**Introdução a Arquitetura REST**

**Introdução a Arquitetura REST**

Representational State Transfer, ou simplesmente REST, é um estilo de arquitetura para sistemas distribuídos com base em princípios da web, como HTTP, URI e formato de dados como JSON ou XML, para permitir a comunicação entre sistemas distintos de forma simples, escalável e padronizada. Essa arquitetura é amplamente utilizada para a criação de APIs (Application Programming Interface) e serviços web, permitindo que diferentes aplicativos se comuniquem e compartilhem recursos entre si. O REST enfatiza a interoperabilidade entre diferentes sistemas e a separação de preocupações entre servidor e cliente, tornando os sistemas mais flexíveis, fáceis de manter e evoluir.

A arquitetura REST foi criada pelo cientista da computação Roy Fielding durante seu doutorado pela Universidade da Califórnia. Diferentemente de web services SOA (Simple Object Access Protocol), que é um protocolo, REST não é um protocolo e sim uma arquitetura que faz uso do protocolo HTTP como solução para requisições e respostas. É importante destacar que API REST não é a única nomenclatura aceita, embora seja a mais utilizada, podemos ver referências citando API RESTful ou Web Service REST.

**O que é API REST?**

Uma API REST é um tipo de API baseado na arquitetura REST que permite a comunicação entre diferentes sistemas de softwares. Ela define um conjunto de operações (métodos HTTP) que podem ser usadas para criar, ler, atualizar e excluir recursos (dados) em um servidor web. Uma API REST usa URIs (Identificadores Uniformes de Recursos) para identificar os recursos e o formato JSON ou XML para representar dados transferidos entre os sistemas. Com uma API REST, diferentes aplicativos podem se comunicar e compartilhar recursos entre si de forma padronizada e escalável. Isso permite a criação de soluções de softwares mais flexíveis e integráveis.

– **Clientes**  
Quando você acessa uma API como a do Youtube ou Google Maps, o acesso será realizado por meio de uma aplicação cliente. Essa aplicação pode ser o navegador web do seu laptop, um aplicativo em seu smartphone, tablet ou até mesmo smart tv. As APIs REST podem ser classificadas em três tipos com base em seu nível de acesso: privadas, públicas e híbridas.

* **API REST privada:** é uma API que só pode ser acessada por aplicativos e sistemas específicos, geralmente dentro da mesma organização ou rede. Isso garante que somente aplicativos autorizados possam usar a API e acessar os recursos protegidos.
* **API REST pública:** é uma API que pode ser acessada por qualquer aplicativo por meio da internet. Essas APIs são frequentemente usadas por desenvolvedores para integrar seus aplicativos com serviços populares de terceiros, como Google Maps, Facebook, Twitter, etc.
* **API REST híbrida:** é uma combinação de API pública e privada, onde a API pode ser acessada por qualquer aplicativo, mas alguns recursos e funcionalidades são protegidos e só podem ser acessados por aplicativos autorizados.

A escolha de qual tipo de API REST usar depende do nível de segurança necessário e dos recursos que se deseja disponibilizar para os aplicativos clientes. As APIs privadas são geralmente usadas para proteger dados sensíveis e garantir a segurança dos aplicativos internos. As APIs públicas são usadas para disponibilizar serviços para o público em geral e para promover a integração de aplicativos de terceiros. As APIs híbridas são usadas quando é necessário equilibrar a segurança dos dados com a acessibilidade pública dos recursos.

Uma aplicação cliente poderá ser desenvolvida em qualquer linguagem de programação que faça uso do protocolo HTTP. Essa é uma vantagem para os desenvolvedores de aplicações clientes, já que não ficam obrigados a utilizar uma linguagem de programação que não estão habituados ou mesmo que não tenham o domínio necessário.

– **O que são Recursos**  
Os recursos são os registros que as APIs REST fornecem a seus clientes. Eles podem ser de diversos tipos como textos, imagens, vídeos, etc. Os recursos são representados por URIs (Uniform Resource Identifiers), que são endereços únicos que identificam o recurso a ser acessado. Cada recurso pode ser acessado e manuseado usando um conjunto de métodos HTTP (como GET, POST, PUT, DELETE) que são definidas pela API. Os recursos são a base da arquitetura REST e permitem que os clientes manipulem os dados da API de forma padronizada e intuitiva.

