# Projeto de Software - BIRD

BIRD

2025

# Sumário


**1 Requisitos do Sistema** **1**
- 1.1 Requisitos Funcionais . . . 1
- 1.2 Requisitos Não-Funcionais . . . 2
- 1.3 Protótipo . . . 2
  - 1.3.1 Detalhamento das Telas do Protótipo . . . 3

**2 Projeto de Software** **4**
- 2.1 Diagramas de Interação . . . 4
- 2.2 Diagramas de Classes . . . 4
- 2.3 PersistÊncia de Dados . . . 4
- 2.4 Mapeamento de Ferramentas . . . 4
  - 2.4.1 Linguagem de Programação . . . 4
  - 2.4.2 Frameworks e Bibliotecas . . . 5
  - 2.4.3 Ferramentas de Desenvolvimento . . . 5
  - 2.4.4 Containerização e Automação . . . 5

**3 Versionamento** **6**
- 3.1 Introdução . . . 6
- 3.2 Configuração do Ambiente . . . 6
- 3.3 Versionamento Semântico (SemVer) . . . 6
  - 3.3.1 Estrutura da Versão (X.Y.Z) . . . 6
- 3.4 O Modelo Git Flow . . . 6
  - 3.4.1 Branches Permanentes . . . 6
  - 3.4.2 Feature Branches (`feature/*`) . . . 6
  - 3.4.3 Release Branches (`release/*`) . . . 7
  - 3.4.4 Hotfix Branches (`hotfix/*`) . . . 7
- 3.5 Exemplo: . . . 7
- 3.6 Glossário de Comandos . . . 8
  - 3.6.1 Configuração Inicial . . . 8
  - 3.6.2 Operações Diárias . . . 8
  - 3.6.3 Boas Práticas de Commit . . . 8

**4 Deploy** **9**
- 4.1 Infraestrutura . . . 9
- 4.2 Processo de Deploy . . . 9

**5 Ambiente de Produção e Homologação** **10**
- 5.1 Ambiente de Homologação . . . 10
- 5.2 Ambiente de Produção . . . 10
- 5.3 Controle de Acesso e Segurança (Opcional) . . . 10

**6 QA** **11**
- 6.1 Tipos de Testes . . . 11
  - 6.1.1 Testes de Requisitos (Validação Inicial) . . . 11
  - 6.1.2 Testes Funcionais . . . 11
  - 6.1.3 Testes de Integração . . . 11
  - 6.1.4 Testes de Regressão . . . 11
  - 6.1.5 Testes de Aceitação (UAT) . . . 11
  - 6.1.6 Testes de Performance Básicos . . . 11

6.1.7 Smoke Test Pós-Deploy . . . 11
6.2 Processo de QA (Passo a Passo) . . . 12
6.2.1 Planejamento (Antes da Implementação) . . . 12
6.2.2 Design dos Testes . . . 12
6.2.3 Preparação . . . 12
6.2.4 Execução . . . 12
6.2.5 Reteste e Regressão . . . 12
6.2.6 Encerramento . . . 12
6.3 Critérios de Entrada e Saída . . . 13
6.3.1 Entrada para Início dos Testes da Sprint/Projeto . . . 13
6.3.2 Saída para Liberação em Produção . . . 13
6.4 Métricas de Qualidade . . . 13

**7 Padrões de Adoção de Versões no Mercado 14**
7.1 APIs de Telecomunicações (CAMARA / Open Gateway) . . . 14
7.2 Java / JDK . . . 14
7.3 Protocolos de Rede . . . 14
7.4 APIs de Pagamento e Financeiro . . . 14
7.5 Linguagens de Programação . . . 14
7.6 Bancos de Dados . . . 15
7.7 Infraestrutura / DevOps . . . 15
7.8 Recomendações . . . 15
7.9 Fontes Recomendadas . . . 15

**8 Processo de Integração 16**
8.1 Tipos e Padrões de Integração . . . 16
8.1.1 Tipos de Integração por Tecnologia . . . 16
8.1.2 Padrões de Integração Empresarial (EIP) . . . 16

**9 Ferramentas e Tecnologias de Integração 18**
9.1 Plataformas de Integração . . . 18
9.2 Integração por API (Application Programming Interface) . . . 18
9.2.1 Tipos de APIs . . . 18
9.2.2 Melhores Práticas de Design de APIs (RESTful) . . . 18
9.2.3 Ciclo de Vida da API . . . 19
9.3 Comparativo ESB vs iPaaS . . . 20

**10 Procedimento para Integração de Software 21**
10.1 Fases do Projeto . . . 21
10.2 Melhores Práticas . . . 22

**11 Passo a Passo para o Desenvolvimento de APIs RESTful 23**
11.1 Design da API (API-First) . . . 23
11.2 Implementação e Codificação . . . 23
11.3 Testes e Validação . . . 23
11.4 Governança e Evolução . . . 24

**12 Segurança de Software 25**
12.1 Fundamentos Teóricos . . . 25
12.1.1 Da Visão SWEBOK v4 . . . 25
12.1.2 OWASP Top 10 (2021) . . . 25

12.1.3 Práticas de Implementação (Codificação Segura) . . . 25
12.1.4 As 10 Melhores Práticas de Segurança do CERT/CC . . . 26
12.2 Principais Responsabilidades . . . 27
12.2.1 Na Fase de Definição e Design . . . 27
12.2.2 Durante Implementação . . . 27
12.2.3 Durante Testes e Entrega . . . 28

**13 Princípios Fundamentais (Clean Code) 30**
13.1 KISS (Keep It Simple, Stupid) . . . 30
13.1.1 O que é Simplicidade? . . . 30
13.1.2 Sinais de Alerta (Code Smells) . . . 30
13.1.3 Técnica Prática: Guard Clauses . . . 30
13.2 DRY (Don't Repeat Yourself) . . . 31
13.2.1 O Problema da Duplicação . . . 31
13.2.2 A "Regra de Três" (Rule of Three) . . . 32
13.2.3 Falsa Duplicação (Cuidado) . . . 32
13.2.4 Exemplo Prático: Centralização de Lógica . . . 32
13.3 SOLID Principles . . . 33
13.3.1 S - Single Responsibility Principle (SRP) . . . 33
13.3.2 O - Open/Closed Principle (OCP) . . . 33
13.3.3 L - Liskov Substitution Principle (LSP) . . . 34
13.3.4 I - Interface Segregation Principle (ISP) . . . 35
13.3.5 D - Dependency Inversion Principle (DIP) . . . 36

**14 Convenções de Estilo e Nomenclatura 37**
14.1 Idioma do Código: Inglês . . . 37
14.2 Sintaxe: Tabela de Referência por Linguagem . . . 37
14.3 Semântica de Nomenclatura (Regras Universais) . . . 37
14.3.1 Funções são Ações (Verbos) . . . 37
14.3.2 Classes são Entidades (Substantivos) . . . 37
14.3.3 Variáveis Booleanas (Perguntas) . . . 38
14.4 Segurança de Tipos (Type Safety) . . . 38
14.5 Exemplo Prático: Refatoração e Clareza . . . 38

**15 Ferramentas de Automação (Qualidade Contínua) 40**
15.1 Pilar 1: Formatter Automatizado . . . 40
15.2 Pilar 2: Analisador Estático (Linter) . . . 40
15.3 Pilar 3: Type Checker . . . 41

**16 Documentação e Legibilidade 42**
16.1 Regra de Ouro . . . 42
16.2 Padrões de Docstrings (API) . . . 42
16.2.1 Estrutura Obrigatória . . . 42
16.2.2 Exemplo Prático (Python - Google Style) . . . 42
16.3 Comentários Internos (O "Porquê") . . . 43
16.4 Tags de Manutenção (Anotações) . . . 44

**17 Tratamento de Erros e Observabilidade (Logs) 45**
17.1 "A Morte do `print`" . . . 45
17.2 Logs Estruturados (JSON) . . . 45
17.3 Níveis de Log (Padronização) . . . 45

17.4 Segurança no Log (Sanitização) . . . 46
17.5 Tratamento de Exceções (Exception Handling) . . . 46
17.5.1 Regra 1: Não engula exceções (Silent Failure) . . . 46
17.5.2 Regra 2: Envelopamento (Pattern de Camadas) . . . 46
17.5.3 Regra 3: Correlation ID (Rastreabilidade) . . . 47

**18 Segurança na Implementação (AppSec) 48**
18.1 Gerenciamento de Segredos (Credenciais) . . . 48
18.2 Blindagem contra Injeção (SQL Injection) . . . 48
18.3 Validação e Sanitização de Entrada . . . 48
18.4 Vazamento de Informação (Error Handling) . . . 49
18.5 5. Dependências Vulneráveis (Supply Chain) . . . 49

**19 Integração e Fluxo de Trabalho 50**
19.1 Fluxo de Entrada (Antes de Codificar) . . . 50
19.2 Fluxo de Apoio (Durante a Codificação) . . . 50
19.3 Fluxo de Saída (Entrega) . . . 50

**20 Checklist de Code Review (Pull Request) 52**
20.1 Padrões e Legibilidade . . . 52
20.2 Arquitetura e Design (SOLID/KISS) . . . 52
20.3 Segurança e Performance (Crítico) . . . 52
20.4 Operação e Observabilidade . . . 52
20.5 Testes . . . 53

**21 Anexo Técnico: Setup do Ambiente de Desenvolvimento 54**
21.1 Perfil A: Stack Python (Projetos de Backend / Scripts) . . . 54
21.1.1 Instalação . . . 54
21.1.2 Configuração (.pre-commit-config.yaml) . . . 54
21.2 Perfil B: Stack C# / .NET . . . 55
21.2.1 Instalação das Ferramentas . . . 55
21.2.2 Automação (Husky.Net ou Script) . . . 55
21.3 Perfil C: Stack Java . . . 56
21.3.1 Configuração no pom.xml (Maven) . . . 56
21.4 Integração com IDE (VS Code) . . . 56

**22 Usabilidade 57**
22.1 Visão Geral . . . 57
22.2 Fundamentos Teóricos . . . 57
22.3 Principais Responsabilidades . . . 57
22.3.1 Na Fase de Definição e Design . . . 57
22.3.2 Na Fase de Avaliação (Testing) . . . 57
22.4 Integração com o Time . . . 57
22.4.1 Com Engenharia de Requisitos . . . 58
22.4.2 Com Q&A / Testes . . . 58

**23 Leitura Recomendada 59**

# 1 Requisitos do Sistema

Texto Base (Instrução):

Esta seção deve definir o propósito deste documento ERS. O texto deve explicar por que o documento está sendo escrito e quem é o público-alvo (ex: desenvolvedores, testadores, cliente). Ele deve deixar claro que este documento servirá como um "contrato" ou acordo formal entre a equipe de desenvolvimento e o cliente sobre o que o software fará.

Exemplo:

"O objetivo desta ERS (Especificação de Requisitos de Software) consiste em documentar os requisitos do software a ser produzido. Este documento visa garantir que o cliente (usuário do software) e os desenvolvedores tenham um entendimento comum e inequívoco de todas as funcionalidades, capacidades e restrições do software, servindo como base para o design, desenvolvimento, testes e validação do produto final."

Texto Base (Instrução):

Descreva o escopo do produto de software. Responda às perguntas:

- Qual é o nome do sistema?
- Qual é o propósito principal do sistema? (Qual problema ele resolve?)
- Quais são os principais objetivos de negócio?
- (Opcional, mas recomendado) O que o sistema não fará? (Escopo Negativo). Isso é crucial para gerenciar as expectativas.

Exemplo:

"O sistema tem como propósito [**Propósito principal, ex: otimizar o processo de gestão de inventário**] para a [**Empresa/Cliente**]. Dentre os principais objetivos destacam-se um maior controle de [**Entidade, ex: estoque**] e a organização de [**Entidade, ex: pedidos**].1 O software permitirá ao usuário [**Função principal 1, ex: cadastrar produtos**], [**Função principal 2, ex: registrar entradas e saídas**] e [**Função principal 3, ex: gerar relatórios de inventário**].

**Escopo Negativo:** Este sistema não fará o processamento de folha de pagamento ou faturamento, que continuarão sendo realizados pelo sistema ERP existente."

## 1.1 Requisitos Funcionais

Neste item devem ser apresentados os requisitos funcionais que especificam ações que um sistema deve ser capaz de executar, ou seja, as funções do sistema. Os requisitos funcionais geralmente são melhor descritos em diagramas de caso de uso, juntamente com o detalhamento dos atores e de cada caso de uso. Cada ator do diagrama de caso de uso deve ser descrito de forma sucinta e cada caso de uso deve ser especificado. A seguir são apresentados itens básicos para a especificação dos casos de uso do diagrama.

