COTI INFORMÁTICA
ESCOLA DE NERDS

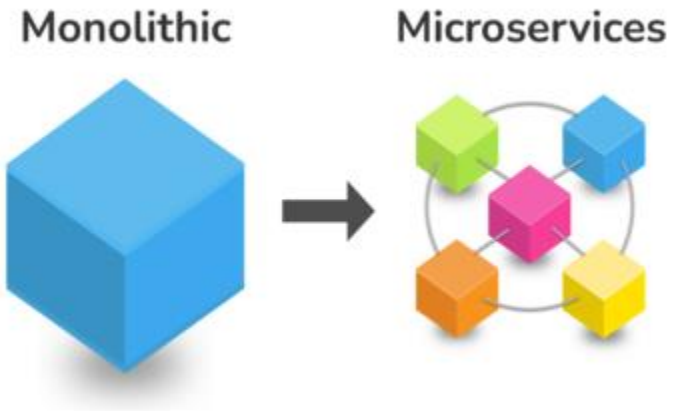
Java WebDeveloper
Formação FullStack

Professor Sergio Mendes

Aula 17 (03/03/23)

COTI INFORMÁTICA
ESCOLA DE NERDS

Construindo APIs em aplicações Java



Uma pergunta difícil de ser respondida quando trabalhamos com **microsserviços** é com relação ao “tamanho” adequado das aplicações que constituem o ecossistema.

Apenas para citarmos alguns exemplos e evidenciar a importância do assunto, serviços de granularidade inadequada podem implicar em:

- Aumento do custo de manutenção e do índice de retrabalho das equipes;
- Prejuízo nos requisitos não funcionais como escalabilidade, elasticidade e disponibilidade;
- Agravamento do impacto de arquitetura descentralizada em termos de performance;
- Complexidade acidental no monitoramento e detecção de falhas da aplicação.

Critérios de desintegração dos microsserviços.

Afinal de contas, quais seriam os critérios que justificariam quebrar um serviço em aplicações menores?

São eles:

- Escopo e funcionalidade;
- Código de alta volatilidade;
- Escalabilidade;
- Tolerância à falha;
- Segurança;
- Extensibilidade

Escopo e funcionalidade

Essa é a justificativa mais comum na quebra de granularidade de um serviço. Um microsserviço objetiva ter alta coesão. Deve fazer uma única coisa e fazê-la muito bem.

A natureza subjetiva desse critério pode induzir a decisões equivocadas de arquitetura.

Como “responsabilidade única” acaba dependendo da avaliação e interpretação individual de cada um, é muito difícil afirmar com precisão quando essa recomendação é válida.

Código de alta volatilidade

A velocidade que o código fonte muda é uma ótima diretriz para fundamentar a decomposição de uma aplicação. Imagine um serviço de títulos financeiros, no qual o módulo de histórico tem novas implementações a cada semana, enquanto os módulos de títulos a pagar e receber são alterados a cada seis meses.

Nessa situação, a decomposição arquitetural pode ser uma decisão sábia para reduzir o escopo de testes antes de cada liberação.
Essa decisão também trará ganho de agilidade e manterá nosso risco de deploy controlado, garantindo que o serviço de títulos não seja mais afetado pelas frequentes mudanças na lógica do serviço de histórico.

Escalabilidade e throughput

Muito semelhante ao item anterior, a escalabilidade de um serviço pode ser uma ótima justificativa para a quebra da aplicação.

Diferentes níveis de demanda, em diferentes funcionalidades, podem exigir que o serviço tenha que escalar de formas distintas e independentes.

Manter a aplicação centralizada pode impactar diretamente a capacidade e os custos da arquitetura em termos de escalabilidade e elasticidade.

Dependendo do contexto de negócio, este critério, por si só, pode ser suficiente para justificar sua decisão.

Tolerância à falha

O termo **tolerância à falha** descreve a capacidade de uma aplicação de continuar operando mesmo quando uma determinada parte desta aplicação deixa de funcionar.

Vamos considerar o exemplo anterior. Imagine um cenário onde o serviço de histórico, por integrar com diversas aplicações de terceiros fora da nossa arquitetura, costuma falhar com certa frequência, chegando ao ponto de reiniciar todo o serviço de títulos financeiros e gerar indisponibilidade.

Nesse caso, uma decisão compreensível seria: separar a rotina problemática em um serviço isolado. Para, assim, manter nossa aplicação funcional a despeito de eventuais falhas catastróficas no serviço de históricos.

