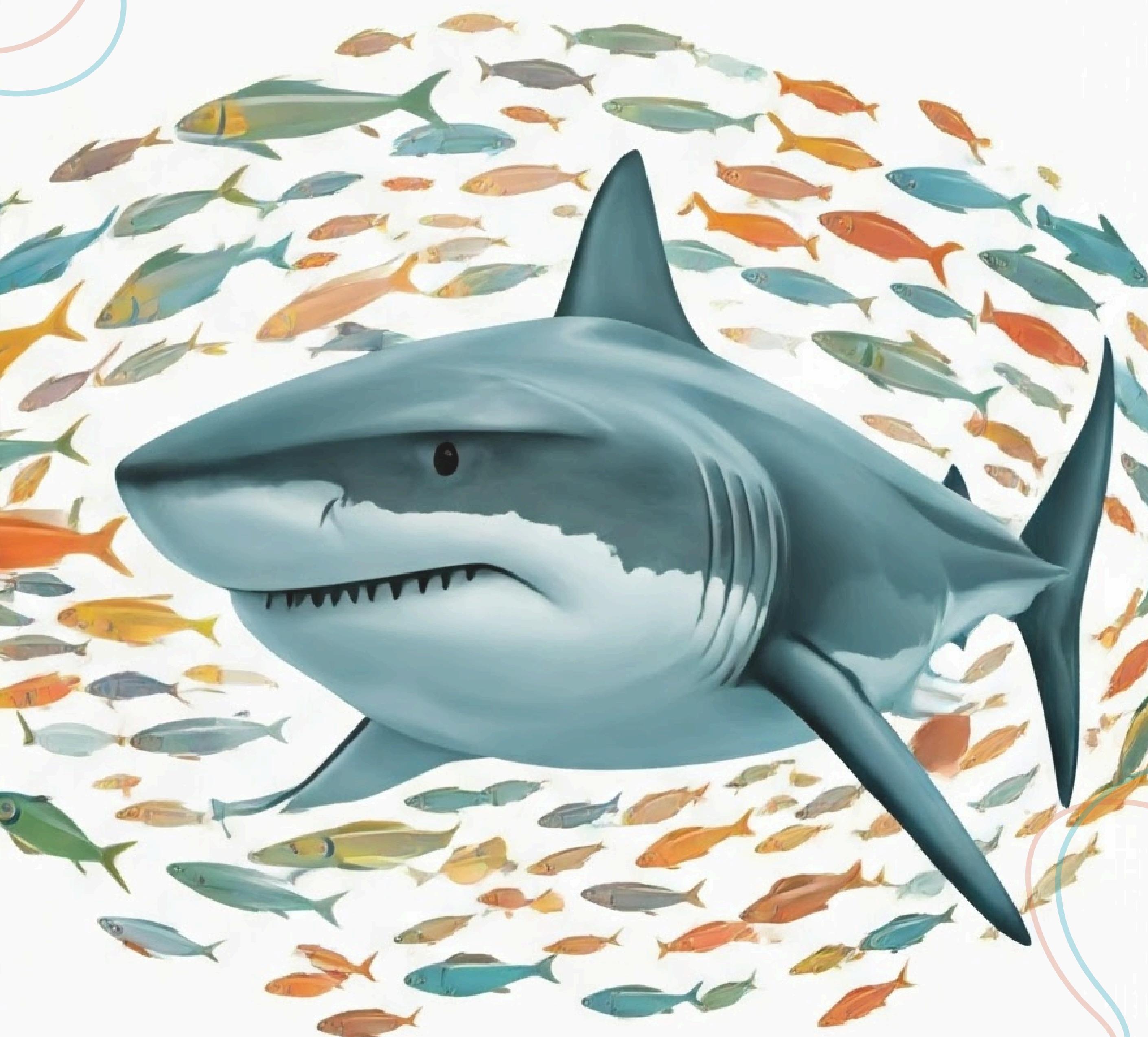


COMO PENSAR E DESENVOLVER UM SOFTWARE PARA O CRESCIMENTO SUSTENTÁVEL E ESCALÁVEL



Níkollas David Oliveira Rufino

Fabrício Carneiro Costa

Italo Gonçalves Luciano

Kauan Ribeiro Feijó Gondim

Lázaro Rytson da Silva Bezerra

Jonathan Tharles Ferreira de Oliveira

Laryssa Brilhante Pessoa

Jose Nathannael Cavalache de Alencar

Luiz Lopes de Alcântara

Antoni Demétrius Bezerra Batista

Emmanuel Soares Silva

Joely Sousa Silva

COMO PENSAR E DESENVOLVER UM SOFTWARE PARA O CRESCIMENTO SUSTENTÁVEL E ESCALÁVEL



Níkollas David Oliveira Rufino
Fabrício Carneiro Costa
Italo Gonçalves Luciano
Kauan Ribeiro Feijó Gondim
Lázaro Rytson da Silva Bezerra
Jonathan Tharles Ferreira de Oliveira

Laryssa Brilhante Pessoa
Jose Nathannael Cavalache de Alencar
Luiz Lopes de Alcântara
Antoni Demétrius Bezerra Batista
Emmanuel Soares Silva
Joely Sousa Silva

Como pensar e desenvolver um software para o crescimento

sustentável e escalável

© 2024 Copyright by Learn Skills

TODOS OS DIREITOS RESERVADOS

Capa, Diagramação e Editoração

Learn Skills

Dados Internacionais de Catalogação na Publicação (CIP) (Câmara Brasileira do Livro, SP, Brasil)

Como pensar e desenvolver um software para o crescimento sustentável e escalável [livro eletrônico]. -- Juazeiro do Norte, CE : Learn Skills Cursos, 2024.
PDF

Bibliografia.

ISBN 978-65-83475-00-8

1. Desenvolvimento de sistemas 2. Segurança
3. Software - Desenvolvimento 4. Sustentabilidade.

24-243649

CDD-004.2

Índices para catálogo sistemático:

1. Software : Arquitetura, projeto e desenvolvimento : Especificações técnicas : Ciência da computação 004.2

Eliete Marques da Silva - Bibliotecária - CRB-8/9380

Como pensar e desenvolver um software para o crescimento sustentável e
escalável

2024 Copyright by Learn Skills

A reprodução e a difusão dessa obra em qualquer formato, impresso ou eletrônico,
são livres, desde que garantidos os direitos autorais.

O conteúdo aqui apresentado é de responsabilidade exclusiva dos autores.

O padrão ortográfico e o sistema de citações e referências bibliográficas são
prerrogativas de cada autor.

É expressamente proibida a comercialização desse material para fins lucrativos.

SUMÁRIO

APRESENTAÇÃO	7
1. INTRODUÇÃO	8
2. REFERENCIAL TEÓRICO	10
2.1 Gerenciamento de Projeto	10
2.2 Arquitetura de software	11
2.3 Boas práticas de desenvolvimento de software	11
2.4 Segurança de software	11
2.5 Performance de software	13
2.6 Microsserviços	14
3. MATERIAIS E MÉTODOS	16
4. DESENVOLVIMENTO	17
4.1 Definição de Projeto e Metodologia Ágeis	17
4.1.1 Levantamento de requisitos	18
4.1.2 Criação de histórias	20
4.1.2.1 Atribuição de pontos	21
4.1.3 Definição de prioridades	23
4.1.4 Acompanhamento de desenvolvimento	24
4.2 Arquitetura de Software	26
4.2.1 Monolítico	27
4.2.2 Monolítico Simples	28
4.2.3 Monolítico Modular	28
4.2.4 Microsserviço	30
4.3 Boas práticas de desenvolvimento	33
4.3.1 Tecnologias	33
4.3.1.1 Linguagem de programação e framework	33
4.3.1.2 Banco de dados SQL	34
4.3.1.3 Banco de dados NoSQL	35
4.3.1.4 ORM	36
4.3.1.5 Docker	37
4.3.1.6 Git e Gitflow	39
4.3.2 SOLID	40
4.3.3 Código Limpo	41
4.3.4 Arquitetura limpa	43
4.3.5 Test-Driven Development (TDD)	45
4.3.6 Domain-Driven Design (DDD)	46
4.3.7 Pirâmide de testes	48
4.3.8 CI/CD (Continuous Integration/Continuous Delivery)	49
4.4 Segurança	50
4.4.1 Princípios básicos	51
4.4.1.1 Princípio do privilégio mínimo	51
4.4.1.2 Defesa em profundidade	52
4.4.1.3 Incluir a segurança no processo de entregas	53
4.4.2 As cinco funções da cibersegurança	54
4.4.2.1 Identificar	54
4.4.2.2 Proteger	54
4.4.2.3 Detectar	54
4.4.2.4 Responder	55
4.4.2.5 Recuperar-se	55
4.4.3 Credenciais e variáveis de ambiente	56

SUMÁRIO

4.4.4 Backup	56
4.4.5 Validação de inputs	57
4.4.6 Criptografia	58
4.4.7 Autenticação	59
4.4.8 Bibliotecas	61
4.4.9 Código	62
4.4.10 Logs	63
4.4.11 Proxy and Reverse Proxy	64
4.4.12 Vulnerabilidades mais exploradas	65
4.4.12.1 Controle de acesso quebrado	66
4.4.12.2 Falhas de criptografia	66
4.4.12.3 Injeção	66
4.4.12.4 Design inseguro	67
4.4.12.5 Configuração incorreta da segurança	67
4.4.12.6 Componentes vulneráveis e desatualizados	67
4.4.12.7 Falhas na identificação e autenticação	68
4.4.12.8 Falhas na integridade de software e dados	68
4.4.12.9 Falhas no registro de segurança e no monitoramento	68
4.4.12.10 Falsificação de solicitação do lado do servidor (SSRF)	69
4.4.12.11 DDOS	69
4.5 Performance	69
4.5.1 Over Engineering	70
4.5.2 Métricas: Identificando gargalos	71
4.5.3 Refatoração.....	73
4.5.4 Balanceador de carga	74
4.5.5 Cache	74
4.5.6 CQRS.....	76
4.5.7 Filas assíncronas	77
4.5.8 Pooling de conexões	77
4.5.9 Indexação de Banco de Dados	78
4.5.10 Multithreading	80
4.6 Migrando para microsserviços	81
4.6.1 Porquê microsserviços	81
4.6.2 Padrão de migração: aplicação Strangler Fig	82
4.6.3 Decomposição do banco de dados	83
4.6.4 Observabilidade	84
4.6.5 Resiliência	86
5. Conclusão	88
REFERÊNCIAS	89
INFORMAÇÕES ADICIONAIS	

APRESENTAÇÃO

Nos últimos anos, o setor de tecnologia especialmente o desenvolvimento de software, passou por uma revolução sem precedentes, trazendo uma crescente demanda por profissionais capacitados. Este crescimento acelerado gerou uma busca incessante por conhecimento e formação, com muitos indivíduos ingressando no campo na esperança de aproveitar as vastas oportunidades oferecidas pelo mercado. No entanto, essa busca rápida e intensa pela capacitação tem, muitas vezes, negligenciado a construção de uma base sólida de conhecimento, essencial para o sucesso a longo prazo.

Este livro nasce da necessidade de fornecer uma abordagem prática para aqueles que desejam não apenas desenvolver softwares funcionais, mas também criar soluções escaláveis e sustentáveis. A crescente complexidade dos projetos de software exige que os desenvolvedores, engenheiros e arquitetos de software adotem uma mentalidade estratégica, onde decisões bem fundamentadas são a chave para evitar problemas recorrentes, como duplicação de código, complexidade excessiva, falhas de segurança, e a ausência de testes automatizados, entre outros.

Aqui, exploramos a importância do planejamento meticuloso e da construção de uma base teórica robusta, antes de partir para a implementação. Comparar o desenvolvimento de software com a construção de uma casa é uma analogia poderosa: iniciar sem uma base sólida, sem compreender os requisitos e o escopo do projeto, é um erro que pode levar a retrabalho, desperdício de recursos e, por fim, ao fracasso. Este trabalho propõe uma reflexão profunda sobre a importância da fundamentação teórica no processo de desenvolvimento de software, fornecendo ferramentas práticas para enfrentar os desafios do dia a dia de quem constrói sistemas de alta qualidade e alto desempenho.

Ao longo desta obra, serão abordados temas cruciais que visam capacitar o profissional a tomar decisões informadas e assertivas, a fim de superar as dificuldades típicas do desenvolvimento de software moderno. O objetivo é que, ao concluir a leitura, o leitor tenha uma compreensão mais clara e uma visão estratégica para construir softwares que sejam não apenas funcionais, mas também escaláveis, seguros e sustentáveis a longo prazo.

Fabricio Carneiro Costa
Dezembro de 2024

1. INTRODUÇÃO

De acordo com os dados publicados por Simon Kemp na *Data Reportal* em 2023, constata-se que, em média, as pessoas passam 6 horas e 37 minutos conectadas a serviços online diariamente. Esse estudo revela que cerca de 26,5% do dia é dedicado à utilização apenas de serviços online.

Esses números evidenciam a significativa presença de softwares em nosso cotidiano, manifestando-se em diversas atividades rotineiras, como o despertador do smartphone, em momentos de entretenimento, como assistir a séries em serviços de streaming ou mídias sociais, sistemas de locomoção urbana, como o Uber, sistemas de navegação para aviões e navios, softwares do mercado financeiro e ferramentas educacionais.

Essa interconexão constante com a tecnologia destaca a relevância dessas ferramentas em nossa sociedade contemporânea. Para manter essa relevância e contribuir para o aprimoramento de nossas vidas, é crucial que os softwares sejam construídos com qualidade. Eles devem atender às necessidades dos usuários mais exigentes, garantindo disponibilidade, velocidade e segurança em suas funcionalidades.

Diante desse contexto, o tema central deste trabalho é trazer estratégias e abordagens para pensar e desenvolver software de forma a garantir seu crescimento sustentável e escalável. Apresentando questões fundamentais de engenharia de software, técnicas de escalabilidade, performance, arquitetura de sistemas, segurança e outros aspectos relevantes.

Durante o processo de criação de softwares, a negligência de aspectos fundamentais ao desenvolvimento de software, resulta em diversos problemas, estes são: sistemas vulneráveis a falhas, dificuldade de manutenção e com baixa capacidade de escalabilidade.

Observações gerais indicam a presença de desafios como não ter um bom conhecimento sobre o domínio do software, a duplicação de código, a falta de uma arquitetura adaptável, nomenclaturas inadequadas de funções, ausência de testes automatizados, deficiências na definição de requisitos. Essas falhas refletem a falta de uma compreensão holística dos princípios subjacentes ao desenvolvimento de software para um crescimento sustentável e escalável. Além disso, de acordo com o estudo produzido pelo Consortium for information & Software Quality (CISQ) em 2020, apresenta que apenas nos Estados Unidos, os softwares de baixa qualidade deram um prejuízo de U\$ 2,08 trilhões para as empresas, nesse contexto tendo falha de software, projetos malsucedidos, problemas de sistemas legados e vulnerabilidades nos sistemas.

Ademais, um estudo feito pela *Standish Group* que em seu banco de dados possui 50.000 projetos, que afirmaram que apenas 35% dos projetos foram totalmente bem sucedidos em relação ao tempo e ao orçamento, no relatório CHA-

OS 2020: *Beyond Infinity*. Portanto, pode-se notar que a falta de conhecimento e atenção aos aspectos fundamentais do desenvolvimento de software resulta em uma série de problemas recorrentes, que vão desde sistemas vulneráveis a falhas até projetos mal sucedidos e prejuízos financeiros significativos para as empresas.

Os desenvolvedores muitas vezes negligenciam a importância de compreender o domínio do software, de estabelecer uma arquitetura adaptável e de implementar boas práticas de engenharia de software, como testes automatizados e definição clara de requisitos.

Justifica-se, assim, a necessidade de abordar sistematicamente esses desafios, visando capacitar os profissionais da área a tomarem boas decisões e a implementarem soluções eficazes. A *Standish Group* afirma que as taxas de sucesso aumentam drasticamente quando se utiliza boas práticas como usar *Agile + DevOps* de forma madura, sendo uma taxa de 90% de sucesso para projetos maduros e enquanto os projetos imaturos possuem taxa de sucesso de apenas 10%. Partindo dessa premissa, a hipótese subjacente a este estudo é que a adoção de uma abordagem que priorize a fundação teórica, aliada a práticas de desenvolvimento de software bem estabelecidas, pode contribuir significativamente para a construção de sistemas de software robustos, seguros e escaláveis.

O resultado pretendido deste trabalho é fornecer conteúdos que consiga lhe direcionar de acordo com suas necessidades, guiado pelo princípio de “*avoid over-engineering*” (evitar o excesso de engenharia), para se pensar na melhor forma de como desenvolver software para o crescimento sustentável e escalável que se alinhem estritamente às suas necessidades e recursos disponíveis.

Espero que este projeto ofereça *insights* valiosos e orientações práticas para os profissionais da área, que permita-aos otimizar os processo de desenvolvimento de software, garantindo a eficiência e eficácia na entrega de soluções que atendam às expectativas e requisitos, enquanto maximiza a utilização dos recursos disponíveis, além de capacitando-os a enfrentar os desafios inerentes a seus contextos específicos e assim construir sistemas de software que atendam às demandas de um mundo tecnologicamente avançado.

2. REFERENCIAL TEÓRICO

2.1 Gerenciamento de Projeto

Para gerenciar um projeto de forma eficaz, é imperativo ter um entendimento abrangente do escopo e das necessidades envolvidas. Isso demanda uma série de questionamentos cruciais:

- Qual é a natureza do projeto?
- Quais são as demandas e expectativas dos usuários?
- O conhecimento do domínio é profundo o suficiente para guiar o desenvolvimento?
- Existem restrições de recursos a serem consideradas?
- Qual é a extensão do escopo do projeto?
- E, crucialmente, quais são os requisitos funcionais e não funcionais a serem atendidos?

Essas indagações não apenas aprofundam a compreensão do projeto, mas também servem como alicerce para a aplicação de estratégias de gerenciamento de projeto eficientes. Elas delineiam o caminho para o levantamento de requisitos, a criação de histórias de usuários, a atribuição de prioridades e o monitoramento contínuo do desenvolvimento do projeto. Além disso, influenciam diretamente na definição da arquitetura do sistema, na escolha das tecnologias a serem empregadas, nos protocolos de segurança e nos aspectos de desempenho.

Assim, o gerenciamento de projeto se revela como um guia essencial na jornada rumo à criação de software de qualidade. Para assegurar eficácia e excelência no desenvolvimento, é vital incorporar metodologias ágeis. Essas abordagens, conforme descritas por Martin (2020, p. 32) são:

um processo em que um projeto é subdividido em iterações. A saída de cada iteração é calculada e usada para avaliar continuamente o planejamento. As funcionalidades são implementadas de acordo com o valor de negócio agregado, de modo que as coisas mais valiosas sejam implementadas primeiro. O nível de qualidade é mantido o mais alto possível. O cronograma é principalmente gerenciado conforme a manipulação do escopo.

Para fundamentar tais estratégias, recorro ao livro Desenvolvimento Ágil Limpo do autor Martin (2020), utilizando suas técnicas como bússola para orientar minha abordagem. Estas questões têm o propósito de promover uma análise minuciosa do projeto, considerando uma gama de aspectos que influenciam seu desenvolvimento e sucesso. Aprofundando o entendimento da natureza do projeto e seus requisitos, abre-se espaço para a identificação de oportunidades de otimização e a adoção de práticas que visam a eficiência e a excelência no processo de desenvolvimento de software.

Portanto, a aplicação das estratégias propostas de gerenciamento de projeto contribuirá significativamente para a realização dos objetivos estabelecidos, garantindo um desenvolvimento ágil, limpo e eficiente do projeto em questão.

2.2 Arquitetura de software

Após adquirirmos um entendimento profundo do que será construído, torna-se essencial a definição da arquitetura de software, alinhada às características do projeto. Nesse sentido, exploraremos padrões arquiteturais, discutindo sua relevância e aplicabilidade no contexto do desenvolvimento de sistemas.

Dentre os temas a serem abordados, destacam-se a arquitetura monolítica e suas ramificações e a baseada em microserviços, visando identificar a mais adequada para o nosso software, considerando o momento atual e as necessidades específicas do projeto. Essa compreensão ampla desses conceitos arquiteturais é essencial para o desenvolvimento sustentável de sistemas.

Para embasar essas discussões, Criando Microserviços do autor Newman (2022). Por meio da exploração desses padrões e princípios arquiteturais, objetiva-se destacar suas vantagens, desafios e adequação a diferentes contextos e necessidades de desenvolvimento de software. A compreensão desses conceitos fundamentais orientará a tomada de decisões arquitetônicas durante o processo de desenvolvimento, assegurando a criação de sistemas robustos, escaláveis e de fácil manutenção.

Dessa forma, a análise desses padrões arquiteturais, à luz das contribuições de Newman, oferecerá insights valiosos para o desenvolvimento de software, permitindo a adoção de abordagens arquiteturais eficazes e alinhadas com as características do projeto em questão.

2.3 Boas práticas de desenvolvimento de software

Após definir a arquitetura, é crucial iniciar o desenvolvimento seguindo as melhores práticas para garantir a sustentabilidade do projeto. Para isso, é essencial realizar um levantamento abrangente das práticas que visam melhorar o desenvolvimento de software, promovendo seu crescimento sustentável e escalável ao longo do tempo. Entre essas práticas, destacam-se o princípio SOLID, o código limpo, o *Test-Driven Development* (TDD), o *Domain-Driven Design* (DDD), a pirâmide de testes, a *Continuous Integration/Continuous Delivery* (CI/CD) e a definição de camadas e limites de funções.

O princípio SOLID, por exemplo, é uma abordagem fundamental para a criação de um código modular e flexível, facilitando a manutenção e evolução do software. Já o código limpo refere-se à escrita de um código claro, legível e de fácil compreensão, essencial para a colaboração eficaz entre os membros da equipe de desenvolvimento.

O *Test-Driven Development* (TDD) é uma prática que envolve escrever testes automatizados antes mesmo de implementar o código, garantindo que o software atenda aos requisitos definidos desde o início do desenvolvimento. O *Domain-Driven Design* (DDD), por sua vez, foca na modelagem de domínios complexos, per-

mitindo uma arquitetura escalável e flexível.

A pirâmide de testes é uma estratégia valiosa para garantir a qualidade do software, priorizando a realização de testes automatizados em diferentes níveis, como testes unitários, de integração e de aceitação. A *Continuous Integration/Continuous Delivery* (CI/CD) automatiza o processo de entrega de software, reduzindo o tempo entre o desenvolvimento e a disponibilização em produção.

Por fim, a definição de camadas e limites de funções é crucial para garantir uma arquitetura sólida e bem estruturada, facilitando a manutenção e evolução do sistema ao longo do tempo.

Ao examinar essas práticas, é possível compreender melhor como criar sistemas de software robustos, flexíveis e sustentáveis, contribuindo para o sucesso do projeto de desenvolvimento. Para referenciar essas discussões, utilizarei livros como Arquitetura Limpa do autor Martin (2019), Criando Microsserviços do autor Newman (2022) e Domain-Driven Design do autor Evans (2020).

Ao examinar essas práticas, pretendo destacar suas importâncias na criação de sistemas de software robustos, flexíveis e sustentáveis. Serão discutidos os princípios subjacentes a cada prática, bem como sua aplicação prática no contexto do desenvolvimento de software.

2.4 Segurança de software

No desenvolvimento do software é necessário abordar os princípios fundamentais de segurança da informação, como o princípio do privilégio mínimo e a defesa em profundidade, é essencial compreender sua relevância na proteção dos sistemas de software contra ameaças cibernéticas.

O princípio do privilégio mínimo preconiza que os usuários devem possuir apenas os privilégios necessários para realizar suas tarefas, reduzindo assim a superfície de ataque potencial. Já a defesa em profundidade sugere a implementação de múltiplas camadas de segurança para proteger o sistema de maneira abrangente, mesmo que uma camada seja comprometida.

Além disso, é fundamental explorar as vulnerabilidades de segurança mais exploradas no mundo do software, como injeção de SQL, *cross-site scripting* (XSS) e autenticação inadequada. Para mitigar esses riscos, estratégias como a validação de dados, a encriptação de senhas, a gestão adequada de permissões no software e a identificação de usuários são essenciais. A validação de dados, por exemplo, ajuda a garantir que apenas dados válidos sejam aceitos pelo sistema, evitando assim ataques de injeção de SQL.

Ademais, políticas de *Cross-Origin Resource Sharing* (CORS) e *Rate Limiting* são utilizadas para controlar o acesso de recursos a partir de origens distintas e para limitar o número de requisições que um cliente pode fazer em um determinado período de tempo, respectivamente. O uso de bibliotecas confiáveis e a

criptografia de dados sensíveis em tráfego de rede também são práticas recomendadas para proteger as informações durante a comunicação entre os sistemas.

Para fundamentar essa análise, farei uso de referências do livro Criando Microsserviços do autor Newman (2022), também me basearei em estudos da *Open Worldwide Application Security Project* (OWASP), pela sua lista das 10 principais vulnerabilidades de segurança em sistemas de software e o minicurso *Cybersecurity Architecture Series* da IBM disponível no YouTube.

Ao explorar esses princípios e vulnerabilidades de segurança, meu objetivo é oferecer resoluções para proteger sistemas de software contra ameaças cibernéticas. Pretendo não apenas destacar como identificar e corrigir vulnerabilidades, mas também enfatizar a importância da implementação de medidas preventivas eficazes para reduzir os riscos de segurança. Buscando contribuir para a conscientização e o fortalecimento das práticas de segurança da informação no desenvolvimento de software.

Essa abordagem visa assegurar a proteção adequada dos sistemas contra ameaças e ataques, promovendo um ambiente digital seguro e confiável para os usuários finais.

2.5 Performance de software

Um dos assuntos de extrema importância em um sistema é sua performance, pois trata-se do desempenho que os usuários vão se beneficiar. Mas, é necessário se questionar sobre o uso excessivo de soluções visando melhorar o desempenho do software prematuramente, é crucial oferecer alternativas que possibilitem um monitoramento eficaz, permitindo a identificação do momento adequado para otimizações e a localização de possíveis gargalos.

Além disso, serão apresentadas soluções para escalar o software de forma eficiente, como o uso de cache para armazenamento temporário de dados frequentemente acessados, a implementação do padrão *Command Query Responsibility Segregation* (CQRS) para separar as operações de leitura e escrita, e a utilização de sistemas de filas para processamento assíncrono de tarefas. Também será abordada a prática da refatoração como uma estratégia contínua para melhorar a qualidade e a escalabilidade do código, mantendo-o limpo e organizado ao longo do tempo.

Para tratarmos essas questões, utilizarei as seguintes obras: Criando Microsserviços e Migrando Sistemas Monolíticos para Microsserviços, ambos do autor Newman (2022, 2020) e o livro Refatoração do autor Fowler (2019), reconhecido no campo da melhoria contínua de software. Também será explorado o conteúdo disponível no canal do YouTube de Fabio Akita, um experiente desenvolvedor que compartilha conhecimentos sobre escalabilidade e desempenho

de sistemas.

Ao abordar esses tópicos, a ideia é fornecer orientações práticas e fundamentadas para auxiliar os desenvolvedores na tomada de decisões relacionadas à melhoria de desempenho e escalabilidade de seus sistemas de software. Pretende-se oferecer um conjunto de estratégias acessíveis e eficazes, baseadas em princípios sólidos e experiências reais, que possam ser aplicadas de forma pragmática no contexto do desenvolvimento de software.

2.6 Microsserviços

Por fim, é crucial abordar informações essenciais sobre microsserviços, mergulhando mais profundamente nos aspectos fundamentais que regem sua aplicação e reconhecendo a importância dessa arquitetura na escalabilidade de softwares.

Nesse contexto, é fundamental compreender não apenas o que são os microsserviços, mas também o contexto ideal para sua implementação. Isso inclui uma análise cuidadosa das necessidades do projeto, das demandas do negócio e das características específicas do domínio em que o software será aplicado. Reconhecer que os microsserviços oferecem benefícios significativos em termos de escalabilidade, flexibilidade e manutenção é essencial para sua adoção eficaz.

Além disso, ao discutir a migração de uma arquitetura monolítica para microsserviços, é importante considerar não apenas os aspectos técnicos, mas também os desafios organizacionais e culturais envolvidos.

Essa transição requer não apenas mudanças na estrutura do software, mas também uma mudança na mentalidade da equipe de desenvolvimento e na forma como o trabalho é organizado e coordenado. Destacar os desafios enfrentados ao adotar essa abordagem é crucial para uma compreensão completa dos riscos e oportunidades envolvidos.

Questões como observabilidade e resiliência são aspectos críticos a serem considerados, pois garantem que o software possa lidar eficazmente com falhas e anomalias, mantendo-se operacional mesmo em condições adversas.

A observabilidade refere-se à capacidade de entender o que está acontecendo dentro de um sistema complexo, permitindo que os desenvolvedores rastreiem e investiguem o comportamento do software em tempo real. Isso é essencial para identificar e diagnosticar problemas rapidamente, garantindo uma resposta eficaz a eventos inesperados.

