QUALITY ASSURANCE (QA) ENGINEERING

THE COMPLETE GUIDE

From Tester to Qualiiy Enginser: Fundamentals, Automation (Cypress, Robot, API) and Career

Índice Clicável

PARTE I: FUNDAMENTOS E CONCEITOS ESSENCIAIS DE QA

PARTE II: PLANEJAMENTO E DOCUMENTAÇÃO

PARTE III: FERRAMENTAS E AUTOMAÇÃO DE TESTES

Prefácio

A Engenharia de Quality Assurance (QA) transcendeu o papel de simples "caçador de bugs" para se tornar uma disciplina estratégica e essencial no ciclo de desenvolvimento de software moderno. Em um mundo onde a velocidade de entrega é crucial, garantir a qualidade desde o início do processo (shift-left) é o que diferencia produtos de sucesso de fracassos custosos.

Este e-book foi concebido como um guia completo, prático e profundo para profissionais que desejam iniciar ou aprimorar sua carreira em QA. Ao longo de mais de 100 páginas, cobrimos desde os fundamentos teóricos mais sólidos, passando por técnicas avançadas de planejamento e documentação, até a aplicação prática das ferramentas de automação mais requisitadas pelo mercado.

Utilizamos como base a valiosa trilha de estudos fornecida pelo leitor, complementando-a com informações aprofundadas, exemplos de código prontos para uso, guias passo a passo e orientações de carreira, garantindo que você tenha em mãos um recurso robusto para construir um portfólio vencedor e se destacar no mercado de trabalho.

PARTE I: FUNDAMENTOS E CONCEITOS ESSENCIAIS DE QA

Capítulo 1: O Universo da Qualidade de Software

1.1. Definição e Importância da Qualidade de Software

A **Qualidade de Software (QS)** pode ser definida como o grau em que um sistema, componente ou processo atende aos requisitos especificados e às necessidades ou expectativas do cliente/usuário. É um conceito multifacetado que abrange não apenas a ausência de defeitos, mas também a usabilidade, eficiência, manutenibilidade, confiabilidade, segurança e performance do produto.

O Custo da Não Qualidade (CoNQ)

O **Custo da Não Qualidade (CoNQ)** é um conceito crucial que todo profissional de QA deve compreender profundamente. Ele representa todos os custos incorridos devido a:

- **Falhas em produção:** Indisponibilidade do sistema, perda de dados, corrupção de informações
- **Retrabalho:** Correção de bugs, refatoração de código, reteste
- **Suporte ao cliente:** Atendimento a reclamações, resolução de problemas
- **Perda de reputação:** Danos à marca, perda de confiança dos usuários
- **Oportunidades de negócio perdidas:** Clientes que não adotam o produto, churn de usuários

Estudos demonstram que quanto mais tarde um defeito é encontrado no ciclo de desenvolvimento, mais caro se torna corrigi-lo. A regra empírica é:

Fase de Descoberta	Custo Relativo de Correção
Requisitos	1x
Design	3-6x
Desenvolvimento	10-15x
Teste	15-40x
Produção	100-1000x

Um bug encontrado em produção pode custar até **1000 vezes mais** do que se fosse encontrado na fase de requisitos. Isso justifica o investimento em QA desde o início do projeto.

Mentalidade de "Quality First"

A **mentalidade de "Quality First"** (Qualidade em Primeiro Lugar) é a base da Engenharia de QA moderna. Ela exige que:

1. **A qualidade seja uma responsabilidade compartilhada** por toda a equipe (desenvolvedores, designers, gerentes de produto, QA e liderança)
2. **A qualidade seja considerada desde o início** do projeto, não apenas como uma etapa final
3. **Os processos sejam robustos** para prevenir defeitos, não apenas detectá-los
4. **A automação seja priorizada** para permitir feedback rápido e contínuo
5. **A cultura de qualidade seja cultivada** em toda a organização

1.2. Quality Assurance (QA) vs. Quality Control (QC)

Embora frequentemente usados como sinônimos, **Quality Assurance (QA)** e **Quality Control (QC)** representam abordagens distintas, mas complementares, na gestão da qualidade.

Característica	Quality Assurance (QA)	Quality Control (QC)
Foco	Processos e Prevenção	Produto e Detecção
Natureza	Proativo	Reativo
Quando Ocorre	Durante todo o ciclo de desenvolvimento	Após a conclusão do produto ou módulo
Objetivo	Garantir que os processos corretos sejam seguidos para evitar defeitos	Identificar e corrigir defeitos no produto final
Exemplos	Revisão de requisitos, Definição de padrões de codificação, Treinamento, Auditorias, Planejamento de testes	Execução de testes (funcionais, regressão), Inspeções, Revisão de código, Testes de aceitação
Responsabilidade	Toda a equipe	Principalmente QA/Testadores
Pergunta Chave	"Como vamos garantir qualidade?"	"O produto atende aos padrões de qualidade?"

O **QA** é o guarda-chuva estratégico que estabelece o *como* a qualidade será alcançada. Ele garante que o processo de desenvolvimento (o *meio*) seja robusto, documentado e repetível. O **QC** é a tática operacional que verifica se o produto final (o *fim*) atende aos padrões estabelecidos.

Um **Engenheiro de QA moderno** atua em ambas as frentes, mas com uma forte inclinação para a prevenção (QA). Isso significa:

- Participar de reuniões de requisitos para garantir clareza
- Revisar designs e arquiteturas para identificar riscos
- Colaborar com desenvolvedores durante a implementação
- Executar testes estratégicos (QC) quando apropriado
- Automatizar testes para feedback contínuo

1.3. O Ciclo de Vida do Desenvolvimento de Software (SDLC)

O QA está presente em todas as fases do SDLC, independentemente da metodologia (Cascata, Ágil, DevOps). Vamos detalhar cada fase:

Fase 1: Requisitos

Atividades de QA:

- Análise de requisitos para clareza, completude e testabilidade
- Identificação de ambiguidades e inconsistências
- Definição de critérios de aceitação
- Participação em reuniões de refinamento

Exemplo: Se um requisito diz "O sistema deve ser rápido", o QA questiona: "Rápido significa menos de 1 segundo? 2 segundos? Para qual operação?" Isso garante que os testes sejam objetivos.

Fase 2: Design

Atividades de QA:

- Revisão da arquitetura e design para identificar potenciais pontos de falha
- Análise de fluxos de dados e integrações
- Identificação de cenários de erro e edge cases
- Planejamento de estratégia de teste

Fase 3: Implementação

Atividades de QA:

- Desenvolvimento de testes de unidade (muitas vezes pelo próprio desenvolvedor)
- Desenvolvimento de testes de integração (SDET ou QA)
- Code review com foco em testabilidade
- Preparação de ambientes de teste

Fase 4: Teste

Atividades de QA:

- Execução de testes funcionais e não funcionais
- Execução de testes de aceitação (UAT)

- Reporte de defeitos com informações detalhadas
- Validação de correções

Fase 5: Implantação

Atividades de QA:

- Testes de fumaça (smoke tests) no ambiente de produção
- Validação de dados migrados
- Monitoramento de logs e métricas
- Suporte ao rollback se necessário

Fase 6: Manutenção

Atividades de QA:

- Testes de regressão para novas funcionalidades ou correções
- Monitoramento de qualidade em produção
- Análise de incidentes
- Otimização de testes baseada em feedback

1.4. Mentalidade Shift-Left e Quality First

Shift-Left é um conceito que significa mover as atividades de teste para a esquerda no timeline do projeto, ou seja, começar mais cedo. Em vez de testar apenas no final, o QA participa desde o início.

Benefícios do Shift-Left:

1. **Detecção precoce de defeitos:** Bugs encontrados no início são mais baratos de corrigir
2. **Feedback contínuo:** Os desenvolvedores recebem feedback sobre qualidade em tempo real
3. **Redução de retrabalho:** Menos surpresas desagradáveis no final do projeto
4. **Melhor colaboração:** QA e desenvolvedores trabalham juntos desde o início
5. **Automação eficaz:** Testes automatizados podem rodar continuamente

Implementação Prática:

- **Pair Testing:** QA trabalha ao lado do desenvolvedor enquanto o código é escrito
- **Test-Driven Development (TDD):** Testes são escritos antes do código
- **Behavior-Driven Development (BDD):** Testes são escritos em linguagem natural que todos entendem
- **Code Review:** QA participa de revisões de código

Capítulo 2: Tipos e Níveis de Testes de Software

2.1. A Pirâmide de Testes

A **Pirâmide de Testes** é um modelo conceitual que sugere a proporção ideal de diferentes tipos de testes em um projeto. Ela foi popularizada por Mike Cohn e é fundamental para uma estratégia de teste eficaz.

Estrutura da Pirâmide

Base (70%): Testes de Unidade

Características:

- Testam a menor parte testável do código (funções, métodos, classes)

- Rápidos (executam em milissegundos)
- Baratos (fáceis de escrever e manter)
- Isolados (não dependem de outros componentes)
- Executados frequentemente (a cada commit)

Exemplo:

```
// Teste de unidade para uma função de cálculo de desconto
function calculateDiscount(price, discountPercent) {
  return price * (1 - discountPercent / 100);
}

test('calculateDiscount deve retornar preço correto', () => {
  expect(calculateDiscount(100, 10)).toBe(90);
  expect(calculateDiscount(50, 20)).toBe(40);
});
```

Meio (20%): Testes de Integração/Serviço

Características:

- Testam a comunicação entre componentes
- Incluem interações com banco de dados, APIs externas, serviços
- Mais lentos que testes de unidade (executam em segundos)
- Mais caros de escrever e manter
- Executados frequentemente (a cada build)

Exemplo:

```
// Teste de integração para uma API de usuário
test('GET /users/:id deve retornar usuário', async () => {
  const response = await request(app)
    .get('/users/1')
    .expect(200);

  expect(response.body).toHaveProperty('id', 1);
  expect(response.body).toHaveProperty('name');
});
```

Topo (10%): Testes de UI/E2E

Características:

- Testam o fluxo completo do usuário
- Incluem interações com a interface (cliques, digitação)
- Lentos (executam em segundos ou minutos)
- Caros de escrever e manter
- Frágeis (quebram facilmente com mudanças na interface)
- Executados com menos frequência (antes de releases)

Exemplo:

```
// Teste E2E com Cypress
describe('Fluxo de Login', () => {
  it('Usuário deve fazer login com sucesso', () => {
    cy.visit('https://example.com/login');
    cy.get('input[name="email"]').type('user@example.com');
    cy.get('input[name="password"]').type('senha123');
    cy.get('button[type="submit"]').click();
    cy.url().should('include', '/dashboard');
  });
});
```

Por que essa proporção?

A regra é: **quanto mais baixo na pirâmide, mais testes, mais rápidos e mais baratos**. O objetivo é encontrar a maioria dos bugs na base, onde o custo de correção

é menor.

Se você inverter a pirâmide (muitos testes E2E, poucos testes de unidade), você terá:

- Suite de testes lenta (demora horas para rodar)
- Custo alto de manutenção
- Feedback lento para os desenvolvedores
- Maior chance de falsos positivos

2.2. Testes Funcionais

Testes funcionais verificam se o sistema atende aos requisitos e especificações de negócio. Eles focam no “o que” o sistema faz, não em “como” ele faz.

