ChatGPT

 # Diretrizes e Boas Práticas de Desenvolvimento (Next.js + TypeScript Monolito)

1. Boas Práticas de Codificação (TypeScript e Next.js)

- **Estrutura de Projeto:** Siga as convenções do Next.js para organizar pastas e arquivos. Mantenha o front-end (páginas, componentes) e back-end (rotas API) bem estruturados – por exemplo, usando diretórios separados por funcionalidade ou módulo (ex: `pages/`, `components/`, `services/`, `utils/`) [1] [2]. No Next 13+, aproveite o diretório `app/` com subpastas para rotas aninhadas e arquivos especiais (`layout.tsx`, `page.tsx`, `route.ts` etc.) conforme recomendação da documentação [3] [4]. Isso garante uma base de código organizada e modular, facilitando a manutenção.
- **Tipagem Forte:** Utilize ao máximo o sistema de tipos do TypeScript. Habilite o modo estrito no `tsconfig.json` e declare explicitamente os tipos de parâmetros, retornos de funções e variáveis importantes para evitar ambiguidade [5]. **Evite** usar o tipo `any` – ele anula os benefícios da tipagem estática e pode mascarar erros graves [6] [7]. Quando for tentador usar `any`, prefira uma tipagem mais específica ou o tipo `unknown`, que é mais seguro (exige verificação de tipo antes do uso) [8] [9].
- **Interfaces e Types:** Prefira definir **interfaces** para descrever a forma de objetos em vez de type aliases de literais de objeto. Interfaces são mais expansíveis e claras, além de serem recomendadas pelo guia do TypeScript do Google para modelos de objeto [10] [11]. Use **type aliases** para outros propósitos (uniões, tuplas, tipos primitivos compostos), mas para objetos complexos mantenha o padrão de interfaces – isso melhora a consistência e legibilidade do código [10] [11].
- **Componentização e Separação de Responsabilidades:** Em projetos Next.js monolíticos, isole responsabilidades em componentes e módulos. Crie componentes React pequenos e reutilizáveis (funções curtas e focadas são mais fáceis de testar e manter [12]). Mantenha lógica de negócio separada da apresentação – por exemplo, coloque funcionalidades de dados/negócio em arquivos de serviço ou utilitários, enquanto os componentes focam apenas em renderização/UI [13]. Esse princípio de separação limpa (separation of concerns) torna o código mais **manutenível** e facilita escalabilidade conforme o projeto cresce.
- **Imports e Módulos:** Organize as dependências de modo claro. Utilize caminhos de importação absolutos ou aliases configurados no `tsconfig` (ex: `@/components/MeuComponente`) para evitar imports relativos longos [14] [15]. O TypeScript/Next suporta path aliases, tornando o código mais legível. Além disso, evite dependências cíclicas e organize os módulos de forma hierárquica (por feature ou camada) para reduzir o acoplamento.

2. Convenções de Estilo e Padronização de Código

- **Guia de Estilo Consistente:** Adote um guia de estilo reconhecido para TypeScript. Grandes empresas como o Google seguem padrões rigorosos – por exemplo, o **Google TypeScript Style Guide** padroniza desde formatação até nomenclatura. Ferramentas como o `gts` (Google TS Style) combinam linter e formatter para aplicar essas regras automaticamente [16]. O importante é definir convenções claras (uso de camelCase para variáveis e funções, PascalCase para componentes React, etc.) e segui-las uniformemente em todo o repositório [17].

