**A IMPORTÂNCIA DE USAR PADRÕES DE DESIGN EM APIS REST**

Caio Cesar Alves Borges

Pós-graduando em Desenvolvimento de

Aplicações Web e Móveis Escaláveis – Uni-FACEF

caiocesarborges89@gmail.com

Kenneth Gottschalk de Azevedo

Pós-graduando em Desenvolvimento de

Aplicações Web e Móveis Escaláveis – Uni-FACEF

[kg\_azevedo@hotmail.com](mailto:kg_azevedo@hotmail.com)

**Resumo**

Atualmente, a utilização da Interface de Programação de Aplicações (API) vem se tornando cada vez mais comum no contexto corporativo, o que não é de se estranhar, já que o seu uso proporciona diversos benefícios envolvendo aspectos como praticidade, performance e segurança. Dentre os benefícios, existe a possibilidade de integrar aplicações distintas, no entanto, para que isso aconteça, é imprescindível que a aplicação requisitante respeite os padrões de documentação impostos pela aplicação fornecedora antes de usufruir dos seus recursos, e é neste momento que se deve enfatizar a importância do uso de padrões de design em APIs. O objetivo deste trabalho é, com base em pesquisas, apresentar os possíveis problemas causados pela falta de padrão e sugerir boas práticas de desenvolvimento usando como referência uma API feita pelos próprios autores em virtude dos conhecimentos adquiridos durante o curso, sempre enfatizando de que nada descrito tem a obrigação de ser seguido.

**Palavras-chave**: Padrões de Design. API REST. Integração de Aplicações.

**Submissão**: 06/03/2021. **Aprovação**:

**1 Introdução**

A partir do momento que o desenvolvimento de APIs ganhou destaque na área da tecnologia, várias empresas começaram a utilizá-las em seu cotidiano, porém, muitas destas empresas não estabeleciam padrões de design em suas APIs, ocasionando diversos problemas durante a integração de aplicações. Esse tipo de situação também ocorre com frequência nos dias atuais.

Essas informações foram obtidas perante a análise das diversas APIs públicas expostas na internet e, tendo em vista este cenário, foi feita uma pesquisa sobre quais deveriam ser os pontos mais relevantes a serem levados em consideração pelas APIs para que elas ficassem legíveis, proporcionando assim, uma integração de forma simplificada.

No decorrer das pesquisas, foram encontrados diversos tipos de padrões de desenvolvimento válidos e, baseando-se neles, concluiu-se que é fundamental que uma API utilize os verbos com a intenção de representar uma ação, contenha URLs que estejam relacionadas às entidades, disponibilize a opção de paginar e ordenar registros, retorne os códigos de *status* corretamente com base em cada tipo de situação, siga um padrão de nomenclatura e forneça uma documentação fácil de ser interpretada.

A API desenvolvida pelos autores engloba todos os aspectos abordados anteriormente, possuindo operações de busca, inclusão, alteração e exclusão de registros. Os exemplos citados neste trabalho estão baseados nesta API e o link da sua documentação encontra-se nas referências.

Levando em consideração a complexidade do tema abordado, os autores visaram explicar e exemplificar cada um dos assuntos abordados na pesquisa com o propósito de facilitar o entendimento de cada um deles, buscando assim, agregar todos os tipos de leitores.

**2 API REST**

Não se pode falar sobre padrões de design em APIs REST sem antes entender o significado de API e como ela é caracterizada como REST.

2.1 Definição

A sigla API é a abreviatura de *Application Programming Interface* e se trata de uma interface que possibilita a comunicação entre duas ou mais aplicações distintas. Se uma API requer que os dados sejam enviados através da rede, ela é caracterizada como um Web Service. Desse modo, todo Web Service é considerado uma API, porém, nem toda API é considerada um Web Service.

*Os Web Services surgiram como consequência natural da utilização da Internet. Alguns consideram essa utilização massificada, como um processo que produz a evolução desse meio de comunicação entre pessoas, e também como grande rede de computadores, o que naturalmente levou à possibilidade de se escrever aplicações e disponibilizá-las ao público em grande escala. (ABINADER; LINS. 2006, p. 10)*

Já a sigla REST, que é a abreviatura de *Representational State Transfer*, se trata de um conjunto de padrões de arquitetura que devem ser seguidos durante o desenvolvimento de Web Services. RESTful é um termo atribuído à API que segue os padrões REST.

