Além dos comentários

/** *Criando Documentação de Valor em PHP* */

João Vitor Santos

@param $person

Quem sou eu?

Carreira e Formação

- </> Desenvolvedor PHP na IXCSoft
- Técnico em Redes de Computadores
- Graduado em Análise e Des. de Sistemas
- Atualmente cursando Ciência da Computação

Fora do código

- Mestre de Xadrez online (meu rating não sai de 600)
- Vibe-codo Apps (in)útéis nas horas vagas
- Pai de pet

O Problema: Quando o comentário atrapalha

```php
if ($cm[$ib]) > ($vl[$ib])) {//se o custo médio é maior que o valor de venda do item$this→dados['controla_estoque']
    $v = $cm[$ib] - $vl[$ib]; // diferença custo médio - valor de venda
    //contabiliza a débito a diferença na conta de despesa do grupo de produtos.
}
```

```php
if (/*...condição longa e complexa...*/)) {
    //aqui insere o valor com desconto ou valor total realmente ?
    $this→conta[$produto['id_conta_receita']]['credito'] += $movimento['valor_total'];
}
```

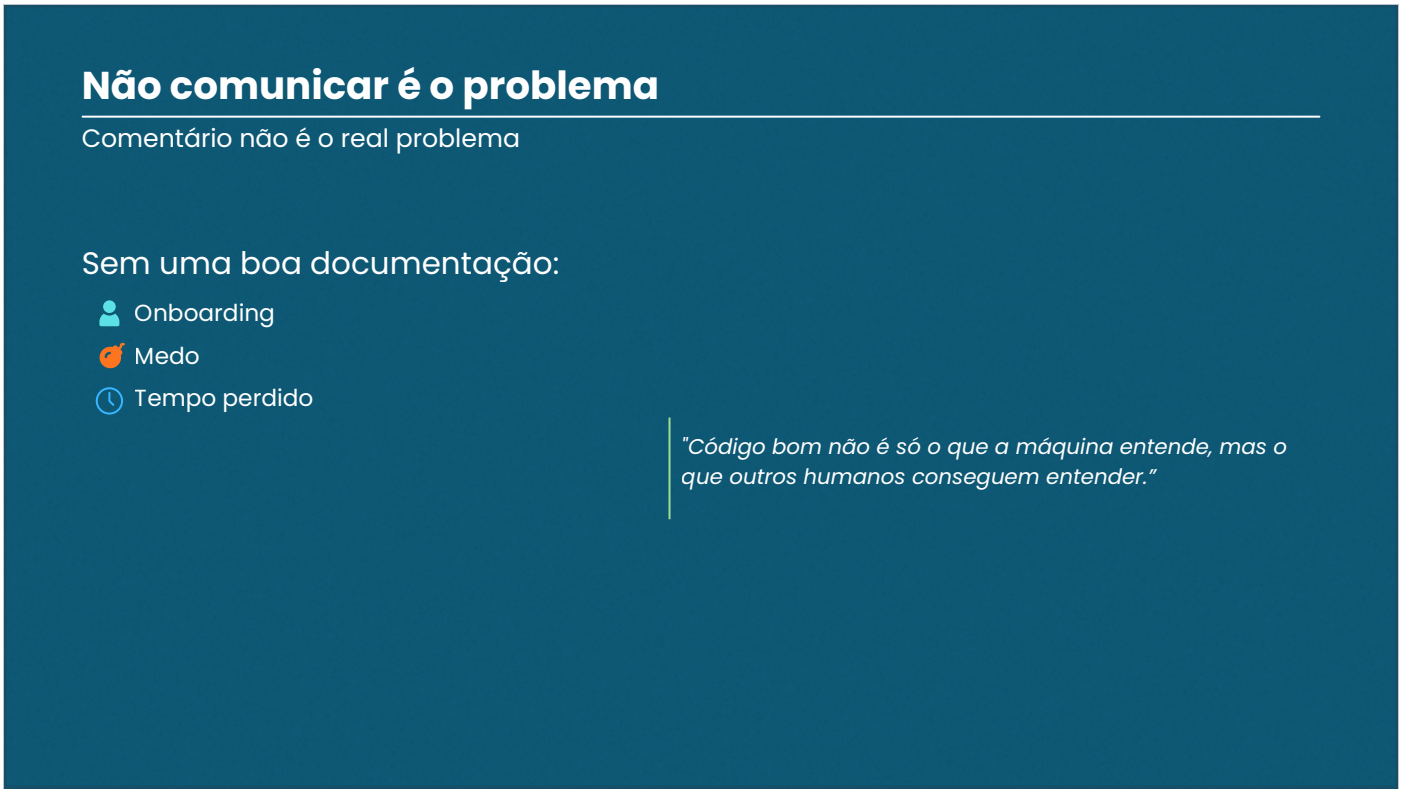
Temos aqui um problema: Quando o Comentário Atrapalha
Acho que todo mundo já se deparou com um código assim...

