ChatGPT

Guia de Boas Práticas de Testes de Back-end (Node.js + TypeScript com Vitest)

Visão Geral dos Tipos de Teste (Unitário, Integração e E2E)

Em aplicações back-end, adotamos diferentes **tipos de testes** para garantir qualidade em vários níveis do sistema. Os principais são: **testes unitários**, **testes de integração** e **testes de ponta a ponta (E2E)**. Cada um tem propósito e granularidade distintas:

- **Testes Unitários** – Validam partes isoladas do código (funções, classes ou módulos individuais) sem dependências externas. São rápidos e confiáveis, facilitando identificar e corrigir bugs precocemente [1] [2]. Por exemplo, testar uma função de cálculo ou um método de serviço em isolamento. *Dica:* Use mocks/stubs para simular dependências (como BD ou serviços externos) e manter o teste independente e hermético (sem I/O real) [3] [4].

- **Testes de Integração** – Verificam a interação entre múltiplos componentes do sistema (por exemplo, um serviço chamando o banco de dados ou duas camadas da aplicação juntas) [5]. Aqui já exercitamos *infraestrutura real* sempre que possível – como a API REST rodando contra um banco de dados de teste – para cobrir contratos entre componentes. Esses testes ainda devem ser automatizados e razoavelmente rápidos, embora mais lentos que unitários. *Dica:* Ao testar uma API REST internamente, pode-se subir a aplicação em modo de teste e usar ferramentas como **Supertest** ou **axios** para fazer requisições HTTP em memória, validando respostas JSON, código de status etc., sem precisar de servidor externo.

- **Testes End-to-End (E2E)** – Simulam cenários completos do ponto de vista do usuário final ou de sistemas externos, exercitando **todo o fluxo** da aplicação (da requisição à persistência) [6]. Por exemplo, realizar uma chamada HTTP real aos endpoints REST em um ambiente completo, verificando desde a autenticação até a resposta final. Esses testes dão maior confiança por reproduzirem usos reais, porém tendem a ser os mais **lentos e sujeitos a instabilidades (flakiness)** se não bem controlados [7]. Deve-se usá-los com moderação para validar trajetórias críticas, enquanto a maioria dos cenários é coberta por testes de nível inferior.

➤ **Pirâmide de testes:** Grandes empresas como a Google recomendam uma proporção aproximada de **70/20/10** entre unitários, integração e E2E, respectivamente [8]. Em outras palavras, a maior parte dos testes deve ser de baixa granularidade (unitários), uma quantidade média de testes de integração, e poucos testes end-to-end completos. Essa distribuição (conhecida como **Pirâmide de Testes**) garante feedback rápido e isolamento nos testes menores, sem abrir mão da verificação de fluxo completo nos testes maiores [9]. Evita-se assim a "pirâmide invertida" (excesso de E2E lentos) ou o "formato ampulheta" (muitos unitários e E2E, mas falta de integração) [10].

> *Exemplo:* Em um projeto Node/TS típico, podemos ter dezenas de testes unitários para cada módulo (rodando em milissegundos), alguns testes de integração cobrindo a API REST com

banco de teste, e apenas 1 ou 2 testes E2E cobrindo um fluxo de uso completo. Isso segue a filosofia do Google, onde **70%** dos testes são unitários, **20%** de integração e **10%** E2E [8].

Além da quantidade, vale destacar a diferença de **papel**: *"Testes pequenos garantem a qualidade do código; testes maiores garantem a qualidade do produto"* [11]. Os unitários fortalecem cada componente internamente, enquanto integrações e E2E asseguram que o sistema como um todo atende os requisitos do usuário final. Ambos são necessários para uma estratégia completa de qualidade.

➤ **Independência e confiabilidade:** Todo teste deve ser **determinístico** e independente dos outros. No Google, uma regra fundamental é que testes não devem depender de estado gerado por outros testes e não devem deixar efeitos colaterais persistentes no sistema [12]. Isso significa que podemos rodar os testes em qualquer ordem (ou em paralelo) obtendo sempre o mesmo resultado. Para atingir essa independência, isole cada caso de teste: limpe ou restaure estados compartilhados (como o banco de dados, arquivos ou configurações) e evite compartilhar dados entre testes. Se um teste falhar, deve ser claramente por um problema no código em teste, não porque outro teste poluiu o ambiente.

➤ **Rapidez do feedback:** Testes automatizados só trazem produtividade se puderem rodar com rapidez. Idealmente desenvolvedores executam a suíte com frequência (em desenvolvimento e nos pipelines CI) sem perder tempo excessivo. Esforce-se para que testes unitários rodem em milissegundos (o próprio Google considera ~100ms um limite aceitável por teste unitário [13]) e mesmo integrações/E2E devem ser otimizados (p. ex., Google sugere que mesmo testes maiores nunca ultrapassem alguns minutos cada [13]). Testes rápidos incentivam a execução contínua e evitam que correções se acumulem sem validação [14].

Testes Unitários: Boas Práticas 🔍

Para testes **unitários** (focados em funções/módulos individuais), siga estas diretrizes:

- **Isolamento total de dependências:** Use mocks, stubs ou fakes para representar componentes externos (como chamadas HTTP, consultas de banco ou caches). Com Vitest, podemos usar `vi.fn()` e `vi.mock()` para criar funções simuladas e mockar módulos facilmente [3] [4]. Isso garante que o teste avalie apenas a lógica interna da unidade, sem fatores externos influenciando. *Exemplo:* simular a resposta de uma função de busca no BD em vez de acessar um BD de verdade, controlando cenários de retorno (vazio, com dados, erro) via `mockResolvedValue` / `mockRejectedValue`.

- **Rapidez e escopo reduzido:** Um teste unitário deve cobrir uma funcionalidade pequena em escopo e executar muito rápido. Evite I/O real (arquivo, rede, DB) – foque em cálculos, validações, transformação de dados, regras de negócio isoladas. Se o teste está lento, verifique se não está acessando algo externo indevidamente. Mantendo os testes unitários rápidos, podemos rodá-los centenas de vezes ao dia durante o desenvolvimento.