As APIs têm como principal função expor recursos na Web, sendo que cada recurso possui uma maneira diferente de ser adquirido. Para isso, o *client* (aplicação requisitante de recursos) e o *server* (aplicação fornecedora de recursos) se comunicam através de mensagens HTTP compostas por verbo, URL, cabeçalho(s), corpo e código de *status*.

2.2 Verbos

Um dos dados obrigatórios de requisição durante a chamada de API é o verbo (ou método). Ele representa a ação que será realizada e é essencial que seja utilizado da maneira correta. Os principais verbos são:

* GET — obtenção dos dados de um ou mais registros;
* POST — inclusão de um novo registro;
* PUT — alteração dos dados de um registro;
* PATCH — alteração dos dados de um registro (de forma parcial);
* DELETE — exclusão de um ou mais registros.

Há casos em que o verbo POST é utilizado para realizar operações lógicas ao invés de incluir novos registros, como, por exemplo, no retorno de um *Webhook*.

2.3 URL

A URL (*Uniform Resource Locator*) também é um dado de requisição obrigatório e, quando junta ao verbo, são utilizados para identificar unicamente um recurso na Web. A sintaxe de uma URL é *protocolo://domínio:porta/caminho*, sendo:

* Protocolo — HTTP ou HTTPS;
* Domínio — nome do servidor Web ou endereço IP;
* Porta — número da porta utilizada pela aplicação;
* Caminho — nome e localização de determinado recurso.

As URLs também podem possuir variáveis de caminho e/ou parâmetros de requisição, que podem representar um filtro de busca ou identificar um registro.

A Figura 1 mostra exemplos de junção do verbo com a URL (ressaltando que, nas rotas de inclusão e alteração, os dados de requisição do registro ficam no corpo da mensagem):

Figura 1 — Exemplo de rotas para buscar, incluir, alterar ou excluir registros



Fonte: Autores

Existem desenvolvimentos que causam uma quebra no contrato, não sendo possível reaproveitar as rotas já existentes. Esse tipo de problema ocorre em situações como: alteração do verbo, alteração no caminho da URL, adição de dados de requisição obrigatórios, alteração de nome ou tipagem de um dado, remoção de dados de resposta e alteração no código de *status*.

Para resolver esse tipo de problema é necessário adicionar uma nova versão nas rotas que foram desenvolvidas (manter a *v1* e adicionar a *v2*, por exemplo) e adequar os *clients* para utilizá-las.

2.4 Cabeçalhos

Os cabeçalhos são dados opcionais introduzidos nas requisições e nas respostas de uma chamada de API. Geralmente são distintos e não possuem relação com a entidade que a URL se associa.

Figura 2 — Exemplo de adição de um token JWT no cabeçalho das requisições



Fonte: Autores

Uma ótima maneira de se realizar autenticações é introduzindo chaves de API no cabeçalho da requisição, independentemente do tipo da autenticação. No caso da API desenvolvida pelos autores, utilizou-se a autenticação JWT e a chave se encontra exposta na documentação para facilitar os testes feitos pelos usuários (lembrando que não é uma boa prática expor dados sensíveis em uma documentação, sendo este, um caso de exceção).

2.5 Corpo

O corpo é um conteúdo opcional introduzido nas requisições e nas respostas de uma chamada de API. Este conteúdo geralmente possui um formato de texto informado no cabeçalho, sendo que os mais utilizados são XML e JSON.

Na Figura 3, o corpo da mensagem de resposta é separado em *meta* (contendo dados de paginação, nome do servidor e versão da aplicação) e *records* (contendo o conteúdo dos registros). Esse tipo de padrão facilita a integração por parte das aplicações requisitantes, pois pode-se considerar que todas as respostas terão o mesmo formato e que sempre haverá registros nas chamadas com sucesso.