Segurança

Considere um exemplo onde um serviço que trata das informações básicas de um usuário (endereço, telefone, nome etc.) precisa gerenciar dados sensíveis dos seus cartões de crédito.

Essas informações podem ter requisitos distintos com relação ao acesso e a proteção dos mesmos. Quebrar o serviço, neste caso, pode auxiliar em:

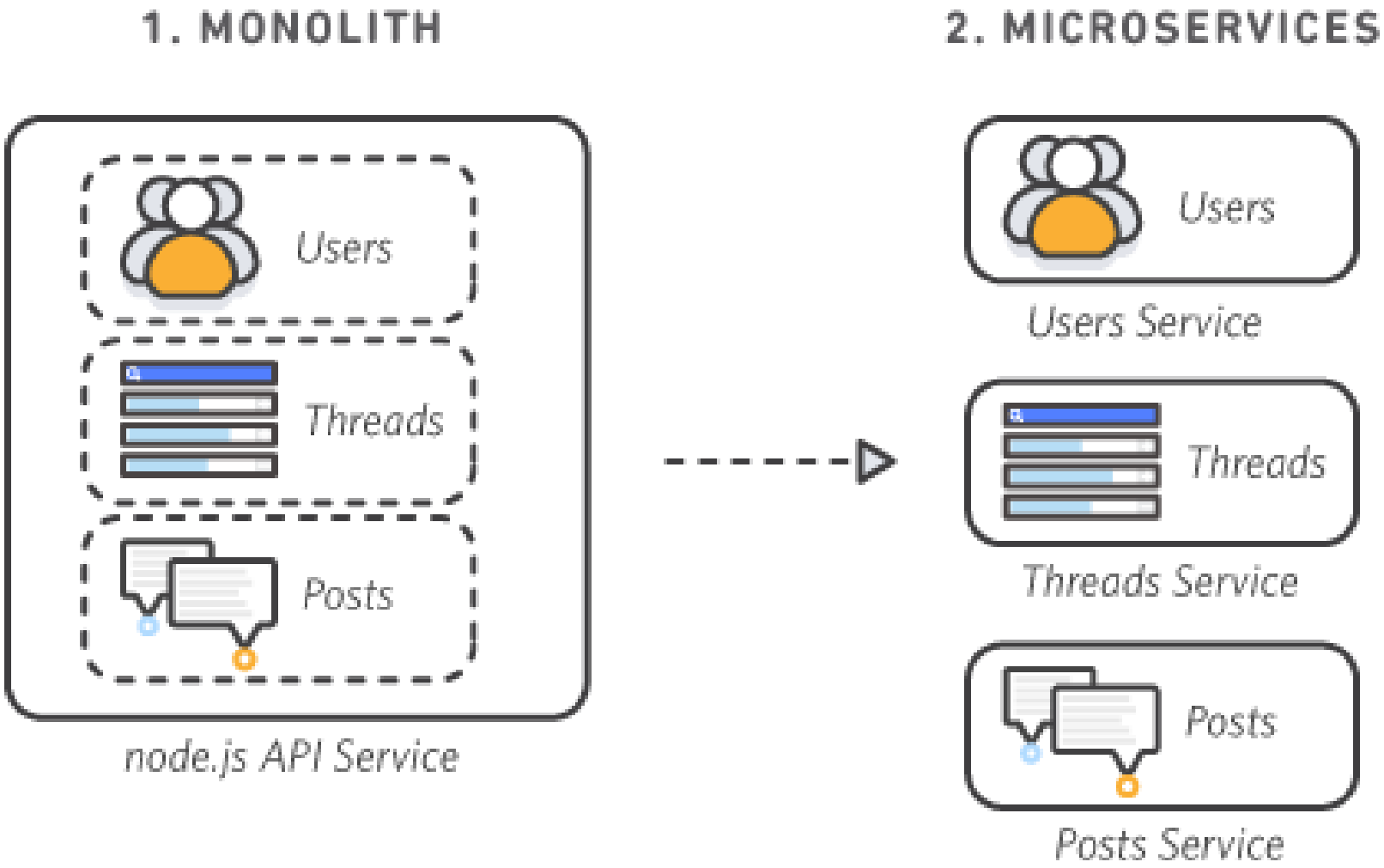
- Restringir ainda mais o acesso ao código cujos critérios de segurança sejam mais rigorosos;
- Evitar que o código menos restrito seja impactado com complexidade acidental de outros módulos

