O GUIA COMPLETO DE APIS RESTFUL

por Thiago Lima

LinkApi

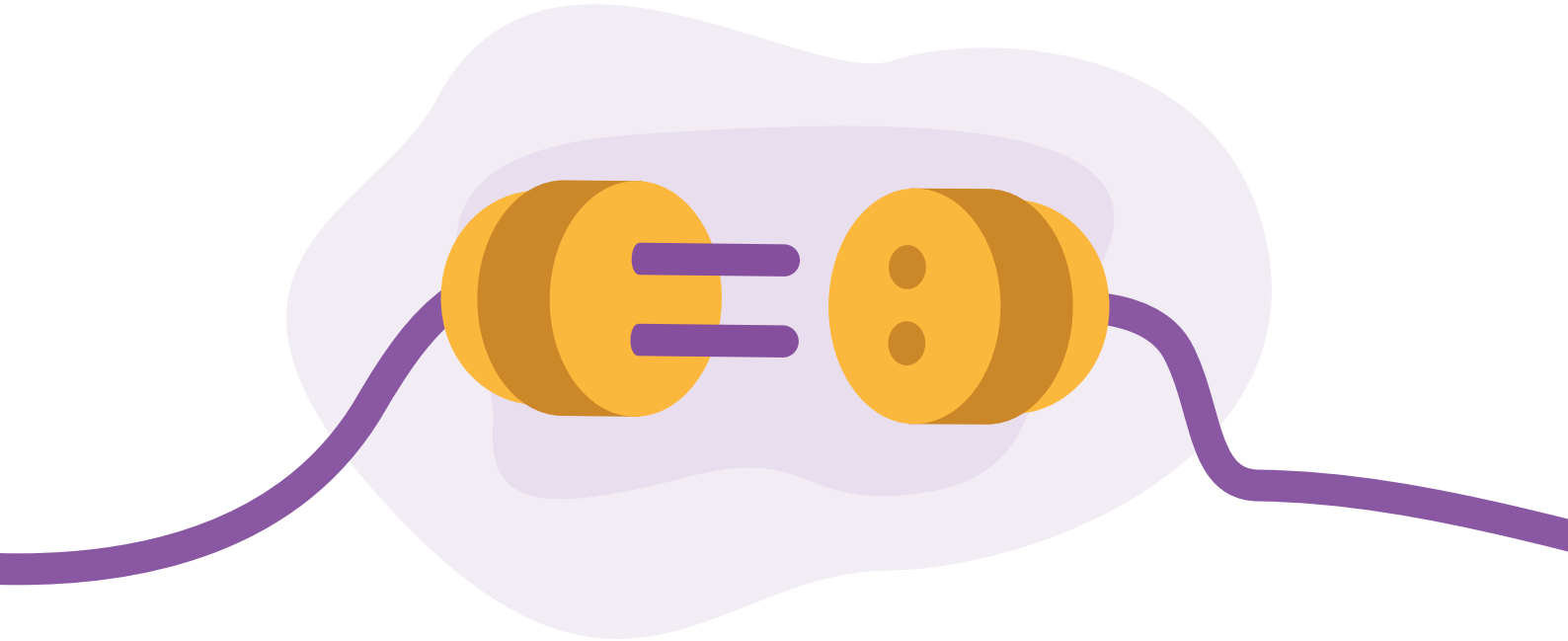
Conteúdo

Um desenvolvedor norteamericano usa em média

18 APIs

para construir uma aplicação. E todo ano surgem

2 mil novas APIs



O mercado de APIs como serviço espera ultrapassar

1 bilhão de dólares

até 2020, com taxa de crescimento anual acima de 30%.

O que preciso saber para entrar nesse mercado?

Mas afinal, o que é uma API RESTful?

API quer dizer “Application Programming Interface”, em outras palavras, é uma interface de comunicação entre desenvolvedores.

Em relação ao REST (Representational State Transfer), ele é um conjunto de regras e boas práticas para o desenvolvimento dessas APIs.

Pense no seguinte cenário como programador, imagine que você tem um grande conhecimento em gastronomia, e então você cria um banco de dados para armazenar esses dados. Mas depois você percebe que outras pessoas podem se beneficiar disso. Pronto, é aí que entram as APIs, nesse caso, você poderia desenvolver uma interface, ou seja sua API, para que outros desenvolvedores pudessem criar aplicativos ao redor desse conhecimento, por exemplo: recomendação de vinhos, ranking de receitas, e por aí vai.

Como funciona uma API na prática?

Primeiro ponto que precisa ser esclarecido, quando falamos de APIs toda a comunicação dessa interface é feita via web, ou seja, tudo é feito através de uma requisição a uma URL, que por sua vez, traz uma resposta.

Voltando ao exemplo anterior, pense que você é um desenvolvedor e quer saber dos melhores vinhos da região sul do Brasil, nesse caso você faria uma requisição dessa informação para uma URL dessa API, e ela te retornaria uma resposta. Simples, não?

Entendendo uma requisição

O endpoint

A URL nada mais é que o caminho para fazer a requisição, porém é interessante ressaltar que ela segue a seguinte estrutura:

Base URL

Esse é o início da URL da requisição, aqui você basicamente falará a informação de domínio que se repete em qualquer requisição. Por exemplo:

```
https://api.minhagastronomia.com
```

Resource ou Path

O recurso é o tipo de informação que você está buscando, ou seja, vamos simular que estamos buscando saber sobre vinhos, então acrescentamos o recurso vinhos:

```
https://api.minhagastronomia.com/vinhos
```

Query String

A query string são os parâmetros daquela requisição, então, se eu quisesse saber os melhores vinhos da região sul do Brasil, eu incluiria esses parâmetros ?pais=brasil&regiao=sul e nossa URL ficaria assim:

```
https://api.minhagastronomia.com/vinhos?pais=brasil&regiao=sul
```

Como podem ver acima, por se tratar de parâmetros da URL você usa a “?” e caso queira utilizar mais de um parâmetro você utiliza o “&”.

Observação: A Query String não é somente utilizada para filtros, ela pode ser utilizada como parâmetros de paginação, versionamento, ordenação e muito mais.

O método

O método te ajuda a informar o tipo de ação que você está fazendo naquela requisição. Dentre os principais métodos, temos:

GET	POST	PUT/PATCH	DELETE
Busca dados	Envia dados	Atualiza dados	Exclui dados

Headers

Headers ou cabeçalhos permitem que você envie informações adicionais na requisição. Ele pode ser utilizado para inúmeras funções, como: autenticação, formatação de objeto, e muito mais.

Para utilizá-lo é simples você coloca a propriedade, seguido dois pontos e o valor, tudo entre aspas, exemplo:

```
"Authorization: token123242343534"
```

Body

O body é o corpo da mensagem que você quer enviar na requisição. Ele é utilizado somente nos métodos de POST, PUT, PATCH, ou seja, ele contém o dado a ser processado pela API, e por isso ele não é necessário em métodos de leitura de dados.

HTTP Status Codes

Para facilitar o entendimento das respostas das APIs existem padrões de códigos de status que podem ser utilizados.

Os códigos mais utilizados para as respostas de uma requisição são o 200 (OK), o 201 (created), o 204 (no content), o 404 (not found), o 400 (bad request), e 500 (internal server error).

Como padrão, os códigos de sucesso tem o prefixo 20x, os de redirecionamento 30x, os de erro do cliente 40x e os de erro de servidor 50x.

Autenticação

Obviamente não podemos falar de APIs sem segurança, afinal estamos falando da WEB. Como principais métodos de autenticação de APIs, temos:

Basic authentication

Baseado em usuário e senha codificados em Base64 e utilizado no header da requisição.

Secret token

Token de acesso que pode ser limitado a escopo, e que é enviado na requisição pelo Header ou pela Query String. Nesse caso temos padrões famosos como oAuth e JWT.

Se quiser entender ainda mais sobre os requests, recomendo olhar uma API de testes para você conseguir "brincar" com algumas chamadas, como por exemplo o https://reqres.in.

Pronto! Agora você já sabe o básico sobre o funcionamento de uma API, e um pouco do que ela é capaz de fazer.

Agora é o momento de nos aprofundarmos, então, o próximo capítulo vai abordar o protocolo HTTP, seus métodos e os códigos de status.

Métodos e códigos de status do protocolo HTTP

O HTTP, ou, HyperText Transfer Protocol é o protocolo de comunicação de dados do mundo da internet (Web). Em outras palavras, ele facilita e define os padrões de comunicação dos dados entre um Webbrowser e um Web server.

Quando falamos de métodos eles basicamente são ações permitidas dentro de uma API. E ao falar de códigos de status, eles são os retornos padrões de uma API. E ambos fazem parte das definições do protocolo HTTP.

Agora sim, vamos nos aprofundar mais sobre esses métodos e códigos de status permitidos na utilização desse protocolo.

Métodos

Vamos falar basicamente dos 5 principais, e que provavelmente será os que você irá passar 99% do seu tempo como desenvolvedor.

O método GET

Esse será o método que você mais irá utilizar, pois ele é utilizado para buscar dados em APIs. A ideia dele é simples, você utiliza uma estrutura de Query String para enviar os parâmetros da sua consulta e o servidor retorna os dados em caso de sucesso, ou caso você não envie a Query String na URL, a API irá retornar todos os dados sem o filtro. Vamos continuar o exemplo de uma API que traz dados sobre gastronomia e sua URL base é

```
https://api.minhagastronomia.com
```

Como desenvolvedor eu gostaria de consultar dados sobre vinhos brancos de até 100 reais. Então eu faria a requisição

GET **https://api.minhagastronomia.com/vinhos?ate_preco=100&tipo=branco**

Lembrando que a API teria que fornecer os parâmetros da Query String (ate_preco e tipo) em sua documentação para utilização na requisição.

Ou então, imaginem que eu já tenho a lista de vinhos devido a minha última requisição, e quero me aprofundar mais em um vinho específico que tem o identificador único 22. Nesse caso, eu faria a requisição abaixo, no qual, o 22 seria o valor do parâmetro {id} da URL. Nesse caso, não é utilizada uma Query String, e sim um parâmetro de URL, pois o {id} é obrigatório.

GET **https://api.minhagastronomia.com/vinhos/22**

O método POST

Agora pense que você não encontrou um vinho que gosta, e gostaria de cadastrar esse vinho nessa API. Nesse caso, você utilizaria o método POST, e enviaria os dados desse novo vinho no corpo (Body) da requisição, por exemplo:

POST **https://api.minhagastronomia.com/vinhos**

Body

```
{
  "nome": "Maycas Reserva Sumaq Chardonnay 2016",
  "preco": 49,
  "pais": "Chile",
  "recomendado": "peixe, frutos do mar",
  "tipo": "branco"
}
```

E pronto, você acabou de cadastrar um novo vinho.

Os métodos PUT e PATCH

O método PUT é utilizado em cenários de criação e atualização de dados, ou seja, se eu enviasse o vinho e ele já existisse, então ele apenas seria atualizado, caso contrário, seria criado um novo vinho.

Lembrando que isso é uma recomendação do protocolo e que tudo depende do desenvolvedor implementar ou não.