– **Documentação**  
A documentação de uma API REST é formada por um conjunto de informações que descrevem a funcionalidade, os recursos e os endpoints disponíveis na API. Ela é destinada aos desenvolvedores que desejam consumir a API por meio de aplicações clientes e fornece detalhes sobre como acessar e usar seus recursos.  
A documentação da API REST também pode incluir exemplos de código que mostram como usar a API em diferentes linguagens de programação e plataformas, bem como informações de autenticação e segurança, limites de taxa de solicitação e outras informações relevantes. Uma boa documentação da API REST é importante para garantir que os desenvolvedores de aplicações cliente possam integrar facilmente a API em seus aplicativos, entender como usar seus recursos e resolver problemas quando eles surgem.

**Benefícios de APIs REST**

O modelo arquitetural REST se tornou preferível em relação ao SOAP com o passar dos anos. Isso ocorreu por várias razões, algumas delas citadas a seguir:

* **Escalabilidade:** As APIs REST são altamente escaláveis e podem lidar com grandes volumes de solicitações sem comprometer o desempenho ou a disponibilidade. Isso é possível devido ao fato da arquitetura REST ser Stateless, ou seja, não é mantido um estado ou contexto entre a aplicação cliente e o servidor.
* **Flexibilidade:** As APIs REST permitem que os desenvolvedores criem soluções de software flexíveis e modulares, que podem ser facilmente adaptadas para atender às necessidades em constante mudança dos usuários.
* **Padronização:** As APIs REST são baseadas em padrões abertos e amplamente adotados, como HTTP, URIs e JSON ou XML, o que torna mais fácil para os desenvolvedores integrar diferentes sistemas de software.
* **Reutilização:** As APIs REST permitem que os recursos e serviços sejam compartilhados e reutilizados em diferentes aplicativos, o que pode economizar tempo e recursos de desenvolvimento.
* **Integração:** As APIs REST permitem a integração fácil de diferentes aplicativos e sistemas de software, permitindo que as organizações compartilhem dados e informações entre si.
* **Acessibilidade:** As APIs REST podem ser acessadas por diferentes plataformas e dispositivos, incluindo navegadores da web, aplicativos móveis e dispositivos IoT, o que amplia o alcance da solução de software.
* **Segurança:** As APIs REST oferecem recursos de segurança, como autenticação e autorização, para garantir que apenas usuários autorizados possam acessar os recursos protegidos.

Esses benefícios tornam as APIs REST uma solução popular e eficiente para a construção de aplicativos e sistemas de software modernos e escaláveis.

**Segurança em APIs REST**

Existem diferentes tipos de segurança que podem vir a ser utilizados em APIs REST e assim, proteger os dados e garantir a integridade das solicitações e respostas. Alguns dos tipos de segurança mais aplicados em API REST são:

* **Autenticação:** verificar a identidade do usuário que está solicitando acesso à API. Os métodos de autenticação incluem senhas, tokens de acesso, certificados digitais e autenticação de dois fatores.
* **Autorização:** verificar se o usuário tem permissão para acessar o recurso solicitado na API. Isso é geralmente feito com base em regras de acesso predefinidas, como as funções do usuário.
* **Criptografia:** protege os dados em trânsito, tornando-os inelegíveis para qualquer pessoa que possa interpretá-los. A criptografia pode ser aplicada a diferentes níveis, como TLS/SSL no nível de transporte ou criptografia ponta a ponta para dados sensíveis.
* **Monitoramento de atividades:** rastreia as atividades do usuário na API para detectar comportamentos suspeitos e garantir a conformidade com as políticas de segurança. Isso pode incluir registros de auditoria, monitoramento de eventos e alertas de segurança.
* **Gerenciamento de tokens:** protege os tokens de acesso usados na autenticação, garantindo que apenas usuários autorizados possam acessar os recursos da API. Isso pode incluir a validação de tokens, a expiração de tokens e a revogação de tokens em caso de comprometimento de segurança.

Os diferentes tipos de segurança são de grande valia para garantir que as APIs REST se mantenham seguras e protejam os dados confidenciais dos usuários e das organizações.

**API REST - Fundamentos**

A arquitetura REST trabalha sobre o protocolo HTTP e cada URI funciona como um recurso da API. Portanto, devemos usar substantivos para determinar recursos em vez de verbos.

URI - Uniform Resource Identifier, ou Identificador Uniforme de Recursos  
Exemplo: https://mballem.com/cursos/  
A URI une o Protocolo (https://);  
A localização do recurso (URL - [mballem.com](http://mballem.com/));  
E o nome do recurso (URN - /cursos/)

Endpoints estilo RPC usam verbos, por exemplo, api/v1/getPersons. Em contrapartida, em REST, esse endpoint deve ser escrito como api/v1/persons. Então, você deve estar se perguntando como poderemos diferenciar as ações executadas em um recurso REST? Pois bem, é agora que faremos uso dos chamados Métodos HTTP (HTTP METHOD). A ideia é fazer com que os métodos HTTP trabalhem como um verbo, por exemplo, GET (recuperar), DELETE (remover), POST (criar), PUT (modificar) e PATCH (atualização parcial), entre outros.