Já a resiliência diz respeito à capacidade do sistema de se recuperar automaticamente de falhas e interrupções, mantendo a disponibilidade e a integridade do serviço. Isso envolve a implementação de estratégias de tolerância a falhas, como redundância, isolamento de componentes e recuperação automática, garantindo que o sistema permaneça operacional mesmo diante de condições ad-

versas.

Essas considerações serão embasadas nos livros Criando Microsserviços e Migrando Sistemas Monolíticos para Microsserviços do autor Newman (2022, 2020). Ambas as obras tratam sobre arquitetura de microsserviços, incluindo diretrizes práticas para migração e abordagens para lidar com os desafios associados a essa transição. Abordando esse tema, meu propósito é trazer uma visão abrangente e fundamentada sobre a adoção de microsserviços, ajudando os profissionais a tomar decisões para a evolução de suas arquiteturas de software de acordo com sua necessidade e possibilidade. Assim, promover uma compreensão mais sólida dos princípios e práticas relacionados a microsserviços, contribuindo assim para o sucesso e a qualidade dos projetos de desenvolvimento.

3. MATERIAIS E MÉTODOS

Este projeto utilizou uma combinação de materiais de referência, voltada para desenvolvedores com pouco conhecimento teórico, mas que buscam construir software de qualidade e escalar suas aplicações com base em boas práticas e decisões informadas. Abaixo, detalho os materiais consultados escolhidos e o planejamento meticuloso que guiou a execução do projeto.

Para a pesquisa bibliográfica, selecionei um conjunto diversificado de fontes que inclui livros, artigos acadêmicos, vídeos de especialistas, e sites de grandes empresas de tecnologia.

A escolha de materiais de autores renomados como Cohn, Evans, Fowler, Martin, Newman e Woods, especialistas em desenvolvimento, gerenciamento e arquitetura de software foi estratégica, com o objetivo de adquirir uma base sólida e atualizada sobre as melhores práticas do setor. Fontes de empresas líderes, como Atlassian, GIT, IBM, JWT, Microsoft, Nginx, NIST (National Institute of Standards and Technology), OWASP (Open Worldwide Application Security Project), Redis, RedHat, foram essenciais para entender os padrões mais avançados em arquitetura e práticas de desenvolvimento.

Esses conteúdos fornecem uma visão abrangente e embasada para apoiar desenvolvedores a tomarem decisões adequadas sobre arquitetura, escalabilidade e performance em seus próprios projetos.

Essa seleção de materiais visa garantir que o conteúdo apresentado esteja alinhado com as melhores práticas e diretrizes contemporâneas, auxiliando os desenvolvedores a obter embasamento teórico confiável para construir software eficiente e sustentável. Além disso, as fontes de especialistas e empresas estabelecidas oferecem insights práticos que facilitam a aplicação direta dessas práticas ao longo do desenvolvimento.

4. DESENVOLVIMENTO

4.1 Definição de Projeto e Metodologia Ágeis

Antes de iniciar o desenvolvimento de qualquer software, é imperativo responder à pergunta fundamental: **O que será desenvolvido?** A partir dessa questão central, outras questões cruciais poderão ser abordadas, determinando o direcionamento do desenvolvimento do software. Saber qual será a função do software, ou seja, o propósito específico que ele visa cumprir, é o ponto de partida essencial. Por exemplo, no caso do Uber, a função principal é conectar pessoas sem meios de transporte a motoristas, possibilitando a prestação de serviços de transporte.

Com essa definição inicial, emergem outras questões essenciais para o desenvolvimento da solução, tais como: **Quais são as necessidades dos usuários?** Aqui, é necessário analisar o público-alvo para determinar a demanda que o software atenderá. É preciso saber se o software será utilizado por um pequeno grupo de usuários, como no caso de um produto interno de uma empresa, ou se terá projeção para atender milhares ou até milhões de pessoas em um curto período após o lançamento, exigindo alta escalabilidade.

Você possui conhecimento profundo sobre o domínio do sistema? O domínio refere-se ao conjunto de características e funcionalidades que o sistema deve possuir. Por exemplo, o domínio de um software de gestão escolar inclui aspectos como matrícula de alunos, registro de notas e horários de aula, enquanto o domínio de um software de gestão financeira abrange contabilidade, fluxo de caixa e faturamento.

Quais são as restrições de recursos? Essa questão, muitas vezes negligenciada, é de extrema importância. O desenvolvimento deve ser orientado pelos recursos disponíveis, sejam eles financeiros, temporais ou tecnológicos. Assim como na construção de uma casa, onde os andares adicionais dependem de uma fundação sólida, no desenvolvimento de software é necessário planejar o crescimento futuro sem comprometer a qualidade e a integridade do sistema.

Qual é o escopo do projeto? O escopo define os limites do projeto, estabelecendo quais funcionalidades são essenciais e quais são secundárias. Tomemos como exemplo o Mercado Livre: o objetivo principal é facilitar a compra e venda de produtos, o que exige funcionalidades como listagem de produtos, busca avançada, carrinho de compras e processos de pagamento seguros. Contudo, também existem objetivos secundários, como promover uma experiência de usuário personalizada e fornecer ferramentas de gestão de vendas. Definir um escopo inicial claro permite o desenvolvimento focado nos objetivos principais, com a possibilidade de expandir as funcionalidades conforme os recursos e o tempo permitirem.

Quais são os requisitos funcionais e não funcionais? Após definir o escopo, é necessário detalhar os requisitos funcionais, que descrevem as funções específicas que o sistema deve realizar, e os requisitos não funcionais, que abordam características

cas como desempenho, segurança e usabilidade. Os requisitos não funcionais, embora não estejam diretamente relacionados às funcionalidades específicas, são igualmente críticos para o sucesso do sistema.

Por fim, para garantir a efetividade e a qualidade no desenvolvimento de software, a incorporação de metodologias ágeis é fundamental. Conforme destaca Martin (2020), as metodologias ágeis são:

um processo em que um projeto é subdividido em iterações. A saída de cada iteração é calculada e usada para avaliar continuamente o planejamento. As funcionalidades são implementadas de acordo com o valor de negócio agregado, de modo que as coisas mais valiosas sejam implementadas primeiro. O nível de qualidade é mantido o mais alto possível. O cronograma é principalmente gerenciado conforme a manipulação do escopo.

A adoção de metodologias ágeis proporciona um ambiente propício para um desenvolvimento sustentável, promovendo a flexibilidade e a adaptabilidade necessárias para lidar com os desafios e mudanças frequentes ao longo do ciclo de vida do projeto de software. Ao permitir uma abordagem iterativa e incremental, as metodologias ágeis facilitam a rápida identificação e correção de problemas, além de possibilitar uma resposta ágil às necessidades do cliente.

4.1.1 Levantamento de requisitos

Com o escopo do projeto definido, o próximo passo crucial é o levantamento das funcionalidades que o software deverá oferecer. Esse processo de elicitação de requisitos é fundamental para garantir que todas as necessidades do projeto sejam atendidas de forma eficaz. Tomando como exemplo a plataforma de e-commerce Mercado Livre, podemos identificar uma série de requisitos funcionais e não funcionais que são essenciais para o seu funcionamento.

Requisitos Funcionais:

- **Listagem de Produtos:** O sistema deve permitir que os vendedores cadastrem produtos para venda, incluindo detalhes como título, descrição, categoria, preço e quantidade disponível.
- **Busca Avançada:** Os usuários devem ter a capacidade de buscar produtos utilizando filtros avançados, como categoria, faixa de preço, localização do vendedor, entre outros critérios.
- **Carrinho de Compras:** O sistema deve oferecer a funcionalidade de adicionar, remover e ajustar a quantidade de produtos no carrinho de compras, facilitando o processo de compra.
- **Processo de Compra:** O checkout deve ser intuitivo e seguro, permitindo que os compradores insiram informações de pagamento e endereço de entrega com facilidade.
- **Sistema de Reputação:** Deve haver um sistema de reputação para vendedores e compradores, permitindo a avaliação e feedback mútuo sobre as transações realizadas.
- **Integração de Métodos de Pagamento:** O software deve suportar uma ampla va-

riedade de métodos de pagamento, incluindo cartões de crédito, débito, transferências bancárias e plataformas como PayPal.

- **Comunicação entre Compradores e Vendedores:** A plataforma deve incluir um sistema de mensagens para facilitar a comunicação direta entre compradores e vendedores, permitindo a discussão de detalhes específicos da transação.
- **Notificações Personalizadas:** O sistema deve enviar notificações personalizadas para os usuários, informando sobre produtos de interesse, promoções e atualizações relevantes sobre suas transações.
- **Painel de Controle do Vendedor:** Deve ser disponibilizado aos vendedores um painel de controle para gestão de listagens, pedidos, pagamentos e relatórios de vendas.

Requisitos Não Funcionais:

- **Desempenho:** O sistema deve ser altamente responsivo, capaz de lidar eficientemente com grandes volumes de tráfego, especialmente durante períodos de alta demanda, como promoções.
- **Segurança:** A segurança é uma prioridade, e o sistema deve implementar medidas robustas, como criptografia de dados e autenticação de dois fatores, para proteger as transações e os dados dos usuários.
- **Confiabilidade:** A plataforma deve garantir alta disponibilidade e confiabilidade, minimizando o tempo de inatividade e assegurando a conclusão bem-sucedida das transações.
- **Escalabilidade:** O software deve ser escalável, permitindo o aumento da capacidade conforme o crescimento do número de usuários e do volume de transações.
- **Usabilidade:** A interface do usuário deve ser intuitiva, proporcionando uma experiência de uso agradável tanto para compradores quanto para vendedores.
- **Compatibilidade:** O sistema deve ser compatível com uma ampla variedade de dispositivos e navegadores, assegurando uma experiência consistente em diferentes plataformas.
- **Regulamentação:** O software deve estar em conformidade com todas as regulamentações e leis aplicáveis, incluindo aquelas relacionadas ao comércio eletrônico, privacidade de dados e proteção ao consumidor.

A partir dessa lista de requisitos, é essencial considerar a aplicação da regra da "Restrição Tripla" no gerenciamento de projetos, conforme destacado por Martin (2020). Essa regra estabelece que, em um projeto, é possível escolher apenas três dos seguintes atributos: bom, rápido, barato e concluído. Essa restrição sublinha a necessidade de avaliar cuidadosamente as limitações de escopo, prazo e recursos disponíveis para maximizar a eficiência e eficácia do desenvolvimento.

Diante dessa restrição, é crucial compreender que nem todos os requisitos de um projeto podem ser atendidos simultaneamente. A priorização é, portanto, uma etapa crítica do processo de desenvolvimento. Dependendo das restrições do projeto

pode ser necessário considerar a eliminação ou simplificação de alguns requisitos, como a redução da complexidade da escalabilidade, a limitação da compatibilidade com dispositivos apenas para web em uma versão inicial, ou até mesmo a exclusão de métodos de pagamento menos prioritários ou a função de notificações personalizadas.

A escolha de quais requisitos priorizar deve sempre ser orientada pelas restrições e objetivos específicos do projeto, lembrando que é possível escolher apenas três atributos entre bom, rápido, barato e concluído.

4.1.2 Criação de histórias

Histórias de usuário são descrições concisas de funcionalidades do software, escritas do ponto de vista do usuário final. Cada história de usuário visa capturar uma necessidade ou objetivo específico que o usuário deseja alcançar ao interagir com o sistema.

Essas histórias normalmente seguem uma estrutura simples, descrevendo o que o usuário pretende realizar e estabelecendo critérios de aceitação que determinam quando a funcionalidade é considerada completa e satisfatória. Elas desempenham um papel fundamental no alinhamento das expectativas entre os *stakeholders* e a equipe de desenvolvimento, garantindo que o produto final atenda às necessidades reais dos usuários. Tomando como exemplo o requisito de "Processo de Compra", que demanda um processo de checkout claro e intuitivo, onde os compradores possam inserir informações de pagamento e endereço de entrega, podemos desmembrar esse requisito em várias histórias de usuário.

História de Usuário 1: Revisão do Carrinho de Compras

Como comprador, desejo revisar meu carrinho de compras antes de iniciar o processo de checkout, para confirmar os itens selecionados.

Critérios de Aceitação: Deve haver um botão visível em todas as páginas do site para acessar o carrinho de compras. O carrinho de compras deve exibir claramente todos os itens selecionados, suas quantidades e preços. Os compradores devem poder adicionar, remover ou ajustar a quantidade dos itens diretamente no carrinho de compras.

História de Usuário 2: Seleção do Método de Pagamento

Como comprador, desejo selecionar o método de pagamento preferido durante o processo de checkout, para concluir a transação de forma conveniente.

Critérios de Aceitação: Deve haver uma opção para selecionar o método de pagamento desejado, incluindo opções como cartão de crédito, débito, e Pix. Os compradores devem ser guiados por um processo passo a passo durante o checkout, com instruções claras sobre como inserir informações de pagamento. O sistema deve validar e processar as informações de pagamento de maneira segura e confiável, utilizando um gateway de pagamento, como o Stripe.

História de Usuário 3: Inserção do Endereço de Entrega

Como comprador, desejo inserir meu endereço de entrega durante o processo de checkout, para garantir que os itens sejam entregues no local desejado.

Critérios de Aceitação: Devem ser fornecidos campos claros e intuitivos para a inserção do endereço de entrega, incluindo nome, endereço, cidade, estado, CEP, bairro (opcional) e ponto de referência (opcional). O sistema deve oferecer opções de entrega, como entrega padrão e expressa, integradas via API dos Correios. Os compradores devem poder revisar e confirmar o endereço de entrega antes de finalizar a compra.

História de Usuário 4: Revisão e Confirmação da Compra

Como comprador, desejo revisar e confirmar todos os detalhes da minha compra antes de finalizar o processo de *checkout*, para garantir precisão e evitar erros.

Critérios de Aceitação: Deve haver uma página de resumo da compra, exibindo todos os detalhes do pedido, como itens selecionados, métodos de pagamento, endereço de entrega e custos totais. Os compradores devem poder revisar e editar as informações do pedido, se necessário, antes de prosseguir para a etapa final de finalização da compra. Deve haver um botão claramente visível para confirmar e finalizar a compra após a revisão dos detalhes do pedido.

Essas histórias de usuário proporcionam uma compreensão clara e objetiva dos requisitos do sistema, mantendo o foco nas necessidades do usuário final. Elas são essenciais para garantir que a equipe de desenvolvimento esteja alinhada com as expectativas dos *stakeholders*, facilitando a comunicação e a colaboração entre os membros da equipe.

Além disso, as histórias de usuário auxiliam na priorização das tarefas, na estimativa de esforço necessário para a implementação e na avaliação do progresso do desenvolvimento do software, contribuindo para um ciclo de desenvolvimento mais eficiente e orientado aos resultados.

4.1.2.1 Atribuição de pontos

Após a criação das histórias de usuário, é essencial reunir a equipe de desenvolvimento para atribuir pontuações a cada uma delas. Essas pontuações refletem o nível de dificuldade e complexidade envolvido na implementação das funcionalidades descritas. Um método comum é utilizar uma escala de 1 a 6 pontos, onde 1 representa menor complexidade e 6 maior complexidade. Atribuir pontos de forma colaborativa ajuda a estabelecer uma estimativa mais precisa do esforço necessário para concluir as tarefas.

O processo de atribuição de pontos geralmente começa com a escolha de uma “história de ouro”, que servirá como referência para classificar as demais. A história de ouro é uma história de usuário que a equipe considera de complexidade moderada e pode ser utilizada como base comparativa para as outras. Uma vez selecionada a história de ouro, as demais histórias são pontuadas em relação a ela. 21

Para ilustrar o processo, consideremos as histórias de usuário descritas anteriormente:

História de Ouro: Revisão do Carrinho de Compras (História 1)

Nesta história, o carrinho de compras já foi implementado, mas agora deve ser tornado visível em todo o site. Além disso, o usuário deve poder revisar itens, quantidades e valores, com a opção de modificar o conteúdo do carrinho.

A maioria dos desenvolvedores classificou esta história com 2 pontos, considerando que, embora envolva manipulação de dados e interface, o nível de complexidade é moderado. Apenas um desenvolvedor sugeriu 1 ponto, mas a decisão final foi manter a pontuação em 2, refletindo a complexidade relativa da tarefa.

História 2: Seleção do Método de Pagamento

Esta história envolve a implementação de um processo de *checkout*, incluindo a integração com um gateway de pagamento (por exemplo, Stripe). Inicialmente, houve divergência entre os desenvolvedores, com alguns atribuindo 4 pontos e outros 6 pontos.

Após uma discussão mais aprofundada, onde foi destacado que o uso de um *gateway* de pagamento pode simplificar a implementação, a maioria concordou em atribuir 4 pontos à história. Essa decisão reflete um consenso sobre a complexidade moderada, mas maior do que a da história de ouro.

História 3: Inserção do Endereço de Entrega

A História 3 foi avaliada em 2 pontos, similar à história de ouro. Embora envolva a conexão com uma API externa para calcular o frete, a complexidade geral é considerada baixa. A tarefa pode ser realizada dentro de um período de tempo comparável ao da História 1, justificando assim a mesma pontuação.

História 4: Revisão e Confirmação da Compra

Finalmente, a História 4 foi classificada com 1 ponto, visto que grande parte das funcionalidades necessárias para sua implementação já foram desenvolvidas em histórias anteriores. Esta história depende da conclusão das anteriores, mas, por si só, requer um esforço relativamente pequeno, justificando a pontuação mínima.

A utilização da história de ouro como referência é fundamental para garantir uma atribuição de pontos mais precisa e consistente. Isso permite que a equipe de desenvolvimento avalie a complexidade das tarefas de forma mais assertiva, possibilitando um planejamento e acompanhamento mais eficazes do progresso do projeto. Em um contexto mais amplo, esse processo de atribuição de pontos contribui para uma gestão de projeto mais eficiente, que será explorada em seções subsequentes.

4.1.3 Definição de prioridades

A definição das prioridades das histórias de usuário é uma etapa crucial no planejamento do desenvolvimento de software. A priorização deve ser orientada pelo conceito de retorno sobre o investimento(ROI), focando nas histórias que proporcionam maior valor ao negócio. É importante considerar que algumas histó-

histórias dependem de outras para serem realizadas, o que exige uma abordagem cuidadosa para garantir um desenvolvimento eficiente e coeso.

Uma ferramenta útil para auxiliar na priorização é a matriz de custo e valor, que permite classificar as histórias de usuário com base em seu valor para o negócio e no custo de implementação. A matriz pode ser organizada da seguinte forma:

		CUSTO ALTO	CUSTO BAIXO
VALOR ALTO	CUSTO ALTO	Faça Depois	Faça Agora
	CUSTO BAIXO	Nunca Faça	Faça por último

A partir dessa matriz, é possível classificar as histórias de usuário discutidas anteriormente:

História 1 (Revisão do Carrinho de Compras)

Esta história possui um valor alto, pois é essencial para a experiência do usuário, e um custo baixo, devido à sua baixa complexidade. Assim, ela se enquadra na categoria "faça agora", sendo uma prioridade imediata no desenvolvimento.

História 2 (Seleção do Método de Pagamento)

Apesar de também ter um valor alto, esta história apresenta um custo elevado devido à integração com APIs externas, como o gateway de pagamento. Por isso, ela se enquadra na categoria "faça depois". Embora seja crucial para o funcionamento do sistema, sua complexidade sugere que deva ser abordada após as histórias de menor custo.

História 3 (Inserção do Endereço de Entrega)

Com valor alto e custo baixo, esta história também se enquadra na categoria "faça agora". No entanto, devido ao seu valor ligeiramente inferior e ao custo maior em comparação com a História 1, ela deve ser priorizada após a conclusão da primeira.

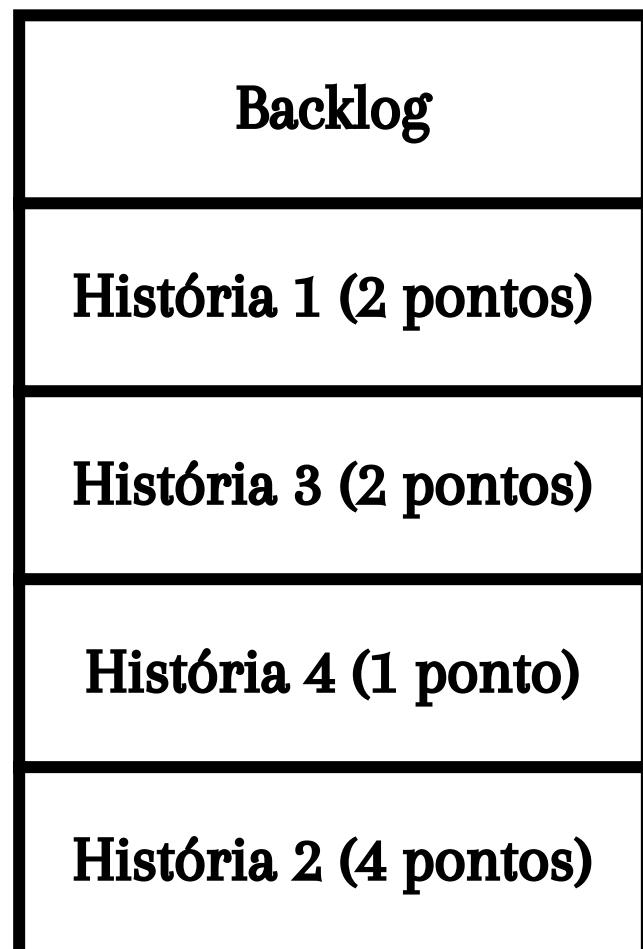
História 4 (Revisão e Confirmação da Compra)

Esta história, embora essencial para a conclusão do processo de compra, depende de funcionalidades previamente desenvolvidas. Ela possui valor alto e custo baixo, mas, devido à sua dependência de outras histórias, deve ser classificada para ser realizada após as anteriores.

Com base nessa análise, a sequência de implementação recomendada seria: História 1, História 3, História 4 e, por fim, História 2. Essa classificação deve ser realizada em conjunto com os stakeholders, ou seja, as partes interessadas que têm influência ou são impactadas pelo projeto. Os stakeholders podem incluir clientes, gerentes, investidores, desenvolvedores e proprietários do projeto. A participação ativa dessas partes é essencial para garantir que as decisões de priorização estejam alinhadas com os objetivos estratégicos do negócio e as necessidades dos usuários.

4.1.4 Acompanhamento de desenvolvimento

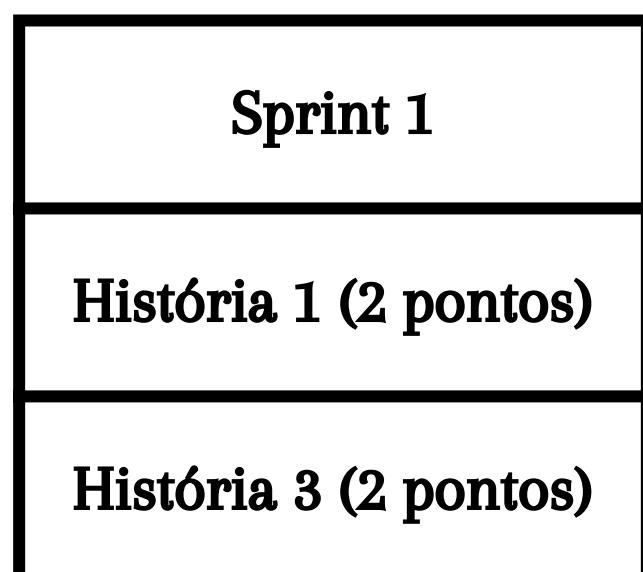
Após a atribuição de pontuações e a definição das prioridades das histórias de usuário, o próximo passo é iniciar o desenvolvimento e seu acompanhamento sistemático. Esse processo começa com a criação de um Backlog, onde todas as histórias de usuário são organizadas em ordem de prioridade.



O *Backlog* serve como um repositório centralizado de tarefas, permitindo que a equipe de desenvolvimento tenha uma visão clara das histórias que precisam ser realizadas, organizadas pela sua importância estratégica.

A partir do *Backlog*, inicia-se a *sprint*, que é uma iteração de desenvolvimento onde um conjunto específico de histórias é selecionado para ser concluído. Cada sprint geralmente dura entre uma a duas semanas, mas, para o exemplo em questão, consideramos uma duração de uma semana.

Nesse caso, como é a primeira *sprint* não temos uma média de pontos por iteração, então os desenvolvedores vão definir quais histórias eles conseguem fazer na primeira *sprint*, no nosso exemplo a primeira ficou assim:



Nossa equipe de desenvolvimento definiu que em uma *sprint* consegue finalizar essas duas histórias, que dar o total de 4 pontos. Suponhamos que ao final dessa sprint, a equipe conseguiu finalizar realmente essas duas tarefas e não conseguiu pegar outra história do *Backlog*, portanto, já podemos identificar que nossa equipe consegue em média realizar 4 pontos por interação. Então a partir dessa média, podemos planejar a próxima sprint, que deve ter 4 pontos, ficando assim:

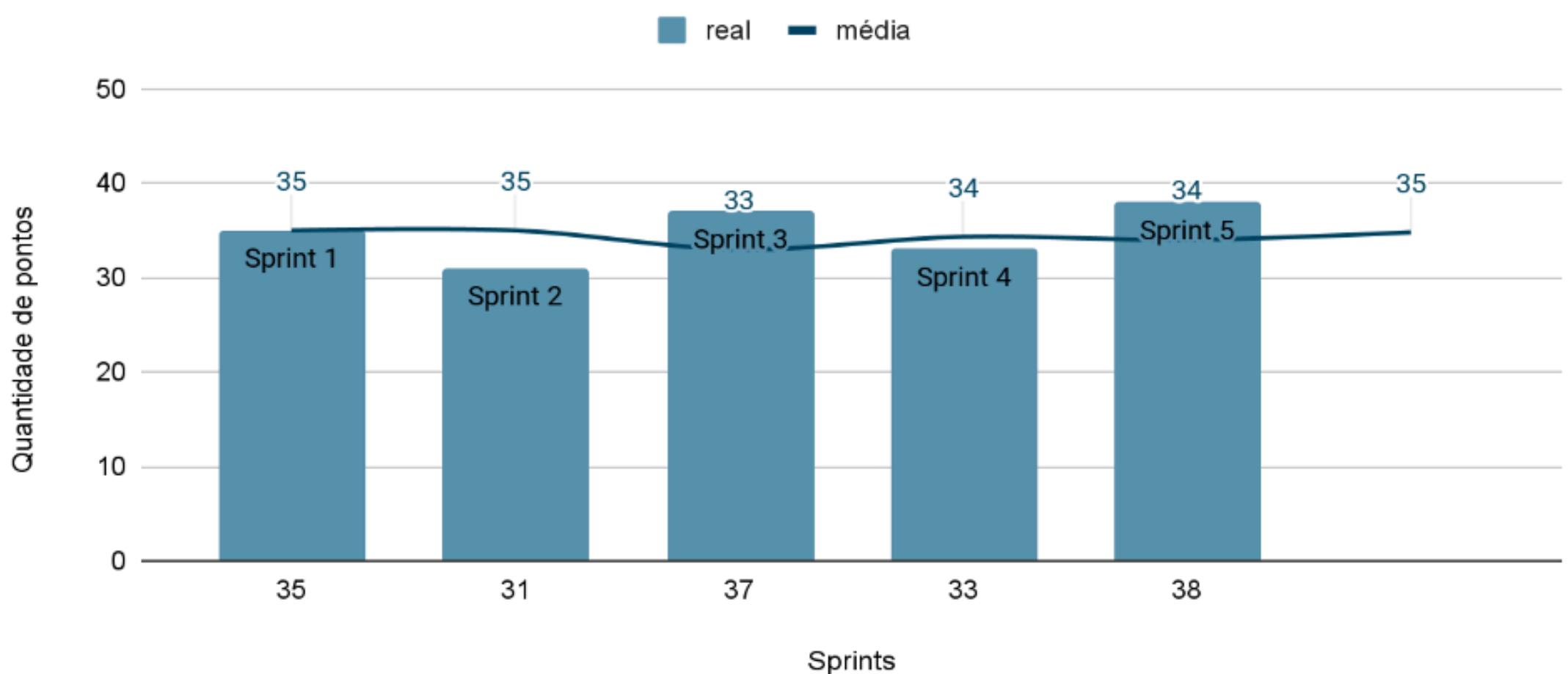


Por conta da prioridade essa sprints ficou com uma pontuação maior que a média de interações, logo, é possível que as duas histórias não sejam concluídas nesta *sprint*, sendo necessário passar para a próxima as atividades pendentes.