- Nome do Caso de Uso
- Breve descrição
- Atores envolvidos
- Pré-condições
- Fluxo Principal de Eventos

- Pós-condições
- Fluxo Secundário de Eventos
- Observações

## 1.2 Requisitos Não-Funcionais

Neste item devem ser apresentados os requisitos não funcionais, que especificam restrições sobre os serviços ou funções providas pelo sistema. Eles abrangem aspectos como desempenho, segurança, usabilidade, confiabilidade e escalabilidade.

- Requisitos de sistema : Requisitos que especificam o comportamento do produto.Ex. portabilidade; tempo na execução; confiabilidade,mobilidade, etc.
- Requisitos da organização: Requisitos decorrentes de políticas e procedimentos corporativos. Ex. padrões, infra-estrutura,etc.
- Requisitos externos: Requisitos decorrentes de fatores externos ao sistema e ao processo de desenvolvimento. Ex. requisitos de interoperabilidade, legislação,localização geográfica etc.
- Requisitos de facilidade de uso. Ex.: usuários deverão operar o sistema após um determinado tempo de treinamento.
- Requisitos de eficiência. Ex.: o sistema deverá processar n requisições por um determinado tempo.
- Requisitos de confiabilidade. Ex.: o sistema deverá ter alta disponibilidade, por.exemplo, 99
- Requisitos de portabilidade. Ex.: o sistema deverá rodar em qualquer plataforma.
- Requisitos de entrega.Ex.: um relatório de acompanhamento deverá ser fornecido toda segunda-feira.
- Requisitos de implementação.: Ex.: o sistema deverá ser desenvolvido na linguagem Java.
- Requisitos de padrões.: Ex. uso de programação orientada a objeto sob a plataforma A.
- Requisitos de interoperabilidade.:Ex. o sistema deverá se comunicar com o SQL Server.
- Requisitos éticos. Ex.: o sistema não apresentará aos usuários quaisquer dados de cunho privativo.
- Requisitos legais. Ex.: o sistema deverá atender às normas legais, tais como padrões, leis, etc.
- Requisitos de Integração. Ex.: o sistema integra com outra aplicação.

## 1.3 Protótipo

O protótipo do sistema é a materialização inicial e das funcionalidades principais. Ele serve como um produto mínimo viável (MVP) visual, que é construído e aprimorado de forma incremental. O objetivo central é estabelecer um ciclo de feedback contínuo e atingir o consenso total com o usuário final sobre os requisitos do sistema.

### 1.3.1 Detalhamento das Telas do Protótipo

A implementação visual das telas pode ser realizada utilizando ferramentas de mockup dedicadas ou o ambiente de desenvolvimento final. É mandatório que cada tela seja acompanhada de uma especificação funcional detalhada, que deve contemplar os seguintes atributos-chave:

- Objetivo: Declaração concisa da finalidade da tela.
- Relação de Navegação: Indicação do módulo ou tela de origem e a lista de telas sucessoras que podem ser invocadas.
- Regras e Restrições:
  - Regras de Domínio: Especificações técnicas dos componentes de input, tais como formato de dados (ex: numérico, alfanumérico), restrições de cardinalidade (tamanho), e valores padrão (default).
  - Perfis de Acesso: Definição clara dos papéis de usuário (ex: Administrador, Operador, Leitor) autorizados a visualizar e interagir com a tela.

# 2 Projeto de Software

Texto Base (Instrução):

Esta seção descreve o design (o "como") do software, traduzindo os requisitos (o "o quê") em uma especificação de implementação. Esta seção é voltada para os desenvolvedores.

## 2.1 Diagramas de Interação

Texto Base (Instrução):

Insira aqui os Diagramas de Interação da UML (Diagramas de Sequência detalhados ou Diagramas de Comunicação). Diferente do DSS (3.1.4), estes diagramas mostram a colaboração interna entre as classes e objetos de software (ex: Controladores, Repositórios, Entidades) para realizar um Caso de Uso.

## 2.2 Diagramas de Classes

Texto Base (Instrução):

Insira aqui o Diagrama de Classes de Projeto (UML). Este diagrama é mais detalhado que o Modelo Conceitual (3.2). Ele deve incluir as classes de software (ex: classes de Interface, Controle e Entidade), seus atributos (com tipos de dados e visibilidade -/+) e seus métodos (com parâmetros e visibilidade).

## 2.3 PersistÊncia de Dados

O banco de dados utilizado no projeto é o [**nome do SGBD – ex.: PostgreSQL, MySQL, MongoDB**], classificado como [**relacional — não relacional**]. Sua escolha se deve a [**justificativa sucinta, como desempenho, escalabilidade, compatibilidade tecnológica ou suporte a transações**].

Para bancos de dados relacionais, a persistência dos dados é realizada por meio de mapeamento objeto-relacional (ORM), no qual as classes do Diagrama de Classes são mapeadas para tabelas, e seus atributos para colunas, respeitando chaves primárias, estrangeiras e relacionamentos.

Para bancos de dados não relacionais, a persistência ocorre por meio de coleções e documentos, onde as classes são representadas por documentos (por exemplo, JSON), estruturados conforme a modelagem definida no Diagrama de Classes.

## 2.4 Mapeamento de Ferramentas

Texto Base (Instrução):

Descreva as ferramentas utilizadas no desenvolvimento, versionamento, testes e deploy do sistema, justificando brevemente a escolha de cada uma em relação aos requisitos do projeto, como escalabilidade, manutenção, produtividade e integração.

### 2.4.1 Linguagem de Programação

A linguagem [**LINGUAGEM**] foi adotada neste projeto por apresentar melhor aderência às necessidades da aplicação, especialmente no que se refere a [**tipo de aplicação, ex: aplicações web, APIs, processamento de dados, sistemas distribuídos**]. Entre os principais motivos para sua escolha destacam-se:

- vantagem 1: ex. produtividade, tipagem, performance, comunidade

- vantagem 2
- vantagem 3

### 2.4.2 Frameworks e Bibliotecas

O framework [**FRAMEWORK PRINCIPAL**] foi utilizado para estruturar a aplicação, pois oferece [**motivação: ex. padrão arquitetural, segurança, rapidez no desenvolvimento, suporte a ORM, middlewares**]. Outras bibliotecas relevantes incluem:

- Biblioteca 1: utilizada para [**finalidade**]
- Biblioteca 2: utilizada para [**finalidade**]

### 2.4.3 Ferramentas de Desenvolvimento

Para o desenvolvimento e apoio ao projeto, foram utilizadas as seguintes ferramentas:

- IDE/Editor – [**ex: VS Code**]: para edição e organização do código-fonte
- Ferramenta de modelagem – [**ex: draw.io, StarUML**]: para criação de diagramas UML
- Ferramenta de testes de API – [**ex: Postman, Insomnia**]: para validação das rotas e serviços
- Ferramenta de mock/teste – [**ex: WireMock**]: para simulação de serviços externos (quando aplicável)

### 2.4.4 Containerização e Automação

A ferramenta [**Docker / Outra**] foi utilizada para padronizar o ambiente de desenvolvimento e facilitar o deploy da aplicação, garantindo que ela funcione de forma consistente em diferentes ambientes.

# 3 Versionamento

## 3.1 Introdução

Nesse guia padronizaremos a forma como o time desenvolve, integra e entrega software. Usaremos o **Git Flow** para organizar as ramificações e o **Versionamento Semântico** para comunicar o impacto das mudanças.

## 3.2 Configuração do Ambiente

Teremos uma organização do Bird no github, com os pesquisadores adicionados como colaboradores, onde serão criados repositórios específicos para cada projeto.

## 3.3 Versionamento Semântico (SemVer)

Vamos usar o padrão **Major.Minor.Patch** (ex: `v1.2.0`). A alteração dos números depende do impacto das mudanças feitas no código:[1]

### 3.3.1 Estrutura da Versão (X.Y.Z)

- **MAJOR (X.0.0) - Quebra de Compatibilidade**: Incrementada quando há mudanças drásticas. É geralmente quando acaba a retrocompatibilidade.
- **MINOR (0.Y.0) - Nova Funcionalidade**: Incrementada quando tem novas funcionalidades adicionadas, mas que ainda são compatíveis com versões anteriores. Exemplo: Adicionar um novo botão na interface.
- **PATCH (0.0.Z) - Correção de Bug**: Incrementada para correções de falhas simples que não alteram funcionalidades. Exemplo: Corrigir um erro de digitação, ajustar uma cor CSS ou corrigir um cálculo.

## 3.4 O Modelo Git Flow

A estrutura do repositório vai ser composta pelas seguintes branches principais:[2]

### 3.4.1 Branches Permanentes

- `main`: Representa a **Produção**. **Não recebe commit direto**. Só recebe código via Merge de `release` ou `hotfix`. Cada commit deve ter uma **Tag** de versão.
- `develop`: Representa o **Desenvolvimento Contínuo**, é a branch de integração. Ela **contém as funcionalidades completas para a próxima versão**.

### 3.4.2 Feature Branches (`feature/*`)

- **Objetivo:** Desenvolver uma nova funcionalidade.
- **Nasce em:** `develop`.
- **Morre em:** `develop`.

[1] Se quiserem ler mais sobre versionamento semântico podem acessar a especificação.

[2] Se quiserem ler mais sobre o modelo do Git Flow, podem encontrar o artigo original do Vincent Driessen (2010), "A successful Git branching model" ou esse em português.

- **Fluxo:** O dev cria a branch, trabalha nela e abre um PR para a `develop`. Após o merge, a branch local pode ser apagada.

#### 3.4.3 Release Branches (`release/*`)

- **Objetivo:** Congela o código para testes de QA e preparação final (documentação, versão), aqui acontece o `Staging`. **Ela é exclusiva para isso, nenhuma feature nova entra aqui.**
- **Nasce em:** `develop` (quando o time decide que vai lançar uma versão).
- **Morre em:** Dois lugares. Ao finalizar a release, ela é mergeada na:
  1. `main`: Para atualizar a produção.
  2. `develop`: Para garantir que correções de bugs feitas durante a fase de release voltem para o desenvolvimento.
- O nome da branch deve seguir o SemVer (ex: `release/v1.2.0`).

#### 3.4.4 Hotfix Branches (`hotfix/*`)

- **Objetivo:** Resolver bugs críticos em produção.
- **Nasce em:** `main`.
- **Morre em:** Assim como a release, ela é mergeada na:
  1. `main`: Para corrigir o erro imediatamente (gera nova Tag Patch).
  2. `develop`: Para garantir que o erro não volte a aparecer na próxima release.
- Geralmente incrementa o **Patch** (ex: `hotfix/v1.2.1`).

### 3.5 Exemplo:

Por exemplo, imagine que estamos na **versão v1.1.0**.

1. **Início do Trabalho:** O dev quer criar um "Modo Escuro". Ele cria a branch `feature/dark-mode` a partir da `develop`.
2. **Integração:** Ele termina, abre PR e mergeia na `develop`. Outros devs também mergeiam suas features.
3. **Corte da Release:** O time decide lançar. É criada a branch `release/v1.2.0` a partir da `develop`.
4. **Fase de QA:** O QA testa a `release/v1.2.0`. Encontra um bug no CSS.
5. **Correção na Release:** O dev corrige o bug na branch `release/v1.2.0` (commit de fix).
6. **Lançamento:** A release é aprovada.
   - Mergeiam a `release/v1.2.0` na `main` → Cria-se a Tag `v1.2.0`.
   - MErgeiam a `release/v1.2.0` na `develop` (o bug é corrigido na develop também).
7. **Hotfix:** No dia seguinte, descobrem que o login parou de funcionar na produção (`main`).

8. **Correção do Hotfix:**
    - Criam `hotfix/v1.2.1` a partir da `main`.
    - Corrigem o erro.
    - Mergeiam na `main` (Tag `v1.2.1`) e na `develop`.

## 3.6 Glossário de Comandos

Aqui temos um glossário dos comandos que mais usaremos no git, caso alguém não se lembre ou não esteja acostumado. Pode também rodar o comando `git --help` direto no terminal, acessar a documentção do Git ou o glossário da Atlassian.

### 3.6.1 Configuração Inicial

```
git clone https://github.com/Brain-BIRDs/seu-projeto.git
cd seu-projeto
```

### 3.6.2 Operações Diárias

**Fluxo Básico para Feature:**

```
# 1. Garanta que esta atualizado
git checkout develop
git pull origin develop

# 2. Crie sua branch
git checkout -b feature/minha-tarefa

# ... Trabalho sendo feito ...

# 3. Salve e envie
git add .
git commit -m "feat: Adiciona nova tela"
git push origin feature/minha-tarefa
```

### 3.6.3 Boas Práticas de Commit

Para que seja fácil entender e encontrar o que desejamos, é bom seguir padrões de commit:

- **feat:** Nova funcionalidade.
- **fix:** Correção de bug.
- **docs:** Alteração em documentação.
- **style:** Formatação (ponto e vírgula, espaços).
- **refactor:** Melhoria de código sem mudar funcionalidade.

