# Por que minha REST API é uma droga?

## O que é uma API?

Interface de programação de aplicações (Application programming interface) é um conjunto de definições e protocolos para construir e integrar software de aplicação.

Em outras palavras, se você quiser interagir com um computador ou sistema para recuperar informações ou realizar uma função, uma API ajuda você a comunicar o que deseja a esse sistema para que ele possa entender e atender à solicitação.

## Tipos de API

### Bibliotecas e frameworks

A interface para uma biblioteca de software é um tipo de API. A API descreve e prescreve o "comportamento esperado" (uma especificação) enquanto a biblioteca é uma "implementação real" desse conjunto de regras.

A seguir, alguns exemplos de frameworks e bibliotecas:

| **Linguagem** | **Framework/ Biblioteca** |
|---|---|
| JavaScript/Node.js | Express.js |
| | NestJS |
| Python | Flask |
| | Django Rest Framework (DRF) |
| Java | Spring Boot |
| Go (Golang) | Gin |
| | Fiber |
| PHP | Laravel |
| C# | `ASP.NET Core` |
| Rust | Rocket |
| | Actix Web |

### Sistemas operacionais

Uma API pode especificar a interface entre uma aplicação e o sistema operacional.

- **POSIX:** fornece um conjunto de especificações de API comuns que visam permitir que uma aplicação escrita para um sistema operacional compatível com POSIX seja compilada para outro sistema operacional compatível com POSIX.
- **Windows API (Win32):** API designada para que aplicativos mais antigos possam ser executados em versões mais novas do Windows usando uma configuração específica do executável chamada "Modo de Compatibilidade".

### Web APIs

APIs Web são um serviço acessado a partir de dispositivos clientes para um servidor web usando o Protocolo de Transferência de Hipertexto (HTTP). Dispositivos clientes enviam uma solicitação na forma de uma requisição HTTP e recebem uma mensagem de resposta.

## Arquitetura de Web APIs

| Estilo de API | Prós | Contras | Casos de Uso |
|---|---|---|---|
| REST | Simples, flexível, escalável, amigável para a web | Sem contrato claro, sem consultas complexas, tratamento de erros fraco | Aplicações orientadas a dados, serviços web, cache, sem estado |
| SOAP | Contrato claro, consultas complexas, bom tratamento de erros | Complexo, verboso, não escalável, não amigável para a web | Aplicações transacionais, segurança, autenticação, interoperabilidade |
| GraphQL | Independente de linguagem, único ponto de extremidade, fortemente tipado, eficiência de dados | Complexo, difícil de usar, sem cache, tratamento de erros fraco | Modelos de dados complexos e dinâmicos, requisitos impulsionados pelo cliente, largura de banda e desempenho |
| gRPC | Independente de linguagem, contrato claro, consultas complexas, rápido e eficiente | Complexo, difícil de usar, não amigável para a web, inflexível | Modelos de dados complexos e dinâmicos, comunicação entre microsserviços, velocidade e eficiência |
| WebSocket | Comunicação rápida e eficiente, bidirecional e multiplexada, comunicação em tempo real e orientada a eventos | Não suportado por alguns navegadores ou proxies mais antigos, não seguro por padrão, não mantém estado | Troca de dados rápida e interativa, como chat, jogos ou streaming |
| Webhooks | Simples, fácil de usar, amigável para a web, escalável | Sem contrato claro, sem consultas complexas, tratamento de erros fraco | Modelos de dados simples e estáveis, notificações orientadas a eventos, desempenho e escalabilidade |

## REST

REST é uma maneira de usar padrões da web e endereços para trabalhar com dados em um servidor. Eles são populares, fáceis de implementar e usam métodos HTTP. A maioria dos serviços da web com os quais você interage diariamente, como Twitter ou YouTube, são alimentados por APIs Restful. Por exemplo, um cliente pode usar GET para obter dados, POST para criar novos dados, PUT para alterar dados ou DELETE para remover dados. Os dados geralmente estão em formatos como JSON ou XML. Alguns dos principais princípios do REST são:

1. **Stateless (Sem Estado):** Cada requisição feita pelo cliente para o servidor deve conter toda a informação necessária para entender e processar a requisição. Isso significa que cada requisição deve ser independente das requisições anteriores, ou seja, o servidor não deve manter nenhum estado da sessão do cliente entre requisições.
2. **Client-Server (Cliente-Servidor):** O cliente e o servidor devem ser separados, permitindo que eles evoluam independentemente. Isso promove a escalabilidade ao permitir que o cliente e o servidor sejam desenvolvidos e melhorados separadamente.