Tipo de Teste Funcional	Foco Principal	Explicação Abrangente	Quando Usar
Unidade (Unit)	Componentes isolados (funções, métodos)	Testam a menor parte testável do código. Essenciais para garantir a lógica interna. Escritos pelo desenvolvedor.	Sempre, a cada commit
Integração (Integration)	Fluxo de dados entre módulos	Verificam se os módulos interagem corretamente, incluindo conexões com bancos de dados e serviços externos.	Após testes de unidade, antes de testes de sistema
Sistema (System)	O sistema como um todo	Testam o comportamento completo do sistema em um ambiente que simula a produção.	Após integração estar completa
Regressão (Regression)	Funcionalidades existentes	Garantem que novas alterações (correções ou novas features) não quebraram funcionalidades que já estavam funcionando.	Após cada mudança no código
Aceitação (UAT)	Necessidades do usuário final	Testes formais realizados pelo cliente ou Product Owner para aceitar ou rejeitar o sistema.	Antes da release para produção
Fumaça (Smoke)	Funcionalidades críticas	Testes rápidos que verificam se as funcionalidades críticas estão funcionando.	Após deploy em novo ambiente

Exemplo Prático: Teste de Regressão

Cenário: Você corrigiu um bug na tela de checkout onde o cupom de desconto não estava sendo aplicado. Agora você precisa executar testes de regressão para garantir que a correção não quebrou outras funcionalidades.

Casos de Teste de Regressão:

1. Login com credenciais válidas
2. Adicionar item ao carrinho
3. Visualizar carrinho

4. Aplicar cupom de desconto válido
5. Remover cupom de desconto
6. Alterar quantidade de itens
7. Prosseguir para checkout
8. Preencher dados de entrega
9. Selecionar método de pagamento
10. Confirmar pedido

Se todos esses testes passarem, você tem confiança de que a correção não causou regressões.

2.3. Testes Não Funcionais

Testes não funcionais avaliam o quão bem o sistema funciona, focando em atributos de qualidade como performance, segurança, confiabilidade e usabilidade.

Tipo de Teste Não Funcional	Foco Principal	Explicação Abrangente	Métrica Típica
Desempenho (Performance)	Velocidade e responsividade	Mede o tempo de resposta do sistema sob condições normais. Exemplo: Página deve carregar em menos de 2 segundos.	Tempo de resposta (ms)
Carga (Load)	Comportamento sob carga esperada	Simula o número esperado de usuários para verificar a estabilidade. Exemplo: Sistema deve suportar 1000 usuários simultâneos.	Usuários simultâneos
Estresse (Stress)	Comportamento sob carga extrema	Simula uma carga muito acima do esperado para encontrar o ponto de falha do sistema.	Ponto de ruptura
Segurança (Security)	Vulnerabilidades e proteção de dados	Identifica falhas de segurança, como injeção SQL, XSS, CSRF e vulnerabilidades de autenticação.	Vulnerabilidades encontradas
Usabilidade (Usability)	Facilidade de uso e experiência do usuário	Avalia a interface e a experiência do usuário para garantir que o sistema seja intuitivo e fácil de usar.	Satisfação do usuário
Compatibilidade (Compatibility)	Diferentes ambientes (navegadores, SOs)	Garante que o software funciona corretamente em diferentes plataformas, navegadores e dispositivos.	Plataformas suportadas
Confiabilidade (Reliability)	Estabilidade ao longo do tempo	Testa se o sistema permanece estável durante um período prolongado de operação.	MTBF (Mean Time Between Failures)
Recuperação (Recovery)	Capacidade de recuperação após	Testa se o sistema consegue se recuperar de falhas de	RTO (Recovery Time Objective)

Tipo de Teste Não Funcional	Foco Principal	Explicação Abrangente	Métrica Típica
	falhas	forma graciosa.	

Exemplo Prático: Teste de Carga

Cenário: Você está testando um e-commerce que espera 5000 usuários simultâneos durante a Black Friday.

Teste de Carga:

1. Simular 5000 usuários acessando o site simultaneamente
2. Cada usuário navega por 10 páginas, adiciona 3 itens ao carrinho e faz checkout
3. Medir:
    - Tempo de resposta médio
    - Tempo de resposta máximo (P95, P99)
    - Taxa de erro
    - Throughput (requisições por segundo)
    - Utilização de CPU e memória

Resultado Esperado:

- Tempo de resposta médio: < 2 segundos
- P95: < 5 segundos
- Taxa de erro: < 0,1%
- Throughput: > 1000 req/s

Se o teste falhar, você identifica o gargalo (banco de dados, servidor, rede) e otimiza antes da Black Friday.

2.4. Técnicas de Design de Testes

As técnicas de design de testes ajudam o QA a criar casos de teste eficazes, cobrindo o máximo de cenários com o mínimo de esforço.

Testes de Caixa Branca (White-Box)

Baseados na estrutura interna do código. O testador precisa conhecer o código-fonte.

Técnicas:

- **Cobertura de Código (Code Coverage):** Garante que todas as linhas de código foram executadas
- **Cobertura de Branches:** Garante que todos os caminhos possíveis (if/else) foram testados
- **Cobertura de Condições:** Garante que todas as condições lógicas foram testadas

Exemplo:

```
function validateAge(age) {
  if (age < 0) {
    return "Idade não pode ser negativa";
  } else if (age < 18) {
    return "Menor de idade";
  } else if (age > 120) {
    return "Idade inválida";
  } else {
    return "Maior de idade";
  }
}

// Testes para cobertura total:
test('validateAge com idade negativa', () => {
  expect(validateAge(-5)).toBe("Idade não pode ser negativa");
});

test('validateAge com menor de idade', () => {
  expect(validateAge(15)).toBe("Menor de idade");
});

test('validateAge com idade válida', () => {
  expect(validateAge(25)).toBe("Maior de idade");
});

test('validateAge com idade muito alta', () => {
  expect(validateAge(150)).toBe("Idade inválida");
});
```

Testes de Caixa Preta (Black-Box)

Baseados nos requisitos e funcionalidades, sem conhecimento da estrutura interna.

Técnicas:

- **Particionamento de Equivalência:** Dividir as entradas em grupos que devem se comportar de forma similar
- **Análise de Valor Limite:** Testar os limites dos grupos de equivalência
- **Tabela de Decisão:** Testar combinações de condições

Exemplo:

```
// Requisito: "O sistema deve aceitar idades entre 18 e 100 anos"

// Particionamento de Equivalência:
// Grupo 1: Idades válidas (18-100)
// Grupo 2: Idades inválidas (< 18)
// Grupo 3: Idades inválidas (> 100)

// Análise de Valor Limite:
// Testes: 17, 18, 19, 99, 100, 101

test('Aceita idade 18 (limite inferior)', () => {
  expect(isValidAge(18)).toBe(true);
});

test('Rejeita idade 17 (abaixo do limite)', () => {
  expect(isValidAge(17)).toBe(false);
});

test('Aceita idade 100 (limite superior)', () => {
  expect(isValidAge(100)).toBe(true);
});

test('Rejeita idade 101 (acima do limite)', () => {
  expect(isValidAge(101)).toBe(false);
});
```

Testes de Caixa Cinza (Gray-Box)

Combinação das duas, onde o testador tem conhecimento parcial da estrutura interna (ex: acesso a logs ou banco de dados).

2.5. Testes Exploratórios e Ad-Hoc

Testes Exploratórios são testes não roteirizados onde o testador explora o sistema de forma criativa, buscando encontrar defeitos que testes formais podem não encontrar.

Características:

- Não seguem um plano rígido
- Baseados na experiência e intuição do testador
- Úteis para encontrar defeitos inesperados
- Documentados em tempo real

Exemplo de Sessão de Teste Exploratório:

1. Objetivo: Encontrar defeitos na tela de cadastro de usuário
2. Tempo: 1 hora
3. Atividades:
    - Tentar cadastrar com email vazio
    - Tentar cadastrar com email inválido (sem @)
    - Tentar cadastrar com senha muito curta
    - Tentar cadastrar com caracteres especiais no nome
    - Tentar cadastrar com espaços em branco
    - Tentar cadastrar o mesmo email duas vezes
    - Tentar cadastrar com navegador em modo offline
4. Defeitos encontrados:
    - Campo de email aceita espaços em branco
    - Mensagem de erro não é clara
    - Botão de envio fica desabilitado por muito tempo

Capítulo 3: Métricas e Indicadores de Qualidade

3.1. Cobertura de Testes

A **Cobertura de Testes** mede a porcentagem de código que foi executada pelos testes.

Tipos de Cobertura:

Tipo	Definição	Fórmula	Objetivo
Cobertura de Linhas	Porcentagem de linhas de código executadas	(Linhas executadas / Total de linhas) × 100	> 80%
Cobertura de Branches	Porcentagem de caminhos de código executados	(Branches executados / Total de branches) × 100	> 75%
Cobertura de Funções	Porcentagem de funções testadas	(Funções testadas / Total de funções) × 100	> 80%
Cobertura de Condições	Porcentagem de condições lógicas testadas	(Condições testadas / Total de condições) × 100	> 70%

Exemplo com JavaScript/Jest:

```
// arquivo.js
function calculateTotal(items) {
  let total = 0;
  for (let item of items) {
    if (item.discount) {
      total += item.price * (1 - item.discount);
    } else {
      total += item.price;
    }
  }
  return total;
}

// arquivo.test.js
describe('calculateTotal', () => {
  test('Calcula total sem desconto', () => {
    const items = [{ price: 100, discount: 0 }];
    expect(calculateTotal(items)).toBe(100);
  });

  test('Calcula total com desconto', () => {
    const items = [{ price: 100, discount: 0.1 }];
    expect(calculateTotal(items)).toBe(90);
  });

  test('Calcula total com múltiplos itens', () => {
    const items = [
      { price: 100, discount: 0 },
      { price: 50, discount: 0.2 }
    ];
    expect(calculateTotal(items)).toBe(140);
  });
});

// Cobertura esperada: 100% de linhas, branches e funções
```

3.2. Densidade de Defeitos

A **Densidade de Defeitos** mede a quantidade de defeitos encontrados por unidade de código.

Fórmula:

```
Densidade de Defeitos = (Número de Defeitos / Linhas de Código) × 1000
```

Interpretação:

- 0-0.5 defeitos por 1000 linhas: Excelente
- 0.5-1.0 defeitos por 1000 linhas: Bom
- 1.0-2.0 defeitos por 1000 linhas: Aceitável
- > 2.0 defeitos por 1000 linhas: Ruim

Exemplo:

- Módulo A: 10 defeitos, 5000 linhas de código
  - Densidade = (10 / 5000) × 1000 = 2.0 (Ruim)
- Módulo B: 3 defeitos, 5000 linhas de código
  - Densidade = (3 / 5000) × 1000 = 0.6 (Bom)

3.3. Taxa de Falha e Confiabilidade

A **Taxa de Falha** mede quantas vezes o sistema falha em um período.

Métricas Relacionadas:

Métrica	Definição	Fórmula
MTBF	Tempo médio entre falhas	Total de horas / Número de falhas
MTTR	Tempo médio para reparar	Total de horas de reparo / Número de falhas
Disponibilidade	Porcentagem de tempo que o sistema está disponível	(MTBF / (MTBF + MTTR)) × 100

Exemplo:

- Sistema funcionou 720 horas em um mês
- Teve 4 falhas
- Cada falha levou em média 2 horas para reparar

Cálculos:

- MTBF = 720 / 4 = 180 horas
- MTTR = (4 × 2) / 4 = 2 horas
- Disponibilidade = (180 / (180 + 2)) × 100 = 98.9%

PARTE II: PLANEJAMENTO E DOCUMENTAÇÃO

Capítulo 4: O Plano de Teste Profissional

O **Plano de Teste** é o documento fundamental que descreve o escopo, a abordagem, os recursos e o cronograma das atividades de teste. Ele serve como um guia para a equipe de QA e como um contrato de qualidade com as partes interessadas.

4.1. Estrutura de um Plano de Teste

A estrutura de um Plano de Teste deve ser clara, abrangente e seguir padrões reconhecidos como IEEE 829.