# 4 Deploy

Texto Base (Instrução):

Descreva como o sistema é empacotado e disponibilizado para execução, incluindo infraestrutura, serviços utilizados e forma de publicação. O ambiente de deploy do sistema é composto por [**tipo de infraestrutura – ex: servidor local, VPS, cloud**], onde a aplicação é distribuída utilizando [**tecnologia – ex: Docker, pipeline CI/CD, scripts manuais**].

## 4.1 Infraestrutura

A aplicação é hospedada em [**PROVEDOR ou AMBIENTE – ex: AWS, Azure, VPS própria**], utilizando:

- Sistema Operacional: [**ex: Linux Ubuntu XX**]
- Servidor de aplicação: [**ex: Gunicorn, Node.js, Tomcat**]
- Servidor web (quando aplicável): [**ex: Nginx, Apache**]

## 4.2 Processo de Deploy

O processo de deploy ocorre da seguinte forma:

1. ex: Build da aplicação
2. Criação de imagens Docker
3. Publicação no servidor
4. Inicialização dos containers/serviços

Esse processo pode ser manual / automatizado, dependendo da configuração adotada.

# 5 Ambiente de Produção e Homologação

Texto Base (Instrução):
Descreva os ambientes de homologação e produção, destacando suas diferenças e finalidades.

## 5.1 Ambiente de Homologação

O ambiente de homologação é destinado a testes e validações, sendo utilizado para:

- Testes funcionais
- Validação de regras de negócio
- Avaliação de integrações

Ele possui configuração similar ao ambiente de produção, porém com:

- Base de dados de teste
- Acesso restrito
- Logs em nível mais detalhado

## 5.2 Ambiente de Produção

O ambiente de produção é o ambiente final do sistema, destinado aos usuários finais. Neste ambiente:

- São utilizados dados reais
- acesso é controlado por [mecanismos de segurança]
- O sistema opera com foco em desempenho, estabilidade e disponibilidade

## 5.3 Controle de Acesso e Segurança (Opcional)

São adotadas as seguintes medidas de segurança:

- Autenticação / Autorização
- Uso de HTTPS
- Controle de permissões
- Backups periódicos

# 6 QA

## 6.1 Tipos de Testes

### 6.1.1 Testes de Requisitos (Validação Inicial)

- Revisar histórias e requisitos com foco em:
  - clareza;
  - critérios de aceitação;
  - cenários de exceção.
- Entrega: requisitos "prontos para teste", ou seja, compreensíveis e testáveis.

### 6.1.2 Testes Funcionais

- Verificar se cada funcionalidade faz exatamente o que foi especificado.
- Basear os testes nos critérios de aceitação e cenários de negócio.

### 6.1.3 Testes de Integração

- Verificar a comunicação entre módulos e sistemas(API's, webhooks, serviços externos).
- Validar contratos de integração, formatos de mensagens e tratamento de erros.

### 6.1.4 Testes de Regressão

- Executar um conjunto de casos principais a cada nova *release*.
- Garantir que funcionalidades já existentes não foram quebradas por alterações recentes.

### 6.1.5 Testes de Aceitação (UAT)

- envolver representantes de negócio ou usuários internos, quando fizer sentido.
- Validar se a solução atende às expectativas reais de uso antes de ir para produção.

### 6.1.6 Testes de Performance Básicos

- Avaliar tempo de resposta em cenários típicos.
- Identificar travamentos ou lentidões evidentes nas principais jornadas.

### 6.1.7 Smoke Test Pós-Deploy

- Após o *deploy*, checar se o sistema "respira":
  - acessos ao sistema;
  - login;
  - fluxo principal de negócios;
  - funcionalidades críticas.

## 6.2 Processo de QA (Passo a Passo)

### 6.2.1 Planejamento (Antes da Implementação)

- Participação do QA nas reuniões de requisitos e projeto.
- Identificação de riscos, dependências e funcionalidades críticas.
- Início da lista de cenários e casos de teste.

### 6.2.2 Design dos Testes

- Criação ou atualização dos casos de teste.
- Definição da massa de dados de teste.
- Identificação de necessidades específicas de ambiente.

### 6.2.3 Preparação

- Garantir que o ambiente de testes está pronto:
  - versão correta implantada;
  - acessos liberados;
  - massa de dados criada ou carregada.
- Validar se os artefatos necessários (requisitos, fluxogramas, protótipos) estão disponíveis.

### 6.2.4 Execução

- Executar os cases de teste planejados.
- Registrar o resultado de cada caso como: Aprovado, Reprovado ou Bloqueado.
- Registrar bugs com, no mínimo:
  - passos para reproduzir;
  - ambiente em que ocorreu;
  - evidências (prints, vídeos, logs);
  - severidade e prioridade sugeridas.

### 6.2.5 Reteste e Regressão

- Após a correção do bug, o QA deve retestar o cenário.
- Em defeitos críticos, executar uma regressão rápida nos fluxos impactados.

### 6.2.6 Encerramento

- Verificar se os critérios de saída foram atendidos.
- Registrar um resumo de testes da release, incluindo:
  - quantidade de casos executados;
  - taxa de aprovação;
  - número de bugs por severidade;
  - principais riscos conhecidos.

## 6.3 Critérios de Entrada e Saída

### 6.3.1 Entrada para Início dos Testes da Sprint/Projeto

Os testes só devem iniciar quando:

- requisitos estiverem completos, aprovados e disponíveis;
- protótipos, fluxogramas ou demais artefatos (quando existirem) estiverem acessíveis;
- build adequada estiver implantada no ambiente de testes;
- QA tiver usuários, acessos e massa de teste mínima disponível.

### 6.3.2 Saída para Liberação em Produção

A release só deve ser liberada para produção quando:

- 100% dos casos de teste críticos e altos estiverem aprovados;
- não houver bugs críticos abertos;
- bugs médios e baixos estiverem conhecidos, documentados e aceitos pelo time de negócio;
- houver registro do resumo de testes e dos riscos remanescentes.

## 6.4 Métricas de Qualidade

As métricas abaixo serão utilizadas para acompanhar a qualidade dos produtos e a efetividade do processo de testes:

- Percentual de casos de teste executados por sprint: razão entre casos executados e casos planejados.
- Percentual de aprovação dos casos: proporção de casos aprovados sobre o total executado.
- Número de bugs por severidade: contagem de defeitos categorizados em Crítico, Alto, Médio e Baixo.
- Bugs encontrados em produção: quantidade de defeitos que escaparam do QA e foram detectados após o deploy.

Essas métricas serão analisadas periodicamente pela área de KPI em conjunto com o QA e demais áreas envolvidas, servindo como base para ações de melhoria contínua no processo da Fábrica de Software.

# 7 Padrões de Adoção de Versões no Mercado

**Objetivo:** Identificar a diferença entre as versões mais recentes das tecnologias e aquelas efetivamente adotadas pelo mercado, auxiliando fábricas de software a alinhar decisões técnicas com a realidade do mercado.

## 7.1 APIs de Telecomunicações (CAMARA / Open Gateway)

| API | Recente | Mercado | Observação |
|---|---|---|---|
| SIM Swap | v2.1.0 | v0.4.0 | Versão alpha ainda é padrão global |
| Number Verification | v1.0.0+ | v0.3.0 | Muitas operadoras usam versão inicial |
| Device Location | v1.0.0+ | v0.2 / v0.3 | Adoção lenta por privacidade |
| OTP Validation | v0.2.0+ | v0.1.0 | Primeira versão predominante |

## 7.2 Java / JDK

| Versão Recente | Mercado | Observação |
|---|---|---|
| JDK 21+ (LTS) | JDK 8 / 11 | Java 8 ainda muito utilizado; JDK 11 em ampla adoção |

## 7.3 Protocolos de Rede

| Tecnologia | Recente | Mercado | Observação |
|---|---|---|---|
| HTTP | HTTP/3 | HTTP/1.1 / 2 | Baixa adoção do HTTP/3 |
| TLS | TLS 1.3 | TLS 1.2 | Compatibilidade mantém 1.2 |
| IPv4 vs IPv6 | IPv6 | IPv4 | IPv4 ainda domina o tráfego |
| DNS | DNS over HTTPS | DNS tradicional | DoH ainda pouco adotado |

## 7.4 APIs de Pagamento e Financeiro

| API | Recente | Mercado | Observação |
|---|---|---|---|
| PIX | DICT v2.0+ | v1.x | Instituições em estabilização |
| Open Banking BR | Fase 4 | Fase 2/3 | Adoção parcial |
| PCI DSS | 4.0 | 3.2.1 | Migração até 2025 |
| 3D Secure | 2.3+ | 2.1 / 2.2 | Gateways defasados |

## 7.5 Linguagens de Programação

| Linguagem | Recente | Mercado | Observação |
|---|---|---|---|
| Python | 3.12+ | 3.8 / 3.9 / 3.10 | Produção concentra-se em 3.10+ |
| Node.js | v22+ | v18 / v20 LTS | Priorizar LTS |
| PHP | 8.3+ | 7.4 / 8.0 / 8.1 | Legado ainda relevante |
| .NET | .NET 8+ | .NET 6 / Framework 4.8 | Alto uso do Framework |

## 7.6 Bancos de Dados

| Banco | Recente | Mercado | Observação |
|---|---|---|---|
| PostgreSQL | 16+ | 12–14 | Versões mais usadas |
| MySQL | 8.x | 5.7 / 8.0 | 5.7 ainda comum |
| MongoDB | 7.x | 4.4–6.0 | Atualização gradual |
| Redis | 7.x | 6.x | Padrão em produção |

## 7.7 Infraestrutura / DevOps

| Tecnologia | Recente | Mercado | Observação |
|---|---|---|---|
| Kubernetes | 1.30+ | 1.26–1.28 | Sempre algumas versões atrás |
| Docker | 25+ | 20–24 | Diferença entre Desktop e Engine |
| Terraform | 1.7+ | 1.0–1.5 | Estabilidade priorizada |

## 7.8 Recomendações

- Pesquisar versões realmente suportadas pelo mercado.
- Priorizar versões LTS.
- Manter compatibilidade com versões anteriores.
- Documentar justificativas técnicas.

## 7.9 Fontes Recomendadas

- CAMARA Project
- GSMA Open Gateway
- JetBrains Developer Survey
- Stack Overflow Developer Survey
- Banco Central do Brasil (PIX)

# 8 Processo de Integração

A integração de software é um processo fundamental no contexto de desenvolvimento de software, onde a produção contínua de novas aplicações e a manutenção de sistemas legados exigem que componentes díspares trabalhem de forma coesa e eficiente [1]. A integração de sistemas (System Integration) é a disciplina que visa conectar diferentes subsistemas ou aplicações de software, permitindo que eles troquem dados e coordenem funcionalidades, transformando-os em um ecossistema unificado [2].

A integração é crucial para:

- **Reutilização de Componentes**: Conectar novos módulos a serviços existentes, acelerando o desenvolvimento.
- **Consistência de Dados**: Garantir que as informações sejam sincronizadas e precisas em todos os sistemas.
- **Automação de Processos**: Criar fluxos de trabalho de ponta a ponta que atravessam múltiplas aplicações.

## 8.1 Tipos e Padrões de Integração

A integração pode ser classificada de diversas maneiras, dependendo da tecnologia e do padrão arquitetural adotado.

### 8.1.1 Tipos de Integração por Tecnologia

| **Tipo de Integração** | **Descrição** | **Exemplo de Uso** |
|---|---|---|
| Integração de Dados | Foco na sincronização ou transferência de dados entre bancos de dados ou arquivos. | ETL (Extract, Transform, Load) para Data Warehousing. |
| Integração de Aplicações (A2A) | Conexão de funcionalidades de sistemas de software distintos. | Uso de APIs para que um sistema de CRM envie dados de clientes para um sistema de Faturamento. |
| Integração de Processos | Orquestração de atividades que envolvem múltiplos sistemas. | Automação de um processo de pedido que passa por E-commerce, Estoque e Logística. |

### 8.1.2 Padrões de Integração Empresarial (EIP)

Os Padrões de Integração Empresarial, popularizados por Hohpe e Woolf, fornecem soluções comprovadas para problemas comuns de integração [3]. Eles se baseiam principalmente em mensageria.

| **Padrão de Comunicação** | **Descrição** | **Vantagens** |
|---|---|---|
| Mensageria (Messaging) | Sistemas se comunicam trocando mensagens assíncronas através de um canal intermediário (Message Broker). | Desacoplamento (Loose Coupling), escalabilidade, resiliência. |
| Invocação Remota de Procedimento (RPC - Remote Procedure Call) | Sistemas se comunicam diretamente, com o solicitante esperando uma resposta síncrona. | Simplicidade, familiaridade com chamadas de função. |

# 9 Ferramentas e Tecnologias de Integração

A escolha da ferramenta de integração é um fator crítico que impacta a arquitetura, a escalabilidade e a manutenibilidade do ecossistema de software.