Endpoint  
O endpoint REST é um URI exclusivo que representa um recurso REST.  
Por exemplo, http://app/api/v1/persons é um endpoint REST. Além disso, /api/v1/persons é o caminho do endpoint e as pessoas são recurso REST.

**Métodos HTTP**

O protocolo HTTP define um conjunto de métodos ou verbos de requisição responsáveis por indicar a ação a ser executada para um dado recurso. Em APIs REST alguns desses verbos são mais comuns que outros de serem utilizados. Isso acontece não por preferência, mas por conta dos tipos de ações sobre os recursos. Por exemplo, o método GET é utilizado para operações que recuperam um recurso e provavelmente uma API REST terá muito mais ações do tipo GET do que do tipo DELETE (usado para exclusão de recursos).

Um conceito muito falado quando começamos a estudar REST é quais métodos são ou não são idempotentes. Na verdade, esse não seria um conceito específico da arquitetura REST, mas sim, do protocolo HTTP. Porém, por algum motivo, sempre foi muito citado em estudos sobre REST.  
Um método HTTP idempotente é um método HTTP que pode ser chamado várias vezes sem alterar os resultados. Não importa se o método é chamado apenas uma vez ou dezenas de vezes. O resultado deve ser o mesmo. Importante destacar, idempotência se aplica ao resultado, não ao recurso. Entretanto, esse assunto pode gerar controvérsias, assim como vemos na postagem [*O que é ser idempotente em REST? O debate continua*](https://www.infoq.com/br/news/2013/05/idempotent/) do site Infoq.

GET

O GET é usado para recuperar um recurso do servidor. É uma operação segura e idempotente, ou seja, várias solicitações para o mesmo recurso retornarão o mesmo resultado e não modificará o estado do recurso no servidor.

Alguns casos de uso do GET:

* Recuperar uma página da web
* Buscar a lista de postagens de um blog
* Download de arquivo de um servidor

POST

POST é usado para criar um novo recurso no servidor. Não é considerado um método idempotente, o que significa que várias solicitações para criar o mesmo recurso resultará em vários recursos sendo criados. As solicitações POST também podem modificar o estado do recurso no servidor.

Alguns casos de uso do POST:

* Criação de uma nova conta de usuário
* Carregar uma nova imagem ou arquivo
* Enviar um formulário a um servidor para criar um novo recurso

PUT

PUT é usado para atualizar um recurso existente no servidor. É considerado idempotente, o que significa que várias solicitações para atualizar o mesmo recurso resultarão em o mesmo estado do recurso no servidor. O PUT também pode criar um novo recurso se o recurso que está sendo atualizado não existe.

Casos de uso para PUT:

* Atualizar o conteúdo de uma postagem de blog
* Modificar os detalhes de uma conta de usuário
* Substituir um arquivo ou imagem em um servidor por um nova versão
* Atualizar todo o endereço de um contato

PATCH

O método PATCH é usado para fazer uma atualização parcial em um recurso existente no servidor. O objetivo é realizar atualizações em campos ou propriedades específicas de um recurso, sem exigir que todo o recurso seja enviado ao servidor.

Casos em que o PATCH é usado:

* Atualizar o título ou descrição de uma postagem de blog
* Alterar a senha ou endereço de e-mail associado a uma conta de usuário
* Fazer uma atualização parcial de um arquivo ou imagem

DELETE

O verbo DELETE é usado para excluir um recurso do servidor. É considerado idempotente, o que significa que várias solicitações para excluir o mesmo recurso resultarão no mesmo estado do servidor. Isso parece estranho e gera muitas dúvidas. Mas na verdade, ao executar a primeira solicitações de DELETE para o recurso, ele será excluído e a resposta será 200 (OK) ou 204 (No Content). As próximas solicitações para o mesmo recurso (já excluído) retornarão 404 (não encontrado). Claramente, a resposta é diferente da primeira solicitação, mas não há mudança de estado para nenhum recurso do lado do servidor porque o recurso original já foi excluído.