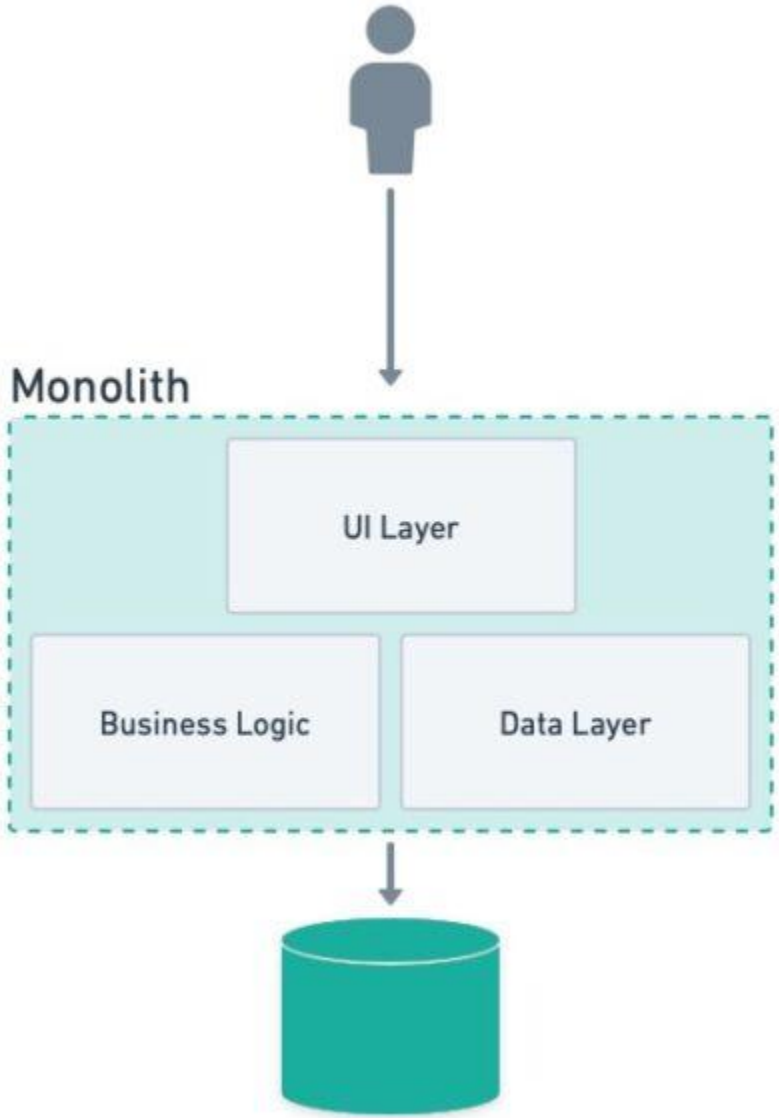
Extensibilidade

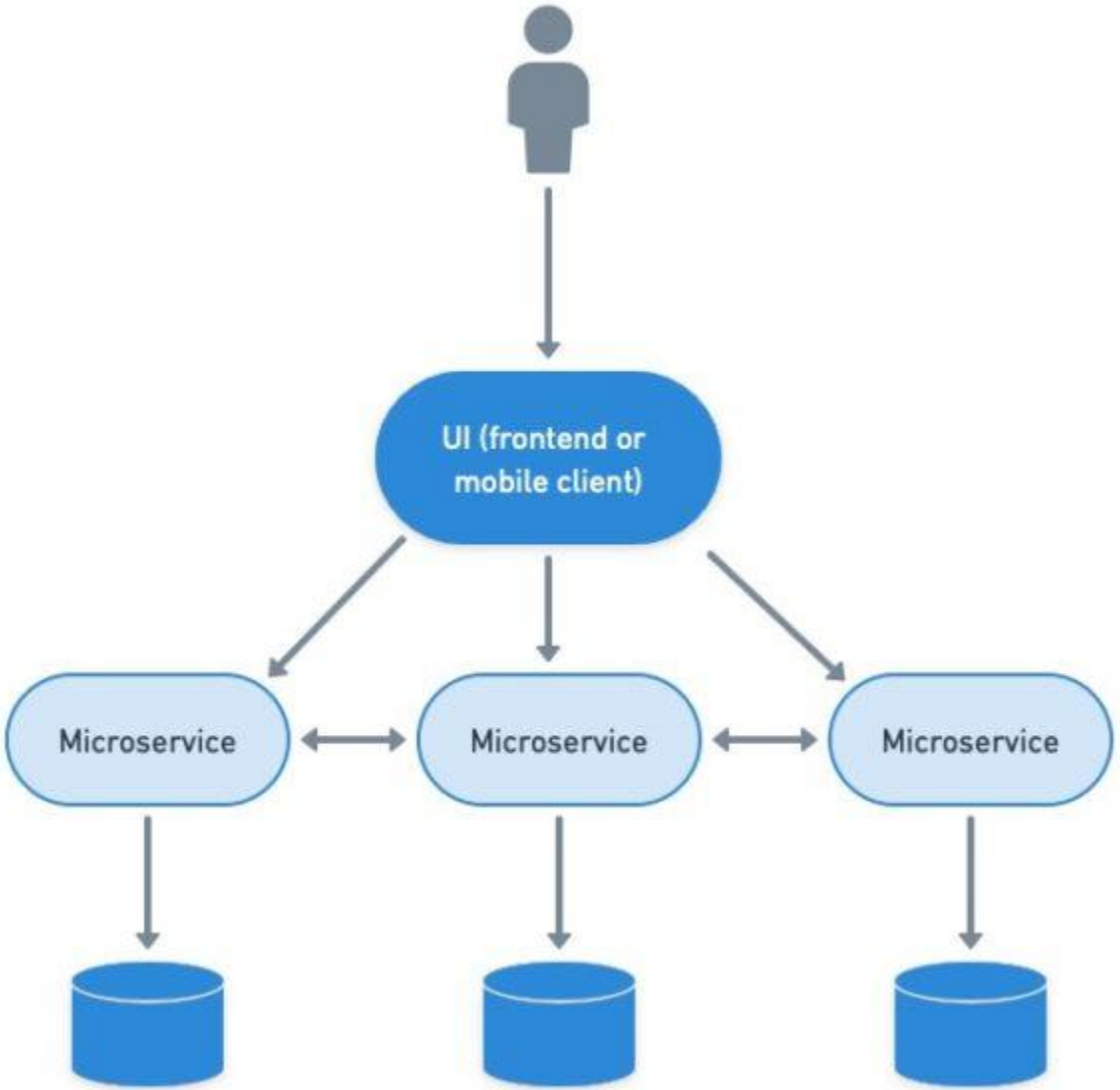
Uma **solução extensível** tem a capacidade de ter novas funcionalidades adicionadas com facilidade conforme o contexto de negócio cresce.

Essa habilidade também pode ser um forte motivador para segregar uma aplicação. Imagine que uma empresa tem um serviço centralizado para gerenciar formas de pagamento e deseja suportar novos métodos.

Certamente, seria possível consolidar tudo isso em um único serviço. Porém, a cada nova inclusão, o escopo de testes se tornaria mais e mais complexo, aumentando o risco de liberação, e, com isso, o custo de novas modificações. Uma forma de mitigar esse problema, seria separar cada forma de pagamento em um serviço exclusivo.


1. MONOLITH
Users
Threads
Posts
node.js API Service
2. MICROSERVICES
Users
Users Service
Threads
Threads Service
Posts
Posts Service


