
**QUALIDADE DE SOFTWARE**

Renan Yassumoto Ferreira

# Escolhendo Linguagem e Frameworks

Barueri
2025

## Sumário

## 1. Introdução

Este trabalho apresenta a estratégia de testes para uma plataforma web com frontend em React, backends em .NET e Java e aplicativos mobile em React Native, integrada a serviços de terceiros.

A proposta prioriza testes rápidos e confiáveis, reservando os E2E aos fluxos críticos. No Web, o Cypress é adotado para testes de componentes e jornadas no navegador, com simulação de APIs para estabilidade. No Mobile, a automação funcional utiliza Appium orquestrado pelo WebdriverIO, com execução também em device farms para garantir compatibilidade em dispositivos reais.

As APIs são validadas de forma leve com cy.request(), e o GitHub concentra colaboração, revisão e pipelines, assegurando rastreabilidade, qualidade e velocidade nas entregas.

## 2. Caso de estudo

**Características técnicas:**

- Plataforma Web.
- Integrações com serviços de API de terceiros.
- Frontend feito em React.
- Backend feito parte em .Net e parte em Java.

**Composição dos times:**

- 2 equipes compostas por:
  - 1 Dev Front React
  - 2 Dev Backend Java
  - 1 Dev Backend .Net
  - 1 QA e 1 UX Designer
- 4 equipes compostas por:
  - 2 Devs Front React Native
  - 2 QAs
  - 1 Designer
- Há apenas 1 Product Owner e um Scrum Master para as equipes

### 2.1 Objetivo

A - Descrever quais tecnologias/ferramentas você orientaria a adoção para a sua estratégia de testes, considerando o caso de estudo apresentado acima.

B - Submeter como resposta o detalhamento das escolhas e os motivos pelo qual considera as mesmas adequadas para este contexto.

## 3. Tecnologias e ferramentas recomendadas

### 3.1 Visão-geral

A estratégia de testes deve seguir o formato pirâmide, dando mais foco aos testes de unidade e de integração, e menos aos testes de ponta a ponta.

Trabalharemos com o princípio do “shift-left”, ou seja, os testes começam cedo — já nas etapas iniciais de desenvolvimento — utilizando versões simuladas (mocks) dos serviços externos.

Deve buscar a padronização entre as diferentes linguagens usadas no projeto (JavaScript, Java e .NET) e o uso de ferramentas que funcionem bem em qualquer ambiente, especialmente para os testes de interface, desempenho, segurança e geração de relatórios.

### 3.2 Frontend Web (React)

**Componentes e unidade**

- Cypress Component Testing para validar componentes React de forma próxima ao uso real, interagindo com a tela.
- Jest + React Testing Library para partes muito pequenas ou regras puras.

**Fluxos de ponta a ponta (navegador)**

- Cypress E2E para os caminhos essenciais (login, cadastro, checkout, pagamento).
- Interceptação de chamadas (cy.intercept) para simular APIs de terceiros e manter os testes estáveis.
- Produtividade com cypress-testing-library, cypress-grep, e evidências automáticas (screenshots e vídeos).
- Relatórios com Mochawesome (ou o HTML do Cypress) publicados no CI.

**Justificativa**

O Cypress combina rapidez, estabilidade e boa experiência de escrita/leitura de testes. Cobre tanto a camada de componentes quanto os fluxos no navegador, reduzindo retrabalho e falhas intermitentes.

### 3.3 Mobile (React Native)

Para garantir a qualidade no mobile, seguiremos com:

1. Automação funcional com Appium orquestrado pelo WebdriverIO.
2. Execução distribuída em dispositivos reais por meio de device farms (Sauce Labs, BrowserStack, TestingBot ou AWS Device Farm).

**Ferramentas**

- Node.js, JDK (OpenJDK), Android Studio (SDK/AVD), bundletool.
- Diagnóstico e inspeção: Appium Doctor e Appium Inspector.
- Framework de testes: WebdriverIO para conectar no Appium Server.
- Relatórios: usar Allure ou o reporter nativo do WDIO, anexando vídeos/screenshots das execuções.