- **Estrutura AAA (Arrange-Act-Assert):** Siga o padrão **Arrange (Preparação) – Act (Ação) – Assert (Verificação)** dentro de cada teste [15]. Separe claramente: primeiro configurar o estado ou inputs necessários, em seguida executar a função/unidade sob teste, e por último verificar resultados (assertivas). Isso torna o teste mais legível e evita misturar etapas. *Exemplo:* "Dado um usuário com

papel X (arrange), quando chamar a função de acesso Y (act), então deve retornar verdadeiro (assert)".

- **Nomeação clara dos testes:** Dê nomes descritivos aos casos de teste, indicando o cenário e o resultado esperado. Uma boa prática é responder nas descrições: *o que está sendo testado*, *em que condições* e *qual o resultado esperado*. Por exemplo: `it("deve rejeitar login com senha incorreta", ...)` já indica cenário de autenticação falhando [16]. Nomes claros ajudam a entender falhas no relatório rapidamente e servem como documentação de comportamentos esperados.

- **Não duplique lógica nos testes:** Os testes devem validar a lógica do código-fonte, não replicá-la. Evite escrever código complexo dentro do teste – mantenha-os o mais simples possível [17]. Se muita lógica se repete em vários testes (por ex., criação de objetos complexos), considere usar funções helper ou *fixtures*, mas com parcimônia para que cada teste continue fácil de ler. Lembre-se da "regra de ouro": *código de teste não é código de produção* [17] – priorize clareza em vez de abstrações excessivas nos testes.

- **Cobertura de cenários e casos-limite:** Para cada unidade, escreva testes que cubram caminhos felizes (inputs válidos) e também condições de erro ou extremas. Por exemplo, teste função de cálculo com valores positivos, zero, negativos; funções de string com string vazia, muito longa, caracteres especiais, etc. Isso aumenta a confiança de que a unidade se comporta corretamente em qualquer situação razoável.

- **Use spies/stubs para verificar efeitos colaterais:** Quando a função testada chama outra (ex: função de log ou envio de email), utilize *spies* para inspecionar se essas chamadas ocorreram corretamente. O Vitest permite espionar funções facilmente, por exemplo `const spy = vi.spyOn(moduloX, 'metodoY')`. Assim, após executar o código (Act), você pode `expect(spy).toHaveBeenCalledWith(...)` para garantir que a interação ocorreu como esperado. Prefira essa abordagem em vez de realmente executar a ação externa. Ferramentas como **Sinon.JS** também oferecem stubs/spies tradicionais caso necessário [18], mas muitas vezes as utilidades nativas do Vitest bastam.

- **Teste resultados e comportamentos, não implementações internas:** Foque em verificar o *comportamento público* da unidade. Evite testar estados internos privados ou detalhes de implementação que possam mudar – isso tornaria o teste frágil a refatorações. Por exemplo, em vez de testar se uma função usou exatamente um determinado algoritmo, teste o resultado final produzido. (Se precisar muito testar algo "privado", talvez isso indique que essa lógica deveria ser extraída para outra função pública testável).

Testes de Integração: Boas Práticas

Os **testes de integração** verificam se as partes do sistema funcionam corretamente em conjunto – por exemplo, um teste que chama a API REST da aplicação e verifica se ela interage corretamente com o banco de dados e retorna a resposta esperada. Diretrizes para esses testes:

- **Ambiente o mais próximo do real:** Sempre que viável, use componentes reais do stack no teste de integração. Isso significa rodar a aplicação (ou ao menos a parte necessária, como o servidor HTTP) e conectar a um **banco de dados de teste real** (ou equivalente), em vez de mockar tudo. Assim, você valida configurações de ORM, queries SQL, serialização JSON, etc. – comportamentos que um mock não pegaria. *Exemplo:* subir a API Express/Fastify em modo de teste e usar `supertest` (ou `axios` apontando para `http://localhost:` porta de teste) para fazer requisições às rotas, validando não só o código mas também a integração com o roteamento, middleware, banco etc.

- **Isolamento dos dados (estratégia de BD):** É crucial garantir que cada teste de integração tenha **controle sobre os dados** que usa, para não interferir em outros testes. Há diversas abordagens possíveis:

- **Banco dedicado por teste:** Uma opção robusta é usar um banco de dados isolado para cada execução de teste (por exemplo, cada teste usa um schema ou banco distinto, ou utiliza instâncias em memória). Isso previne contaminação de dados – um teste não "vê" os registros inseridos por outro [19] [20] . Embora possa aumentar a complexidade (configurar múltiplos bancos) ou uso de recursos, essa abordagem elimina flakiness por concorrência de dados [21] .
- **Limpeza de dados consistente:** Alternativamente, usar um **esquema de limpeza** no ciclo de vida de testes. As duas estratégias comuns são limpar o banco **após cada teste** ou **ao final de todos os testes** [20] .
    - *Limpeza após cada teste:* Garante que cada teste comece com o BD vazio, evitando interferência. Porém, pode ser custosa e difícil em execução paralela – por exemplo, se testes rodam simultaneamente, um teste pode apagar dados enquanto outro está usando, causando falhas intermitentes [22] . Se optar por essa via, considere rodar testes de integração de forma sequencial (desabilitar paralelismo) ou atribuir bancos separados por processo.
    - *Limpeza após toda a suíte:* Os testes reaproveitam o estado do banco ao longo da execução, apenas limpando no final (ou em intervalos definidos) [20] . Isso permite rodar em paralelo sem que um apague o que o outro inseriu, desde que **cada teste manipule somente dados que ele mesmo criou** (não presumindo banco vazio) [23] . Essa abordagem simula um cenário mais real, com banco populado, e pode até revelar bugs que só aparecem com muitos registros [24] . A desvantagem é que o teste precisa filtrar/buscar especificamente os registros que inseriu (não pode, por exemplo, contar "deve haver 1 registro na tabela" sem qualificar por ID, já que outros testes podem ter adicionado outros registros). Também requer atenção para campos únicos – inserir valores fixos repetidamente causaria conflitos entre testes; a solução é usar valores únicos (ex: adicionar sufixo aleatório em strings únicas) [25] .
- **Não limpar em caso de falha:** Independentemente da estratégia, **não limpe os dados imediatamente quando um teste falha** [26] . Manter o estado final daquele teste ajuda a inspecionar e debuggar o problema. Em ambiente de desenvolvimento, pode-se até pausar a limpeza para investigar manualmente os dados residuais.