- **Linting e Formatação Automática:** Configure linters (ESLint) e formatadores (Prettier) integrados ao projeto para garantir estilo consistente e evitar "bike-shedding" em reviews. Por exemplo, a configuração padrão do Google TS já inclui ESLint + Prettier para aplicar indentação, aspas, ponto-e-vírgula, etc., de forma consistente [16]. Utilize hooks de commit ou pipelines CI para rodar o lint/format – assim, todo commit já segue o padrão definido. Isso aumenta a qualidade e legibilidade do código, além de poupar tempo de revisão em detalhes de formato.
- **Nomenclatura e Legibilidade:** Siga convenções de nomenclatura legíveis. Escolha nomes de variáveis e funções descritivos (evitando abreviações obscuras). Mantenha a consistência: por exemplo, use substantivos para nomes de classes/objetos, verbos para funções que executam ações, e CONSTANTES em maiúsculas se aplicável. Nomes de arquivos podem usar **kebab-case** ou **camelCase** de acordo com o padrão do time, mas de forma unificada. Arquivos que exportam um único componente React geralmente usam PascalCase no nome do arquivo correspondente ao componente (ex: `MeuFormulario.tsx` contendo `function MeuFormulario()`). Uma convenção consistente facilita a busca de arquivos e reduz a curva de aprendizado para novos contribuidores.
- **Código Autoexplicativo e Comentários Necessários:** Esforce-se para escrever código claro que *"se explica"* – ou seja, que a função e intenção sejam evidentes pela nomeação e estrutura. Contudo, complemente com **comentários** onde for útil para contexto adicional. Use comentários de implementação (`// ...`) para explicar partes complexas ou hacks necessários, e comentários de documentação JSDoc (`/** ... */`) para descrever o uso de funções, parâmetros e retornos não óbvios [18] [19]. Grandes empresas padronizam isso: comentários JSDoc devem agregar informação (não apenas repetir o nome da função) e esclarecer o "porquê" do código, ajudando outros desenvolvedores a entender rapidamente o propósito de cada componente [19].

3. Boas Práticas de Testes Automatizados (Unitários, Integração, E2E)

- **Cobertura de Testes e Pirâmide de Testes:** Escreva testes automatizados para todo código crítico. Inspire-se na cultura de testes do Google: eles almejam cerca de **80% de testes pequenos (unitários)**, ~15% de testes de integração e ~5% de testes de ponta-a-ponta [20]. Essa "pirâmide de testes" favorece uma base ampla de testes rápidos e isolados, complementada por alguns testes integrados e apenas o necessário de E2E (que são mais lentos e complexos) [21]. Em outras palavras, priorize muitos testes unitários (rápidos e determinísticos) para validar funções e componentes isoladamente, utilize testes de integração para cobrir a interação entre módulos/APIs (ex.: chamar uma rota Next API e verificar resposta), e escreva testes end-to-end para fluxos completos do usuário.
- **Ferramentas de Teste:** Utilize frameworks consolidados. Para testes unitários em Next/React, frameworks como **Jest** (com **React Testing Library**) permitem simular componentes e verificar saída renderizada. Para testes de integração de APIs, você pode usar o próprio **Jest** ou **Mocha** junto com utilitários como Supertest (por exemplo, iniciando o servidor Next em ambiente de teste). Para testes E2E (navegador), adote ferramentas como **Cypress** ou **Playwright**, que automatizam um browser real para simular cliques, navegação e validar comportamentos reais da aplicação. Considere também integrar esses testes de forma contínua (por exemplo, rodando em pipelines CI em cada pull request).
- **Práticas de Escrição de Testes:** Siga boas práticas ao codar testes: teste comportamentos esperados e cenários de falha, não detalhes de implementação. Cada teste deve ser independente e **hermético**, ou seja, configurar seu próprio contexto e limpar/resetar o estado ao terminar [22]. Evite

dependência entre testes ou suposições sobre ordem de execução. Use dados fictícios (mocks, stubs ou fakes) para simular dependências externas: por exemplo, em vez de chamar um banco real, use um banco em memória ou mocks para garantir rapidez e isolamento. O Google, por exemplo, prefere fakes de alta fidelidade a mocks estritos, pois fakes se comportam mais como a implementação real [23] [24] . Garanta também que os testes cubram cenários de erro – *"teste tudo o que você não quer que quebre"*, seguindo a "regra Beyoncé": *"If you liked it then you should have put a test on it"* [20] . Em resumo, testes abrangentes e bem escritos servem como uma rede de segurança contra regressões e documentam o funcionamento correto do sistema.

- **Coverage e Qualidade:** Acompanhe a métrica de cobertura de código pelos testes (code coverage) como um indicador auxiliar. Algumas organizações definem metas – e.g., no Google considera-se ~60% de cobertura aceitável, 75% bom e 90% excelente em muitos projetos [25] . Embora a porcentagem por si só não garanta qualidade, uma boa cobertura ajuda a identificar partes não testadas do código. Foque principalmente em cobrir as partes críticas e de lógica complexa. Integre a execução dos testes na pipeline de integração contínua, falhando builds caso testes quebrem ou cobertura fique abaixo do acordado. Essa disciplina garante que nenhuma mudança seja integrada sem passar pelos "freios de qualidade" automatizados fornecidos pelos testes.