## 9.1 Plataformas de Integração

| Ferramenta | Conceito | Uso Típico |
|---|---|---|
| ESB (Enterprise Service Bus) | Arquitetura centralizada que atua como um barramento de comunicação entre aplicações locais (on-premise). Oferece roteamento, transformação e orquestração. | Integração de sistemas legados e complexos dentro de um datacenter. |
| iPaaS (Integration Platform as a Service) | Plataforma baseada em nuvem que fornece ferramentas de autoatendimento para desenvolver, executar e governar fluxos de integração. | Integração de aplicações SaaS (Software as a Service), ambientes híbridos (nuvem e local) e projetos com foco em agilidade. |
| API Gateway | Ponto de entrada único para todas as APIs. Lida com segurança (autenticação/autorização), limitação de taxa (rate limiting), roteamento e monitoramento. . | Exposição controlada e segura de serviços de backend para clientes externos ou internos. |
| Message Broker | Software intermediário que gerencia a troca de mensagens entre sistemas de forma assíncrona. Exemplos: Apache Kafka, RabbitMQ. | Implementação de arquiteturas orientadas a eventos (EDA) e garantia de entrega de mensagens em ambientes distribuídos. |

## 9.2 Integração por API (Application Programming Interface)

A integração por API é o método mais prevalente e flexível em arquiteturas modernas, como microsserviços e sistemas distribuídos. Uma API atua como um contrato bem definido que permite que dois sistemas se comuniquem sem conhecer os detalhes internos um do outro.

### 9.2.1 Tipos de APIs

### 9.2.2 Melhores Práticas de Design de APIs (RESTful)

O design de APIs deve ser tratado como um produto, focado na experiência do desenvolvedor (Developer Experience - DX).

- Recursos (Resources): Use substantivos (ex: `/clientes, /pedidos`) em vez de verbos nos endpoints. Os verbos HTTP definem a ação (GET para buscar, POST para criar, etc.).

| Tipo | Padrão de Comunicação | Características | Uso Típico |
|---|---|---|---|
| REST (Representational State Transfer) | Síncrona (HTTP) | Leve, sem estado (stateless), utiliza verbos HTTP (GET, POST, PUT, DELETE) e recursos (resources). Formato de dados mais comum é JSON. | Integração web, APIs públicas, microsserviços. |
| SOAP (Simple Object Access Protocol) | Síncrona (XML/HTTP) | Baseado em XML, fortemente tipado, utiliza WSDL (Web Services Description Language) para contrato. Mais complexo, mas com alta segurança e transacionalidade. | Integração com sistemas legados, ambientes corporativos (Enterprise). |
| GraphQL | Síncrona (HTTP) | Linguagem de consulta para APIs. Permite que o cliente solicite exatamente os dados de que precisa, evitando over-fetching ou under-fetching. | Aplicações móveis e web com requisitos de dados complexos e variáveis. |
| gRPC (Google Remote Procedure Call) | Síncrona (HTTP/2) | Baseado em RPC, utiliza Protocol Buffers para serialização. Focado em alta performance, baixo consumo de banda e comunicação entre microsserviços. | Comunicação interna de alto desempenho entre serviços. |

- Versionamento: Inclua a versão da API na URL (ex: `/api/v1/clientes`) ou no cabeçalho (Header) para permitir a evolução sem quebrar clientes existentes.
- Códigos de Status HTTP: Utilize os códigos de status padrão (200 OK, 201 Created, 400 Bad Request, 404 Not Found, 500 Internal Server Error) de forma consistente para indicar o resultado da operação.
- Paginação e Filtragem: Implemente mecanismos de paginação (ex: `?page=1&size=20`) e filtragem para otimizar o desempenho e o uso de recursos.
- Documentação: Mantenha a documentação da API (ex: usando OpenAPI/Swagger) sempre atualizada, detalhando endpoints, parâmetros, exemplos de requisição/resposta e códigos de erro.

### 9.2.3 Ciclo de Vida da API

A integração por API segue um ciclo de vida que deve ser gerenciado ativamente:

1. Planejamento: Definir o propósito, o público-alvo e os requisitos de negócio.
2. Design: Modelar o contrato da API (especificação) antes da implementação.
3. Desenvolvimento: Implementar a lógica de negócio e os adaptadores de dados.
4. Teste: Realizar testes unitários, de integração e de carga.
5. Publicação: Disponibilizar a API através de um API Gateway para gerenciamento e segurança.

6. Monitoramento: Acompanhar o desempenho, a latência e os erros em produção.

7. Descontinuação (Retirement): Gerenciar a transição para novas versões e a eventual desativação de versões antigas.

## 9.3 Comparativo ESB vs iPaaS

A tendência moderna em fábricas de software é a migração de arquiteturas ESB tradicionais para soluções iPaaS, especialmente em ambientes de nuvem e híbridos [4].

| **Característica** | **ESB (Enterprise Service Bus)** | **iPaaS (Integration Platform as a Service)** |
|---|---|---|
| Modelo de Implantação | Geralmente local (on-premise) ou em IaaS. | Baseado em nuvem (SaaS). |
| Foco | Integração de sistemas legados e internos. | Integração de SaaS, nuvem e ambientes híbridos. |
| Governança | Centralizada e tipicamente gerenciada por uma equipe de integração dedicada. | Distribuída, permitindo que equipes de desenvolvimento e de negócios criem suas próprias integrações (Citizen Integrators). |
| Escalabilidade | Limitada pela infraestrutura local. | Altamente escalável e elástica, gerenciada pelo provedor de nuvem. |

# 10 Procedimento para Integração de Software

Um projeto de integração bem-sucedido em uma fábrica de software segue um ciclo de vida estruturado, garantindo que os requisitos de negócio e técnicos sejam atendidos com qualidade e segurança.

## 10.1 Fases do Projeto

O procedimento pode ser dividido nas seguintes fases:

**Fase 1: Planejamento e Análise de Requisitos**

- Definição de Objetivos: Clarificar o porquê da integração (ex: reduzir redundância de dados, automatizar processo X).
- Mapeamento de Sistemas: Identificar os sistemas de origem (Source) e destino (Target), suas tecnologias e capacidades de comunicação (APIs, bancos de dados, arquivos).
- Análise de Dados: Mapear os campos de dados que serão trocados, definindo a transformação (Transformation) e o formato (Schema) necessários.
- Seleção da Tecnologia: Escolher o padrão de integração (síncrono/assíncrono) e a ferramenta (iPaaS, API Gateway, etc.) mais adequados.

**Fase 2: Design e Arquitetura**

- Desenho do Fluxo: Criar diagramas (ex: BPMN, UML) que detalham o fluxo de mensagens, o roteamento e a lógica de orquestração.
- Definição de Contratos: Formalizar os contratos de interface (ex: especificações OpenAPI/Swagger para APIs, schemas XSD/JSON para mensagens). No caso de APIs, a especificação OpenAPI é a prática recomendada.
- Segurança: Projetar mecanismos de autenticação (ex: OAuth 2.0, JWT) e autorização, garantindo a criptografia dos dados em trânsito (TLS/SSL).
- Tratamento de Erros: Definir estratégias de retry, logging, e mecanismos de compensação (rollback) para falhas.

**Fase 3: Implementação e Desenvolvimento**

- Desenvolvimento dos Adaptadores: Criar os componentes que se comunicam com os sistemas de origem e destino.
- Implementação da Lógica: Codificar a lógica de transformação, roteamento e orquestração na plataforma de integração escolhida.
- Versionamento: Utilizar sistemas de controle de versão (ex: Git) para gerenciar o código da integração.

**Fase 4: Testes e Qualidade**

- Testes Unitários: Testar individualmente os componentes de transformação e adaptadores.
- Testes de Integração: Validar o fluxo completo entre os sistemas em um ambiente de homologação (Staging).

- Testes de Performance e Carga: Simular o volume de transações esperado para garantir que a solução suporte a demanda.
- Monitoramento: Configurar ferramentas de monitoramento e alertas para rastrear o desempenho e as falhas da integração em tempo real.

**Fase 5: Implantação e Operação**

- Implantação (Deployment): Mover a solução para o ambiente de produção, preferencialmente utilizando práticas de CI/CD (Continuous Integration/Continuous Delivery).
- Go-Live e Validação: Acompanhar o desempenho inicial e validar a consistência dos dados.
- Manutenção e Evolução: A integração deve ser tratada como um produto de software, sujeita a manutenção contínua, refatoração e evolução conforme os sistemas conectados mudam.

## 10.2 Melhores Práticas

- Desacoplamento (Loose Coupling): Os sistemas devem ter o mínimo de dependência possível. O uso de Message Brokers e APIs bem definidas promove o desacoplamento.
- Padrões de Integração: Sempre que possível, utilize os Enterprise Integration Patterns para resolver problemas comuns de forma padronizada e robusta [3].
- Observabilidade: Implementar logging detalhado, tracing distribuído e métricas para garantir a visibilidade completa do fluxo de dados.
- Idempotência: Garantir que a repetição de uma mensagem ou chamada de API não cause efeitos colaterais indesejados no sistema de destino.
- Governança de APIs: Tratar as APIs como produtos, com documentação clara, versionamento e um ciclo de vida bem definido. A utilização de um API Gateway é essencial para aplicar políticas de segurança, limitação de taxa e monitoramento.

# 11 Passo a Passo para o Desenvolvimento de APIs RESTful

O desenvolvimento de APIs RESTful em uma Fábrica de Software deve seguir uma abordagem estruturada, preferencialmente **API-First**, onde o contrato da API é definido antes da implementação do código.

## 11.1 Design da API (API-First)

1. **Identificação de Recursos:** Defina os principais recursos (entidades) que a API irá gerenciar (ex: Clientes, Produtos, Pedidos).
2. **Definição de Endpoints e Verbos:** Mapeie as operações CRUD (Create, Read, Update, Delete) para os verbos HTTP e URLs dos recursos:
   - `POST /recursos` (Criar)
   - `GET /recursos` (Listar)
   - `GET /recursos/{id}` (Buscar por ID)
   - `PUT /recursos/{id}` (Atualizar Completo)
   - `PATCH /recursos/{id}` (Atualizar Parcial)
   - `DELETE /recursos/{id}` (Excluir)
3. **Modelagem de Dados (Payloads):** Defina a estrutura exata dos dados de requisição e resposta (JSON ou XML), garantindo consistência e clareza.
4. **Especificação do Contrato:** Crie o contrato formal da API usando ferramentas como **OpenAPI (Swagger)**. Este arquivo de especificação servirá como a única fonte de verdade para o desenvolvimento do backend e para o consumo do frontend/clientes.

## 11.2 Implementação e Codificação

1. **Geração de Código (Opcional):** Utilize o arquivo OpenAPI para gerar *stubs* de código (esqueletos) para o servidor e para os clientes, acelerando o desenvolvimento.
2. **Implementação da Lógica de Negócio:** Conecte os *endpoints* definidos à lógica de negócio e à camada de persistência de dados.
3. **Tratamento de Erros:** Implemente o tratamento de erros de forma consistente, retornando códigos de status HTTP apropriados (`4xx` para erros do cliente, `5xx` para erros do servidor) e mensagens de erro claras no corpo da resposta.
4. **Segurança:** Implemente autenticação (ex: OAuth 2.0, JWT) e autorização em todos os *endpoints*, garantindo que apenas usuários autorizados possam acessar os recursos.

## 11.3 Testes e Validação

1. **Testes Unitários:** Teste a lógica de negócio e os *controllers* da API isoladamente.
2. **Testes de Integração:** Valide o fluxo completo da API, incluindo a comunicação com o banco de dados e outros serviços.

3. **Testes de Contrato:** Use ferramentas como **Postman** ou **Dredd** para garantir que a API implementada esteja em total conformidade com o contrato definido no OpenAPI.

4. **Testes de Performance:** Verifique a latência e a capacidade de carga da API sob estresse.

## 11.4 Governança e Evolução

1. **Documentação Automática:** Utilize o arquivo OpenAPI para gerar documentação interativa (Swagger UI) que é acessível aos consumidores da API.

2. **Versionamento:** Garanta que o versionamento (ex: `v1`, `v2`) seja aplicado desde o início para permitir a evolução da API sem quebrar clientes legados.

3. **Monitoramento:** Configure ferramentas de monitoramento e alertas para rastrear o desempenho, a latência e os erros em produção.

# 12 Segurança de Software

A área de **Segurança do Software** atua como um pilar fundamental que permeia todo o processo de CI/CD na Fábrica, garantindo que a qualidade do código e do design não se restrinja apenas à funcionalidade, mas também à sua resiliência contra ameaças. O propósito central desta área é incorporar a segurança em todas as etapas do ciclo de vida do desenvolvimento, promovendo uma cultura de **Security by Design**.

Em linhas gerais, "a segurança está relacionada ao grau/nível que um produto ou sistema consegue proteger dados e informações, de tal forma que as pessoas, produtos/sistemas tenham apenas acesso adequado às informações específicas, conforme seu tipo e nível de autorização" [**?**].