Por sua vez, o método PATCH serve para atualizações parciais daquele registro, ou seja, vamos imaginar que aquele vinho cadastrado gerou o identificador 55, e que agora eu gostaria de atualizar apenas o preço dele, a requisição ficaria dessa forma:

PATCH **https://api.minhagastronomia.com/vinhos/55**

Body

```
{
  "preco": 49,
}
```

O método DELETE

Como o próprio nome já diz, o método DELETE é o responsável por deletar os registros. Portanto, supondo que você queira deletar o mesmo vinho que você acabou de atualizar, você faria a seguinte requisição:

DELETE **https://api.minhagastronomia.com/vinhos/55**

Outros métodos

Existem também outros métodos como OPTIONS, HEAD e TRACE, porém eles são raramente são utilizados.

Códigos de status

Agora vamos aos códigos de status permitidos pelo HTTP, e que podem ser agrupados por tipo. Esses códigos são retornados pelo servidor, e servem para informar ao desenvolvedor o que aconteceu após a sua requisição.

Grupo 1

Grupo de status que dá respostas informativas, existem poucas e raramente são utilizadas.

Grupo 2

Grupo de status utilizado em caso de sucesso na requisição. Os mais utilizados desse grupo são:

200 OK — Código mais utilizado, e que indica que a requisição foi processada com sucesso.

201 Created — Indica que a requisição foi bem sucedida e que um novo registro foi criado como resultado. Resposta utilizada em requisições do método POST.

Grupo 3

Define respostas de redirecionamento. São utilizadas com o intuito de informar o cliente sobre mudanças na requisição e o redirecionamento para uma nova URL. Esses são os mais comuns:

301 Moved Permanently — Informa que o recurso A agora é o recurso B, de forma que quando o cliente solicitar o recurso A ele será automaticamente encaminhado ao recurso B.

304 Not Modified — Resposta utilizada em cenários de cache. Informa ao cliente que a resposta não foi modificada, e nesse sentido, o cliente pode usar a mesma versão em cache da resposta.

307 Temporary redirect — Se trata de um redirecionamento de uma página para outro endereço, porém que é com caráter temporário, e não permanente.

Grupo 4

Preste bastante atenção aqui, pois esse grupo informa os erros do lado do cliente. Em outras palavras, caso você monte a requisição de forma incorreta, o servidor irá retornar um erro desse grupo. Vamos aos principais:

400 Bad Request— Essa resposta significa que o servidor não consegue entender sua requisição, pois existe uma sintaxe ou estrutura inválida.

401 Unauthorized — Erro que informa que existe uma camada de segurança no recurso solicitado ao servidor, e que você não está utilizando as credenciais corretas nessa requisição. Lembrando que as credenciais podem ser algo como usuário e senha, token, etc. E tudo vai depender da API que você estará consumindo.

403 Forbidden — Nesse caso as credenciais do cliente são reconhecidas, porém o servidor informa que ele não tem os direitos de acesso necessários para aquele recurso.

404 Not Found — Código que informa que o servidor não encontra o recurso solicitado pelo cliente.

429 Too Many Requests — Não é tão comum, mas pode ser utilizado para informar ao cliente que ele excedeu o limite permitido de requisições.

Grupo 5

Tão importante quanto o grupo 4, nesse grupo estão os erros do lado do servidor. Em outras palavras, pensem que nesse caso a requisição foi feita corretamente pelo lado do cliente, porém houve um erro de processamento no servidor. Vamos aos principais códigos desse grupo:

500 Internal Server Error — Erro mais genérico desse grupo e muito utilizado. Ele informa que o servidor encontrou um cenário inesperado de erro que não soube tratar, e por isso não conseguiu retornar uma resposta na requisição do cliente.

503 Service Unavailable — Erro normalmente utilizado para informar que o servidor está fora do ar, em manutenção ou sobrecarregado. Existem diversos códigos de status disponíveis para uso no protocolo HTTP, mas sem dúvidas, nesse capítulo, citei os que são mais utilizados.

Segurança em APIs RESTful

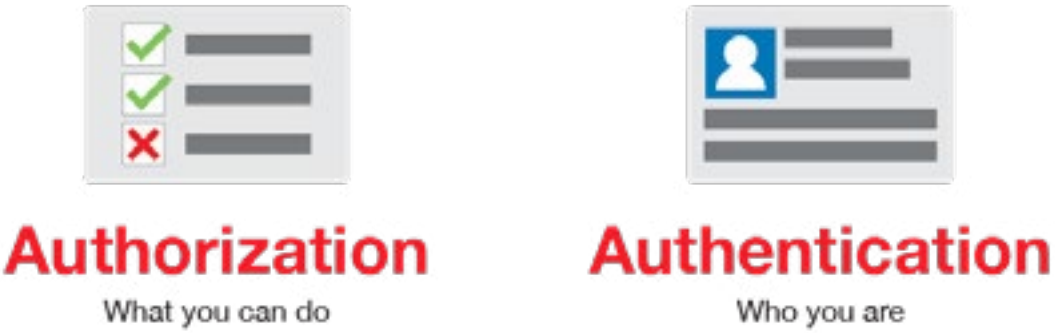
Nesse capítulo vou abordar um tema polêmico e importante sobre APIs — Segurança. Já ouvi muita empresa perguntar "mas esse negócio de APIs é seguro mesmo?"

Além disso, indústrias como a bancária, que normalmente é paranóica no assunto segurança, será obrigada a expor APIs ao mercado, o famoso Open Banking até 2020 por uma mudança na legislação.

Autenticação vs Autorização

Autenticação é como se você fosse para uma festa, e o segurança exigisse suas credenciais como nome na lista e RG para liberar sua entrada.

Autorização é como se você já estivesse dentro da festa e quisesse subir ao palco do show, mas para isso você precisaria ser autorizado pelo segurança.



Principais métodos de autenticação

Basic Authentication

Basic authentication é um esquema de autenticação bem simples especificado no protocolo HTTP.

O cliente envia uma requisição com o header (Authorization) que contém a palavra Basic e o nome de usuário e senha, separados por dois pontos no formato de base64.

Por exemplo, para autorizar o usuário thiago com senha thiago@123, o client enviaria na requisição, o seguinte header:

```
Authorization: Basic dGhpYWdvOnRoaWFnb0AxMjM=
```

Como o formato base64 é muito fácil de ser decodificado, a basic authentication só é recomendada quando são utilizados outros complementos de segurança, como por exemplo, o certificado HTTPS.

São comuns casos em que é substituído o usuário e senha por um token, mas isso é exceção e não é recomendado pela especificação.

API Keys

API Keys é um método de autenticação que teoricamente veio para resolver alguns problemas do modelo de Basic Authentication.

A ideia do método de API Keys é simples, o servidor gera uma chave de acesso única para o client, e para utilizar a API, o client envia essa chave única em toda requisição. É um método bem simples e rápido de ser implementado, e durante anos foi o método mais utilizado pelos desenvolvedores.

A grande questão é que esse método serve apenas para autenticação, e não para autorização, ou seja, em teoria um usuário com uma Key válida tem acesso a todas as operações de uma API.

Além disso, ela funciona como qualquer requisição HTTP, e se algum ponto de rede estiver inseguro, ela pode ser facilmente interceptada e utilizada para acesso indevido da API.

Normalmente essa key é utilizada no header, por exemplo, imagine que você recebeu do servidor a seguinte key: “thi123456”, nesse sentido para eu me autenticar na API eu faria uma requisição com o seguinte header:

```
Api-key: thi123456
```

Vale ressaltar que o parâmetro de header pode variar de acordo com a API, ou seja, pode ser “x-Api-key” por exemplo, ou então, pode ser que o envio dessa key seja via Query String, e não pelo header, algo menos seguro ainda, pois fica extremamente visível para quem vai atacar.

OAuth

O OAuth está na sua versão 2.0, e não é apenas um método de autenticação, e sim um protocolo completo com diversas especificações de segurança.

Ele é extremamente útil para o processo de autenticação e autorização, e por isso, atualmente é o método mais recomendado para o cenário de APIs. Vamos entender alguns conceitos básicos do OAuth 2:

- **Resource Owner:** entidade dona do recurso, ou seja, que é capaz de controlar o acesso a um recurso. Normalmente é o próprio usuário.

- **Resource Server:** servidor que possui os recursos, e por sua vez, recebe as requisições.

- **Authorization Server:** servidor que gera tokens de acesso para permitir que o client acesse os recursos autorizados pelo resource owner.

- **Client:** aplicação que acessa os recursos do resource server.

Para entender melhor, vamos supor que você desenvolveu uma aplicação que utiliza dados do usuário do Facebook, então vamos simular como seria um fluxo básico de autenticação via OAuth 2.0:

1. O usuário acessa seu site ou app que teria um botão de “integre ao facebook”, sendo que o seu site ou app seria o client.

2. Ao clicar no botão, o usuário é redirecionado para a tela de login do Facebook (authorization server).

3. Após o usuário informar as credenciais, o Facebook fornece um código de acesso ao client.

4. Então o client solicita autorização aos recursos (endpoints da API do Facebook) para o resource owner (que é o próprio usuário) enviando o código de acesso recebido anteriormente.

5. O authorization server por sua vez, verifica se o código de acesso é valido, e caso positivo, ele gera um token de acesso para retornar ao client.

6. Por último, agora que o client já tem o token de acesso e autorização aos recursos, então a cada requisição, o resource server (API do facebook) irá responder com os dados protegidos.

Outros assuntos acerca da especificação do oAuth e que são interessantes para implementação:

OpenID Connect: Extensão da funcionalidade de autenticação do OAuth. Como o OAuth foi projetado para autorização, o OpenID Connect acaba sendo um ótimo complemento na segurança de APIs, conseguindo ajudar no sentido de provar que o usuário é realmente quem ele diz que é.

JWT: Formato de token seguro que utiliza JSON como base.

SAML

Temos também o SAML, ou Security Assertion Markup Language. Que é um padrão aberto que permite que provedores de identidade (IDP) passem credenciais de autorização para provedores de serviços (SP). Em outras palavras, o usuário pode usar as mesmas credenciais para entrar em aplicações diferentes.

Na prática, o SAML ativa o Single Sign-On (SSO), algo bem interessante para a experiência dos usuários, pois eles fazem o login uma única vez e consegue navegar em todos as aplicações com o padrão implementado.

Não vejo outro cenário de aplicação do SAML além de casos que o login único seja premissa, pois ele utiliza o XML na comunicação, que já é algo ultrapassado para o desenvolvimento WEB que estamos vivendo atualmente.

Boas práticas de segurança

Em uma visão direta ao ponto, essas são as principais recomendações de segurança para a grande maioria dos cenários de APIs RESTful:

Mantenha simples a sua implementação

Pense sempre na experiência do desenvolvedor que irá consumir, muitas APIs perdem engajamento da comunidade pela complexidade no consumo devido a implementações de segurança.

Sempre utilize HTTPS

Essa é uma dica básica de segurança na WEB para você manter as mensagens trafegadas seguras e criptografadas, dificultando qualquer interceptação da requisição. É natural que haja uma queda de performance, porém, se você estiver utilizando HTTP 2 existem diversos workarounds de otimização.

Não exponha dados sensíveis na URL

Uma má prática comum é a passagem de dados relativos a segurança via URL.

```
GET https://api.test.com/orders/?apiKey=thiago1234
```

Isso facilita muito a vida de alguém com más intenções na sua API.

Considere a utilização de OAuth

O OAuth hoje é sem dúvidas a especificação mais completa para APIs RESTful, por isso, utilize tudo que há de melhor no OAuth, apenas tomem muito cuidado para não deixar a implementação muito complexa e impactar negativamente a experiência do desenvolvedor que irá consumir sua API.