4. Documentação e Comentários no Código

- **Comentários Úteis e Padronizados:** Cultive o hábito de documentar seu código com comentários significativos. Utilize **JSDoc** para comentar interfaces públicas (funções, classes, componentes) – descreva o que fazem, quais parâmetros esperam e o que retornam, especialmente se não for óbvio [18] [19] . Evite comentários redundantes que apenas repetem o nome de algo; em vez disso, forneça contexto do "porquê" e dicas de uso. Por exemplo, se uma função realiza uma lógica complexa ou um algoritmo não trivial, inclua um comentário explicando a ideia. Comentar partes críticas melhora a **manutenibilidade**: novos desenvolvedores entenderão rapidamente a intenção original.
- **Documentação Externa (README e Wiki):** Mantenha um README.md atualizado no repositório, detalhando como rodar o projeto, estrutura de pastas, convenções adotadas e exemplos de uso da API (se aplicável). Esse README pode incluir um resumo destas diretrizes para que todos do time as sigam. Para projetos maiores, considere documentar arquiteturas ou decisões de design em documentos markdown ou wiki do projeto. Grandes empresas investem em documentação porque entendem que o conhecimento deve ser compartilhado e não ficar só na cabeça dos desenvolvedores – "se você aprendeu algo, **escreva-o**" e compartilhe [26] .
- **Auto-documentação e Legibilidade:** Esforce-se para que o próprio código seja legível e **autoexplicativo**. Nomeie variáveis e funções de forma clara, estruture o código em funções/métodos pequenos com responsabilidades bem definidas, de modo que a necessidade de comentários seja mínima. Revise trechos confusos e refatore para melhorar a clareza. No Google, por exemplo, existe até um processo formal de *"readability"*, onde certos engenheiros certificados avaliam se o código novo é fácil de entender antes de ser integrado [27] . Embora você não precise desse formalismo, adote a mesma mentalidade: código **legível** e bem documentado (quando necessário) facilita revisões, evita bugs e acelera o onboarding de novos membros.
- **Comentários de Implementação:** Para detalhes internos ou temporários, use comentários de implementação `//` no código explicando *"truques"* ou partes não óbvias da lógica. Todavia, evite deixar comentários obsoletos – mantenha-os atualizados ou remova-os se não fizerem mais sentido. Comentários desatualizados são enganosos. Inclua TODOs claros se há trabalho futuro pendente, de preferência com contexto ou referência de acompanhamento (ex:

`// TODO: melhorar algoritmo X para casos de limite...`). Assim a equipe tem visibilidade de débitos técnicos e pode resolvê-los depois.