...temos um comentário que diz o quê o código faz (//diferença custo médio - valor de venda), mas não por quê ele faz isso. Por que precisa dessa contabilização?

E o meu favorito: o comentário-dúvida.

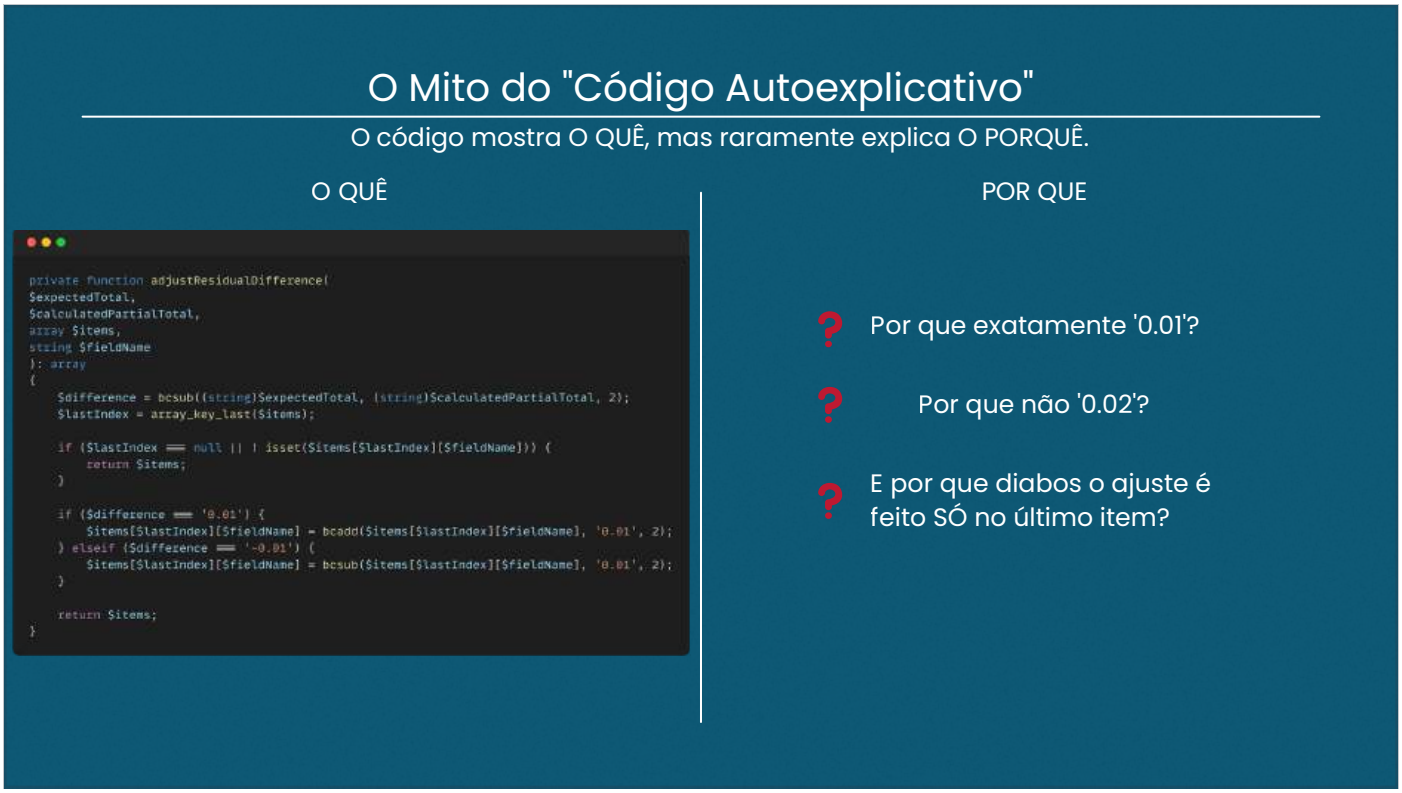
Isso não é documentação. É um pedido de ajuda.