Recomendação: Em projetos menores, limpar após cada teste (ou usar um BD em memória renovado a cada teste) pode ser simples e eficaz. Em projetos maiores com muitos testes em paralelo, a estratégia de **dados persistentes durante a suíte** tende a ser mais performática e realista (desde que combinada com boas práticas como usar IDs únicos e não depender de tabelas vazias) [23] [27]. Escolha uma estratégia clara e documente-a para toda equipe seguir de forma consistente.

- **Uso de Banco de Dados real vs. simulado:** Para integração, prefira usar o **mesmo SGBD da produção** (ou o mais próximo possível) em vez de simular. Por exemplo, se a aplicação usa PostgreSQL, idealmente rodar um contêiner PostgreSQL para testes. Ferramentas como **Docker Compose** ajudam a levantar o BD (e outros serviços, ex: fila) no ambiente de teste de maneira replicável [28]. Isso evita problemas de discrepâncias de dialeto SQL, recursos específicos, etc. Caso não seja viável (ou para desenvolver localmente com mais agilidade), pode-se usar um **banco em memória**: alguns bancos oferecem modo in-memory (ex: SQLite em memória, ou **MongoDB Memory Server** para Mongo). Bancos em memória são rápidos e isolados, mas atenção à compatibilidade: ex., SQLite pode não suportar tudo que um Postgres suporta. Use-os com cautela e preferencialmente apenas se forem compatíveis o bastante com sua camada de acesso a dados [29].

- **Dados de teste bem definidos:** Controle quais dados estão presentes no banco antes de rodar cada teste:

- Evite depender de dados residuais de outros testes (novamente, isolamento!).
- **Seed mínimo necessário:** Se for preciso pré-carregar dados no BD (ex.: listas estáticas ou referência), faça isso de forma centralizada *uma vez* antes da suíte de testes (ex: seeding de metadados como países, tipos, etc., que todos os testes usarão) [30] [31]. Mas mantenha esse seed enxuto – apenas dados que **são realmente independentes dos cenários de teste**. Cada teste deve inserir/manipular os dados específicos do cenário que está validando (por exemplo, criar o usuário e pedido necessários para testar o fluxo de compra) [32] [33]. Assim, a leitura do teste fica clara, pois todos os registros relevantes ao caso estão definidos ali, e evitamos efeitos colaterais entre testes.

- Use **dados realistas** nos testes. Por exemplo, em vez de usar strings “foo” ou números mágicos, pode-se usar valores plausíveis (ex.: nomes de usuário, e-mails, datas válidas). Isso torna os testes mais compreensíveis e pode revelar problemas com caracteres especiais, formatos, etc. Ferramentas como Faker podem ajudar a gerar dados falsos porém realistas (ex.: CPF, nomes, etc.) [34]. Mas cuidado para não introduzir aleatoriedade não determinística – se usar Faker/Chance, fixe seeds ou use valores constantes para que o teste tenha resultado reprodutível.

- **Cobertura de integração típica:** Foque em casos como:

- **Consultas ao banco:** Testar que buscar um recurso pela API realmente traz os dados corretos do BD. Inclua cenários de dado existente (200 OK com conteúdo) e não existente (404 ou resposta vazia conforme design).
- **Criação/atualização de dados:** Testar via API que inserir ou editar um recurso persiste as mudanças no banco. Após um POST/PUT bem sucedido, faça um GET no mesmo recurso para confirmar que os dados estão lá e consistentes [35] [36]. *Boa prática:* verificar estado novo preferencialmente através da **própria API pública** (GET) em vez de ler direto do BD [35] – assim o teste garante que tanto a gravação quanto a leitura (endpoint) estão corretos e que o contrato da API realmente reflete o estado do banco.

- **Transações e consistência:** Se sua lógica envolve múltiplas etapas (ex.: criar pedido deve diminuir estoque e gerar log), o teste de integração deve verificar os efeitos colaterais esperados em cada parte (multiplos agregados modificados, eventos disparados, etc.). Use o escopo completo da aplicação no teste – por exemplo, confirme que após chamar a rota de criar pedido, o registro de pedido existe *e* o estoque do produto foi decrementado.

- **Erros integrados:** Induza falhas para ver se são tratadas corretamente – por exemplo, requisitar algo com ID inválido (esperar erro 400), causar violação de constraint no banco (esperar código 500 ou tratamento definido), simular indisponibilidade do serviço externo (talvez mockar a chamada externa para lançar erro e ver se a API retorna 502, por exemplo).

