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

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, desempenho 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 exigidos 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 estabeleceram padrões de design em suas APIs, não tendo ciência dos possíveis problemas que isto poderia causar no futuro.

Estas informações foram obtidas perante a análise das diversas APIs públicas que se encontram expostas na internet hoje em dia e, tendo em vista este cenário, foram feitas várias pesquisas sobre quais deveriam ser os pontos mais relevantes a serem levados em consideração pelas APIs para que elas se tornassem propícias a realizar integrações com outras aplicações de forma clara e objetiva.

No decorrer das pesquisas, foram encontrados diversos tipos de padrões de desenvolvimento de APIs considerados válidos e, baseando-se neles, recomenda-se que uma API:

- Utilize os métodos com a intenção de representar uma ação;
- Possua rotas que estejam relacionadas às entidades;
- Disponibilize as opções de paginar e ordenar registros;
- Retorne os códigos de *status* corretos com base em cada tipo de situação;
- Siga um padrão de nomenclatura nas rotas e nos dados trafegados;
- Forneça uma documentação que seja 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.

Considerando a complexidade do tema apresentado neste trabalho, os autores visaram explicar e exemplificar cada um dos assuntos abordados, buscando facilitar o entendimento deles e contemplar, inclusive, os leitores mais leigos.

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 seus dados sejam trafegados 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 a exposição de 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 método, URL, cabeçalho(s), corpo e código de *status*.

2.2 Métodos

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

O método POST também pode ser utilizado para realizar outros tipos de processos, como, por exemplo, efetuar o *login* em um sistema.

2.3 URL

A URL (*Uniform Resource Locator*) também é um dado obrigatório de requisição e, quando está junta com o método, identifica-se 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 propriedades de caminho e/ou parâmetros de requisição, podendo representar um filtro de busca ou identificar um registro.

A Figura 1 mostra alguns exemplos da junção de URL com o método, onde cada rota representa uma operação diferente. Vale ressaltar que, nas rotas de inclusão e alteração, os dados de requisição do registro se encontram no corpo da mensagem (este item será explicado melhor posteriormente).

Figura 1 — Rotas de busca, inclusão, alteração e exclusão de desenvolvedores

Fonte: BORGES; AZEVEDO, 2021.

Existem desenvolvimentos que causam uma quebra de contrato, ou seja, adaptações em rotas que fazem as aplicações requisitantes pararem de funcionar. Esse tipo de problema ocorre em situações como:

- Alteração do método;
- Alteração na rota da URL;
- Adição de dados obrigatórios de requisição;
- Alteração de nome ou tipagem de um dado;
- Remoção de dados de resposta;
- Alteração no código de *status*.