É por isso que precisamos ir além dos comentários."



Eu sinceramente não me importo com comentários em códigos, porque o problema real não é escrever, é comunicar, e o que estava sendo comunicado com aquele comentário?

E a falta de uma boa documentação, causa:
O onboarding de novos desenvolvedores vai ser doloroso.
O medo de refatorar (“Se eu mexer aqui, com certeza vai quebrar”).
Tempo perdido debugando para entender a lógica por trás dessa regra.



O Tio Bob que me perdoe, mas eu acredito que é uma grande besteira quando ele diz que 'O código deve ser autoexplicativo'.

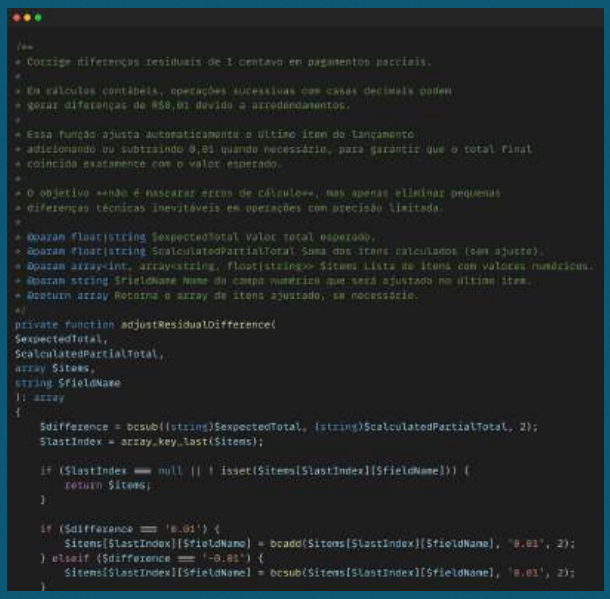
O código mostra O QUE ele faz, mas raramente explica O PORQUÊ ele faz daquele jeito.

Deem uma olhada nesse código. Esse código fui eu que fiz. Com algum contexto, a gente entende o que ele faz: ele pega uma diferença, provavelmente de centavos, e ajusta no último item de um array.

Mas... por que exatamente '0.01'? Por que não '0.02'? E por que o ajuste é feito SÓ no último item? Recentemente precisei dar manutenção nisso e... eu não fazia a menor ideia. Eu me esqueci.
E como essa regra foi discutida diretamente com a PO e acabou não sendo documentada em lugar nenhum, o porque se perdeu.

Agora, o Código Explica o "Porquê"