- **Ferramentas úteis:**

- **Supertest** (ou similar) facilita fazer requisições HTTP ao app Express/Fastify em modo de teste, sem precisar de servidor real escutando porta.
- **Vitest** suporta testes assíncronos com `async/await`, então você pode `await supertest(app).get('/minha/rota')` e fazer assertions na resposta.
- **Node-mocks-http** pode ser útil para testar camadas como middleware ou controllers isoladamente: você fornece req/res fictícios e verifica como a função os manipula [37] [38]. Útil quando não se quer levantar todo o servidor, mas quer testar integração parcial (ex: middleware de parsing).
- **Docker Compose** para orquestrar serviços de apoio (BD, cache, filas) durante testes [28]. Por exemplo, um `docker-compose.test.yml` que sobe um banco de dados configurado para testes (com menos durabilidade para performance, por ex.). **Dica:** configure o BD de teste com parâmetros de performance (desativar fsync, journaling, etc.) se isso não afetar a lógica – pode acelerar muito sem prejudicar a validade do teste [39] [40].
- **Teste de contrato**: Considere testar também o contrato da API – por exemplo, validar que o JSON de resposta contém os campos esperados e tipos corretos (especialmente campos gerados dinâmicamente, como timestamps ou IDs). Você pode usar validação por JSON Schema ou OpenAPI nos testes [41], ou snapshots de resposta se o formato for estável. Grandes empresas investem em **contract tests** para garantir que mudanças não quebrem consumidores – pode ser algo a incluir se sua API for usada por terceiros.

Testes de Autenticação e Autorização

A segurança da API – autenticação de usuários e autorização de acesso – deve ser rigorosamente testada. Como fazer isso:

- **Fluxo de Login (Autenticação):** Teste o endpoint de login/autenticação de forma abrangente:
- Cenário de sucesso: dado credenciais válidas de um usuário de teste, a API deve autenticar e retornar o token (JWT, por exemplo) ou cookie de sessão esperado, junto com o código 200/201 e payload correto (ex.: dados do usuário logado) – verifique estes elementos. *Dica:* Use um usuário fixo de teste no banco (ou crie um usuário novo no arrange do teste) com senha conhecida. A senha pode ser um hash calculado via a mesma função usada em produção para garantir consistência.
- Cenário de falha: tente logar com senha incorreta, usuário inexistente, ou outros casos inválidos e espere o comportamento correto (normalmente, resposta 401 Unauthorized ou uma mensagem de erro genérica para não revelar detalhes). Verifique que nenhum token é emitido e que a mensagem de erro segue o padrão combinado.

- Controles adicionais: se há limites de tentativa ou CAPTCHA após falhas sucessivas, seria interessante simular múltiplas requisições com senha errada e ver se o API responde com bloqueio após o limite (se implementado). Esses cenários garantem que proteções de força bruta funcionam.

- **Testes de JWT e Tokens:** Se a autenticação é via JWT, teste a mecânica de geração e verificação de tokens:

- **Emissão de token:** No login bem-sucedido, valide que o token JWT retornado é válido – por exemplo, use a mesma biblioteca JWT do servidor no teste para decodificar o token e conferir claims (p. ex., contém o ID do usuário esperado, expiração, etc.). *Obs:* Configure o mesmo segredo de JWT no ambiente de teste para poder verificar o token. **Não use segredos de produção**; tenha um segredo de teste (pode ser um valor fixo como `"test_jwt_secret"` definido via variável de ambiente em modo test).
- **Token inválido/expirado:** Simule chamadas autenticadas com tokens inválidos. Por exemplo, chame um endpoint protegido sem o header `Authorization` – deve retornar 401 [42]. Faça o mesmo com um token JWT malformado ou com assinatura inválida (ex.: altere alguns caracteres do token) – a API deve rejeitar com 401. Se aplicável, teste um token **expirado** (pode criar um JWT com exp passado ou manipular a data no servidor durante o teste) e verifique que é tratado como não autorizado.

- **Refresh token (se houver):** Caso sua arquitetura use refresh tokens, teste o fluxo de refresh – trocar um refresh válido por novo JWT (sucesso) e usar refresh inválido/expirado (deve recusar).

- **Acesso Autorizado vs. Proibido (Authorization):** Para cada nível de permissão em sua aplicação, escreva testes garantindo que as restrições estão corretas:

- **Usuário não autenticado:** Verifique que todas as rotas que exigem login retornam **401 Unauthorized** quando acessadas sem credenciais. Faça chamadas sem token ou sem cookie de sessão e espere esse código. Isso garante que seu middleware de auth está ativo em todas as rotas sensíveis.
- **Usuário comum vs. administrador:** Se há papéis, crie (ou use) um usuário de teste para cada papel. Por exemplo, com um JWT do usuário comum, tente acessar uma rota admin-only – a resposta deve ser **403 Forbidden** (ou 404 ocultando a existência) conforme o design de segurança. Com um JWT de admin, a mesma rota deve responder 200 e executar a ação. Esses testes certificam que a autorização por papéis está implementada.
- **Escopo de dados:** Se a autorização envolve escopo (ex.: um usuário só pode acessar seus próprios recursos), simule cenários cruzados. Por exemplo, usuário A tentando acessar o recurso do usuário B deve receber 403/404, enquanto acessar o seu próprio retorna 200 e dados corretos.

- **Middleware de autorização isolado:** Você pode testar a lógica de middleware de forma unitária também. Usando um stub de `req` e `res` (com algo como node-mocks-http), injete diferentes usuários/roles e verifique se o middleware chama `next()` ou gera resposta de erro conforme esperado. Isso complementa os testes de integração cobrindo casos específicos de autorização sem precisar orquestrar toda a rota.

- **Sessões e Cookies:** Se sua API usa sessões via cookie (em vez de JWT), considere:

- Simular login para obter o cookie de sessão (algumas libs de teste suportam manter cookies, ex. **supertest** tem `.agent()` que retém cookies entre requisições). Depois usar esse cookie em chamadas subsequentes para ver rotas autenticadas funcionando.
- Testar que sem o cookie ou com cookie inválido a rota é barrada (similar ao JWT case).

- Opcional: testar cookie com flag de segurança (HttpOnly, Secure) – geralmente não acessível via XHR, mas se sua aplicação depende disso, verifique nas respostas HTTP se o `Set-Cookie` tem os atributos corretos (pode usar supertest `.expect('set-cookie', /HttpOnly/)` etc.).

- **Fluxos de logout/revogação:** Se aplicável, teste o logout: chamar o endpoint de logout deve invalidar o token ou sessão (talvez limpando cookie). Verifique que após logout o token/cookie anterior não dá mais acesso (ex.: requisição com ele retorna 401).