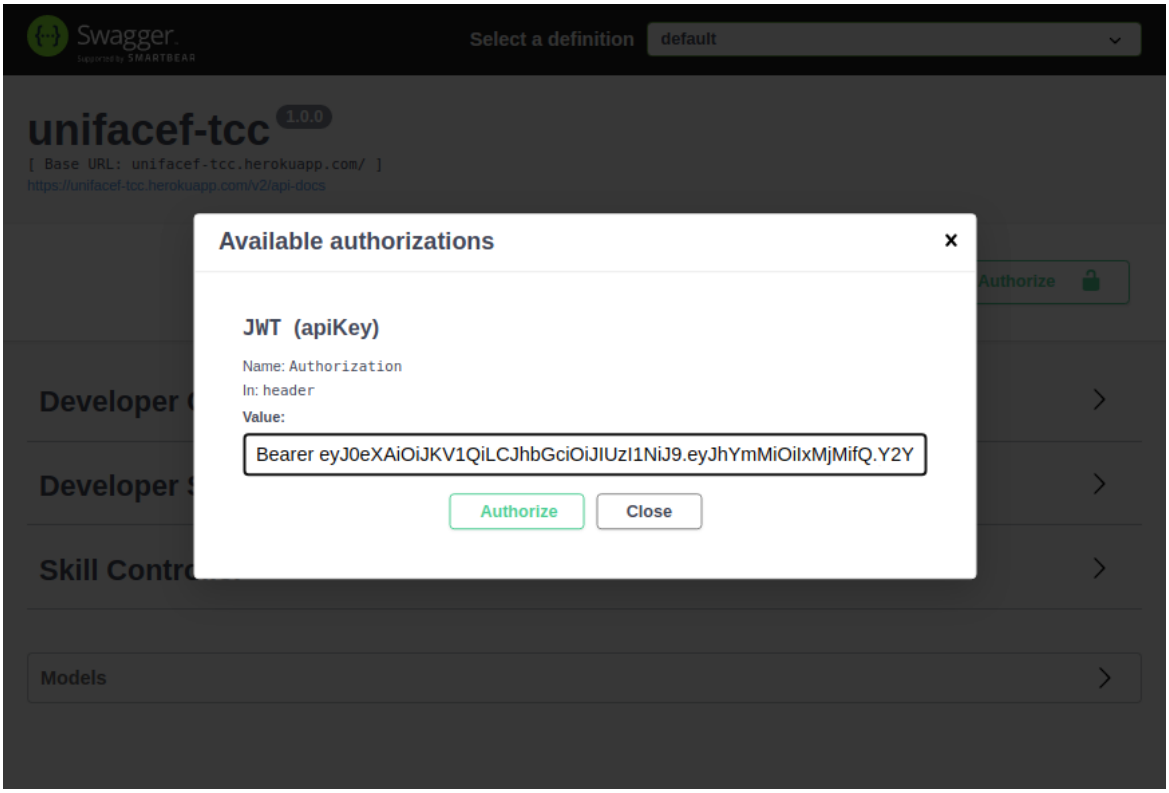
Para resolver esse tipo de problema é sugerível a utilização do versionamento de rotas, mantendo a versão das rotas já existentes e adicionando uma nova versão para as rotas que serão desenvolvidas. Dessa maneira, as aplicações requisitantes que utilizarem a versão atual não sofrerão impacto e as aplicações requisitantes que consumirão a nova versão terão que se adequar às modificações realizadas nela. Um exemplo de versionamento de rotas é adicionar um prefixo nelas para representar a versão (como mostrado na Figura 1).

2.4 Cabeçalhos

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

Um exemplo de cabeçalho é a utilização de chaves de API para fins de autenticidade (como mostrado na Figura 2). Esse tipo de autenticação serve como forma de segurança, garantindo que uma API será utilizada apenas por aplicações requisitantes que são confiáveis. Existem diversos tipos de autenticações em APIs, sendo que as mais comuns são o *Oauth* e o JWT.

Figura 2 — Adição de um *token* JWT no cabeçalho das requisições



Fonte: BORGES; AZEVEDO, 2021.

No caso da API desenvolvida pelos autores, foi utilizada a autenticação JWT e o seu *token* pode ser encontrado na documentação (lembrando que, por se tratar de um dado de segurança, não é uma boa prática deixá-lo exposto em uma documentação, sendo este, um caso de exceção para facilitar os testes feitos pelos usuários).