Utilize o Rate Limit

Essa é uma prática interessante e que pode ajudar a prevenir ataques que focam em deixar sua infraestrutura indisponível por uma quantidade alta de requisições. Com a implementação desse header você consegue restringir o número de requisições do client.

Com a implementação, você sempre retorna os seguintes headers na resposta de uma requisição:

X-Rate-Limit-Limit — Número de requisições permitidas durante um período específico (dia, hora, etc)

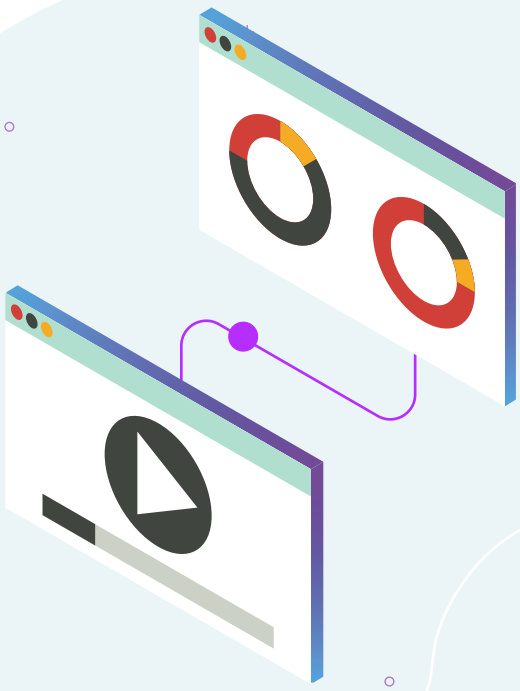
X-Rate-Limit-Remaining —Número de requisições restantes do período corrente

X-Rate-Limit-Reset — Segundos restantes do período corrente

Além disso, em casos que o client ultrapasse o limite de requisições, você retorna o status code 429 — Too Many Requests.

Timestamp na requisição

Se for uma API em que segurança é um ponto crítico, uma boa dica é criar um header customizado com o timestamp. Assim o server pode comparar o timestamp atual, e aceitar apenas as requisições que estiverem próximas a um período de tempo (por exemplo: 2 minutos).



Isso é um procedimento simples, e que pode prevenir ataques básicos de replay em tentativas de força bruta.

Valide o parâmetros da requisição

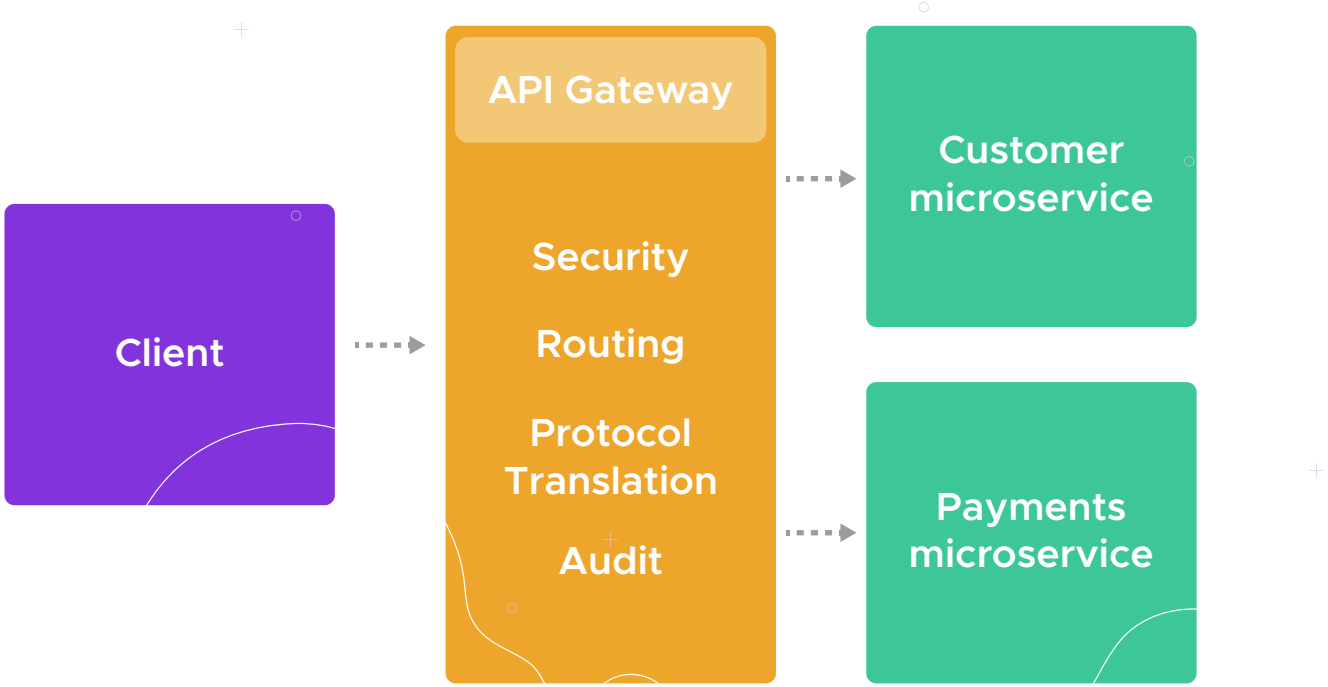
Por questões de segurança, sempre valide os paramêtros da requisição antes da execução da lógica do negócio.

Caso a requisição esteja com parâmetros não especificados na sua documentação, rejeite imediatamente a requisição, pois pode ser indício de um ataque malicioso.

Além disso, para não impactar negativamente a experiência de quem não está mal intencionado, retorne mensagens de erros claras sobre o assunto, e disponibilize exemplos da requisição feita corretamente.

Utilize uma ferramenta de API Gateway

Existem diversas ferramentas que facilitam as implementações dessas recomendações centralizando toda a inteligência de segurança em uma camada específica de arquitetura. Nesse desenho conseguimos entender esse modelo:



Algumas ferramentas recomendadas para essa arquitetura: LinkApi, Kong e WSO2 e Apigee. Ficou clara a importância de ter a segurança da sua API em conformidade com as melhores práticas do mercado, né?

Versionando APIs RESTful

Sabe aquela situação horrível de ter desenvolvido um aplicativo, e ele parar de funcionar, pois alguém atualizou a versão da API que você consumia?

Se você desenvolve ou quer desenvolver APIs, a solução desse problema é simples, e se chama "versionamento".

Existem diversas maneiras de versionar sua API, e nesse capítulo vou listar as principais formas.

Usaremos o exemplo daquela primeira API de gastronomia do primeiro capítulo:

```
https://api.minhagastronomia.com
```

Versionamento pela URL

Quando versionamos pela URL temos três maneiras de fazer isso: subdomínio, path ou query string. No modelo de **subdomínio** você especifica a versão logo no início da URL, por exemplo:

```
GET https://api-v1.minhagastronomia/vinhos
```

Notem que logo após o api, eu coloquei o "-v1", que especifica a versão, e nesse caso quem for consumir pode alterar apenas o subdomínio na requisição.

No modelo de **path** você especifica a versão após a base url, por exemplo:

```
GET https://api.minhagastronomia/v1/vinhos
```

Essa é uma das abordagens mais utilizadas, pois além de dar um visual mais clean na URL, facilita a navegação para outras versões da API, ou seja, é mais dev-friendly comparado a outras abordagens.

Tem quem é mais criativo, e prefere especificar a **versão via query string**, por exemplo:

```
GET https://api.minhagastronomia/vinhos?version=1.0&pais=brasil
```

Notem que na URL eu coloquei o parâmetro via query string "version" e especifiquei a versão 1.0 da API.

Essa é uma abordagem que já foi muito utilizada, mas que caiu em desuso, pois além de prejudicar a navegação para outras versões, a legibilidade da URL fica ruim em cenários de muitos parâmetros na query string.

Versionamento pelo Header

O versionamento via header pode ser **via HTTP content-type**, ou seja, utilizando o header "Accept" na requisição, como por exemplo:

```
GET https://api.minhagastronomia/vinhos
Accept: application/vnd.gastronomia.v2+json
```

Se formos seguir a especificação ao pé da letra, essa é a maneira correta de fazer o versionamento de APIs.

Também existe a abordagem de utilização de um **header customizado** para especificar a versão, como por exemplo:

```
GET https://api.minhagastronomia.com/vinhos
api-version: 2
```

Note que nesse caso eu criei um novo header chamado "api-version" para que o consumidor especifique a versão desejada na requisição.

O ponto positivo é que sua URL permanece clean e intacta, o ponto negativo é que não é dev-friendly, pois a requisição tem que ser feita com muito mais cuidado.

Boas práticas

Versionamento é uma das maiores discussões ao redor da comunidade de APIs, pois a própria especificação não cria uma boa experiência ao developer.

Por isso, minha recomendação é seguir o melhor das duas abordagens com foco em proporcionar uma ótima experiência ao desenvolvedor que irá consumir a API, ou seja, o versionamento não pode influenciar negativamente na jornada do consumidor.

Por outro lado, o criador da API também deve conseguir ter um mecanismo simples que facilite a implementação de conceitos de DEVOPS, como CI (Continuous Integrations) e CD (Continuous Delivery).

Nesse sentido, gosto da abordagem de utilização da versão na URL via PATH para ser responsável pela estabilidade estrutural da API, e por pacotes maiores de atualização.

Acrescentando também um header customizado que serve para especificar a data da sub-versão, e é o responsável por atualizações menores, como: campos do metadado, parâmetros da Query String, e coisas do tipo. Nessa abordagem, a requisição ficaria da seguinte forma:

```
GET https://api.minhagastronomia.com/v1/vinhos
    api-version: “2019-06-01”
```

Como eu disse, não existe mundo perfeito quando falamos de versionamento em APIs, é uma discussão que muitas vezes vai pelo gosto pessoal do desenvolvedor, então o melhor a se fazer é entender bem o cenário, e implementar a abordagem que faz mais sentido!

E se quiser menos “dor de cabeça” nessa implementação, escolha um API Management para te ajudar.

Paginação, Ordenação e Filtros

Imaginem que agora nossa API de gastronomia já está com muitos dados, e que ao fazer um GET de vinhos, ela me retorne mais que 10 mil registros, e isso está demorando muito para retornar aos clients que consomem esse recurso.

É exatamente isso que a paginação, a ordenação e os filtros resolvem.

Em resumo, é a implementação capaz de fazer sua API trabalhar de forma mais eficiente com os dados na comunicação entre client e server, trafegando apenas o necessário!

Para facilitar a visão, vou listar aqui os principais motivos de você implementar esses conceitos:

Experiência do client

Como você estará fragmentando o resultado no server, a resposta da mensagem trafegada fica bem menor, e por isso, o client consegue o resultado da requisição muito mais rápido, criando uma experiência agradável em termos de performance ao desenvolvedor que está consumindo a API.

Utilização inteligente de infraestrutura

Como o consumo dos dados fica sob demanda, isso ajuda muito na escalabilidade vertical e horizontal de uma API. Na prática, esse modelo utiliza recursos de infraestrutura de forma mais inteligente.

Chega de teorias, agora vamos analisar os principais tipos utilizados no mercado.

