Em um cenário normal, o Route Prefix de uma controller poderia ser algo como *api/v1*. Entretanto, com a possibilidade de termos múltiplas versões, o mais correto seria termos as versões diretamente nas actions, deixando o prefixo apenas como *api*. Abaixo pode ser visto um exemplo da forma correta a ser utilizada:



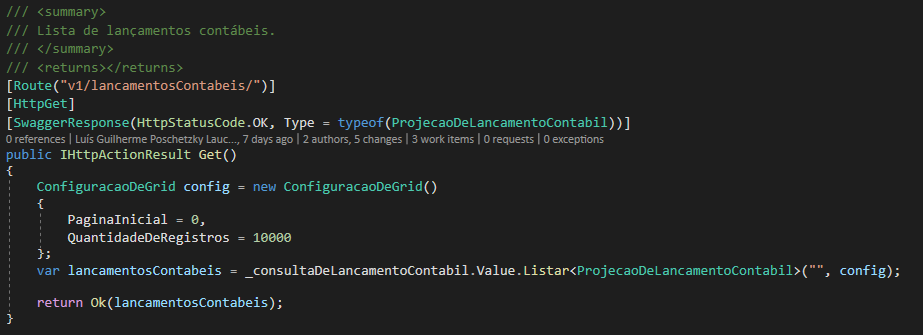
Para as actions das controllers é necessário definir o atributo Route, mesmo para os casos padrão, como pode ser visto abaixo:

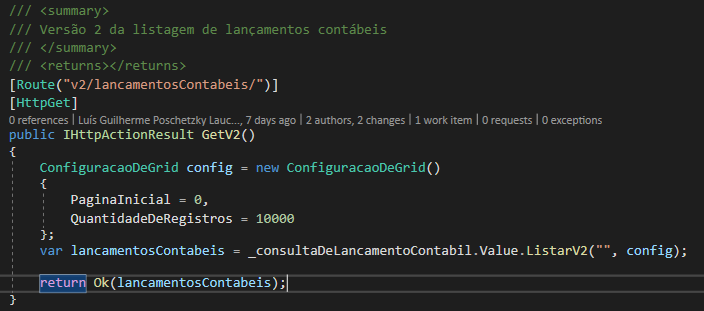


Caso uma action esteja disponibilizada em mais de uma versão da API, basta definir dois atributos de Route, como no exemplo abaixo:



Em um cenário onde seja necessária a duplicação, teríamos dois endpoints diferentes da seguinte forma:





### Camada de negócio e dados

Nas camadas de negócio e dados deve-se duplicar os métodos que possuam retornos diferentes, tomando o cuidado de identificar a versão com a qual eles são compatíveis. Abaixo um exemplo de cenário onde isto foi implementado: 

### DTO’s, Projeções e Entidades

Apesar de não existirem grandes impactos para a aplicação em geral acaba se fazendo necessário a duplicação de Projeções e DTO’s de forma que a documentação de cada versão possa ser corretamente isolada, do contrário propriedades novas acabariam aparecendo também na documentação de versões antigas, o que não é desejado. Com relação às entidades, não se deve duplicar nada, ela deve ser híbrida em relação à todas as versões suportadas.

### Mapeadores de DTO

Como no caso de múltiplas versões existirá mais de uma DTO é necessário, também, criar um mapeador específico pra cada variação da DTO. Os métodos de serviço e consultas duplicados devem chamar a versão correspondente do seu mapeador. Isso resolverá as questões de mapeamento de propriedades que não existem em versões antigas, as quais deverão ignorar as propriedades novas.

~~Em qualquer opção que formos seguir de como compartilhar o código do back-end da aplicação, seja evitando duplicação, seja duplicando tudo ou até parcialmente duplicando, vamos ter que focar bastante em testes. Imagino que nenhum método de nenhuma versão da API poderia ser publicado sem um teste validando ele, para que, ao evoluirmos algo, possamos garantir que nada irá quebrar. Uma opção seria termos uma aplicação bem genérica, que permite inserir uma url em um campo na tela, fazer a chamada, construir os campos dinamicamente na tela e validar os resultados via automação funcional, criando os campos dinamicamente na tela conforme a necessidade. Basicamente teríamos que criar um Postman simplificado para testar todas as API’s.~~

#### Como definir o fim do suporte à uma determinada versão da API na evolução do produto

# 

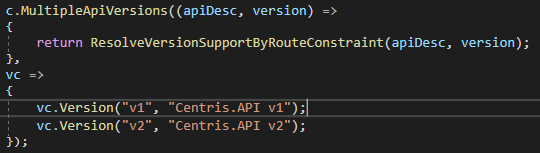
# Documentação

A documentação de APIs deve sempre ser disponibilizada, facilitando para o usuário final o consumo. Cada tecnologia possui suas especificidades para documentação, que serão definidas abaixo.