Seção Essencial	Conteúdo Detalhado	Exemplo
1. Introdução	Objetivo do plano, escopo do produto, referências (documentos de requisitos).	“Este plano descreve a estratégia de teste para o módulo de pagamento da versão 2.0 do e-commerce.”
2. Itens a Serem Testados	Módulos, funcionalidades ou requisitos que serão incluídos no teste.	“Processamento de pagamento, validação de cartão, confirmação de pedido”
3. Itens Não Testados	O que explicitamente NÃO será testado e por quê.	“Interface de administrador (fora do escopo), integração com sistema legado (em desenvolvimento)”
4. Estratégia de Teste	Abordagem geral (manual/automação, tipos de testes, prioridades).	“70% automação, 30% manual; Foco em testes de regressão e aceitação”
5. Recursos	Pessoas, ferramentas, ambientes e orçamento.	“2 QA, 1 SDET, Jira, Cypress, Postman, Ambiente de staging”
6. Cronograma	Datas de início e fim de cada fase de teste.	“Teste funcional: 01-15 de março; Teste de regressão: 16-20 de março”
7. Critérios de Entrada	Condições que devem ser atendidas antes de iniciar os testes.	“Código deve estar compilado, testes de unidade devem passar, ambiente deve estar configurado”
8. Critérios de Saída	Condições que devem ser atendidas para considerar os testes concluídos.	“Cobertura > 80%, 0 bugs críticos, 95% dos testes devem passar”
9. Riscos e Contingências	Riscos identificados e planos de mitigação.	“Risco: Ambiente de staging pode ficar indisponível. Mitigação: Usar ambiente de backup”
10. Aprovações	Assinaturas de stakeholders.	“Gerente de Projeto, Líder de QA, Product Owner”

4.2. Escopo e Estratégia de Teste

Definindo o Escopo

O escopo define exatamente o que será e o que não será testado. Isso é crucial para evitar ambiguidades.

Exemplo de Escopo Bem Definido:

```
ESCOPO INCLUÍDO:
✓ Login com email e senha
✓ Recuperação de senha
✓ Cadastro de novo usuário
✓ Edição de perfil
✓ Logout

ESCOPO NÃO INCLUÍDO:
✗ Login com redes sociais (integração com terceiros)
✗ Autenticação de dois fatores (em desenvolvimento)
✗ Interface de administrador (será testada em sprint posterior)
```

Estratégia de Teste

A estratégia define como você vai testar.

Exemplo de Estratégia:

```
ESTRATÉGIA DE TESTE - Módulo de Autenticação

1. TIPOS DE TESTES:
   - Testes de Unidade (desenvolvedor)
   - Testes de Integração (QA)
   - Testes Funcionais (QA)
   - Testes de Segurança (QA especializado)

2. PROPORÇÃO:
   - 60% Automação (Cypress para testes E2E)
   - 40% Manual (testes exploratórios, segurança)

3. PRIORIDADE:
   - P1 (Crítica): Login, logout, recuperação de senha
   - P2 (Alta): Validação de email, força de senha
   - P3 (Média): Mensagens de erro, UX

4. AMBIENTE:
   - Desenvolvimento: Testes de unidade
   - Staging: Testes de integração e funcionais
   - Produção: Testes de fumaça após deploy

5. FERRAMENTAS:
   - Cypress (testes E2E)
   - Postman (testes de API)
   - OWASP ZAP (testes de segurança)
   - Jira (rastreamento de defeitos)
```

4.3. Critérios de Entrada e Saída

Critérios de Entrada

Condições que devem ser atendidas ANTES de iniciar os testes.

Exemplo:

```
CRITÉRIOS DE ENTRADA:
☐ Código deve estar compilado sem erros
☐ Testes de unidade devem passar (100%)
☐ Ambiente de staging deve estar disponível
☐ Dados de teste devem estar carregados
☐ Plano de teste deve ser aprovado
☐ Ferramentas de teste devem estar configuradas
☐ Acesso a contas de teste deve ser fornecido
```

Se algum critério não for atendido, os testes não devem começar.

Critérios de Saída

Condições que devem ser atendidas para considerar os testes concluídos.

Exemplo:

```
CRITÉRIOS DE SAÍDA:
☐ Cobertura de testes > 80%
☐ 0 bugs críticos ou bloqueadores
☐ 95% dos testes devem passar
☐ Todos os bugs P1 devem ser corrigidos
☐ Todos os casos de teste devem ser executados
☐ Relatório de teste deve ser gerado
☐ Aprovação do gerente de projeto
```

4.4. Exemplo Prático de Plano de Teste

PLANO DE TESTE - E-commerce v2.0 (Módulo de Carrinho)

PLANO DE TESTE - Módulo de Carrinho de Compras

1. Introdução
Este plano descreve a estratégia de teste para o módulo de carrinho de compras
da versão 2.0 do e-commerce. O objetivo é garantir que os usuários possam adicionar, remover e modificar itens no carrinho de forma confiável.

2. Escopo
Incluído:
- Adicionar item ao carrinho
- Remover item do carrinho
- Alterar quantidade de itens
- Aplicar cupom de desconto
- Visualizar total do carrinho
- Persistência de carrinho (entre sessões)

Não Incluído:
- Integração com gateway de pagamento
- Envio de email de carrinho abandonado
- Recomendações de produtos

3. Estratégia de Teste
- **Tipos de Testes**: Funcional, Integração, E2E
- **Proporção**: 70% Automação, 30% Manual
- **Ferramentas**: Cypress, Postman, Jira
- **Ambiente**: Staging

4. Recursos
- 1 QA (Teste Manual)
- 1 SDET (Automação)
- 1 Desenvolvedor (Suporte)
- Ambiente de Staging
- Dados de teste (100 produtos, 50 cupons)

5. Cronograma
- Teste Funcional: 01-05 de março
- Teste de Integração: 06-08 de março
- Teste E2E: 09-10 de março
- Teste de Regressão: 11-12 de março
- Relatório Final: 13 de março

6. Critérios de Entrada
- ☐ Código compilado
- ☐ Testes de unidade passando

```
- ☐ Ambiente de staging disponível
- ☐ Dados de teste carregados
- ☐ Acesso a contas de teste

## 7. Critérios de Saída
- ☐ Cobertura > 85%
- ☐ 0 bugs críticos
- ☐ 98% dos testes passando
- ☐ Todos os casos de teste executados

## 8. Riscos
- **Risco**: Ambiente de staging pode ficar indisponível
  **Mitigação**: Usar ambiente de backup
- **Risco**: Dados de teste insuficientes
  **Mitigação**: Preparar dados adicionais antecipadamente

## 9. Aprovações
- Gerente de Projeto: _____
- Líder de QA: _____
- Product Owner: _____
```

Capítulo 5: Casos de Teste e Rastreabilidade

5.1. Estrutura de um Caso de Teste

Um **Caso de Teste** é um conjunto de condições ou variáveis sob as quais um testador determinará se um sistema atende aos requisitos.

Componentes Essenciais:

Componente	Descrição	Exemplo
ID	Identificador único	TC-001
Título	Descrição breve do que está sendo testado	“Validar login com email e senha corretos”
Pré-condições	Estado do sistema antes do teste	“Usuário não está logado, conta existe no sistema”
Passos	Ações específicas que o testador deve executar	1. Ir para página de login; 2. Digitar email; 3. Digitar senha; 4. Clicar em Login
Dados de Entrada	Valores específicos para o teste	Email: “user@example.com”, Senha: “senha123”
Resultado Esperado	O que deve acontecer se o teste passar	“Usuário é redirecionado para dashboard”
Resultado Atual	O que realmente aconteceu	Preenchido durante a execução
Status	Pass/Fail/Blocked	Pass
Prioridade	Crítica/Alta/Média/Baixa	Crítica
Observações	Qualquer informação adicional	“Teste executado em Chrome 120”

5.2. Matriz de Rastreabilidade (RTM)

A **Matriz de Rastreabilidade (Requirements Traceability Matrix - RTM)** garante que cada requisito tenha pelo menos um caso de teste associado.

Exemplo de RTM:

ID Requisito	Descrição do Requisito	ID Caso de Teste	Status	Observações
REQ-001	Usuário deve fazer login com email e senha	TC-001, TC-002	Coberto	2 casos de teste
REQ-002	Usuário deve receber erro com credenciais inválidas	TC-003, TC-004	Coberto	Testa email inválido e senha inválida
REQ-003	Usuário deve poder recuperar senha	TC-005, TC-006	Coberto	Testa link válido e expirado
REQ-004	Sessão deve expirar após 30 minutos	TC-007	Coberto	Teste de timeout
REQ-005	Senha deve ter mínimo 8 caracteres	TC-008	Coberto	Validação de força de senha

Benefícios da RTM:

- Garante cobertura completa de requisitos
- Identifica requisitos sem testes
- Facilita rastreamento de defeitos para requisitos
- Ajuda na análise de impacto de mudanças

5.3. Exemplo Prático de Casos de Teste

CASOS DE TESTE - Módulo de Login

```
# CASO DE TESTE TC-001
## Título: Login com credenciais válidas

**ID**: TC-001
**Prioridade**: Crítica
**Módulo**: Autenticação

### Pré-condições:
- Usuário não está logado
- Conta "user@example.com" existe no sistema
- Senha da conta é "senha123"

### Passos:
1. Abrir navegador e acessar https://example.com/login
2. Digitar "user@example.com" no campo de email
3. Digitar "senha123" no campo de senha
4. Clicar no botão "Login"

### Dados de Entrada:
- Email: user@example.com
- Senha: senha123

### Resultado Esperado:
- Usuário é redirecionado para a página de dashboard
- URL muda para https://example.com/dashboard
- Nome do usuário aparece no canto superior direito
- Mensagem de sucesso é exibida (opcional)

### Resultado Atual:
- ✓ Usuário redirecionado para dashboard
- ✓ URL correta
- ✓ Nome do usuário exibido
- ✓ Sem mensagem de erro

### Status: PASS
### Data de Execução: 2025-03-01
### Testador: João Silva
### Navegador: Chrome 120
### SO: Windows 10

---

# CASO DE TESTE TC-002
## Título: Login com email inválido
```

ID: TC-002
Prioridade: Alta
Módulo: Autenticação

Pré-condições:
- Usuário não está logado
- Email "invalido@example.com" não existe no sistema

Passos:
1. Abrir navegador e acessar https://example.com/login
2. Digitar "invalido@example.com" no campo de email
3. Digitar qualquer senha no campo de senha
4. Clicar no botão "Login"

Dados de Entrada:
- Email: invalido@example.com
- Senha: qualquersenha

Resultado Esperado:
- Usuário permanece na página de login
- Mensagem de erro é exibida: "Email ou senha incorretos"
- Campo de senha é limpo
- Campo de email mantém o valor digitado

Resultado Atual:
- ✓ Usuário permanece na página de login
- ✓ Mensagem de erro exibida
- ✓ Campo de senha limpo
- ✓ Campo de email mantém valor

Status: PASS
Data de Execução: 2025-03-01
Testador: João Silva

CASO DE TESTE TC-003
Título: Login com senha inválida

ID: TC-003
Prioridade: Alta
Módulo: Autenticação

Pré-condições:
- Usuário não está logado
- Conta "user@example.com" existe no sistema

```
- Senha correta é "senha123"

### Passos:
1. Abrir navegador e acessar https://example.com/login
2. Digitar "user@example.com" no campo de email
3. Digitar "senhaerrada" no campo de senha
4. Clicar no botão "Login"

### Dados de Entrada:
- Email: user@example.com
- Senha: senhaerrada

### Resultado Esperado:
- Usuário permanece na página de login
- Mensagem de erro é exibida: "Email ou senha incorretos"
- Campo de senha é limpo
- Campo de email mantém o valor digitado

### Resultado Atual:
- ✓ Usuário permanece na página de login
- ✓ Mensagem de erro exibida
- ✓ Campo de senha limpo
- ✓ Campo de email mantém valor

### Status: PASS
### Data de Execução: 2025-03-01
### Testador: João Silva
```

PARTE III: FERRAMENTAS E AUTOMAÇÃO DE TESTES

Capítulo 6: Jira - Gestão de Testes e Defeitos

O **Jira** é a ferramenta mais popular para rastreamento de issues, gestão de projetos e organização de testes em ambientes ágeis.