Voltando a suposição do nosso exemplo, ao final dessa *sprint*, nossa equipe conseguiu finalizar apenas a História 4 e desenvolver uma parte da História 2, portanto, nossa *sprint* teve apenas 1 ponto pelas histórias concluídas, mas podemos atribuir os pontos da história que não foi concluída, no caso, a História 2 está 75% concluída, como ela possui 4 ponto, 75% significa 3 ponto, logo, nosso interação teve 4 pontos. Por fim, temos apenas 1 ponto de uma história remanescente, como conseguimos fazer 4 pontos por *sprint*, nosso projeto será finalizado na próxima interação.

Depois de algumas *sprints*, podemos ter real noção de quantos pontos a equipe consegue em média realizar por interação, no nosso caso, esse valor é 4 e com isso podemos montar um gráfico de pontos que irá auxiliar no nosso cronograma.

Vamos supor um exemplo no qual temos um *Backlog* com 250 pontos e já se passaram 5 *sprints*, com as seguintes pontuações que temos no gráfico:



À medida que mais *sprints* são realizadas, torna-se possível calcular uma média de pontos por *sprint* com maior precisão. Por exemplo, se após 5 *sprints* a equipe tiver concluído um total de 174 pontos de um *Backlog* com 250 pontos, restando 76 pontos, pode-se estimar que serão necessárias mais 3 *sprints* para finalizar o projeto, considerando uma média de 35 pontos por *sprint*.

O acompanhamento do progresso por meio de gráficos de pontos é uma ferramenta valiosa, pois permite prever datas de conclusão com base em dados con-

cretos. Além disso, o monitoramento contínuo das *sprints* oferece insights sobre a qualidade do código e a eficiência do desenvolvimento. Uma manutenção estável da pontuação ao longo das *sprints* sugere um projeto organizado e de alta qualidade, enquanto uma pontuação decrescente pode indicar problemas como acoplamento excessivo, baixa qualidade nos testes ou duplicação de código.

Para complementar o acompanhamento, é essencial realizar reuniões diárias, conhecidas como *Daily Meetings*. Estas reuniões rápidas, com duração de 10 a 15 minutos, são realizadas no início de cada dia de trabalho. Durante a *daily*, cada membro da equipe responde a três perguntas-chave:

- O que eu fiz desde a última reunião?
- O que farei até a próxima reunião?
- Existe algum impedimento em meu caminho?

Todos têm cerca de 30 segundos para responder cada uma dessas perguntas. Essas perguntas têm o objetivo de alinhar a equipe, facilitando a comunicação e permitindo que problemas sejam identificados e resolvidos rapidamente. Ao garantir que todos estejam cientes do andamento do projeto e das atividades dos outros membros, as *dailies* promovem a colaboração e a eficiência no desenvolvimento do software.

4.2 Arquitetura de Software

A escolha da arquitetura de software é um dos passos mais críticos na definição do comportamento de um sistema, pois envolve a organização dos componentes, as interações entre eles e as diretrizes que governam o design e a evolução do software ao longo do tempo. A arquitetura serve como um plano de alto nível que guia decisões de design e desenvolvimento, influenciando diretamente aspectos como qualidade, escalabilidade, desempenho, segurança e manutenção do sistema.

A arquitetura de software atua como uma abstração que captura as principais decisões de design, estabelecendo a estrutura fundamental sobre a qual o sistema será construído. Essas decisões têm um impacto duradouro no sucesso do projeto, moldando não apenas a forma como o software será desenvolvido, mas também como ele será mantido e evoluído ao longo do tempo. Uma arquitetura bem projetada pode facilitar a escalabilidade, melhorar o desempenho, aumentar a segurança e simplificar a manutenção, enquanto uma arquitetura inadequada pode levar a problemas de desempenho, dificuldades na manutenção e até mesmo falhas de segurança.

Por essas razões, é essencial definir a arquitetura do software antes de iniciar o desenvolvimento. Essa definição deve ser baseada no tipo de software que se deseja criar, levando em consideração os requisitos funcionais e não funcionais, as restrições de projeto, e o contexto tecnológico em que o software será desenvolvido e operado. A escolha da arquitetura deve refletir a visão do sistema como um todo, integrando diferentes componentes de maneira coesa e eficiente.

4.2.1 Monolítico

Arquitetura monolítica é quando todas as funcionalidades de um sistema tiverem de ser implementadas em um único conjunto, assim é a definição de Newman (2022). Isso implica que, independentemente da diversidade de domínios de negócio presentes em um software — como relatórios, pagamentos, usuários e autenticação —, todos devem ser desenvolvidos e operados como um bloco único. Embora essa abordagem possa trazer desafios, especialmente na escalabilidade, onde a necessidade de escalar um domínio específico, como relatórios, exige que todos os outros domínios sejam escalados simultaneamente, acarretando em impactos mútuos, ela também oferece vantagens significativas.

Newman (2020), destaca que a simplicidade inerente à arquitetura monolítica evita complexidades desnecessárias comuns em sistemas distribuídos. Isso facilita o fluxo de desenvolvimento, permitindo que melhorias sejam implementadas mais rapidamente pelos desenvolvedores. Além disso, atividades como monitoramento, refatoração, questões de desempenho e testes tendem a ser mais simplificadas em um ambiente monolítico, onde todos os componentes estão centralizados. A reutilização de código é outro benefício, resultando em um trabalho mais eficiente e menos complexo.

A arquitetura monolítica é particularmente adequada em contextos específicos. Como Sam Newman (2020) argumenta, para produtos e startups emergentes, a implementação de uma arquitetura de microsserviços pode ser inadequada, pois o domínio de negócio ainda está em fase de definição e pode sofrer alterações significativas. Mudanças frequentes nas fronteiras dos serviços em uma arquitetura de microsserviços podem gerar custos elevados. Além disso, a validação do produto deve preceder a criação de sistemas complexos destinados a atender grandes demandas.

Outros fatores, como limitações de recursos e equipe, também podem influenciar a escolha de uma arquitetura monolítica, dado que o desenvolvimento de microsserviços é mais complexo e custoso. Portanto, para projetos novos, de menor complexidade, com recursos técnicos limitados ou quando o conhecimento do domínio ainda é superficial, a arquitetura monolítica se apresenta como a melhor opção.

Entretanto, é crucial adotar boas práticas de design ao desenvolver um sistema monolítico, especialmente se houver a possibilidade de que o software precise, no futuro, atender a uma demanda maior e migrar para uma arquitetura de microsserviços. Nesse caso, a separação do monolítico em módulos bem definidos pode facilitar uma eventual transição, tornando essa abordagem viável para produtos em fase de validação e sem certeza sobre a adesão do mercado.

Por outro lado, para sistemas que atendem a um número restrito de usuários, com baixa complexidade e demanda limitada, um monolítico simples e bem estruturado pode ser a escolha mais assertiva.

4.2.2 Monolítico Simples

A arquitetura monolítica simples é caracterizada pela implementação de todas as funcionalidades de um software em uma única unidade indivisível. Nesse modelo, todos os componentes do sistema estão fortemente acoplados, o que significa que qualquer alteração em uma parte pode exigir modificações em outras partes inter-relacionadas. Isso proporciona uma vantagem significativa em termos de simplicidade e velocidade no desenvolvimento, uma vez que o sistema é gerido como um único artefato, eliminando a necessidade de criar e manter fronteiras bem definidas entre diferentes domínios de negócio.

A simplicidade estrutural do monolítico simples facilita não apenas o desenvolvimento, mas também aspectos como segurança, desempenho, monitoramento e gestão da infraestrutura. A ausência de complexidade inerente à arquitetura distribuída permite que o sistema seja mais facilmente monitorado e otimizado, sem as complicações associadas à coordenação de múltiplos serviços ou módulos. No entanto, essas vantagens são limitadas a sistemas de baixa complexidade, onde as demandas em termos de escalabilidade e modularidade são mínimas.

Esse tipo de arquitetura é particularmente recomendado para aplicações que não exigem uma escalabilidade robusta ou que possuem um ciclo de vida curto, onde a simplicidade e a rapidez de desenvolvimento são prioritárias. Para sistemas que atendem a uma pequena base de usuários ou que não precisam lidar com uma grande variedade de domínios de negócios, a arquitetura monolítica simples se apresenta como uma solução prática e eficiente. Contudo, à medida que a aplicação cresce em termos de complexidade e demanda, as limitações desse modelo podem se tornar evidentes, especialmente se houver a necessidade de escalabilidade ou de uma gestão mais granular dos componentes do sistema.

Por essas razões, a escolha pela arquitetura monolítica simples deve ser feita com cautela, levando em consideração a natureza do projeto, as expectativas de crescimento e os recursos disponíveis. Embora essa abordagem possa ser ideal para projetos de pequeno porte, seu uso em sistemas mais complexos ou com necessidades futuras de escalabilidade pode resultar em desafios significativos, exigindo uma possível reestruturação ou migração para arquiteturas mais modulares no futuro.

4.2.3 Monolítico Modular

A arquitetura monolítica modular se distingue pela organização do sistema em módulos que correspondem a diferentes domínios de negócios, como relatórios, pagamentos, usuários e autenticação. Esses módulos são projetados para serem independentes ou semi-independentes, permitindo a definição clara de fronteiras entre os domínios, com o objetivo de minimizar o acoplamento entre as partes do sistema. Apesar dessa independência, ainda é possível reutilizar códigos e funções de outros módulos, o que evita a duplicação de código e promove a eficiência no desenvolvimento. Newman (2022), em seu livro Criando Microsserviços define que o monolítico modular é uma variação na qual o processo único do monolítico é composto de módulos separados.

Cada módulo dentro de um sistema monolítico modular opera como uma unidade quase autônoma, interagindo com os demais por meio de interfaces bem definidas. Isso traz diversas vantagens, como a possibilidade de desenvolver, testar, implantar e manter os módulos de maneira mais isolada, o que facilita o trabalho das equipes de desenvolvimento. Além disso, essa abordagem modular permite que diferentes equipes trabalhem simultaneamente em módulos distintos, o que pode acelerar o processo de desenvolvimento.

Para implementar uma arquitetura monolítica modular, é fundamental dividir o sistema em módulos que representem diferentes domínios de negócios, mantendo a coesão e reduzindo o acoplamento. O passo a passo a seguir detalha como realizar essa implementação:

- **Definição dos Domínios de Negócio:** Inicie identificando os principais domínios de negócio da aplicação, como relatórios, pagamentos, usuários, autenticação e quaisquer outros requisitos específicos. Cada domínio deve refletir uma área específica da lógica de negócios, garantindo que suas funcionalidades e regras sejam agrupadas dentro de um módulo dedicado. Entenda as interdependências entre esses domínios para planejar a organização do código e minimizar a comunicação entre os módulos.
- **Criação de Módulos para Cada Domínio:** Com os domínios definidos, crie módulos para cada um, onde cada módulo deve ser responsável por uma funcionalidade específica do domínio. A estrutura de cada módulo deve incluir suas próprias classes, interfaces, repositórios, serviços e controladores, encapsulando toda a lógica de negócio relacionada ao domínio em um local específico. Por exemplo, o módulo de usuários deve conter todas as operações de cadastro, consulta, atualização e remoção de usuários, enquanto o módulo de pagamentos gerencia transações e métodos de pagamento.
- **Definição de Fronteiras Claras entre os Módulos:** Garanta que cada módulo possua fronteiras bem definidas, interagindo com outros módulos somente por meio de interfaces públicas ou APIs internas. Evite o acesso direto aos dados de um módulo por outro; em vez disso, crie métodos de acesso ou serviços que atuem como interfaces, minimizando a dependência entre os módulos.
- **Implementação de Interfaces de Comunicação Interna:** Defina interfaces para permitir a comunicação entre módulos onde necessário, garantindo que a interação seja mínima e focada em dados essenciais. Por exemplo, se o módulo de relatórios precisar de dados de usuários, utilize uma interface pública para acessar os dados, mantendo as fronteiras de domínio preservadas. Utilize padrões de projeto, como o *Facade* ou *Adapter*, para simplificar a comunicação e isolar mudanças nos módulos.
- **Organização do Banco de Dados:** Opte por uma única base de dados com belas relacionadas a cada módulo, mas separe-as logicamente para refletir os domínios. Considere o uso de schemas distintos para cada módulo, se o banco de dados suportar essa funcionalidade, para organizar melhor os dados e reforçar a modularidade. Planeje a estrutura de dados de modo que as mudanças em um domínio tenham o menor impacto possível nos outros.

- **Desenvolvimento e Testes Isolados para Cada Módulo:** Desenvolva cada módulo de forma independente, usando testes unitários para validar a funcionalidade e as interações dentro do domínio. Implemente testes de integração para verificar a comunicação entre módulos, assegurando que as interfaces sejam seguras e funcionem conforme esperado. Esse isolamento nos testes facilita a manutenção e permite identificar problemas mais rapidamente.
- **Empacotamento e Implantação como um Monólito Modular:** Ao empacotar o sistema, mantenha a organização modular, mesmo que ele seja implantado como uma única unidade. Essa organização modular permitirá migrações futuras, como para uma arquitetura de microsserviços, caso seja necessário escalar partes específicas do sistema. Garanta que a documentação do código reflita as fronteiras de cada módulo, facilitando a compreensão para futuras manutenções ou migrações.
- **Monitoramento e Planejamento para Migrações Futuras:** Após o sistema ser lançado, monitore o uso e o desempenho dos módulos para entender melhor os padrões de carga e identificar possíveis gargalos. À medida que o sistema cresce, avalie a possibilidade de migrar módulos específicos para serviços independentes, conforme as necessidades de escalabilidade aumentarem. Essa abordagem modularizada torna a transição para microsserviços mais controlada, pois a divisão de domínios já está estruturada e documentada.

Uma das principais vantagens da arquitetura monolítica modular é que ela facilita a futura migração para uma arquitetura de microsserviços. À medida que os domínios do sistema tornam-se mais definidos e independentes, é possível decompor gradualmente o sistema monolítico em microsserviços. Esse processo pode começar com a eliminação dos acoplamentos entre módulos, seguido pela criação de bancos de dados independentes para cada módulo, e finalmente, a separação completa do módulo do sistema monolítico para ser implantado como um serviço isolado. Esse método de migração permite enfrentar os desafios dos microsserviços de maneira mais controlada e incremental, o que é especialmente benéfico para equipes com pouca experiência nessa arquitetura.

Portanto, a arquitetura monolítica modular é indicada para projetos novos que ainda não possuem um domínio bem definido, que precisam ser desenvolvidos rapidamente para validar o produto no mercado, e que têm a perspectiva de crescer significativamente em número de usuários. Além disso, é uma escolha ideal para projetos complexos que enfrentam limitações de recursos financeiros e de pessoal, oferecendo um caminho viável para evoluir para uma arquitetura mais distribuída à medida que o sistema e as necessidades do negócio se expandem.

4.2.4 Microsserviço

Newman (2022), define que a arquitetura de microsserviços é um tipo de arquitetura orientada a serviços, com uma definição bem clara sobre como as fronteiras dos serviços devem ser traçadas, e para qual a possibilidade de implementações independentes é fundamental, sendo uma arquitetura independente de tecnologia.

A arquitetura de microsserviços se caracteriza por dividir um sistema em vários serviços independentes, cada um focado em um domínio de negócio específico, como relatórios, pagamentos, usuários e autenticação. Esses serviços operam de forma isolada, permitindo que um serviço possa ser escalado ou modificado sem impactar os demais. Por exemplo, se o serviço de usuário estiver sobrecarregado, é possível escalar apenas esse serviço sem afetar o desempenho dos outros.

Essa abordagem também facilita a manutenção e o desenvolvimento, pois cada serviço pode ser desenvolvido e atualizado de maneira independente, promovendo entregas contínuas. Além disso, os microsserviços permitem a utilização de diferentes tecnologias para cada serviço, o que possibilita escolher a melhor ferramenta para resolver problemas específicos.

Apesar dessas vantagens, a arquitetura de microsserviços apresenta desafios significativos. Um entendimento profundo do domínio de negócio é essencial para evitar acoplamentos indesejados entre os serviços, o que poderia comprometer a independência e aumentar a complexidade do sistema.

A comunicação entre microsserviços também deve ser cuidadosamente planejada para manter o baixo acoplamento; normalmente, é preferível utilizar comunicação assíncrona via eventos, em vez de interações síncronas diretas, para preservar a autonomia dos serviços.

Os microsserviços são frequentemente descritos como "caixas pretas" que encapsulam funcionalidades específicas, com cada serviço possuindo sua própria tecnologia e banco de dados. A interação entre eles é feita através de interfaces externas bem definidas, garantindo que as mudanças internas de um serviço não afetem os demais. Isso resulta em um sistema altamente coeso e com fronteiras de serviço estáveis, reduzindo o acoplamento e permitindo uma maior flexibilidade no desenvolvimento e na manutenção.

Entretanto, a implementação de uma arquitetura de microsserviços requer uma equipe grande e bem estruturada, recursos financeiros substanciais, e um alto grau de conhecimento técnico. Cada serviço é tratado como um projeto separado, o que implica lidar com questões de segurança, monitoração, infraestrutura, e comunicação de forma individualizada para cada serviço.

Além disso, a orquestração dos microsserviços, ou seja, a coordenação das interações entre os serviços para garantir que funcionem de maneira coesa, é um desafio significativo. A separação dos bancos de dados também pode criar dependências complexas, onde alterações em um banco podem impactar outros serviços, exigindo uma abordagem cuidadosa nos testes e na comunicação entre os serviços para assegurar a funcionalidade correta de todo o sistema.

Para implementar uma arquitetura de microsserviços, é fundamental seguir um conjunto de etapas bem definidas, desde o planejamento inicial até a implantação final em produção. Cada etapa é crítica para garantir que os microsserviços sejam estruturados de forma a maximizar a independência, a escalabilidade e a facilidade de manutenção. O passo a passo a seguir detalha como realizar essa implementação:

- **Definição dos Serviços e Identificação de Domínios:** O primeiro passo para construir um sistema de microsserviços é dividir o sistema em domínios de negócio distintos. Cada microsserviço deve ser projetado em torno de uma funcionalidade central (como usuários, pagamentos ou relatórios), mantendo um escopo bem específico e sem sobreposição de responsabilidades. Essa etapa envolve uma análise aprofundada do domínio, conhecida como domain-driven design, para assegurar que cada serviço represente um "*bounded context*" ou contexto delimitado dentro do sistema.
- **Estabelecimento de Fronteiras de Serviço Estáveis:** Uma vez que os domínios estão definidos, é essencial determinar as fronteiras estáveis entre os serviços. Essas fronteiras definem onde um serviço começa e termina, bem como as interfaces que ele expõe. Esse isolamento garante que mudanças internas de um serviço não afetem outros, promovendo a independência. As interfaces são cuidadosamente projetadas para que cada serviço seja capaz de se comunicar de forma segura e eficiente com os demais sem criar dependências fortes.
- **Escolha da Tecnologia Adequada para Cada Serviço:** Uma das principais vantagens dos microsserviços é a flexibilidade para usar diferentes tecnologias para cada serviço, o que permite que as equipes escolham as melhores ferramentas e linguagens para resolver os problemas específicos de cada contexto. Por exemplo, um serviço que exige alta performance em cálculos pode ser escrito em uma linguagem de baixo nível, enquanto outro focado em processamento de dados pode utilizar uma linguagem de alto nível e mais orientada para análises.
- **Definição de Estratégias de Transferência de Dados e Comunicação:** A comunicação entre microsserviços deve ser desenhada para minimizar o acoplamento. A escolha entre comunicação síncrona e assíncrona depende das necessidades do sistema e da independência dos serviços. Em geral, opta-se por comunicação assíncrona para reduzir a dependência direta entre serviços, permitindo que cada serviço funcione de forma independente e desacoplada de respostas em tempo real.
- **Orquestração e Coordenação de Microsserviços:** A orquestração envolve a coordenação das interações entre os serviços, assegurando que eles trabalhem de forma coesa. Existem duas abordagens principais: orquestração centralizada, onde um serviço central coordena as operações e coreografia, onde cada serviço age de forma autônoma com base em eventos emitidos por outros serviços. A escolha entre elas depende do grau de complexidade e interdependência dos serviços.
- **Implementação de Testes de Integração e entre Serviços:** Cada microsserviço é testado individualmente, mas testes de integração são essenciais para garantir que as interações entre os serviços ocorram corretamente. Esses testes simulam fluxos de trabalho reais, validando a comunicação entre os serviços, a consistência de dados e a resposta do sistema a cenários de falhas.

Ferramentas de teste como Postman, JUnit e Mock Server são amplamente usadas para automatizar esses testes e garantir uma integração contínua.

- **Configuração de Monitoramento e Observabilidade:** Monitoramento e observabilidade são fundamentais para gerenciar microsserviços em produção. Cada serviço deve ter métricas e logs próprios para rastrear sua saúde e desempenho. A observabilidade pode incluir tracing distribuído para rastrear fluxos de requisição entre serviços, além de dashboards de monitoramento para analisar métricas em tempo real.
- **Automação de Desdobramento e Deploy em Produção:** A automação do deploy, por meio de CI/CD (Integração Contínua e Entrega Contínua), facilita a liberação de novas versões dos microsserviços com segurança e eficiência. Ferramentas como Docker e Kubernetes são amplamente utilizadas para criar e gerenciar contêineres, permitindo a escalabilidade e a atualização dos serviços de forma individual, sem interromper o sistema como um todo.
- **Gerenciamento de Segurança e Autenticação entre Serviços:** Em uma arquitetura de microsserviços, cada serviço precisa ser seguro e confiável. Implementações de autenticação (como OAuth ou JWT) e controle de acesso são cruciais para proteger os dados e as comunicações. Certifique-se de que cada serviço só consiga acessar informações que realmente precisa, aplicando o princípio do menor privilégio.
- **Preparação para Escalabilidade e Manutenção Contínua:** A última etapa antes da produção é a configuração de escalabilidade. Ferramentas de auto-escalonamento, como o *Kubernetes Horizontal Pod Autoscaler*, podem ser configuradas para aumentar ou diminuir os recursos de cada microsserviço conforme a demanda.

Dessa forma, a escolha por uma arquitetura de microsserviços deve ser bem ponderada, considerando a complexidade e os recursos disponíveis, além do grau de conhecimento sobre o domínio e os desafios técnicos envolvidos. É uma solução poderosa, mas que demanda uma organização robusta e bem estruturada para ser bem-sucedida.

4.3 Boas práticas de desenvolvimento

4.3.1 Tecnologias

4.3.1.1 Linguagem de programação e framework

Ao iniciar um projeto de software, a escolha da linguagem de programação e do framework a ser utilizado é uma decisão crítica, mas que deve ser guiada, em grande parte, pelo conhecimento prévio da equipe. Em projetos de menor escala ou com recursos limitados, é muito mais produtivo optar por tecnologias com as quais os desenvolvedores já estão familiarizados. Essa abordagem minimiza o tempo de aprendizagem, evita erros comuns em novas linguagens e frameworks e permite que o time se concentre em resolver problemas de negócios, em vez de enfrentar barreiras técnicas de uma nova stack.

Além disso, o uso de tecnologias já dominadas facilita a manutenção e o suporte do sistema ao longo do tempo, uma vez que os desenvolvedores possuem um nível de expertise maior com essas ferramentas.

No entanto, em projetos de maior escala, com recursos robustos e arquitetura baseada em microserviços, o cenário pode mudar. Em tais casos, cada equipe pode ser responsável por um serviço específico, o que abre a possibilidade de utilizar diferentes linguagens de programação e frameworks para resolver problemas específicos.

A arquitetura de microserviços, por sua natureza, permite que cada serviço seja desenvolvido de maneira independente, com sua própria tecnologia, banco de dados e ambiente de execução. Isso significa que, dependendo do problema a ser resolvido por determinado microserviço, é possível escolher a linguagem de programação mais adequada, seja por questões de performance, facilidade de desenvolvimento ou integração com outras tecnologias.

Por exemplo, um serviço que lida com grande volume de dados em tempo real pode se beneficiar de linguagens como Go ou Rust, que oferecem alta performance e eficiência na utilização de recursos. Já um microserviço focado em operações CRUD tradicionais pode ser construído em *TypeScript* ou Java, aproveitando o amplo ecossistema de bibliotecas e ferramentas disponíveis para essas linguagens. Esse nível de flexibilidade permite que cada equipe escolha a melhor tecnologia para o seu contexto, maximizando a eficiência do desenvolvimento e a qualidade das soluções entregues.

4.3.1.2 Banco de dados SQL

Os bancos de dados SQL (*Structured Query Language*) são amplamente utilizados para armazenar, gerenciar e acessar dados estruturados, sendo uma escolha comum em uma grande variedade de aplicações de software. De acordo com a Microsoft um banco de dados SQL que é também conhecido como bancos de dados relacionais são sistemas que armazenam coleções de tabelas e organizam conjuntos estruturados de dados em um formato de colunas e linhas tabulares, semelhante ao de uma planilha. A principal característica dos bancos SQL é a aderência a um modelo de dados relacional, o que os torna ideais para cenários onde a integridade, consistência e interrelacionamento dos dados são cruciais.

Os bancos de dados SQL oferecem várias vantagens que os tornam amplamente utilizados em diversas aplicações. Como o suporte nativo para transações ACID, que garantem a confiabilidade e consistência dos dados, especialmente em sistemas críticos que não podem tolerar falhas ou inconsistências. As relações entre entidades são bem definidas, o que melhora a integridade e a segurança dos dados. Ferramentas maduras, pois os bancos SQL existem há décadas, o que significa que há um vasto ecossistema de ferramentas, frameworks e suporte, facilitando a integração com sistemas existentes e o desenvolvimento de novas soluções. Além de que, bancos de dados SQL possuem mecanismos robustos de controle de acesso e permissões, permitindo que os administradores configurem usuários com diferentes níveis de privilégios sobre as tabelas e os dados.

Existem várias opções populares de bancos de dados SQL, cada uma com suas próprias características e especializações. Iniciando com o MySQL, um dos bancos de dados SQL mais amplamente utilizados no mundo, popular por sua simplicidade, desempenho e forte comunidade de suporte. PostgreSQL, conhecido por ser um banco de dados relacional avançado, o PostgreSQL oferece uma rica gama de recursos, como suporte a tipos de dados complexos, transações ACID robustas e extensibilidade. É altamente utilizado em sistemas que exigem maior flexibilidade e funcionalidades avançadas. Oracle Database, considerado um dos bancos de dados mais robustos e seguros do mercado, é utilizado em grandes empresas e sistemas críticos que demandam alta disponibilidade, segurança e performance.

Em suma, os bancos de dados SQL continuam sendo uma escolha sólida para uma vasta gama de aplicações, especialmente quando a consistência e a integridade dos dados são prioridades. Com diversas opções disponíveis e um ecossistema maduro, são frequentemente a escolha padrão para projetos que demandam alta confiabilidade e gestão eficiente de dados estruturados.

4.3.1.3 Banco de dados NoSQL

Os bancos de dados NoSQL (Not Only SQL) oferecem uma abordagem alternativa ao modelo relacional tradicional dos bancos SQL, como a Microsoft apresenta, eles se destacam pelo fato de que podem processar grandes volumes de dados não estruturados e em constante mudança de maneiras diferentes de um banco de dados relacional. Sendo amplamente utilizados em cenários que envolvem grandes volumes de dados, alta escalabilidade e requisitos de flexibilidade.

Ao contrário dos bancos de dados relacionais, que estruturam os dados em tabelas e utilizam o SQL para consultas, os bancos NoSQL são projetados para lidar com dados não estruturados ou semiestruturados, oferecendo maior flexibilidade na modelagem e armazenagem de informações. Eles permitem um desempenho otimizado em sistemas distribuídos, sendo a escolha ideal para grandes aplicações web, sistemas de big data e plataformas que exigem escalabilidade horizontal.

É ideal para aplicações que precisam armazenar e gerenciar dados que não têm um formato rígido, como documentos JSON, logs de eventos, dados de redes sociais e informações multimídia.

Ao contrário dos bancos SQL, que geralmente escalam verticalmente (aumentando a capacidade do servidor), os bancos NoSQL foram projetados para escalar horizontalmente, permitindo que o sistema adicione novos servidores à medida que a demanda cresce. Isso é útil em aplicações que lidam com milhões de usuários ou grandes quantidades de dados.

São otimizados para gravações e leituras rápidas, mesmo com grandes volumes de dados. Isso os torna adequados para sistemas que exigem baixa latência e alta capacidade de resposta, como sistemas de recomendações, motores de busca e análises em tempo real.

Há uma variedade de bancos de dados NoSQL populares, cada um com suas características e usos específicos. Como o MongoDB, um dos bancos de dados NoSQL mais populares, MongoDB é um banco de dados orientado a documentos que armazena dados no formato BSON (uma extensão binária de JSON).

É amplamente utilizado para aplicativos web, análise de dados e sistemas que lidam com dados semiestruturados. Cassandra, desenvolvido pela Apache, Cassandra é um banco de dados NoSQL altamente escalável e distribuído, conhecido por sua capacidade de lidar com grandes volumes de dados e alta disponibilidade sem um único ponto de falha. É usado por empresas como Netflix e Twitter. Redis, um banco de dados em memória orientado a chave-valor, Redis é conhecido por seu desempenho extremamente rápido, sendo amplamente utilizado em aplicações que exigem cachê, filas de mensagens e sessões de usuário.