Usar casos para DELETE incluem:

* Remover uma postagem de blog
* Excluir uma conta de usuário
* Remover um arquivo ou imagem

Podemos destacar que uma operação de exclusão pode ser física ou lógica. A física é quando o recurso é realmente excluído da base de dados e a lógica quando uma coluna na tabela do recurso indica que ele foi excluído, mesmo ainda estando presente na tabela. A exclusão lógica é muito usada para manter dados históricos.

**Demais métodos HTTP**

O protocolo HTTP tem mais alguns métodos além dos citados. Para algumas APIs eles poderão vir a ser úteis, mas talvez para outras, nem tanto. Portanto, nossa abordagem será sobre o GET, POST, PUT, PATCH e DELETE. Caso queira conhecer outros métodos, como HEAD, OPTIONS, TRACE, etc, acesse - [Métodos de requisição HTTP](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Methods).

**Status HTTP**

Toda vez que uma solicitação HTTP é enviada a uma API REST, algum tipo de resposta será retornada de volta ao solicitante. Independentemente se a solicitação foi aceita, gerou erro, se o recurso não foi encontrado ou se teve sucesso na execução.  
Essa resposta é chamada de HTTP Status Code e existem cinco categorias de Status Code no protocolo HTTP.

* Informational responses (100 – 199)
* Successful responses (200 – 299)
* Redirects (300 – 399)
* Client errors (400 – 499)
* Server errors (500 – 599)

Alguns dos mais comuns códigos em respostas REST

* **200 - OK** - Padrão de resposta para solicitações HTTP sucesso. A resposta real dependerá do método de solicitação usado. Em uma solicitação GET, a resposta conterá uma entidade que corresponde ao recurso solicitado. Em uma solicitação POST a resposta conterá a descrição de uma entidade, ou contendo o resultado da ação.
* **201 - Create** - O pedido foi cumprido e resultou em um novo recurso que está sendo criado. Usado em solicitação POST e PUT quando criar um novo recurso.
* **204 - No Content** - O servidor processou a solicitação com sucesso, mas não é necessária nenhuma resposta. Resposta comum na solicitação de um DELETE.
* **400 - Bad Request** - O pedido não pôde ser entregue devido à sintaxe incorreta.
* **401 - Unauthorized** - Basicamente usado quando a solicitação exige que o cliente esteja autenticado, entretanto, o cliente ainda não realizou a autenticação. A resposta deve incluir um cabeçalho do campo www-authenticate contendo um desafio aplicável ao recurso solicitado.
* **403 - Forbidden** - O pedido é reconhecido pelo servidor, mas este recusa-se a executá-lo. Ao contrário da resposta “401”, aqui a autenticação não fará diferença e o pedido não deve ser requisitado novamente.
* **404 - Not Found** - O recurso solicitado não foi encontrado, mas pode ser disponibilizado novamente no futuro. As solicitações subsequentes pelo cliente são permitidas.
* **405 - Method Not Allowed** - Foi feita uma solicitação de um recurso usando um método de pedido que não é compatível com esse recurso, por exemplo, usando GET em um formulário, que exige que os dados sejam apresentados via POST, PUT; ou usados em um recurso somente de leitura.
* **409 - Conflict**- Indica que a solicitação não pôde ser processada por causa do conflito no pedido, como um conflito de edição.
* **422 - Unprocessable Entity** - O pedido foi bem formatado, mas foi incapaz de ser seguido devido a erros de semântica.
* **500 - Internal Server Error** - Indica um erro do servidor ao processar a solicitação.
* **501 - Not Implemented** - O servidor ainda não suporta a funcionalidade ativada.
* **503 - Service Unavailable** - O servidor está em manutenção ou não consegue dar conta dos processamentos de recursos devido à sobrecarga do sistema. Isto deve ser uma condição temporária.

É muito importante que todas as respostas de solicitações possuam um Status Code definido para caso da solicitação ter sido processada ou ter gerado algum tipo de erro. O Status Code é a forma como a aplicação cliente vai saber tratar a resposta da requisição. Por exemplo, se a resposta for 200 o cliente poderá listar os dados recebidos, mas se for 403 poderá informar ao usuário que ele não tem permissão de acesso àquele recurso.

Para saber mais saber mais sobre status code do HTTP acesse - [HTTP response status codes](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status)

#### ****API REST - Boas Práticas****

Quando desenvolvemos uma API REST é considerado seguir aquilo que está classificado como boas práticas. Por exemplo, responder uma solicitação com o HTTP Status Code apropriado, seja para sucesso ou para erro, é uma boa prática. Outra boa prática muito importante, mas que muitas vezes não é seguida por desenvolvedores de APIs, é usar corretamente o nome dos recursos.