- **Ferramentas e Convenções para Auth em testes:**

- Mantenha dados de usuários de teste separados dos reais. Popule usuários de teste no banco de teste com senhas conhecidas (hashes predefinidos). Uma boa prática é ter um seed de usuários padrão para testes de auth (ex.: user "alice" com senha "123456").
- Nunca codifique senhas reais ou segredos reais nos testes – use somente valores de desenvolvimento.
- Se usar OAuth externo (Google, Facebook login): em ambiente de teste pode ser impraticável acionar o provedor real. Solução: isolar a integração OAuth atrás de uma interface, e **mockar** essa interface nos testes. Por exemplo, se seu código chama uma API do Google para validar token, substitua por um stub que simplesmente valida localmente para fins de teste. Ou utilize ambientes de sandbox/teste dos provedores se disponíveis, mas normalmente mocks são mais estáveis.
- Teste também comportamentos de segurança: por exemplo, muitas tentativas de login erradas – seu sistema bloqueia o IP ou avisa? Se sim, escreva testes simulando múltiplas requisições (pode usar um loop) e verifique o resultado após N tentativas.

Em suma, **garanta que somente usuários autorizados obtêm acesso** e que credenciais inválidas são sempre rejeitadas de forma adequada. Esses testes são vitais para prevenir regressões de segurança.

Cobertura de Código e Estruturação dos Testes 📊

Manter um bom nível de **cobertura de código** ajuda a monitorar quais partes do sistema estão sendo exercitadas pelos testes. Porém, cobertura por si só não garante qualidade – é possível ter 100% de linhas cobertas com testes fracos. Considere as seguintes práticas:

- **Configure relatórios de cobertura:** Utilize a ferramenta integrada do Vitest para gerar relatório de cobertura. No `vitest.config.ts`, ative os reports (texto, HTML) [43]. Por exemplo:

```
// vitest.config.ts
export default defineConfig({
  test: {
    coverage: { reporter: ['text', 'html'] }
```

```
  }
})
```

Execute os testes com `vitest --coverage` para produzir os dados [44]. Isso gerará um resumo no console e arquivos HTML detalhados com linhas cobertas ou não. Inclua esses relatórios no processo de CI para acompanhamento contínuo. **Dica:** Empresas maduras sempre analisam cobertura nos pipelines – é um indicador importante para identificar áreas não testadas [45].

- **Defina metas de cobertura inteligentes:** Um alvo comum é, por exemplo, *80% de cobertura* no mínimo. Entretanto, não foque apenas no número global – observe a **cobertura por módulo** e, principalmente, cubra código crítico (ex.: regras de negócio, cálculos financeiros, segurança) perto de 100%. Cobertura de 100% em tudo nem sempre é custo-benefício, mas **áreas sensíveis ou complexas devem ter cobertura máxima**. Ferramentas como o relatório do coverage mostram quais arquivos/linhas estão descobertos; use isso para dirigir novos testes se perceber lógicas importantes sem teste.

- **Cobertura de ramos (branches):** Além de linhas executadas, tente alcançar boa cobertura de *branches* lógicos (condicionais *if/else*, *switch*, tratamento de erros). Por exemplo, se há um `if` que em caso de erro lança exceção, certifique-se de ter um teste que provoca esse erro para cobrir o ramo excepcional. Isso evita falsos sentimentos de segurança onde uma linha pode ter sido executada mas um ramo alternativo nunca foi testado.

- **Estrutura e organização dos arquivos de teste:** Organize a base de código de testes de forma análoga ao código fonte para fácil localização. Algumas opções:

- Colocar os testes ao lado dos arquivos fonte (ex.: `src/calculadora.ts` e `src/calculadora.test.ts` juntos) – facilita navegar entre código e teste.
- Ou manter em diretórios separados espelhando a estrutura (ex.: `tests/unit/...` e `tests/integration/...`). Ambas abordagens são válidas, escolha uma e seja consistente.
- Nomeie arquivos de teste com sufixo `.test.ts` ou `.spec.ts` para que o Vitest os reconheça automaticamente. Por convenção, use inglês ou português de forma padronizada nos nomes (ex.: `usuario.service.test.ts`).

- Separe testes de unidade, integração e E2E em pastas ou pelo nome dos arquivos, se você planeja executá-los separadamente. Por exemplo, pode marcar testes de integração com um padrão no nome (`*.int.test.ts`) e configurar scripts npm ou o Vitest para rodá-los isoladamente quando preciso.

- **Uso de Describe para estruturação:** Utilize blocos `describe` para agrupar testes relacionados por tema, módulo ou funcionalidade [46]. Por exemplo:

```
describe('Serviço de Usuários', () => {
  describe('login()', () => {
    it('deve autenticar com credenciais válidas', ...);
    it('deve falhar com senha inválida', ...);
  });
```

```
  describe('registro()', () => { ... });
});
```

Isso melhora a organização e legibilidade na saída (os nomes aninhados aparecem juntos). Além disso, permite factorar um setup/teardown específico por grupo usando `beforeEach/afterEach` dentro do describe.

- **Hooks de configuração (before/after):** Utilize `beforeAll` / `afterAll` para operações de setup global (por exemplo, iniciar um servidor antes dos testes de integração e derrubá-lo ao final, ou fazer seed de dados estáticos) [47] [48]. Use `beforeEach` / `afterEach` para isolamento entre casos (por exemplo, resetar mocks, limpar alguma coleção em memória, restaurar configurações modificadas durante o teste). Esses hooks ajudam a evitar repetição, mas cuidado para não esconder demais a lógica de preparação – quem ler o teste deve entender facilmente o contexto.