5. Revisão de Código, Pair Programming e Colaboração

- **Política de Code Review:** Nenhuma mudança vai direto para a branch principal sem passar por revisão de código (*code review*). Adote a regra de pelo menos um outro desenvolvedor (idealmente dois em mudanças maiores) revisar cada *pull request*. O objetivo principal do code review deve ser **melhorar a saúde geral do código a cada alteração**, conforme a filosofia do Google [28]. Isso significa verificar não só correção, mas se a mudança torna o código mais limpo, compreensível e consistente. O revisor deve ser diligente em identificar potenciais problemas de design, bugs ocultos, faltas de cobertura de testes e aderência aos padrões do projeto. Em contrapartida, também deve manter um balanço – não exigir perfeição absoluta em detalhes triviais se a mudança já traz uma melhoria clara [29] [30]. Use comentários do tipo "Nit:" para sugestões não mandatórias de pequenos ajustes [30]. Valorize fatos técnicos e guia de estilo sobre preferências pessoais [31]. No fim, o código só é mergeado quando atende aos critérios de qualidade acordados e ao guia de estilo – essa disciplina mantém a base de código coesa no longo prazo.
- **Colaboração e Cultura de Equipe:** Fomente um ambiente onde todos se sintam responsáveis pelo código e à vontade para dar e receber feedback. Práticas ágeis como **pair programming** são encorajadas em tarefas complexas ou críticas – dois pares de olhos capturam problemas mais cedo e disseminam conhecimento. Muitas empresas observam que pair programming serve como uma revisão de código em tempo real, aumentando a eficiência e qualidade do que é produzido [32]. Mesmo fora do pair programming formal, incentive desenvolvedores a discutirem soluções juntos, fazer *peer reviews* informais e ajudar uns aos outros. No Google, valoriza-se a colaboração sobre o "mito do gênio solitário": o sucesso de um projeto depende mais de comunicação e trabalho em equipe do que de contribuições isoladas [33]. Mantenha uma cultura de **humildade, respeito e confiança** – feedback deve ser técnico e nunca pessoal, celebrando acertos e corrigindo erros de forma construtiva. Uma equipe unida e colaborativa invariavelmente produz software de melhor qualidade.
- **Processo de Review Eficiente:** Estabeleça um fluxo de code review que não trave o progresso. Defina SLAs internos – por exemplo, tentar revisar um PR dentro de 1 dia útil. Use ferramentas de CI para automatizar parte da revisão (lint, testes) para que os reviewers foquem no design/correção. Se um PR é grande demais, talvez peça ao autor para dividi-lo em partes menores (facilitando a revisão incremental). Documente em um guia de contribuição como funciona o processo de review no projeto (quem aprova, necessidade de *LGTM*, etc.). **Sempre registre as decisões** de revisão no próprio PR ou em documentação associada, para referência futura. Lembre-se: o objetivo do review é compartilhar conhecimento e melhorar o código – trate-o como uma oportunidade de aprendizado mútuo e não como barreira.
- **Pair Programming & Mentoria:** Além de reviews, incentive sessões de pair programming regulares, seja para resolver um bug complicado, para onboarding de um novo membro ou simplesmente para compartilhar contexto em uma feature. O Google observa que esse tipo de colaboração direta aumenta a sabedoria coletiva e reduz o "fator ônibus" (risco de conhecimento concentrado em uma só pessoa) [34]. Desenvolvedores menos experientes aprendem com os mais experientes e vice-versa. Considere rotacionar pares e revisar código de diferentes partes do sistema, assim todos ganham visão ampla do monolito full-stack. A longo prazo, essa cultura de mentoria contínua eleva o nível técnico de toda a equipe.

6. Versionamento e Deploy em Monolitos

- **Trunk-Based Development:** Opte por uma estratégia de branching simples para um monorepo monolítico. Grandes empresas como Google adotam **trunk-based development** – ou seja, *todos desenvolvedores trabalham em uma única branch principal (trunk)*, fazendo commits frequentes e integrando mudanças pequenas continuamente [35]. Branches de feature curtas podem ser usadas, mas devem ser mescladas de volta ao trunk rapidamente para evitar divergências longas. Essa abordagem minimiza conflitos de merge e garante que a versão principal esteja sempre funcional e pronta para release [35] [36]. Com trunk-based, integra-se testes automatizados e CI/CD: cada commit gatilha build + testes, garantindo que o trunk se mantenha verde. Isso habilita **entregas contínuas** – a qualquer momento a branch principal está estável para implantar uma nova versão.
- **Controle de Versão e Tags:** Mesmo em deploy contínuo, mantenha um esquema de versionamento claro das releases. O uso de **versionamento semântico (SemVer)** é altamente recomendado para comunicar mudanças: por ex., `vMAJOR.MINOR.PATCH` – incremente a versão MAJOR para mudanças incompatíveis (breaking), MINOR para novas features retrocompatíveis, e PATCH para correções de bugs [37] [38]. Alternativamente, algumas organizações adotam versionamento por data (CalVer) para releases frequentes, mas o importante é ser consistente [39]. Use tags do Git para marcar releases com sua versão, facilitando rastrear exatamente qual commit foi para produção. Documente brevemente no CHANGELOG as alterações de cada versão (novidades, melhorias, fixes) – isso ajuda na transparência e em eventuais necessidades de rollback.
- **Automação de Deploy (CI/CD):** Implemente pipelines automatizadas para build, teste e deploy. Cada commit na branch principal deve acionar um pipeline que: roda lint e testes, gera o build de produção (e.g. `next build`), e, se tudo passar, implementa nos ambientes alvo (ex.: homologação, depois produção). Use ferramentas de CI/CD confiáveis (GitHub Actions, GitLab CI, Jenkins, etc.) e infraestruturas reproduzíveis – por exemplo, conteinerize a aplicação (Docker) para padronizar os ambientes entre dev/staging/prod. Em produção, considere estratégias como **blue-green deployment** ou **canary releases** se possível, para reduzir downtime: implante a nova versão em paralelo, realize smoke tests e depois direcione tráfego gradualmente. Isso tudo minimiza riscos ao atualizar o monolito.
- **Feature Flags e Lançamentos Graduais:** Em um monolito full-stack, uma prática poderosa é utilizar *feature flags*. Ou seja, encapsular novas funcionalidades atrás de "chaves" que podem ser ligadas/desligadas. Desse modo, você pode implantar código novo em produção **desabilitado** para usuários, ativando apenas quando estiver validado. Essa técnica, adotada por empresas de ponta, permite lançamentos graduais e rollback instantâneo de features problemáticas sem revert de código [40] [41]. Ferramentas como LaunchDarkly, ConfigCat ou mesmo flags caseiras no código podem gerenciar isso. Em resumo, deploye features inacabadas desligadas, teste em produção com usuários piloto ou porcentagem do tráfego, e vá habilitando conforme a confiança aumenta – mantendo o deploy do monolito frequente sem comprometer estabilidade.
- **Gerenciamento de Configurações:** Separe configurações do código, conforme o princípio de 12-Factor App. Para monolitos Next.js, use as variáveis de ambiente (`.env`) para dados sensíveis e configs que variam por ambiente (chaves de API, strings de conexão, etc.). **Nunca** codifique segredos no repositório. Mantenha arquivos `.env.development`, `.env.production` etc., e certifique-se de listá-los no `.gitignore` [42]. Apenas variáveis explícitas para uso no front-end devem ter o prefixo `NEXT_PUBLIC_` e mesmo assim, evite expor qualquer dado crítico [42]. Gerencie essas configs via CI (por exemplo, usando os secrets do deploy ou serviços de vault) para que cada ambiente receba suas credenciais de forma segura durante o deploy. Isso permite que o