Isto é ir "Além dos Comentários"



```php
/**
 * Corrige diferenças residuais de 1 centavo em pagamentos parciais.
 *
 * Em cálculos contábeis, operações sucessivas com casas decimais podem
 * gerar diferenças de R$0,01 devido a arredondamentos.
 *
 * Essa função ajusta automaticamente o último item do lançamento
 * adicionando ou subtraindo 0,01 quando necessário, para garantir que o total final
 * coincida exatamente com o valor esperado.
 *
 * O objetivo **não é mascarar erros de cálculo**, mas apenas eliminar pequenas
 * diferenças técnicas inevitáveis em operações com precisão limitada.
 *
 * @param float|string $expectedTotal Valor total esperado.
 * @param float|string $calculatedPartialTotal Soma dos itens calculados (sem ajuste).
 * @param array<int, array<string, float|string>> $items Lista de itens com valores numéricos.
 * @param string $fieldName Nome do campo numérico que será ajustado no último item.
 * @return array Retorna o array de itens ajustado, se necessário.
 */
private function adjustResidualDifference(
$expectedTotal,
$calculatedPartialTotal,
array $items,
string $fieldName
): array
{
    $difference = bcsub((string)$expectedTotal, (string)$calculatedPartialTotal, 2);
    $lastIndex = array_key_last($items);

    if ($lastIndex === null || ! isset($items[$lastIndex][$fieldName])) {
        return $items;
    }

    if ($difference === '0.01') {
        $items[$lastIndex][$fieldName] = bcadd($items[$lastIndex][$fieldName], '0.01', 2);
    } elseif ($difference === '-0.01') {
        $items[$lastIndex][$fieldName] = bcsub($items[$lastIndex][$fieldName], '0.01', 2);
    }
```

❓ Por que '0.01'?

✅ "...diferenças de R$0,01 devido a arredondamentos em cálculos contábeis."

❓ Por que o último item?

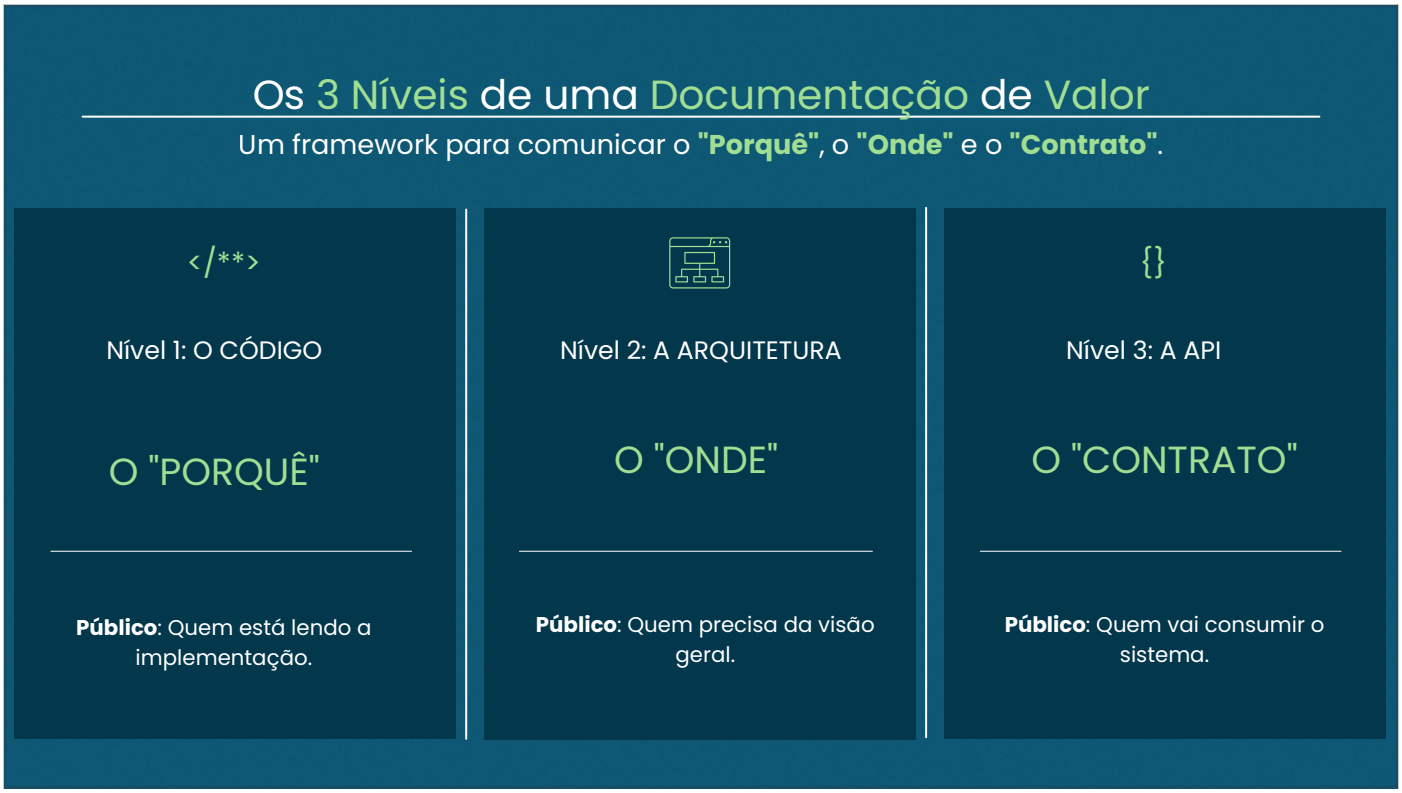
✅ "...para garantir que o total final coincida exatamente com o valor esperado."

