14 — Qualidade e pré-commit

14 — Qualidade e pré-commit

> Três gates: `vite-plugin-checker` durante o dev, `pre-commit` no commit, `npm run build` no build. **Não há suíte de testes** — assumido e declarado, não esquecido. Regra: o gate corrige o que dá para corrigir e barra o que não dá.

Os gates

Quando	O que roda	Barra?
`npm run dev`	`vite-plugin-checker` (type-check contínuo)	Não — mostra overlay
`git commit`	Node version, flags de debug, `lint-staged`	Sim
`git pull`	`post-merge` → `npm install` se preciso	Não
`npm run build`	`lint` → `tsc` → `vite build`	Sim

Ausência de testes — declarado

Este padrão **não tem suíte de testes automatizados**. Não há Vitest, Jest, Testing Library nem Playwright. Isso não é omissão da documentação: é o estado da referência.

O que substitui:

- `strict: true` + `noUnusedLocals` + `noUnusedParameters` no TypeScript
- ESLint com `--max-warnings 0`
- `npm run build` como verificação de integração (compila tudo)

- Verificação manual da tela contra o backend real

Se o projeto novo quiser testes, decida no início. Adicionar depois exige refatorar componentes que nasceram acoplados a Context e ao `UseCaseManager`. Se adotar, atualize 19 para incluir o comando na Definition of Done.

ESLint

Flat config (`eslint.config.js`) — ESLint 9+ abandonou o formato `.eslintrc.json`. Não é uma escolha do guia, é obrigatório a partir da v9; `--ext ts,tsx` também deixou de ser aceito no CLI (o escopo de arquivos agora vem do `files` de cada bloco de config).

```
// eslint.config.js
import js from "@eslint/js";
import tsPlugin from "@typescript-eslint/eslint-plugin";import tsParser from "@typescript-
eslint/parser";
import reactHooks from "eslint-plugin-react-hooks";import prettier from "eslint-config-prettier";
import globals from "globals";
export default [
  { ignores: ["dist", "build", "vite.config.ts", "setEnvironment.js"] },
js.configs.recommended,
  {
    files: ["**/*.{ts,tsx}"],
    languageOptions: {
      parser: tsParser,      parserOptions: { ecmaVersion: "latest", sourceType: "module"
},
      globals: { ...globals.browser, ...globals.node, ...globals.es2020 }
    },
    plugins: { "@typescript-eslint": tsPlugin, "react-hooks": reactHooks },
    rules: {
      ...tsPlugin.configs.recommended.rules,
      ...reactHooks.configs.recommended.rules,
      "no-unused-vars": "off",
      "no-undef": "off",
      "@typescript-eslint/no-unused-vars": "error",      "react-hooks/exhaustive-deps":
"warn",
```

```
      "no-debugger": "error",      "no-console": ["error", { allow: ["warn", "error", "info"]
}],
      "@typescript-eslint/no-explicit-any": "error",      "@typescript-eslint/no-empty-
interface": "error",
      "no-shadow": "off",
      "@typescript-eslint/no-shadow": "error",
      "no-throw-literal": "error",
      "no-return-await": "error",
      "no-duplicate-imports": "error",
      eqeqeq: ["error", "always"],
      "no-var": "error",
      "prefer-const": "error",
      "use-isnan": "error"
    }
  },
  prettier
];
```

Requer `@eslint/js` e `globals` como devDependencies novas (não existiam no formato antigo).

`prettier` (de `eslint-config-prettier`) por último desliga as regras de formatação do ESLint — Prettier manda no formato, ESLint manda na correção.

> `no-undef: "off"` **não é opcional.** O `@typescript-eslint/recommended` do formato antigo (`.eslintrc.json`) desligava `no-undef` internamente ao resolver o `extends`; pegando o objeto de regras direto (`tsPlugin.configs.recommended.rules`) em flat config, esse desligamento **não vem junto** — sem repetir explicitamente, `no-undef` acusa falso positivo em tipos ambient do TS (`EventListener`, `React` usado só como tipo em `.ts`). O `tsc` já cobre isso; deixe o ESLint fora do caminho.

Regras que merecem explicação

Regra	Efeito prático
`no-console` (permite warn/error/info)	`console.log` esquecido barra o commit
`@typescript-eslint/no-explicit-any`	`any` exige `eslint-disable` justificado

Regra	Efeito prático
`react-hooks/exhaustive-deps: warn`	Ver abaixo
`eqeqeq`	`==` proibido
`no-shadow`	Variável interna não pode mascarar externa

> **`exhaustive-deps` como `"warn"`, não `"off"`.** Os `useEffect` de abertura de caso de uso (06) dependem de rodar em condições específicas, e a regra reclamaria de arrays intencionalmente incompletos — resolva com `// eslint-disable-next-line` justificado nesses poucos casos, não desligando a regra inteira.