Em suma, os bancos de dados NoSQL oferecem uma solução poderosa para sistemas que precisam lidar com grandes volumes de dados não estruturados ou que exigem alta flexibilidade e escalabilidade. Com uma ampla variedade de opções e aplicações, são uma escolha cada vez mais popular para sistemas modernos que exigem desempenho, adaptabilidade e eficiência em ambientes distribuídos.

4.3.1.4 ORM

O *Object-Relational Mapping* (ORM) é uma técnica de programação que permite que desenvolvedores interajam com bancos de dados relacionais utilizando objetos do paradigma da programação orientada a objetos (POO), em vez de escrever diretamente consultas SQL. O ORM age como uma ponte entre o código da aplicação e o banco de dados, facilitando o desenvolvimento de sistemas, pois permite que os dados sejam manipulados como objetos do código, sem a necessidade de lidar diretamente com as tabelas e consultas SQL.

O ORM permite que desenvolvedores trabalhem com dados sem precisar escrever consultas SQL diretamente. As operações com o banco de dados são feitas utilizando objetos e métodos, o que simplifica o desenvolvimento e reduz a possibilidade de erros em consultas SQL complexas.

Ao automatizar a geração de consultas e a manipulação de dados, o ORM reduz o tempo necessário para desenvolver a camada de persistência, permitindo que os desenvolvedores foquem mais na lógica de negócios e no desenvolvimento de funcionalidades da aplicação.

O ORM permite que as mudanças na estrutura do banco de dados sejam feitas de forma mais simples, propagando automaticamente essas alterações para o restante da aplicação sem a necessidade de ajustar manualmente cada consulta SQL. Isso facilita a manutenção a longo prazo.

Como o ORM abstrai a interação direta com o banco de dados, ele torna a aplicação mais portátil entre diferentes sistemas de gerenciamento de banco de dados. Isso significa que a aplicação pode ser movida de um banco de dados para outro (por exemplo, de MySQL para PostgreSQL) com mudanças mínimas no código.

Existem diversas ferramentas de ORM populares, cada uma adaptada para diferentes linguagens de programação e tipos de banco de dados. Alguns dos ORM's mais usados incluem o Prisma que é um ORM moderno para Node.js e TypeScript, utilizado principalmente com bancos de dados SQL como PostgreSQL, MySQL e SQLite. Ele se destaca pela facilidade de uso, geração automática de modelos a partir de esquemas de banco de dados e suporte a TypeScript. Prisma é especialmente útil em aplicações que precisam de uma forte tipagem estática e que desejam manter a coerência entre o código e o banco de dados. Mongoose, é um ORM muito popular para MongoDB no ambiente Node.js. Ele facilita o mapeamento de documentos MongoDB para objetos JavaScript, permitindo que desenvolvedores trabalhem com dados no MongoDB de forma mais estruturada. Mongoose é frequentemente utilizado em aplicações web que requerem um banco de dados NoSQL, como plataformas de redes sociais e sistemas de gerenciamento de conteúdo. Além de outros como o Hibernate para o ecossistema Java, o Doctrine para PHP e o Entity Framework para C# e .NET.

Em suma, os ORM's são ferramentas essenciais para desenvolver sistemas modernos que utilizam bancos de dados relacionais, pois eles simplificam a interação com os dados, aumentam a produtividade, melhoram a segurança e facilitam a manutenção do código. Embora o uso de ORM's traga diversas vantagens, é importante avaliar se as necessidades do projeto justificam a abstração e o impacto que ela pode ter no desempenho em cenários de grande escala, pois podem introduzir sobrecarga em alguns cenários, como em altos volumes de transações e consultas complexas.

4.3.1.5 Docker

O Docker é uma plataforma de código aberto que automatiza a implantação de aplicações dentro de contêineres, permitindo que desenvolvedores empacotem uma aplicação com todas as suas dependências, como bibliotecas, arquivos de configuração e binários, em um ambiente isolado. A RedHat afirma que com o Docker é possível lidar com os containers como se fossem máquinas virtuais modulares e extremamente lightweight. Além disso, os containers oferecem maior flexibilidade para você criar, implantar, copiar e migrar um container de um ambiente para outro. A tecnologia de contêineres oferece uma maneira leve e eficiente de virtualizar ambientes de software, possibilitando que aplicações sejam executadas de maneira consistente em qualquer infraestrutura, seja em ambientes de desenvolvimento, testes ou produção.

Docker é uma ferramenta que usa contêineres para garantir que uma aplicação funcione uniformemente em diferentes ambientes, eliminando problemas de compatibilidade entre o ambiente de desenvolvimento e o de produção. Diferentemente de máquinas virtuais (VMs), onde o sistema operacional completo é replicado, os containers Docker compartilham o kernel do sistema operacional host, o que torna a execução de múltiplos contêineres simultâneos muito mais leve e eficiente.

Essencialmente, Docker encapsula o código da aplicação e suas dependências em uma imagem imutável, que pode ser executada em qualquer servidor compatível com Docker. Isso facilita o transporte de software por diferentes fases do ciclo de desenvolvimento, sem que sejam necessárias adaptações ao ambiente.

Com Docker, uma aplicação pode ser empacotada em um contêiner que contém tudo o que é necessário para sua execução. Isso significa que ela pode ser transferida entre diferentes máquinas e servidores (inclusive em diferentes provedores de nuvem), garantindo que sempre funcione da mesma forma, independentemente do ambiente. Esse atributo elimina o clássico problema de “funciona na minha máquina” enfrentado por desenvolvedores.

Cada container Docker é executado de forma isolada do sistema host e de outros contêineres, permitindo que múltiplas aplicações sejam executadas em um único servidor sem que interfiram umas nas outras. Isso também facilita a execução de múltiplas versões da mesma aplicação ou de serviços diferentes em paralelo.

Como os contêineres compartilham o kernel do sistema operacional host, eles são muito mais leves do que as máquinas virtuais tradicionais, que precisam replicar todo um sistema operacional. Isso permite um uso mais eficiente dos recursos de hardware, resultando em menor custo de infraestrutura.

Docker facilita o escalonamento horizontal de aplicações. Em vez de ter que provisionar uma nova VM ou instalar manualmente um novo ambiente, os desenvolvedores podem simplesmente criar mais instâncias de contêineres para lidar com aumentos de demanda. Ferramentas como Docker Swarm e Kubernetes permitem a orquestração e o gerenciamento automático de múltiplos contêineres, tornando o processo de escalabilidade ainda mais eficiente.

Com Docker, desenvolvedores podem criar ambientes de desenvolvimento idênticos aos de produção, eliminando inconsistências e facilitando a reprodução de erros. Além disso, a automação da construção, teste e implantação de contêineres reduz o tempo necessário para passar de uma fase do ciclo de desenvolvimento para outra, o que acelera o lançamento de novas funcionalidades.

Docker é amplamente utilizado em pipelines de CI/CD, pois facilita a automação da construção e teste de aplicações em diferentes ambientes. Os contêineres podem ser rapidamente criados, testados e implantados, o que acelera o ciclo de feedback e garante que novas funcionalidades ou correções de bugs sejam lançadas com rapidez e confiabilidade.

Docker revolucionou a forma como desenvolvedores e empresas lidam com a criação, teste e implantação de aplicações. Sua capacidade de isolar ambientes e garantir a consistência entre diferentes fases do ciclo de vida de desenvolvimento o torna uma ferramenta essencial para projetos modernos. A portabilidade, a eficiência de recursos e o suporte a escalabilidade fazem do Docker uma escolha ideal para o desenvolvimento de aplicações em ambientes de nuvem, arquiteturas de microsserviços e integração contínua.

4.3.1.6 Git e Gitflow

Git é um sistema de controle de versão distribuído amplamente utilizado no desenvolvimento de software para gerenciar o histórico de mudanças e facilitar o trabalho colaborativo entre equipes. Ele permite que desenvolvedores acompanhem modificações no código-fonte ao longo do tempo, mantenham diferentes versões de um projeto e colaborem de forma eficiente, mesmo em projetos de grande escala.

Isso garante que as equipes possam trabalhar de maneira independente e offline, realizando *commits* locais (registros de mudanças), e sincronizar com o repositório principal (também conhecido como *origin*) quando estiverem prontas para compartilhar suas alterações. Além disso, Git é extremamente eficiente para gerenciar projetos grandes com múltiplos colaboradores e integração contínua.

Gitflow é um modelo de fluxo de trabalho que organiza a maneira como os desenvolvedores utilizam branches no Git, tornando o desenvolvimento colaborativo mais previsível e eficiente. Esse modelo é particularmente útil para equipes que trabalham em grandes projetos, onde múltiplos desenvolvedores estão envolvidos e há a necessidade de controlar diferentes estágios de desenvolvimento, como novas funcionalidades, correções de bugs e versões de produção.

O Gitflow define um conjunto de práticas e convenções que orientam o uso de branches para diferentes tipos de trabalho, divididos em:

- **Master (ou Main):** Este branch contém sempre o código que está em produção ou pronto para ser lançado. É a versão estável do projeto.
- **Develop:** O branch develop é onde o desenvolvimento ativo acontece. As novas funcionalidades e correções são integradas aqui antes de serem promovidas para o branch master.
- **Feature Branches:** São criados a partir do branch develop e usados para desenvolver novas funcionalidades. Quando o trabalho em uma feature está completo, ele é integrado de volta ao develop por meio de um merge ou pull request.
- **Release Branches:** Antes de uma nova versão ser lançada, é comum criar um branch de release a partir de develop, onde são feitas correções finais e ajustes. Depois de concluído, esse branch é integrado ao master e também ao develop, garantindo que as correções estejam presentes em ambos os branches.
- **Hotfix Branches:** Quando há um problema crítico na produção, um hotfix branch é criado diretamente do master. A correção é feita e integrada tanto ao master quanto ao develop, garantindo que o problema seja resolvido em produção sem interromper o desenvolvimento em andamento.

Tanto o Git quanto o Gitflow são ferramentas essenciais para o controle de versão e o gerenciamento de projetos de software. Git oferece uma maneira flexível e eficiente de gerenciar o código e as alterações feitas ao longo do tempo, enquanto o Gitflow oferece um modelo de trabalho organizado que torna o desenvolvimento colaborativo mais estruturado e previsível. Juntos, eles são fundamentais para garantir que equipes de desenvolvimento possam colaborar de maneira eficiente, mantendo a qualidade do software e facilitando a entrega contínua de novas funcionalidades e correções.

4.3.2 SOLID

Os princípios SOLID são um conjunto de diretrizes de design de soft-ware que visam criar sistemas mais flexíveis, manuteníveis e expansíveis. Proposta por Martin (2019), os princípios SOLID são amplamente adotados na engenharia de software para melhorar a qualidade e a estrutura do código, garantindo que ele seja mais fácil de entender, modificar e testar. O acrônimo SOLID refere-se a cinco princípios fundamentais que, quando aplicados corretamente, ajudam a evitar problemas comuns, como acoplamento excessivo, violação de encapsulamento e código frágil. São esses *Single Responsibility Principle* (SRP) — Princípio da Responsabilidade Única, *Open/Closed Principle* (OCP) — Princípio Aberto/Fechado, *Liskov Substitution Principle* (LSP) — Princípio da Substituição de Liskov, *Interface Segregation Principle* (ISP) — Princípio da Segregação de Interface e *Dependency Inversion Principle* (DIP) — Princípio da Inversão de Dependência.

O **Princípio da Responsabilidade Única** afirma que uma classe deve ter apenas uma única razão para mudar, ou seja, ela deve ter uma única responsabilidade ou propósito. Quando uma classe assume múltiplas responsabilidades, torna-se difícil mantê-la, pois qualquer mudança em uma responsabilidade pode afetar outras partes do código.

Exemplo: Em vez de ter uma classe Livro que trata tanto de informações do livro quanto da persistência de dados em um banco de dados, podemos separar essas responsabilidades em duas classes distintas: Livro, que armazena os dados do livro, e LivroRepository, que lida com a persistência.

Vantagens: Facilita a manutenção e a modificação do código e evita o acoplamento excessivo de responsabilidades.

O Princípio Aberto/Fechado estabelece que módulos, classes ou funções devem estar abertos para extensão, mas fechados para modificação. Isso significa que é possível adicionar novas funcionalidades sem alterar o código existente, reduzindo o risco de introduzir erros em funcionalidades já desenvolvidas.

Exemplo: Em vez de modificar diretamente um a classe de cálculo de impostos para lidar com um novo tipo de imposto, pode-se estender a classe base através de herança ou implementar uma interface para adicionar a nova lógica de cálculo sem alterar o comportamento original.

Vantagens: Promove a extensão de funcionalidades sem comprometer o código existente e minimiza a probabilidade de introduzir novos bugs.

O **Princípio da Substituição de Liskov** define que objetos de uma classe derivada devem ser substituíveis por objetos da classe base sem alterar a corretude do programa. Em outras palavras, uma subclasse deve ser capaz de substituir sua classe base sem quebrar o comportamento esperado.

Exemplo: Se temos uma classe **Animal** com um método **andar()**, todas as subclasses como **Cachorro** ou **Pássaro** devem ser capazes de sobrescrever esse método sem alterar seu comportamento geral, garantindo que o código que utiliza objetos **Animal** não precise se preocupar com as classes específicas. Vantagens: Garante que o código seja modular e que as subclasses possam ser usadas de forma segura e previsível e evitar comportamentos inesperados ao substituir classes.

O **Princípio da Segregação de Interface** afirma que os clientes não devem ser forçados a depender de interfaces que não utilizam. Isso significa que, em vez de criar interfaces grandes e genéricas, é preferível criar interfaces menores e mais específicas, atendendo a necessidades mais focadas.

Exemplo: Se uma interface **Trabalhador** define os métodos **trabalhar()** e **descansar()**, mas um robô só pode trabalhar e não precisa descansar, seria melhor ter duas interfaces menores: **Trabalhador** com o método **trabalhar()** e **Descansar** com o método **descansar()**, permitindo que cada classe implemente apenas o que for necessário.

Vantagens: Evita que classes dependam de métodos que não usam, tornando o código mais coeso e facilitando a manutenção e a implementação de novas funcionalidades.

O **Princípio da Inversão de Dependência** afirma que módulos de alto nível não devem depender de módulos de baixo nível; ambos devem depender de abstrações. Além disso, as abstrações não devem depender de detalhes; os detalhes devem depender das abstrações. Isso promove o desacoplamento entre componentes, tornando o código mais modular e flexível.

Exemplo: Em vez de uma classe Pedido instanciar diretamente um objeto EmailService para enviar notificações, é melhor depender de uma abstração, como uma interface Notificacao, que pode ser implementada tanto por EmailService quanto por SMSservice. Assim, Pedido não depende de detalhes concretos de como a notificação é enviada.

Vantagens: Aumenta a flexibilidade e modularidade do código e facilita a substituição de dependências sem alterar o código do módulo de alto nível.

Os princípios SOLID fornecem um conjunto de boas práticas que ajudam a criar software robusto, flexível e fácil de manter. Aplicando esses princípios, os desenvolvedores podem reduzir o acoplamento entre classes, melhorar a coesão e tornar o código mais legível e modular. Embora possam parecer complexos inicialmente, seguir esses princípios contribui significativamente para o sucesso a longo prazo de projetos de software, garantindo que o código possa evoluir e se adaptar às mudanças de requisitos de maneira eficiente e com menos risco de introduzir bugs ou problemas estruturais.

4.3.3 Código Limpoo

Clean Code, ou código limpo, refere-se a um conjunto de práticas e princípios que visam a criação de software legível, fácil de entender e de manter. Popularizado por Martin (2009), o conceito se baseia na ideia de que o código deve ser escrito não apenas para que funcione corretamente, mas também para que seja compreendido por outros desenvolvedores (e por nós mesmos no futuro). O código limpo facilita a manutenção, refatoração e evolução do software ao longo do tempo, garantindo que o sistema se mantenha robusto e adaptável às mudanças. E traz alguns fundamentos que iremos abordar posteriormente.

Um dos princípios fundamentais de *Clean Code* é a importância de nomes claros e descritivos para variáveis, funções e classes. **Bons nomes** comunicam a intenção do código, tornando-o mais legível sem a necessidade de comentários excessivos.

Exemplo: Ao invés de nomear uma variável como **x**, podemos usar **numeroDePedidos** para que fique claro o que ela representa.

Benefícios: Facilita o entendimento do código por parte de outros desenvolvedores e reduz a necessidade de documentação adicional para explicar o que o código faz.

Funções devem ser curtas e ter uma **única responsabilidade**. Isso significa que uma função deve realizar apenas uma tarefa bem definida, e o nome da função deve refletir isso de forma precisa. Exemplo: Em vez de uma função grande como **processarPedidosEEnviarEmails** é melhor dividir em duas funções menores: `processarPedidos()` e `enviarEmails()`. Assim, cada uma foca em sua própria responsabilidade.

Benefícios: Aumenta a legibilidade do código, facilitando a depuração e manutenção e facilita a reutilização de funções em outros contextos.

Comentários podem ser úteis em alguns casos, mas devem ser evitados quando o código é claro o suficiente para ser compreendido por si só. Comentários desnecessários podem se tornar obsoletos ou enganosos com o tempo, enquanto o código permanece fiel à sua intenção original.

Exemplo: Em vez de usar um comentário para explicar o que um bloco de código faz, refatore o código para que a intenção fique clara através de nomes descritivos e estrutura adequada.

Benefícios: Evita a confusão que pode surgir com comentários desatualizados ou errôneos e incentiva a escrita de código autoexplicativo.

O princípio DRY (Don't Repeat Yourself) sugere que devemos evitar a duplicação de código. Sempre que o mesmo comportamento ou lógica é repetido em diferentes partes do código, isso pode levar a inconsistências e maior dificuldade de manutenção.

Exemplo: Se uma mesma lógica de cálculo é usada em várias partes do sistema, em vez de duplicar o código, crie uma função reutilizável.

Benefícios: Reduz a quantidade de código redundante, facilitando a manutenção e garante que alterações em uma funcionalidade sejam feitas em um único lugar.

O tratamento de erros deve ser feito de maneira previsível e clara. Exceções e retornos de erro devem ser tratados com cuidado, fornecendo informações úteis para entender o que deu errado e facilitando a depuração.

Exemplo: Em vez de usar try-catch genérico ou silenciar erros, capture exceções específicas e forneça mensagens claras sobre o que aconteceu e como corrigir.

Benefícios: Facilita a identificação e correção de problemas e melhora a confiabilidade do software, garantindo que o código falhe de forma controlada e previsível.

Um código limpo deve ser fácil de testar. Práticas como ***Test-Driven Development (TDD)*** incentivam a escrita de testes automatizados antes mesmo de o código ser implementado, garantindo que ele cumpra os requisitos esperados.

Exemplo: Funções pequenas e com responsabilidade única são mais fáceis de testar, pois seus comportamentos são previsíveis e podem ser verificados isoladamente.

Benefícios: Aumenta a confiança na qualidade do código e facilita a refatoração, garantindo que mudanças não quebrem funcionalidades existentes.

A consistência na **formatação do código** é essencial para manter sua legibilidade. Isso inclui a padronização de indentação, espaçamento e organização dos arquivos. Exemplo: Usar uma convenção de estilo consistente (como a adotada pelo time ou pelo projeto) e ferramentas de formatação automática garante que o código siga padrões estabelecidos.

Benefícios: Facilita a leitura e revisão do código por diferentes desenvolvedores e evita discussões sobre estilo e foco em questões mais importantes do design.

Código morto é qualquer parte do código que não está sendo utilizada ou que não contribui para a funcionalidade do sistema. Mantê-lo no projeto aumenta a complexidade desnecessariamente e pode levar a confusão no futuro.

Exemplo: Se uma função ou classe não está mais sendo usada, é melhor removê-la completamente do código.

Benefícios: Reduz a quantidade de código a ser mantido, tornando o sistema mais enxuto e fácil de entender e diminui o risco de confusão com funcionalidades obsoletas ou redundantes.

Seguir os princípios de Clean Code resulta em software de alta qualidade, mais fácil de manter e evoluir. A clareza, simplicidade e organização do código são essenciais para que ele seja compreendido por toda a equipe de desenvolvimento, independentemente de sua complexidade. Além disso, a aplicação de Clean Code proporciona maior agilidade nas mudanças, reduz o número de bugs e problemas, e facilita a escalabilidade do sistema ao longo do tempo. Um código bem estruturado não só facilita o trabalho presente, mas também garante que o software permaneça sustentável à medida que os requisitos evoluem.

4.3.4 Arquitetura limpa

A Arquitetura Limpa (ou Clean Architecture) é uma abordagem de design de software proposta por Martin (2019), em seu livro Arquitetura Limpa. Seu principal objetivo é criar sistemas modulares, independentes de frameworks e tecnologias, com foco em facilidade de manutenção, testabilidade e evolução a longo prazo. Esse estilo de arquitetura baseia-se em separar as responsabilidades do sistema em camadas, permitindo que as partes internas do software não dependam de detalhes externos, como bancos de dados, interfaces de usuário ou frameworks específicos.

A Arquitetura Limpa é frequentemente representada por um diagrama concêntrico, onde as camadas mais internas contêm as regras de negócio mais

puras e as externas são responsáveis pelos detalhes de implementação, como acesso a dados ou interfaces gráficas. Essa estrutura modularizada promove um forte desacoplamento, facilitando a alteração ou substituição de componentes sem afetar a integridade do sistema.

A Arquitetura Limpa é organizada em camadas concêntricas, onde as dependências sempre fluem de fora para dentro. O núcleo do sistema contém as regras de negócios essenciais e não conhece nada sobre as camadas externas.

No centro da arquitetura estão as **entidades**, que representam as regras de negócio mais fundamentais. Elas são abstrações dos conceitos centrais do domínio da aplicação, independentemente de frameworks, banco de dados ou qualquer tecnologia específica.

Exemplo: Em um sistema de e-commerce, uma entidade pode ser Pedido, com regras relacionadas a como um pedido deve ser gerado, validado ou processado.

A próxima camada são os casos de uso, que representam as operações específicas que o sistema realiza. Aqui residem as regras de aplicação, que utilizam as entidades para orquestrar o comportamento do sistema de acordo com os requisitos dos usuários.

Exemplo: Um caso de uso pode ser ProcessarPagamento, que coordena as regras para verificar o saldo do cliente, calcular o valor total e confirmar a compra.

Mais externamente, temos as interfaces de usuário e interface de entrada, responsáveis por receber os comandos dos usuários ou sistemas externos e convertê-los para o formato utilizado internamente pelos casos de uso. Isso inclui APIs REST, interfaces gráficas e controladores de sistemas.

Exemplo: Um endpoint REST que recebe uma requisição para criar um novo pedido e transforma os dados da requisição em objetos de entidade e caso de uso.

Na camada mais externa, encontramos a infraestrutura, incluindo acesso a dados, frameworks e bibliotecas externas. Aqui estão implementados os repositórios, adaptadores de bancos de dados, serviços de mensageria e APIs de terceiros.

Exemplo: Um repositório que interage com o banco de dados para salvar ou recuperar entidades como Cliente ou Pedido.

A Arquitetura Limpa segue alguns princípios fundamentais que ajudam a organizar o código de forma sustentável e evolutiva:

Independência de frameworks: O sistema não deve depender de bibliotecas ou frameworks externos. Eles podem ser utilizados, mas devem ser tratados como "detalhes", facilmente substituíveis.

Testabilidade: O design deve permitir a fácil criação de testes automatizados, possibilitando o isolamento de regras de negócio para testes unitários eficazes.

Independência de UI: A interface de usuário é considerada um detalhe externo. O sistema deve ser funcional mesmo sem a interface visual.

Independência de banco de dados: A lógica de negócios não deve estar atrelada a uma tecnologia de banco de dados específica. Isso facilita a troca de bancos sem impactar outras áreas do sistema.

Independência de agentes externos: Qualquer dependência de sistemas externos, como APIs de terceiros, deve estar na camada mais externa, protegendo o núcleo do sistema de mudanças.

A Arquitetura Limpa proporciona uma abordagem sólida para o desenvolvimento de software modular e escalável, permitindo a construção de sistemas de longo prazo que podem evoluir sem comprometer a integridade das regras de negócio. Ao desacoplar as responsabilidades e manter as dependências controladas, essa arquitetura facilita a manutenção, a testabilidade e a substituição de componentes tecnológicos, sendo uma escolha ideal para projetos que exigem flexibilidade e robustez.

4.3.5 Test-Driven Development (TDD)

O Test - Driven Development (TDD) é uma metodologia de desenvolvimento de software em que o design e a implementação do código são orientados pelos testes. Essa abordagem tem como objetivo principal garantir a qualidade do software através da escrita de testes automatizados antes da implementação das funcionalidades. Como Newman classifica em seu livro Criando Microsserviços (2022), o TDD se enquadra melhor em testes de unidade, testando uma única chamada de função ou método, sendo de extrema importância por ser testes extremamente rápidos, se forem feitos corretamente. Além disso, segue um ciclo rigoroso de etapas que ajudam a manter o código enxuto, testável e funcional.

O processo de TDD é geralmente descrito em três etapas, conhecidas como Red-Green-Refactor:

- **Red** (Escreva o Teste), o ciclo do TDD começa com a escrita de um teste que reflete um comportamento ou requisito específico que ainda não foi implementado. Esse teste, que geralmente falha inicialmente, define o que o código precisa fazer, orientando o desenvolvedor em relação à funcionalidade que está por vir. Exemplo: Se você deseja implementar uma função de cálculo de soma, você escreve um teste que valida se somar(2, 3) resulta em 5.
- **Green** (Faça o Teste Passar), com o teste escrito e falhando, o próximo passo é escrever o código mínimo necessário para passar no teste. Nessa fase, o foco não é otimizar o código ou pensar em melhorias, mas apenas garantir que a funcionalidade básica passe no teste. Exemplo: Implementase a função somar, que recebe dois números e retorna sua soma. O objetivo é apenas fazer o teste passar, mesmo que o código seja básico.
- **Refactor** (Refatore o Código), com o teste passando, é hora de refatorar o código. A refatoração visa melhorar a estrutura, legibilidade e eficiência do código sem alterar sua funcionalidade. O teste serve como uma garantia de que a refatoração não quebrou o comportamento esperado. Exemplo: Após garantir que o teste passou, você pode refatorar a função de soma para lidar com casos especiais ou melhorar a performance, sempre mantendo o teste como verificador.

Esse ciclo se repete continuamente ao longo do desenvolvimento, ga-

-rantiendo que o código esteja sempre coberto por testes e que novos recursos não quebrem funcionalidades anteriores.

O TDD oferece uma série de benefícios que contribuem diretamente para a qualidade e manutenibilidade do software:

Maior confiança no código: Ao escrever testes antes do código, os desenvolvedores têm uma garantia maior de que o software funcionará conforme o esperado, já que cada funcionalidade é validada automaticamente.

Menor taxa de bugs: Como o desenvolvimento é guiado por testes, muitos erros são encontrados e corrigidos imediatamente, antes de se tornarem problemas maiores em ambientes de produção.

Código mais simples e modular: A prática de escrever apenas o código necessário para passar no teste incentiva os desenvolvedores a produzir soluções mais simples e modulares, evitando a criação de funcionalidades desnecessárias.

Facilidade na refatoração: Com a cobertura de testes adequada, os desenvolvedores podem refatorar o código com confiança, sabendo que os testes garantem que a funcionalidade original não foi afetada.

Documentação viva: Os testes também funcionam como uma espécie de documentação, demonstrando claramente o comportamento esperado do código. Isso facilita a compreensão do sistema, tanto para desenvolvedores novos quanto para os antigos.

O Test-Driven Development (TDD) é uma abordagem poderosa para melhorar a qualidade do software e garantir sua confiabilidade desde o início do desenvolvimento. Ao escrever testes antes do código, os desenvolvedores podem se concentrar em entregar soluções mais simples, modulares e testáveis. Embora o TDD exija uma mudança de mentalidade e possa ter uma curva de aprendizado, seus benefícios a longo prazo compensam, tornando-se uma prática essencial para equipes que buscam criar software sustentável e de alta qualidade.

4.3.6 Domain-Driven Design (DDD)

O Domain-Driven Design (DDD), ou Design Orientado ao Domínio, é uma abordagem para o desenvolvimento de software complexos, que foca na criação de soluções que estejam diretamente alinhadas com o domínio de negócios em questão. Proposto por Evans em seu livro *Domain-Driven Design* (2020), o DDD tem como princípio central a colaboração estreita entre especialistas de domínio e desenvolvedores, buscando garantir que o software representa com precisão as regras e processos do negócio.

O "Domínio" refere-se ao problema ou à área de conhecimento que o software busca resolver. Em vez de focar em tecnologias ou implementações de baixo nível, o DDD direciona o desenvolvimento para que seja centrado nos conceitos e nas regras do domínio de negócio. Por exemplo, em um sistema de gerenciamento de estoque, o domínio é o próprio gerenciamento de inventário, enquanto as tecnologias utilizadas são secundárias em relação às

regras de negócio.