- **Testes determinísticos e reentrantes:** Os testes devem produzir o mesmo resultado em cada execução, independente do ambiente ou da ordem. Evite dependências de horários reais (use datas fixas ou fake timers se necessário), de aleatoriedade pura (se precisar de random, fixe seed ou verifique propriedades ao invés de valores exatos), e não crie dependência de ordem de execução (por exemplo, teste A insere algo que teste B usa – isso é mau design). Uma prática é rodar a suíte em ordem aleatória ocasionalmente para detectar dependências ocultas. Vitest suporta rodar testes aleatoriamente se configurado, ou você pode simplesmente embaralhar via CLI, mas garantindo independência isso não deve importar.

- **Manutenção dos testes:** Trate o código de teste como cidadão de primeira classe no projeto. Refatore testes quando refatorar código, garanta que nomes de variáveis e asserts sejam atualizados conforme a lógica muda. Remova ou ajuste testes obsoletos (um teste falhando porque o requisito mudou deve ser atualizado para o novo comportamento esperado, não simplesmente deixado de lado). Revisões de código devem cobrir também os testes: incentive o time a revisar se os testes realmente estão cobrindo o que precisa e seguindo essas boas práticas.

- **Relatório e acompanhamento:** Integre a execução de testes no seu pipeline de CI/CD. Configure para que, no mínimo, rodem todos os unitários e integração a cada push/PR, falhando em caso de qualquer teste quebrado ou cobertura abaixo do mínimo. Gere os relatórios de cobertura e, se possível, publique-os em alguma página interna ou comentário de PR para visibilidade. Empresas como a Google têm dashboards internos de qualidade – você pode mimetizar isso em escala menor mantendo um histórico da porcentagem de cobertura e densidade de falhas ao longo do tempo. Monitorar tendências ajuda a prevenir queda de qualidade (ex.: se cobertura está caindo release após release, é um alerta de dívida técnica em testes).

Ferramentas e Convenções no uso do Vitest 🛠

O projeto utiliza **Vitest** como framework de testes junto com TypeScript. O Vitest é semelhante ao Jest em sintaxe e funcionalidades, oferecendo performance otimizada para projetos modernos (especialmente integrados ao Vite). Aqui vão recomendações específicas para tirar proveito do Vitest:

- **Configuração TypeScript:** Certifique-se de incluir os tipos do Vitest no projeto para ter autocompletar e evitar erros de compilação nos testes. Você pode adicionar no topo do seu arquivo de config:

```
/// <reference types="vitest" />
import { defineConfig } from 'vite';
export default defineConfig({
  test: { /* ... */ }
});
```

(Nota: a sintaxe de triple-slash reference acima pode mudar em versões futuras do Vitest [49], mas por ora assegure-se de tê-la). Além disso, incluir `"types": ["vitest/globals"]` em seu tsconfig na seção `compilerOptions` também permite usar as funções globais de teste (describe, it, expect) sem imports explícitos.

- **Execução paralela e isolamento:** Por padrão, Vitest executa testes em paralelo (em workers separados). Isso melhora desempenho, mas reforça a necessidade de **isolamento** entre testes. Tenha em mente:

- Qualquer estado global do Node (variáveis globais, singletons, caches em memória) *não* é compartilhado entre testes em workers distintos. Se você precisar compartilhar algo globalmente entre todos os testes, talvez usar um `--single-thread` (execução serial) ou um módulo de setup manual. Mas o ideal é não depender de global mesmo.
- Ao usar banco de dados ou arquivos, se os testes rodarem simultaneamente, considere **separar recursos por teste** (como mencionado na seção de BD). Por exemplo, configure cada worker para usar um banco/teste diferente via variáveis de ambiente ou prefixos de tabela. Isso evita colisões. Se isso não for possível facilmente, você pode rodar Vitest com `--threads false` para executar testes de integração sequencialmente, garantindo ordem mas sacrificando velocidade.

- O Vitest oferece opções de filtragem de testes (por nome, por tags, etc.) [50]. Uma convenção útil é incluir hashtags ou palavras-chave nos títulos, como `it("#integration deveria conectar ao DB", ...)`. Assim, podemos rodar `vitest -t "@integration"` para executar somente os testes de integração, por exemplo. Use essa funcionalidade para categorizar testes lentos, de forma que durante o desenvolvimento diário você rode só unitários, e reserve integração/E2E para rodar menos frequentemente ou em CI.

- **Mocks e fakes no Vitest:** A API de mocking do Vitest é compatível com a do Jest:

- Use `vi.mock('module')` para mockar um módulo inteiro. Dentro você pode retornar implementações fakes das funções daquele módulo. Isso é útil para simular módulos de terceiros (por ex., uma biblioteca de envio de e-mail) nos testes unitários.
- Use `vi.fn()` para criar funções espiãs ou stubs rápidas. Ex.: `const stub = vi.fn().mockResolvedValue(42)` cria uma função assíncrona que sempre resolve 42 – perfeito para injetar no lugar de uma chamada ao BD em um teste unitário.
- Você pode limpar/resetar mocks facilmente com `vi.resetAllMocks()` em um `afterEach`, garantindo que um teste não influencia outro [51] [52].

- Lembre que Vitest também suporta **spyOn** (vi.spyOn) para interceptar métodos existentes sem totalmente substituí-los, o que é útil para verificar chamadas a funções internas.

- **Snapshots:** Embora mais utilizados em front-end, testes de snapshot podem ser aproveitados para back-end em alguns casos (por exemplo, para verificar que a resposta JSON de uma API não mudou inesperadamente). Vitest suporta snapshots com `expect(valor).toMatchSnapshot()`. Use com cautela – snapshots grandes podem mascarar detalhes, mas para contratos de resposta estáveis eles podem ajudar a detectar mudanças não intencionais.

- **Plugins e extensões:** A comunidade Vitest/Jest possui muitas extensões que podem ser úteis:

- **vitest-localdev-server** (hipotético) ou estratégias para rodar e derrubar servidores antes dos testes.
- **Supertest** já citado para HTTP.
- **@vitest/coverage-c8** é usado por padrão para cobertura (baseado no C8 do Node).
- Matchers adicionais: você pode adicionar bibliotecas como **jest-dom** (no contexto de DOM) ou outras asserções customizadas se precisar verificar coisas específicas (por exemplo, formato de string, regex etc.). O Vitest permite extender `expect` com matchers custom.