Como já foi citado anteriormente, um recurso não deve ser um verbo e sim um substantivo. O verbo, ou seja, a ação, será identificado pelo HTTP METHOD (GET, POST, PUT, …). Entretanto, o nome do recurso é a primeira parte do caminho de acesso a esse recurso e ele deve ser sempre um substantivo no plural.

Algumas regras sobre como nomear corretamente seus recursos:

1. O nome do recurso deve ser um substantivo
2. O nome do recurso deve estar no plural
3. O nome do recurso deve estar por inteiro em letras minúsculas
4. Em caso de nome composto, os nomes devem ser separados por hífen
5. Evite usar / ao final da URI

Veja alguns exemplos:

1. GET http://example.com/clientes/1

O recurso acima busca por um cliente com identificador igual a **1**. Observe que temos um substantivo como nome do recurso, clientes, e não um verbo buscarClientes ou getClientes. O recurso faz alusão a uma coleção de registros, por isso, deve estar no plural.  
É possível também ter um sub recurso, às vezes chamado de recurso aninhado, associado a outro recurso.

1. GET http://example.com/clientes/15/pedidos

Observe que o recurso /clientes/15/ tem um sub recurso chamado pedidos. Assim, todos os pedidos, ligados ao cliente de identificador igual a **15**, seriam o alvo da solicitação.

Ao montar a URI de um recurso é sempre válido considerar o uso dos caminhos. Os caminhos, ou PATH em inglês, são as partes da URI que identifica qual recurso deve ser acessado. Por exemplo, /clientes/1 tem dois caminhos. O primeiro, mais a esquerda, é a raiz do recurso, o que indica que **clientes** será acessado. O segundo caminho está depois da segunda barra, o qual indica a intenção de recuperar exclusivamente o recurso identificado como **1**.

Já no caso de /clientes/15/pedidos temos três caminhos, onde o primeiro aponta para o recurso **clientes**, o segundo para o cliente alvo (15) e o terceiro para o sub recurso **pedidos** do cliente alvo. Seria possível ainda ter um quarto caminho, indicando qual o pedido deveria ser recuperado, por exemplo: /clientes/15/pedidos/2

Caso existem recursos que indiquem pedidos concluídos, poderíamos nomeá-lo da seguinte

forma pedidos-concluidos:

1. GET http://example.com/clientes/15/pedidos-concluidos

Dessa forma, a solicitação buscaria todos os sub recursos do cliente 15 que estão concluídos.

Considere uma API REST com os seguintes recursos:

1. POST http://example.com/clientes
2. GET http://example.com/clientes

A primeira vista podemos interpretar como algo errado. Isso porque, os dois recursos possuem a mesma URI. Como bem sabemos, uma URI deve ser única, já que aponta para um recurso específico. Entretanto, quando falamos de REST, não é somente a URI que será considerada, mas também o verbo HTTP.

Como as URIs apresentadas têm verbos diferentes, elas se tornam distintas, assim, quando o cliente enviar uma solicitação com o verbo POST a API saberá qual recurso o cliente está solicitando, sem fazer qualquer tipo de confusão. Por isso, é importante sempre definir corretamente os verbos HTTP de suas APIs.

#### ****Path e Query Params****

Suponha que um recurso de usuários possui os campos de armazenamento para id, nome, data de nascimento e cpf. Ao criar um recurso para localizar por id temos duas possibilidades aceitáveis:

1. GET http://example.com/usuarios/1
2. GET http://example.com/usuarios/id/1

Entretanto, especificamente para um campo do tipo id, não é costumeiro usar a path id. Sendo assim, embora aceitável, é considerada uma boa prática não usá-lo.  
Agora, suponha que a ideia seja recuperar o usuário pelo CPF. Neste caso, será necessário indicar qual o campo fará parte da busca:

1. GET http://example.com/usuarios/cpf/23735034098

Esse tipo de operação faz uso do que chamamos de PATH PARAMETERS, onde informamos os parâmetros via caminhos da URI. Entretanto, podemos fazer uso de QUERY PARAMETERS:

1. GET http://example.com/usuarios?cpf=23735034098

As duas formas são consideradas uma boa prática. Porém, existem alguns especialistas que fazem uma distinção no uso de Path ou Query Parameters. É sugerido que, apenas os campos de valores únicos e que não  mudem com o tempo, sejam usados com Path Params. Os campos que, não são considerados únicos ou que mudem com o tempo, sejam selecionados com Query Params. Seguindo esse princípio, o campo de id será sempre único e imutável, é provável que o campo de CPF também seja único. Já os campos para nome e data de nascimento poderiam se repetir na tabela. Assim, esses dois últimos deveriam ser usados apenas com Query Params:

1. GET http://example.com/usuarios?nome=beltrano
2. GET http://example.com/usuarios?dtNascimento=2000-10-05

Dessa forma, você só está usando path params ao especificar qual recurso buscar, mas isso não classifica os recursos de forma alguma. Quem classifica os recursos são os atributos que se repetem ou podem ser modificados no futuro. Por exemplo, um recurso carros possui os atributos id, marca, modelo, chassi e cor.