3. **Cacheable (Cacheável):** As respostas do servidor devem ser explicitamente indicadas como cacheáveis ou não-cacheáveis. Isso permite que as respostas sejam armazenadas em cache pelo cliente ou por intermediários, melhorando a eficiência e a escalabilidade do sistema.
4. **Layered System (Sistema em Camadas):** O sistema deve ser composto por camadas, onde cada componente (como um proxy ou gateway) só conhece a camada imediatamente inferior. Isso promove a escalabilidade ao permitir que os sistemas sejam expandidos em várias camadas, sem que cada componente conheça toda a complexidade do sistema.
5. **Uniform Interface (Interface Uniforme):** O sistema deve ter uma interface uniforme entre os componentes, o que simplifica e desacopla a arquitetura geral. Esta interface deve incluir recursos identificáveis, manipulação de recursos através de representações, mensagens auto-descritivas e hiperlinks para navegar entre recursos relacionados.

# Planejamento e Design

## Definição de recursos

Em REST, um recurso é um bloco de construção fundamental que representa um item de dados específico, como um usuário, um produto ou um pedido. Recursos são identificados por URIs (Uniform Resource Identifiers), que são endereços únicos que podem ser usados para acessar e manipular o recurso.

Em uma API RESTful, cada recurso é tratado como uma entidade separada que pode ser acessada e manipulada independentemente. Por exemplo, um recurso para um usuário pode ser representado pela URI "`/users/123`", onde "`123`" é o identificador de um usuário específico.

É importante observar que os recursos em REST não se limitam a dados armazenados em um banco de dados. Eles também podem representar outros recursos como imagens, vídeos ou até mesmo outros serviços da web. O princípio chave é que os recursos devem ser tratados como entidades autocontidas que podem ser manipuladas através da API REST.

Um recurso devidamente nomeado torna uma API simples de usar e intuitiva. A mesma API, quando implementada incorretamente, pode parecer complicada e ser desafiadora de usar e compreender.

### ▼ Use substantivos para representar recursos, não verbos

Sempre certifique-se de que seus URIs são nomeados com substantivos para especificar o recurso em vez de usar verbos. Os URIs não devem indicar operações CRUD (Criar, Ler, Atualizar, Excluir). Além disso, evite combinações de verbo-substantivo: hifenizado, snake_case, camelCase.

| Exemplos ruins | Exemplos bons |
|---|---|
| `http://api.example.com/v1/store/CreateItems/{item-id}` | `http://api.example.com/v1/store/items/{item-id}` |
| `http://api.example.com/v1/store/getEmployees/{emp-id}` | `http://api.example.com/v1/store/employees/{emp-id}` |
| `http://api.example.com/v1/store/update-prices/{price-id}` | `http://api.example.com/v1/store/prices/{price-id}` |
| `http://api.example.com/v1/store/deleteOrders/{order-id}` | `http://api.example.com/v1/store/orders/{order-id}` |

### ▼ Use substantivos no plural para recursos

Use o plural sempre que possível, a menos que sejam recursos únicos.

| Exemplos ruins | Exemplos bons |
| --- | --- |
| `http://api.example.com/v1/store/item/{item-id}` | `http://api.example.com/v1/store/items/{item-id}` |
| `http://api.example.com/v1/store/employee/{emp-id}/address` | `http://api.example.com/v1/store/employees/{emp-id}/address` |

## ▼ Use hífens (-) para melhorar a legibilidade dos URIs.

Não use sublinhados. Separar palavras com hífens será fácil para você e para outros interpretarem. É mais amigável ao usuário quando se trata de URIs segmentados com caminhos longos.