Um dos pilares do DDD é a linguagem ubíqua, uma linguagem comum e compartilhada entre todos os envolvidos no projeto (desenvolvedores, especialistas de negócio, *stakeholders*). O objetivo da linguagem ubíqua é eliminar ambiguidades, permitindo que todos discutam os mesmos conceitos utilizando termos consistentes. Ao utilizar essa linguagem, a comunicação é facilitada, o que garante que o software reflita com precisão as necessidades do negócio. Por exemplo, se o domínio envolve processamento de pedidos, termos como "cliente", "produto", "pedido", e "estoque" devem ser utilizados consistentemente por todos, tanto na comunicação quanto no código, para evitar discrepâncias ou mal-entendidos.

Um conceito chave no DDD é o Bounded Context (contexto delimitado). À medida que o sistema cresce, é comum que diferentes partes do negócio utilizem o mesmo termo de maneiras diferentes. Para evitar conflitos e ambiguidades, o DDD sugere a delimitação clara de contextos onde a linguagem e as regras de negócio são aplicadas. Cada contexto delimitado é uma espécie de fronteira onde os termos e as regras têm significado claro e específico.

Por exemplo, em um sistema de e-commerce, "pedido" pode ter um significado diferente no contexto de compras em relação ao contexto de logística. Ao definir essas fronteiras, é possível garantir que diferentes equipes possam trabalhar com clareza e sem conflitos entre diferentes partes do sistema.

No DDD, grande parte do esforço de desenvolvimento é dedicado à modelagem do domínio. O objetivo é representar as Entidades, Objetos de Valor e Agregados que compõem o negócio de forma que o software reflita de maneira fiel a realidade do domínio.

Entidades são objetos que têm identidade própria e persistem ao longo do tempo, mesmo que seus atributos mudem. Por exemplo, um "Cliente" é uma Entidade, pois ele pode mudar de endereço, mas continua sendo o mesmo cliente no sistema.

Objetos de Valor são objetos que não possuem identidade própria e são definidos inteiramente por seus atributos. Por exemplo, um "Endereço" pode ser considerado um Objeto de Valor, pois é definido pelo conjunto de atributos (rua, número, cidade, etc.), e a alteração de qualquer um desses atributos resultaria em um novo endereço.

Agregado é um agrupamento lógico de Entidades e Objetos de Valor que são tratados como uma única unidade. O Agregado possui uma Entidade principal chamada "Raiz do Agregado", que é a única responsável por garantir a integridade dos dados dentro desse grupo. Por exemplo, uma "Ordem de Compra" pode ser um Agregado que inclui Entidades como "Itens de Compra" e "Pagamentos".

Nem toda funcionalidade pode ser diretamente associada a uma Entidade ou Objeto de Valor. Para essas situações, o DDD sugere o uso de Serviços de Domínio, que encapsulam operações relacionadas ao Domínio e

que não pertencem a uma única Entidade. Esses serviços são focados em ações que afetam o Domínio, e não em dados. Por exemplo, em um sistema de pagamento, o cálculo de taxas pode ser um serviço de Domínio, já que não está diretamente ligado a uma única Entidade, mas afeta várias partes do sistema.

Embora o DDD foque principalmente no design do **Domínio** e nas regras de negócio, é inevitável que o sistema precise interagir com bancos de dados e outros mecanismos de persistência. Repositórios são utilizados para abstrair a lógica de armazenamento, garantindo que as entidades possam ser salvas e recuperadas sem expor detalhes de implementação.

O *Domain-Driven Design* (DDD) é uma abordagem poderosa para lidar com a complexidade de sistemas de software, especialmente aqueles que exigem um alinhamento profundo com o domínio de negócios. Ao focar na linguagem compartilhada, na modelagem cuidadosa do domínio e na definição clara de contextos, o DDD permite que desenvolvedores e especialistas de negócio colaborem de forma mais eficaz, garantindo que o software atenda às necessidades do negócio de maneira precisa e sustentável.

4.3.7 Pirâmide de testes

A pirâmide de testes é um conceito amplamente utilizado no desenvolvimento de software para estruturar e organizar os diferentes tipos de testes em um projeto, visando alcançar alta cobertura de código com menor custo e esforço.

Criada por Cohn, e apresentada no seu livro *Succeeding with Agile* (2009) essa estratégia de visualizar os testes como uma pirâmide, onde cada camada representa um tipo de teste, variando desde testes de unidade, mais frequentes e rápidos, até os testes de interface de usuário (UI), mais complexos e custosos. A pirâmide de testes é composta por três principais camadas:

- **Testes de Unidade** (Base da Pirâmide): Focados em pequenas unidades do código, como funções ou métodos, isoladamente. Esses testes são rápidos, baratos e fornecem *feedback* quase instantâneo sobre falhas. Devido à sua simplicidade e rapidez, é recomendável que essa camada tenha o maior número de testes.
- **Testes de Integração** (Meio da Pirâmide): Validam a interação entre diferentes módulos ou componentes da aplicação. Eles garantem que os módulos funcionem corretamente juntos, como por exemplo, verificar a comunicação entre o front-end e o back-end ou entre o sistema e o banco de dados. Esses testes são mais lentos que os de unidade e demandam mais configuração.
- **Testes de Interface de Usuário** (UI): Validam o sistema como um todo do ponto de vista do usuário, verificando a interface e a experiência do usuário. Eles simulam a interação real com a aplicação, a recomendação é que esses testes sejam mais escassos, cobrindo os fluxos mais críticos.

A estrutura em forma de pirâmide sugere que o número de testes diminui conforme subimos na pirâmide, ou seja, mais testes de unidade, uma quantidade moderada de testes de integração e menos testes de UI. Isso otimiza o esforço de desenvolvimento e o custo de manutenção dos testes, pois testes de unidade são mais fáceis de manter e rápidos de executar, enquanto testes de interface são mais complexos e demorados.

A pirâmide de testes é um modelo essencial para garantir a qualidade do software, maximizando a eficiência dos testes e minimizando o custo de manutenção. Seguindo essa abordagem, as equipes de desenvolvimento podem garantir que suas aplicações estejam bem cobertas por testes em diferentes níveis, resultando em um sistema mais confiável e de fácil manutenção. Com uma boa estratégia de testes, problemas podem ser detectados mais cedo, facilitando a resolução e evitando surpresas nas fases finais de desenvolvimento.

4.3.8 CI/CD (*Continuous Integration/Continuous Delivery*)

CI/CD é um conjunto de práticas modernas no desenvolvimento de software que visa automatizar e otimizar a integração, teste e entrega de código, tornando o processo de desenvolvimento mais ágil e eficiente. Essas práticas ajudam as equipes a reduzir erros, aumentar a qualidade e fornecer novos recursos de forma mais rápida e confiável aos usuários. A seguir, abordaremos os principais conceitos de Integração Contínua (CI) e Entrega Contínua/Implantação Contínua (CD), destacando seus benefícios, funcionamento e importância no ciclo de vida de desenvolvimento de software.

Em seu livro *Criando Microserviços*, Newman (2022) afirma que o principal objetivo do CI é manter todos em sincronia, o que é feito garantindo frequentemente que um código cujo *check-in* tenha sido feito recentemente se integre de modo apropriado ao código existente, assim o serviço de Integração Contínua detecta que houver *commit* de código, faz seu *check-out* e executa alguma espécie de verificação, por exemplo, garanta que o código compila e que os testes sejam bem-sucedidos.

Funcionando da seguinte forma: os desenvolvedores fazem pequenas alterações no código e as enviam (*commit*) para o repositório central, onde o código é integrado de forma automática. Um servidor de CI, como Jenkins, GitLab CI, CircleCI ou Travis CI, executa uma série de testes automatizados para verificar se o novo código funciona corretamente e se não quebrou nenhuma parte do sistema existente. Caso algum teste falhe ou o código não passe nas verificações, o sistema alerta a equipe, permitindo uma correção rápida.

Seus benefícios são a detecção precoce de erros, já que os testes são executados automaticamente a cada nova integração, problemas são detectados cedo, facilitando a correção. Menos conflitos de merge, pois pequenas mudanças de código, integradas com frequência, tornam mais fácil a fusão

de código entre diferentes desenvolvedores, reduzindo os conflitos de merge. Maior qualidade do código, com testes automáticos frequentes, o código tende a estar sempre em um estado mais confiável e livre de erros e agilidade no desenvolvimento, já que ao identificar problemas rapidamente, os desenvolvedores podem focar na implementação de novos recursos, sem ficar presos em correções demoradas.

Já a Entrega Contínua, de acordo com Jez Humble e Dave Farley no livro *Continuous Delivery* (2010) é a abordagem por meio da qual temos um *feedback* contínuo para saber se todo e qualquer *check-in* está pronto para o ambiente de produção; desse modo, todo e qualquer *check-in* é tratado como candidato para uma versão a ser lançada. O processo ocorre após a CI e visa garantir que o código aprovado esteja sempre pronto para ser liberado em produção. Após a integração bem-sucedida e a execução dos testes automatizados, o código é preparado para ser entregue de forma contínua ao ambiente de produção ou de homologação (pré-produção). No entanto, na entrega contínua, a publicação em produção ainda requer uma ação manual, como uma aprovação final.

Funcionando da seguinte forma, após a aprovação da CI, a entrega contínua automatiza o processo de empacotamento, implantação e configuração do código em um ambiente de teste ou produção. A equipe pode realizar o *deploy* com apenas um clique, garantindo que a versão mais recente do código está pronta para ser liberada em produção. Ferramentas como GitLab CI, Jenkins, AWS CodePipeline e Octopus Deploy auxiliam na automação do pipeline de entrega contínua.

Tendo as seguintes vantagens, como o menor tempo de entrega, pois o software está sempre pronto para ser implantado em produção, o que reduz o tempo de espera para novas funcionalidades ou correções. Redução de riscos, como o código passa por várias etapas automatizadas de teste e verificação antes de ser entregue, o risco de bugs ou falhas graves na produção é reduzido e confiabilidade logo com a automatização do pipeline de entrega minimiza a ocorrência de erros humanos durante a implantação, o que resulta em releases mais confiáveis.

O CI/CD é uma abordagem fundamental para equipes que buscam agilidade, qualidade e eficiência no desenvolvimento de software. Ao automatizar os testes, a integração e a entrega do código, as equipes podem garantir lançamentos frequentes e confiáveis, com menos falhas e maior rapidez. Essa prática é especialmente valiosa em projetos de grande escala ou em ambientes que exigem respostas rápidas às mudanças do mercado, tornando o processo de desenvolvimento contínuo e adaptável.

4.4 Segurança

Em um cenário tecnológico cada vez mais conectado e complexo, a segurança no desenvolvimento de software se tornou um dos pilares fundamentais para garantir a integridade, confidencialidade e disponibilidade

dos sistemas. Vulnerabilidades e ataques cibernéticos estão em constante evolução, e a proteção dos dados dos usuários e a resiliência das aplicações diante dessas ameaças são aspectos críticos que precisam ser endereçados desde as fases iniciais do desenvolvimento. Implementar boas práticas de segurança não é apenas uma exigência técnica, mas uma responsabilidade ética e legal, que pode evitar danos à reputação da empresa, prejuízos financeiros e perda de confiança por parte dos usuários. Neste contexto, este abordaremos os erros mais comuns e que podem passar despercebidos pelos desenvolvedores.

4.4.1 Princípios básicos

4.4.1.1 Princípio do privilégio mínimo

O Princípio do Privilégio Mínimo descreve a ideia de que, ao conceder um acesso, devemos conceder o mínimo necessário a uma entidade para que ela execute a funcionalidade, e somente pelo tempo que precisa, assim destaca Sam Newman em seu livro Criando Microsserviços. Isso determina que usuários, processos e sistemas devem ter acesso apenas aos recursos estritamente necessários para desempenhar suas funções.

Essa abordagem reduz significativamente a superfície de ataque e minimiza os riscos de exploração, garantindo que qualquer comprometimento afete o mínimo possível da infraestrutura. No contexto do desenvolvimento de software, a implementação desse princípio envolve limitar as permissões de cada usuário ou serviço com base em seu papel no sistema, por meio da criação de grupos ou roles que definem as permissões e responsabilidades de cada componente.

Esses grupos de permissões são definidos conforme as necessidades operacionais de cada tipo de usuário. Por exemplo, o **usuário comum** tem permissões restritas, limitadas a funcionalidades básicas, sem acesso a dados sensíveis ou áreas administrativas, o que é crucial em sistemas de larga escala, como plataformas de e-commerce. Já o **administrador** possui permissões mais amplas, com acesso a configurações de sistemas e dados sensíveis, mas também está sujeito a rigorosas práticas de segurança, como autenticação multifator e monitoramento constante. Outro grupo importante é o dos **moderadores**, que têm permissões intermediárias, geralmente usados em plataformas de conteúdo, onde podem editar ou remover dados sem comprometer informações críticas do sistema. **Desenvolvedores** também possuem permissões específicas, muitas vezes restritas a ambientes de desenvolvimento e testes, sem acesso direto a sistemas de produção.

A adoção do Princípio do Privilégio Mínimo proporciona diversas vantagens para a segurança e a gestão dos sistemas. Primeiramente, reduz a superfície de ataque ao limitar os acessos apenas ao estritamente necessário. Além disso, caso um usuário ou serviço seja comprometido, o impacto é limitado às áreas nas quais eles possuem permissão. Também permite um controle mais granular de acessos, assegurando que somente usuários ade-

quadros realizem operações críticas, e ajuda a cumprir regulamentações de privacidade e segurança, como a LGPD e o GDPR, que exigem práticas rigorosas de controle de acesso.

4.4.1.2 Defesa em profundidade

A Defesa em Profundidade é uma estratégia de segurança baseada na implementação de múltiplas camadas de proteção ao redor dos sistemas e dados sensíveis. Inspirada em práticas militares, essa abordagem visa criar barreiras sucessivas que tornam mais difícil para um invasor comprometer um sistema, garantindo que, mesmo que uma camada de segurança seja violada, outras camadas permaneçam ativas para mitigar o impacto. No contexto do desenvolvimento de software e sistemas, a defesa em profundidade é essencial para assegurar a resiliência contra ataques, combinando várias técnicas de proteção em diferentes níveis da arquitetura de um sistema.

A primeira camada de defesa costuma ser o controle de acesso, onde o princípio do privilégio mínimo é aplicado para limitar o que usuários e sistemas podem acessar. Ferramentas como autenticação multifator (MFA) são amplamente utilizadas para garantir que apenas usuários autorizados possam acessar áreas críticas do sistema. Em seguida, vêm as medidas de criptografia, tanto para proteger dados em trânsito quanto em repouso, garantindo que, mesmo se interceptados, os dados sejam incompreensíveis sem a chave de decriptação apropriada.

Além do controle de acesso e criptografia, outra camada importante é a segurança na rede, que inclui firewalls, sistemas de detecção e prevenção de intrusões (IDS/IPS) e segmentação de redes. Esses mecanismos atuam para bloquear acessos não autorizados e monitorar o tráfego de rede em busca de atividades suspeitas. No nível de aplicação, a validação de entrada e saída de dados também desempenha um papel crucial na defesa em profundidade, prevenindo ataques como injeção de SQL e Cross-Site Scripting (XSS), que podem explorar vulnerabilidades em formulários e APIs.

Outra camada é a proteção contra malwares e vulnerabilidades, que envolve manter sistemas e softwares atualizados com os últimos patches de segurança. Além disso, soluções de endpoint, como antivírus e monitoramento de integridade, garantem que comportamentos maliciosos sejam detectados antes que causem danos significativos.

Por fim, a auditoria e monitoramento contínuos fornecem uma linha de defesa essencial. Logs detalhados, revisões periódicas de segurança e a implementação de um sistema de alerta em tempo real ajudam a identificar e responder rapidamente a tentativas de invasão. Quando ocorrem violações, os logs de auditoria podem ser analisados para entender a origem do ataque e tomar medidas corretivas eficazes.

A vantagem principal da Defesa em Profundidade é que ela não confia em uma única medida de segurança. Em vez disso, oferece redundância, garantindo que mesmo que uma camada seja comprometida, as outras conti-

em a proteger o sistema. Isso cria um ambiente mais resiliente a falhas e ataques, aumentando a probabilidade de detecção e resposta antes que um ataque cause danos irreparáveis.

4.4.1.3 Incluir a segurança no processo de entregas

Incluir a segurança no processo de entregas é uma boa prática para garantir que o software seja não apenas funcional, mas também seguro desde o desenvolvimento até o ambiente de produção. O foco na segurança em cada etapa do ciclo de vida do desenvolvimento de software (SDLC - Software Development Life Cycle) ajuda a mitigar vulnerabilidades antes que elas cheguem aos usuários finais e se tornem ameaças reais.

Uma abordagem comum para implementar a segurança nas entregas é adotar o conceito de DevSecOps, que é a abreviação de desenvolvimento, segurança e operações, é uma prática de desenvolvimento de aplicações que automatiza a integração de práticas de segurança e proteção em todas as fases do ciclo de vida do desenvolvimento de software, desde o design inicial até a integração, testes, entrega e implementação, de acordo com a IBM.

Assim, diferente do modelo tradicional, onde a segurança é tratada de forma isolada e apenas na fase final de testes, o DevSecOps promove uma cultura onde todos, desde desenvolvedores e engenheiros de operações, são responsáveis pela segurança. Isso significa que práticas de segurança são incorporadas desde a fase de planejamento e desenvolvimento, passando por testes automatizados e revisões de código, até o momento da entrega em produção.

No contexto de *Continuous Integration/Continuous Delivery* (CI/CD), que envolve a automação de testes e a implantação frequente de novas versões de software, a segurança precisa ser parte integrante do pipeline de entrega. Ferramentas de análise estática de código (SAST) podem ser integradas ao pipeline para verificar vulnerabilidades no código antes da compilação. Além disso, testes de segurança dinâmicos (DAST) podem ser automatizados para identificar vulnerabilidades em aplicações em execução, como falhas em APIs e problemas de autenticação.

Outros aspectos importantes incluem a verificação de dependências e bibliotecas de terceiros. Ferramentas como o OWASP Dependency-Check ajudam a identificar vulnerabilidades conhecidas em bibliotecas que o software utiliza, garantindo que apenas versões seguras sejam implantadas. A configuração segura de ambientes também deve ser verificada automaticamente em cada entrega, garantindo que permissões excessivas, portas desnecessárias e configurações padrão inseguras não estejam presentes na aplicação ou infraestrutura de produção.

Além dos testes automáticos, a auditoria manual de segurança é essencial para revisar pontos críticos do sistema, principalmente em entregas maiores ou em componentes sensíveis. Equipes especializadas podem conduzir testes de intrusão (*pentests*) periodicamente para identificar brechas

que podem não ser detectadas por ferramentas automatizadas.

Integrar a segurança ao processo de entrega não apenas melhora a qualidade e robustez do software, mas também reduz significativamente os riscos de incidentes de segurança pós-implantação, aumentando a confiança do usuário final no sistema e reduzindo o custo de remediação de vulnerabilidades.

4.4.2 As cinco funções da cibersegurança

A cibersegurança é uma disciplina fundamental para proteger sistemas, redes e dados contra ameaças e ataques cibernéticos. No contexto de segurança da informação, as cinco funções primárias foram estabelecidas pelo NIST (National Institute of Standards and Technology) como um framework para garantir uma abordagem abrangente e eficaz. Essas funções são Identificar, Proteger, Detectar, Responder e Recuperar-se, e fornecem um ciclo contínuo de ações para prevenir e mitigar incidentes de segurança.

4.4.2.1 Identificar

A primeira função, Identificar, envolve a compreensão dos riscos de segurança cibernética para os sistemas e ativos da organização. Isso inclui identificar os recursos críticos, entender as vulnerabilidades presentes e avaliar as ameaças potenciais. A identificação dos ativos e dados importantes permite que as organizações priorizem a proteção dos mesmos e criem estratégias adequadas de mitigação de riscos. Além disso, essa fase inclui o mapeamento dos processos organizacionais e a atribuição de responsabilidades para a segurança da informação.

4.4.2.2 Proteger

A função Proteger está diretamente relacionada à implementação de medidas para garantir que os ativos identificados estejam seguros. Nesta fase, a organização coloca em prática controles de segurança para limitar ou impedir o impacto de um possível ataque. Isso inclui a adoção de políticas de acesso, criptografia, firewalls, treinamento de colaboradores para boas práticas de segurança e a implementação de mecanismos de autenticação robustos. O objetivo principal aqui é criar barreiras eficazes para prevenir que ameaças cibernéticas tenham sucesso.

4.4.2.3 Detectar

A função Detectar é fundamental para identificar, em tempo hábil, quando uma ameaça ou ataque ocorre. Isso envolve a implementação de sistemas de monitoramento, alertas e ferramentas de detecção de intrusões (IDS) que possibilitem a identificação de atividades maliciosas ou anômalas. A

detecção eficaz de ameaças é vital para a resposta rápida, e pode envolver o uso de logs, análise de comportamento, e inteligência artificial para detectar variações nos padrões de tráfego de rede ou acessos suspeitos.

4.4.2.4 Responder

Uma vez que uma ameaça é detectada, a organização deve estar preparada para Responder. Isso inclui a ativação de planos de resposta a incidentes, onde as equipes de segurança agem para conter, mitigar e neutralizar a ameaça. A função de resposta envolve a comunicação adequada entre as partes interessadas, a investigação da origem do incidente e a implementação de correções e patches para eliminar vulnerabilidades. A resposta rápida e coordenada pode reduzir significativamente o impacto de um ataque.

4.4.2.5 Recuperar-se

A última função, Recuperar-se, concentra-se em restaurar a funcionalidade dos sistemas e serviços após um incidente de segurança. Esta etapa envolve a implementação de planos de recuperação de desastres e continuidade de negócios, a reparação dos sistemas comprometidos e a análise das lições aprendidas com o incidente. O objetivo é não apenas retornar à operação normal, mas também fortalecer as defesas para evitar que o mesmo tipo de ataque aconteça no futuro.

4.4.3 Credenciais e variáveis de ambiente

O uso seguro de credenciais e variáveis de ambiente é um dos pilares da segurança em desenvolvimento de software, principalmente em sistemas distribuídos e aplicações em nuvem. Credenciais, como senhas, chaves de API e tokens, são necessárias para acessar recursos e serviços sensíveis, enquanto variáveis de ambiente são uma forma comum de armazenar essas informações de maneira que possam ser facilmente alteradas sem necessidade de modificar o código-fonte da aplicação.

Manter as credenciais seguras é essencial para prevenir ataques como roubo de dados, sequestro de sessão ou acesso não autorizado a serviços críticos. Um erro comum em projetos de software é armazenar essas credenciais diretamente no código, o que aumenta significativamente o risco de exposição, especialmente em repositórios públicos.

As variáveis de ambiente oferecem uma solução prática, pois permitem que as credenciais sejam armazenadas fora do código e fornecidas em tempo de execução. Essa abordagem facilita a troca de credenciais entre ambientes (como desenvolvimento, teste e produção), sem a necessidade de alterações no código.

Algumas boas práticas são recomendadas para o uso seguro de credenciais e variáveis de ambiente. As variáveis de ambiente devem ser usadas para configurar as credenciais, evitando o armazenamento em arquivos de configuração ou no código-fonte. Além disso, é recomendada a rotação periódica de credenciais para mitigar o impacto de possíveis vazamentos. Outro aspecto importante é a aplicação do princípio do privilégio mínimo, garantindo que as credenciais tenham apenas as permissões estritamente necessárias para a operação da aplicação. Por fim, é fundamental garantir a criptografia das credenciais tanto em repouso quanto em trânsito, para que, mesmo em caso de interceptação, os dados não sejam facilmente acessíveis.

Essas práticas promovem a segurança e a gestão eficiente de credenciais e variáveis de ambiente, minimizando os riscos de exposição e acessos indevidos em sistemas de software.

4.4.4 Backup

O backup é uma prática fundamental no desenvolvimento de software e na administração de sistemas, com o objetivo de garantir a segurança e a disponibilidade dos dados em caso de falhas, desastres ou ataques cibernéticos. Ele consiste na cópia e no armazenamento de dados críticos em um local seguro, permitindo a sua recuperação em situações de perda, corrupção ou comprometimento. A implementação de um bom plano de backup é essencial para a continuidade de negócios e a mitigação de riscos associados à perda de dados.

Existem diversas abordagens para backup, como o completo, diferencial e incremental. No backup completo, uma cópia integral de todos os dados é realizada, o que garante a total recuperação, mas pode exigir grandes quantidades de espaço de armazenamento e tempo. O backup diferencial, por sua vez, copia apenas os dados que mudaram desde o último backup completo, economizando tempo e espaço, mas aumentando o tempo de restauração, uma vez que é necessário ter o último backup completo e o diferencial. Já o backup incremental armazena apenas as mudanças feitas desde o último backup de qualquer tipo (completo ou incremental), sendo mais eficiente em termos de armazenamento, porém mais complexo para restaurar.

A estratégia de backup deve ser desenhada levando em consideração a criticidade dos dados, a frequência de alterações e o tempo máximo aceitável de inatividade para a organização. Além disso, boas práticas recomendam a utilização da regra 3-2-1: manter três cópias dos dados, em dois formatos diferentes, e uma delas fora do local original, para assegurar redundância e resiliência contra desastres locais, como incêndios ou inundações.

Além da realização dos backups, é igualmente importante testar regularmente a recuperação dos dados para garantir que, em caso de necessidade, a restauração será bem-sucedida. Sem esse cuidado, a simples existência de backups pode dar uma falsa sensação de segurança, quando na

verdade os dados podem não estar adequadamente protegidos ou disponíveis.

Em resumo, o backup é uma peça-chave em qualquer estratégia de segurança da informação e continuidade de negócios. Sua correta implementação, combinada com uma rotina de testes de recuperação, garante que as organizações estejam preparadas para lidar com situações adversas que possam comprometer os dados críticos.

4.4.5 Validação de inputs

A validação de *inputs* é uma prática crucial em qualquer aplicação, sendo um dos primeiros passos para garantir a segurança e a integridade dos dados que são processados. Validar todos os dados que entram na aplicação significa garantir que apenas informações no formato correto, dentro dos limites esperados e sem valores maliciosos sejam aceitas, evitando problemas como erros de sistema, falhas de operação e, principalmente, vulnerabilidades de segurança.

Ao não validar corretamente os *inputs*, a aplicação pode se tornar vulnerável a diversos ataques, como injeção de código SQL, XSS (*Cross-Site Scripting*) e *buffer overflow*, entre outros. Esses tipos de ataques exploram entradas maliciosas que não foram devidamente tratadas, permitindo que o atacante manipule a aplicação ou tenha acesso indevido a informações confidenciais.

A validação de *inputs* deve ocorrer tanto no lado cliente (*frontend*), para proporcionar uma experiência de usuário mais fluida, quanto no lado servidor (*backend*), onde a validação é crítica para proteger o sistema de entradas que podem ser manipuladas. A validação no *frontend* ajuda a garantir que dados incorretos não sejam enviados para o servidor, enquanto a validação no *backend* assegura que dados maliciosos ou alterados não comprometam o sistema.

Algumas práticas recomendadas para validação de *inputs* incluem garantir que os dados sejam verificados tanto quanto ao tipo (números, *strings*, datas, etc.), quanto ao formato esperado, como e-mails e números de telefone. É essencial restringir os tipos de dados aceitos de acordo com o contexto, limitando o tamanho dos campos para evitar sobrecargas ou ataques de *buffer overflow*. Além disso, é fundamental definir faixas de valores aceitáveis (por exemplo, garantir que um campo de idade não aceite números negativos ou irreais) e usar expressões regulares para validar padrões específicos, como o formato de um CPF, um endereço de e-mail ou um número de cartão de crédito.

No desenvolvimento moderno, ferramentas e bibliotecas de validação de dados facilitam esse processo, permitindo que a validação seja centralizada e eficiente. Como o Zod, uma biblioteca de validação de schemas em TypeScript e JavaScript que permite definir e validar objetos de maneira altamente expressiva. O Hibernate Validator, ele oferece um conjunto de anotações para validar campos diretamente nas classes de entidades ou DTOs, facilitando a validação automática de inputs de dados em aplicações Java.

Além do Laravel Validation para aplicações Laravel, o framework já traz embutido um sistema de validação robusto e fácil de usar, tanto em Request Objects quanto em controladores.

Portanto, validar todos os dados que entram em uma aplicação é uma camada essencial de segurança e robustez, evitando erros, protegendo contra ameaças e garantindo que a aplicação funcione corretamente mesmo quando exposta a entradas inesperadas ou maliciosas.

4.4.6 Criptografia

A criptografia é uma técnica essencial para garantir a segurança da informação em sistemas de software, sendo fundamental na proteção de dados sensíveis e assegurando que apenas usuários autorizados possam acessá-los. No contexto do desenvolvimento de software, a criptografia é amplamente utilizada para encriptar senhas, dados sensíveis e o tráfego de dados, evitando que informações confidenciais sejam comprometidas, mesmo que um invasor tenha acesso a esses dados. A seguir, detalhamos as principais práticas de encriptação de senhas e dados sensíveis, além da criptografia no tráfego de dados.