❓ Qual o objetivo?

✅ "Eliminar pequenas diferenças técnicas... não mascarar erros de cálculo."

Antes tínhamos um código que nos deixou com várias perguntas. O 'porquê' estava completamente perdido.

Esse é o mesmo código, mas agora com uma documentação de verdade. Não um comentário, mas um docblock que explica o contexto, a intenção e o 'porquê'."
(Aponte para a lista de "Respostas" no lado direito)
"'Lembra da pergunta 'Por que 0.01'? Está aqui: é uma diferença residual de arredondamento contábil.'"
"'E 'Por que o último item?' Está aqui: é para ajustar o valor total e garantir que ele coincida com o esperado.'"
"'E o mais importante: qual o objetivo? Está aqui: é para lidar com diferenças técnicas, e não mascarar um erro de cálculo.'"
"O próximo desenvolvedor que pegar esse código – ou eu mesmo daqui a seis meses – agora tem 100% do contexto antes mesmo de ler a primeira linha da função. O 'porquê' foi comunicado."



Agora que vimos na prática a diferença entre um código que só mostra 'o quê' e um código que realmente explica o 'porquê'...

Eu esbocei um framework simples para a organizar o resto da conversa. Eu chamo de 'Os 3 Níveis de uma Documentação de Valor'.

A ideia central é que 'documentação' não é uma coisa só. Ela serve a públicos diferentes em momentos diferentes. E para ser útil, ela precisa comunicar a coisa certa para o público certo.

O Nível 1 é o CÓDIGO. É exatamente o que acabamos de ver nos exemplos. É a documentação que vive dentro da implementação, como os docblocks em PHP. O objetivo principal dela é explicar o 'PORQUÊ' das regras de negócio e das decisões técnicas. O público dela é quem está lendo a implementação: o próximo dev a dar manutenção, ou você mesmo daqui a seis meses.

Depois, damos um 'zoom out' para o Nível 2: a ARQUITETURA. Aqui, não estamos mais olhando para uma função, mas para o sistema. A documentação aqui explica 'ONDE' as coisas estão. Como os serviços se conectam? Qual é o fluxo de dados? Onde esse módulo se encaixa no projeto? O público aqui é quem precisa da visão geral: um novo membro na equipe, um arquiteto ou até o time de produto.

Por fim, temos o Nível 3: a API. Essa é a documentação 'para fora'. Ela não se importa com como seu sistema funciona por dentro. Ela define o 'CONTRATO'. Ela diz: 'Se você me enviar estes dados, eu garanto te devolver aquilo'. O público dela é qualquer um que vai consumir seu

sistema, seja um time interno, um parceiro externo ou um cliente.

Nível 1: Documentando o "**Porquê**" com PHPDoc

Vá além da sintaxe, capture a intenção

O que a IDE faz por você (Ruído)

```php
/**
 * @param float $valorDoPedido
 * @param string $estado
 * @return float
 */
function calcularFrete(float $valorDoPedido, string $estado): float
{
    $frete = self::VALOR_FIXO_FRETE;

    if ($valorDoPedido > self::VALOR_MINIMO_PEDIDO_ISENCAO_FRENTE
        && $estado === self::ESTADO_COM_FRETE_GRATIS
    ) {
        $frete = 0.00;
    }

    return $frete + self::TAXA_MANUSEIO;
}
```

? Qual a política de isenção de frete?