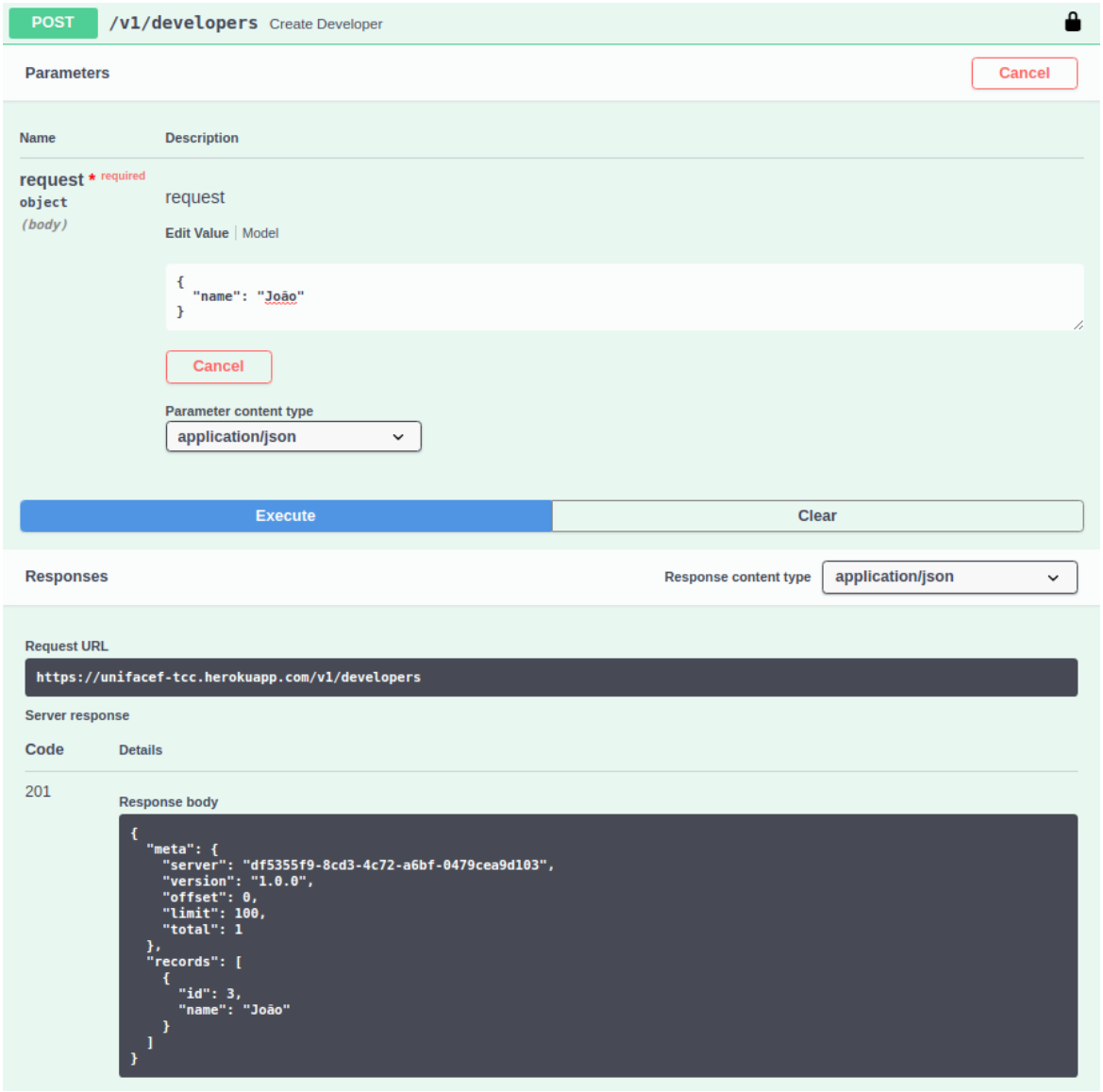
2.5 Corpo

O corpo é um conteúdo opcional introduzido nas requisições e nas respostas de uma chamada de API, sendo necessário informar o tipo do conteúdo no cabeçalho das requisições. Os tipos mais utilizados são XML e JSON.

Na Figura 3 há um exemplo de corpo de requisição e de resposta durante uma chamada de API. Neste exemplo, o corpo de resposta é separado em *meta* (dados de paginação, nome do servidor e versão da aplicação) e *records* (conteúdo dos registros). Esse tipo de padronização facilita a integração por parte das aplicações requisitantes, pois considera-se que todas as respostas vão possuir o

mesmo formato e que sempre haverá registros nas chamadas realizadas com sucesso.

Figura 3 — Dados de requisição e de resposta na inclusão de um desenvolvedor



Fonte: BORGES; AZEVEDO, 2021.

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 durante a remoção de registros).