Paginação

Cursor

A paginação baseada em cursor funciona por uma chave única e sequencial que indica a partir de que registro os dados serão retornados.

Em outras palavras, imaginem que você quer os registros a partir do vinho de ID 156 até o ID 200. Então, você faria a seguinte requisição:

```
GET https://api.minhagastronomia/vinhos?since_id=156&max_id=200
```

Notem que foram adicionados os parâmetros since_id e max_id, que nesse caso, são os cursores de navegação dentro da API, e que delimitam os resultados a serem retornados pela requisição.

Normalmente as APIs com paginação baseadas em cursores, retornam no resultado da requisição os parâmetros para navegação da página anterior, e também da próxima. O nome dos parâmetros de cursores podem mudar de acordo com a implementação da API.

Page e PageSize

A paginação baseada em page e pagesize, como o próprio nome já diz, é utilizada através dos parâmetros de número da página a ser navegada e o seu respectivo tamanho (em número de registros).

Vamos supor que quer os dados de 10 vinhos da terceira página, então você faria a seguinte requisição:

```
GET https://api.minhagastronomia/vinhos?page=3&page_size=10
```

Offset e Limit

A paginação baseada em offset e limit é utilizada a partir de um deslocamento de registros.

Em outras palavras, você especifica no offset a partir de qual registro você quer os dados, e no limit você especifica o limite de registros a serem retornados.

Agora vamos imaginar que você quer 30 vinhos a partir do décimo registro, então você faria a seguinte requisição:

```
GET https://api.minhagastronomia/vinhos?offset=10&limit=30
```

Filtros e Ordenação

Sobre filtros e ordenação não tem muito segredo, via parâmetros de query string você consegue dar mais inteligência a sua API, além de dar mais capacidade ao client para trabalhar com os dados sob demanda. Exemplo prático de filtro:

```
GET https://api.minhagastronomia/vinhos?pais=Brasil&estado=RS&de_preco=100&ate_preco=200&status=disponivel
```

No exemplo acima, estou solicitando todos os vinhos do Brasil, do estado do Rio Grande do Sul, com o preço entre 100 e 200 reais e que estão disponíveis para compra. Também temos a possibilidade de utilizar **alias** para filtros muito utilizados, por exemplo:

```
GET https://api.minhagastronomia/vinhos/disponiveis?estado=RS&de_preco=100&ate_preco=200
```

Nesse caso, eu utilizo o sub-recurso “disponiveis” que contém o filtro pré-estruturado e facilita o consumo do desenvolvedor.

Exemplo prático de ordenação:

```
GET https://api.minhagastronomia/vinhos?sort=pais:asc,estado
```

No exemplo acima, notem que eu inclui o parâmetro “sort” que contém o campo chave da ordenação seguido por “:” para especificar se quero os dados em ordem ascendente ou descendente, além da vírgula de separação dos campos.

Esse não é o único padrão dessa técnica, mas particularmente acho bem clean.

Boas práticas

Offset e limit com filtros de “range”

Minha recomendação é a utilização da técnica offset e limit para paginação, pois além de ser o método mais utilizado ultimamente, é o que proporciona melhor experiência ao consumidor da API.

A navegação por page e page size, requer recálculos das páginas em cenários de mudança do page size, e isso acaba criando uma complexidade adicional e que prejudica a experiência do desenvolvedor.

E se você for utilizar page e page size, por favor, especifique na sua documentação em que página é iniciada a busca dos dados, pois já vi inúmeros casos de que a página 0 e 1, retornam os mesmos registros!

Tenha default values

Tenha valores padrões em parâmetros de paginação, como por exemplo, limit padrão de 500 objetos, e limit máximo de 1000 objetos, dessa forma, além de você facilitar o consumo em casos do client não enviar o parâmetro, você também protege a performance e escalabilidade da sua API.

Não exagere em parâmetros de Query String

Minha recomendação nesse assunto é que você não exagere em parâmetros de query string, ou seja, faça uma análise do que realmente faz sentido, caso contrário, você estará poluindo a URL desnecessariamente, e deixando tudo mais complexo para o desenvolvedor que irá consumir a API.

Outro ponto importante também é não colocar parâmetros como obrigatórios casos que não fazem sentido, por exemplo:

```
GET https://api.minhagastronomia/vinhos?pais=Brasil&estado=RS&de_preco=100&ate_preco=200&status=disponivel
```

Imaginem que nessa requisição acima, o parâmetro de status fosse obrigatório, isso não faria o menor sentido, pois, caso eu quisesse todos os vinhos independente do status, eu teria que concatenar diversas requisições.

Em outras palavras, através da obrigatoriedade de um parâmetro, eu criei uma experiência ruim no consumo da minha API.

E por último, pense sempre em casos de uso da sua API quando for criar um novo recurso, isso irá facilitar bastante na definição dos parâmetros e dos metadados.

Políticas em APIs RESTful

Esse é um assunto muito importante, pois normalmente uma API é desenvolvida para ser uma interface pública de comunicação.

Como tudo na vida, se você disponibiliza algo liberado e sem regras, você pode ter problemas.

Nesse sentido, se você conhecer bem sobre políticas de APIs, você conseguirá manter tudo sob controle.

Além disso, existem outros benefícios de utilização dessa técnica, como maior segurança e escalabilidade, pois você praticamente obriga o consumidor da sua API a utilizar sua infraestrutura de forma mais consciente. Agora vamos entender mais sobre isso na prática:

Throttling e Rate limiting

Esses são os termos mais utilizados quando falamos de políticas em APIs, e muitas vezes throttling e rate limiting podem ser utilizados como sinônimos, mas tecnicamente eles não são.

Em linhas teóricas, rate limit e throttling são complementares mas não tem as mesmas funções.

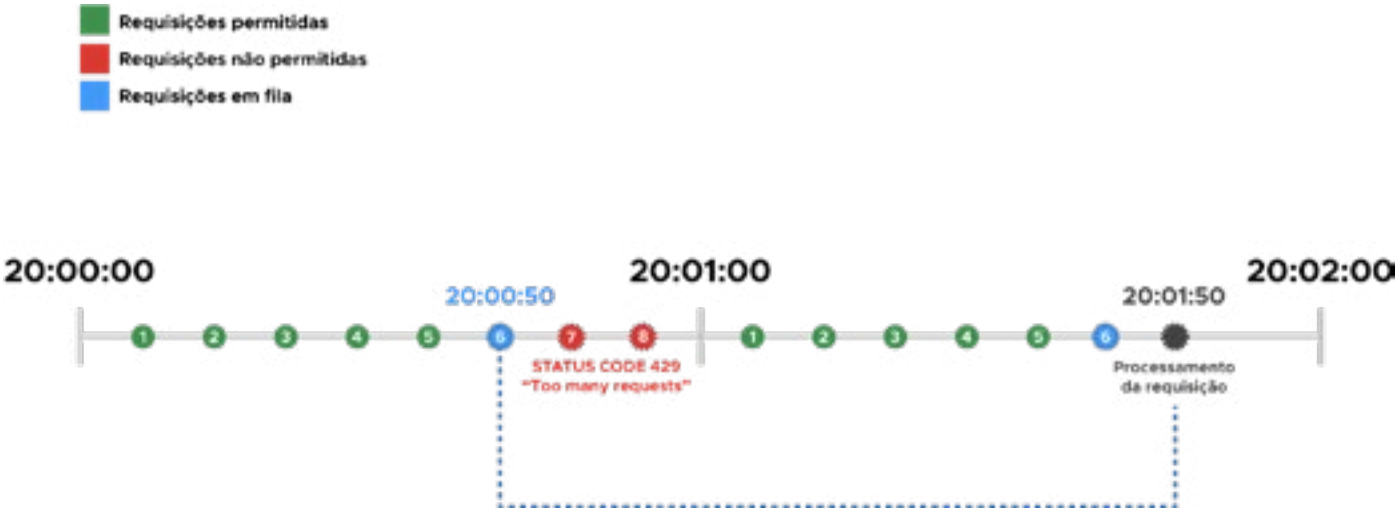
Rate-limiting é a configuração do limite de requisições aceitas pela API em uma determinada janela de tempo, e que pode ser em segundos, horas, dias, etc. Por exemplo: 100 requisições a cada 1 hora.

Throttling por sua vez é a configuração da fila de requisições excedidas para processamento em uma janela de tempo posterior, ou seja, é o tempo de delay do processamento da requisição após o client exceder os parâmetros de rate limit. Por exemplo: 2 retentativas com delay de 1 minuto.

Ambos podem ser configurados de forma granular, como por exemplo: por client, por recurso, pela API, etc.

Em resumo, rate limiting protege sua API de um alto consumo de requisições, enquanto o throttling prepara sua API para cenários de picos de acesso. Usando o exemplo da API dos capítulos anteriores, imaginem que eu queira configurar o meu rate-limit para 5 requisições por minuto, e um throttling de 1 retentativa com 1 minuto de delay.

Caso o client faça uma requisição às 20:00, nossa API deveria se comportar da seguinte forma:



Notem que ao chegar na sétima requisição, então a API não deverá permitir mais requisições, e retornará o HTTP status code 429 - Too Many Requests.

API Quota

Em uma visão comercial de uso da API e com um período maior de renovação das cotas de consumo, o termo API quota é bastante utilizado.

Por exemplo, a API Quota do cliente X é 5 mil requisições por mês.

Notem que agora estou falando de um período mensal, e não mais minutos, ou horas, que são períodos bem mais curtos.

Isso normalmente é utilizado quando o consumo da API é cobrado do client, pois facilita a visualização do consumo na fatura (É quase como uma conta de celular.)

É importante ressaltar que as API quotas podem ser combinadas com o rate-limiting, pois você pode limitar um client de consumir 5 mil requisições em um mês via API Quota e ao mesmo tempo limitar um client de consumir 100 requisições por minuto via Rate Limit, notem que os períodos não batem, pois o client não irá sempre consumir 100 requisições por minuto, então no fundo as técnicas não tem dependência direta.

API Burst (Plus)

Se você tem uma infraestrutura ociosa, e quer disponibilizar temporariamente mais performance de processamento e requisições para um client específico, então você pode usar a técnica de API Burst.

Por exemplo, sua configuração de rate-limit é de 10 requisições por minuto, porém um client envia 20 requisições de uma vez, porém sua API está com capacidade ociosa no servidor, então você processa todas as requisições em alta performance e retorna ao client.

Cada linguagem pode ter uma implementação diferente das técnicas, então recomenda-se utilizar frameworks existentes ou um API Management.

Performance em APIs RESTful

Eu sempre comento por aí que desenhar a arquitetura de uma solução é uma arte. E por mais que existam diversos patterns e técnicas, um projeto real tem inúmeras variáveis, por isso, não é algo tão trivial quanto parece.

Um dos assuntos amplamente discutidos em arquitetura é performance, pois influencia diretamente na utilização de recursos de servidores, e consequentemente, no seu nível de escalabilidade.

Nesse capítulo, vou esclarecer um pouco mais esse tema e ensinar como ter mais performance em APIs RESTful.

O que é cache

Cache é uma estrutura computacional de armazenamento focada em manter cópias de dados que são acessados com frequência.

O propósito do cache é acelerar a busca de dados que são muito utilizados e poupar a utilização de recursos de um servidor. Com o cache, você tem os seguintes benefícios na sua API:

- Redução da latência de rede
- Redução de carga de processamento dos servidores
- Otimização de tempo de resposta ao client

Os caches são divididos em duas grandes categorias:

Shared cache — Armazena respostas de servidores para reutilização entre diversos usuários.