Figura 3 — Exemplo de conteúdo no corpo da mensagem



Fonte: Os Autores

É aconselhável que o corpo das mensagens possua uma relação com a entidade que a rota se associa (em exceção dos casos de exclusão, pois não há conteúdo de resposta).

2.6 Código de *status*

O código de *status* se trata de um número que identifica se a resposta de uma chamada de API foi processada com sucesso ou não. Eles são classificados em famílias:

* Família 100 (de 100 à 199) — informação;
* Família 200 (de 200 à 299) — sucesso;
* Família 300 (de 300 à 399) — redirecionamento;
* Família 400 (de 400 à 499) — erro por parte do *client*;
* Família 500 (de 500 à 599) — erro por parte do *server*.

Figura 4 — Exemplo dos possíveis códigos de *status* na exclusão de um registro



Fonte: Autores

A Figura 5 mostra a enorme variedade de códigos de *status* criados até o momento:

Figura 5 — Lista de todos os códigos de *status* até o momento



Fonte: Http Status Codes Cheat Sheet – Konstantin Stepanov

Implementar todos os códigos de *status* à risca é uma tarefa complexa e que exige muita paciência. Para isso, foi feito um levantamento dos códigos de *status* mais utilizados pelas APIs junto com a especificação de cada um deles.

Lista dos códigos de *status* de sucesso mais utilizados:

* 200 (*OK*) — utilizado em rotas não paginadas do tipo GET, em rotas do tipo PUT/PATCH e em rotas do tipo POST que realizam operações lógicas ao invés de inserir novos registros;
* 201 (*CREATED*) — utilizado em rotas do tipo POST que realizam a inserção de novos registros;
* 202 (*ACCEPTED*) — utilizado em rotas que realizam processos de maneira assíncrona;
* 204 (*NO CONTENT*) — utilizado em rotas que não possuem conteúdo de resposta, como por exemplo, rotas do tipo DELETE;
* 206 (*PARTIAL CONTENT*) — utilizado em rotas que possuem paginação;
* 207 (*MULTI-STATUS*) — utilizado em rotas que processam vários dados ao mesmo tempo, podendo haver dados processados com sucesso e dados processados com erro, tudo em uma mesma chamada.

Lista dos códigos de *status* de erro por parte do *client* mais utilizados:

* 400 (*BAD REQUEST*) — utilizado quando o *client* preenche algum dado de requisição incorretamente;
* 401 (*UNAUTHORIZED*) — utilizado quando os dados de autenticação informados pelo *client* são inexistentes ou inválidos;
* 403 (*FORBIDDEN*) — utilizado quando o *client* não possui permissão para realizar determinada ação;
* 404 (*NOT FOUND*) — utilizado quando determinado caminho/registro não é encontrado;
* 408 (*TIMEOUT*) — utilizado quando ocorre *timeout* durante o processamento de algum dado;
* 409 (*CONFLICT*) — utilizado quando o *client* tenta criar um registro já existente ou processar um dado já processado;
* 410 (*GONE*) — utilizado quando o *client* tenta processar um dado que, por algum motivo, foi expirado;
* 412 (*PRECONDITION FAILED*) — utilizado quando ocorre erro nas validações feitas antes do processamento dos dados;
* 422 (*UNPROCESSABLE ENTITY*) — utilizado quando o *client* informa dados de requisição válidos, porém, não processáveis;
* 429 (*TOO MANY REQUESTS*) — utilizado quando o servidor atinge o limite máximo de processamento de dados.

Lista dos códigos de *status* de erro por parte do *server* mais utilizados:

* 500 (*INTERNAL SERVER ERROR*) — utilizado quando ocorre algum tipo de erro inesperado durante o processamento dos dados;
* 502 (*BAD GATEWAY*) — utilizado quando uma dependência externa apresenta algum tipo de comportamento inesperado.

2.7 Paginação e ordenação

As rotas que retornam mais de um registro precisam ser paginadas pois, desta forma, será trafegado somente o necessário e a API se tornará mais performática, oferecendo uma melhor experiência às aplicações requisitantes.

Uma das formas existentes de paginar registros é informar, via parâmetros de requisição, o número da página atual e a quantidade de registros por página. É desejável que o total de registros também seja retornado pela API para que a aplicação requisitante saiba a quantidade de páginas existentes.