Diferente de uma abordagem reativa, onde vulnerabilidades são corrigidas após a detecção em produção, o foco é na **prevenção de vulnerabilidades** desde a fase de concepção. Isso se traduz diretamente na redução de risco operacional para a Fábrica, minimizando a superfície de ataque e os custos associados a incidentes de segurança. A atuação contínua assegura a garantia de conformidade com padrões de mercado (como o **OWASP Top 10**) e a validação contínua dos artefatos de software.

## 12.1 Fundamentos Teóricos

A Engenharia de Segurança do Software na Fábrica é alicerçada em consonância com padrões globais e práticas internas consolidadas, garantindo uma abordagem técnica e abrangente.

### 12.1.1 Da Visão SWEBOK v4

O *Guide to the Software Engineering Body of Knowledge* (**SWEBOK v4**) [**?**] estabelece a segurança como uma Área de Conhecimento (KA) crítica e integrada ao ciclo de vida. Os fundamentos adotados pela Fábrica, com base nos Capítulos 13, 3 e 4 do SWEBOK, incluem:

### 12.1.2 OWASP Top 10 (2021)

O **OWASP Top 10 (2021)** [**?**] é o padrão global de conscientização sobre os riscos de segurança mais críticos para aplicações web.

### 12.1.3 Práticas de Implementação (Codificação Segura)

A codificação segura é a aplicação prática dos princípios de *Construction for Security* (SWEBOK Cap. 4) e das diretrizes do Guia de Melhores Práticas de Implementação [**?**]. Adota-se uma postura de defesa em profundidade no nível do código:

- **Gerenciamento de Segredos e Credenciais:** Segredos devem ser injetados no ambiente de execução via *Vaults* (como HashiCorp Vault, AWS Secrets Manager) e não apenas via variáveis de ambiente. Além disso, cada componente de software deve operar com o **menor conjunto de permissões estritamente necessário** para sua função (Princípio do Menor Privilégio).
- **Blindagem contra Injeção (*Input Validation* e *Output Encoding*):** É mandatório o uso de **Validação de Entrada** (*Input Validation*) para garantir que os dados recebidos estejam no formato esperado e o **Codificação de Saída** (*Output Encoding*) para neutralizar dados antes de serem renderizados no navegador (prevenção contra XSS).

Table 1: Fundamentos de Segurança do Software segundo SWEBOK v4

| **Fundamento** | **Área de Conhecimento (KA)** | **Descrição e Aplicação** |
|---|---|---|
| Segurança como disciplina de engenharia | Software Security (Cap. 13) | Tratar a segurança não como um recurso opcional, mas como um requisito não-funcional essencial, integrado ao planejamento e execução. |
| Segurança orientada a requisitos | Software Security (Cap. 13) | Definição clara de requisitos de segurança (ex: autenticação, autorização, auditoria) na fase de Engenharia de Requisitos. |
| Padrões de design seguro | Software Design (Cap. 3) | Aplicação de padrões arquiteturais que minimizem riscos (ex: menor privilégio, segregação de responsabilidades), incluindo tolerância a falhas e tratamento de erros seguro. |
| Construction for Security | Software Construction (Cap. 4) | Uso de técnicas de Defensive Programming e tratamento robusto de exceções (*Exception Handling*) para evitar estados inseguros do sistema. |
| Testes de segurança | Software Security (Cap. 13) | Incorporação de testes estáticos (SAST), dinâmicos (DAST) e testes de penetração (*Pen-testing*) como parte da Garantia da Qualidade. |
| Gerenciamento de Vulnerabilidade | Software Security (Cap. 13) | Processo contínuo de identificação, classificação, priorização e remediação de vulnerabilidades. |

- **Redução de Vazamento de Informações e Logs Seguros:** O tratamento de exceções deve evitar a exposição de detalhes técnicos em produção. A prática de **Logs Seguros** exige a sanitização de dados sensíveis (PII, senhas, tokens) antes do registro.
- **Gestão Proativa de Dependências Vulneráveis:** A Análise de Composição de Software (SCA) deve ser integrada ao pipeline de CI/CD para bloquear automaticamente *builds* que contenham dependências com vulnerabilidades críticas (**CVSS > 7.0**).

### 12.1.4 As 10 Melhores Práticas de Segurança do CERT/CC

O *Computer Emergency Response Team* (**CERT/CC**) [**?**] publica diretrizes de segurança essenciais para a construção de software seguro:

1. **Validar a entrada (*Validate input*):** Nunca confie em dados externos. Valide formato, tipo, tamanho e conteúdo de toda entrada.
2. **Prestar atenção aos avisos do compilador (*Heed compiler warnings*):** Trate avisos de compilador como erros, pois podem indicar vulnerabilidades (ex: estouro de buffer).
3. **Arquitetar e projetar para políticas de segurança:** A segurança deve ser um requisito de design. Inclua a modelagem de ameaças (*Threat Modeling*).

4. **Manter a simplicidade (*Keep it simple*):** Reduza a complexidade para minimizar a superfície de ataque e facilitar a auditoria.

5. **Negação por padrão (*Default deny*):** Por padrão, todo acesso, permissão ou funcionalidade deve ser negado.

6. **Aderir ao princípio do menor privilégio:** Cada componente deve ter apenas as permissões mínimas necessárias para executar sua função.

7. **Sanitizar dados enviados a outro software:** Remova ou neutralize conteúdo malicioso antes de enviar dados para outro componente (ex: banco de dados, navegador).

8. **Praticar defesa em profundidade (*Practice defense in depth*):** Implemente múltiplas camadas de segurança independentes.

9. **Usar técnicas eficazes de garantia de qualidade:** Utilize testes de segurança (SAST, DAST, *fuzzing*) e revisões de código rigorosas.

10. **Adotar um padrão de codificação segura:** Utilize e siga um conjunto de regras de codificação segura (ex: padrões CERT C/C++ ou Java).

## 12.2 Principais Responsabilidades

O responsável pela Segurança do Software atua como um consultor técnico e auditor, garantindo que os princípios de segurança sejam aplicados em todas as fases do desenvolvimento de software.

### 12.2.1 Na Fase de Definição e Design

Esta fase é crítica para o *Security by Design*.

- **Security Requirements:** Colaborar com a Engenharia de Requisitos (Leonardo) para definir requisitos não-funcionais de segurança claros e mensuráveis (ex: MFA).
- **Ameaças (*Threat Modeling*):** Conduzir a modelagem de ameaças para identificar potenciais vetores de ataque e vulnerabilidades no *design* antes da codificação.
- **Regras arquiteturais mínimas:** Definir padrões de segurança para a arquitetura (ex: segmentação de rede, uso de WAF, padrões de criptografia).
- **Revisão de riscos:** Avaliar o risco de segurança de novas funcionalidades ou integrações.

### 12.2.2 Durante Implementação

Apoiar a Implementação (Gabriel) na aplicação das práticas de codificação segura.

- **Codificação Segura:** Garantir que as práticas detalhadas no Guia de Implementação [**?**] (Seção 7) sejam seguidas (sanitização de *inputs*, *logs* seguros, tratamento de exceções).
- **Análise Estática de Código (SAST):** Configurar e monitorar ferramentas de SAST nos *pipelines* de CI/CD para identificar padrões de código inseguros.

### 12.2.3 Durante Testes e Entrega

Garantir que o produto final esteja em conformidade com os padrões de segurança antes da liberação.

- **Segurança em APIs:** Revisão de segurança de APIs, garantindo a aplicação correta de autenticação e autorização em todos os *endpoints*.
- **Testes de intrusão:** Coordenar e validar os resultados de testes de intrusão (*Pen-tests*) realizados por Q&A (Giovana) ou terceiros.
- **Revisão de dependências:** Auditoria final de todas as dependências de terceiros para garantir que não haja vulnerabilidades críticas conhecidas.
- **Conformidade com OWASP:** Certificar que o software não apresente nenhuma das vulnerabilidades listadas no OWASP Top 10.

Table 2: OWASP Top 10 (2021) e Práticas de Mitigação Internas

| **Categoria OWASP (2021)** | **Resumo do Risco** | **Prática Interna de Mitigação** |
|---|---|---|
| A01: Quebra de Controle de Acesso | Falhas na restrição do que usuários autenticados podem acessar ou fazer. | Implementação rigorosa de políticas de autorização em todas as camadas (API e UI). |
| A02: Falhas Criptográficas | Falhas relacionadas à criptografia de dados sensíveis em trânsito e em repouso. | Uso obrigatório de protocolos seguros (TLS/HTTPS) e algoritmos de criptografia fortes e validados. |
| A03: Injeção | Dados não confiáveis enviados ao interpretador como parte de um comando ou consulta. | Uso de *Prepared Statements* (consultas parametrizadas) e ORMs para blindagem contra SQL Injection. |
| A04: Design Inseguro | Falhas de segurança que resultam de um design ausente ou ineficaz. | Aplicação de *Threat Modeling* na fase de design e revisão arquitetural mínima. |
| A05: Configuração Incorreta de Segurança | Configurações padrão inseguras, recursos desnecessários habilitados ou erros de configuração de nuvem. | Uso de imagens base seguras e padronizadas para *deploy* e automação de validação de configuração. |
| A06: Componentes Vulneráveis e Desatualizados | Uso de bibliotecas, *frameworks* ou outros módulos de software com vulnerabilidades conhecidas. | Revisão automatizada de dependências (SAST/SCA) e atualização proativa de pacotes. |
| A07: Falhas de Identificação e Autenticação | Falhas que permitem que atacantes comprometam senhas, chaves ou tokens de sessão. | Uso de mecanismos de autenticação centralizados e fortes (MFA obrigatório). |
| A08: Falhas de Integridade de Software e Dados | Falhas na integridade de dados e *pipelines* de atualização. | Validação de integridade de *uploads* e uso de assinaturas digitais em atualizações críticas. |
| A09: Falhas de Log e Monitoramento | Falhas que impedem a detecção, escalonamento ou resposta a um ataque. | Implementação de *Logs Estruturados* (JSON) com níveis padronizados e alertas configurados. |
| A10: Falsificação de Solicitação do Lado do Servidor | O aplicativo busca um recurso remoto sem validar a URL fornecida pelo usuário. | Validação rigorosa de todas as URLs externas e uso de *whitelists* para recursos permitidos. |

# 13 Princípios Fundamentais (Clean Code)

Todo código desenvolvido na fábrica deve aderir aos seguintes princípios:

## 13.1 KISS (Keep It Simple, Stupid)

A complexidade é o inimigo da segurança e da manutenção. Evite super-engenharia. Se uma função faz "coisas demais", ela deve ser quebrada. O objetivo da fábrica não é produzir código "inteligente" que ninguém entende, mas sim código óbvio que funciona.

### 13.1.1 O que é Simplicidade?

Simplicidade não significa simplismo. Significa resolver o problema sem adicionar camadas desnecessárias de abstração ou "complexidade acidental".

- Se você precisa de um diagrama complexo para explicar uma única função de 20 linhas, ela viola o KISS.
- Se você está implementando uma estrutura genérica para "caso a gente precise no futuro", pare. (Ver princípio YAGNI - *You Ain't Gonna Need It*).

### 13.1.2 Sinais de Alerta (Code Smells)

O revisor deve rejeitar o código se encontrar:

- **Ninhada Profunda (Deep Nesting):** Muitos 'if' dentro de 'for' dentro de 'if'. Isso aumenta a carga cognitiva.
- **Funções Gigantes:** Funções com mais de 20-30 linhas geralmente fazem coisas demais.
- **Nomes Genéricos:** Variáveis chamadas 'data', 'info' ou 'manager' geralmente escondem complexidade mal definida.

### 13.1.3 Técnica Prática: Guard Clauses

Para aplicar o KISS e evitar a "seta de código" (código que cresce para a direita devido à indentação), utilize *Guard Clauses* (retorno antecipado).

```

# VIOLACAO DO KISS (Complexo e aninhado)

def process_payment(order):

    if order:

        if order.status == 'OPEN':

            if order.balance > 0:

                order.pay()

                return True

            else:

                return False

```

```
        else:

            return False

    else:

        return False


# APLICANDO KISS (Simples e plano)

def process_payment(order):

    # Validacoes iniciais (Guard Clauses)

    if not order:

        return False

    if order.status != 'OPEN':

        return False

    if order.balance <= 0:

        return False


    # Execucao principal limpa

    order.pay()

    return True
```

Listing 1: Aplicando KISS com Guard Clauses

## 13.2 DRY (Don't Repeat Yourself)

O princípio DRY preconiza que "cada parte do conhecimento deve ter uma representação única, não ambígua e definitiva dentro do sistema". Não se trata apenas de economizar digitação, mas de garantir consistência.

### 13.2.1 O Problema da Duplicação

A duplicação é a maior causa de bugs de regressão (quando algo que funcionava para de funcionar).