| Exemplos ruins | Exemplos bons |
| --- | --- |
| `http://api.example.com/v1/store/vendormanagement/{vendor-id}` | `http://api.example.com/v1/store/vendor-management/{vendor-id}` |
| `http://api.example.com/v1/store/itemmanagement/{item-id}/producttype` | `http://api.example.com/v1/store/item-management/{item-id}/product-type` |
| `http://api.example.com/v1/store/inventory_management` | `http://api.example.com/v1/store/inventory-management` |

## ▼ Use barras (/) para hierarquia, mas não use uma barra no final (/)

As barras são usadas para mostrar a hierarquia entre recursos individuais e coleções.

| Exemplo ruim | Exemplo bom |
| --- | --- |
| `http://api.example.com/v1/store/items/` | `http://api.example.com/v1/store/items` |

## ▼ Versione suas APIs

Sempre tente versionar suas APIs. Você pode fornecer um caminho de atualização sem fazer alterações fundamentais nas APIs existentes ao versionar suas APIs. Você também pode informar aos usuários que as versões atualizadas da API estão acessíveis nos seguintes URIs totalmente qualificados.

`http://api.example.com/v1/store/employees/{emp-id}`

A introdução de qualquer atualização importante que quebre a compatibilidade pode ser evitada com o seguinte /v2.

`http://api.example.com/v1/store/items/{item-id}`

`http://api.example.com/v2/store/employees/{emp-id}/address`

## ▼ Use o componente de consulta para filtrar coleções de URI

Você frequentemente encontrará requisitos que exigem que você classifique, filtre ou limite um grupo de recursos dependendo de um atributo de recurso específico. Em vez de criar APIs adicionais, habilite a classificação, filtragem e paginação na API de coleção de recursos e forneça os parâmetros de entrada como parâmetros de consulta para atender a esse requisito.

`http://api.example.com/v1/store/items?group=124`

`http://api.example.com/v1/store/employees?department=IT&region=USA`

## ▼ Aninhar Recursos para relacionamentos

Outro aspecto importante do design de API é aninhar recursos para refletir os relacionamentos entre entidades. Em APIs REST, o aninhamento deve refletir a hierarquia natural e os relacionamentos entre entidades, auxiliando os usuários a entenderem as relações entre

diferentes pontos de extremidade. No entanto, é importante limitar esse aninhamento a um nível para melhorar a clareza e evitar a complexidade.

| Sem Aninhamento | Com Aninhamento |
| --- | --- |
| Ter endpoints separados como `/api/books` e `/api/authors` pode não mostrar claramente a relação entre livros e seus autores. | `/api/authors/123/books` indica que você está acessando os livros relacionados ao autor com ID 123. Essa estrutura aninhada mostra claramente a relação entre autores e seus livros, tornando mais fácil para os usuários da API entender como acessar recursos relacionados. No entanto, lembre-se de manter o aninhamento em um nível sempre que possível para manter a simplicidade e evitar caminhos de endpoint excessivamente complexos. |

## Aproveitando os Métodos HTTP

Aproveitar os métodos HTTP é tão importante quanto nomear precisamente os pontos de extremidade. Os métodos HTTP são utilizados para realizar ações em recursos, cada método representando operações CRUD específicas.

### ▼ Operações CRUD com Métodos HTTP

Os principais métodos HTTP usados para operações CRUD são:

- **POST:** usado para criar novos recursos.
- **GET:** usado para recuperar uma representação de um recurso.
- **PUT:** usado para atualizar ou criar um recurso.
- **DELETE:** usado para excluir um recurso.
- **PATCH:** alterações parciais em um recurso.

## Uso Adequado dos Códigos de Status HTTP

Os códigos de status HTTP são cruciais para comunicar os resultados de uma chamada de API REST. Para melhorar o tratamento de erros, as APIs devem usar códigos de status apropriados para informar aos clientes sobre o sucesso ou falha de suas solicitações. Por exemplo, um código de status `200 - Sucesso` indica que a solicitação foi bem-sucedida, enquanto um código de status `400 - Solicitação Inválida` indica um erro baseado no cliente.