## .NET

Para estas tecnologias uma das opções mais conhecidas para auto-documentação ou mesmo documentação guiada por atributos qualificadores é o Swagger UI. Existe para .Net um pacote que simplifica o uso deste documentador, é o [Swashbuckle](https://github.com/domaindrivendev/Swashbuckle), que após incluído em um projeto de Web api permite, com configuração quase zero, a auto-documentação, gerando, inclusive, um endpoint de exposição. O código gerado a partir da versão 5.0 já é compatível com Swagger 2.0/OpenAPi 2.0, apesar de, até o presente momento, não ter suporte para a versão 3.0, visto que o pacote foi descontinuado para .NET Framework e o suporte mantido apenas para o .NET Core. Além dos benefícios da documentação ele permite integrações/uso com outras ferramentas [Swagger/OpenAPI](https://swagger.io/) para geração de clientes para api e outros benefícios. A seguir os procedimentos para a configuração do Swashbuckle em um projeto em .NET Framework:

* Ao ser instalado o pacote [SwashBuckle](https://github.com/domaindrivendev/Swashbuckle), ele cria um registro automático com configurações padrão sob a pasta App\_Start da web API;
* As configurações do arquivo SwaggerConfig.cs devem ser ajustadas;
* Caso haja suporte a múltiplas versões é necessário realizar a configuração do atributo MultipleApiVersions, como pode ser visto abaixo:



A resolução de como o Swagger irá tratar a diferenciação de versões deve ser definida em um método, como no caso do ResolveVersionSupportByRouteConstraints, que leva em consideração a versão incluída na rota, nosso padrão. Também é necessário estar atento às diferentes possibilidades de atributos que auxiliam na documentação, alguns atributos úteis podem ser visto abaixo:

* SwaggerResponse - Permite identificar os possíveis retornos das actions, com os diferentes códigos de status e o tipo de retorno, permitindo a padronização de uso do *IHttpActionResult* e a utilização dos helpers padrão:



No caso da OutSystems não existem configurações avançadas, apenas ligar ou desligar a documentação. O JSON com a representação OpenApi da api exposta em OutSystems pode ser baixado/visualizado na raiz da api incluindo /Swagger.json que pode ser usado para geração de clientes para consumo da api.

## PHP

No caso do PHP a tecnologia mais indicada é o swagger, o mesmo possui a biblioteca **darkaonline/l5-swagger** que integra nativamente com Laravel e PHP 5.4 ou superior.

A documentação da Api é realizada via annotations que podem ser colocadas nas Controllers, Models e demais entidades php pertinentes dentro da pasta App (o swagger realiza a varredura completa de todas entidades ao ser gerado a documetação). Por padrão o ideal é que essas annotations sejam realizadas nas Controllers idealmente.

A documentação é gerada sob demanda via linha de comando, acessando o diretório raiz do projeto Laravel e executando o comando **php artisan l5-swagger:generate**.

Durante a primeira execução da geração da documentação, é necessário realizar a publicação da mesma através do comando **php artisan l5-swagger:publish**.

## Configuração geral

## Filtros/restrições globais

Dê preferência a exceções sobre padrões quando a segurança for impactada.

Exemplo: Use um atributo de acesso anônimo em um método e use um filtro global que imponha acesso restrito por padrão e nunca o contrário. Assim uma action nova sem marcação(atributo) não termina em acesso anônimo por omissão.

## Obrigatoriedade para TLS/HTTPS

# Nível da API

Em 2008 Leonard Richardson propôs a seguinte classificação de nível para apis:

Level 0: Define somente uma URI, e todas as operações usam verbo POST nas requisições para a URI.

Level 1: Criar URIs separadas para os recursos.

Level 2: Usar verbos HTTP para definir as operações sobre os recursos.

Level 3: Usar hypermedia (HATEOAS, incluir um link).

A recomendação inicial é mantermos nossas apis no nível 2, pois o uso de hypermedia ainda requer algumas discussões para que seu uso não recaia em não padronização na parte menos explícita da especificação.

## 

# Referências

[API design - Microsoft Docs](https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design#versioning-a-restful-web-api)

[RFC2616 - Hypertext Transfer Protocol -- HTTP/1.1](https://tools.ietf.org/html/rfc2616)

[RFC5246 - The Transport Layer Security (TLS) Protocol Version 1.2](https://tools.ietf.org/html/rfc5246)

[RFC5789 - PATCH Method for Http](https://tools.ietf.org/html/rfc5789)

[RFC7230 - Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing](https://tools.ietf.org/html/rfc7230)

[RFC7231 - Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content](https://tools.ietf.org/html/rfc7231)

[RFC7232 - Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests](https://tools.ietf.org/html/rfc7232)

[RFC7233 - Hypertext Transfer Protocol (HTTP/1.1): Range Requests](https://tools.ietf.org/html/rfc7233)

[RFC7234 - Hypertext Transfer Protocol (HTTP/1.1): Caching](https://tools.ietf.org/html/rfc7234)