`eslint-plugin-react-hooks` v7 trouxe regras novas — decida o alcance

O `recommended` da v7 é bem mais amplo que o da v4 (que só tinha `rules-of-hooks` + `exhaustive-deps`). Ele inclui regras no estilo "React Compiler" — por exemplo `react-hooks/set-state-in-effect`, que acusa qualquer `setStateX(...)` chamado direto no corpo de um `useEffect` (fora de um callback de evento/subscrição).

Isso pega padrões legítimos de código já existente: estado 100% derivado de outro estado/prop que era sincronizado via `useEffect`. A correção correta **não é suprimir a regra**, é aplicar o padrão que o React recomienda:

- Se o valor é **derivado** de outro estado/prop (ex.: uma lista filtrada de uma query), calcule direto no corpo do componente — não precisa de `useState`/`useEffect` nenhum.
- Se precisa resincronizar quando uma prop muda (ex.: um hook que lê de `localStorage` por `key`), ajuste o estado **durante o render** (guardando a prop anterior em outro `useState` e comparando), não dentro de um `useEffect`.

Adotar o `recommended` completo é a recomendação — ele pega bug real de cascata de render. Se o projeto novo preferir adiar, ligue só `rules-of-hooks` + `exhaustive-deps` (equivalente ao v4) e trate o resto depois, mas isso é uma escolha explícita, não o padrão.

`--max-warnings 0`

```
```

```
eslint . --report-unused-disable-directives --max-warnings 0
```

Warning é erro. Sem isso, warnings acumulam até ninguém mais ler a saída. `--report-unused-disable-directives` acusa `eslint-disable` que não suprime mais nada — remove supressão obsoleta. Sem `--ext`: em flat config o CLI não aceita mais essa flag, o escopo de arquivos vem do `files` de cada bloco em `eslint.config.js`.

Prettier

```
// .prettierrc
{
  "semi": true,
  "singleQuote": false,
  "tabWidth": 2,
  "trailingComma": "none",
  "printWidth": 120,
  "arrowParens": "always",
  "endOfLine": "crlf",
  "bracketSpacing": true,
  "bracketSameLine": true,
  "proseWrap": "preserve"
}
```

```
# .prettierignore
node_modules
dist
build
coverage
# Gerado em runtime por setEnvironment.js
public/config.json
package-lock.json
```

Dois pontos não-óbvios:

- `endOfLine: "crlf"` — o time é Windows. Se houver dev em Linux/macOS, troque para `"auto"` e configure `.gitattributes`, senão todo commit reescreve o arquivo inteiro.
- `bracketSameLine: true` — o `>` de tag multi-linha fica na última prop, não em linha própria. Incomum, mas é o padrão da base.

husky

Instalação

```
// package.json — repositório de módulo único
"scripts": { "prepare": "husky" }
```

```
// package.json — projeto web dentro de um monorepo"scripts": { "prepare": "cd ../.. && husky caminho/do/modulo/.husky" }
```

A forma do monorepo instala os hooks apontando para `.husky/` dentro do módulo web, enquanto o `core.hooksPath` é configurado na raiz do repositório.

> **Armadilha do monorepo.** `core.hooksPath` é config **local** do clone, mas `.husky/` é **versionado**. Trocar de branch pode ligar ou desligar os hooks em silêncio. Depois de clonar ou de trocar para uma branch que mexe em `package.json`, rode `npm run prepare` e confirme com `git config core.hooksPath`.

`pre-commit`

```
# .husky/pre-commit# git roda o hook da raiz do repo; projeto Node pode estar em subpasta
hookdir="$(cd "$(dirname -- "$0")" && pwd)"
cd "$hookdir/.." || exit 1
# valida Node sem depender de fnm no shell
. "$hookdir/check-node-version.sh"
. "$hookdir/check-debug-flags.sh"
```

```
npx lint-staged
```

Três verificações, nesta ordem: versão de Node → flags de debug → lint-staged.

check-node-version.sh

```
# .husky/check-node-version.sh# Sourced pelo hook, ja com cwd no projeto Node. Nao depende de fnm/nvm
# estarem carregados (GUIs de git nao herdam o profile) -- compara o node# do PATH com .node-version (aceita >=).expected_version=$(tr -d '[:space:]' < .node-version 2>/dev/null)
actual_version=$(node -v 2>/dev/null | sed 's/^v//')
if [ -z "$actual_version" ]; then  echo "hook: 'node' nao encontrado no PATH. Ative o Node >= ${expected_version:-do projeto} (ex: 'fnm use')."
  exit 1
fi
if [ -n "$expected_version" ]; then  lower_version=$(printf '%s\n%s\n' "$expected_version" "$actual_version" | sort -V | head -n1)
  if [ "$lower_version" != "$expected_version" ]; then    echo "hook: Node v$actual_version encontrado, necessario >= v$expected_version (.node-version)."
    exit 1
  fi
fi
```