6.1. Configuração e Tipos de Issues

Tipos de Issues Essenciais para QA

Tipo de Issue	Descrição	Exemplo
Bug	Defeito encontrado durante testes	“Botão de login não funciona em Safari”
Story	Requisito de negócio a ser desenvolvido	“Como usuário, quero fazer login com email e senha”
Task	Tarefa técnica ou administrativa	“Configurar ambiente de staging”
Test Case	Caso de teste (com plugin Xray ou Zephyr)	“Validar login com credenciais válidas”
Sub-task	Subtarefa de uma issue maior	“Escrever testes de unidade para login”
Epic	Conjunto grande de funcionalidades	“Implementar sistema de autenticação”

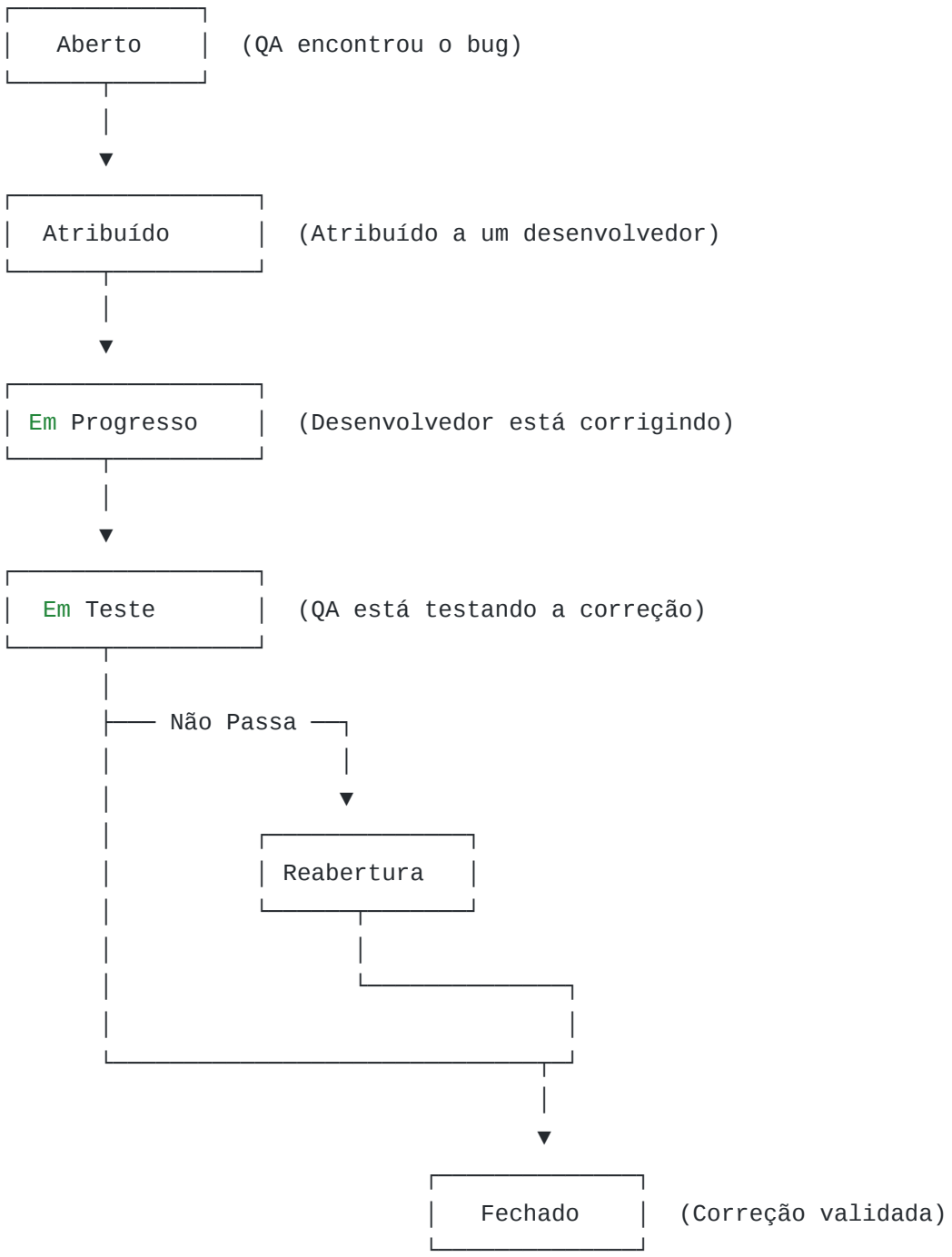
Campos Importantes para QA

Campo	Descrição	Valores Típicos
Summary	Título da issue	"Login não funciona em Safari"
Description	Descrição detalhada	"Passos para reproduzir, resultado esperado vs. atual"
Priority	Prioridade da issue	Blocker, Critical, High, Medium, Low
Severity	Impacto do defeito	Critical, Major, Minor, Trivial
Assignee	Pessoa responsável	Nome do desenvolvedor ou QA
Reporter	Pessoa que reportou	Nome do QA que encontrou o defeito
Status	Estado atual	To Do, In Progress, In Test, Done
Component	Módulo afetado	Login, Carrinho, Pagamento
Labels	Tags adicionais	regression, security, performance
Fix Version	Versão que corrige	v2.0, v2.1
Environment	Onde o defeito foi encontrado	Chrome 120, Windows 10, Staging

6.2. Workflow de Defeitos

Um **workflow** define os estados pelos quais uma issue passa durante seu ciclo de vida.

Workflow Típico para Bugs:



Transições e Regras:

De	Para	Quem	Condição
Aberto	Atribuído	QA/Gerente	Bug é válido e tem impacto
Atribuído	Em Progresso	Desenvolvedor	Desenvolvedor começa a trabalhar
Em Progresso	Em Teste	Desenvolvedor	Correção está pronta para teste
Em Teste	Fechado	QA	Correção foi validada
Em Teste	Reabertura	QA	Correção não funciona ou causa regressão
Reabertura	Em Progresso	Desenvolvedor	Desenvolvedor continua trabalhando

6.3. Dashboards e Relatórios

Dashboard de Qualidade

Um dashboard típico para QA mostra:

Métricas Principais:

- **Total de Bugs**: 45 (10 Críticos, 15 Altos, 20 Médios)
- **Taxa de Resolução**: 80% (36 de 45 bugs corrigidos)
- **Tempo Médio de Resolução**: 3 dias
- **Bugs Abertos por Prioridade**: Gráfico de pizza
- **Bugs por Módulo**: Gráfico de barras
- **Tendência de Bugs**: Gráfico de linha

Relatório de Teste

Um relatório de teste típico inclui:

RELATÓRIO DE TESTE - Sprint 15

Resumo Executivo
- **Período**: 01-15 de março de 2025
- **Módulos Testados**: Login, Carrinho, Pagamento
- **Total de Casos de Teste**: 150
- **Casos Executados**: 145 (96.7%)
- **Taxa de Sucesso**: 92% (133 de 145)
- **Bugs Encontrados**: 12 (8 Críticos, 4 Altos)

Detalhes por Módulo

Login
- Casos de Teste: 40
- Executados: 40
- Passaram: 38
- Falharam: 2
- Taxa de Sucesso: 95%
- Bugs Críticos: 1

Carrinho
- Casos de Teste: 60
- Executados: 60
- Passaram: 56
- Falharam: 4
- Taxa de Sucesso: 93%
- Bugs Críticos: 3

Pagamento
- Casos de Teste: 50
- Executados: 45
- Passaram: 39
- Falharam: 6
- Taxa de Sucesso: 87%
- Bugs Críticos: 4

Bugs Encontrados

ID	Título	Prioridade	Status
BUG-001	Login não funciona em Safari	Crítica	Fechado
BUG-002	Carrinho não persiste após logout	Crítica	Em Teste
BUG-003	Cupom de desconto não aplica	Alta	Em Progresso
BUG-004	Pagamento falha com cartão de crédito	Crítica	Fechado

```
## Recomendações

1. Aumentar cobertura de testes para o módulo de Pagamento
2. Implementar testes de regressão automatizados
3. Melhorar testes de compatibilidade com navegadores
4. Realizar teste de carga antes da próxima release

## Assinado por
QA Lead: João Silva
Data: 2025-03-15
```

6.4. Integração com Ferramentas de Teste

O Jira pode ser integrado com ferramentas de automação para criar issues automaticamente quando testes falham.

Exemplo de Integração Cypress + Jira:

```
// cypress/plugins/index.js
const axios = require('axios');

module.exports = (on, config) => {
  on('task', {
    createJiraIssue(data) {
      const jiraUrl = 'https://seu-jira.atlassian.net/rest/api/3/issue';
      const auth = Buffer.from('seu-email@example.com:seu-token-
api').toString('base64');

      return axios.post(jiraUrl, {
        fields: {
          project: { key: 'QA' },
          summary: data.title,
          description: data.description,
          issuetype: { name: 'Bug' },
          priority: { name: 'High' },
          labels: ['cypress', 'automated']
        }
      }, {
        headers: {
          'Authorization': `Basic ${auth}`,
          'Content-Type': 'application/json'
        }
      });
    }
  });
};

// cypress/e2e/login.cy.js
describe('Login Tests', () => {
  it('Should create Jira issue on failure', () => {
    cy.visit('https://example.com/login');
    cy.get('input[name="email"]').type('user@example.com');
    cy.get('input[name="password"]').type('senha123');
    cy.get('button[type="submit"]').click();

    cy.url().then(url => {
      if (!url.includes('/dashboard')) {
        cy.task('createJiraIssue', {
          title: 'Login test failed',
          description: 'Login button did not redirect to dashboard'
        });
      }
    });
```

```
  });
});
```

Capítulo 7: JavaScript e Cypress - Automação Web Moderna

7.1. Fundamentos de JavaScript para QA

JavaScript é a linguagem de programação da web e é essencial para automação com Cypress.