Figura 6 — Exemplo de uma busca de registros com paginação



Fonte: Autores

Em relação à ordenação, é aconselhável que a funcionalidade fique dentro da API para evitar que haja desenvolvimento nas aplicações requisitantes.

Uma forma válida de ordenar registros é informando o nome do campo que será usado pra realizar a ordenação junto ao tipo de ordenação (crescente ou decrescente), por exemplo: *v1/developers?offset=0&limit=3&sort=+name*.

**3 Padrões de nomenclatura**

É essencial que uma API possua um padrão de nomenclatura nas rotas e nos dados de requisição/resposta, pois as aplicações requisitantes consideram isto como uma premissa para realizar integrações. Há diversos tipos de padrões de nomenclatura, porém, é imprescindível que apenas um deles seja utilizado.

Alguns exemplos de padrões utilizados em rotas são: Singular, Plural, *Camel Case*, *Snake Case* e *Spinal Case*.

No caso dos dados de requisição e de resposta, é aconselhável a utilização do plural em dados do tipo lista e o singular para o restante dos dados. O *Camel Case*, *Snake Case* e *Spinal Case* também são exemplos de padrões que podem ser aplicados a estes dados.

Vale salientar que também é importante utilizar apenas uma língua durante o desenvolvimento de uma API (não misturar português e inglês, por exemplo).

**4 Documentação**

Após a finalização do desenvolvimento de uma API, é necessário a exposição de suas funcionalidades às aplicações requisitantes e, para isso, uma documentação deve ser feita. Uma ótima ferramenta para este propósito é o *Swagger*.

*A principal vantagem do Swagger é que o mesmo possui uma dependência para as principais linguagens de programação do mercado, e funciona basicamente lendo as anotações do código, o que facilita muito que a cada alteração no código, sua documentação permaneça atualizada. (SILVA, 2018)*

Figura 7 — Exemplo de documentação via Swagger (versão 3.0.0)



Fonte: Autores

Uma outra grande vantagem de se utilizar o *Swagger* é a possibilidade de chamar as APIs através dos *clients* gerados por ele, facilitando a obtenção dos recursos e a realização dos testes.

**5 Considerações Finais**

Se uma API possuir todas as especificações citadas neste trabalho, ela estará preparada para ser utilizada por outras aplicações. O desenvolvimento de APIs em si é um assunto muito amplo, onde é necessário um estudo mais aprofundado em design de APIs para criar um padrão mais conciso. É possível construir uma API coesa seguindo os princípios básicos de documentação, utilizando corretamente os status codes, verbos http, paginação e ordenação dos registros, e estandardizar a nomenclatura de contratos. Todos esses pontos citados anteriormente garantem uma API bem estruturada e de fácil entendimento.

Em certas situações, uma API deverá ser desenvolvida de uma maneira diferente para atender aos requisitos de uma determinada demanda, onde os desenvolvedores acabam por violar os padrões básicos que foram apresentados nesse artigo, gerando um grande retrabalho para refazer a API e deixa-la bem documentada. Esse artigo foi feito para apresentar as melhores práticas e padrões, para que uma API seja construída com documentação e padrão de design, facilitando assim a simples integração com outras aplicações.

**Referências**

ABINADER, Jorge Abílio; LINS, Rafael Dueire. *Web Services em Java*. Rio de Janeiro: Brasport, 2006.

SILVA, William da. *Design de API Rest*. Disponível em: <https://medium.com/@wssilva.willian/design-de-api-rest-9807a5b16c9f>. Acesso em: 15 jan. 2021.

STEPANOV, Konstantin. *HTTP Status Codes Cheat Sheet*. Disponível em: <https://cheatography.com/kstep/cheat-sheets/http-status-codes/>. Acesso em: 20 fev. 2021.

BORGES, Caio Cesar Alves; AZEVEDO, Kenneth Gottschalk de. *Documentação da API do Trabalho de Conclusão de Curso*. Disponível em: <https://unifacef-tcc.herokuapp.com/swagger-ui/>. Acesso em: 01 mar. 2021.