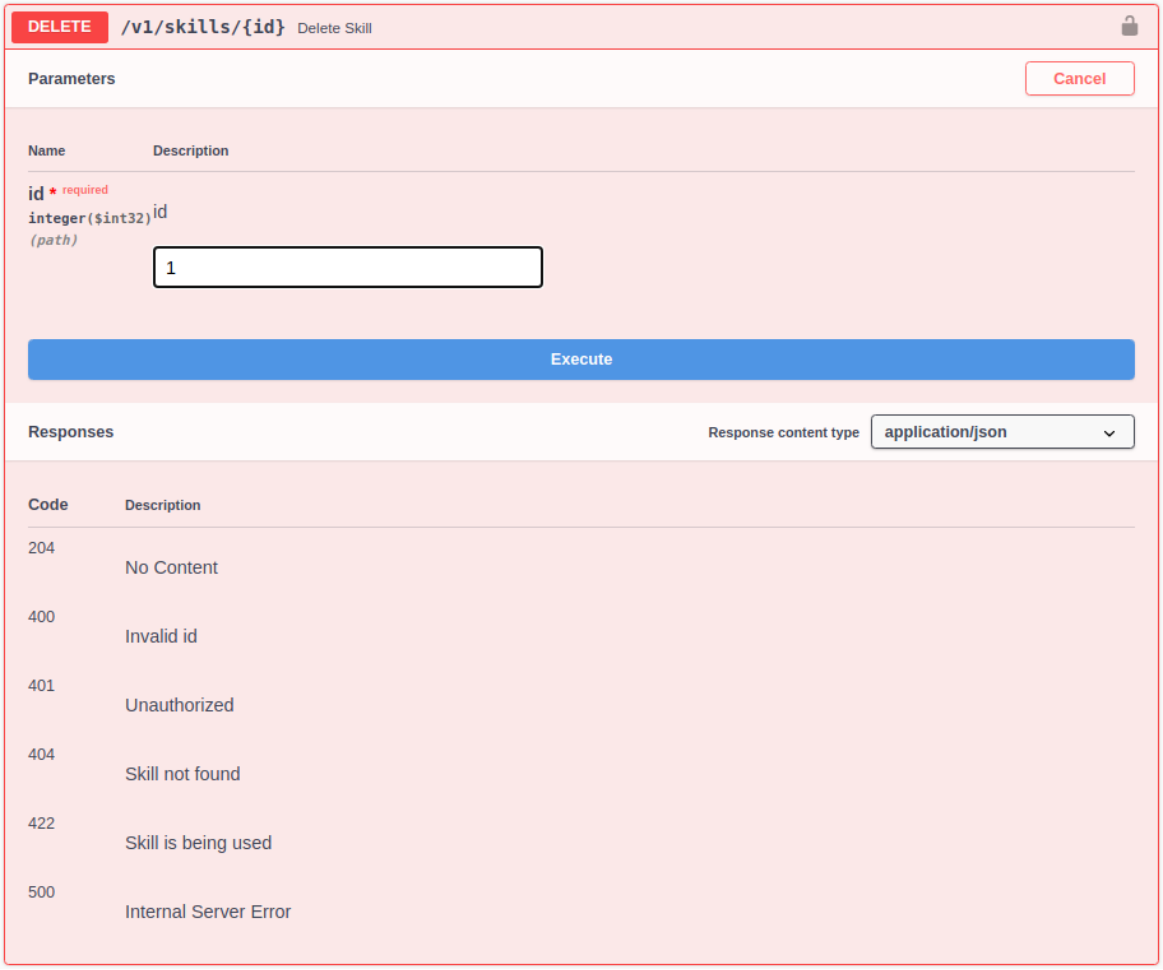
2.6 Código de *status*

O código de *status* trata-se de um número que identifica se a resposta de uma chamada de API foi processada 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*.

A Figura 4 mostra um exemplo dos possíveis códigos de *status* que podem ser retornados durante a exclusão de um registro.

Figura 4 — Exclusão de habilidade



Fonte: BORGES; AZEVEDO, 2021.

Existem diversos códigos de *status* criados até o momento (como mostrado na Figura 5), sendo que cada um deles é utilizado para representar uma situação específica.