Private cache — Armazena respostas de servidores para utilização de um usuário único.

Esses caches podem ser implementados através do browser, servidor proxy, gateway, CDN, proxy reverso, ou load balancer de servidores WEB.

Cache no client

Se você leu até aqui, já sabe que uma API RESTful utiliza o protocolo HTTP para comunicação entre cliente e servidor.

Dado esse cenário, o protocolo HTTP tem suas especificações (RFC) para utilização de políticas de cache.

Todas elas são especificadas no header de uma requisição, agora vamos entender quais são as políticas disponíveis. Aqui não vou citar a especificação do HTTP 1.0, porque os métodos dessa versão estão depreciados.

Controle de cache

O header cache-control é recomendado para declaração de mecanismos de cache. As principais diretivas disponíveis são:

No caching

Com a diretiva "no-store", você estará especificando para que nem na requisição do client, e nem na resposta do server seja utilizado cache. Em outras palavras, toda vez que uma requisição for feita, o client fará o download da resposta do server.

```
Cache-Control: no-store
```

Cache but revalidate

Com a diretiva "no-cache", o browser irá enviar uma requisição ao servidor para validação desses dados. A ideia aqui é economizar largura de banda, porém com a certeza de que aquele cache está atualizado, ou seja, caso o servidor responda ao browser que o cache está atualizado, então o browser traz os dados do cache, e não faz o download dos dados do servidor novamente.

```
Cache-Control: no-cache
```

Private and public caches

A diretiva "public" permite que os dados de resposta do servidor sejam armazenados em um cache compartilhado. Enquanto a diretiva "private" permite que os dados de resposta do servidor sejam armazenados no browser para utilização de apenas um usuário.

Expiration

A diretiva “max-age” é utilizada para setar o máximo de tempo (segundos) em que os dados ficarão disponíveis em cache.

Essa diretiva diz implicitamente ao browser que ele pode armazenar a página em cache, mas deve validar novamente com o servidor se a duração máxima for excedida.

```
Cache-Control: max-age=30
```

Validation

A diretiva “must-revalidate” obriga que o browser sempre revalide os dados com o servidor antes de utilizar o cache, ou seja, nesse caso, o cache continua sendo utilizado normalmente, entretando ele é revalidado a cada nova requisição.

```
Cache-Control: must-revalidate
```

Entity Tags

O header ETag é um dos mais importantes quando falamos de cache.

Ele é utilizado como resposta do server para especificar uma versão dos dados que estão sendo retornados.

Nesse sentido, ele serve como uma chave única consistente, e possibilita um fluxo inteligente de consumo dos dados em cache pelo client.

O fluxo fica da seguinte forma:

1. O client faz uma requisição de GET

2. O server envia os dados e uma chave de ETag, que é armazenada em um storage local do client

3. Client faz uma requisição com o valor da ETag no header “If-None-Match”.

4. O server valida se aquele dado foi modificado, e caso ele não tenha sofrido alterações, o server retorna o HTTP status code 304 — Not modified.

Cache no Server

No tópico anterior entendemos como são as instruções de cache a nível de client HTTP.

Portanto, entendemos como manipular o gerenciamento de cache via frontend, mas agora precisamos entender como isso funciona na camada de server.

Então vamos entender quais são as principais técnicas disponíveis para quem vai desenvolver uma API RESTful.

Load Balancing

Em cenários que sua API tem uma quantidade de requisições alta para apenas um servidor, então você utiliza essa técnica para distribuir a capacidade de processamento das requisições e tráfego de rede, com isso você torna sua API mais escalável e segura, pois se um servidor falhar, você terá outros funcionando. O load balancing por sua vez não tem uma funcionalidade default de cache, mas tem uma funcionalidade de armazenamento de sessão.

Como sabemos, o HTTP é stateless, ou seja, não armazena a sessão, porém quando é utilizado um load balancing, para fazer a melhor distribuição das requisições, ele faz o armazenamento da sessão do client, e por isso você pode utilizar como uma espécie de cache.

Existem diversos algoritmos de load balancing, e as principais implementações são feitas através de configurações do próprio Web server, como: Nginx, Apache, IIS.

Proxy Reverso

Um proxy reverso é basicamente uma interface pública da sua API, ou seja, ele funciona como um intermediador de todas as requisições externas.

As principais vantagens de utilização de proxy reverso são:

Segurança — Funcionalidades para proteção de ataques DDoS e configurações de certificados SSL.

Performance — Funcionalidades nativas de compressão dos dados trafegados e cache, pois como ele atua como intermediador das requisições, ele consegue tratar os dados de forma inteligente antes de retornar ao client. Veja na imagem a seguir a utilização de um proxy reverso com load balancer:

As implementações normalmente também são feitas através de configurações do próprio Web server, como: Nginx, Apache, IIS.

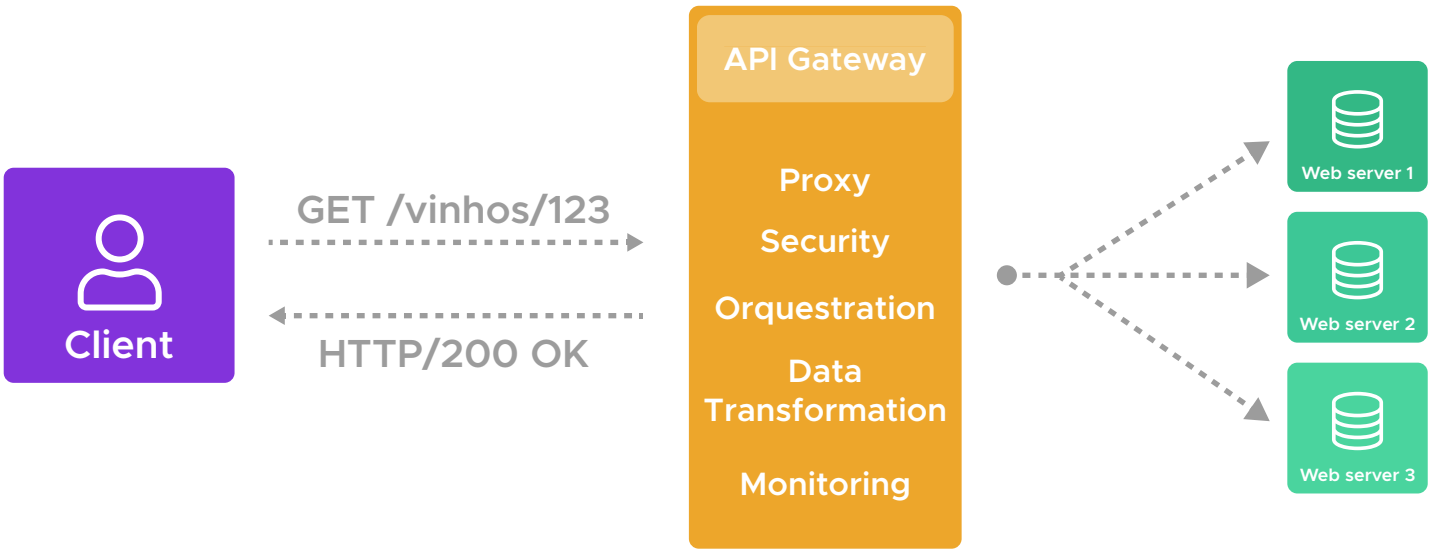
Gateway

Pense num proxy reverso com mais funcionalidades, pensou? Então esse é o Gateway... Ele tem todas as funcionalidades de um proxy reverso, e muito mais.

Na prática além de ser a interface de acesso a sua API para a internet, o gateway normalmente traz funcionalidades interessantes como: roteamento, monitoramento, autenticação e autorização, transformação de dados, segurança avançada por políticas, etc.

Ou seja, ele é o padrão de arquitetura e a técnica de utilização de cache mais indicada para uma API RESTful.

Nesse caso, ao invés de utilizar o web server, você utiliza plataformas de API Management, como LinkApi, por exemplo. Veja na imagem a seguir:



Um API Gateway já traz toda inteligência de cache embarcada, pois utiliza a própria funcionalidade de cache do proxy reverso, e para questões mais avançadas ele utiliza outros mecanismos de armazenamento de dados de alta performance, como Redis e Memcached, por exemplo.

CDN

CDN é um conceito muito difundido no mundo Web, e quer dizer Content Delivery Network — Rede de Distribuição de Conteúdo. Ela funciona como uma rede de servidores que mantem réplicas de conteúdos Web e fazem uma distribuição mais otimizada desses conteúdos.

A grande "sacada" desse modelo é que as réplicas são feitas em geolocalizações estratégicas, e por isso, a CDN consegue redirecionar as requisições do client para um servidor mais próximo, reduzindo significamente a latência de rede.

Como a latência de rede é reduzida, o tempo de resposta da requisição também é reduzido, portanto é uma ótima alternativa para uma API de alta performance que tem consumo em diferentes geolocalizações.

Essa técnica pode ser utilizada para Web no geral, ou seja, não é restrita apenas para o mundo de APIs. Os principais players de CDN são: Akamai e CloudFront.

Compressão de dados [Bônus]

O padrão REST permite diversos formatos de dados como XML, JSON, HTML, etc. Em todos eles, é possível comprimir a mensagem, fazendo com que o servidor trafegue menos dados, e gerando uma melhoria de performance na API. Para isso, é necessário a utilização de alguns headers na requisição:

Accept-Encoding

Como client, o header da requisição ficaria da seguinte forma (sendo que o Gzip é o formato padrão de compressão de dados):

```
Accept-Encoding: gzip,compress
```

Content-Encoding

No cenário em que o servidor tem a implementação da compressão de dados, além de obviamente compactar os dados para retornar ao client de forma otimizada, ele também retorna um indicativo desse processo através do header Content-Encoding, ou seja, o client deve receber o seguinte resultado:

```
200 OK
Content-Type: text/html
Content-Encoding:gzip
```

Algumas considerações importantes:

Se o Accept-Encoding estiver preenchido de forma não reconhecida pelo servidor, ele deverá retornar o status code 406 — Not Acceptable.

Se for um formato conhecido, porém o servidor não tiver implementado, então ele deverá retornar o status code 415 — Unsupported Media Type.