- **Manutenção Pesadelo:** Se a regra de validação de CPF muda, e você tem essa validação espalhada em 3 telas diferentes, a chance de esquecer de atualizar uma delas é altíssima.
- **Inconsistência:** O usuário percebe o sistema como "quebrado" quando a API recusa um dado que o Front-end aceitou (lógicas duplicadas e divergentes).

### 13.2.2 A "Regra de Três" (Rule of Three)

Evite abstração prematura. Às vezes, criar uma função genérica cedo demais aumenta a complexidade (violando o KISS). Utilize a seguinte heurística:

1. **Primeira vez:** Escreva o código.
2. **Segunda vez:** Copie e cole (se necessário), mas fique alerta.
3. **Terceira vez: Pare.** Refatore para uma função, classe ou componente reutilizável.

### 13.2.3 Falsa Duplicação (Cuidado)

Nem tudo que parece igual é duplicado. Se dois trechos de código fazem a mesma coisa, mas por **motivos de negócio diferentes** (ex: validação de cadastro de cliente vs. validação de cadastro de fornecedor), eles podem evoluir de formas diferentes. Unificá-los forçadamente cria um acoplamento ruim.

### 13.2.4 Exemplo Prático: Centralização de Lógica

```

# VIOLACAO DO DRY (Logica repetida)

# File A (Report)

final_price = product.value * 1.15 # Taxa de 15% hardcoded

print(f"Total: {final_price}")


# File B (Checkout)

total_to_pay = cart.sum * 1.15 # A mesma taxa repetida

print(f"Total: {total_to_pay}")


# ------------------------------------------------------------


# APLICANDO DRY

# File: constants.py

SERVICE_TAX_RATE = 1.15


def calculate_price_with_tax(base_value):

    return base_value * SERVICE_TAX_RATE


# Uso no sistema
```

```

final_price = calculate_price_with_tax(product.value)

total_to_pay = calculate_price_with_tax(cart.sum)
```

Listing 2: Aplicando DRY (Single Source of Truth)

## 13.3 SOLID Principles

O acrônimo SOLID representa cinco princípios de design de classes orientados a objetos. O objetivo não é seguir regras cegamente, mas criar software que tolere mudanças.

### 13.3.1 S - Single Responsibility Principle (SRP)

**"Uma classe deve ter um, e apenas um, motivo para mudar."**

Se você tem uma classe chamada `PedidoManager` que: 1) Calcula o total, 2) Salva no banco e 3) Envia e-mail de confirmação, ela está errada. Se a regra de e-mail mudar, você corre o risco de quebrar o cálculo do pedido.

```

# VIOLACAO (Classe "Deus" que faz tudo)

class Order:

    def calculate_total(self): ...

    def save_to_database(self): ... # Mistura persistencia

    def send_email_confirmation(self): ... # Mistura notificacao


# CORRETO (Cada um com sua responsabilidade)

class Order:

    def calculate_total(self): ... # Regra de negocio


class OrderRepository:

    def save(self, order): ... # Banco de dados


class EmailService:

    def send_confirmation(self, order): ... # Notificacao
```

Listing 3: Aplicando SRP

### 13.3.2 O - Open/Closed Principle (OCP)

**"Entidades de software devem estar abertas para extensão, mas fechadas para modificação."**

Você deve ser capaz de adicionar novas funcionalidades sem alterar o código fonte existente. Isso evita introduzir bugs em funcionalidades que já estão estáveis.

```
# VIOLACAO (Muitos IFs)
class Discount:
    def calculate(self, type, value):
        if type == "VIP": return value * 0.8
        elif type == "BLACK_FRIDAY": return value * 0.5


# CORRETO (Uso de Interface/Heranca)
class DiscountRule(ABC):
    @abstractmethod
    def calculate(self, value): pass


class VipDiscount(DiscountRule):
    def calculate(self, value): return value * 0.8


class BlackFridayDiscount(DiscountRule):
    def calculate(self, value): return value * 0.5
```

Listing 4: Aplicando OCP com Polimorfismo

### 13.3.3 L - Liskov Substitution Principle (LSP)

**"Subclasses devem ser substituíveis por suas classes base."**

Se a classe `B` herda de `A`, o sistema deve funcionar usando `B` no lugar de `A` sem quebrar. O exemplo clássico é: um Pinguim é uma Ave, mas se a classe `Ave` tem um método `voar()`, o Pinguim não pode herdar dela (ou lançará um erro inesperado).

```
# VIOLACAO
class Bird:
    def fly(self): ...


class Penguin(Bird):
    def fly(self):
        raise Exception("Penguins can't fly!") # Quebra o contrato!


# CORRETO
```

```python

class Bird: ... # Classe base geral


class FlyingBird(Bird):

    def fly(self): ...


class Penguin(Bird): ... # Nao herda de FlyingBird
```

Listing 5: Respeitando a Substituicao de Liskov

### 13.3.4 I - Interface Segregation Principle (ISP)

**"Muitas interfaces específicas são melhores do que uma interface única geral."**

Não force uma classe a implementar métodos que ela não usa. Isso cria dependências fantasmas.

```python

# VIOLACAO (Interface gorda)

class SmartDevice(ABC):

    def print(self): pass

    def scan(self): pass

    def fax(self): pass


class SimplePrinter(SmartDevice):

    def print(self): print("Printing...")

    def scan(self): pass # Forcado a implementar inutilmente

    def fax(self): pass  # Forcado a implementar inutilmente


# CORRETO

class Printer(ABC):

    def print(self): pass


class Scanner(ABC):

    def scan(self): pass


class SimplePrinter(Printer): ...
```

Listing 6: Segregacao de Interfaces

### 13.3.5 D - Dependency Inversion Principle (DIP)

**"Dependa de abstrações, não de implementações."**

Este é o ponto mais crucial para a **Qualidade** e **Testes**. Classes de alto nível (Regra de Negócio) não devem instanciar classes de baixo nível (Conexão MySQL) diretamente dentro delas. Elas devem receber a dependência "injetada".

```
1
2  # VIOLACAO (Alto acoplamento)
3
4  class ReportService:
5
6      def __init__(self):
7
8          # Preso ao MySQL para sempre. Dificil de testar.
9
10         self.db = MySQLConnection()
11
12
13
14 # CORRETO (Injecao de Dependencia)
15
16 class ReportService:
17
18     # Aceita QUALQUER coisa que siga o contrato "DatabaseInterface"
19
20     def __init__(self, db: DatabaseInterface):
21
22         self.db = db
23
24
25
26 # Production:
27
28 service = ReportService(MySQLConnection())
29
30 # Tests (Mock):
31
32 service = ReportService(MockDatabase())
```

Listing 7: Inversao de Dependencia

# 14 Convenções de Estilo e Nomenclatura

Embora a Fábrica de Software trabalhe com múltiplas tecnologias, a **legibilidade** é um princípio universal. Um código bem escrito deve ser autoexplicativo, independente se é Python, C# ou Java.

A responsabilidade de configurar as ferramentas de validação é da área de **Padrões**, mas a execução diária é dever de quem implementa.

## 14.1 Idioma do Código: Inglês

Para alinhar a Fábrica com padrões globais e facilitar a integração open-source, o idioma oficial do código (variáveis, funções, classes) será o Inglês.

**Exceção (Domínio Específico):** Termos de negócio estritamente brasileiros ou siglas da Algar devem ser mantidos no original para evitar perda de sentido (ex: `cpf`, `pix`, `bairro`).

## 14.2 Sintaxe: Tabela de Referência por Linguagem

Como cada linguagem tem sua "gramática" própria, respeite o padrão nativo da tecnologia:

| brainBlue | | | |
|---|---|---|---|
| **Python** | `snake_case`<br>*user_id* | `snake_case`<br>*get_user()* | `PascalCase`<br>*UserHandler* |
| **Java / TS** | `camelCase`<br>*userId* | `camelCase`<br>*getUser()* | `PascalCase`<br>*UserHandler* |
| **C#** | `camelCase`<br>*userId* | `PascalCase`<br>*GetUser()* | `PascalCase`<br>*UserHandler* |

## 14.3 Semântica de Nomenclatura (Regras Universais)

Independente da linguagem, o **significado** do nome deve seguir estas regras:

### 14.3.1 Funções são Ações (Verbos)

O nome da função deve dizer o que ela faz. Se você precisa ler o código da função para entender o nome, refatore.

- Ruim: `pdf_report()` (Parece um objeto).
- Bom: `generate_pdf_report()` (Python) ou `GeneratePdfReport()` (C#).
- **Prefixos comuns:** `get`, `set`, `is`, `has`, `calc`, `validate`.

### 14.3.2 Classes são Entidades (Substantivos)

Classes representam o "sujeito" da ação.

- Ruim: `ManageUser` (Verbo).
- Bom: `UserManager` ou `UserRepository` (Substantivo).

### 14.3.3 Variáveis Booleanas (Perguntas)

Variáveis que guardam `True/False` devem soar como perguntas de sim ou não.

- Ruim: `open`, `valid`, `admin`.
- Bom: `is_open`, `is_valid`, `has_admin_permission`.

## 14.4 Segurança de Tipos (Type Safety)

Erros de tipo são a maior causa de bugs em produção.

- **Em C#/Java:** A tipagem é obrigatória pelo compilador. Use tipos explícitos em vez de `var` sempre que a leitura ficar ambígua.
- **Em Python:** O uso de *Type Hints* é **obrigatório** nas assinaturas de métodos públicos.

```
1
2  from typing import List, Dict
3
4
5
6  # RUIM (O que e 'data'? O que retorna?)
7
8  def process(data):
9
10     return data['val'] * 2
11
12
13
14 # BOM (Contrato claro)
15
16 def process_transaction(transaction_data: Dict[str, float]) -> float:
17
18     """
19
20     Receives transaction data and returns the final value.
21
22     """
23
24     return transaction_data.get('value', 0.0) * 2
```

Listing 8: Exemplo de Tipagem (Python Reference)

## 14.5 Exemplo Prático: Refatoração e Clareza

O exemplo abaixo está em Python, mas o conceito de **"Evitar Números Mágicos"** aplica-se a C#, Java e qualquer outra linguagem.

```
1
2  # RUIM (Mistura de idiomas e numeros magicos)
3
4  # O que e 86400? Por que estamos multiplicando?
5
6  def converter_dias(lista):
7
8      res = []
9
```

```
    for x in lista:

        res.append(x * 86400)

    return res


# ------------------------------------------------------------


# BOM (Ingles Tecnico, Constantes e Clareza)

SECONDS_IN_A_DAY = 86400


def convert_days_to_seconds(days_list: List[int]) -> List[int]:

    seconds_list = []

    for day in days_list:

        seconds = day * SECONDS_IN_A_DAY

        seconds_list.append(seconds)

    return seconds_list
```

Listing 9: De Código Obscuro para Clean Code

# 15 Ferramentas de Automação (Qualidade Contínua)

Para garantir que a equipe produza código com padrão industrial e não artesanal, o uso de ferramentas de análise estática é **mandatório**.

O objetivo não é burocratizar, mas sim **automatizar o esforço operacional desnecessário**. O Code Review deve focar em lógica de negócio e arquitetura, e não em discussões sobre espaços, vírgulas ou indentação.

Nossa estratégia de automação se baseia em três pilares fundamentais, que devem ser aplicados em qualquer linguagem utilizada no projeto:

## 15.1 Pilar 1: Formatter Automatizado

Cada linguagem tem uma ferramenta que reescreve o código automaticamente para seguir o guia de estilo oficial.

- **O que faz:** Remove espaços extras, ajusta quebras de linha e padroniza a indentação ao salvar o arquivo.
- **Por que usar:** Elimina 100% das discussões subjetivas sobre estética. O código de um estagiário e de um sênior tornam-se visualmente idênticos.
- **Ferramentas Oficiais:**
  - **Python:** `Black` (Rigoroso, sem configuração).
  - **C#:** `dotnet format` (Nativo do SDK .NET).
  - **Java:** `Google Java Format` (Padrão de mercado).

## 15.2 Pilar 2: Analisador Estático (Linter)

Enquanto o formatador cuida da estética, o Linter cuida da "saúde" do código.

- **O que faz:** Analisa o código estaticamente em busca de:
  - Variáveis declaradas mas não usadas.
  - Funções complexas demais (violação do KISS).
  - Bugs lógicos óbvios (ex: `if (x == x)`).
- **Por que usar:** Impede que "code smells" (cheiro de código ruim) se acumulem, garantindo que a dívida técnica seja paga antes do commit.
- **Ferramentas Oficiais:**
  - **Python:** `Pylint` ou `Flake8`.
  - **C# / Java:** `SonarLint` (Plugin poderoso que roda direto na IDE).

## 15.3 Pilar 3: Type Checker

Erros de tipo são os bugs mais comuns e evitáveis em engenharia de software.