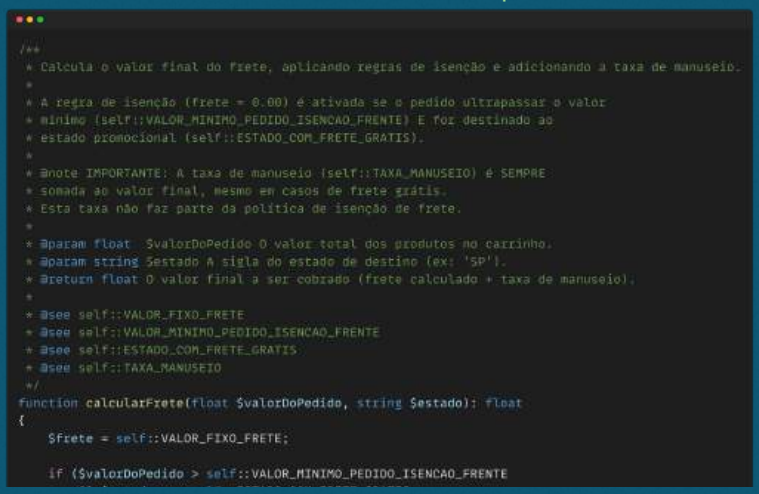
? De onde vêm essas constantes mágicas?

? A taxa de manuseio é somada no frete grátis?

Nível 1: Documentando o "**Porquê**" com PHPDoc

Vá além da sintaxe, capture a intenção

O Foco Deve Ser o "Porquê":



```php
/**
 * Calcula o valor final do frete, aplicando regras de isenção e adicionando a taxa de manuseio.
 *
 * A regra de isenção (frete = 0.00) é ativada se o pedido ultrapassar o valor
 * mínimo (self::VALOR_MINIMO_PEDIDO_ISENCAO_FRENTE) E for destinado ao
 * estado promocional (self::ESTADO_COM_FRETE_GRATIS).
 *
 * @note IMPORTANTE: A taxa de manuseio (self::TAXA_MANUSEIO) é SEMPRE
 * somada ao valor final, mesmo em casos de frete grátis.
 * Esta taxa não faz parte da política de isenção de frete.
 *
 * @param float  $valorDoPedido O valor total dos produtos no carrinho.
 * @param string $estado A sigla do estado de destino (ex: 'SP').
 * @return float O valor final a ser cobrado (frete calculado + taxa de manuseio).
 *
 * @see self::VALOR_FIXO_FRETE
 * @see self::VALOR_MINIMO_PEDIDO_ISENCAO_FRENTE
 * @see self::ESTADO_COM_FRETE_GRATIS
 * @see self::TAXA_MANUSEIO
 */
function calcularFrete(float $valorDoPedido, string $estado): float
{
    $frete = self::VALOR_FIXO_FRETE;

    if ($valorDoPedido > self::VALOR_MINIMO_PEDIDO_ISENCAO_FRENTE
```

✅ A isenção é ativada acima do valor X para o estado Y.

✅ As tags @see apontam para as constantes.

✅ SIM! O @note avisa que a taxa é sempre somada

Nível 2: Documentando o "Onde" (A Arquitetura)

Dando "zoom out" para a visão geral do sistema

1. O README.md

A "porta de entrada" do serviço

Seções Essenciais:

- **O que é?** (Propósito do serviço)
- **Como executar?** (Instalação e run)
- **Arquitetura / Dependências** (Com quem ele fala?)
- **Variáveis de Ambiente**

2. O Diagrama (Simples)

O "mapa" do fluxo de dados.

Exemplo:

[Usuário] → [API Gateway] → [Serviço de Frete]

Responde: "De onde vem a chamada e para onde ela vai?"

3. O ADR (Decisão)

O "porquê" da arquitetura.

Estrutura Essencial:

Título: (Ex: "Uso de RabbitMQ")
Contexto: (Qual era o problema?)
Decisão: (O que foi escolhido)
Justificativa: (Por que isso e não outro?)