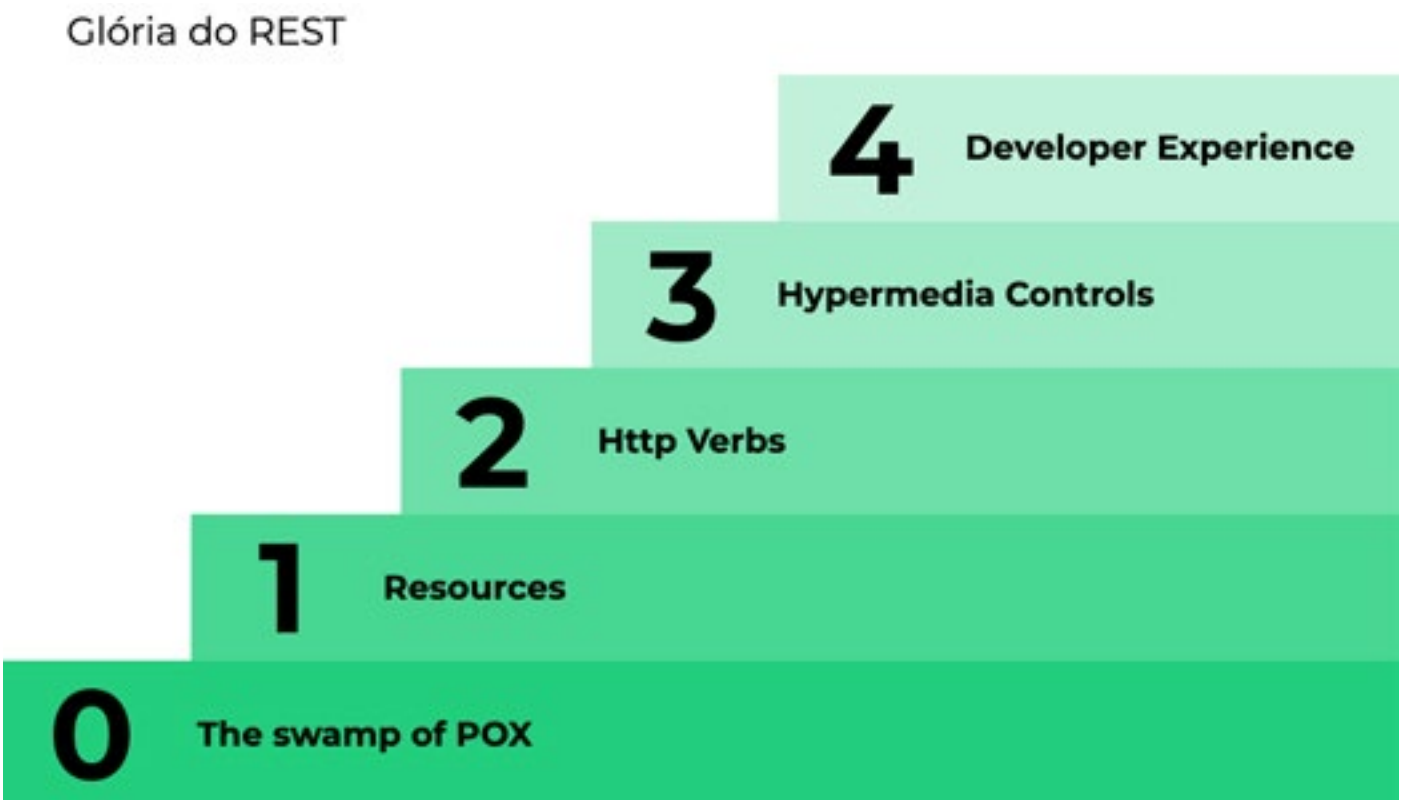
Um fato interessante é que a maioria dos browsers automaticamente fazem a requisição utilizando essas técnicas de compressão de dados.

O nível supremo em APIs RESTful

Nesse capítulo, vamos entender como chegar ao nível supremo de experiência em APIs RESTful.

A ideia agora é exemplificar o que não fazer e o que fazer na sua API.

Para isso, vamos utilizar um framework chamado *Richardson Maturity Model* com uma pequena modificação, que é a inclusão do nível 4.



Nível 1

Esse nível é tão básico que nem vou perder muito tempo nele.

Nesse primeiro nível você já está sabendo utilizar os recursos de forma individual, ou seja, cada URL com uma responsabilidade e um objeto específico, porém você ainda não sabe como utilizar os métodos HTTP e outras funções mais avançadas.

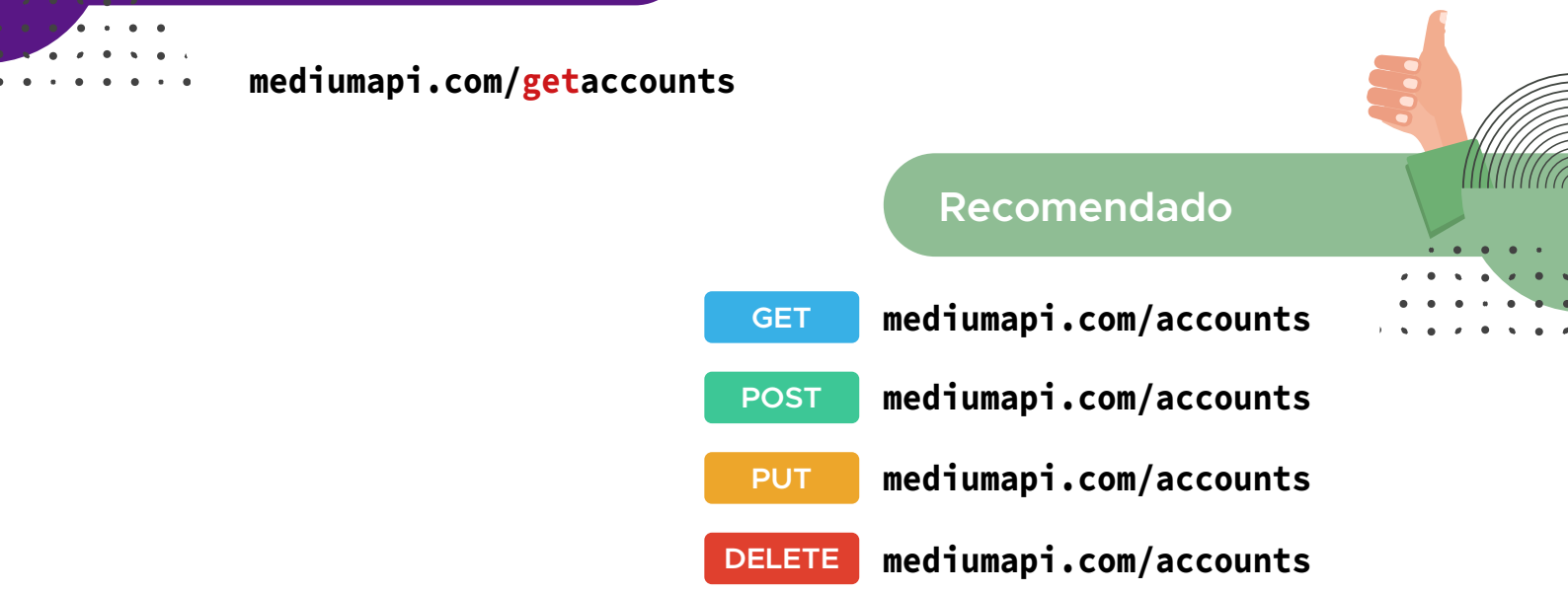
Em outras palavras, você só está utilizando o REST como mecanismo de transporte, e não se importa com boas práticas e coisas do tipo.

Nível 2

No nível 2 você já entende um pouco mais sobre REST, e por isso já se preocupa em utilizar a especificação do HTTP através de métodos, headers, códigos de status, e paginação.

Não use verbos na URL

O protocolo HTTP já tem seus próprios métodos, então você não precisa colocar verbos na URL, não é mesmo?



Use Content-Type no header

Eu sei que ainda é muito comum utilizar o formato dos dados na URL via extensão, mas existe um header específico para isso.

Não recomendado

```
GET /entities/123/payable_accounts.json
```

Recomendado

```
GET /entities/123/payable_accounts
Accept: application/json
Content-Type: application/json
```

Tenha uma hierarquia simples na URL

Se você fizer uma navegação com mais de três níveis de hierarquia, isso pode complicar a vida de quem utiliza a sua API. Portanto, se tiver um cenário muito complexo de navegação de objetos, use a combinação com Query String.

Não recomendado

```
/accounts/00928376/transaction/8738903
/zip/6500/city/dublin/state/ca
```

Recomendado

```
/accounts/00928376/action/8738903
?zip=6500&city=dublin&state=ca
```

Utilize status code HTTP pra tudo

Essa eu não precisava nem falar, mas vamos lá... Se você quer proporcionar uma comunicação padronizada e que utiliza as melhores práticas, então você precisa utilizar os códigos de status padrões do protocolo HTTP.

Paginação e filtros

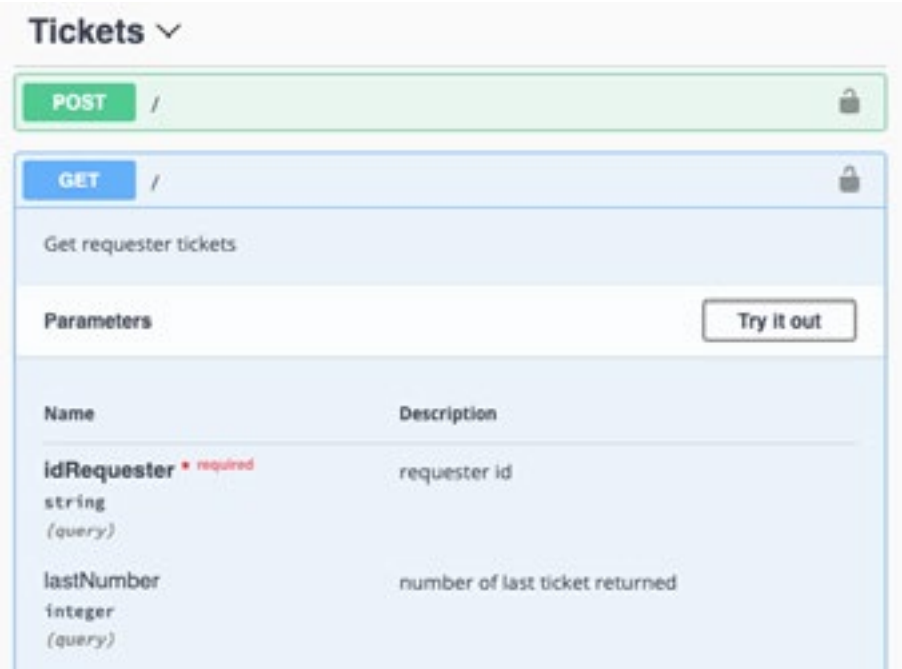
Para proporcionar uma boa experiência em termos de performance e busca de dados, então você precisa tomar os seguintes cuidados:

1. Primeiro de tudo, implementar alguma técnica de paginação, se ainda não sabe como, eu expliquei sobre isso em um dos capítulos deste ebook.

2. Sempre especificar na documentação em qual página inicia uma busca, exemplo: 0 ou 1, pois se você não especifica, o consumidor pode utilizar qualquer uma e obter o mesmo resultado, confuso não?

3. Disponibilizar ranges de datas no filtro, como por exemplo: CreatedBefore, CreatedAfter.

4. Não colocar filtros obrigatórios que anulam o objetivo da busca, exemplo imagem abaixo:



Notem que a API acima é para buscar tickets, mas como parâmetro obrigatório ela solicita o Requester, o que anula diretamente o objetivo principal do endpoint e prejudica a experiência do consumidor da API.

Nível 3

No nível 3, você já está implementando questões mais avançadas como versionamento, rastreabilidade e HATEOAS.

Outra característica desse nível é que você já tem uma certa preocupação com a experiência do developer que utiliza sua API, e por isso, faz o uso de padrões universais, e cria objetos pensando em legibilidade.