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

1xx: HTTP Informational Codes	
100	Continue
101	Switching Protocols
102	Processing WebDAV
103	Checkpoint draft POST PUT
122	Request-URI too long IE7

2xx: HTTP Successful Codes	
200	OK
201	Created
202	Accepted
203	Non-Authoritative Information 1.1
204	No Content
205	Reset Content
206	Partial Content
207	Multi-Status WebDAV 4918
208	Already Reported WebDAV 5842
226	IM Used 3229 GET

3xx: HTTP Redirection Codes	
300	Multiple Choices
301	Moved Permanently
302	Found
303	See Other 1.1
304	Not Modified
305	Use Proxy 1.1
306	Switch Proxy unused
307	Temporary Redirect 1.1
308	Permanent Redirect 7538

307 and **308** are similar to **302** and **301**, but the new request method after redirect must be the same, as on initial request.

4xx: HTTP Client Error Code	
400	Bad Request
401	Unauthorized
402	Payment Required res
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Required
408	Request Timeout
409	Conflict
410	Gone
411	Length Required
412	Precondition Failed
413	Request Entity Too Large
414	Request-URI Too Long
415	Unsupported Media Type
416	Requested Range Not Satisfiable
417	Expectation Failed
418	I'm a teapot 2324
422	Unprocessable Entity WebDAV 4918
423	Locked WebDAV 4918
424	Failed Dependency WebDAV 4918
425	Unordered Collection 3648
426	Upgrade Required 2817
428	Precondition Required draft
429	Too Many Requests draft
431	Request Header Fields Too Large draft
444	No Response nginx
449	Retry With MS
450	Blocked By Windows Parental Controls MS
451	Unavailable For Legal Reasons draft
499	Client Closed Request nginx

5xx: HTTP Server Error Codes	
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout
505	HTTP Version Not Supported
506	Variant Also Negotiates 2295
507	Insufficient Storage WebDAV 4918
508	Loop Detected WebDAV 5842
509	Bandwidth Limit Exceeded nostd
510	Not Extended 2774
511	Network Authentication Required draft
598	Network read timeout error nostd
599	Network connect timeout error nostd

HTTP Code Comments	
WebDAV	WebDAV extension
1.1	HTTP/1.1
GET, POST, PUT, POST	For these methods only
IE	IE extension
MS	MS extension
nginx	nginx extension
2518, 2817, 2295, 2774, 3229, 4918, 5842	RFC number
draft	Proposed draft
nostd	Non standard extension
res	Reserved for future use
unused	No more in use, deprecated

Wikipedia was used to produce all HTTP codes content: http://en.wikipedia.org/wiki/HTTP_status

Fonte: STEPANOV, 2012.

Implementar estes códigos de *status* à risca é uma tarefa complexa e demorada. Para isso, foi feito um levantamento dos códigos de *status* mais utilizados pelas APIs junto à 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:

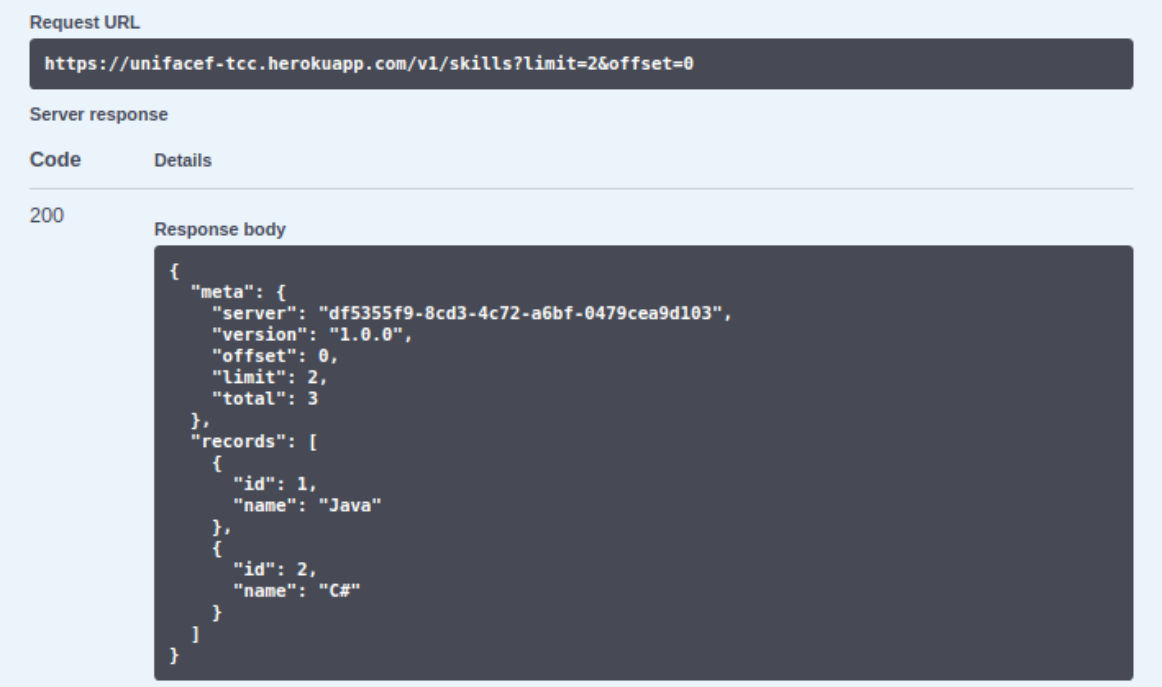
- 500 (*INTERNAL SERVER ERROR*) — utilizado quando ocorre algum tipo de erro inesperado durante o processamento dos dados;
- 502 (*BAD GATEWAY*) — utilizado quando uma dependência externa apresenta algum tipo de comportamento inesperado.