mesmo pacote de build seja implantado em dev/staging/prod com comportamento diferente controlado por config, sem alterar código.

7. Regras de Segurança e Validação de Entrada

- **Validação de Entrada (Input Validation): Nunca** confie apenas na validação do lado cliente – valide todos os dados no servidor também. Implemente verificações de tipo, formato e tamanho em todas entradas que seu back-end Next.js receber (query params, corpo de requisições JSON, etc.) [43] . Utilize técnicas como *whitelisting* (permitir apenas formatos esperados) e rejeite prontamente inputs malformados. Por exemplo, se espera um email, valide com regex; se espera um número, verifique range válido. A validação multilayer (cliente *e* servidor) dificulta ataques, pois mesmo que o invasor burle o front, o back-end terá sua própria proteção [44] .
- **Sanitização e Encoding:** Além de validar, **sanitize** entradas para remover ou escapar caracteres potencialmente maliciosos. Isso previne ataques de injeção de código. Use funções de *escape* ou bibliotecas ao inserir dados do usuário em HTML (prevenindo XSS) e ao montar consultas a bancos de dados (prevenindo SQL Injection) [43] . No acesso a banco, sempre que possível utilize *query parameters* ou ORM que faça tratamento adequado em vez de concatenar strings SQL [43] . Da mesma forma, ao renderizar dados no React/Next, lembre que o React escapa por padrão o conteúdo – então **evite usar** `dangerouslySetInnerHTML` a não ser que tenha certeza que o conteúdo é seguro. Caso precise renderizar HTML vindo do usuário, sanitize-o (por ex., com uma biblioteca como DOMPurify) antes. Para prevenir XSS, também é boa prática habilitar uma Content Security Policy estrita no app (Next.js permite configurar isso facilmente) [42] [45] .
- **Autenticação e Autorização:** Siga práticas robustas para autenticar usuários e proteger rotas. Utilize provedores confiáveis (p. ex. NextAuth, OAuth) ao invés de reinventar autenticação. Armazene senhas apenas de forma criptografada segura (hash forte + salt, ex: bcrypt). Implemente controle de sessão com tokens seguros (JWT assinado e com expiração, ou cookies httpOnly/secure). Valide em cada requisição autenticada se o usuário tem permissão para aquele recurso (authZ). No Next.js, proteja páginas/rotas sensíveis server-side (getServerSideProps checando autenticação, ou Middleware para redirecionar não logados). *Nunca* confie somente em checagens client-side de role, pois podem ser burladas – o servidor sempre deve reforçar regras de autorização.
- **Proteções Web Gerais:** Fique atento às vulnerabilidades comuns (OWASP Top 10) e mitigue-as sistematicamente. Previna **SQL Injection** usando queries preparadas como dito. Previna **XSS** escapando saída e usando CSP. Impeça **CSRF** em operações sensíveis – Next.js já inclui proteções em rotas API se usar tokens anti-CSRF ou checar origens; ao usar formulários, considere tokens CSRF ou utilize métodos idempotentes sempre que possível. **Clickjacking**: defina cabeçalhos como `X-Frame-Options: DENY`. **Segurança de headers**: ative `Strict-Transport-Security` (HSTS) para exigir HTTPS, `X-XSS-Protection`, `X-Content-Type-Options: nosniff` e assim por diante (muitas já configuráveis no `next.config.js` ou via middleware). Mantenha as dependências atualizadas aplicando patches de segurança rapidamente – um monolito up-to-date reduz riscos de exploits conhecidos. Por fim, considere rodar scans de vulnerabilidade e testes de invasão (pentests) periodicamente, especialmente antes de releases importantes, para identificar brechas que passaram despercebidas [46] . Revisões de código com foco em segurança também são válidas (por ex., ter alguém no time verificando se cada PR segue essas práticas). Em resumo: integridade e segurança dos dados do usuário são prioridade máxima; invista tempo para fortificar sua aplicação contra ataques comuns.