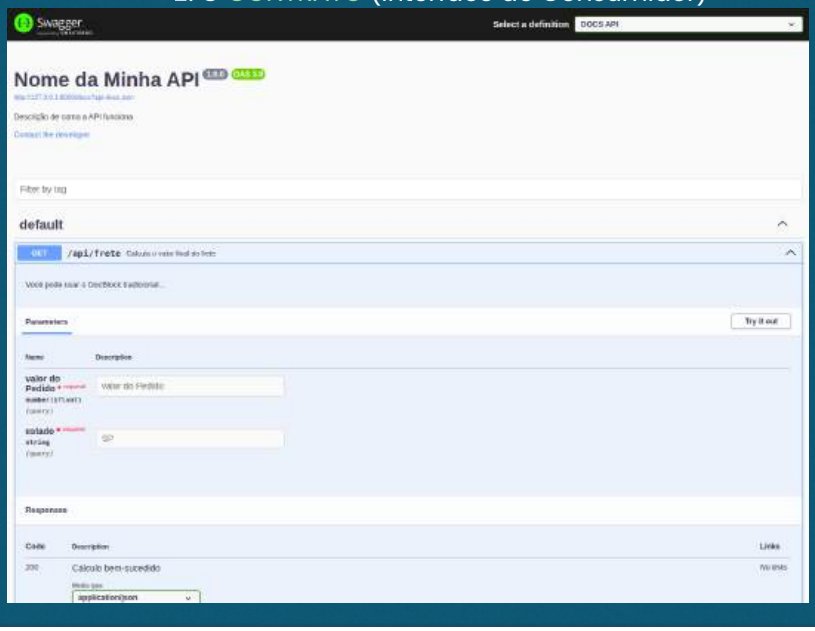
E chegamos ao Nível 3, o Contrato. O público aqui é o consumidor. A melhor forma de fazer isso é documentar direto no nosso código, na fonte da verdade. Usando bibliotecas como a swagger-php, nós anotamos nossos controllers...

e ao rodar um comando no terminal...

Nível 3: Documentando o "Contrato" (A API)

A documentação como "ponte" entre o Backend e o Consumidor.

2. O CONTRATO (Interface do Consumidor)



nós geramos automaticamente essa interface do Swagger UI. Uma documentação interativa, bonita, onde o dev frontend pode ver todos os endpoints, o que ele precisa enviar, e o que ele vai receber de volta.

Por que esta abordagem funciona

- **Fonte Única da Verdade:** A documentação vive junto com o código. Se o código muda, a documentação muda.
- **Fim da Desatualização:** Adeus ao "esqueci de atualizar o Postman / Wiki".
- **Interface Interativa:** O consumidor (Frontend, outro dev) pode testar a API diretamente no navegador.

Isso é um contrato que funciona. O backend e o frontend estão olhando para a mesma documentação, porque ela foi gerada do mesmo código.

Criando uma Cultura de Documentação

Não é sobre "fiscalizar", é sobre "colaborar"

✅

1. Na Definição de "Pronto"

Inclua "documentar o porquê" como critério de aceite da task. Se a regra de negócio não estiver clara, a task não está pronta.

💬

2. No Code Review

Use a documentação para guiar a revisão. A melhor pergunta não é "Faltou documentar", mas sim "Não entendi o porquê disso, podemos documentar?"

🎯

3. Comece Pequeno

Ninguém documenta um sistema legado inteiro em um dia. Escolha UMA parte crítica do sistema e melhore a documentação dela. Crie o hábito.

Resumo das Ideias-Chave

Nível 1: O CÓDIGO → Explica o **"PORQUÊ"**

Nível 2: A ARQUITETURA → Explica o **"ONDE"**

Nível 3: A API → Define o **"CONTRATO"**

Documente o "porquê", não o "o quê".

/** "A documentação é um presente para o seu 'eu' do futuro." */

Obrigado!

Perguntas?

@param $person **João Vitor Santos**

https://www.linkedin.com/in/joao-vitorss/