- **Testing Library**: Se algum componente de sua aplicação for de UI (React/Vue no front integrado com back?), o Vitest se integra bem com Testing Library para testes funcionais de componentes. No contexto puramente back-end, isso não se aplica diretamente, mas bom saber que o ecossistema é compatível.

- **Saída e depuração:** O Vitest tem um modo interativo (UI) e reporter legível. Você pode rodar `vitest --ui` para abrir uma interface web que mostra testes e resultados em tempo real – útil durante desenvolvimento. Para debugar um teste, use `vitest --inspect` para abrir o debugger do Node (pode colocar breakpoints no VSCode). Além disso, `vitest --watch` roda em modo observador, reexecutando testes relacionados a cada mudança de arquivo – acelera o ciclo TDD.

- **Convenções adicionais:**

- Mantenha os testes atualizados conforme mudanças no código. Evite comentar ou pular testes sem um bom motivo – um teste `.skip` permanente é dívida no projeto. Se estiver instável (flaky), resolva a causa (ex.: dependência de tempo, concorrência) ao invés de ignorar o teste.
- Documente no README ou Wiki do projeto como rodar os testes, como configurar variáveis de ambiente para teste (ex.: strings de conexão do banco de teste), e políticas de qualidade (ex.: cobertura mínima, etc.). Novos desenvolvedores devem conseguir rodar a suíte com um simples `npm test` e ter todos verdes localmente.

- Utilize integração contínua (Github Actions, GitLab CI, etc.) para rodar os testes automaticamente em cada PR. Isso garante que nenhuma mudança sem teste passe despercebida. No CI, rode com a mesma config do dev (Vitest também suporta um modo `--coverage` no CI para falhar se a cobertura ficar abaixo do threshold definido).

Em resumo, aproveite o Vitest para ter uma execução de testes rápida e integrada ao seu ambiente Node/TypeScript. Siga as convenções de estrutura e naming para que o suite de testes seja tão organizada quanto o código de produção.

Padrões de Teste em Grandes Empresas (Google e outras)

Empresas como Google, Amazon, Microsoft e outras de grande porte tratam testes de software com extrema seriedade, desenvolvendo padrões e culturas específicas. Ao criar sua estratégia de testes, vale a pena se inspirar em alguns desses princípios adotados por equipes de elite:

- **Pirâmide e Proporção de Testes:** Conforme já destacado, o Google popularizou a ideia de **70% unitários, 20% integração, 10% E2E** [8]. Esse modelo busca maximizar feedback rápido e minimizar testes lentos/flaky. Outras empresas seguem aproximações semelhantes – por exemplo, a Microsoft enfatiza ampla suíte de unit tests nos builds e alguns testes integrados em pipelines, e a Amazon historicamente frisa testes automatizados em todos os níveis (incluindo muitos testes de serviço e alguns E2E para user journeys críticos). O importante é manter a **forma de pirâmide** (mais testes na base, menos no topo) [53], evitando tanto a dependência excessiva de testes manuais quanto o antipadrão do "sorvete" (muitos testes UI em comparação aos unitários).

- **Confiabilidade e manutenção contínua:** Grandes empresas têm zero tolerância a testes instáveis (*flaky tests*). Um teste que falha ocasionalmente sem causa clara é caçado e corrigido ou removido rapidamente, pois prejudica a confiança no pipeline. Adote essa postura: se um teste falha de forma intermitente, investigue imediatamente – muitas vezes é sinônimo de condição de corrida, dependência de ordem ou suposições incorretas. Escreva testes determinísticos e isole fontes de flakiness (ex.: chamadas externas não controladas, dependência de timing exato, concorrência não sincronizada). Um mantra do Google é que **testes devem ser 100% confiáveis** – equipe nenhuma tolera falsa falha em CI porque isso atrasa deploys e consome tempo de engenharia à toa.

- **Testes como parte da cultura de desenvolvimento:** No Google, desenvolvedores escrevem testes como parte intrínseca do processo (não existe "departamento de teste separado" tradicional). Estruture seu projeto assim também: todo novo código deve vir acompanhado de testes relevantes. Code reviews devem exigir testes para novas funcionalidades ou correções de bug (quando aplicável). Crie um acordo na equipe de que **"feito" significa "testado"**. Essa mentalidade de *"Engineering Productivity"*, como chamam no Google, assegura que a base de código se mantém saudável conforme cresce [54] [55].

- **Tamanhos de testes (Small/Medium/Large):** O Google classifica testes não só por tipo, mas por tamanho/escopo:

- *Small tests* (pequenos) = unitários ou quase isso, sem I/O ou dependências; executam em < 100ms [13].

- *Medium tests* (médios) = envolvem dois ou mais componentes integrados, ainda dentro do processo (por ex., testes de integração leve, talvez usando BD em memória ou simulando chamadas de rede com stubs).
- *Large tests* (grandes) = testes end-to-end completos, possivelmente multi-processo, podendo levar até minutos (Google define limite ~15 min para um teste grande rodar [13], mas isso é em ambiente bem distribuído).

Essa categorização ajuda a orquestrar pipelines: testes pequenos rodam em cada commit (feedback imediato), médios talvez rodam a cada merge ou hora, grandes rodam em nightly builds ou em ambientes dedicados. No seu projeto, você pode não precisar de tanta formalidade, mas entender a ideia ajuda: **priorize executar os testes rápidos com frequência** (idealmente pré-commit ou pre-push) e os mais lentos um pouco menos frequente, mas sempre antes de releases. Se a suíte total começar a ficar muito demorada, considere rodar unitários no feedback imediato e talvez rodar integrações/E2E em paralelo ou em estágio separado.