8. Performance e Escalabilidade no Next.js

- **Renderização Estática vs. Dinâmica:** Tire proveito dos recursos do Next.js para otimizar performance. Sempre que possível, use **Static Site Generation (SSG)** ou **Incremental Static Regeneration (ISR)** para páginas que não precisam ser renderizadas a cada requisição. Páginas estáticas são geradas no build e servidas via CDN, tornando o tempo de resposta muito rápido. O Next.js por padrão já faz renderização estática de componentes sempre que aplicável e cacheia os resultados para melhorar o throughput [47]. Use **Dynamic Rendering (SSR)** apenas onde necessário (por ex., páginas personalizadas por usuário, dados que mudam a cada acesso). Mesmo em SSR, aproveite o *caching*: Next permite cachear respostas de fetch (revalidate) e até páginas inteiras em memória ou CDN usando cabeçalhos adequados [48]. Avalie o uso de `<Suspense>` e streaming (no Next 13+) para enviar partes da página ao cliente assim que prontas, evitando bloquear o carregamento completo [49].
- **Code Splitting e Carregamento Sob Demanda:** Mantenha o bundle JavaScript do front-end o mais enxuto possível. O Next.js já realiza **code-splitting automático por rota**, dividindo o código por páginas e carregando somente o necessário para cada uma [50]. Além disso, utilize **lazy loading** para componentes não essenciais no primeiro paint – por exemplo, carregue componentes de gráfico ou mapas apenas quando o usuário interagir ou quando entrarem em viewport (React.lazy ou dynamic imports do Next) [50]. Isso evita enviar um grande volume de JS inicial. Verifique também dependências externas: remova libs não usadas e monitore tamanhos. Você pode usar o plugin `@next/bundle-analyzer` para identificar módulos pesados no bundle e tomar ações (ex: dividir, carregar de CDN, etc.). Um bundle menor significa carregamento mais rápido e melhor desempenho, especialmente em redes móveis.
- **Otimização de Imagens e Assets:** Imagens são frequentemente o maior peso em páginas web – use o componente `<Image>` do Next para carregamento otimizado. Ele gera imagens em formatos modernos (WebP/AVIF), dimensiona conforme o dispositivo e aplica lazy load automático [51]. Isso melhora o *loading* e evita layout shift (melhor pontuação de Core Web Vitals). Otimize também outros assets: fontes customizadas podem ser carregadas via Next Font Optimization para eliminar flashes e reduzir requisições externas [52]. Arquivos estáticos (JS, CSS, imagens) que ficarem em `/public` serão servidos diretamente e podem ter cache de longo prazo – configure cabeçalhos de cache adequados (Next já pode colocar `Cache-Control: immutable` em assets com hash no nome).
- **Monitore Desempenho (Web Vitals):** Adote uma postura pró-ativa de monitoração de performance. Use ferramentas como **Lighthouse** (no Chrome ou via `npm Next.js` script) para auditar a aplicação em modo produção e identificar gargalos em métricas como First Paint, Time to Interactive, CLS, etc. [53]. O Next.js expõe um hook `useReportWebVitals` que pode capturar *Core Web Vitals* reais dos usuários – integre isso com um serviço de analytics para ter dados do mundo real. Analise e otimize conforme necessário (por ex., se LCP está alto em uma página, pode ser imagem não otimizada ou payload grande de dados). Além disso, monitore o uso de CPU/Memória do servidor Node (caso esteja usando server próprio e não serverless) – identifique endpoints mais lentos ou com alto consumo e avalie melhorias (caching, algoritmos mais eficientes, etc.).
- **Escalabilidade Horizontal:** Prepare o monolito para escalar quando a carga aumentar. O Next.js pode rodar em modo **serverless** (cada função/rota API isolada) ou como aplicação Node tradicional. Em ambos os casos, mantenha a aplicação **stateless** o quanto possível – sessão de usuário armazenada em cookies/JWT ou em store central (Redis, banco) ao invés de memória local – assim você pode subir múltiplas instâncias em paralelo sem inconsistências. Utilize um balanceador de carga (ex: ingress no Kubernetes, ou plataforma como Vercel que já faz isso automaticamente) para