2.7 Paginação e ordenação

É aconselhável que as rotas que retornam mais de um registro possuam a opção de paginação, pois, desta forma, será trafegado apenas os dados utilizados e a API se tornará mais performática.

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 (como mostrado na Figura 6). 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 — Buscar habilidades com paginação



Fonte: BORGES; AZEVEDO, 2021.

Em relação à ordenação, é aconselhável que esta funcionalidade fique dentro da API para evitar que as aplicações requisitantes desenvolvam ela.

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

3 Padrões de nomenclatura

É importante que uma API possua um padrão de nomenclatura nas rotas e nos dados de requisição/resposta, pois as aplicações requisitantes partirão desta premissa quando iniciarem o processo de integração. Há diversos tipos de padrões de nomenclatura, porém, é aconselhável que apenas um deles seja utilizado.

Alguns exemplos de padrões utilizados em rotas são:

- Singular — por exemplo: */v1/product*;
- Plural — por exemplo: */v1/products*;
- *Camel Case* — por exemplo: */v1/marketplaceProducts*;
- *Snake Case* — por exemplo: */v1/marketplace_products*;
- *Spinal Case* — por exemplo: */v1/marketplace-products*.

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. *Camel Case*, *Snake Case* e *Spinal Case* também são exemplos de padrões que podem ser aplicados a estes tipos de dados.