Conceitos Básicos

Variáveis e Tipos de Dados:

```
// Variáveis
let email = 'user@example.com';  // let (recomendado)
const password = 'senha123';      // const (imutável)
var username = 'João';            // var (evitar)

// Tipos de dados
let numero = 42;                  // Number
let texto = 'Hello';              // String
let booleano = true;              // Boolean
let nulo = null;                  // Null
let indefinido;                   // Undefined
let objeto = { name: 'João' };    // Object
let array = [1, 2, 3];            // Array
```

Operadores:

```
// Aritméticos
10 + 5;    // 15
10 - 5;    // 5
10 * 5;    // 50
10 / 5;    // 2
10 % 3;    // 1 (resto da divisão)

// Comparação
5 == '5';   // true (comparação de valor)
5 === '5';  // false (comparação de tipo e valor)
5 != '5';   // false
5 !== '5';  // true
5 > 3;      // true
5 < 3;      // false
5 >= 5;     // true
5 <= 5;     // true

// Lógicos
true && false;  // false (AND)
true || false;  // true (OR)
!true;          // false (NOT)
```

Estruturas de Controle:

```javascript
// if/else
if (idade >= 18) {
  console.log('Maior de idade');
} else if (idade >= 13) {
  console.log('Adolescente');
} else {
  console.log('Criança');
}

// switch
switch (dia) {
  case 'segunda':
    console.log('Início da semana');
    break;
  case 'sexta':
    console.log('Quase fim de semana');
    break;
  default:
    console.log('Dia comum');
}

// for
for (let i = 0; i < 5; i++) {
  console.log(i);  // 0, 1, 2, 3, 4
}

// while
let contador = 0;
while (contador < 5) {
  console.log(contador);
  contador++;
}

// forEach
const numeros = [1, 2, 3];
numeros.forEach(num => {
  console.log(num);
});
```

Funções:

```
// Função básica
function saudacao(nome) {
  return `Olá, ${nome}!`;
}
console.log(saudacao('João'));  // "Olá, João!"

// Arrow function (moderna)
const saudacao2 = (nome) => {
  return `Olá, ${nome}!`;
};

// Arrow function (simplificada)
const saudacao3 = nome => `Olá, ${nome}!`;

// Função com múltiplos parâmetros
function calcular(a, b, operacao) {
  if (operacao === '+') return a + b;
  if (operacao === '-') return a - b;
  if (operacao === '*') return a * b;
  if (operacao === '/') return a / b;
}
console.log(calcular(10, 5, '+'));  // 15
```

Objetos e Arrays:

```
// Objeto
const usuario = {
  nome: 'João',
  email: 'joao@example.com',
  idade: 30,
  endereco: {
    rua: 'Rua A',
    cidade: 'São Paulo'
  }
};

// Acessando propriedades
console.log(usuario.nome);              // "João"
console.log(usuario['email']);          // "joao@example.com"
console.log(usuario.endereco.cidade);   // "São Paulo"

// Array
const numeros = [1, 2, 3, 4, 5];
console.log(numeros[0]);    // 1
console.log(numeros.length); // 5

// Métodos de Array
numeros.push(6);            // [1, 2, 3, 4, 5, 6]
numeros.pop();              // [1, 2, 3, 4, 5]
numeros.map(n => n * 2);    // [2, 4, 6, 8, 10]
numeros.filter(n => n > 2); // [3, 4, 5]
```

7.2. Introdução ao Cypress

O que é Cypress?

Cypress é um framework de teste end-to-end (E2E) moderno, desenvolvido especificamente para testes de aplicações web. Ele oferece uma experiência de teste superior com:

- Execução rápida e confiável
- Interface visual intuitiva
- Debugging fácil
- Documentação excelente

Instalação:

```
# Criar projeto Node.js
npm init -y

# Instalar Cypress
npm install --save-dev cypress

# Abrir Cypress
npx cypress open
```

Estrutura de Projeto:

```
projeto/
├── cypress/
│   ├── e2e/              # Testes E2E
│   │   └── login.cy.js
│   ├── support/          # Arquivos de suporte
│   │   ├── commands.js
│   │   └── e2e.js
│   └── fixtures/         # Dados de teste
│       └── users.json
├── cypress.config.js     # Configuração do Cypress
└── package.json
```

7.3. Seletores e Localizadores

Seletores CSS:

```
// Por ID
cy.get('#login-button')

// Por classe
cy.get('.error-message')

// Por atributo
cy.get('input[type="email"]')
cy.get('button[data-testid="submit"]')

// Por tag
cy.get('button')

// Combinações
cy.get('form input[type="email"]')
cy.get('.container > .button')
```

Seletores XPath:

```
// XPath (menos recomendado, mas possível)
cy.xpath('//button[@id="login-button"]')
cy.xpath('//input[@type="email"]')
cy.xpath('//div[contains(text(), "Error")]')
```

Melhores Práticas:

```
// ✓ BOM: Usar data-testid
cy.get('[data-testid="email-input"]').type('user@example.com');

// ✗ RUIM: Usar seletores frágeis
cy.get('input:nth-child(2)').type('user@example.com');

// ✓ BOM: Usar get com texto
cy.contains('button', 'Login').click();

// ✗ RUIM: Usar seletores muito específicos
cy.get('body > div:nth-child(1) > form > input:nth-child(2)');
```

7.4. Exemplos de Scripts Cypress

Exemplo 1: Teste de Login

```
// cypress/e2e/login.cy.js
describe('Login Tests', () => {
  beforeEach(() => {
    cy.visit('https://example.com/login');
  });

  it('Should login successfully with valid credentials', () => {
    cy.get('[data-testid="email-input"]')
      .type('user@example.com');

    cy.get('[data-testid="password-input"]')
      .type('senha123');

    cy.get('[data-testid="login-button"]')
      .click();

    cy.url().should('include', '/dashboard');
    cy.get('[data-testid="user-name"]')
      .should('contain', 'João Silva');
  });

  it('Should show error with invalid email', () => {
    cy.get('[data-testid="email-input"]')
      .type('invalido@example.com');

    cy.get('[data-testid="password-input"]')
      .type('senha123');

    cy.get('[data-testid="login-button"]')
      .click();

    cy.get('[data-testid="error-message"]')
      .should('contain', 'Email ou senha incorretos');
  });

  it('Should show error with invalid password', () => {
    cy.get('[data-testid="email-input"]')
      .type('user@example.com');

    cy.get('[data-testid="password-input"]')
      .type('senhaerrada');

    cy.get('[data-testid="login-button"]')
      .click();
```

```
    cy.get('[data-testid="error-message"]')
      .should('contain', 'Email ou senha incorretos');
  });

  it('Should require email field', () => {
    cy.get('[data-testid="password-input"]')
      .type('senha123');

    cy.get('[data-testid="login-button"]')
      .click();

    cy.get('[data-testid="email-error"]')
      .should('contain', 'Email é obrigatório');
  });
});
```

Exemplo 2: Teste de Carrinho de Compras

```
// cypress/e2e/shopping-cart.cy.js
describe('Shopping Cart Tests', () => {
  beforeEach(() => {
    cy.visit('https://example.com');
    cy.login('user@example.com', 'senha123');
  });

  it('Should add item to cart', () => {
    cy.get('[data-testid="product-1"]')
      .click();

    cy.get('[data-testid="add-to-cart-button"]')
      .click();

    cy.get('[data-testid="cart-count"]')
      .should('contain', '1');

    cy.get('[data-testid="success-message"]')
      .should('contain', 'Produto adicionado ao carrinho');
  });

  it('Should remove item from cart', () => {
    // Adicionar item
    cy.get('[data-testid="product-1"]').click();
    cy.get('[data-testid="add-to-cart-button"]').click();

    // Ir para carrinho
    cy.get('[data-testid="cart-link"]').click();

    // Remover item
    cy.get('[data-testid="remove-button"]').click();

    cy.get('[data-testid="empty-cart-message"]')
      .should('contain', 'Seu carrinho está vazio');
  });

  it('Should apply discount coupon', () => {
    // Adicionar item
    cy.get('[data-testid="product-1"]').click();
    cy.get('[data-testid="add-to-cart-button"]').click();

    // Ir para carrinho
    cy.get('[data-testid="cart-link"]').click();

    // Aplicar cupom
```

```
    cy.get('[data-testid="coupon-input"]')
      .type('DESCONTO10');

    cy.get('[data-testid="apply-coupon-button"]')
      .click();

    cy.get('[data-testid="discount-message"]')
      .should('contain', '10% de desconto aplicado');

    // Verificar que o total foi reduzido
    cy.get('[data-testid="total-price"]')
      .should('contain', 'R$ 90,00');
  });

  it('Should update item quantity', () => {
    // Adicionar item
    cy.get('[data-testid="product-1"]').click();
    cy.get('[data-testid="add-to-cart-button"]').click();

    // Ir para carrinho
    cy.get('[data-testid="cart-link"]').click();

    // Aumentar quantidade
    cy.get('[data-testid="quantity-input"]')
      .clear()
      .type('3');

    cy.get('[data-testid="update-button"]')
      .click();

    cy.get('[data-testid="total-price"]')
      .should('contain', 'R$ 300,00');
  });
});
```

Exemplo 3: Teste com Dados Dinâmicos

```
// cypress/fixtures/users.json
[
  {
    "email": "user1@example.com",
    "password": "senha123",
    "name": "Usuário 1"
  },
  {
    "email": "user2@example.com",
    "password": "senha456",
    "name": "Usuário 2"
  }
]

// cypress/e2e/dynamic-tests.cy.js
describe('Dynamic Login Tests', () => {
  beforeEach(() => {
    cy.fixture('users').as('users');
  });

  it('Should login with multiple users', function() {
    this.users.forEach(user => {
      cy.visit('https://example.com/login');

      cy.get('[data-testid="email-input"]')
        .type(user.email);

      cy.get('[data-testid="password-input"]')
        .type(user.password);

      cy.get('[data-testid="login-button"]')
        .click();

      cy.url().should('include', '/dashboard');
      cy.get('[data-testid="user-name"]')
        .should('contain', user.name);

      // Logout
      cy.get('[data-testid="logout-button"]')
        .click();
    });
  });
});
```

7.5. Boas Práticas e Padrões

Page Object Model (POM):

O Page Object Model é um padrão de design que melhora a manutenibilidade dos testes.

```
// cypress/support/pages/LoginPage.js
class LoginPage {
  visit() {
    cy.visit('https://example.com/login');
  }

  fillEmail(email) {
    cy.get('[data-testid="email-input"]').type(email);
    return this;
  }

  fillPassword(password) {
    cy.get('[data-testid="password-input"]').type(password);
    return this;
  }

  clickLoginButton() {
    cy.get('[data-testid="login-button"]').click();
    return this;
  }

  getErrorMessage() {
    return cy.get('[data-testid="error-message"]');
  }

  login(email, password) {
    this.fillEmail(email);
    this.fillPassword(password);
    this.clickLoginButton();
    return this;
  }
}

export default new LoginPage();

// cypress/e2e/login-pom.cy.js
import LoginPage from '../support/pages/LoginPage';

describe('Login Tests with POM', () => {
  beforeEach(() => {
    LoginPage.visit();
  });

  it('Should login successfully', () => {
    LoginPage.login('user@example.com', 'senha123');
```

```
    cy.url().should('include', '/dashboard');
  });

  it('Should show error with invalid credentials', () => {
    LoginPage.login('invalid@example.com', 'wrongpassword');
    LoginPage.getErrorMessage()
      .should('contain', 'Email ou senha incorretos');
  });
});
```

Custom Commands:

```
// cypress/support/commands.js
Cypress.Commands.add('login', (email, password) => {
  cy.visit('https://example.com/login');
  cy.get('[data-testid="email-input"]').type(email);
  cy.get('[data-testid="password-input"]').type(password);
  cy.get('[data-testid="login-button"]').click();
  cy.url().should('include', '/dashboard');
});

Cypress.Commands.add('logout', () => {
  cy.get('[data-testid="logout-button"]').click();
  cy.url().should('include', '/login');
});

// Uso nos testes
describe('Tests with Custom Commands', () => {
  it('Should use custom login command', () => {
    cy.login('user@example.com', 'senha123');
    cy.get('[data-testid="user-name"]').should('be.visible');
  });

  it('Should use custom logout command', () => {
    cy.login('user@example.com', 'senha123');
    cy.logout();
  });
});
```

Capítulo 8: Python e Robot Framework - Automação Versátil

8.1. Fundamentos de Python para QA

Python é uma linguagem poderosa e versátil para automação de testes.