Versionamento

Existem diversas técnicas de versionamento de APIs, e para exemplificar o assunto, abaixo quero comparar dois modelos distintos de versionamento:

Não recomendado

API Salesforce

```
yourInstance.salesforce.com/services/data/
[{ "version" : "20.0", "label" : "Winter '11",
"url" : "/services/data/v20.0"},
{"version" : "21.0", "label" : "Spring '11",
"url" : "/services/data/v21.0" }]
```

Recomendado

API Stripe

```
api.stripe.com/v1/charges
-u sk_test_4eC39HqLyjWDarjtT1zdp7dc:
-H "Stripe-Version: 2019-03-14"
```

No primeiro exemplo (API Salesforce), podemos notar que tanto a versão quanto a subversão ficam na URL, o que dificulta bastante a navegação da API. Além disso, podemos notar uma quantidade enorme de versões, como por exemplo versão 21. Isso de certa forma pode até "assustar" o desenvolvedor, que ao começar a desenvolver algo, ele nota que as mudanças são constantes.

No segundo exemplo (API Stripe), a versão fica na URL, porém a subversão fica no header. Nesse modelo, a explorabilidade da API se mantém intacta, além de parecer muito mais organizada, e o mais importante, proporciona uma melhor experiência ao developer.

Objetos legíveis e limpos

A legibilidade, ou seja, a facilidade de leitura de um objeto pode parecer individual, mas na prática não é. Quanto mais simples for um objeto, mais fácil é entendê-lo, portanto não utilizem estruturas desnecessárias, como:

Não recomendado

GET **/entities/:entity_id/receivable_accounts ?per_page=15&page=3**

```
{
  "receivable_account": {
  "id": 1,
  "entity_id": 1,
  "status": 1,
  "status_name": "unreceived",
  "due_date": "2011-09-10",
  "occurred_at": null,
  "amount": "100.0",
  "ticket_amount": null,
  "interest_amount": null,
  "discount_amount": null,
  "nominal_amount": null
 }
```

Notem que no exemplo acima, eu já estou fazendo uma requisição ao recurso receivable_account, portanto quando eu retorno o resultado, eu não preciso repetir ele novamente na estrutura de retorno, pois isso só dificulta o utilização do consumir da API.

Gere rastros através de logs e timestamping

Tão importante quanto você permitir uma boa experiência em manipulação de dados na sua API, é você manter uma governança. Para isso use padrões de rastreabilidade que permita você saber quem fez a criação ou alteração de um dado, como por exemplo:

Recomendado

```
{
  "createdAt": 1501869101896,
  "createdBy": "Thiago",
  "updatedAt": 1501869101896,
  "updatedBy": "Thiago"
}
```

Use sempre o padrão ISO 8601 para data e hora

Esse é um assunto chato e que quase nenhum desenvolvedor gosta. Formatação de data é sempre um “caos” no mundo de desenvolvimento, então não reinvente a roda, e use padrões para tratar isso. Utilize sempre o ISO 8601 em UTC.

Recomendado

```
{"createdAt": "2019-07-08T18:02:24.343Z"}
```

Use o HATEOAS — Hypermedia As The Engine Of Application State

Um dos pontos altos do padrão REST é permitir HATEOAS. Para quem não sabe o que é, é basicamente uma forma de retornar em um recurso links navegáveis para objetos relacionados.

Na prática, isso facilita muito o lado do client, pois pense no desenvolvimento frontend. Com uma API que tem HATEOAS você conseguiria fazer um binding dinâmico de um botão através do link. Veja o exemplo:

Recomendado

GET `https://api.github.com/users/codemazeblog`

```
{
  "login": "CodeMazeBlog",
  "id": 29179238,
  "avatar_url": "https://avatars0.
githubusercontent.com/u/29179238?v=4",
  "url": "https://api.github.com/users/
CodeMazeBlog",
  "html_url": "https://github.com/
CodeMazeBlog",
  "followers_url": "https://api.github.
com/users/CodeMazeBlog/followers",
  ...
}
```

Nível 4

O nível 4 é o "supra sumo" de uma API RESTful, e nesse nível, além de entregar uma API funcionando e estável, você estará entregando uma ótima experiência pro developer que irá consumir a sua API.

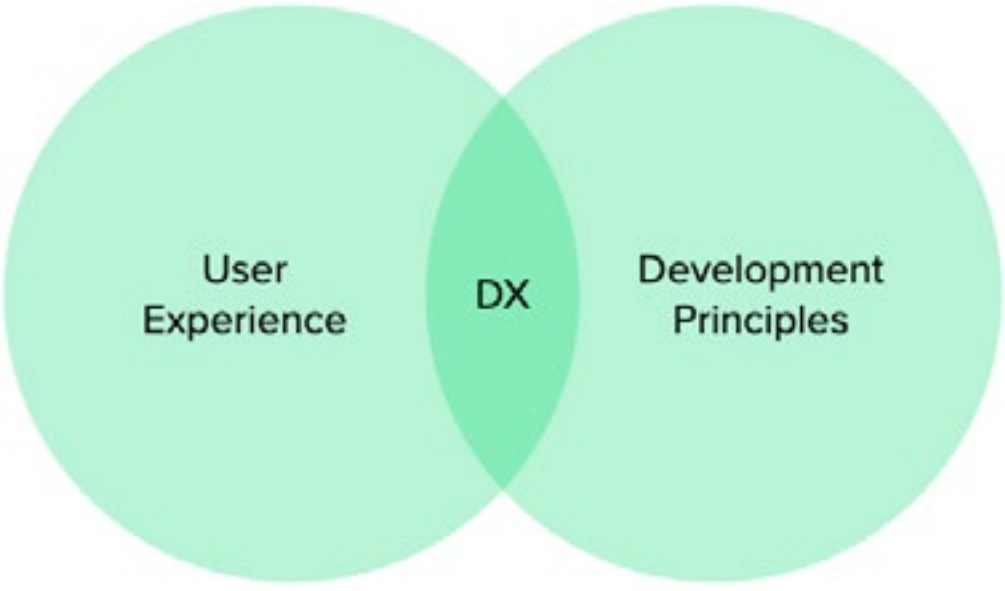
Pense que aqui você já implementou o protocolo HTTP e os padrões REST nas suas melhores práticas, e que agora o desafio está nos detalhes que impactam na experiência de navegação e consumo.

Use conceitos de Developer Experience

Você já deve ter ouvido falar bastante sobre UX ou experiência do usuário, certo? A disciplina de UX tem basicamente a responsabilidade de garantir uma boa experiência do usuário em relação a um produto ou serviço.

Porém, isso normalmente é esquecido quando pensamos no desenvolvedor como usuário. Talvez, por uma falsa impressão de que nós desenvolvedores gostamos de tela preta ou algo do tipo.

O fato é que quando falamos de uma API, também criamos uma experiência a partir dela, e para que a experiência seja positiva devemos aplicar conceitos de DX ou Developer Experience. O DX é uma junção entre princípios de desenvolvimento com princípios de User Experience:



Como esse tema é extenso, vou compartilhar apenas as preocupações básicas de quem aplica esse conceito:

- **Task-Invocation Ratio**: Quantas chamadas na API são necessárias para um desenvolvedor atingir seu objetivo?
- **Estrutura:** Qual a estrutura de um request e de um response? Quão profundo tecnicamente o desenvolvedor precisa navegar para conseguir chegar no dado que ele quer utilizar?
- **Navegação:** Quão complexo é navegar entre recursos na sua API?

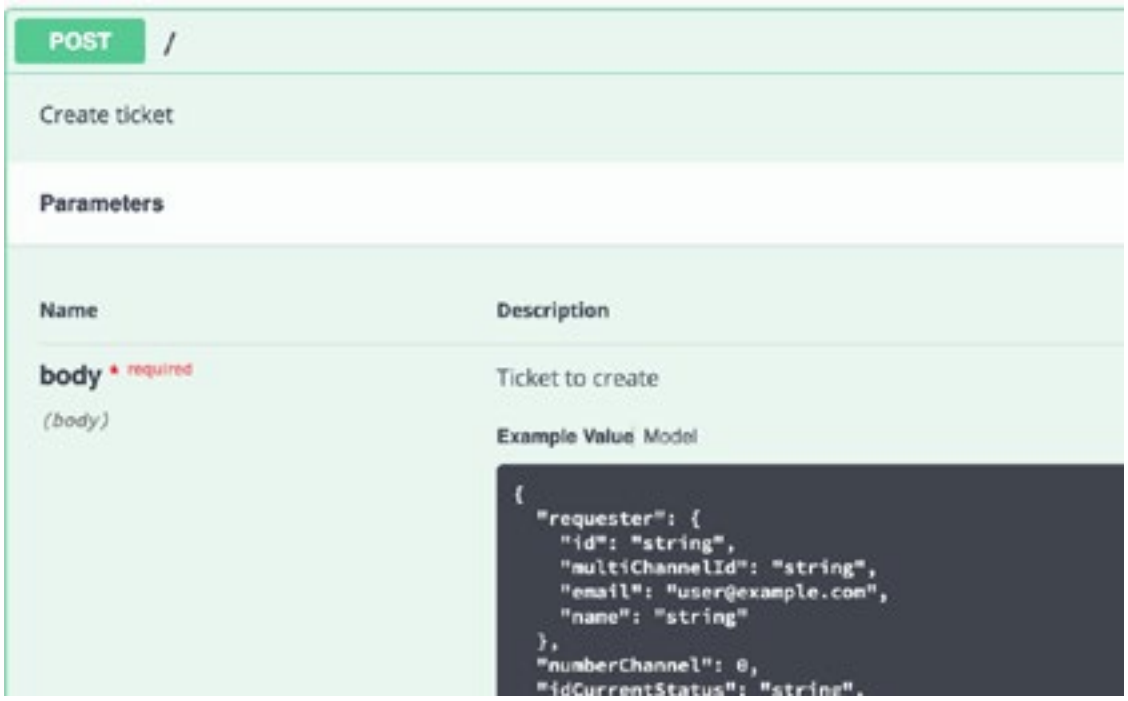
Stack Size: Quantas ferramentas e bibliotecas o desenvolver precisa instalar para usar sua API?

- **Tempo do primeiro request:** Em quanto tempo um desenvolvedor consegue fazer o primeiro request na sua API?
- **Error Handling:** Qual a natureza dos possíveis erros? Quão difícil é resolver?