### ▼ Tratamento de Erros com Códigos de Status

Fornecer respostas de erro detalhadas usando códigos de status apropriados é uma maneira eficaz de comunicar-se com os usuários sobre o resultado de suas solicitações. Por exemplo, códigos de status na faixa `400-499` podem informar ao cliente sobre problemas como solicitações inválidas, falhas de autenticação e problemas de permissão, enquanto um código de status `500` indica um erro genérico do servidor.

## Filtragem e Ordenação com Parâmetros de Consulta

Os parâmetros de consulta permitem a filtragem de dados por critérios como intervalos de datas ou categorias, melhorando a recuperação de dados para os usuários. Além disso, a ordenação de dados é facilitada por meio de um parâmetro de consulta, especificando a ordem e o campo a serem classificados, como ascendente ou descendente.

### ▼ Exemplo de Filtragem:

Suponha que você tenha um endpoint da API para buscar pedidos. Para recuperar pedidos feitos dentro de um intervalo de data específico, a solicitação pode ser assim:

`/api/orders?startDate=2022-01-01&endDate=2022-01-31`

Aqui, startDate e endDate são parâmetros de consulta usados para filtrar pedidos por sua data de realização.

### ▼ Exemplo de Ordenação:

Para classificar esses pedidos por preço em ordem decrescente, a URL pode ser

`/api/orders?sortBy=price&order=desc`

Neste caso, sortBy especifica o campo a ser ordenado (price), e order especifica a direção da ordenação (descendente).

## Técnicas de Paginação

Ao lidar com grandes conjuntos de dados, implementar técnicas de paginação eficazes é crucial. Várias técnicas, incluindo paginação baseada em página, baseada em tempo e baseada em cursor, podem lidar efetivamente com grandes conjuntos de dados e garantir consistência.

### ▼ Exemplo de Paginação Baseada em Página:

Para paginação baseada em página, se você deseja recuperar a terceira página de uma lista de livros, com dez livros por página, a solicitação pode ser assim:

`/api/books?page=3&limit=10`

Aqui, page indica o número da página, e limit especifica o número de itens por página.

### ▼ Exemplo de Paginação Baseada em Tempo:

Para buscar artigos de notícias publicados após um determinado horário, a solicitação poderia ser:

`/api/articles?publishedAfter=2022-07-01T00:00:00Z`

Embora não seja uma técnica de paginação tradicional, é semelhante à paginação baseada em cursor, onde publishedAfter atua como um cursor para o tempo.

### ▼ Exemplo de Paginação Baseada em Cursor:

Para paginação baseada em cursor, se você estiver recuperando o próximo conjunto de atividades do usuário após um ID de atividade específico, a URL poderia ser

`/api/activities?cursor=abc123&limit=20`

Aqui, cursor é o identificador único (ID) da última atividade na busca anterior, e limit controla quantas atividades subsequentes retornar.

# Entendendo as necessidades do cliente

Ao projetar e desenvolver APIs Backend, é crucial não perder de vista o cliente final. É comum focarmos tanto na funcionalidade técnica que nos esquecemos de quem utilizará efetivamente a API. Para garantir que a estrutura seja funcional e atenda ao propósito original, é fundamental considerar como ela será utilizada e como isso impactará a experiência do usuário final.

Entender a finalidade da API e fazer as perguntas certas sobre o produto ajuda a amadurecê-lo e torná-lo mais confiável e escalável. Afinal, uma API bem projetada não apenas atende aos requisitos técnicos, mas também proporciona uma experiência de uso satisfatória para os clientes.

## Documentação

A documentação é fundamental para garantir a compreensão e o uso correto da interface. Ela fornece informações detalhadas sobre os endpoints disponíveis, os parâmetros que podem ser utilizados em cada requisição, os formatos de resposta esperados e exemplos práticos de uso. Além disso, a documentação serve como uma referência única e centralizada, facilitando a integração com outras aplicações e permitindo que novos desenvolvedores entendam rapidamente como interagir com a API. Uma documentação bem elaborada também contribui para a padronização do desenvolvimento, auxiliando na manutenção e evolução da API ao longo do tempo.