Conceitos Básicos

Variáveis e Tipos de Dados:

```
# Variáveis
email = 'user@example.com'
password = 'senha123'
idade = 30
altura = 1.75
ativo = True

# Tipos de dados
type(email)        # <class 'str'>
type(idade)        # <class 'int'>
type(altura)       # <class 'float'>
type(ativo)        # <class 'bool'>

# Conversão de tipos
str(idade)         # '30'
int('30')          # 30
float('1.75')      # 1.75
bool(1)            # True
```

Strings:

```python
# Concatenação
nome = 'João'
sobrenome = 'Silva'
nome_completo = nome + ' ' + sobrenome

# f-strings (recomendado)
mensagem = f'Olá, {nome}!'
print(mensagem)    # 'Olá, João!'

# Métodos de string
texto = 'Hello World'
texto.lower()      # 'hello world'
texto.upper()      # 'HELLO WORLD'
texto.replace('World', 'Python')  # 'Hello Python'
texto.split()      # ['Hello', 'World']
```

Listas:

```python
# Criação
numeros = [1, 2, 3, 4, 5]
nomes = ['João', 'Maria', 'Pedro']

# Acesso
numeros[0]         # 1
numeros[-1]        # 5 (último elemento)
numeros[1:3]       # [2, 3] (slice)

# Métodos
numeros.append(6)  # [1, 2, 3, 4, 5, 6]
numeros.remove(3)  # [1, 2, 4, 5, 6]
len(numeros)       # 5
```

Dicionários:

```python
# Criação
usuario = {
  'nome': 'João',
  'email': 'joao@example.com',
  'idade': 30
}

# Acesso
usuario['nome']    # 'João'
usuario.get('email')  # 'joao@example.com'

# Adição
usuario['telefone'] = '123456789'

# Iteração
for chave, valor in usuario.items():
  print(f'{chave}: {valor}')
```

Estruturas de Controle:

```python
# if/elif/else
idade = 20
if idade >= 18:
  print('Maior de idade')
elif idade >= 13:
  print('Adolescente')
else:
  print('Criança')

# for
for i in range(5):
  print(i)  # 0, 1, 2, 3, 4

# while
contador = 0
while contador < 5:
  print(contador)
  contador += 1

# List comprehension
numeros = [1, 2, 3, 4, 5]
pares = [n for n in numeros if n % 2 == 0]
print(pares)  # [2, 4]
```

Funções:

```python
# Função básica
def saudacao(nome):
  return f'Olá, {nome}!'

print(saudacao('João'))  # 'Olá, João!'

# Função com múltiplos parâmetros
def calcular(a, b, operacao='+'):
  if operacao == '+':
    return a + b
  elif operacao == '-':
    return a - b
  elif operacao == '*':
    return a * b
  elif operacao == '/':
    return a / b

print(calcular(10, 5))      # 15
print(calcular(10, 5, '-')) # 5

# Função com *args e **kwargs
def funcao_flexivel(*args, **kwargs):
  print(args)    # tupla de argumentos
  print(kwargs)  # dicionário de argumentos nomeados

funcao_flexivel(1, 2, 3, nome='João', idade=30)
# (1, 2, 3)
# {'nome': 'João', 'idade': 30}
```

8.2. Introdução ao Robot Framework

O que é Robot Framework?

Robot Framework é um framework de automação de testes genérico e extensível, baseado em Python, que usa uma sintaxe simples e legível em linguagem natural.

Instalação:

```
# Instalar Robot Framework
pip install robotframework

# Instalar SeleniumLibrary (para testes web)
pip install robotframework-seleniumlibrary

# Instalar RequestsLibrary (para testes de API)
pip install robotframework-requestslibrary

# Instalar DatabaseLibrary (para testes de banco de dados)
pip install robotframework-databaselibrary
```

Estrutura de Projeto:

```
projeto/
├── tests/
│   ├── login.robot
│   ├── carrinho.robot
│   └── pagamento.robot
├── resources/
│   ├── keywords.robot
│   └── variables.robot
├── results/
│   ├── report.html
│   ├── log.html
│   └── output.xml
└── robot.ini
```

8.3. Bibliotecas Essenciais

Biblioteca	Uso	Exemplo
SeleniumLibrary	Automação de testes web	`Open Browser`, `Click Button`, `Input Text`
RequestsLibrary	Testes de API REST	`GET`, `POST`, `PUT`, `DELETE`
DatabaseLibrary	Testes de banco de dados	`Connect To Database`, `Query`, `Execute SQL`
BuiltIn	Funções nativas	`Log`, `Sleep`, `Should Be Equal`
Collections	Manipulação de listas e dicionários	`Append To List`, `Get From Dictionary`
String	Manipulação de strings	`Get Substring`, `Replace String`

8.4. Exemplos de Scripts Robot Framework

Exemplo 1: Teste de Login

```
*** Settings ***
Library    SeleniumLibrary

*** Variables ***
${BROWSER}        Chrome
${URL}            https://example.com/login
${EMAIL}          user@example.com
${PASSWORD}       senha123
${EMAIL_INPUT}    [data-testid="email-input"]
${PASSWORD_INPUT} [data-testid="password-input"]
${LOGIN_BUTTON}   [data-testid="login-button"]

*** Test Cases ***
Test Login With Valid Credentials
    Open Browser    ${URL}    ${BROWSER}
    Input Text      ${EMAIL_INPUT}    ${EMAIL}
    Input Text      ${PASSWORD_INPUT}    ${PASSWORD}
    Click Button    ${LOGIN_BUTTON}
    Location Should Contain    /dashboard
    Close Browser

Test Login With Invalid Email
    Open Browser    ${URL}    ${BROWSER}
    Input Text      ${EMAIL_INPUT}    invalid@example.com
    Input Text      ${PASSWORD_INPUT}    ${PASSWORD}
    Click Button    ${LOGIN_BUTTON}
    Page Should Contain    Email ou senha incorretos
    Close Browser

Test Login With Invalid Password
    Open Browser    ${URL}    ${BROWSER}
    Input Text      ${EMAIL_INPUT}    ${EMAIL}
    Input Text      ${PASSWORD_INPUT}    wrongpassword
    Click Button    ${LOGIN_BUTTON}
    Page Should Contain    Email ou senha incorretos
    Close Browser

*** Keywords ***
Login With Credentials
    [Arguments]    ${email}    ${password}
    Open Browser    ${URL}    ${BROWSER}
    Input Text      ${EMAIL_INPUT}    ${email}
    Input Text      ${PASSWORD_INPUT}    ${password}
    Click Button    ${LOGIN_BUTTON}
```

Exemplo 2: Teste de API com RequestsLibrary

```
 *** Settings ***
Library    RequestsLibrary
Library    Collections

*** Variables ***
${BASE_URL}    https://api.example.com
${USER_EMAIL}  user@example.com
${USER_PASSWORD}    senha123

*** Test Cases ***
Test Get Users
    ${response}=    GET    ${BASE_URL}/users
    Should Be Equal As Integers    ${response.status_code}    200
    Should Contain    ${response.text}    user@example.com

Test Create User
    ${data}=    Create Dictionary
    ...    email=newuser@example.com
    ...    password=senha123
    ...    name=New User

    ${response}=    POST    ${BASE_URL}/users
    ...    json=${data}

    Should Be Equal As Integers    ${response.status_code}    201
    Should Contain    ${response.text}    newuser@example.com

Test Update User
    ${data}=    Create Dictionary
    ...    name=Updated Name

    ${response}=    PUT    ${BASE_URL}/users/1
    ...    json=${data}

    Should Be Equal As Integers    ${response.status_code}    200

Test Delete User
    ${response}=    DELETE    ${BASE_URL}/users/1
    Should Be Equal As Integers    ${response.status_code}    204

Test Login API
    ${data}=    Create Dictionary
    ...    email=${USER_EMAIL}
    ...    password=${USER_PASSWORD}

    ${response}=    POST    ${BASE_URL}/login
```

```
    ...    json=${data}

    Should Be Equal As Integers    ${response.status_code}    200
    Should Contain    ${response.text}    token
```

8.5. Integração com CI/CD

Executar Robot Framework em CI/CD:

```
# Executar todos os testes
robot tests/

# Executar teste específico
robot tests/login.robot

# Executar com tags
robot --include smoke tests/

# Gerar relatório
robot --outputdir results tests/
```

Exemplo com GitHub Actions:

```yaml
# .github/workflows/robot-tests.yml
name: Robot Framework Tests

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2

    - name: Set up Python
      uses: actions/setup-python@v2
      with:
        python-version: 3.9

    - name: Install dependencies
      run: |
        pip install robotframework
        pip install robotframework-seleniumlibrary
        pip install robotframework-requestslibrary

    - name: Run Robot Framework tests
      run: robot --outputdir results tests/

    - name: Upload results
      if: always()
      uses: actions/upload-artifact@v2
      with:
        name: robot-results
        path: results/
```

Capítulo 9: Testes de API - A Espinha Dorsal do Software

9.1. Fundamentos de REST e HTTP

O que é REST?

REST (Representational State Transfer) é um estilo arquitetural para projetar aplicações de rede. Ele usa HTTP como protocolo de comunicação e é baseado em recursos.

Conceitos Principais:

- **Recursos:** Entidades que podem ser manipuladas (usuários, produtos, pedidos)
- **Métodos HTTP:** Ações que podem ser realizadas nos recursos
- **Representações:** Formato dos dados (JSON, XML)
- **Stateless:** Cada requisição contém todas as informações necessárias

9.2. Métodos HTTP e Códigos de Status

Métodos HTTP:

Método	Descrição	Exemplo	Idempotente
GET	Recuperar dados	GET /users/1	Sim
POST	Criar novo recurso	POST /users	Não
PUT	Atualizar recurso existente	PUT /users/1	Sim
PATCH	Atualização parcial	PATCH /users/1	Não
DELETE	Deletar recurso	DELETE /users/1	Sim
HEAD	Como GET, mas sem corpo	HEAD /users	Sim
OPTIONS	Descrever opções de comunicação	OPTIONS /users	Sim

Códigos de Status HTTP:

Código	Categoria	Significado	Exemplo
200	2xx (Sucesso)	OK - Requisição bem-sucedida	GET /users retorna 200
201	2xx (Sucesso)	Created - Recurso criado	POST /users retorna 201
204	2xx (Sucesso)	No Content - Sem corpo na resposta	DELETE /users/1 retorna 204
400	4xx (Erro do Cliente)	Bad Request - Requisição inválida	POST /users com dados inválidos
401	4xx (Erro do Cliente)	Unauthorized - Autenticação necessária	GET /users sem token
403	4xx (Erro do Cliente)	Forbidden - Acesso negado	GET /admin sem permissão
404	4xx (Erro do Cliente)	Not Found - Recurso não encontrado	GET /users/999
500	5xx (Erro do Servidor)	Internal Server Error	Erro no servidor
503	5xx (Erro do Servidor)	Service Unavailable	Servidor em manutenção

9.3. Postman - Testes de API Manuais

O que é Postman?

Postman é uma ferramenta popular para testar, documentar e monitorar APIs. Permite criar requisições HTTP de forma visual e organizar testes em coleções.