Monolith
UI Layer
Business Logic
Data Layer


UI (frontend or mobile client)
Microservice
Microservice
Microservice

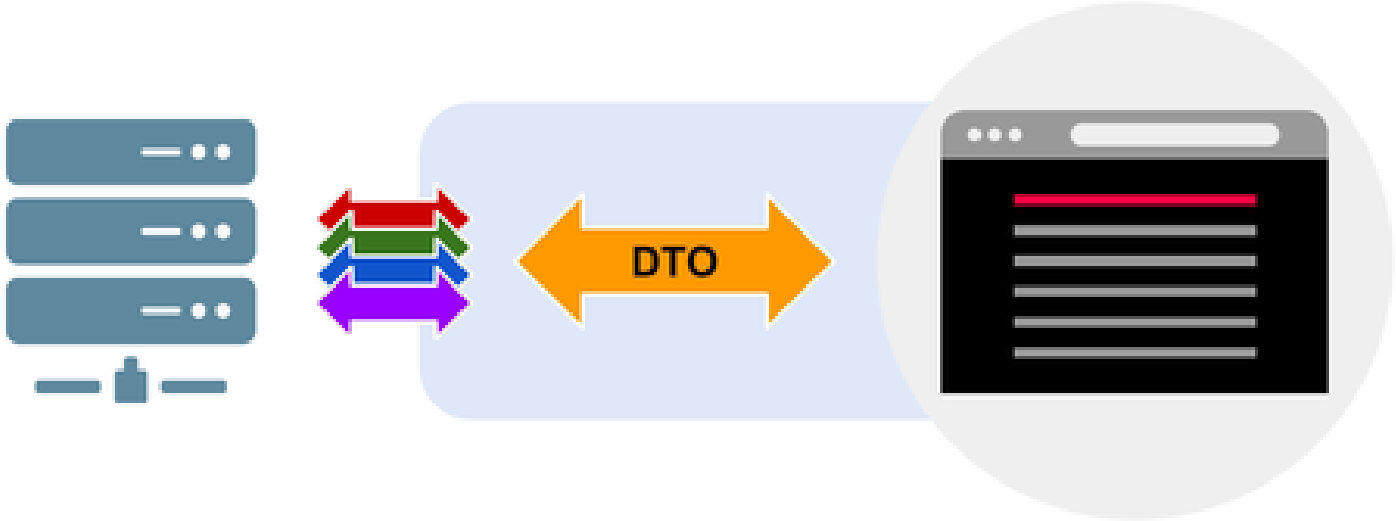
Dificilmente um arquiteto irá acertar de primeira. Requisitos mudam. Conforme aprendemos com nossas ferramentas de telemetria e feedback, a decomposição arquitetural de uma aplicação acaba sendo um processo natural dentro de uma aplicação moderna e evolutiva.

Felizmente, temos exemplos baseados na experimentação prática que nos servem de balizadores nessa empreitada:

- Garanta que seu serviço possui uma estrutura interna modular flexível;
- Avalie com calma os critérios que justifiquem uma decomposição arquitetural;
- Considere cada trade-off face às necessidades do seu contexto de negócio.

COTI INFORMÁTICA
ESCOLA DE NERDS

Utilizando o padrão
DTO Data Transfer Object



Data Transfer Object (DTO) ou simplesmente ***Transfer Object*** é um padrão bastante usado em Java para o transporte de dados entre diferentes componentes de um sistema, diferentes instâncias ou processos de um sistema distribuído ou diferentes sistemas via serialização.

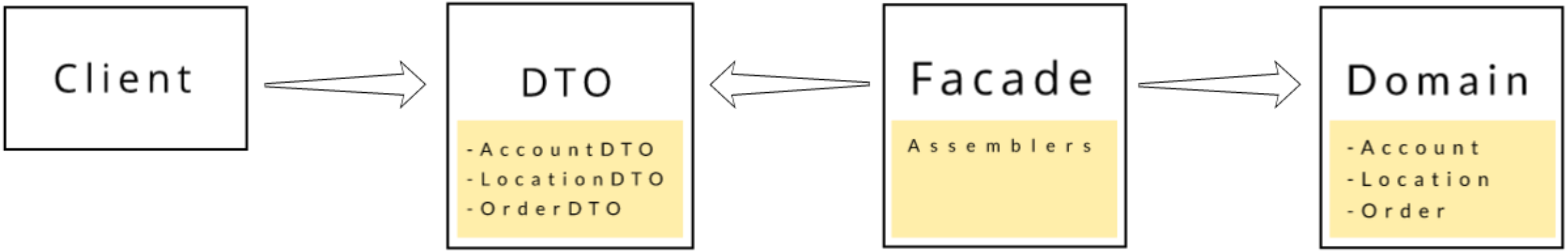
A ideia consiste basicamente em agrupar um conjunto de atributos numa classe simples de forma a otimizar a comunicação.

Numa chamada remota, seria ineficiente passar cada atributo individualmente. Da mesma forma seria ineficiente ou até causaria erros passar uma entidade mais complexa.

Além disso, muitas vezes os dados usados na comunicação não refletem exatamente os atributos do seu modelo. Então, um DTO seria uma classe que provê exatamente aquilo que é necessário para um determinado processo.