distribuir requisições entre instâncias. Se estiver em Node serverfull, use clustering ou escale contêineres verticalmente conforme necessidade. Monitore métricas de infra (CPU, memória, tempo de resposta) e faça *capacity planning*: antes de saturar, adicione recursos. Considere uso de CDN para conteúdo estático e páginas cacheáveis – alivia completamente o trabalho do servidor em conteúdo que não precisa ser dinâmico. Para tarefas pesadas de CPU ou I/O bloqueante, avalie mover para processos assíncronos (queues, lambdas) para não degradar a resposta ao usuário. Em suma, *escalabilidade* no contexto monolito Next.js significa garantir que ele atende bem tanto 100 quanto 100k usuários, através de caching, otimizações de código e arquitetura de implantação elástica.

Referências: Boas práticas inspiradas por guias do Google e da comunidade – incluindo o Google TypeScript Style Guide [10] [11], Engineering Practices do Google (code review, colaboração) [28] [33], diretrizes do Next.js [47] [51], recomendações de segurança OWASP [43], entre outros. Estas recomendações visam manter o código **saudável**, consistente e preparado para crescer, seguindo padrões adotados por empresas líderes do mercado. Cada item acima deve ser revisitado regularmente e adaptado conforme o projeto evolui, garantindo que a equipe esteja sempre na mesma página quanto à qualidade e mantenibilidade do software. [28] [20]

---

[1] [2] [5] [7] [12] [13] [14] [15] [17] A Guide for Next.js with TypeScript | Refine
https://refine.dev/blog/next-js-with-typescript/

[3] [4] Getting Started: Project Structure | Next.js
https://nextjs.org/docs/app/getting-started/project-structure

[6] [8] [9] [10] [11] [18] [19] Google TypeScript Style Guide
https://google.github.io/styleguide/tsguide.html

[16] GitHub - google/gts: ⚗ TypeScript style guide, formatter, and linter.
https://github.com/google/gts

[20] [21] [22] [23] [24] [26] [27] [33] [34] Matt Boyle | My Notes from Software Engineering at Google: Lessons Learned from Programming Over Time
https://mattjamesboyle.com/posts/notes-from-software-eng-at-google/

[25] The Great Code Coverage Holy Wars of the 21st Century - Medium
https://medium.com/nerd-for-tech/the-great-code-coverage-holy-wars-of-the-21st-century-6fb11e7acce4

[28] [29] [30] [31] The Standard of Code Review | eng-practices
https://google.github.io/eng-practices/review/reviewer/standard.html

[32] [35] [36] [40] [41] Explaining Trunk Based Development - Travis CI
https://www.travis-ci.com/blog/explaining-trunk-based-development/

[37] [38] [39] Software Release Versioning - Guide and Best Practices | LaunchDarkly
https://launchdarkly.com/blog/software-release-versioning/

[42] [47] [48] [49] [50] [51] [52] [53] Guides: Production | Next.js
https://nextjs.org/docs/app/guides/production-checklist

[43] [44] [45] [46] Input Validation and Sanitization: Protecting Your Application from Malicious Input | by cyber_pix | Medium
https://medium.com/@use.abhiram/input-validation-and-sanitization-protecting-your-application-from-malicious-input-28fee92ea0d3