Instalação:

1. Baixar em https://www.postman.com/downloads/
2. Criar conta gratuita
3. Abrir aplicação

Estrutura Básica:

```
Workspace
├── Collections
│   ├── User API
│   │   ├── GET /users
│   │   ├── POST /users
│   │   ├── PUT /users/:id
│   │   └── DELETE /users/:id
│   └── Product API
│       ├── GET /products
│       └── POST /products
└── Environments
    ├── Development
    ├── Staging
    └── Production
```

Exemplo de Requisição GET:

```
GET https://api.example.com/users/1

Headers:
Authorization: Bearer token123
Content-Type: application/json

Response:
{
  "id": 1,
  "name": "João Silva",
  "email": "joao@example.com",
  "age": 30
}
```

Exemplo de Requisição POST:

```
POST https://api.example.com/users

Headers:
Content-Type: application/json

Body:
{
  "name": "Maria Silva",
  "email": "maria@example.com",
  "age": 28
}

Response:
{
  "id": 2,
  "name": "Maria Silva",
  "email": "maria@example.com",
  "age": 28,
  "createdAt": "2025-03-01T10:00:00Z"
}
```

9.4. Automação de Testes de API

Exemplo com Python e Requests:

```python
import requests
import json

# URL base da API
BASE_URL = 'https://api.example.com'

# Headers
HEADERS = {
    'Content-Type': 'application/json',
    'Authorization': 'Bearer token123'
}

# Teste GET
def test_get_user():
    response = requests.get(f'{BASE_URL}/users/1', headers=HEADERS)
    assert response.status_code == 200
    data = response.json()
    assert data['id'] == 1
    assert data['name'] == 'João Silva'
    print('✓ GET /users/1 passou')

# Teste POST
def test_create_user():
    payload = {
        'name': 'Maria Silva',
        'email': 'maria@example.com',
        'age': 28
    }
    response = requests.post(f'{BASE_URL}/users', json=payload,
headers=HEADERS)
    assert response.status_code == 201
    data = response.json()
    assert data['name'] == 'Maria Silva'
    print('✓ POST /users passou')

# Teste PUT
def test_update_user():
    payload = {
        'name': 'João Silva Atualizado'
    }
    response = requests.put(f'{BASE_URL}/users/1', json=payload,
headers=HEADERS)
    assert response.status_code == 200
    data = response.json()
    assert data['name'] == 'João Silva Atualizado'
```

```
    print('✓ PUT /users/1 passou')

# Teste DELETE
def test_delete_user():
    response = requests.delete(f'{BASE_URL}/users/1', headers=HEADERS)
    assert response.status_code == 204
    print('✓ DELETE /users/1 passou')

# Executar testes
if __name__ == '__main__':
    test_get_user()
    test_create_user()
    test_update_user()
    test_delete_user()
    print('\n✓ Todos os testes passaram!')
```

Exemplo com Cypress:

```
// cypress/e2e/api-tests.cy.js
describe('API Tests', () => {
  const BASE_URL = 'https://api.example.com';

  it('Should GET user successfully', () => {
    cy.request('GET', `${BASE_URL}/users/1`)
      .then(response => {
        expect(response.status).to.equal(200);
        expect(response.body).to.have.property('id', 1);
        expect(response.body).to.have.property('name', 'João Silva');
      });
  });

  it('Should POST user successfully', () => {
    const payload = {
      name: 'Maria Silva',
      email: 'maria@example.com',
      age: 28
    };

    cy.request('POST', `${BASE_URL}/users`, payload)
      .then(response => {
        expect(response.status).to.equal(201);
        expect(response.body).to.have.property('name', 'Maria Silva');
      });
  });

  it('Should PUT user successfully', () => {
    const payload = {
      name: 'João Silva Atualizado'
    };

    cy.request('PUT', `${BASE_URL}/users/1`, payload)
      .then(response => {
        expect(response.status).to.equal(200);
        expect(response.body).to.have.property('name', 'João Silva
Atualizado');
      });
  });

  it('Should DELETE user successfully', () => {
    cy.request('DELETE', `${BASE_URL}/users/1`)
      .then(response => {
        expect(response.status).to.equal(204);
      });
```

```
  });
});
```

9.5. Validação de Respostas JSON

Estrutura JSON:

```
{
  "id": 1,
  "name": "João Silva",
  "email": "joao@example.com",
  "age": 30,
  "ativo": true,
  "endereco": {
    "rua": "Rua A",
    "cidade": "São Paulo",
    "cep": "01234-567"
  },
  "telefones": [
    "11987654321",
    "1133334444"
  ]
}
```

Validação em Python:

```python
import requests
import json

response = requests.get('https://api.example.com/users/1')
data = response.json()

# Validar estrutura
assert 'id' in data
assert 'name' in data
assert 'email' in data

# Validar tipos
assert isinstance(data['id'], int)
assert isinstance(data['name'], str)
assert isinstance(data['ativo'], bool)

# Validar valores
assert data['id'] == 1
assert data['name'] == 'João Silva'
assert data['age'] > 0

# Validar objetos aninhados
assert data['endereco']['cidade'] == 'São Paulo'

# Validar arrays
assert len(data['telefones']) == 2
assert '11987654321' in data['telefones']

print('✓ Todas as validações passaram!')
```

Validação em Cypress:

```
cy.request('GET', 'https://api.example.com/users/1')
  .then(response => {
    // Validar status
    expect(response.status).to.equal(200);

    // Validar estrutura
    expect(response.body).to.have.property('id');
    expect(response.body).to.have.property('name');
    expect(response.body).to.have.property('email');

    // Validar tipos
    expect(response.body.id).to.be.a('number');
    expect(response.body.name).to.be.a('string');
    expect(response.body.ativo).to.be.a('boolean');

    // Validar valores
    expect(response.body.id).to.equal(1);
    expect(response.body.name).to.equal('João Silva');
    expect(response.body.age).to.be.greaterThan(0);

    // Validar objetos aninhados
    expect(response.body.endereco.cidade).to.equal('São Paulo');

    // Validar arrays
    expect(response.body.telefones).to.have.length(2);
    expect(response.body.telefones).to.include('11987654321');
  });
```

Capítulo 10: Bash, Linux e Terminal - Ambiente do QA

10.1. Comandos Essenciais de Terminal

Navegação de Diretórios:

```
# Mostrar diretório atual
pwd

# Listar arquivos
ls
ls -la              # Com detalhes
ls -lh              # Com tamanho legível

# Mudar de diretório
cd /home/ubuntu
cd ..               # Diretório pai
cd ~                # Home directory
cd -                # Diretório anterior

# Criar diretório
mkdir projeto
mkdir -p projeto/src/main  # Criar hierarquia

# Remover diretório
rmdir projeto       # Vazio
rm -rf projeto      # Com conteúdo
```

Manipulação de Arquivos:

```
# Criar arquivo vazio
touch arquivo.txt

# Copiar arquivo
cp arquivo.txt cópia.txt
cp -r pasta/ cópia_pasta/  # Recursivo

# Mover/renomear
mv arquivo.txt novo_nome.txt
mv arquivo.txt /caminho/destino/

# Remover arquivo
rm arquivo.txt
rm -f arquivo.txt   # Forçar

# Visualizar conteúdo
cat arquivo.txt     # Mostrar tudo
head -n 10 arquivo.txt  # Primeiras 10 linhas
tail -n 10 arquivo.txt  # Últimas 10 linhas
less arquivo.txt    # Paginado (q para sair)
```

10.2. Análise de Logs

Comandos Essenciais:

```
# Monitorar log em tempo real
tail -f /var/log/application.log

# Buscar padrão em arquivo
grep "ERROR" /var/log/application.log

# Contar ocorrências
grep -c "ERROR" /var/log/application.log

# Mostrar contexto
grep -A 5 "ERROR" /var/log/application.log  # 5 linhas depois
grep -B 5 "ERROR" /var/log/application.log  # 5 linhas antes
grep -C 5 "ERROR" /var/log/application.log  # 5 linhas antes e depois

# Busca case-insensitive
grep -i "error" /var/log/application.log

# Expressão regular
grep "ERROR.*timeout" /var/log/application.log

# Múltiplos arquivos
grep "ERROR" /var/log/*.log

# Inverter busca (não contém)
grep -v "INFO" /var/log/application.log
```

Exemplo Prático:

```
# Encontrar todos os erros no último dia
tail -f /var/log/app.log | grep "ERROR"

# Contar erros por tipo
grep "ERROR" /var/log/app.log | grep -o "ERROR.*" | sort | uniq -c

# Encontrar erros de timeout
grep "ERROR.*timeout" /var/log/app.log | tail -20

# Salvar erros em arquivo
grep "ERROR" /var/log/app.log > erros.txt
```

10.3. Conectividade e Rede

Comandos Essenciais:

```
# Testar conectividade
ping google.com
ping -c 4 google.com    # 4 pacotes

# Verificar porta aberta
telnet example.com 80
nc -zv example.com 80   # netcat

# Fazer requisição HTTP
curl https://example.com
curl -X POST https://example.com/api/users \
  -H "Content-Type: application/json" \
  -d '{"name":"João"}'

# Salvar resposta em arquivo
curl https://example.com > resposta.html

# Mostrar headers
curl -i https://example.com

# Seguir redirecionamentos
curl -L https://example.com

# Informações de DNS
nslookup example.com
dig example.com

# Informações de rede
ifconfig            # IP local
netstat -an         # Conexões ativas
ss -an              # Alternativa moderna
```

Exemplo Prático:

```
# Testar API
curl -X GET https://api.example.com/users/1 \
  -H "Authorization: Bearer token123" \
  -H "Content-Type: application/json"

# Criar usuário via API
curl -X POST https://api.example.com/users \
  -H "Content-Type: application/json" \
  -d '{
    "name": "João Silva",
    "email": "joao@example.com"
  }'

# Verificar se servidor está respondendo
curl -s -o /dev/null -w "%{http_code}" https://example.com
```

10.4. Automação com Scripts Bash

Script Básico:

```bash
#!/bin/bash

# Variáveis
NOME="João"
IDADE=30

# Imprimir
echo "Olá, $NOME"
echo "Sua idade é: $IDADE"

# Condicionais
if [ $IDADE -ge 18 ]; then
  echo "Você é maior de idade"
else
  echo "Você é menor de idade"
fi

# Loops
for i in {1..5}; do
  echo "Número: $i"
done

# Funções
function saudacao() {
  echo "Olá, $1!"
}

saudacao "Maria"
```

Script para Testes:

```bash
#!/bin/bash

# Script para executar testes automaticamente

API_URL="https://api.example.com"
LOG_FILE="test_results.log"

# Função para testar endpoint
test_endpoint() {
  local method=$1
  local endpoint=$2
  local expected_code=$3

  echo "Testando: $method $endpoint"

  response=$(curl -s -o /dev/null -w "%{http_code}" -X $method
"$API_URL$endpoint")

  if [ "$response" -eq "$expected_code" ]; then
    echo "✓ PASSOU: $method $endpoint (HTTP $response)" >> $LOG_FILE
    return 0
  else
    echo "✗ FALHOU: $method $endpoint (esperado $expected_code, obteve
$response)" >> $LOG_FILE
    return 1
  fi
}

# Executar testes
test_endpoint "GET" "/users" 200
test_endpoint "GET" "/users/1" 200
test_endpoint "POST" "/users" 201
test_endpoint "GET" "/users/999" 404

# Resumo
echo ""
echo "Testes concluídos. Verifique $LOG_FILE"
```

10.5. Gerenciamento de Processos

Comandos Essenciais:

```
# Listar processos
ps aux                # Todos os processos
ps aux | grep java    # Processos Java

# Monitorar sistema em tempo real
top
htop                  # Alternativa mais amigável

# Matar processo
kill 1234             # Sinal TERM
kill -9 1234          # Sinal KILL (forçado)

# Executar em background
./script.sh &

# Executar em background com nohup (continua após logout)
nohup ./script.sh &

# Ver jobs em background
jobs

# Trazer para foreground
fg %1

# Pausar/retomar
Ctrl+Z                # Pausar
bg                    # Retomar em background
```

Exemplo Prático:

```
# Executar testes em background
nohup npm test > test_results.log 2>&1 &

# Monitorar progresso
tail -f test_results.log

# Verificar se processo ainda está rodando
ps aux | grep npm

# Matar processo se necessário
pkill -f "npm test"
```

PARTE IV: CARREIRA E DESENVOLVIMENTO PROFISSIONAL

Capítulo 12: Construindo um Portfólio Vencedor

12.1. Componentes Essenciais do Portfólio

Um portfólio profissional de QA deve demonstrar suas habilidades práticas e conhecimento teórico.