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

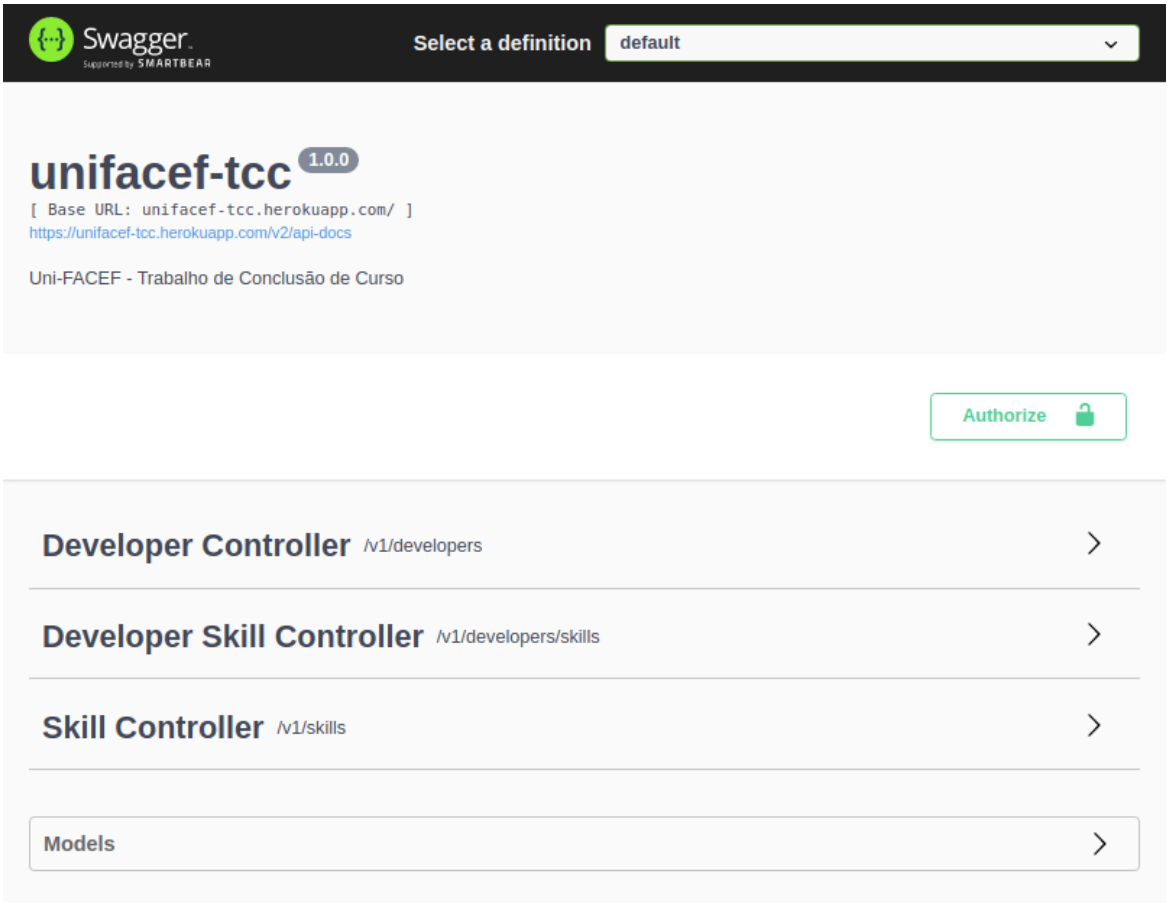
4 Documentação

Após a finalização do desenvolvimento de uma API, é relevante expor as funcionalidades dela às aplicações requisitantes e, para isso, é necessário criar uma documentação para ela. Um exemplo de 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).

A Figura 7 mostra um exemplo de documentação de API feita pela ferramenta *Swagger*.

Figura 7 — Documentação da API desenvolvida pelos autores



Fonte: BORGES; AZEVEDO, 2021.

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

5 Considerações Finais

O desenvolvimento de APIs em si é um assunto muito amplo, sendo necessário um estudo mais aprofundado sobre design de APIs para a criação de um padrão mais preciso e que contemple situações mais específicas. Porém, se uma API possuir as especificações citadas neste trabalho, ela estará pronta para ser integrada com outras aplicações, pois, com base nas pesquisas realizadas, trata-se de um padrão bastante concreto e muito utilizado pela comunidade de desenvolvedores. Considera-se importante criar uma documentação para a API (explicando e exemplificando o padrão utilizado) para que as aplicações requisitantes não tenham dificuldades ao realizar a integração com ela. É normal que existam situações em que uma API deverá ser desenvolvida de uma forma diferente para atender aos requisitos de uma determinada demanda, sendo necessário violar alguns itens apresentados neste trabalho — e isso não é um problema, desde que a API esteja bem documentada. Concluindo-se, os autores obtiveram resultados satisfatórios ao analisar as aplicações que fizeram integrações com a API desenvolvida por eles após exporem ela publicamente na internet.

Referências

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

SILVA, William da. *Design de API Rest*. **Medium**, 2018. Disponível em: https://medium.com/@wssilva.willian/design-de-api-rest-9807a5b16c9f. Acesso em: 15 jan. 2021.

STEPANOV, Konstantin. *HTTP Status Codes Cheat Sheet*. **Cheatography**, 2012. 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*. **Swagger UI**, 2021. Disponível em: https://unifacef-tcc.herokuapp.com/swagger-ui/. Acesso em: 01 mar. 2021.