Encriptar senhas é uma medida de segurança fundamental em qualquer sistema que armazene credenciais de usuários. Em vez de armazenar senhas em texto puro, o que representaria um risco significativo em caso de vazamento de dados, as senhas devem ser encriptadas antes de serem armazenadas no banco de dados. As principais práticas de encriptação de senhas incluem o uso de hashing seguro, salting e derivação de chaves.

O *hashing* seguro consiste em transformar uma senha em um valor fixo e irreversível, utilizando algoritmos como bcrypt, argon2 e scrypt. Esses algoritmos são recomendados por sua resistência a ataques de força bruta, uma vez que implementam técnicas de *salting* e iteração. O *salting*, por sua vez, envolve a adição de um valor aleatório (*salt*) à senha antes da aplicação do *hashing*, o que assegura que senhas iguais resultem em *hashes* diferentes. Essa abordagem é crucial para proteger contra ataques de tabelas arco-íris, que utilizam pré-computação de *hashes* para tentar quebrar senhas.

Além da proteção de senhas, é fundamental garantir a segurança de outros dados sensíveis, como informações pessoais, financeiras e médicas. A encriptação de dados sensíveis visa proteger a integridade e a confidencialidade das informações armazenadas e processadas pelo sistema. As abordagens mais comuns incluem a criptografia simétrica e a criptografia assimétrica.

A criptografia simétrica utiliza uma única chave para encriptar e decriptar os dados. Por outro lado, a criptografia assimétrica utiliza um par de chaves – uma pública e uma privada. A chave pública é utilizada para encriptar os dados, enquanto a chave privada é necessária para decapitá-los. Esse método é particularmente comum em transações que exigem segurança, como na troca de dados em e-commerce e em aplicativos bancários.

Ademais, a encriptação em repouso é uma prática recomendada para proteger dados que estão armazenados em discos, assegurando que mesmo que o banco de dados ou a infraestrutura sejam comprometidos, as informações continuem protegidas. Já a encriptação em trânsito é crucial para proteger dados transmitidos entre o cliente e o servidor, ou entre diferentes sistemas.

A criptografia do tráfego de dados é uma camada adicional de segurança que protege as informações que estão sendo transmitidas entre servidores e clientes, garantindo que terceiros não possam interceptar ou modificar os dados em trânsito. Esse tipo de proteção é especialmente importante em transações bancárias, comunicações privadas e na transmissão de dados médicos.

O uso de HTTPS (HTTP Secure) é uma prática recomendada para todos os sites que lidam com informações sensíveis, como senhas e dados financeiros. O HTTPS utiliza o TLS para encriptar a comunicação entre o navegador e o servidor, assegurando uma navegação segura e protegida. Além disso, o uso de VPNs (Virtual Private Networks) pode ser uma estratégia eficaz, criando um "túnel" encriptado para o tráfego de rede, garantindo que todos os dados que passam por essa conexão estejam protegidos contra interceptação.

Em suma, a criptografia é um pilar central da segurança no desenvolvimento de software. Ao implementar técnicas robustas de encriptação de senhas, dados sensíveis e tráfego de dados, os desenvolvedores podem proteger as informações dos usuários contra ataques e vazamentos. A adoção dessas técnicas de criptografia é vital para a construção de sistemas que respeitem a privacidade dos usuários e protejam suas informações mais sensíveis.

4.4.7 Autenticação

A autenticação é um componente crítico na segurança de sistemas de software, pois garante que apenas usuários autorizados possam acessar informações ou funcionalidades específicas. Uma abordagem moderna e amplamente utilizada para autenticação é o uso de JSON Web Tokens (JWT), que não só oferece um mecanismo de autenticação seguro, mas também facilita a transmissão de informações de forma compacta e autônoma entre partes. A importância do JWT se estende além da simples autenticação; ele serve como um meio eficaz de transportar informações de forma segura e confiável.

O JSON Web Token (JWT) é um padrão aberto (RFC 7519) que define um formato compacto e autônomo para a transmissão segura de informações. Um JWT é composto por três partes principais: o cabeçalho (*header*), o corpo (*payload*) e a assinatura (*signature*). O cabeçalho normalmente contém o tipo de token e o algoritmo de assinatura, que é codificado em Base64Url. O corpo, por sua vez, contém as afirmações (*claims*), que são as informações que se deseja transmitir, como a identidade do usuário e suas permissões.

Assim como o cabeçalho, o corpo também é codificado em Base64Url. Para garantir a autenticidade e a integridade do token, a assinatura é criada utilizando o cabeçalho codificado, o corpo codificado e uma chave secreta (ou chave privada, em algoritmos assimétricos).

O processo de autenticação com JWT envolve várias etapas. Inicialmente, o usuário fornece suas credenciais ao sistema, que as valida. Uma vez validadas, o servidor gera um JWT contendo as informações do usuário e o envia de volta ao cliente. Esse token deve ser armazenado em um local seguro, como o armazenamento local (localStorage) ou cookies, e, ao acessar rotas ou recursos protegidos, o cliente inclui o JWT no cabeçalho da requisição HTTP, utilizando o esquema de autenticação "Bearer". O servidor, ao receber a requisição, extrai o token do cabeçalho, valida sua assinatura e verifica se não está expirado. Se a validação for bem-sucedida, o servidor processa a requisição e concede acesso aos recursos solicitados.

O uso de JWT para autenticação oferece diversas vantagens. Por ser um token autônomo, ele contém todas as informações necessárias para a autenticação, eliminando a necessidade de armazenar sessões no servidor, o que facilita a escalabilidade em ambientes distribuídos. Sua compatibilidade com JSON torna o JWT fácil de usar em aplicações web e móveis, além de permitir a inclusão de informações personalizadas no *payload*. A assinatura do token assegura que o conteúdo não foi alterado, e a possibilidade de definir uma data de expiração aumenta a segurança ao limitar o tempo de validade do token.

Um aspecto fundamental da autenticação por JWT é o uso de tokens de renovação, conhecidos como *refresh tokens*. Enquanto um JWT pode ter um tempo de expiração relativamente curto para minimizar os riscos de segurança, o *refresh token* permite que o usuário permaneça autenticado sem precisar fazer login novamente. Quando o JWT expira, o cliente pode enviar o *refresh token* ao servidor para solicitar um novo JWT. Isso não apenas melhora a experiência do usuário, mas também mantém um nível de segurança elevado, pois o *refresh token* deve ser tratado com um nível de proteção superior e armazenado em um local seguro.

Para implementar autenticação via JWT e *refresh tokens*, os desenvolvedores podem seguir algumas práticas recomendadas. A primeira delas é o uso de HTTPS para proteger a transmissão de tokens contra interceptações. O gerenciamento das chaves secretas é igualmente importante; elas devem ser armazenadas em locais seguros e não expostas no código-fonte. Também é vital implementar mecanismos de validação e revogação de tokens, especialmente se houver suspeitas de comprometimento ou quando um usuário se desconectar. Finalmente, a definição de um tempo de expiração adequado para os tokens, juntamente com a implementação de um sistema de renovação eficiente para os refresh tokens, ajuda a minimizar os riscos de segurança.

A autenticação via JWT representa uma solução moderna e eficaz para garantir a segurança em sistemas de software. Ao possibilitar uma comunica-

ção segura entre clientes e servidores, os desenvolvedores podem construir aplicações mais seguras e escaláveis, atendendo às crescentes demandas por proteção de dados e controle de acesso. A adoção de práticas robustas de autenticação, incluindo o uso de refresh tokens, não só melhora a experiência do usuário, mas também fortalece a segurança geral do sistema.

4.4.8 Bibliotecas

No desenvolvimento de software, as bibliotecas desempenham um papel crucial, proporcionando funcionalidades pré-construídas que facilitam a implementação de características complexas e aceleram o processo de desenvolvimento. No entanto, o uso de bibliotecas desatualizadas ou não confiáveis pode introduzir vulnerabilidades significativas, comprometendo a segurança e a integridade das aplicações. Portanto, a atualização constante das bibliotecas e a seleção de ferramentas confiáveis são práticas fundamentais para garantir a segurança e a robustez do software.

As bibliotecas frequentemente recebem atualizações que corrigem falhas de segurança, melhoram o desempenho e introduzem novas funcionalidades. Ignorar essas atualizações pode deixar o sistema vulnerável a ataques que exploram essas falhas conhecidas. As atualizações também podem incluir melhorias de compatibilidade com novas versões de linguagem ou frameworks, garantindo que a aplicação continue a funcionar sem problemas. Por isso, os desenvolvedores devem implementar um processo regular de verificação e atualização das bibliotecas utilizadas em suas aplicações.

A escolha de bibliotecas confiáveis é igualmente importante. As bibliotecas amplamente adotadas e mantidas por comunidades ativas geralmente são mais seguras, pois são auditadas e testadas por diversos desenvolvedores ao longo do tempo. Antes de incorporar uma nova biblioteca em um projeto, é recomendável verificar seu histórico de manutenção, a frequência de atualizações, a documentação e as avaliações da comunidade. Utilizar bibliotecas de fontes respeitáveis, como repositórios oficiais e organizações reconhecidas, pode reduzir o risco de introduzir vulnerabilidades no código.

Existem diversas ferramentas disponíveis para ajudar os desenvolvedores a gerenciar e atualizar suas bibliotecas de forma eficiente. Por exemplo, gerenciadores de pacotes como npm para JavaScript, pip para Python e Maven para Java oferecem recursos que facilitam a atualização automática de dependências. Essas ferramentas também podem alertar sobre versões desatualizadas e vulnerabilidades conhecidas, permitindo que os desenvolvedores tomem medidas proativas para manter suas aplicações seguras.

A atualização regular de bibliotecas e a escolha de ferramentas confiáveis são essenciais para garantir a segurança e a estabilidade de aplicações de software. Ao adotar práticas rigorosas de gerenciamento de dependências, os desenvolvedores podem proteger seus sistemas contra vulne-

rabilidades e garantir um ambiente de desenvolvimento mais seguro e eficiente. A atenção a esses detalhes não apenas promove a segurança, mas também contribui para a qualidade e a confiabilidade do software desenvolvido.

4.4.9 Código

A verificação de código é uma etapa essencial no desenvolvimento de software, especialmente no que diz respeito à segurança. Vulnerabilidades presentes no código podem ser exploradas por invasores, resultando em incidentes graves, como vazamentos de dados ou interrupções de serviços. Ferramentas de análise estática são utilizadas para identificar potenciais falhas de segurança durante o desenvolvimento, antes que o código seja executado ou chegue ao ambiente de produção. Além dessas ferramentas, testes de segurança específicos, como testes de penetração e revisões de código automatizadas, desempenham um papel fundamental na proteção dos sistemas contra possíveis ameaças.

Ferramentas de análise estática de código têm como objetivo revisar o código-fonte em busca de vulnerabilidades que podem passar despercebidas durante o desenvolvimento. Elas verificam questões como manipulação inadequada de dados de entrada, escapes incorretos e possíveis exposições de dados sensíveis, além de identificar falhas que possam ser exploradas por ataques, como *Cross-Site Scripting (XSS)*, Injeção de SQL, entre outros. Essas ferramentas variam de acordo com a linguagem de programação utilizada, e cada uma oferece suporte a diferentes padrões e requisitos de segurança.

Ao utilizar tais ferramentas, como Bandit para Python, ESLint-Security para Node, SpotBugs para Java ou Gosec para Go, o processo de verificação de código pode ser automatizado, permitindo a identificação precoce de vulnerabilidades durante o ciclo de desenvolvimento. Essas ferramentas garantem que falhas comuns sejam detectadas e corrigidas antes que o código seja enviado para produção, minimizando os riscos de exposição a ataques.

Adotar uma abordagem integrada para verificação de código e testes de segurança é uma prática recomendada para o desenvolvimento seguro. As melhores práticas incluem ferramentas de verificação de código que devem ser incorporadas ao pipeline de integração contínua (CI). Isso garante que todo o código submetido ao repositório seja automaticamente verificado quanto a potenciais falhas de segurança, garantindo uma detecção mais precoce. Tanto os testes estáticos quanto os dinâmicos devem ser realizados ao longo do ciclo de desenvolvimento, garantindo que todas as camadas da aplicação sejam verificadas em diferentes fases. Capacitar os desenvolvedores em boas práticas de codificação segura e no uso eficaz de ferramentas de verificação de código é fundamental para evitar vulnerabilidades. A cultura de segurança deve ser parte integral da rotina de desenvolvimento.

A verificação de código e os testes de segurança são componentes essenciais para o desenvolvimento de software seguro. Utilizando ferramentas

apropriadas para análise estática e dinâmica, é possível detectar e corrigir vulnerabilidades antes que elas comprometam o sistema em produção. Implementar uma estratégia abrangente que combine análises automáticas de código com testes de segurança ajuda a criar uma aplicação mais segura e eficiente, protegendo-a contra potenciais ataques e reduzindo os custos associados à correção de falhas em estágios avançados do ciclo de desenvolvimento.

4.4.10 Logs

Os logs desempenham um papel fundamental na segurança de software, fornecendo uma trilha de auditoria que pode ser usada para monitorar atividades, detectar incidentes de segurança e identificar padrões de comportamento anômalos. Manter um sistema de logs robusto é essencial para a rastreabilidade, análise forense e detecção de possíveis ameaças, permitindo uma resposta rápida a incidentes de segurança.

A coleta e o monitoramento de logs em um sistema podem revelar eventos suspeitos, como tentativas de invasão, acessos não autorizados ou falhas no sistema. Em sistemas seguros, o registro de eventos importantes como autenticações, falhas de autenticação, manipulação de dados sensíveis e alterações nas permissões de acesso são críticos. Esses logs fornecem um histórico das ações realizadas no sistema, permitindo que administradores e equipes de segurança investiguem problemas e tomem medidas corretivas.

Além disso, logs bem estruturados podem facilitar a conformidade com regulamentações de segurança e privacidade, como a GDPR e a LGPD, que exigem um alto nível de responsabilidade e rastreamento de dados.

Imagine um cenário onde um sistema registra tentativas de login. Se um atacante tentar realizar um ataque de força bruta para adivinhar senhas, as tentativas repetidas de falha de login geram uma série de entradas de log com informações como o endereço IP, o nome de usuário e o horário das tentativas. Um sistema de monitoramento de logs bem configurado pode detectar esse padrão de falhas repetidas e ativar uma medida de segurança automática, como o bloqueio temporário do IP ou o envio de alertas à equipe de segurança.

Esses logs poderiam ser utilizados posteriormente para análise forense, ajudando a entender como o ataque foi realizado e em que ponto o sistema foi mais vulnerável. Sem esse tipo de registro detalhado, seria muito mais difícil identificar e mitigar o problema a tempo.

Para garantir a eficácia dos logs na segurança de software, algumas práticas recomendadas incluem o registro de eventos críticos, como autenticações, alterações de permissões, falhas de sistema e acessos a dados sensíveis que devem sempre ser registrados. A proteção dos logs, para garantir que os arquivos de log estejam protegidos contra adulteração e acesso não autorizado é fundamental para manter a integridade das informações.

Retenção e arquivamento, pois manter os logs por um período adequado permite uma análise completa de incidentes, atendendo a requisitos de conformidade e auditoria.

Monitoramento em tempo real, através de ferramentas de monitoramento de logs, como o ELK Stack ou Splunk, permite que incidentes sejam detectados e respondidos em tempo real.

Logs de segurança são essenciais para qualquer sistema, fornecendo uma base para detectar, mitigar e investigar problemas de segurança. Com a implementação correta de uma estratégia de logs, as empresas podem não só aumentar a segurança de seus sistemas, mas também melhorar sua capacidade de resposta a ameaças, garantindo maior proteção de dados e conformidade com regulamentações de segurança.

4.4.11 Proxy and Reverse Proxy

Proxies e reverse proxies desempenham um papel fundamental na arquitetura de segurança de software, oferecendo uma camada adicional de proteção e gerenciamento de tráfego entre clientes e servidores. Ambos atuam como intermediários nas comunicações, desempenhando funções distintas, porém complementares, ao otimizar, filtrar e proteger a rede contra acessos não autorizados ou ataques.

Um proxy é um servidor que age como intermediário entre o cliente, geralmente o navegador de um usuário, e o servidor que hospeda o recurso solicitado. Quando o cliente realiza uma requisição, esta passa pelo proxy antes de alcançar o servidor de destino. Os proxies são amplamente utilizados para várias finalidades de segurança e desempenho, como garantir anonimato, controle de acesso, filtragem de conteúdo e armazenamento em cache. Através do anonimato, o proxy pode ocultar o endereço IP real do cliente, oferecendo uma camada de privacidade e dificultando o rastreamento da origem da requisição.

No controle de acesso, é possível configurar proxies para bloquear ou permitir acessos a determinados sites ou serviços com base em políticas de segurança pré-definidas, o que impede acessos a sites maliciosos. Além disso, a filtragem de conteúdo possibilita que o proxy analise as informações que estão sendo acessadas, bloqueando malwares, anúncios indesejados ou qualquer outro tipo de conteúdo nocivo antes de chegar ao cliente. Por fim, os proxies podem armazenar versões em cache de conteúdos frequentemente acessados, acelerando o tempo de resposta e economizando largura de banda.

Por outro lado, um reverse proxy também atua como intermediário, mas, ao contrário de um proxy tradicional, ele representa o servidor e não o cliente. Todo o tráfego que chega ao servidor passa primeiro pelo reverse proxy, que decide como encaminhar a requisição ao servidor de destino. Esta tecnologia oferece diversos benefícios em termos de segurança e escalabilidade. Por exemplo, o reverse proxy pode ocultar detalhes da infraestrutura interna do servidor, como sua localização e identidade, dificul-

tando que invasores mapeiem a rede. Além disso, a distribuição de carga é um recurso valioso, já que o reverse proxy pode balancear o tráfego entre múltiplos servidores, assegurando que nenhum servidor fique sobrecarregado, o que aumenta a disponibilidade e o desempenho da aplicação. Outra vantagem importante é a autenticação centralizada.

O reverse proxy pode ser configurado para exigir autenticação e autorização, centralizando essas verificações e protegendo os servidores internos contra acessos não autorizados. Ademais, o reverse proxy pode ser responsável pela criptografia SSL/TLS, aliviando essa tarefa dos servidores internos e garantindo maior eficiência e segurança para a aplicação.

Um exemplo prático de uso de reverse proxy em um cenário de segurança é o Nginx configurado para平衡ar o tráfego entre diferentes servidores de aplicação. Em uma aplicação web de grande porte, o reverse proxy redireciona as requisições dos usuários para diversos servidores de backend, conforme a carga de cada servidor. Além disso, o Nginx pode ser configurado para inspecionar o tráfego e bloquear tentativas de ataques, como injeção de SQL ou *Cross-Site Scripting* (XSS), antes que alcancem os servidores de *backend*. Outro aspecto importante é o uso do SSL Termination, que permite ao reverse proxy decifrar o tráfego HTTPS recebido e encaminhá-lo de forma não criptografada para os servidores internos. Isso facilita a gestão dos certificados SSL e assegura que o tráfego criptografado seja devidamente inspecionado antes de chegar ao *backend*.

As vantagens de usar proxies e reverse proxies são numerosas. Em termos de segurança, ambos adicionam uma camada de proteção ao ocultar a infraestrutura real e ao filtrar o tráfego malicioso, protegendo contra tentativas de exploração de vulnerabilidades e ataques de negação de serviço (DDoS). Quanto à escalabilidade, os proxies e reverse proxies distribuem a carga de trabalho entre servidores, ampliando a capacidade de processamento da aplicação sem comprometer o desempenho. Além disso, eles facilitam o gerenciamento centralizado de autenticação, autorização e controle de acesso, aplicando políticas de segurança consistentes em toda a rede.

Tanto proxies quanto reverse proxies são ferramentas valiosas para aumentar a segurança e a eficiência de sistemas de software. Enquanto o proxy protege os clientes, fornecendo anonimato e filtragem de conteúdo, o reverse proxy protege os servidores ao controlar o fluxo de tráfego, distribuir a carga e evitar ataques. A implementação dessas tecnologias, especialmente em arquiteturas de software distribuídas, é uma prática recomendada que fortalece a segurança e otimiza o desempenho geral do sistema.

4.4.12 Vulnerabilidades mais exploradas

O OWASP (Open Web Application Security Project), também conhecido como Projeto Aberto de Segurança em Aplicações Web, é uma organização global dedicada à melhoria da segurança de software.

Um dos trabalhos mais importantes dessa organização é o levantamento das vulnerabilidades mais comuns e mais exploradas em aplicações web. A seguir, são descritas as principais vulnerabilidades identificadas pelo OWASP, que afetam a segurança de diversas aplicações e sistemas.

4.4.12.1 Controle de acesso quebrado

Essa vulnerabilidade ocorre quando as regras que determinam quem pode acessar ou modificar dados ou funcionalidades em um sistema estão mal implementadas. Isso permite que usuários não autorizados visualizem ou manipulem dados confidenciais, realizem ações administrativas ou acessem áreas restritas. Essa falha está frequentemente associada a permissões mal configuradas ou à falta de verificação de autorização em cada solicitação do usuário.

Implementar controles de acesso rigorosos em todas as funcionalidades, verificando a identidade e permissões dos usuários em cada requisição, e utilizando o princípio do menor privilégio, onde os usuários têm apenas as permissões necessárias para desempenhar suas funções. Também é essencial evitar que URLs ou parâmetros manipuláveis permitam bypass de autenticação e garantir que APIs sigam o mesmo rigor na validação de acesso.

4.4.12.2 Falhas de criptografia

Falhas de criptografia ocorrem quando os dados sensíveis, como senhas ou informações financeiras, não estão devidamente protegidos por algoritmos de criptografia adequados ou quando a criptografia é mal implementada. Isso pode permitir que atacantes interceptem ou modifiquem dados, comprometendo a privacidade e a segurança do sistema. Vulnerabilidades como o uso de criptografia fraca ou a ausência de criptografia em dados críticos tornam as informações vulneráveis a ataques como *man-in-the-middle*.

Usar protocolos e algoritmos de criptografia robustos e modernos, como TLS 1.3 para comunicações seguras, e AES-256 para criptografia simétrica de dados em repouso. Garantir que dados em trânsito e em repouso estejam sempre criptografados, e que chaves criptográficas sejam gerenciadas de forma segura, com mecanismos como rotação periódica de chaves e uso de módulos de segurança de hardware (HSM) para armazenamento seguro.

4.4.12.3 Injeção

A injeção, como a injeção de SQL, ocorre quando dados maliciosos fornecidos por um atacante são processados por um sistema sem a devida validação, permitindo que comandos inesperados sejam executados. Isso pode resultar na manipulação de consultas a banco de dados, execução de comandos no servidor ou até mesmo o controle total do sistema. Ataques de injeção são extremamente perigosos, pois podem comprometer diretamente a integridade e confidencialidade dos dados.

Utilizar consultas parametrizadas, que separam o código SQL dos dados fornecidos pelo usuário, impedindo a execução de comandos maliciosos. Além disso, toda entrada do usuário deve ser validada e sanitizada, removendo caracteres que possam ser interpretados como comandos. Ferramentas e frameworks de desenvolvimento modernos, como O'ORMs (Object-Relational Mapping), também ajudam a prevenir ataques de injeção.

4.4.12.4 Design inseguro

Aplicações projetadas sem considerar a segurança desde o início podem apresentar falhas fundamentais que são difíceis de corrigir posteriormente. O design inseguro envolve a ausência de padrões e práticas de segurança em fases críticas do desenvolvimento, como autenticação, gerenciamento de sessões e validação de dados. Isso pode abrir portas para várias vulnerabilidades, como falhas de controle de acesso e injeção.

Incorporar práticas de design seguro desde o início do projeto, aplicando princípios como defesa em profundidade, segmentação de responsabilidades e minimização de superfícies de ataque. Realizar revisões de segurança e análises de ameaças durante todo o ciclo de vida do software, utilizando técnicas como *Threat Modeling* para antecipar possíveis vetores de ataque e projetar contramedidas adequadas.

4.4.12.5 Configuração incorreta da segurança

Configurações padrão ou incorretas de segurança são uma causa comum de vulnerabilidades. Isso pode incluir o uso de senhas fracas ou padrões, servidores mal configurados, portas abertas e a exposição desnecessária de serviços internos. Esses problemas são frequentemente explorados por atacantes para obter acesso a sistemas ou informações confidenciais.

Revisar e aplicar configurações de segurança apropriadas para ambientes de produção, desativando serviços e funcionalidades que não são necessários. Ferramentas de automação podem ser usadas para garantir que as configurações de segurança estejam consistentemente aplicadas. Além disso, processos regulares de auditoria e revisão de configuração ajudam a identificar e corrigir possíveis problemas antes que sejam explorados.

4.4.12.6 Componentes vulneráveis e desatualizados

Muitas aplicações dependem de bibliotecas de terceiros, frameworks e componentes externos. Se esses componentes contêm vulnerabilidades conhecidas e não são atualizados regularmente, eles podem ser explorados por atacantes para comprometer o sistema. Isso é especialmente comum quando há dependência de bibliotecas não mantidas ou desatualizadas.

Manter um inventário atualizado de todos os componentes de software utilizados e garantir que estejam sempre atualizados com os últimos patches de segurança. Utilizar ferramentas de análise de dependências, como o OWASP Dependency-Check, para identificar e alertar sobre componentes com vulnerabilidades conhecidas. Além disso, realizar auditorias periódicas para garantir que componentes antigos e inseguros sejam removidos ou substituídos.

4.4.12.7 Falhas na identificação e autenticação

Essas falhas ocorrem quando mecanismos de autenticação e controle de sessão são implementados de forma inadequada, permitindo que atacantes se passem por usuários legítimos ou capturem sessões ativas. Isso pode resultar em acesso não autorizado a informações sensíveis ou em elevação de privilégios.

Implementar autenticação multifator (MFA) para adicionar uma camada extra de segurança na verificação de identidade. Garantir políticas robustas de senha, como a exigência de senhas fortes e a limitação de tentativas de login. Além disso, tokens seguros, como JSON Web Tokens (JWT), devem ser usados para gerenciamento de sessões, garantindo que essas sessões sejam protegidas contra sequestro.

4.4.12.8 Falhas na integridade de software e dados

A integridade de software e dados é comprometida quando não há mecanismos adequados para garantir que o código ou dados não foram alterados de forma maliciosa ou acidental. Ataques que exploram essa vulnerabilidade podem modificar o comportamento de aplicações ou comprometer a integridade dos dados, sem que haja detecção imediata.

Implementar assinaturas digitais e checksums para verificar a integridade de software e dados durante transferências e armazenamento. Utilizar ferramentas de verificação contínua para detectar alterações não autorizadas nos arquivos do sistema. Configurar alertas e auditorias para que qualquer alteração inesperada seja identificada e tratada rapidamente.

4.4.12.9 Falhas no registro de segurança e no monitoramento

A falta de monitoramento e de um sistema de registro adequado impede que as organizações detectam e respondem a ataques de forma eficiente. Sem logs detalhados e precisos, é difícil identificar tentativas de ataque ou atividades suspeitas, o que permite que invasores permaneçam dentro do sistema por longos períodos sem serem notados.

Implementar sistemas de registro de segurança abrangentes que registrem todas as atividades críticas, como tentativas de login, alterações de configuração e acesso a dados sensíveis. Os logs devem ser centralizados e

monitorados em tempo real para que atividades suspeitas possam ser detectadas rapidamente.

4.4.12.10 Falsificação de solicitação do lado do servidor (SSRF)

Essa vulnerabilidade permite que um atacante manipule uma aplicação web para fazer solicitações HTTP maliciosas a outros sistemas, geralmente internos, resultando em vazamento de dados ou execução de comandos não autorizados.

Validar todas as entradas que possam gerar solicitações HTTP. Restringir a capacidade da aplicação de se comunicar com destinos desconhecidos ou não confiáveis, utilizando listas de permissões para controlar quais servidores podem ser acessados. O monitoramento rigoroso de requisições HTTP também pode ajudar a detectar comportamentos anômalos.

4.4.12.11 DDOS

DDoS é um ataque em que o servidor é sobrecarregado por um grande volume de requisições, tornando-o indisponível para usuários legítimos. Esse tipo de ataque pode paralisar sistemas, causando perda de receita e danos à reputação da empresa.

Implementar *rate limiting*, que limita o número de requisições permitidas por usuário em um determinado período de tempo. Utilizar serviços de mitigação de DDoS, muitas vezes oferecidos por provedores de infraestrutura em nuvem, que podem filtrar tráfego malicioso antes que ele chegue ao servidor. Além disso,平衡adores de carga e firewalls podem ser configurados para distribuir o tráfego e minimizar os impactos do ataque.