Quais desses atributos são únicos e imutáveis com o tempo? Apenas o id e o chassi. Você pode estar se perguntando, mas a marca de um carro não muda. Sim, não muda, você está certo. Mas em uma tabela teríamos inúmeros carros da mesma marca, ou seja, não seria um recurso único. Assim, a marca, o modelo e a cor são considerados atributos de classificação/filtro.

1. GET http://example.com/carros?marca=honda

A solicitação acima está filtrando a busca dos carros da marca honda.

1. GET http://example.com/carros?cor=verde

A solicitação acima está filtrando a busca dos carros de cor verde. A cor é um atributo que além de se repetir pode ser mutável. Ou seja, um carro que hoje é verde, amanhã pode ter sua cor substituída por azul.

1. GET http://example.com/carros/chassi/100990058

A solicitação acima está especificando que deseja o recurso de chassi 100990058.

#### ****Paginação de Recursos****

Ainda sobre Query Params, existe uma boa prática que deve ser sempre seguida. Ela se refere a paginação de recursos.  
Imagine que tenha um recurso de clientes com 1000 registros ou mais. Uma solicitação do tipo GET será realizada para recuperá-los. Como bem sabemos, não é uma boa pratica trazer uma quantidade tão grande de registros em uma única solicitação, então, o aconselhado é usar a paginação de resultados.

Uma paginação contém algumas informações básicas como o número da página, a quantidade máxima de registros retornados e por vezes a ordenação por algum campo específico.

Quando falamos sobre paginação em API REST, os parâmetros usados na paginação devem sempre ser enviados na solicitação como Query Params. Normalmente os nomes dos parâmetros são page para o número da página, size ou limit para a quantidade máxima de registros a serem retornados e sort ou order para a ordenação. Mas, eles podem ser encontrados com outros nomes ou abreviações em alguns APIs. Veja um breve exemplo:

1. GET http://example.com/clientes?page=0&size=10&sort=nome,asc

A solicitação acima está ordenando os clientes pelo nome de forma ascendente (a-z), iniciando o retorno pela página 0 e limitando a quantidade de registros por página em 10.

#### ****HTTP HEADERS (Cabeçalhos)****

Os cabeçalhos HTTP são usados para fornecer informações adicionais sobre uma solicitação ou resposta na arquitetura REST. Existem tanto os cabeçalhos enviados pelo cliente quanto aqueles enviados na resposta pela API.  
Aqui estão alguns cabeçalhos comuns usados nas APIs REST e suas explicações:

**Content-Type**  
O cabeçalho Content-Type pode ser usado para especificar o tipo de mídia dos dados que estão sendo enviados na solicitação ou resposta. Isso permite que o cliente e o servidor concordem sobre o formato dos dados que estão sendo trocados. Por padrão, quem define o tipo de dados é a API e o cliente deve enviar solicitações baseadas nesse tipo. Algumas vezes, uma API pode aceitar mais de um tipo de formato de dados, por exemplo, JSON e XML. Nesse caso, a aplicação cliente deve informar no cabeçalho qual tipo está enviando.

Alguns exemplos:

1. Content-Type: application/json
2. Content-Type: application/x-www-form-urlencoded
3. Content-Type: text/html; charset=utf-8
4. Content-Type: multipart/form-data
5. Content-Type: text/plain

**Accept**  
Este cabeçalho é usado para especificar os tipos de mídia que o cliente pode aceitar na resposta. Se o servidor não pode fornecer uma resposta em um dos especificados tipos de mídia, ele retornará um erro. Como dito, quem vai especificar os tipos de dados sempre será a API. O cliente deverá ser desenvolvido com base nos tipos de dados oferecidos pela API.

Alguns exemplos:

1. Accept: \*/\* (Padrão, aceita o tipo da API)
2. Accept: text/html
3. Accept: image/\*
4. Accept: text/html, application/xhtml+xml, application/xml;q=0.9, \*/\*;q=0.8

**Location**  
Este cabeçalho é usado em uma resposta 201 (Create) para indicar a URL do recurso recém-criado. Ou seja, quando um novo recurso é criado, o cabeçalho Location vai ser enviado na resposta com o endereço completo de acesso ao recurso.