- **O que faz:** Garante que se uma função pede um `Número`, ela não receba um `Texto`.
- **Por que usar:** Em linguagens compiladas (C#/Java), isso é nativo, mas warnings não devem ser ignorados. Em Python, evita quebras em tempo de execução (Runtime Errors).
- **Ferramentas Oficiais:**
  - **Python:** `Mypy` (Verifica a consistência dos Type Hints).
  - **C# / Java:** O próprio Compilador (Configurado com *Treat Warnings as Errors*).

[colback=red!5!white,colframe=red!75!black,title=Regra de Ouro (Atenção)]
Código que não passa nessas ferramentas **não deve ser aceito** no repositório. O **responsável por "Padrões"** deve configurar o pipeline (CI/CD ou Pre-commit) para rejeitar automaticamente qualquer entrega fora do padrão.

# 16 Documentação e Legibilidade

Código é lido muito mais vezes do que é escrito. A documentação não serve para explicar o que o código faz (o código já diz isso), mas sim para explicar 'como usar' (interface) e 'por que foi feito assim' (decisões).

## 16.1 Regra de Ouro

- **Código ruim não deve ser documentado, deve ser refatorado.** Não escreva comentários para explicar variáveis com nomes ruins como `x` ou `val`. Renomeie-as.
- **APIs Públicas:** Toda função, classe ou método que pode ser acessado por outro módulo **deve** ter documentação formal (Docstring).

## 16.2 Padrões de Docstrings (API)

Docstrings são a documentação que acompanha o código e permite a geração automática de manuais (via Sphinx, Swagger, Javadoc). A Fábrica adota os seguintes padrões de mercado:

| brainBlue | | |
|---|---|---|
| **Python** | Google Style Docstrings | Sphinx / MkDocs |
| **C#** | XML Documentation | DocFX / Swagger |
| **Java** | Javadoc | Javadoc / Maven Site |

### 16.2.1 Estrutura Obrigatória

Uma boa documentação de função deve responder a quatro perguntas, nesta ordem:

1. **Resumo:** O que isso faz? (Verbo no imperativo: "Calcula", "Busca", "Envia").
2. **Args (Parâmetros):** O que eu preciso passar? Qual o tipo? Existem restrições?
3. **Returns (Retorno):** O que sai de lá?
4. **Raises (Exceções):** O que pode dar errado? (Essencial para quem vai fazer o `try/catch`).

### 16.2.2 Exemplo Prático (Python - Google Style)

```
1
2  # RUIM (Docstring preguicosa)
3
4  def calculate_churn(users):
5
6      """Calcula o churn."""
7
8      ...
9
10
11
12 # BOM (Padrao Google Style)
13
14 def calculate_churn_rate(active_users: int, lost_users: int) -> float:
15
16     """
```

```
    Calculates the monthly churn rate based on user data.


    Args:
        active_users (int): Total number of users at the start of the period.
        lost_users (int): Number of users who cancelled the service.


    Returns:
        float: The churn rate as a percentage (0.0 to 100.0).


    Raises:
        ValueError: If active_users is zero or negative.
    """
    if active_users <= 0:
        raise ValueError("Active users must be greater than zero.")


    return (lost_users / active_users) * 100.0
```

Listing 10: Documentacao de API Profissional

## 16.3 Comentários Internos (O "Porquê")

Enquanto a Docstring é para quem *usa* a função, o comentário é para quem *mantém* a função. Use comentários para registrar dívidas técnicas e decisões de negócio não óbvias.

- **NÃO COMENTE O ÓBVIO:**

```
    i = i + 1  # Incrementa i (INUTIL - O codigo ja diz isso)


```

- **COMENTE A DECISÃO:**

```
    # Usamos uma query bruta (SQL) aqui em vez do ORM porque
    # a performance do ORM estava causando timeout em relatorios > 1GB.
    # Ver ticket JIRA-123.
    results = db.execute_raw_sql(...)


```

## 16.4 Tags de Manutenção (Anotações)

Em um ambiente colaborativo, use tags padronizadas para sinalizar pendências no código. A maioria das IDEs mapeia isso automaticamente.

- `TODO:` Algo que precisa ser feito, mas não bloqueia a entrega atual.
- `FIXME:` Um código que funciona, mas é "gambiarra" e precisa de correção urgente.
- `DEPRECATED:` Funcionalidade antiga que será removida na próxima versão.
- `NOTE:` Um aviso importante sobre o comportamento do bloco.

```
1
2 def validate_cpf(cpf: str) -> bool:
3
4     # TODO: Implementar validacao completa com digito verificador.
5
6     # Atualmente valida apenas o tamanho para nao travar o MVP.
7
8     return len(cpf) == 11
```

Listing 11: Uso de Tags

# 17 Tratamento de Erros e Observabilidade (Logs)

Esta disciplina é a ponte entre o Desenvolvimento e a Operação. Um sistema sem logs adequados é uma "caixa preta" cara de manter.

Não logamos apenas para "debugar", logamos para 'monitorar a saúde' do negócio.

## 17.1 "A Morte do `print`"

O uso de `print()` (Python) ou `System.out.println` (Java) é "proibido" em código de produção.

- **Por quê?** Prints não possuem 'timestamp', não possuem nível de severidade (ERROR vs INFO) e, em muitas linguagens, bloqueiam a thread principal (I/O blocking), degradando a performance.
- **Solução:** Use sempre a instância de Logger configurada pelo framework (Log4j, Serilog, Python Logging).

## 17.2 Logs Estruturados (JSON)

Em vez de frases soltas, nossos logs devem ser objetos estruturados. Isso permite que ferramentas (ELK Stack, Datadog, CloudWatch) indexem os campos.

```
1
2  # RUIM (Texto Plano - Dificil de filtrar)
3
4  logger.info(f"Usuario {user_id} comprou o item {item_id}")
5
6
7
8  # BOM (Estruturado - Facil de criar dashboards)
9
10 # O log sai como um JSON: {"event": "purchase", "user_id": 123, "item": 99}
11
12 logger.info("Purchase completed", extra={
13
14     "event": "purchase_success",
15
16     "user_id": user_id,
17
18     "item_id": item_id,
19
20     "amount": 50.00
21
22 })
```

Listing 12: Texto vs Logs Estruturados

## 17.3 Níveis de Log (Padronização)

O uso incorreto dos níveis gera alertas falsos ou silêncio perigoso.

| brainBlue | |
|---|---|
| **DEBUG** | Informações granulares para desenvolvimento. **Desligado em Produção**. (Ex: Payload completo de uma requisição). |
| **INFO** | Eventos de negócio bem sucedidos. (Ex: "Pedido criado", "Job de sincronização finalizado"). |
| **WARNING** | Algo inesperado aconteceu, mas o sistema se recuperou. Não requer acordar ninguém de madrugada. (Ex: "Tentativa de login falhou", "API demorou mas respondeu"). |
| **ERROR** | Uma operação falhou. O usuário percebeu o erro. Requer investigação futura. (Ex: "Falha ao salvar no banco", "NullPointerException"). |
| **CRITICAL** | O sistema (ou uma parte vital dele) parou. Requer atuação imediata da Operação. (Ex: "Banco de dados fora do ar"). |

## 17.4 Segurança no Log (Sanitização)

[colback=red!5!white,colframe=red!75!black,title=Risco Crítico (LGPD)]
Nunca, sob hipótese alguma, logue Dados Pessoais Sensíveis (PII), Senhas, Tokens ou Chaves de API.

- Ruim: `logger.info(f''User login: {password}'')`
- Bom: `logger.info(f''User login attempt for: {username}'')`

## 17.5 Tratamento de Exceções (Exception Handling)

Tratar erros não é apenas evitar que o programa feche ("crash"), é garantir que o sistema falhe de forma segura e informativa.

### 17.5.1 Regra 1: Não engula exceções (Silent Failure)

O `catch` vazio é o maior inimigo da manutenção. Se você capturou um erro, você tem três opções:

1. **Logar e lançar:** Registra e deixa o erro subir.
2. **Recuperar:** Aplica uma lógica de correção (ex: tenta de novo).
3. **Envelopar:** Transforma uma exceção técnica em uma exceção de negócio.

### 17.5.2 Regra 2: Envelopamento (Pattern de Camadas)

Não exponha erros de banco de dados (SQL Injection risk) para o usuário final/frontend.

```

try:

    user = db.find_user(user_id)

except DatabaseConnectionError as original_error:

    # 1. Logamos o erro tecnico (para o responsavel pela area de Operacao
    ver no servidor)

```

```
    logger.error("DB connection failed", exc_info=original_error)


    # 2. Lancamos um erro limpo de negocio (para o Frontend receber)

    # O usuario recebe "Servico indisponivel", nao "Error 500 at line 40..."

    raise ServiceUnavailableError("User service is temporarily down.")
```

Listing 13: Envelopamento de Excecao (Python)

### 17.5.3 Regra 3: Correlation ID (Rastreabilidade)

Em sistemas distribuídos (como o Open Gateway), um erro pode ocorrer em um serviço profundo. Todo log deve conter um `correlation_id` (gerado na entrada da requisição) que é repassado para todas as funções internas.

```

def process_payment(order_id, correlation_id):

    try:

        payment_gateway.charge(order_id)

    except Exception as e:

        # O responsavel por Operacao consegue pesquisar pelo ID e ver todo o rastro

        logger.error("Payment failed", extra={

            "correlation_id": correlation_id,

            "order_id": order_id,

            "error": str(e)

        })

        raise
```

Listing 14: Exemplo com Correlation ID

# 18 Segurança na Implementação (AppSec)

Segurança não é responsabilidade exclusiva da área de "Segurança do Software". A vulnerabilidade nasce no momento em que o código é digitado. Adotamos a filosofia *Shift Left*: pensar em segurança desde a primeira linha de código.

## 18.1 Gerenciamento de Segredos (Credenciais)

[colback=red!5!white,colframe=red!75!black,title=Crime Capital]

**NUNCA**, sob hipótese alguma, comite senhas, tokens, chaves de API ou strings de conexão no Git. O histórico do Git é eterno.

- **Problema:** `API_KEY = ''12345''` no código.
- **Solução:** Use Variáveis de Ambiente (`.env`).
- **Ferramenta:** Em Python, use `python-dotenv`. Em C#, use `appsettings.json` (com User Secrets) ou Key Vault.

## 18.2 Blindagem contra Injeção (SQL Injection)

A falha mais antiga e comum. Ocorre quando você concatena strings para formar uma query de banco de dados.

**Regra:** Jamais concatene input de usuário diretamente em comandos SQL ou de Sistema Operacional. Use *Parameterized Queries* (Prepared Statements).

```
1
2  # VULNERAVEL (Concatenacao de String)
3
4  # Se o usuario enviar: " ' OR '1'='1 "
5
6  # Ele apaga ou le todo o seu banco.
7
8  query = f"SELECT * FROM users WHERE name = '{user_input}'"
9
10 cursor.execute(query)
11
12
13
14 # SEGURO (Query Parametrizada)
15
16 # O banco trata o input estritamente como dado, nao como comando.
17
18 query = "SELECT * FROM users WHERE name = %s"
19
20 cursor.execute(query, (user_input,))
```

Listing 15: SQL Injection: O Jeito Errado vs Certo

## 18.3 Validação e Sanitização de Entrada

Adote o princípio de **Zero Trust**. Todo dado que vem de fora (Frontend, API externa, Arquivo) é potencialmente malicioso.

- **Validação de Tipo:** Se o campo é idade, aceite apenas inteiros. Recuse strings.

- **Allow-list (Lista Branca):** Em vez de tentar bloquear caracteres ruins (o que é difícil), aceite apenas os bons.
  - *Exemplo:* Para um campo "UF", aceite apenas `[A-Z]{2}`. Qualquer outra coisa é rejeitada.

## 18.4 Vazamento de Informação (Error Handling)

Erros detalhados são úteis para o desenvolvedor, mas são mapas do tesouro para atacantes.

- **Stack Trace:** Nunca mostre o "caminho das pedras" (ex: `Line 40 in /var/www/auth.py: ConnectionRefused`). Isso revela sua estrutura de pastas e tecnologia.
- **Mensagens Genéricas:**
  - Ruim: "A senha para o usuário 'admin' está incorreta." (Revela que o usuário 'admin' existe).
  - Bom: "Usuário ou senha inválidos."

## 18.5 5. Dependências Vulneráveis (Supply Chain)

Bibliotecas modernas facilitam a vida, mas podem conter falhas. Não use versões antigas.

- O responsável pela área de Segurança do Software pode rodar scanners, mas o desenvolvedor deve estar atento aos alertas do GitHub/GitLab (Dependabot) e atualizar os pacotes (`pip`, `npm`, `nuget`) regularmente.

# 19 Integração e Fluxo de Trabalho

A área de Implementação atua como o motor da fábrica, transformando definições em produto real. Para isso, atua no centro de um fluxo de comunicação constante:

## 19.1 Fluxo de Entrada (Antes de Codificar)

Nesta etapa, o objetivo é garantir que o problema foi bem compreendido antes de gastar horas programando.