Componentes Principais:

1. **GitHub Repositories**
    - Projetos de automação com código limpo
    - README documentado
    - Histórico de commits bem estruturado
    - Testes funcionando
2. **Documentação**
    - Planos de Teste
    - Casos de Teste
    - Relatórios de Teste
    - Estudos de Caso
3. **Certificações**
    - ISTQB Foundation
    - Cypress Certified
    - Cursos reconhecidos
4. **LinkedIn Profile**
    - Foto profissional

- Resumo detalhado
- Recomendações
- Histórico de experiência

5. **Blog/Medium**
   - Artigos sobre QA
   - Tutoriais de ferramentas
   - Compartilhamento de conhecimento

12.2. Projetos de Automação

Projeto 1: Automação Web com Cypress

```
cypress-ecommerce-tests/
├── cypress/
│   ├── e2e/
│   │   ├── login.cy.js
│   │   ├── shopping-cart.cy.js
│   │   ├── checkout.cy.js
│   │   └── payment.cy.js
│   ├── support/
│   │   ├── commands.js
│   │   └── e2e.js
│   └── fixtures/
│       ├── users.json
│       └── products.json
├── cypress.config.js
├── README.md
└── package.json
```

Projeto 2: Automação com Robot Framework

```
robot-api-tests/
├── tests/
│   ├── user_api.robot
│   ├── product_api.robot
│   └── order_api.robot
├── resources/
│   ├── keywords.robot
│   └── variables.robot
├── results/
│   ├── report.html
│   ├── log.html
│   └── output.xml
├── README.md
└── robot.ini
```

Projeto 3: Testes de API com Python

```
python-api-tests/
├── tests/
│   ├── test_users.py
│   ├── test_products.py
│   └── test_orders.py
├── fixtures/
│   ├── users.json
│   └── products.json
├── requirements.txt
├── pytest.ini
├── README.md
└── conftest.py
```

12.3. Documentação e Estudos de Caso

Estudo de Caso: Teste de E-commerce

```
# Estudo de Caso: Teste de E-commerce

## Objetivo
Testar o fluxo completo de compra de um e-commerce, desde o login até o
pagamento.

## Escopo
- Login e autenticação
- Busca e filtro de produtos
- Adição ao carrinho
- Aplicação de cupom
- Checkout
- Pagamento

## Estratégia
- 70% Automação (Cypress)
- 30% Manual (Exploratório)

## Resultados
- 150 casos de teste criados
- 145 executados (96.7%)
- 12 bugs encontrados
- Taxa de sucesso: 92%

## Bugs Encontrados
1. Login não funciona em Safari (Crítico)
2. Carrinho não persiste após logout (Crítico)
3. Cupom não aplica desconto correto (Alta)

## Recomendações
1. Implementar testes de compatibilidade com navegadores
2. Aumentar cobertura de testes para módulo de pagamento
3. Automatizar testes de regressão
```

12.4. GitHub e Versionamento

Estrutura de Repositório:

```
projeto-qa/
├── .github/
│   └── workflows/
│       └── tests.yml
├── .gitignore
├── README.md
├── cypress/
├── tests/
├── docs/
├── package.json
└── LICENSE
```

README.md Profissional:

```
# Projeto de Testes Automatizados - E-commerce

## Descrição
Suite de testes automatizados para validar funcionalidades críticas de um e-commerce, incluindo login, carrinho de compras, checkout e pagamento.

## Tecnologias
- **Cypress** para testes E2E
- **JavaScript** para scripts
- **GitHub Actions** para CI/CD
- **Postman** para testes de API

## Instalação
```bash
npm install
npx cypress open
```
```

Executar Testes

```
# Todos os testes
npm test

# Teste específico
npm test -- --spec cypress/e2e/login.cy.js

# Com tags
npm test -- --env tags=smoke
```

Estrutura de Pastas

- `cypress/e2e/` - Testes E2E
- `cypress/support/` - Comandos customizados
- `cypress/fixtures/` - Dados de teste
- `docs/` - Documentação

Cobertura de Testes

- Login: 40 casos
- Carrinho: 60 casos
- Pagamento: 50 casos
- Total: 150 casos

Relatórios

Após executar os testes, abra `cypress/reports/index.html`

Contribuindo

1. Fork o projeto

2. Crie uma branch (`git checkout -b feature/nova-feature`)
3. Commit suas mudanças (`git commit -am 'Adiciona nova feature'`)
4. Push para a branch (`git push origin feature/nova-feature`)
5. Abra um Pull Request

Autor

João Silva - QA Engineer

Licença

MIT

```
---

## Capítulo 13: Currículo e LinkedIn Profissional

### 13.1. Estrutura de um Currículo Vencedor

**Formato Recomendado:**
```

JOÃO SILVA São Paulo, SP | (11) 98765-4321 | joao.silva@example.com LinkedIn: linkedin.com/in/joaosilva | GitHub: github.com/joaosilva

RESUMO PROFISSIONAL QA Engineer com 5 anos de experiência em automação de testes, desenvolvimento de estratégias de qualidade e liderança de equipes. Especialista em Cypress, Robot Framework e testes de API. Certificado ISTQB Foundation.

EXPERIÊNCIA PROFISSIONAL

QA Engineer Senior - Tech Company (2022 - Presente)

- Liderou equipe de 3 QAs em implementação de testes automatizados
- Aumentou cobertura de testes de 60% para 85%

- Reduziu tempo de regressão de 2 dias para 4 horas
- Implementou CI/CD pipeline com GitHub Actions

QA Engineer - E-commerce Company (2020 - 2022)

- Desenvolveu suite de testes com Cypress (150+ casos)
- Implementou testes de API com Postman
- Criou planos de teste e documentação
- Colaborou com desenvolvedores em pair testing

QA Analyst - Startup (2018 - 2020)

- Executou testes manuais e exploratórios
- Reportou e rastreou defeitos no Jira
- Participou de reuniões de requisitos

HABILIDADES TÉCNICAS

- Ferramentas: Cypress, Robot Framework, Postman, Jira, Git
- Linguagens: JavaScript, Python, Bash
- Metodologias: Ágil, Scrum, Kanban
- Testes: Funcional, Integração, API, Performance, Segurança

CERTIFICAÇÕES

- ISTQB Foundation (2022)
- Cypress Certified (2023)

EDUCAÇÃO

- Bacharelado em Ciência da Computação - Universidade X (2018)

PROJETOS DESTACADOS

- cypress-ecommerce-tests: Suite com 150+ testes E2E
- robot-api-tests: Automação de API com Robot Framework
- python-api-tests: Testes de API com Python e Pytest

” `

13.2. Palavras-Chave e Competências

Palavras-Chave Importantes:

- Cypress
- Robot Framework
- Postman
- Jira
- Automação de Testes
- Testes de API
- Testes E2E
- JavaScript
- Python
- Bash/Linux
- CI/CD
- GitHub Actions
- Testes de Regressão
- Testes Funcionais
- Testes de Performance
- ISTQB
- Agile/Scrum

13.3. Otimização do Perfil LinkedIn

Seções Importantes:

1. **Foto Profissional**
    - Fundo neutro
    - Roupas profissionais
    - Boa iluminação
2. **Headline**

- “QA Engineer | Automação com Cypress e Robot Framework | ISTQB Certified”

3. **Resumo**
    - Descrever experiência e habilidades
    - Incluir links para GitHub e portfólio
    - Mencionar certificações
4. **Experiência**
    - Detalhar responsabilidades
    - Incluir métricas e resultados
    - Mencionar tecnologias usadas
5. **Habilidades**
    - Adicionar todas as habilidades técnicas
    - Pedir endorsements
    - Priorizar as mais relevantes
6. **Recomendações**
    - Pedir recomendações de colegas e gerentes
    - Retribuir com recomendações

13.4. Networking e Oportunidades

Estratégias de Networking:

1. **Comunidades Online**
    - Grupos de QA no LinkedIn
    - Comunidades no Discord/Slack
    - Fóruns de discussão
2. **Eventos**
    - Conferências de QA

- Meetups locais
- Webinars e workshops

3. **Redes Sociais**
    - Compartilhar conhecimento no LinkedIn
    - Publicar artigos no Medium
    - Participar de discussões
4. **Contribuições Open Source**
    - Contribuir em projetos de teste
    - Reportar bugs
    - Melhorar documentação

Capítulo 14: Preparação para Entrevistas e Certificações

14.1. Perguntas Comuns em Entrevistas

Perguntas Técnicas:

1. **Qual é a diferença entre QA e QC?**
    - QA é proativo (prevenção), QC é reativo (detecção)
    - QA foca em processos, QC foca em produto
2. **Explique a Pirâmide de Testes**
    - Base: 70% testes de unidade
    - Meio: 20% testes de integração
    - Topo: 10% testes E2E
3. **Qual é a importância de testes automatizados?**
    - Execução rápida e repetível

- Feedback contínuo
- Redução de custo
- Aumento de cobertura

4. **Como você abordaria testar uma nova feature?**
    - Entender requisitos
    - Criar plano de teste
    - Identificar casos de teste
    - Executar testes
    - Reportar defeitos
5. **Qual é a diferença entre teste funcional e não funcional?**
    - Funcional: verifica se o sistema faz o que deveria
    - Não funcional: verifica como o sistema funciona (performance, segurança)

Perguntas Comportamentais:

1. **Descreva uma situação em que você encontrou um bug crítico**
    - Contexto: quando e onde
    - Ação: como você identificou e reportou
    - Resultado: como foi resolvido
2. **Como você lida com pressão e prazos apertados?**
    - Priorizar testes críticos
    - Comunicar riscos
    - Trabalhar em equipe
3. **Qual foi seu maior aprendizado em QA?**
    - Importância de documentação
    - Colaboração com desenvolvedores
    - Automação de testes

14.2. Certificação ISTQB

O que é ISTQB?

ISTQB (International Software Testing Qualifications Board) é uma certificação internacional reconhecida em QA.

Níveis:

1. **Foundation Level**
    - Conceitos básicos de teste
    - Tipos de teste
    - Planejamento e documentação
    - Duração: 1-2 meses de estudo
2. **Advanced Level**
    - Tópicos avançados
    - Especialidades (Automação, Performance, Security)
    - Duração: 2-3 meses de estudo
3. **Expert Level**
    - Nível mais alto
    - Experiência prática necessária

Tópicos do Foundation:

- Conceitos fundamentais de teste
- Testes durante o ciclo de vida
- Técnicas de teste estático
- Técnicas de design de teste
- Gerenciamento de teste
- Ferramentas de teste

Dicas de Estudo:

1. Usar livros oficiais ISTQB

2. Fazer simulados online
3. Estudar em grupos
4. Revisar conceitos regularmente
5. Praticar com exemplos reais

14.3. Outras Certificações Relevantes

Certificação	Fornecedor	Foco	Duração
Cypress Certified	Cypress	Automação Web	2-4 semanas
Robot Framework Certified	Robot Framework Foundation	Automação Versátil	2-4 semanas
AWS Certified QA	Amazon	QA em Cloud	4-6 semanas
Scrum Master	Scrum Alliance	Metodologia Ágil	2-3 semanas
Postman Certified	Postman	Testes de API	1-2 semanas

14.4. Preparação Técnica e Comportamental

Preparação Técnica:

1. **Revisar conceitos fundamentais**
    - Tipos de testes
    - Ciclo de vida de testes
    - Métricas de qualidade
2. **Praticar com ferramentas**
    - Escrever scripts Cypress
    - Criar testes Robot Framework
    - Testar APIs com Postman