4.5 Performance

A performance é um aspecto crucial na engenharia de software, pois está diretamente ligada à eficiência de um sistema em termos de tempo de resposta, uso de recursos e capacidade de escalabilidade. Um software com boa performance é aquele que consegue processar grandes volumes de dados ou realizar operações complexas de forma rápida, sem consumir recursos excessivos, como CPU, memória ou largura de banda. Em projetos de software, a otimização da performance é especialmente relevante em sistemas que necessitam lidar com alta demanda ou que requerem tempos de resposta mínimos.

Existem diversas métricas utilizadas para medir a performance de um software, como o tempo de execução de determinadas operações, a latência nas comunicações entre sistemas, o *throughput* e o uso de recursos. Essas métricas são analisadas para identificar gargalos de desempenho e otimizar o código ou a infraestrutura para melhorar a experiência do usuário e a eficiência do sistema.

A busca por melhor performance pode ocorrer em diferentes níveis, desde a escolha de algoritmos e estruturas de dados mais eficientes, até a otimização do banco de dados e da arquitetura de sistemas. Algumas abordagens comuns para melhorar a performance incluem identificação de gargalos, refatoração, caching, paralelismo, filas, balanceamento de carga e a utilização de sistemas distribuídos.

No entanto, a busca pela performance deve ser sempre equilibrada com outros aspectos, como a legibilidade do código e a facilidade de manutenção. É importante evitar a otimização prematura, que ocorre quando se tenta melhorar a performance antes de entender completamente as necessidades do sistema, o que pode resultar em aumento desnecessário de complexidade e dificuldade de manutenção. Assim, a prática recomendada é monitorar e testar o desempenho do software durante o desenvolvimento, otimizando somente quando problemas de performance reais forem identificados.

4.5.1 Over Engineering

Over Engineering, refere-se ao desenvolvimento de soluções excessivamente complexas ou sofisticadas para problemas que poderiam ser resolvidos de forma mais simples. Esse fenômeno é comumente encontrado em projetos de software, onde a preocupação em prever todas as possíveis exigências futuras leva à criação de sistemas mais robustos do que o necessário. Embora a intenção seja antecipar demandas e garantir a escalabilidade, o resultado muitas vezes é um aumento na complexidade, nos custos de manutenção e no tempo de desenvolvimento, sem benefícios proporcionais.

Um dos principais problemas da *Over Engineering* é o desperdício de recursos. Soluções demasiadamente elaboradas, além de demandarem mais tempo e esforço para serem desenvolvidas, tornam-se mais difíceis de manter e adaptar. Quanto mais complexo é um sistema, maior é o número de partes interdependentes que precisam ser gerenciadas e atualizadas, o que aumenta a probabilidade de surgimento de erros e a necessidade de correções posteriores.

Além disso, essa complexidade pode criar uma curva de aprendizado mais acentuada para novos desenvolvedores que venham a trabalhar no projeto, dificultando a continuidade do desenvolvimento e a manutenção da solução ao longo do tempo.

Outro problema relacionado à *Over Engineering* é a tendência de se perder o foco nos objetivos principais do projeto. Muitas vezes, ao tentar prever todos os possíveis cenários futuros, desenvolvedores acabam criando funcionalidades desnecessárias ou exagerando na modularização e abstração. Isso pode resultar em sistemas excessivamente genéricos ou em um excesso de camadas que, ao invés de agregar valor, complicam a implementação e o uso. Em cenários onde o tempo de desenvolvimento é um fator crítico, isso

pode ser particularmente prejudicial, pois a busca por uma solução "perfeita" pode atrasar a entrega de funcionalidades que resolvem o problema imediato do cliente ou usuário.

Para mitigar o risco de Over Engineering, é fundamental adotar uma abordagem que busque o equilíbrio entre a simplicidade e a capacidade de adaptação futura. O foco deve ser sempre em resolver o problema atual de maneira eficiente, sem tentar antecipar todas as possíveis necessidades futuras que podem ou não se concretizar.

Além disso, a abordagem iterativa no desenvolvimento de software, como praticada em metodologias ágeis, pode ajudar a evitar a Over Engineering. Através de ciclos curtos de desenvolvimento, é possível entregar incrementos funcionais do sistema que atendem às necessidades imediatas dos usuários, e fazer ajustes conforme novas demandas surgem. Essa prática de feedback contínuo permite que o sistema evolua de maneira controlada, sem a necessidade de antecipar e desenvolver soluções complexas que talvez nunca sejam necessárias.

Outra prática importante para evitar esse problema é a priorização de requisitos, através da qual as funcionalidades são desenvolvidas com base em seu valor de negócio imediato. Requisitos que possuem alto impacto para os usuários devem ser priorizados, enquanto aqueles que podem ou não ser necessários no futuro devem ser adiados. Esse foco em resolver as principais dores do cliente contribui para a entrega de valor constante e evita que recursos sejam desperdiçados em funcionalidades não essenciais.

Por fim, é importante também que as equipes de desenvolvimento e os stakeholders do projeto tenham um diálogo contínuo sobre as prioridades e os objetivos do software. Desenvolvedores devem evitar cair na armadilha de adicionar complexidade em busca de uma solução tecnicamente perfeita e, em vez disso, devem manter o foco nas necessidades reais do usuário. Decisões baseadas em dados, como a análise de métricas de uso e feedback de usuários, são fundamentais para direcionar o desenvolvimento de forma pragmática e evitar a criação de soluções superdimensionadas.

4.5.2 Métricas: Identificando gargalos

O uso de métricas é essencial para a identificação de gargalos em sistemas de software, permitindo que desenvolvedores e equipes técnicas tomem decisões informadas sobre otimização e melhoria de desempenho. Gargalos são pontos dentro de um sistema que limitam seu desempenho global, geralmente resultando em tempos de resposta mais lentos ou consumo excessivo de recursos. Detectá-los é crucial para garantir que o sistema funcione de maneira eficiente e atenda às necessidades dos usuários, especialmente em cenários de alta demanda.

Existem diferentes tipos de métricas que podem ser aplicadas para identificar gargalos, abrangendo desde o consumo de CPU e memória até o tempo de resposta de requisições e o uso de banco de dados. Ferramentas de

análise e monitoração automatizadas, como benchmarks, o SonarLint e outras, desempenham um papel fundamental nesse processo.

Os benchmarks são testes padronizados utilizados para medir o desempenho de sistemas de software. Eles simulam diferentes cargas de trabalho e permitem comparar o comportamento de um sistema sob diferentes condições, ajudando a identificar gargalos em áreas como processamento, memória ou desempenho de rede, esses testes podem ser feitos em cada rota da API, para identificar qual rota está com o tempo de processamento indesejado.

Por exemplo, ao realizar um *benchmark* em uma aplicação web, pode-se observar o tempo de resposta do servidor em diferentes cenários, como durante uma alta quantidade de acessos simultâneos ou consultas complexas no banco de dados. Se o tempo de resposta aumentar significativamente durante esses testes, isso pode indicar um gargalo no processamento de requisições ou na interação com o banco de dados, direcionando a equipe para investigar otimizações nesses pontos.

Benchmarks fornecem *insights* quantitativos que permitem tomar decisões embasadas, como a necessidade de adicionar recursos de hardware, otimizar consultas ao banco de dados, refatoração de código ou ajustar configurações de infraestrutura, como平衡amento de carga ou cache.

SonarLint é uma ferramenta de análise estática de código que auxilia desenvolvedores na identificação de problemas que podem impactar a qualidade e o desempenho de uma aplicação. Diferentemente dos *benchmarks*, que se concentram na medição do desempenho em tempo de execução, o SonarLint é utilizado durante o processo de desenvolvimento para garantir que o código esteja livre de erros, vulnerabilidades e potenciais gargalos.

O ele verifica o código em tempo real enquanto ele está sendo escrito, alertando sobre práticas inefficientes que podem levar a gargalos, como loops desnecessários, uso inadequado de recursos ou consultas SQL mal otimizadas. Ao detectar essas práticas e sugerir melhorias, a ferramenta ajuda a evitar problemas de desempenho antes que eles se manifestem no ambiente de produção.

Além de detectar problemas relacionados à performance, ele também alerta sobre vulnerabilidades de segurança, conformidade com padrões de codificação e manutenção do código. Isso garante que o sistema não apenas tenha um bom desempenho, mas também seja seguro e sustentável a longo prazo.

As métricas de desempenho desempenham um papel fundamental na identificação e resolução de gargalos em sistemas de software. Ferramentas como *benchmarks* ajudam a medir o comportamento do sistema sob diferentes cargas, enquanto o SonarLint auxilia na detecção precoce de problemas no código. Com a aplicação dessas práticas, as equipes de desenvolvimento podem melhorar a performance do software, garantindo maior eficiência, escalabilidade e uma melhor experiência para os usuários finais.

4.5.3 Refatoração

Em seu livro *Refatoração: aperfeiçoando o design de códigos existentes*, Fowler (2019), define refatoração como um processo de modificar um sistema de software de modo que não altere o comportamento externo do código, embora melhore a sua estrutura interna. É uma maneira disciplinada de reorganizar o código, minimizando as chances de introduzir bugs. Em sua essência, ao refatorar, você está aperfeiçoando o design do código depois que ele foi escrito.

Assim, a refatoração não envolve a adição de novas funcionalidades, mas sim a melhoria contínua do código existente para torná-lo mais legível, comprehensível e de fácil manutenção. Isso é essencial em projetos de software de longo prazo, onde o acúmulo de "dívida técnica" – código de baixa qualidade escrito às pressas ou sem uma arquitetura sólida – pode levar a problemas de manutenção, dificuldade em adicionar novas funcionalidades e, eventualmente, um aumento no custo de evolução do sistema.

A prática de refatoração é baseada em pequenas alterações incrementais que garantem que o sistema permaneça funcional a cada passo. Isso é normalmente feito em conjunto com testes automatizados, que garantem que o comportamento esperado do software continue o mesmo, mesmo após as mudanças no código. Isso permite que os desenvolvedores façam modificações com mais segurança, sabendo que eventuais erros serão detectados rapidamente.

Existem várias técnicas de refatoração descritas por Fowler, incluindo a extração de métodos (onde trechos de código duplicados são extraídos para métodos reutilizáveis), renomeação de variáveis e métodos para nomes mais descriptivos, e a eliminação de código morto ou desnecessário. Essas práticas ajudam a evitar a repetição de código, promovem a reutilização e facilitam o entendimento do sistema por novos desenvolvedores que se juntarem ao projeto.

Outro benefício importante da refatoração é a melhoria do design do software. Através da refatoração contínua, padrões de projeto podem ser aplicados corretamente e estruturas mais simples podem ser implementadas, resultando em um código que é mais flexível e mais preparado para mudanças futuras. Um código bem refatorado também facilita a detecção e correção de bugs, pois sua estrutura mais clara e organizada permite um entendimento mais profundo e rápido do sistema.

Em um cenário de desenvolvimento ágil, a refatoração é uma prática essencial. A abordagem ágil valoriza entregas rápidas e incrementais, e, sem refatoração, o código poderia rapidamente se tornar difícil de gerenciar e expandir. Com refatoração constante, a equipe de desenvolvimento mantém o código limpo, permitindo que o software continue evoluindo de forma eficiente e sustentável ao longo do tempo.

4.5.4 Balanceador de carga

O balanceamento de carga é uma técnica essencial para otimização de sistemas de software, especialmente em aplicações web e distribuídas que precisam lidar com grandes volumes de tráfego. Sua função principal é distribuir de forma eficiente as requisições recebidas por um servidor entre várias máquinas, evitando a sobrecarga de um único servidor e garantindo que os recursos do sistema sejam utilizados de maneira eficaz. Nesse contexto, o Nginx se destaca como uma das ferramentas mais amplamente utilizadas, oferecendo funcionalidades tanto de proxy reverso quanto de distribuição de carga, contribuindo para a melhoria da disponibilidade, escalabilidade e segurança de aplicações web.

Um平衡ador de carga distribui as requisições de rede ou tráfego entre diversos servidores de backend. Quando um único servidor recebe muitas requisições simultâneas, ele pode se sobrecarregar, resultando em lentidão ou até falhas no sistema. O balanceador de carga evita esses problemas, distribuindo o tráfego de maneira uniforme entre todos os servidores disponíveis, o que melhora o desempenho e assegura alta disponibilidade para a aplicação. Existem diferentes algoritmos de balanceamento, como Round Robin (requisições distribuídas de maneira sequencial entre os servidores), Least Connections (nova requisição é direcionada ao servidor com menos conexões ativas) e IP Hash (distribui com base no endereço IP do cliente).

O Nginx, além de ser um servidor web e proxy reverso, oferece recursos avançados de balanceamento de carga, sendo capaz de lidar com grandes volumes de tráfego, o que o torna popular em arquiteturas de microservices e grandes aplicações web. Ele distribui as requisições de maneira transparente entre múltiplos servidores, permitindo que as aplicações escalem horizontalmente, adicionando mais servidores ao invés de aumentar a capacidade de um único. Além disso, o Nginx monitora os servidores e retira automaticamente da lista de distribuição aqueles que estão inoperantes, garantindo que apenas servidores ativos recebam tráfego, o que aumenta a resiliência do sistema.

O balanceamento de carga também requer um equilíbrio adequado entre performance e custo. Configurações inadequadas podem resultar em desperdício de recursos ou aumentar a complexidade de manutenção do sistema. Contudo, o Nginx oferece flexibilidade suficiente para que as equipes de desenvolvimento e operações ajustem suas configurações de acordo com as necessidades específicas de desempenho e orçamento.

4.5.5 Cache

O cache é uma técnica amplamente utilizada para otimização de performance em sistemas de software. Como Newman define em seu livro Criando Microsserviços (2022):

O caching é uma otimização de desempenho comum, na qual o resultado anterior de uma operação é armazenado de modo que a requisições subsequentes possam utilizar esse valor armazenado, em vez de gastar tempo e recursos recalcando o valor.

Dessa forma, o cache reduz o tempo de resposta das aplicações e melhora a escalabilidade do sistema.

Há diferentes tipos de cache que podem ser aplicados conforme as necessidades da arquitetura do sistema. O cache de navegador, por exemplo, armazena arquivos estáticos, como imagens, folhas de estilo (CSS) e scripts (JavaScript), diretamente no dispositivo do usuário. Isso permite que esses recursos sejam carregados rapidamente em visitas subsequentes ao site, sem novas requisições ao servidor. Já o cache de aplicação é usado para armazenar dados frequentemente acessados diretamente na aplicação, aproveitando serviços como Redis para otimizar o tempo de resposta de consultas ao banco de dados.

Além disso, o cache de servidor também pode ser utilizado, como no caso de servidores web, que armazenam respostas HTTP inteiras ou parciais para reutilização em requisições subsequentes. Essa estratégia é altamente eficiente para reduzir a carga em servidores de backend. Por fim, o cache de CDN (Content Delivery Network), bastante utilizado em grandes sistemas distribuídos, armazena e serve recursos estáticos a partir de servidores localizados geograficamente mais próximos do usuário, minimizando a latência e otimizando o desempenho da aplicação.

Os principais benefícios do uso de cache incluem a redução da latência, o aumento da escalabilidade e a diminuição do tráfego de rede. No entanto, o cache também apresenta desafios, especialmente em relação à consistência dos dados, já que as informações armazenadas no cache podem se desatualizar em sistemas que possuem atualizações frequentes. Para mitigar esse problema, são utilizadas políticas de expiração ou invalidação do cache, determinando quando os dados devem ser atualizados ou removidos.

Outro desafio comum ao uso de cache é o fenômeno conhecido como cache stampede, que ocorre quando um cache muito acessado se esgota, e várias requisições simultâneas tentam reconstruir o dado em questão, sobrecarregando o sistema. Para evitar esse problema, é essencial implementar estratégias de gerenciamento adequado do cache, monitorando o uso e configurando políticas que garantam a disponibilidade e a consistência dos dados armazenados.

O uso de cache é uma estratégia poderosa para melhorar a performance e escalabilidade de sistemas de software. No entanto, é crucial que sua implementação seja feita de forma estratégica, levando em conta a consistência dos dados e o controle de sobrecarga dos recursos do sistema. Quando bem utilizado, o cache pode proporcionar uma experiência de usuário significativamente melhor, com respostas mais rápidas e utilização eficiente dos recursos computacionais.

4.5.6 CQRS

O CQRS (Command Query Responsibility Segregation) é um padrão arquitetural que separa as operações de leitura (Query) e escrita (Command) em sistemas de software, promovendo uma abordagem mais eficiente e escalável para lidar com grandes volumes de dados e requisitos complexos de processamento. Como destaca Sam Newman em seu livro Criando Microsserviços o CQRS é um modelo alternativo para armazenar e consultar informações, em vez de termos um único modelo tanto para manipular como para consultar os dados, como é comum, as responsabilidades pelas leituras e escritas são tratadas por modelos distintos, esses modelos de leitura e escrita separados, implementados no código, poderiam ser implementados como unidade distintas, o que nos dá a capacidade de escalar as leituras e escritas de modo independente.

No CQRS, as operações de Command (comando) envolvem a modificação do estado do sistema. Elas são responsáveis por executar mudanças, como criação, atualização ou exclusão de dados. Por outro lado, as operações de Query (consulta) são utilizadas apenas para ler dados do sistema, sem modificá-los. Essa separação permite que as operações de leitura e escrita sejam otimizadas individualmente, podendo até usar diferentes modelos de dados ou bancos de dados para cada tipo de operação.

Por exemplo, para operações de leitura que exigem alta performance e grande volume de acessos, pode-se usar um banco de dados otimizado para consultas rápidas, como uma solução NoSQL. Já para operações de escrita, onde é necessário garantir consistência transacional e integridade dos dados, pode-se adotar um banco de dados relacional ou uma abordagem que melhor se adeque às exigências do sistema.

Apesar de suas vantagens, o CQRS também apresenta desafios que devem ser considerados. Um dos principais é a complexidade adicional, especialmente ao implementar uma arquitetura baseada em eventos, que muitas vezes é associada ao CQRS para garantir que os estados de leitura e escrita permaneçam sincronizados. Quando o estado é atualizado através de eventos, pode haver um problema de consistência eventual, onde os dados de leitura e escrita podem estar temporariamente desatualizados até que os eventos sejam totalmente processados.

Além disso, a implementação de CQRS geralmente requer um maior esforço em termos de infraestrutura, já que pode envolver a criação e manutenção de *pipelines* de eventos, sistemas de mensageria, e múltiplos bancos de dados, o que aumenta a complexidade do desenvolvimento e da operação.

Para garantir que o uso de CQRS traga mais benefícios do que desafios, é essencial identificar corretamente os casos de uso. Em sistemas onde há um grande volume de consultas e baixa frequência de alterações, o CQRS é particularmente eficaz. Por outro lado, em sistemas simples ou com baixa demanda, o uso do CQRS pode ser desnecessariamente complexo.

Uma estratégia comum é começar com uma arquitetura tradicional e,

à medida que o sistema evolui e a demanda por escalabilidade aumenta, aplicar o CQRS apenas em áreas que realmente exigem essa separação. Isso permite evitar o problema de over-engineering, onde a aplicação de um padrão complexo como CQRS seria desproporcional às necessidades do sistema.

4.5.7 Filas assíncronas

No desenvolvimento de sistemas modernos, as filas assíncronas é uma solução arquitetural amplamente utilizada para lidar com tarefas que podem ser processadas de maneira não imediata, melhorando a performance e a escalabilidade de aplicações. Ao invés de processar certas operações diretamente durante o fluxo principal do sistema, as tarefas são enviadas para uma fila, onde serão executadas em segundo plano por um ou mais trabalhadores (workers) que as processam de forma assíncrona.

Uma fila assíncrona é uma estrutura que permite que tarefas complexas, demoradas ou de alta carga sejam processadas fora do fluxo principal de execução da aplicação. O objetivo principal é evitar que o tempo de resposta ao usuário final seja prejudicado por operações pesadas, como envio de emails, processamento de imagens, integração com APIs externas ou cálculos intensivos.

Esse tipo de arquitetura permite que a aplicação continue respondendo a novas requisições enquanto as tarefas são executadas em segundo plano. Ao delegar essas tarefas para uma fila, o sistema ganha maior eficiência, distribuindo a carga de trabalho entre múltiplos workers, que podem ser escalados conforme necessário.

Embora o uso de filas para jobs assíncronos traga muitos benefícios, é importante que o desenvolvimento e a configuração dessas filas sejam feitos de maneira equilibrada. Um erro comum é o uso excessivo de filas, colocando tarefas simples e rápidas em segundo plano, quando poderiam ser executadas no fluxo principal sem impacto significativo. Além disso, é importante monitorar o desempenho das filas e workers, garantindo que as tarefas sejam processadas dentro de um tempo aceitável, e que as filas não se tornem gargalos no sistema. Além de gerar mais complexidade para o projeto, então é necessário refletir se vale a pena.

4.5.8 Pooling de conexões

O *pooling* de conexões é uma técnica amplamente utilizada para gerenciar e otimizar o uso de conexões com bancos de dados ou outros serviços externos em uma aplicação. Em vez de abrir uma nova conexão a cada requisição — o que é um processo caro em termos de tempo e recursos — o *pooling* mantém um conjunto (ou "*pool*") de conexões pré-estabelecidas que podem ser reutilizadas sempre que necessário. Isso reduz significativamente a sobrecarga de criar e destruir conexões repetidamente, melhorando o desem-

penho e a eficiência da aplicação, logo “nunca” ou pelo menos quando possível, não utilize loops para fazer requisições ao banco de dados.

Quando uma aplicação precisa interagir com o banco de dados ou outro serviço que exige uma conexão persistente, como um servidor de e-mail ou um sistema de fila, ela pode requisitar uma conexão do pool. Se houver uma conexão disponível no pool, a aplicação a utiliza diretamente. Quando a tarefa é concluída, em vez de fechar a conexão, ela é retornada ao pool, tornando-se disponível para outras requisições.

Se todas as conexões do pool estiverem em uso, a aplicação pode aguardar até que uma conexão seja liberada ou, dependendo da configuração do pool, criar uma nova conexão. Esse comportamento é controlado por parâmetros como o número máximo e mínimo de conexões, o tempo de espera para novas conexões e o tempo máximo que uma conexão pode ficar ociosa.

Os benefícios do pooling de conexões são notáveis para o desempenho e eficiência de uma aplicação. Primeiramente, ele reduz a latência, pois as conexões são mantidas ativas no pool, eliminando o tempo necessário para estabelecer novas conexões a cada requisição. Isso resulta em respostas mais rápidas e na otimização da comunicação com bancos de dados ou serviços externos.

Além disso, o pooling melhora a utilização de recursos, uma vez que a criação e o encerramento frequentes de conexões consomem recursos consideráveis. Com o pooling, o número de conexões abertas e fechadas é reduzido, liberando tanto o cliente quanto o servidor dessa sobrecarga. Outro aspecto importante é a escalabilidade, permitindo que a aplicação suporta múltiplas requisições simultâneas de maneira mais eficiente, sem sobrecarregar o servidor com a criação de novas conexões para cada operação.

O gerenciamento de conexões ociosas também é uma vantagem significativa, já que o pool pode ser configurado para encerrar conexões que estão inativas por muito tempo ou restabelecê-las quando necessário, evitando o desperdício de recursos.

Para obter o máximo desempenho do pooling de conexões, algumas configurações comuns precisam ser ajustadas adequadamente. O tamanho do pool, por exemplo, define o número máximo de conexões que podem estar ativas simultaneamente. Um pool muito pequeno pode causar gargalos em cenários de alta demanda, enquanto um pool muito grande pode desperdiçar recursos do sistema. Outro parâmetro importante é o tempo de vida das conexões, que determina quanto tempo uma conexão pode permanecer ativa no pool antes de ser encerrada e substituída por uma nova. Isso garante que as conexões não fiquem obsoletas. Além disso, o timeout de requisição específica quanto tempo a aplicação deve esperar por uma conexão do pool antes de gerar um erro ou tentar criar uma nova conexão, garantindo que a aplicação não fique bloqueada por tempo excessivo quando o pool estiver cheio.

O pooling de conexões é amplamente utilizado em servidores de apli-

ções que fazem consultas frequentes a bancos de dados. Por exemplo, em uma aplicação Node.js usando o PostgreSQL, bibliotecas como pg-pool gerenciam automaticamente o pool de conexões.

4.5.9 Indexação de Banco de Dados

A indexação de banco de dados é uma técnica essencial para otimizar o desempenho das consultas, especialmente em sistemas que manipulam grandes volumes de dados. O objetivo da indexação é criar estruturas auxiliares que permitam ao banco de dados localizar os registros desejados de forma rápida, sem precisar realizar uma varredura completa da tabela. Assim, ela atua como um índice em um livro, facilitando a busca por informações específicas sem que seja necessário percorrer todo o conteúdo.

A IBM explica que a indexação pode aumentar significativamente a velocidade da procura. No entanto, a desvantagem dos índices é que cada operação de inserção, atualização ou exclusão requer uma atualização dos índices. Quando as tabelas incluem diversos índices, cada índice pode aumentar o tempo necessário para processar as atualizações de tabelas. Assim, se você deseja reduzir o número de índices para melhorar a velocidade de processamento, remova os índices que forem menos úteis para fins de procura. Portanto a indexação no banco de dados, deve ser com cuidado, pensada e analisada, para ter a certeza que o custo benefício será satisfatório.

Os benefícios da indexação são diversos. Um dos principais é a melhoria no desempenho das consultas, especialmente em grandes volumes de dados. Consultas que envolvem colunas não indexadas, ou aquelas que precisam filtrar e buscar em grandes conjuntos de dados, são significativamente aceleradas quando um índice adequado é utilizado. Outro benefício é a eficiência no filtramento de dados, em que filtros de consultas, como WHERE ou JOIN, são processados mais rapidamente ao utilizar índices nas colunas relevantes. Além disso, a indexação facilita operações de ordenação e otimizar consultas que utilizam funções de agregação como COUNT, AVG, MAX e MIN.

No entanto, a indexação também traz desafios e custos associados. Cada índice ocupa espaço em disco, e quanto maior for o número de índices em uma tabela, maior será o consumo de armazenamento. Além disso, operações de escrita (inserções, atualizações e deleções) podem se tornar mais lentas, pois o banco de dados precisa ajustar todos os índices relevantes após cada modificação. Um índice mal configurado pode até piorar o desempenho, se não estiver alinhado aos padrões de consulta mais frequentes.

A utilização de índices é mais eficaz em colunas que são frequentemente usadas em cláusulas WHERE, GROUP BY ou ORDER BY, e que participam de junções em consultas. Por outro lado, o excesso de índices deve ser evitado, já que o *overhead* nas operações de escrita pode comprometer o desempenho da aplicação.

Em termos de boas práticas, é recomendado indexar colunas utilizadas em

filtros e junções e monitorar continuamente a eficácia dos índices, utilizando ferramentas como o comando EXPLAIN em sistemas como MySQL ou PostgreSQL. Além disso, é importante revisar periodicamente os índices para ajustá-los conforme os padrões de acesso aos dados mudem ao longo do tempo.

4.5.10 Multithreading

O conceito de multithreading refere-se à capacidade de um programa de computador em executar várias threads simultaneamente dentro de um único processo. Uma thread é a menor unidade de processamento que pode ser agendada pelo sistema operacional, e cada thread possui seu próprio fluxo de execução, permitindo que um aplicativo realize múltiplas tarefas ao mesmo tempo. O seu uso é fundamental em diversas aplicações, especialmente aquelas que requerem alta performance e eficiência, como servidores web, aplicativos interativos e processamento de dados em larga escala.

A principal vantagem reside na sua capacidade de melhorar a responsividade de aplicações. Em ambientes interativos, como aplicativos de desktop ou interfaces gráficas, a execução de tarefas longas, como downloads de arquivos ou cálculos complexos, pode causar o “congelamento” da interface do usuário. Com o multithreading, essas tarefas podem ser executadas em segundo plano, permitindo que a interface permaneça responsiva e acessível ao usuário.

Outra vantagem significativa é a eficiência no uso de recursos do sistema. Em sistemas com múltiplos núcleos de processador, as threads podem ser distribuídas entre os núcleos disponíveis, permitindo a execução simultânea de várias tarefas. Isso resulta em um aumento significativo na utilização da CPU e, consequentemente, na performance do aplicativo. Além disso, o compartilhamento de memória entre threads de um mesmo processo reduz o overhead em comparação com a criação de processos separados, que necessitam de espaço de memória próprio.

Entretanto, o desenvolvimento de software com multithreading apresenta desafios únicos. A sincronização é um dos principais problemas que os desenvolvedores enfrentam ao implementar multithreading. Como várias threads podem acessar e modificar os mesmos dados simultaneamente, é essencial garantir que essas operações sejam gerenciadas adequadamente para evitar condições de corrida, onde o resultado das operações depende da ordem em que as threads são executadas. Ferramentas como *mutexes* (exclusão mútua) e semáforos são frequentemente utilizadas para controlar o acesso a recursos compartilhados, garantindo a integridade dos dados.