- **Engenharia de Requisitos:** O código deve resolver o problema de negócio descrito no ERS.
  - *Atenção:* Não confie cegamente apenas nos diagramas técnicos. Se o diagrama parecer contradizer a regra de negócio do ERS, consulte o responsável pela área imediatamente. A regra de negócio sempre tem precedência sobre o desenho técnico.
- **Projeto e Modelagem:**
  - **Viabilidade:** Se a arquitetura proposta ou o diagrama de classes for inviável de implementar no prazo estipulado, é dever do Implementador levantar a mão ("Push-back").
  - **Fidelidade:** O código deve refletir os diagramas. Se você precisou mudar a estrutura da classe durante o código, o diagrama precisa ser atualizado. *Código e Documentação devem andar juntos.*

## 19.2 Fluxo de Apoio (Durante a Codificação)

Você não está codando sozinho. Use os especialistas para blindar seu código.

- **Segurança:** Adote a postura de *Shift Left*. Não espere o código estar pronto para perguntar se ele é seguro.
  - *Exemplo:* Perguntando ao responsável por Segurança do Código - "Vou usar essa lib para gerar PDF, ela tem alguma vulnerabilidade conhecida?"
- **Padrões:** Se o Linter ou o Pipeline estiverem travando seu commit injustamente, acione o responsável para ajustar as regras de automação. Não tente burlar as regras locais.

## 19.3 Fluxo de Saída (Entrega)

A implementação só termina quando o próximo da fila consegue trabalhar.

- **QA e Entrega:**
  - **Smoke Test:** Nunca entregue código que "nem builda". Antes de passar para QA, rode o caminho feliz (happy path) na sua máquina.
  - **Testes Unitários:** O código deve ir para QA com a cobertura mínima de testes unitários definida no projeto. QA foca em testes integrados e de sistema, não deveria perder tempo pegando erro de sintaxe.
- **Operação:**

– **“Na minha máquina funciona”:** Essa frase é proibida. Garanta que todas as dependências novas estejam no `requirements.txt` ou `Dockerfile`.
– **Variáveis de Ambiente:** Se você criou uma nova chave ou configuração, avise o responsável da Operação para que ele possa configurá-la no ambiente de Homologação/Produção.

# 20 Checklist de Code Review (Pull Request)

O Code Review é a última linha de defesa antes de um bug ou vulnerabilidade chegar à produção. O revisor não deve aprovar o PR se qualquer um dos itens abaixo não for atendido.

## 20.1 Padrões e Legibilidade

**Idioma:** O código (variáveis, funções) está 100% em Inglês? (Exceto termos de domínio local).

**Clean Code:** Nomes de variáveis e funções revelam claramente a intenção?

**Documentação:** Funções públicas possuem *Docstrings* no padrão definido (Args, Returns, Raises)?

**Sujeira:** Código comentado, prints de debug e imports não usados foram removidos?

**Automação:** O código passou no pipeline de Linter, Formatter e Type Checker sem erros?

## 20.2 Arquitetura e Design (SOLID/KISS)

**KISS:** Existem funções complexas demais que poderiam ser quebradas? (Ninhada de `if/else`).

**DRY:** Existe lógica de negócio duplicada que deveria virar uma função auxiliar?

**Responsabilidade:** A classe/função faz apenas uma coisa? (Princípio SRP).

**Fidelidade:** A implementação reflete os diagramas e arquitetura desenhados pelo time de Projeto?

## 20.3 Segurança e Performance (Crítico)

**Segredos:** GARANTIA de que não há senhas, tokens ou chaves *hardcoded*?

**Injeção:** Queries SQL estão parametrizadas (sem concatenação de string)?

**Validação:** Inputs externos são validados e sanitizados antes do processamento?

**Loops:** Existe algum loop (`for/while`) perigoso que pode travar com grandes volumes de dados?

## 20.4 Operação e Observabilidade

**Logs:** Os logs estão estruturados (JSON)? O nível (INFO/ERROR) está correto?

**LGPD:** Garantia de que nenhum dado sensível (PII) ou senha está sendo logado?

**Tratamento de Erro:** As exceções são tratadas ou envelopadas corretamente (sem `try/catch` vazios)?

## 20.5 Testes

**Cobertura:** Existem testes unitários cobrindo o Happy Path - "Caminho Feliz" - e as principais falhas?

**Independência:** Os testes rodam isolados (Mock) sem depender de banco de dados real?

# 21 Anexo Técnico: Setup do Ambiente de Desenvolvimento

Para garantir a padronização, utilizamos automação de *git hooks*. Abaixo estão as instruções de configuração separadas por stack tecnológica.

## 21.1 Perfil A: Stack Python (Projetos de Backend / Scripts)

Este perfil utiliza o framework nativo `pre-commit` e é o padrão para projetos de ciência de dados e APIs em Python.

### 21.1.1 Instalação

O arquivo `requirements-dev.txt` deve conter: `black`, `mypy`, `pylint`, `pre-commit`.

```

# No terminal (ambiente virtual ativo):

pip install -r requirements-dev.txt

pre-commit install
```

Listing 16: Setup Python

### 21.1.2 Configuração (.pre-commit-config.yaml)

```

repos:

  - repo: https://github.com/psf/black

    rev: 23.9.1

    hooks:

      - id: black

        language_version: python3


  - repo: https://github.com/pre-commit/mirrors-mypy

    rev: v1.5.1

    hooks:

      - id: mypy

        additional_dependencies: [types-requests]


  - repo: local

    hooks:

      - id: pylint
```

```
        name: pylint

        entry: pylint

        language: system

        types: [python]

        args: ["-rn", "-sn"]
```

Listing 17: Configuração Padrão Python

## 21.2 Perfil B: Stack C# / .NET

Para projetos .NET, utilizamos a ferramenta oficial `dotnet format` combinada com hooks locais.

### 21.2.1 Instalação das Ferramentas

```
# Instala o formatador globalmente ou localmente no projeto

dotnet tool install -g dotnet-format
```

Listing 18: Setup C

### 21.2.2 Automação (Husky.Net ou Script)

Recomendamos o uso do pacote **Husky.Net** para gerenciar os commits.

```
dotnet new tool-manifest

dotnet tool install Husky

dotnet husky install
```

Adicione a tarefa no arquivo `task-runner.json` gerado pelo Husky:

```
{

   "tasks": [

      {

         "name": "dotnet-format",

         "command": "dotnet",

         "args": ["format", "--verify-no-changes"],

         "group": "pre-commit"

      }

   ]

```

```
}
```

Listing 19: Tarefa do Husky para C

## 21.3 Perfil C: Stack Java

Para Java, a validação é feita via plugins do Maven/Gradle.

### 21.3.1 Configuração no pom.xml (Maven)

Adicione o plugin **Spotless** (Formatação) e **Checkstyle** (Lint) no `pom.xml`:

```
<plugin>
  <groupId>com.diffplug.spotless</groupId>
  <artifactId>spotless-maven-plugin</artifactId>
  <version>2.40.0</version>
  <configuration>
    <java>
      <googleJavaFormat />
    </java>
  </configuration>
</plugin>
```

Listing 20: Exemplo Spotless Maven

## 21.4 Integração com IDE (VS Code)

Para feedback visual em tempo real, instale as extensões conforme sua linguagem:

- **Python:**
  - Extensão: *Black Formatter* (Microsoft)
  - Extensão: *Mypy Type Checker*
- **C# / .NET:**
  - Extensão: *C# Dev Kit*
  - Extensão: *SonarLint*
- **Java:**
  - Extensão: *Extension Pack for Java*
  - Extensão: *Checkstyle for Java*

# 22 Usabilidade

## 22.1 Visão Geral

A área de Usabilidade é responsável por assegurar que as interações entre os sistemas e seus usuários (sejam eles humanos ou outros sistemas) sejam intuitivas, eficientes e propensas ao sucesso. O objetivo é reduzir a carga cognitiva necessária para operar ou integrar as soluções da empresa.

Aqui, definimos os padrões que garantem a **Consistência** (o sistema se comporta sempre da mesma forma), a **Previsibilidade** (o usuário sabe o que esperar) e a **Recuperabilidade** (facilidade em corrigir erros), atuando como uma ponte de qualidade entre a necessidade do negócio e a implementação técnica.

## 22.2 Fundamentos Teóricos

A prática de usabilidade nesta organização é fundamentada nas seguintes Áreas de Conhecimento (KAs) do SWEBOK v4 e normas globais:

- **Software Design (Cap. 2):** Aplicação de princípios de Design de Interface de Usuário (UI) para garantir interações eficazes.
- **Software Quality (Cap. 10):** Utilização de modelos de qualidade (como a ISO/IEC 25010), onde a Usabilidade é tratada como um requisito não funcional crítico (Operabilidade e Apreensibilidade).
- **Heurísticas de Usabilidade (Nielsen/Norman):** Aplicação de princípios universais como "Visibilidade do Status do Sistema" e "Prevenção de Erros".

## 22.3 Principais Responsabilidades

A atuação da área de Usabilidade permeia todo o ciclo de vida do software, com foco na experiência de quem consome a tecnologia.

### 22.3.1 Na Fase de Definição e Design

Nesta etapa, a Usabilidade estabelece as "regras do jogo" para garantir coerência entre sistemas.

**Ação:** Criação e manutenção de Guias de Estilo (*Style Guides*) e padrões de interação. Definição de um vocabulário controlado para garantir que os mesmos termos sejam utilizados de forma consistente em todos os sistemas.

### 22.3.2 Na Fase de Avaliação (Testing)

Responsabilidade de auditar se a solução proposta é fácil de usar antes de ser massificada, em conformidade com as práticas de Software Testing (SWEBOK, Capítulo 4).

**Ação:** Avaliação heurística das interfaces e APIs. Verificação da clareza das mensagens de feedback (sucesso e erro) e da qualidade da documentação de apoio.

## 22.4 Integração com o Time

A seguir, detalha-se como a área de Usabilidade interage com as demais áreas da Software House.

### 22.4.1 Com Engenharia de Requisitos

**Entrada:** Necessidades do negócio e perfil dos usuários.

**Ação:** Garantir que o requisito não gere complexidade desnecessária. A Usabilidade valida se o fluxo proposto pela Engenharia de Requisitos é cognitivamente simples ou se exige esforço excessivo do usuário final.

### 22.4.2 Com Q&A / Testes

**Entrada:** Versões estáveis do sistema para homologação.

**Ação:** Enquanto o Q&A foca em *defeitos de código* (bugs), a Usabilidade foca em *defeitos de design* (confusão). A área de Usabilidade apoia o Q&A identificando fluxos que, embora tecnicamente corretos, induzem o usuário ao erro.

# 23 Leitura Recomendada


1 Hohpe, G., & Woolf, B. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, Addison-Wesley, 2003. Descrição: O livro clássico e fundamental sobre integração. Apresenta um catálogo de 65 padrões de integração baseados em mensageria, fornecendo um vocabulário e uma notação visual para descrever soluções de integração em larga escala. É a referência principal para entender os mecanismos de comunicação assíncrona.

2 Newman, S. *Building Microservices: Designing Fine-Grained Systems*, O'Reilly, 2015. Descrição: Embora focado em microsserviços, o livro dedica uma parte significativa à integração entre serviços, abordando comunicação síncrona (REST/RPC) e assíncrona (mensageria), além de padrões de integração de dados e transações distribuídas.

3 Fowler, M. *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2002. Descrição: Uma obra essencial sobre arquitetura de software empresarial. Embora não seja estritamente sobre integração, os padrões apresentados (como Data Mapper, Unit of Work, Repository) são cruciais para a construção de sistemas que se integram de forma limpa e eficiente.

4 Bass, L., Clements, P., & Kazman, R. *Software Architecture in Practice*, 4th ed., Addison-Wesley, 2021. Descrição: Aborda a arquitetura de software de forma abrangente, incluindo a importância das qualidades de arquitetura (como desempenho, segurança e manutenibilidade), que são diretamente impactadas pelas decisões de integração.

5 Sommerville, I. *Engenharia de Software*, 9ª ed., Pearson, 2011. Descrição: Um livro-texto clássico de engenharia de software que cobre o ciclo de vida completo do desenvolvimento, incluindo a fase de integração e teste de sistemas.

6 Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. *Distributed Systems: Concepts and Design*, 5th ed., Addison-Wesley, 2011. Descrição: Fornece a base teórica para entender os desafios e as soluções em sistemas distribuídos, que é o contexto de toda integração de software em larga escala. Cobre comunicação, concorrência, tolerância a falhas e segurança.

# References


[1] *Integração de software.*. Disponível em: https://apipass.com.br/integracao-de-software-como-funciona/

[2] *Guia completo sobre Integração de Software.*. Disponível em: https://www.techverdi.com/pt/blog

[3] *Enterprise Integration Patterns.*. Disponível em: https://www.enterpriseintegrationpatterns.com/

[4] *iPaaS vs ESB.* . Disponível em: https://latenode.com/pt-br