Por exemplo, se um cliente for criado e um id número 10 for gerado, o Location será:

1. Location: GET http://example.com/clientes/10
2. Location: http://example.com/clientes/10

**Authorization**  
O cabeçalho de autorização é usado pelo cliente para fornecer informações de autenticação, como um token de acesso, para o servidor. Isso permite que o servidor verifique que o cliente está autorizado a acessar o pedido de recurso.

O formato esperado segue as instruções:

1. Authorization: <auth-scheme> <authorization-parameters>

Basic Authentication:

1. Authorization: Basic <credentials>

Bearer Authentication:

1. Authorization: Bearer <token>

**WWW-Authenticate**  
A API, quando receber uma solicitação a um recurso que necessite de autenticação e o cliente não estiver autenticado, deve responder com o cabeçalho WWW-Authenticate. Essa resposta deve incluir pelo menos um desafio, para indicar quais esquemas de autenticação podem ser usados para acessar o recurso (e quaisquer dados adicionais necessários para cada esquema específico).

1. WWW-Authenticate: "Bearer realm='/api/v1/login'"

O desafio indica que a próxima requisição, para o mesmo recurso, deve conter uma autenticação do tipo Bearer, que deve ser realizada usando a URI /api/v1/login.

Estes são alguns dos cabeçalhos principais entre cliente e servidor quando se trata de APIs REST. Outros cabeçalhos podem vir a ser usados conforme o desejo dos desenvolvedores da API.  
Para saber mais sobre cabeçalhos HTTP acesse - [HTTP headers](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers)

**O Padrão de Projetos DTO**

O padrão Data Transfer Object - DTO

O padrão de projetos DTO (Data Transfer Object) é um padrão de projeto de software que visa transferir dados entre diferentes componentes de um sistema de forma eficiente e consistente.

O DTO é uma classe simples que normalmente contém apenas campos para armazenar dados e não possui comportamento (métodos). Ele é utilizado para encapsular e transportar dados entre camadas de um sistema, como a camada de apresentação, a camada de serviços e a camada de acesso a dados.

Existem várias razões para usar o padrão DTO em um sistema:

**1 - Transferência eficiente de dados:** Ao usar DTOs, é possível transferir apenas os dados necessários entre as camadas do sistema, reduzindo a quantidade de informações transmitidas e melhorando o desempenho.

**2 - Separação de responsabilidades:** O uso de DTOs ajuda a separar as responsabilidades das diferentes camadas do sistema. Cada camada pode ter seus próprios DTOs específicos, adequados para suas necessidades, sem expor diretamente os detalhes internos dos objetos de domínio.

**3 - Versatilidade:** Os DTOs podem ser usados para representar diferentes estruturas de dados, permitindo a flexibilidade na definição dos dados a serem transferidos. Isso é especialmente útil em sistemas complexos, onde diferentes partes do sistema podem exigir diferentes visualizações dos dados.

**4 - Manutenção simplificada:** Ao utilizar DTOs, é possível alterar a estrutura interna dos objetos de domínio sem afetar a interface dos DTOs. Isso facilita a manutenção do sistema, uma vez que as mudanças internas não afetam os componentes que dependem dos DTOs.

Um exemplo prático de uso do padrão DTO seria em um sistema de e-commerce, onde um DTO chamado ProdutoDTO poderia ser usado para transferir dados sobre um produto da camada de acesso a dados para a camada de apresentação. O DTO conteria os campos relevantes do produto, como nome, preço, descrição, etc., e seria responsável por transportar esses dados de forma eficiente entre as diferentes camadas do sistema.

1. public class Produto {
2. private Long id;
3. private String nome;
4. private double preco;
5. private String descricao;
6. private Date dataCadastro;
8. // Construtores, Getters e Setters omitidos no código
9. }
11. public class ProdutoDTO {
12. private String nome;
13. private double preco;
14. private String descricao;
16. // Construtires Getters e Setters omitidos no código
17. }

Para utilizar esse DTO, você pode criar uma instância do ProdutoDTO, preencher os campos com os dados do produto e passá-lo entre as camadas do sistema, como exemplo:

1. ProdutoDTO produtoDTO = new ProdutoDTO();
2. produtoDTO.setNome("Camiseta");
3. produtoDTO.setPreco(29.99);
4. produtoDTO.setDescricao("Uma camiseta confortável e estilosa.");
6. // Passar o produtoDTO para outra camada do sistema
7. // Até converte-lo, quando necessário, em um objeto do tipo Produto.
8. Produto produto = conversorToProduto(produtoDTO)

Assim, o DTO permite encapsular os dados do produto e transportá-los de forma eficiente entre as diferentes partes do sistema, mantendo a separação de responsabilidades e facilitando a manutenção do código.