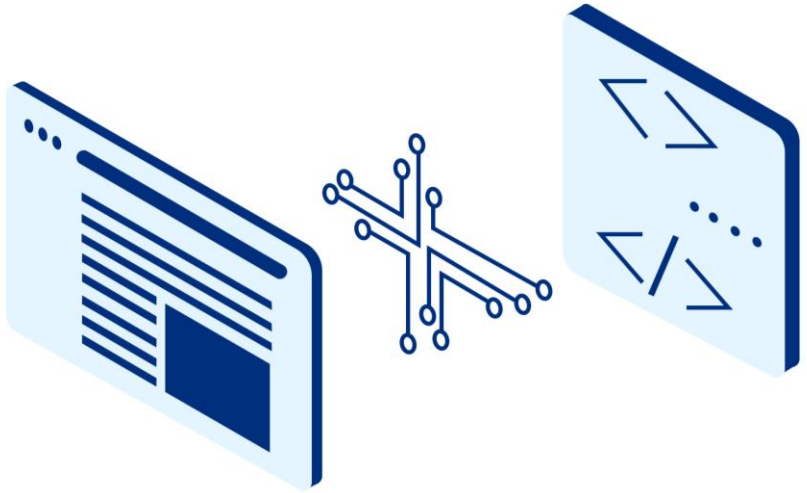
O padrão de projeto DTO é muito útil tanto para receber dados quanto para enviá-los, pois podemos manipular da forma que quisermos tais dados para facilitar a comunicação entre o servidor e o cliente.

Quando se está criando uma API é de extrema importância não apenas pensar como um usuário regular irá interagir, mas também se defender contra usuários maliciosos para que eles não consigam causar nenhum prejuízo para a sua aplicação e para seus usuários.



O padrão *DTO* busca otimizar a comunicação entre as camadas cliente e servidor da *API*, de modo que a Deserialização dos dados se faça por uma maneira simples concisa.

Com intuito de utilizar o padrão *DTO* para transferência de dados de uma forma desacoplada e de fácil interação no SpringBoot é interessante fazer uso do ModelMapper, que é um framework que realiza o mapeamento de modelos de forma simples e genérica.



COTI INFORMÁTICA
ESCOLA DE NERDS

Validando os dados da sua API com **BeanValidation**

Validar os valores preenchidos em campos de entrada de dados é extremamente importante em aplicações. O procedimento evita que armazenemos sujeira em nossas bases de dados e, dependendo do caso, pode até mesmo ter impacto na segurança do sistema.

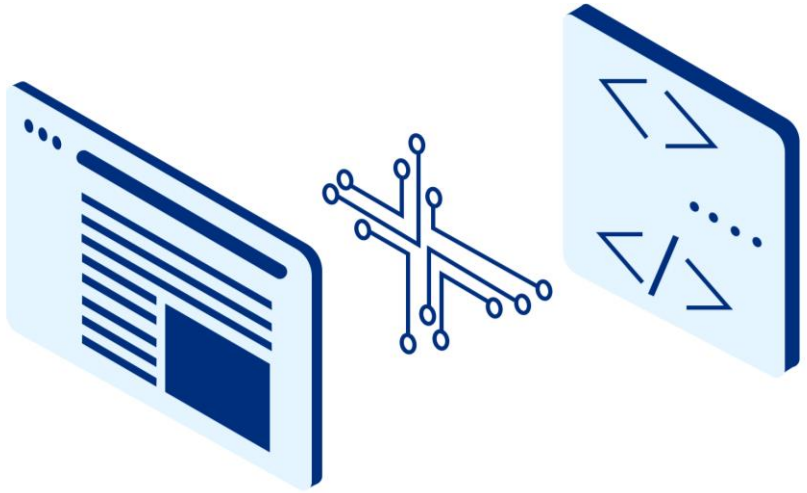
Um dos mecanismos que podemos utilizar para realizar esta verificação surgiu com a liberação da plataforma Java EE 6, na qual foi introduzida a especificação **Bean Validation**. O objetivo principal da biblioteca foi auxiliar os programadores nesta tarefa, que muitas vezes toma bastante tempo durante o desenvolvimento.

COTI INFORMÁTICA
ESCOLA DE NERDS

Utilizando **ModelMapper** para transferência de dados entre objetos.

O padrão *DTO* busca otimizar a comunicação entre as camadas cliente e servidor da *API*, de modo que a Deserialização dos dados se faça por uma maneira simples concisa.

Com intuito de utilizar o padrão *DTO* para transferência de dados de uma forma desacoplada e de fácil interação no SpringBoot é interessante fazer uso do ModelMapper, que é um framework que realiza o mapeamento de modelos de forma simples e genérica.



{ COTI INFORMÁTICA
ESCOLA DE NERDS }