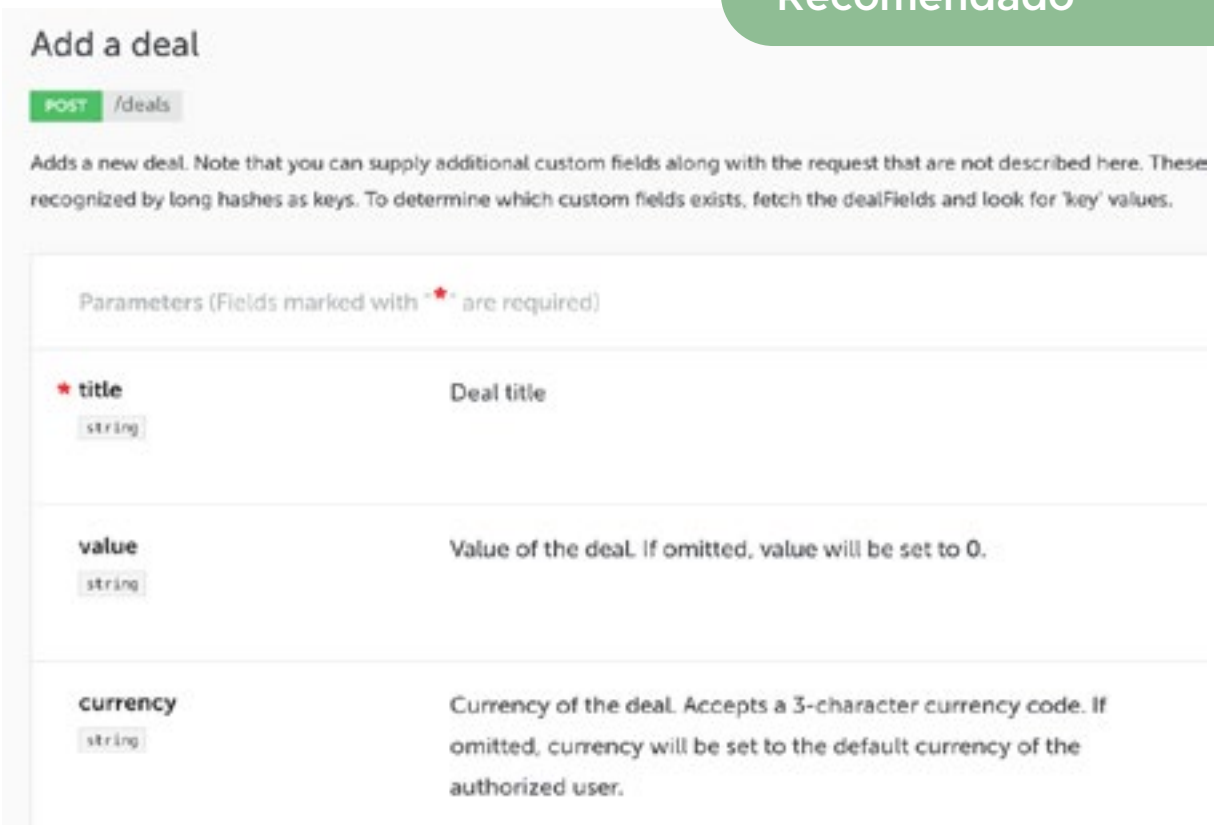
Documente as propriedades dos seus recursos

Ter uma boa documentação é básico quando falamos do nível 4, e não sei por qual motivo, mas vejo muitas documentações que partem do princípio de que os desenvolvedores conseguem adivinhar o que significa cada propriedade, por isso documente as propriedades do seu recurso, ou seja, descreva com detalhes a estrutura de dados de um request e um response. Veja a diferença na prática:

Não recomendado



Recomendado



Cuidado com limites de requisições

Se a sua API não fornece algo inteligente para tratar dados em massa (bulk data) tome bastante cuidado com o limite de requisições na sua API.

Eu sei que é muito importante ter políticas para ter uma infraestrutura mais saudável e escalável, mas isso não pode prejudicar a experiência de um desenvolvedor. Por isso, você não pode exagerar nesse limite.

Por exemplo: Eu já vi diversas APIs com rate-limit de 20 requisições por minuto e com uma paginação máxima de 100 itens por vez.

Agora imagine você consumindo uma API com essas limitações e tendo que construir um BI. Muito provavelmente, você irá se irritar com esse limite tão baixo na hora de criar um mecanismo de paginação.

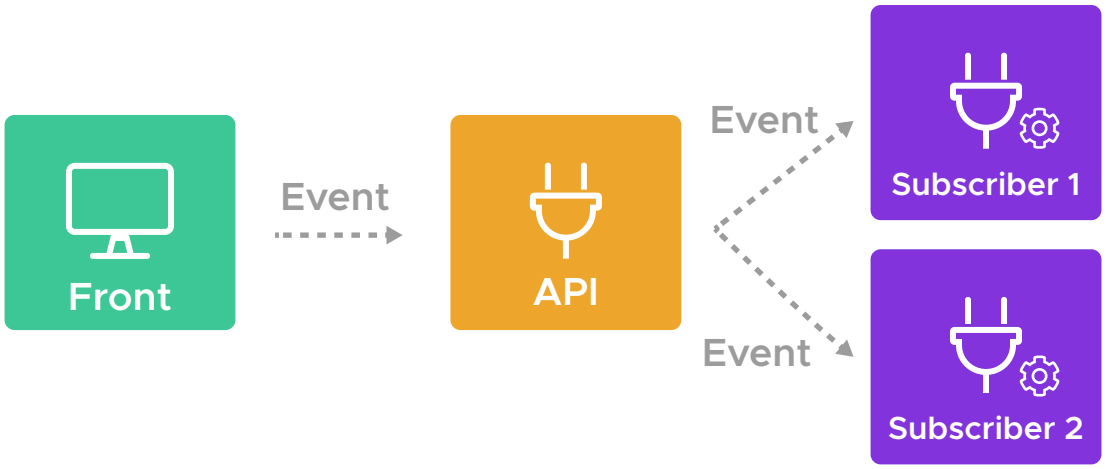
Minha recomendação é de pelo menos um rate-limit de 40 requisições por minuto, mas isso é um feeling do que tenho visto como boa experiência no mercado, e isso pode variar de acordo com o tipo da API.

Tenha uma API event-driven

Esse aqui é bem útil para quem quer ter uma infraestrutura eficiente no tráfego de dados, além de proporcionar uma boa experiência ao client.

Normalmente quando você vai consumir uma API e precisa dos dados atualizados, você tem que ficar fazendo requisições (pooling) o tempo todo para entender se existem dados novos ou não no servidor. Fazendo um paralelo, é quase como perguntar para um vendedor da sua empresa o tempo todo: “Tem venda nova hoje?”

Agora pense na hipótese de que ao invés de você perguntar o tempo todo ao vendedor, quando ele fizer uma nova venda, ele mesmo vai te notificar proativamente. Funciona exatamente assim uma API event-driven, ela tem subscribers, ou seja, clients que se inscrevem para serem notificados, e em caso de novos eventos, ela notifica automaticamente todos os clients inscritos. Para ter uma API event-driven você deve implementar webhooks na sua API.



Mensagens de erro dev-friendly

Existem diferentes abordagens nesse assunto, mas aqui o objetivo é retornar mensagens friendly de erros na sua API, diminuindo a curva de diagnóstico de problemas. Para isso, você apenas precisa customizar as respostas da sua API com uma estrutura de erros mais organizada e detalhada. Notem que interessante essa estrutura de response, além de retornar o status code do HTTP 404, ela te dá um código interno do erro para você navegar na doc, e um link detalhado do erro, com as possíveis soluções. De fato, isso impacta positivamente a experiência do developer que irá consumir a API.

Recomendado

API Twilio

GET **https://api.twilio.com/2010-04-01/Accounts/1234/IncomingPhoneNumbers/1234**

```
Response: Status Code 404
{
  "RestException": {
    "Code": "20404",
    "Message": "The requested resource /2010-04-01/
Accounts/1234/IncomingPhoneNumbers/1234 was not found",
    "MoreInfo": "https://www.twilio.com/docs/errors/20404",
    "Status": "404"
  }
}
```

Ambiente de desenvolvimento (sandbox)

Se você quer dar possibilidade de testarem rapidamente algo na sua API, então você deve disponibilizar um ambiente para desenvolvimento e testes, mais chamado de sandbox. Um ambiente sandbox simula basicamente um ambiente de produção, entretanto, por não ser o ambiente de produção, o desenvolvedor consegue fazer testes mais avançados, e operações sem receios de errar.

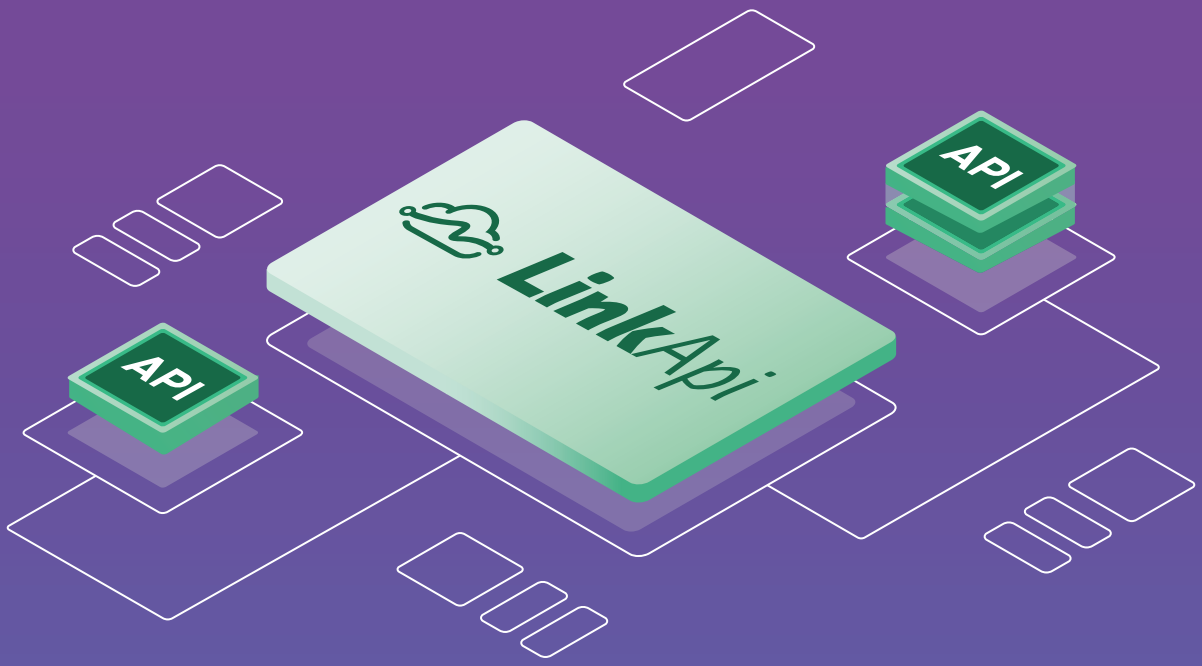
Conclusão

Espero que tenham aprendido bastante com esse conteúdo, e que com isso, consigam me ajudar a colocar o Brasil na lista dos melhores ecossistemas de APIs do mundo.

E vamos juntos, rumo a um mundo melhor de APIs!

Thiago Lima é CEO e fundador da LinkApi, plataforma que possibilita empresas desenvolverem, monitorarem e distribuírem integrações entre diferentes sistemas e APIs. Thiago começou a programar aos 12 anos. Ainda com 17 anos, fundou seu primeiro negócio. A LinkApi nasceu em 2017, e o trabalho de Thiago como CEO inclui desenvolvimento de produto, relacionamento com investidores e formação de pessoas – nesse pilar ele ministra aulas de filosofia e tecnologia para os colaboradores, com o objetivo de desenvolver as pessoas.

Nós queremos fomentar a transformação digital e a inovação aberta



Para as empresas que já nasceram na era digital e estão moldando o novo mercado e para as empresas tradicionais e mais antigas que estão agora passando por essa mudança, o LinkApi funciona como o impulsionador dessas estratégias.

O LinkApi foi desenvolvido a partir de insatisfações que foram sentidas tanto pelas pessoas que desenvolveram a nossa plataforma desde o início, e pelas pessoas que trabalham com APIs e integrações dentro de empresas de vários segmentos diferentes.

O mindset de manter ideias e novas iniciativas dentro de casa já é passado, e tudo nos mostra que a colaboração é o caminho para relações win-win, entre empresas gigantes e contribuidores individuais ou pequenos grupos.

www.linkapi.solutions