**Justificativa**
A combinação Appium + WebdriverIO garante que os testes mobile reflitam o uso real do app, validando gestos, permissões e mudanças de estado (entrar/sair do app, variações de rede), pontos que não aparecem em testes de navegador. O WebdriverIO simplifica a orquestração do Appium, reduz "trabalho braçal" de configuração e oferece relatórios claros (Allure), o que acelera o diagnóstico quando algo falha.

Executar a suíte em device farms traz segurança de compatibilidade em versões de Android e tamanhos de tela diferentes, sem a equipe manter um parque físico de aparelhos. Os vídeos e logs produzidos por esses serviços facilitam entender a causa raiz de erros intermitentes.

Ao automatizar os mesmos fluxos críticos do web (login, cadastro, compra), mantemos critérios de aceite alinhados entre plataformas e reaproveitamos dados e cenários, reduzindo esforço de manutenção.

### 3.4 Backends (.NET e Java)

**Testes do próprio código**

- Java: JUnit 5 + Mockito para regras de negócio; Spring Boot Test para integração.
- .NET: xUnit (ou NUnit) + Moq para isolação; WebApplicationFactory para integração em ASP.NET.

**APIs (caixa-preta) com ferramentas leves**

- Cypress via cy.request() para testes de API (status, corpo e mensagens) com execução rápida no CI.

**Integrações externas**

- **Simulação de terceiros** no front com cy.intercept.
- No back, uso de **WireMock** para simular serviços e manter os testes previsíveis.

**Justificativa**
Regras de negócio são cobertas no próprio serviço. A camada de API recebe uma verificação rápida e confiável. O time ganha feedback contínuo sem ficar refém de ambientes externos.

### 3.5 Organização no GitHub

Usaremos o GitHub como ponto único de organização entre desenvolvedores e times porque ele junta, no mesmo lugar, código, revisão, automação e acompanhamento do trabalho. Os Pull Requests criam um fluxo claro de colaboração (todo ajuste passa por revisão), enquanto as branches protegidas evitam que algo vá para a linha principal sem validação.

As Issues e os Projects ajudam a priorizar e dar visibilidade do que cada equipe está fazendo, e os templates garantem que todos sigam o mesmo padrão ao descrever mudanças e evidências de teste. Com o GitHub Actions, os testes automatizados rodam assim que um PR é aberto, dando feedback rápido sobre quebras e qualidade.

Por fim, recursos como histórico de mudanças, logs de auditoria e alertas de dependências aumentam a segurança e a rastreabilidade, reduzindo retrabalho e facilitando a tomada de decisão.

### 3.6 Divisão do trabalho entre as equipes

**Times Web**

- Devs React + QA: Cypress (Component + E2E) no dia a dia e cy.intercept para isolar o front.
- Devs de back (.NET/Java): manter unidade/integração nos serviços e apoiar o QA com rotas e dados previsíveis.

**Times Mobile (React Native)**

- QA + Devs RN: WebdriverIO + Appium para os fluxos espelho dos cenários web críticos e Appium Inspector para mapear seletores. Device farm para validar em dispositivos reais.

**Product Owner e Scrum Master**

- Critérios de aceite incluindo: teste de componente, caminho feliz E2E no Cypress (Web) e cenário espelho no Appium (Mobile).
- Quadro de qualidade com métricas do GitHub Actions (tempo de pipeline, taxa de retrabalho, flakiness).

## 4. Conclusão

A combinação de Cypress para Web, Appium + WebdriverIO para Mobile e governança via GitHub entrega uma base sólida: feedback rápido, testes estáveis, rastreáveis e fáceis de evoluir.

As integrações externas são controladas com simulações, as APIs têm documentação e validação e a automação roda de ponta a ponta, tanto em emuladores quanto em dispositivos reais por meio de device farms, garantindo relevância prática e escalabilidade.