- **Infraestrutura de testes robusta:** Empresas de ponta investem em ferramentas internas para facilitar testes. Por exemplo, o Google criou o **Bazel** (sistema de build) justamente para escalar compilação e execução de testes distribuída, com cache etc. Você pode não precisar desenvolver ferramentas do zero, mas use o que existe a seu favor: pipelines CI, containers para consistência do ambiente, scripts automatizados de setup/teardown de testes, etc. Automatize ao máximo o ambiente de teste para que seja fácil rodar localmente e idem no CI. Um desenvolvedor novo deve conseguir clonar o repo e rodar todos testes com um comando – se isso requer muita configuração manual, melhore o processo (escreva scripts, use docker, docs claras). **Teste os testes:** sim, certifique-se de que o pipeline de teste falha quando deve falhar (experimente introduzir um bug propositadamente para ver se o CI pega).

- **Clean Code nos testes:** Assim como código de produção deve ser limpo e legível, grandes equipes também mantêm um alto padrão nos testes. Evite duplicação excessiva (use funções helper se necessário), mas também evite *abstração excessiva* que dificulte entender o teste (cada teste deve ser legível como um caso específico). Google e outras empresas promovem práticas como:

- One assertion per test (discutível; não é regra rígida, mas busca fazer cada teste verificar uma coisa específica) – isso pode aumentar quantidade de testes mas deixa o escopo de cada um claro.
- Não faça assert de múltiplas coisas não relacionadas num único teste – melhor separar em testes diferentes. Assim quando um falhar, você sabe exatamente o que quebrou.
- Use dados consistentes e meaningfully nomeados nos testes (evite variáveis "x", "y" – use `usuarioValido`, `senhaIncorreta`, etc. para facilitar leitura).
- Remova código morto/inútil nos testes também – se uma função helper não é mais usada, limpe.

- Mantenha os testes atualizados conforme o sistema evolui. Teste quebrado ou desatualizado deve ser prioridade de conserto, não algo a adiar indefinidamente. Empresas de alta qualidade tratam a suite de testes verde como sagrada: um build vermelho interrompe o fluxo até ser resolvido.

- **Cobertura e qualidade medida:** Google e cia muitas vezes estabelecem metas quantitativas – ex.: não aceitar pull request sem testes, ou exigir X% de cobertura para considerar um módulo "verde". Vocês podem instituir no projeto, por exemplo, uma regra no CI que falha se cobertura global cair abaixo de um mínimo (como 80%). Outra prática em algumas empresas é o **Test Certification** – times atingem níveis conforme aumentam cobertura, reduzem flakiness, documentam casos. Pode

ser interessante gamificar isso internamente: ex., mostrar relatório mensal de quais partes do sistema têm menos teste e focar melhoria ali.

- **Feedback rápido e contínuo:** Times do Google rodam testes localmente antes de commitar (até integrou isso nas ferramentas de codificação deles). Imite isso configurando, se possível, *pre-commit hooks* ou git hooks que rodem pelo menos os testes unitários rapidamente antes de permitir subir código. Ou configure seu IDE/Editor para rodar testes do arquivo em edição ao salvar. O ponto é inserir os testes no ciclo de desenvolvimento e não apenas depois.

Por fim, lembre-se: **boas práticas de teste não são luxo, são necessidade** para projetos escaláveis. Adotando essas diretrizes – combinando a agilidade dos testes unitários, a fidelidade dos testes integrados, a segurança dos testes E2E e a disciplina de grandes empresas – seu projeto Node/TypeScript estará no caminho certo para alta qualidade e confiança em cada release. Happy testing!

Referências & Leituras Recomendadas:
- Guia de Melhores Práticas em Testes Node.js [56] [57]
- Artigo “Just Say No to More End-to-End Tests” – Google Testing Blog [53] [58]
- Node.js Testing Best Practices (Yoni Goldberg) – seções sobre Banco de Dados e Mocking [22] [59]
- “The Practical Test Pyramid” – Martin Fowler & Ham Vocke [60] [8]
- Post “Best Testing Practices in Node.js” – AppSignal Blog (2024) [61] [62]

---

[1] [5] [6] [8] [9] [10] Tipos de testes de software: Saiba como testar a sua aplicação | by Comercial Jera | Jera | Medium
https://medium.com/jera-apps/tipos-de-testes-de-software-saiba-como-testar-a-sua-aplica%C3%A7%C3%A3o-d2bdb41a7923

[2] [7] [14] [53] [58] Google Testing Blog: Just Say No to More End-to-End Tests
https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html

[3] [4] [43] [44] [46] [59] Best Techniques to Create Tests with the Vitest Framework - DEV Community
https://dev.to/wallacefreitas/best-techniques-to-create-tests-with-the-vitest-framework-9al

[11] [12] [13] [54] [55] Thoughts on „How Google Tests Software“ – Jakob Breu
https://jakobbr.eu/2021/06/27/thoughts-on-how-google-tests-software/

[15] [16] [17] [18] [19] [20] [21] [26] [34] [37] [38] [45] [50] [60] [61] [62] Best Testing Practices in Node.js | AppSignal Blog
https://blog.appsignal.com/2024/10/16/best-testing-practices-in-nodejs.html

[22] [23] [24] [25] [27] [28] [29] [30] [31] [32] [33] [35] [36] [39] [40] [41] [47] [48] [56] [57] GitHub - goldbergyoni/nodejs-testing-best-practices: Beyond the basics of Node.js testing. Including a super-comprehensive best practices list and an example app (April 2025)
https://github.com/goldbergyoni/nodejs-testing-best-practices

[42] KaianDev/node-courses: Uma API REST moderna e ... - GitHub
https://github.com/KaianDev/node-courses

[49] Configuring Vitest | Vitest
https://vitest.dev/config/

[51] [52] How to use Typescript with Vitest – Run That Line
https://runthatline.com/how-to-use-typescript-with-vitest/