Resolve um problema real: commitar por GUI de Git com Node 8 ativo faz o `lint-staged` falhar com erro incompreensível.

check-debug-flags.sh

```
# .husky/check-debug-flags.sh# Bloqueia commit com enableLogs: true (flag de depuracao de useUseCaseControls).
#
# So chama 'exit' no caminho de erro -- este arquivo e' sourced, entao# 'exit 0' encerraria o hook antes do lint-staged rodar.
repo_root=$(git rev-parse --show-toplevel)staged_files=$(git diff --cached --name-only --diff-
```

```
filter=ACM -- '*.ts' '*.tsx')
if [ -n "$staged_files" ]; then
  matches=""
  for file in $staged_files; do    file_match=$(grep -n "enableLogs:[[:space:]]*true"
"$repo_root/$file" 2>/dev/null || true)
    if [ -n "$file_match" ]; then
      matches="$matches$file:
$file_match
"
    fi
  done
  if [ -n "$matches" ]; then    echo "hook: 'enableLogs: true' encontrado em arquivo(s) staged -
remova antes de commitar:" >&2
    printf '%s' "$matches" >&2
    exit 1
  fi
fi
```

O comentário sobre `exit 0` não é decorativo. O arquivo é carregado com `.` (source). Um `exit 0` no fim encerraria o `pre-commit` inteiro antes do `lint-staged`, e o commit passaria sem lint. Ao escrever um hook novo neste estilo, só chame `exit` no caminho de erro.

Adapte a lista de flags ao projeto: qualquer flag de depuração que não deva ser commitada entra aqui.

`post-merge`

```
# .husky/post-merge# Instala dependencias quando o merge/pull alterou package.json/lock.
changed=$(git diff-tree -r --name-only --no-commit-id ORIG_HEAD HEAD)
case "$changed" in
  *caminho/do/modulo/package-lock.json*|*caminho/do/modulo/package.json*)    echo "post-merge:
dependencias mudaram -> rodando npm install..."    hookdir="$(cd "$(dirname -- "$0")" &&
pwd)"
    cd "$hookdir/.." || exit 1
    . "$hookdir/check-node-version.sh"
    npm install
```

```
    ;;
esac
```

Elimina a classe de bug "puxei a branch e o app não sobe". Ajuste os caminhos do `case` ao layout do projeto novo.

lint-staged

```
// package.json
"lint-staged": {
  "src/**/*.{ts,tsx}": [    "eslint . --ext ts,tsx --report-unused-disable-directives --max-warnings 0",
    "prettier --write ."
  ],
  "**/*.{json,css,scss,md,html,yml,yaml}": ["prettier --write ."]
}
```

> **Duas imprecisões comuns nesta configuração, que valem corrigir.**
>
> 1. Os comandos usam `.` (projeto inteiro) em vez dos arquivos staged. O `lint-staged` passa a lista de arquivos como argumento, que aqui é ignorada. Funciona, mas fica lento e formata arquivos que você não tocou.
> 2. `prettier --write .` faz o hook **reescrever e re-stagear** arquivos. Um commit pode sair com formatação que você não fez.
>
> Forma correta:
>
> ```
> "lint-staged": {
>   "src/**/*.{ts,tsx}": [    "eslint --fix --report-unused-disable-directives --max-warnings 0",
>     "prettier --write"
>   ],
>   "**/*.{json,css,scss,md,html,yml,yaml}": ["prettier --write"]
> ```

```
}
```

Sem `.`, o `lint-staged` anexa só os arquivos staged.

`vite-plugin-checker`

```
checker({
  typescript: true,
  overlay: { initialIsOpen: false, position: "tl" },
  terminal: true
});
```

Type-check contínuo durante `npm run dev`, em overlay e no terminal. **Não substitui `tsc`** no build — o `npm run build` roda `tsc` separadamente, porque o checker só verifica o que foi tocado na sessão.

Contornar os gates

`git commit --no-verify` pula os hooks. Reserve para emergência real (commit de WIP em branch pessoal). Nunca em branch compartilhada.

`HUSKY=0` desabilita os hooks no ambiente — útil em CI, onde o pipeline já roda lint e build.

Revision #2
Created Thu, Aug 20, 2026 5:38 PM by Geraldo Barbosa
Updated Tue, Aug 25, 2026 4:54 PM by Geraldo Barbosa