[RFC7235 - Hypertext Transfer Protocol (HTTP/1.1): Authentication](https://tools.ietf.org/html/rfc7235)

[RFC7540 - Hypertext Transfer Protocol Version 2 (HTTP/2)](https://tools.ietf.org/html/rfc7540)

[OpenApi v3.0](https://github.com/OAI/OpenAPI-Specification) no GitHub ou exposto no [site do Swagger](https://swagger.io/specification/)

[Semantic Versioning 2.0](https://semver.org/)

[Ferramentas compatíveis com OpenApi](https://openapi.tools/)

# Discussões/Pendências

* **Criptografia dos IDs (Gerenciamento de chaves)**
  + Usar criptografia de chave assimétrica(**RSA**);
  + Definir o tamanho da chave em bits, o menor possível(+-512/1024);
  + Cada sistema deve disponibilizar um endpoint para o download da chave pública para criptografar os valores a serem passados para a api;
    - Já temos alguma implementação em OS;
  + O melhor seria a construção de uma dll em .NET com exposição ao COM para reuso em várias tecnologias;
  + Para uso dentro da OS entre duas apis suas precisa outro tipo de implementação sem a dll e api, pois tudo estará acessível dentro do próprio ambiente.(Área 87)
* Documentação
  + ~~Documentar padrão de passagem de parâmetros primitivo x complexos (Demanda do TECH)~~
  + ~~Documentar padrão versionamento no documento padrão (Demanda do TECH)~~
  + ~~Ver tratamento de datas e horas nestas apis: Ex:~~ **~~ISO~~**~~, Unix..~~
  + ~~Documentar necessidade de comentário no código para geração da documentação. (Demanda do TECH)~~
    - Detalhar tipos de retorno com anotações para casos comuns como 404, 40?,..
  + ~~Documentar padrão de documentação do PHP/College (Vinicius Lima)~~
  + ~~Avaliar questão de versionamento x documentação (Demanda do TECH)~~
    - ~~Verificar o isolamento das versões e a exibição correta do conteúdo de cada versão.~~
  + Avaliar questão de versionamento x documentação (Área 87)
    - Verificar o isolamento das versões e a exibição correta do conteúdo de cada versão.
* Autenticação na documentação.
* Questão dos testes, cada time precisa definir. Boa prática obrigatória.
  + ~~Criação de teste API consulta de saldo e apresentar ao grupo de padronização (Demanda TECH)~~
* Ideia de elaborar termos de aceite de uso de API
* Iniciar documentação dos termos de aceite das APIs (Vinícius, Jonas)
* Definir padrão de filtros, paginação e ordenação de listas de dados.
* ~~Avaliar duplicações controller (Demanda do TECH)~~
* Validar padrão API’s captação (Paulo).
* Log sequencial de requisições
  + Log sequencial com identificação de request com contexto de processos de negócio.
  + Log de erros (Política limpeza logs)
* Recomenda-se o uso de endpoints padrão como /health, /version, /status, /metrics e /debug, esta abordagem é interessante para nós também?
* Internacionalização?
* Há alguma chance de querermos controlar/barrar o fluxo de requisições por quantidade, tempo, massa de dados? É uma possibilidade de monetização para a GVdasa?

## Pensar em orientar que uma atualização completa de CRUD seja precedida de um GET e seja validada a concorrência otimista

O caso de concorrência é um ponto importante a ser definido. Já usamos em quase todos os cenários de back-end, mas protegendo somente o lapso de tempo da recuperação do registro da base e a atualização sequencial dele já no back. No caso da api esse lapso de tempo inclui uma request GET anterior ao PUT, ou mesmo POST na criação de um novo registro onde a identificação única tive a criação após o cliente da api ter feito o GET validando a inexistência do registro. Porém isso necessitaria a exposição do atributo de controle da concorrência no retorno do GET para validação posterior e o envio dele no PUT para que o back da api possa sinalizar que outro usuário alterou aquele dado, como é feito no uso de telas.

Pela RFC 7231, mais recente, a resposta adequada para o erro de concorrência seria o Conflict(409).

Lembrar que o versionamento é a única forma de não perder dados em atualizações completas com PUT onde um cliente desatualizado pode ler os dados via api e submetê-los com partes faltantes por erros de parse do json tipados com versões anteriores.

Mais confusão na documentação do IIS >7

<https://support.microsoft.com/en-us/help/943891/the-http-status-code-in-iis-7-0-iis-7-5-and-iis-8-0>

Ele decompõe vários status complementares que não existem nas especificações oficiais, as RFCs.

Como várias das validações/retornos não são feitos pelo dev e sim pela infra como o IIS o consumidor de uma api precisa de um mecanismo padrão de leitura do status code com um de > para para situações de sucesso/Erro.

Isso pode levar a erro se o cliente da api validar pelo texto que representa o status e não pelo status. Vi que no caso do IIS tem várias distorções dos status codes padronizados.