Além disso, a depuração de aplicativos *multithread* pode ser complexa. Erros podem não ocorrer de maneira consistente devido à natureza não determinística da execução das *threads*, tornando a identificação e correção de *bugs* mais desafiadora. Ferramentas especializadas e técnicas de teste são necessárias para monitorar o comportamento das threads e garantir que o sis-

tema funcione como esperado.

Em termos de aplicações práticas, o *multithreading* é amplamente utilizado em servidores web, onde cada requisição do cliente pode ser processada por uma *thread* separada. Isso permite que o servidor atenda a múltiplas requisições simultaneamente, melhorando a capacidade de resposta e a escalabilidade do sistema. Em outras áreas, como processamento de imagens ou vídeos, o *multithreading* permite que diferentes partes do arquivo sejam processadas ao mesmo tempo, acelerando o tempo total de processamento.

Em resumo, o *multithreading* é uma técnica poderosa e essencial no desenvolvimento de software, permitindo a execução simultânea de tarefas e melhorando a responsividade e a eficiência das aplicações. Embora traga desafios em termos de sincronização e depuração, o seu uso é fundamental para a criação de sistemas modernos que atendem às crescentes demandas de desempenho e escalabilidade. A adoção de práticas adequadas e ferramentas de desenvolvimento pode mitigar os problemas associados ao *multithreading*, permitindo que os desenvolvedores aproveitem ao máximo essa abordagem em suas aplicações.

4.6 MIGRANDO PARA MICROSERVIÇOS

A migração de uma arquitetura monolítica para microsserviços é um processo que muitas empresas e equipes de desenvolvimento adotam para lidar com as crescentes necessidades de escalabilidade, flexibilidade e manutenção em seus sistemas. Microsserviços são pequenos serviços independentes que funcionam de forma autônoma e que se comunicam entre si por meio de APIs, geralmente utilizando protocolos como HTTP ou mensageria. Essa arquitetura oferece uma série de vantagens, mas também apresenta desafios que devem ser cuidadosamente gerenciados.

4.6.1 Porquê microsserviços

No capítulo 2 do seu livro *Migrando sistemas monolíticos para microsserviços*, Newman (2020) aborda diversas motivações para você utilizar a arquitetura baseada em microsserviços, como equipes autônomas, lançamentos individuais, escalabilidade independente de serviços, diversidade de tecnologias e ferramentas, além do aumento de robustez. Mas apesar de ele dar todas essas motivações, ele também traz possíveis formas de você ter uma parte desses benefícios sem a utilização de microsserviços, portanto, uma análise crítica e colocar na balança se é realmente necessário a utilização da arquitetura de microsserviços, pois apesar de suas grandes vantagens, vêm enormes desafios.

Com microsserviços, cada serviço é desenvolvido, testado e implantado de forma independente. Isso permite que equipes menores e mais especializadas trabalhem em diferentes partes do sistema simultaneamente,

sem a necessidade de coordenar todas as mudanças em um único código-fonte. Além disso, cada serviço pode ser escalado individualmente, de acordo com sua necessidade de uso, resultando em uma utilização mais eficiente de recursos.

Outra vantagem é a possibilidade de utilizar diferentes tecnologias em cada microserviço. Em uma arquitetura monolítica, a escolha de tecnologia é limitada por toda a aplicação, mas em uma arquitetura de microsserviços, cada equipe pode escolher a linguagem de programação, framework e banco de dados mais adequados para o problema que estão resolvendo.

Migrar para microsserviços, no entanto, não é uma tarefa simples. É um processo complexo que envolve uma série de decisões técnicas e organizacionais. Um dos maiores desafios é a gestão da comunicação entre os serviços. Diferente de uma arquitetura monolítica, onde a comunicação entre os módulos acontece dentro do mesmo espaço de memória, os microsserviços precisam de mecanismos de comunicação interprocesso, o que pode introduzir latência e complexidade no gerenciamento de falhas.

4.6.2 Padrão de migração: aplicação Strangler Fig

O padrão Strangler Fig é uma abordagem amplamente utilizada para a migração de sistemas legados para novas arquiteturas, como microsserviços. Fowler (2024) inspirado pela árvore estranguladora, que cresce ao redor de outra árvore até substituí-la completamente, o padrão permite que novos componentes sejam implementados de forma incremental, substituindo gradualmente partes de um sistema legado sem a necessidade de uma reformulação abrupta. Essa estratégia de migração minimiza riscos e interrupções, garantindo que a aplicação continue a operar enquanto partes específicas são modernizadas.

No processo de migração usando o padrão Strangler Fig, a primeira etapa envolve identificar funcionalidades ou componentes isolados do sistema legado que podem ser extraídos e substituídos por novos serviços ou módulos. Em seguida, o tráfego relacionado a essas funcionalidades é redirecionado para os novos componentes, permitindo que o sistema legado continue a lidar com o restante das operações. Esse processo é repetido até que todas as partes críticas tenham sido substituídas, permitindo que o sistema legado seja eventualmente desativado.

Uma das principais vantagens do padrão Strangler Fig é a capacidade de realizar a migração de maneira incremental, o que reduz o impacto no usuário final e distribui os riscos ao longo do tempo. Além disso, ele oferece a flexibilidade de melhorar o sistema de maneira contínua, implementando novas funcionalidades diretamente na nova arquitetura enquanto o sistema legado ainda está em uso. Isso é especialmente útil em ambientes de produção, onde a migração abrupta pode ser arriscada.

No contexto da migração de um monólito para microsserviços, o Strangler Fig é uma ferramenta valiosa. Um exemplo prático seria a migração

de um módulo de faturamento de uma aplicação monolítica para um microserviço independente. Nesse caso, o novo microserviço é desenvolvido, e as requisições de faturamento são redirecionadas do monolito para o novo serviço. À medida que outras funcionalidades são migradas de maneira semelhante, o código legado referente a essas funcionalidades pode ser gradualmente removido do monolito.

Entretanto, embora o padrão ofereça uma abordagem segura e controlada, é importante gerenciar corretamente a coexistência entre o sistema legado e os novos componentes durante o processo de migração. A integração entre as partes, a complexidade do roteamento de tráfego e o monitoramento contínuo são fatores críticos para o sucesso dessa estratégia. Dessa forma, o padrão Strangler Fig se apresenta como uma solução eficiente para a modernização de sistemas legados, oferecendo uma transição mais segura e gradual para novas arquiteturas, como microsserviços.

4.6.3 Decomposição do banco de dados

A decomposição do banco de dados é uma prática essencial quando se realiza a migração de uma arquitetura monolítica para uma baseada em microsserviços ou durante a otimização de sistemas de grande escala. Pois, como Newman diz em seu livro *Migrando sistemas monolíticos para microsserviços* (2020):

Os microsserviços funcionam melhor quando trabalhamos ocultando informações, o que, por sua vez, nos leva geralmente em direção a microsserviços que encapsulam totalmente suas próprias armazenagens de dados e os métodos para acessá-los. Isso nos leva a conclusão que, ao migrarmos para uma arquitetura de microsserviços, devemos separar o banco de dados de nosso sistema monolítico se quisermos tirar o máximo proveito da transição.

Em sistemas monolíticos, é comum que o banco de dados seja centralizado, sendo utilizado por diversas partes do sistema, o que pode gerar gargalos em termos de desempenho, escalabilidade e manutenção. A decomposição do banco de dados busca distribuir as responsabilidades de armazenamento entre diferentes serviços ou módulos, permitindo que cada um tenha seu próprio banco ou esquema de dados especializado.

Esse processo de decomposição envolve dividir o banco de dados monolítico em unidades menores e mais coesas, alinhadas com os domínios de negócios. Dessa forma, em vez de um único banco de dados gerenciar todos os dados da aplicação, cada microserviço ou componente do sistema assume o controle sobre seu conjunto de dados, respeitando o princípio de "*Separation of Concerns*" (Separação de Preocupações). Isso melhora a modularidade e a coesão do sistema, já que cada serviço pode gerenciar seus dados de forma independente, evitando dependências que aumentem o acoplamento entre diferentes partes da aplicação.

Entre os benefícios dessa prática, destaca-se a escalabilidade, que permite que cada serviço escale de forma independente, sem impactar o desempenho de outras áreas da aplicação. Isso é crucial em sistemas que pre-

cisam lidar com grandes volumes de dados ou altos picos de tráfego. A manutenção também é melhorada, já que a modularização dos dados facilita a manutenção e a evolução do sistema, reduzindo a complexidade para as equipes de desenvolvimento, que podem focar em seus respectivos domínios. Além disso, a decomposição aumenta a resiliência do sistema, uma vez que uma falha em um serviço ou banco de dados específico afeta apenas aquela parte do sistema, permitindo que as demais áreas continuem operando normalmente.

No entanto, a decomposição do banco de dados também traz desafios, especialmente em termos de consistência de dados. Quando os dados são distribuídos entre diferentes microsserviços, a coordenação de transações distribuídas se torna mais complexa. Técnicas como Eventual Consistency e padrões de Sagas são frequentemente utilizadas para garantir que as operações entre serviços mantenham a consistência dos dados. Além disso, a decomposição pode aumentar a latência, já que os dados agora estão espalhados em diferentes locais, exigindo mecanismos eficientes de comunicação, caching e mensageria para minimizar o impacto no desempenho.

Portanto, a decomposição do banco de dados é uma prática crucial na transição para arquiteturas escaláveis e modulares, permitindo que sistemas evoluam de forma mais ágil, resiliente e preparada para as demandas de desempenho e escalabilidade das arquiteturas modernas. Quando realizada com um planejamento cuidadoso, ela contribui significativamente para a melhoria na manutenção, escalabilidade e robustez do sistema.

4.6.4 Observabilidade

A observabilidade é um conceito fundamental em sistemas de software modernos, especialmente em arquiteturas distribuídas e baseadas em microsserviços. Ela se refere à capacidade de entender o estado interno de um sistema a partir de seus dados de saída, como logs, métricas e traços. A implementação de observabilidade é essencial para garantir a detecção rápida de problemas, monitorar a saúde do sistema e otimizar seu desempenho, proporcionando uma visão clara do comportamento da aplicação, como Newman declara no livro Criando Microsserviços (2022):

Quando um problema ocorre, precisamos descobrir o que podemos fazer para que o sistema volte a funcionar novamente; assim que a poeira baixar, queremos ter informações suficientes à disposição para descobrir o que, afinal de contas, houve de errado, e o que poderá ser feito para evitar que esse problema ocorra novamente.

Então, precisamos ter informações necessárias para resolver o problema da forma correta, para isso Newman traz algumas abordagens de como podemos melhorar a observabilidade do nosso sistema com os principais componentes da observabilidade que incluem agregação de logs, agregação de métricas, tracing distribuído e alertas.

A agregação de logs consiste na centralização dos logs gerados por

diferentes partes do sistema, detectar padrões de erro e identificar problemas que afetam o desempenho ou a disponibilidade da aplicação.

Ferramentas como o ELK Stack (Elasticsearch, Logstash, Kibana) ou o Loki são amplamente utilizadas para coletar, armazenar e visualizar logs. Elas permitem que logs de várias fontes sejam unificados em um só lugar e pesquisados por meio de consultas sofisticadas, facilitando a análise em tempo real e a detecção de falhas.

A agregação de métricas envolve a coleta e monitoramento de dados quantitativos que refletem o comportamento do sistema, como uso de CPU, memória, latência de requisições, número de requisições por segundo, entre outros. As métricas fornecem uma visão clara do desempenho do sistema e são essenciais para identificar gargalos, sobrecarga de recursos e outros problemas de desempenho. Ferramentas como Prometheus e Grafana são populares para coleta, agregação e visualização de métricas. Prometheus, por exemplo, coleta métricas de serviços em intervalos regulares, permitindo que os desenvolvedores monitorem tendências e definem limites para criar alertas automáticos quando os valores excedem os limites aceitáveis. Essas métricas ajudam a manter a saúde do sistema em nível operacional e permitem uma resposta proativa a possíveis falhas.

O tracing distribuído é uma técnica que permite rastrear e monitorar requisições que percorrem diferentes serviços dentro de uma arquitetura distribuída. Ele é especialmente útil para diagnosticar problemas em sistemas de microserviços, onde uma única requisição pode passar por vários componentes, dificultando a identificação do ponto de falha ou do gargalo de desempenho.

Ferramentas como Jaeger e Zipkin são utilizadas para implementar tracing distribuído. Essas ferramentas inserem identificadores exclusivos em cada requisição, possibilitando o rastreamento de sua jornada por diferentes serviços. Isso ajuda a identificar pontos de latência, falhas em interações entre serviços e melhorar o desempenho geral do sistema, oferecendo uma visão detalhada de como as requisições fluem no ambiente distribuído.

Os alertas são fundamentais para a detecção proativa de problemas em um sistema de software. Eles são configurados com base em métricas, logs ou eventos de tracing distribuídos, e são acionados quando determinados limiares são excedidos, indicando anomalias ou falhas. A implementação de alertas permite que as equipes de desenvolvimento e operações sejam notificadas rapidamente sobre problemas críticos, como indisponibilidade de serviços, degradação de desempenho ou falhas de segurança.

Os alertas podem ser configurados para diferentes tipos de eventos e enviados por meio de diversos canais, como e-mail, Slack ou sistemas de incidentes como PagerDuty. Um bom sistema de alertas garante que as notificações sejam precisas e acionáveis, evitando alertas falsos que podem gerar "*alert fatigue*" e desviar o foco da equipe.

A observabilidade, ao integrar a agregação de logs, métricas, tracing distribuído e alertas, fornece uma visão abrangente do comportamento de sis-

temas complexos. Essa prática é essencial para a detecção precoce de problemas, diagnóstico eficaz e monitoramento contínuo do desempenho, garantindo que as aplicações mantenham a alta disponibilidade e eficiência.

Ao adotar um modelo de observabilidade robusto, as equipes de desenvolvimento e operações podem garantir que o sistema esteja sempre em conformidade com os requisitos de desempenho e segurança, ao mesmo tempo que reduzem o tempo de resposta a incidentes.

4.6.5 Resiliência

A resiliência em sistemas de software refere-se à capacidade de um sistema manter seu funcionamento de maneira eficiente e confiável, mesmo em condições adversas, como falhas, erros ou picos de demanda. Sistemas resilientes são projetados para lidar com imprevistos, garantindo a disponibilidade e a integridade dos serviços. Segundo David D. Woods, em seu artigo *Four Concepts for Resilience and the Implications for the Future of Resilience Engineering*, a resiliência é constituída por quatro conceitos principais: robustez, recuperação, extensibilidade com elegância e adaptabilidade sustentável. Cada um desses elementos desempenha um papel crucial para a capacidade de o sistema resistir, adaptar-se e recuperar-se diante de situações inesperadas, mantendo sua funcionalidade de forma confiável.

A robustez é a habilidade de um sistema resistir a falhas e variações ambientais sem perda de funcionalidade. Isso inclui sua capacidade de lidar com problemas como aumentos súbitos de tráfego ou falhas em componentes de hardware e software, mantendo-se estável. Para alcançar robustez, as arquiteturas de software frequentemente adotam práticas de redundância, como replicação de servidores e serviços, mecanismos de failover e técnicas de tolerância a falhas. Um aspecto importante da robustez é a contenção de falhas para que problemas localizados não se espalhem pela infraestrutura, o que pode ser obtido por meio de *circuit breakers* e mecanismos de controle de taxa (rate limiting), prevenindo a sobrecarga de recursos.

A recuperação envolve a capacidade do sistema de se restaurar rapidamente ao estado de funcionamento normal após uma falha ou interrupção. Sistemas resilientes são equipados para detectar falhas automaticamente e iniciar procedimentos de restauração de forma rápida, minimizando o impacto para o usuário. Esse processo pode incluir reinicializações automáticas de serviços, redirecionamento de tráfego para instâncias de backup e recuperação de dados comprometidos. Ferramentas de monitoramento e alertas são essenciais para essa recuperação eficaz, permitindo que a equipe técnica identifique problemas e atue proativamente. Em muitos casos, o uso de orquestradores para reiniciar serviços e平衡adores de carga para redirecionar tráfego agiliza esse processo, otimizando a continuidade do serviço.

A extensibilidade com elegância é a capacidade de um sistema se expan-

dir ou evoluir em resposta a novas demandas de forma coerente e sem impacto negativo na operação. Esse conceito é fundamental para manter a qualidade do sistema ao longo do tempo, pois permite que o software cresça em resposta a novas funcionalidades, sem sacrificar sua performance. Woods enfatiza que a elegância na extensibilidade envolve pensar em como os componentes do sistema podem ser aumentados ou adaptados sem aumentar a complexidade desnecessariamente, assegurando que o sistema não se torne excessivamente complicado ou propenso a falhas com a expansão.

A adaptabilidade sustentável é a capacidade de um sistema não apenas de se adaptar a mudanças, mas de fazê-lo de forma que preserve sua integridade a longo prazo. Sistemas adaptáveis são projetados para evoluir com mudanças nos requisitos de negócios, ambiente tecnológico e condições operacionais. Eles integram flexibilidade desde sua concepção, utilizando arquiteturas modulares, práticas de desenvolvimento ágeis e técnicas de automação que facilitam a implementação de atualizações e novas funcionalidades. Além disso, a auto escalabilidade – a capacidade de ajustar dinamicamente os recursos conforme a demanda – contribui para a adaptabilidade sustentável, permitindo que o sistema cresça sem sobrecarregar a infraestrutura.

A resiliência em sistemas de software é essencial para manter a continuidade do serviço diante de falhas e mudanças, preservando a experiência do usuário e minimizando interrupções. Com base nos quatro conceitos propostos por Woods – robustez, recuperação, extensibilidade com elegância e adaptabilidade sustentável – é possível desenvolver sistemas modernos e resilientes, capazes de suportar condições desafiadoras e operar de forma eficiente e confiável.

5. CONCLUSÃO

A disponibilização de conteúdos significativos e abrangentes a uma ampla variedade de tópicos cruciais para o desenvolvimento de software, incluindo gerenciamento de projetos, arquitetura, boas práticas, segurança e performance.

Essa abordagem contribui para a formação de uma compreensão robusta e integrada sobre a teoria e prática no desenvolvimento de software. Os conteúdos auxilia desenvolvedores na construção de softwares escaláveis, sustentáveis e de alta qualidade, prevenindo problemas recorrentes, como *overengineering*, duplicação de código, nomenclaturas inadequadas, dependências excessivas entre componentes, vulnerabilidades de segurança, complexidade desnecessária, falta de testes automatizados, erros na definição de requisitos, baixa performance e ausência de uma arquitetura adequada. Ao direcionar o usuário para práticas recomendadas, o artigo promove um processo de desenvolvimento mais consciente e eficiente, que antecipa desafios e evita a propagação de problemas comuns.

Isso faz o projeto alcançar seu principal objetivo — auxiliar desenvolvedores a conceber e implementar soluções de software ajustadas às demandas e limitações específicas de cada projeto, trazendo o desenvolvimento sustentável e escalável —, este projeto contribui para a formação de uma comunidade de desenvolvedores mais capacitada e informada. Dessa forma, ao melhorar significativamente a qualidade técnica dos projetos, o artigo gera benefícios que se estendem à sociedade, ao fomentar a criação de softwares mais eficazes e resilientes, capazes de resolver problemas reais com eficiência.

Assim, o artigo não só aprimora a prática de desenvolvimento individual, mas também impacta positivamente o ecossistema de software como um todo, promovendo uma cultura de excelência técnica e facilitando a criação de soluções que agregam valor à comunidade.

REFERÊNCIAS

AKITA, F. Como fazer o Ingresso.com escalar?: Conceitos Intermediários de Web. Youtube: Fabio Akita, 2021. Disponível em: <https://www.youtube.com/watch?v=0TMr8rsmU-k>. Acesso em: 6 maio 2024.

AKITA, F. Entendendo Conceitos Básicos de CRIPTOGRAFIA: Parte 1/2. Youtube: Fabio Akita, 2019. Disponível em: https://www.youtube.com/watch?v=CcU5Kc_FN_4. Acesso em: 6 maio 2024.

AKITA, F. Entendendo Conceitos Básicos de CRIPTOGRAFIA: Parte 2/2. Youtube: Fabio Akita, 2019. Disponível em: <https://www.youtube.com/watch?v=HCHqtpipwu4>. Acesso em: 6 maio 2024.

AMARATUNGA, K. Multithreading. MIT, 2000. Disponível em: <https://ocw.mit.edu/courses/1-124j-foundations-of-software-engineering-fall-2000/pages/lecture-notes/multithreading/>. Acesso em: 10 out. 2024.

ATLASSIAN, Saiba tudo sobre o Gitflow Workflow. Disponível em: <https://www.atlassian.com/br/git/tutorials/comparing-workflows/gitflow-workflow>. Acesso em: 10 jul. 2024.

CLOUDFLARE. O que é um proxy reverso?: Servidores proxy explicados. Disponível em: <https://www.cloudflare.com/pt-br/learning/cdn/glossary/reverse-proxy/>. Acesso em: 10 out. 2024.

COHN, M. Succeeding with Agile. Upper Saddle River: Addison-Wesley, 2009.

CRUME, J. Cybersecurity Architecture Series. [SI]: IBM Technology, 2023. Disponível em: <https://www.youtube.com/playlist?list=PL0spHqNVtKADkWLft9OcziQF7EatuANSY>. Acesso em: 6 maio 2024.

EVANS, E. Domain-Driven Design: Atacando as Complexidades no Coração do Software. 3. ed. Rio de Janeiro: Alta Books, 2020.

FOWLER, M. Refatoração: Aperfeiçoando o design de códigos existentes. 2. ed. São Paulo: Novatec Editora, 2019.

FOWLER, M. Strangler Fig. MartinFowler.com, 2024. Disponível em: <https://martinfowler.com/bliki/StranglerFigApplication.html>. Acesso em: 10 jul. 2024.

FOWLER, S. J. Microsserviços prontos para a produção: Construindo sistemas padronizados em uma organização de engenharia de software. São Paulo: Novatec Editora, 2020.

REFERÊNCIAS

GIT, Getting Started: What is Git?. 2005. Disponível em: <https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F>. Acesso em: 10 jul. 2024.

GIT. 2005. Disponível em: <https://git-scm.com/>. Acesso em: 10 jul. 2024.

HUMBLE, J.; FARLEY, D. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Upper Saddle River: Addison-Wesley, 2010.

IBM. Indexação de Banco de Dados. 2021. Disponível em: <https://www.ibm.com/docs/pt-br/mfci/7.6.2?topic=databases-database-indexing>. Acesso em: 10 out. 2024.

IBM. O que é DevSecOps?. Disponível em: <https://www.ibm.com/br-pt/topics/devsecops#:~:text=O%20DevSecOps%20introduz%20processos%20de,t%20ratados%20assim%20que%20s%C3%A3o%20identificados>. Acesso em: 10 out. 2024.

JOHNSON, J. CHAOS Report: Beyond Infinity. 2020. Disponível em: <https://standishgroup.myshopify.com/collections/frontpage/products/copy-of-chaos-report-beyond-infinity-digital-version>. Acesso em: 10 maio 2024.

JWT. Introduction to JSON Web Tokens. 2016. Disponível em: <https://jwt.io/introduction>. Acesso em: 10 out. 2024.

KEMP, S. Digital 2023: Global Overview Report, 2023. Disponível em: <https://datareportal.com/reports/digital-2023-global-overview-report>. Acesso em: 23 abr. 2024.

KRASNER, H. The Cost of Poor Software Quality in the US: A 2020 Report, 2020. Disponível em: <https://www.it-cisq.org/the-cost-of-poor-software-quality-in-the-us-a-2020-report>. Acesso em: 27 fev. 2024.

MARTIN, R. C. Arquitetura Limpa: O Guia do Artesão para Estrutura e Design de Software. Rio de Janeiro: Alta Books, 2019.

MARTIN, R. C. Código Limpo: Habilidades Práticas do Agile Software. Rio de Janeiro: Alta Books, 2009.

MARTIN, R. C. Desenvolvimento Ágil Limpo: De Volta às Origens. Rio de Janeiro: Alta Books, 2020.

REFERÊNCIAS

MICROSOFT. Banco de Dados NoSQL - O que é NoSQL?: Uma visão geral como introdução. Disponível em: [https://azure.microsoft.com/pt-br/resources/cloud-computing-dictionary/what-is-nosql-database#:~:text=Os%20bancos%20de%20dados%20NoSQL%20podem%20ser%20chamados%20de%20%22n%C3%A3o,SQL\)%20com%20linhas%20e%20tabelas](https://azure.microsoft.com/pt-br/resources/cloud-computing-dictionary/what-is-nosql-database#:~:text=Os%20bancos%20de%20dados%20NoSQL%20podem%20ser%20chamados%20de%20%22n%C3%A3o,SQL)%20com%20linhas%20e%20tabelas). Acesso em: 10 jul. 2024.

MICROSOFT. O que é um Banco de Dados SQL?: Obtenha uma visão geral da tecnologia, dos benefícios e dos casos de uso do SQL. Disponível em: <https://azure.microsoft.com/pt-br/resources/cloud-computing-dictionary/what-is-sql-database>. Acesso em: 10 jul. 2024.

NEWMAN, S. Criando Microsserviços: Projetando sistemas com componentes menores e mais especializados. 2. ed. São Paulo: Novatec Editora, 2022.

NEWMAN, S. Migrando sistemas monolíticos para microsserviços: Padrões evolutivos para transformar seu sistema monolítico. São Paulo: Novatec Editora, 2020.

NGINX. 2004. Disponível em: <https://nginx.org/>. Acesso em: 10 out. 2024.

NIST. Cybersecurity Framework. 2018. Disponível em: <https://www.nist.gov/cyberframework/csf-11-archive>. Acesso em: 10 out. 2024.

OWASP. STOCK, A.; GLAS, B.; SMITHLINE, N.; GIGLER, T. OWASP Top 10. 2022. Disponível em: <https://owasp.org/Top10/#thank-you-to-our-sponsors>. Acesso em: 6 maio 2024.

REDHAT. Docker: desenvolvimento de aplicações em containers. 2023. Disponível em: <https://www.redhat.com/pt-br/topics/containers/what-is-docker>. Acesso em: 10 jul. 2024.

REDIS. 2009. Disponível em: <https://redis.io/company/>. Acesso em: 10 out. 2024.

Richards, M.; Ford, N. Fundamentals of Software Architecture. Massachusetts: O'Reilly Media, 2020.

WOODS, D. Four concepts for resilience and the implications for the future of resilience engineering. 2015. Disponível em: https://www.researchgate.net/publication/276139783_Four_concepts_for_resilience_and_the_implications_for_the_future_of_resilience_engineering. Acesso em: 10 jul. 2024.

INFORMAÇÕES ADICIONAIS

ORGANIZADORES

Níkollas David Oliveira Rufino

nikollasdavidor@aluno.fapce.edu.br

Sistemas de Informação

Centro Universitário Paraíso

Fabrício Carneiro Costa

fabricio.carneiro@fapce.edu.br

Coordenador do Curso de Sistemas de Informação

Mestre em engenharia de software com ênfase em Learning Analytics

Centro Universitário Paraíso

Italo Gonçalves Luciano

italog.luciano@fapce.edu.br

Sistemas de Informação

Centro Universitário Paraíso

Kauan Ribeiro Feijó Gondim

Kauanribeiro@aluno.fapce.edu.br

Sistemas de Informação

Centro Universitário Paraíso

Lázaro Rytson da Silva Bezerra

lazarorytson@aluno.fapce.edu.br

Sistemas de Informação

Centro Universitário Paraíso

Jonathan Tharles Ferreira de Oliveira

jonathanoliveira123@aluno.fapce.edu.br

Sistemas de Informação

Centro Universitário Paraíso

Laryssa Brilhante Pessoa

laryssamavis@aluno.fapce.edu.br

Sistemas de Informação

Centro Universitário Paraíso

Jose Nathannael Cavalache de Alencar

nathancavalache@aluno.fapce.edu.br

Análise e desenvolvimento de sistemas

Centro Universitário Paraíso

INFORMAÇÕES ADICIONAIS

ORGANIZADORES

Luiz Lopes de Alcântara

luizlopesbr@aluno.fapce.edu.br

Sistemas de Informação

Centro Universitário Paraíso

Antoni Demétrius Bezerra Batista

antonidemetrius@aluno.fapce.edu.br

Sistemas da informação

Centro Universitário Paraíso

Emmanuel Soares Silva

emmanuelsoaresilva@aluno.unifapce.edu.br

Sistemas de Informação

Centro universitário Paraíso

Joely Sousa Silva

joely.silva@aluno.unifapce.edu.br

Sistemas de Informação

Centro universitário Paraíso

A *Learn Skills Cursos* é uma empresa de caráter educacional com ênfase na capacitação acadêmica e profissional na área da saúde, agora, também, se especializando na editoração, publicação e divulgação de pesquisa e produções científicas.





LEARN
SKILLS CURSOS
aprendendo por competências

ISBN: 978-65-83475-00-8

9 786583 475008