Biblitoecas de suporte a DTO

Existem várias bibliotecas em Java que podem ajudar a automatizar o uso de DTOs. Algumas das mais populares são:

**Dozer:** O [Dozer](https://github.com/DozerMapper/dozer) é uma biblioteca de mapeamento de objetos que permite configurar o mapeamento entre classes usando arquivos de configuração XML. Ele oferece suporte a mapeamento de objetos complexos e pode ser usado para automatizar o mapeamento entre entidades e DTOs.

**MapStruct:** O [MapStruct](https://mapstruct.org/) é uma biblioteca de mapeamento de objetos que pode gerar automaticamente código para mapear entidades para DTOs e vice-versa. Ele usa anotações para configurar o mapeamento e gera o código necessário durante o processo de compilação. Isso elimina a necessidade de escrever manualmente o código de mapeamento entre as classes.

**ModelMapper:** O [ModelMapper](https://modelmapper.org/) é outra biblioteca de mapeamento de objetos que permite a configuração de regras de mapeamento entre classes. Ele também pode ser usado para mapear automaticamente entidades para DTOs e vice-versa, eliminando a necessidade de escrever código de mapeamento manualmente.

Essas são apenas algumas das bibliotecas disponíveis em Java para automatizar o uso de DTOs. Cada uma delas tem sua própria sintaxe e abordagem para configuração e geração de código de mapeamento. A escolha da biblioteca depende das necessidades específicas do seu projeto e das suas preferências pessoais.

Analise da classe Usuario e porque criar DTO:

Analisando os campos da classe Usuario podemos verificar que alguns deles não deveriam fazer parte de um corpo de requisição como também de um corpo de resposta.

1. public class Usuario implements Serializable {
2. private Long id;
3. private String username;
4. private String password;
5. private Role role;
6. private LocalDateTime dataCriacao;
7. private LocalDateTime dataModificacao;
8. private String criadoPor;
9. private String modificadoPor;
11. // Construtores, Getters e Setters omitidos
12. }

Em se tratando de corpo de requisição, o campo role não deveria estar presente no JSON enviado pelo cliente. Isso porque, não é o cliente/usuário quem deve escolher qual perfil vai se cadastrar na API. Além disso, os campos de auditoria também não devem ser cadastrados pelo cliente.

Em se tratando de corpo de resposta, veja o resultado após uma consulta pelo id:

1. {
2. "id": 1,
3. "username": "[email protected]",
4. "password": "101010",
5. "role": "ROLE\_ADMIN",
6. "dataCriacao": null,
7. "dataModificacao": null,
8. "criadoPor": null,
9. "modificadoPor": null
10. }

Alguns campos são desnecessários serem retornados para o cliente e, novamente, estamos falando sobre os campos de auditoria. O cliente não precisa e nem tem porque, ter qualquer motivo para visualizar os dados de auditoria. Esses valores são específicos para informações internas do gerenciamento da API.

Mas o mais grave nesta resposta é ter o campo password presente nela. Mesmo que o valor da senha estivesse criptografado, qual seria o motivo plausível para que a senha fosse retornada ao cliente? O campo de senha deve ficar restrito a API. Nunca deveria ser retornado como parte de uma resposta ao cliente.

Por conta desses motivos o ideal é ter DTOs para lidar tanto com os valores recebidos no corpo da requisição como também com os valores respondidos no corpo de resposta.

Outro fato importante a se destacar se refere a manipulação dos registros. Por exemplo, o perfil ROLE\_ADMIN é uma instrução definida na API, para controlar dentro da API quem é administrador. Caso essa informação chegue ao cliente, ela não precisa chegar exatamente como ROLE\_ADMIN. Apenas uma informação ADMIN é o bastante. Isso porque, o campo de resposta já é role e por isso, podemos evitar a redundancia. Mas não poderiamos manipular o valor ao retornar um objeto Usuario como corpo da resposta. Porém, poderiamos fazer a manipulação do valor caso tivessemos um DTO como resposta.

Em resumo, o padrão de projetos DTO é uma abordagem para transferir dados entre diferentes componentes de um sistema de forma eficiente, consistente e flexível, permitindo a separação de responsabilidades e melhorando a manutenção do sistema.