

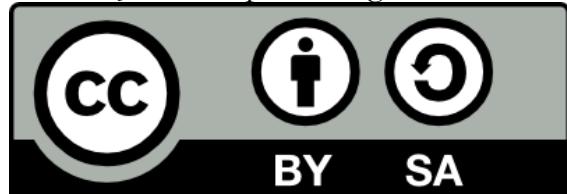
Arquitetura de Sistemas Web

Princípios, Práticas e Tecnologias

Edição 2023

Marco Mendes

Copyright © por Marco Aurélio de Souza Mendes
Este trabalho está licenciado sob uma Licença Creative Commons
Atribuição-CompartilhaIgual 4.0 Internacional.



Edição 2023
Versão 2.0.2

Sobre o Autor



Sou mestre em Ciência da Computação pelo DCC/UFMG (1996) e bacharel em Ciência da Computação pelo DCC/UFMG (1993). Atuo como professor de cursos *lato-sensu* da PUC Minas desde 2004 nas temáticas de Arquitetura de Software, Engenharia de Software e Métodos Ágeis. Sou também consultor Kanban e DevOps pela Arkhi Business Agility.

Tenho experiência profissional no Brasil desde 1993 e no exterior desde 1998, sendo especialista em arquiteturas corporativas, arquitetura de aplicações Web, arquitetura Java EE, .NET, APIs, microserviços, engenharia de software e métodos ágeis.

Maiores informações e contatos profissionais disponíveis aqui no LinkedIn
<http://linkedin.com/in/marcomendes>

1	Como Arquitetar Sistemas de Software	12
1.1	O Que são arquiteturas de software	12
1.2	A Função dos Arquitetos de Software.....	13
1.3	Um Processo Mínimo para Arquitetar Sistemas	14
1.4	A Relação do Arquiteto com o Time do Projeto	15
1.5	Para Saber Mais.....	16
2	Requisitos Arquiteturais.....	17
2.1	O que são Requisitos Arquiteturais	17
2.2	Como Descrever Bons Requisitos Arquiteturais.....	17
2.3	Requisitos de Acessibilidade e Usabilidade	18
2.4	Requisitos de Autenticação e Autorização	19
2.5	Requisitos de Confidencialidade e Integridade	19
2.6	Requisitos Amplos de Segurança	20
2.7	Requisitos de Alta Disponibilidade	20
2.8	Requisitos de Tolerância a Falhas.....	20
2.9	Requisitos de Performance	21
2.10	Para Saber Mais	21
3	Estilos Arquiteturais Básicos	22
3.1	Estilo Arquitetural em Camadas.....	22
3.1.1	PRÍNCIPIO ABERTO FECHADO	22
3.1.2	CAMADAS FÍSICAS.....	23
3.2	Estilo MVC	23
3.3	Estilo MVVM	24
3.4	DDD – Desenho Dirigido por Domínios	26
3.4.1	O QUE É o DDD.....	26
3.4.2	DDD VERSUS MVC E MVVM	26
3.4.3	CONCEITOS DDD	27
3.4.4	LINGUAGEM UBÍQUA NO DDD.....	27
3.4.5	EXEMPLO DE LINGUAGEM UBÍQUA.....	28
3.4.6	MODELO DE DOMÍNIO E O DDD.....	28
3.4.7	EXEMPLO DE MODELO DE DOMÍNIO	29
3.4.8	CAMADAS TÍPICAS DO DDD.....	30
3.4.9	DEPENDÊNCIAS ENTRE CAMADAS NO DDD	31
3.4.10	O DOMÍNIO É IGNORANTE.....	31
3.4.11	EXEMPLO DE CÓDIGO	32
3.4.12	PADRÕES DE DESENHO NO DDD	35
3.4.13	RISCOS NO DDD	37
4	Estilos Arquiteturais Avançados	38
4.1	Pipelines	38
4.1.1	PIPES E FILTERS	38
4.2	MicroKernel	39
4.3	Serviços	40
4.4	Microsserviços	40
4.5	Arquiteturas Baseadas em Eventos.....	42
4.5.1	BROKERS E MEDIATORS	43

4.6 Arquitetura Baseada em Espaços.....	43
5 Estilo Arquitetural de APIs.....	45
5.1 O que é uma API.....	45
5.2 Plataforma de APIs como Aceleradores.....	45
5.3 O API Gateway.....	46
5.4 APIs Baseadas em REST.....	46
5.4.1 MODELO DE MATURIDADE DE RICHARDSON.....	47
5.4.2 EXEMPLO DE API REST NO NIVEL 2	47
5.4.3 EXEMPLO DE API REST NO NÍVEL 3	48
5.5 APIs Baseadas em RPC.....	49
5.6 gRPC.....	49
5.7 REST versus RPC	51
5.8 GraphQL	51
5.9 WebSocket	53
5.9.1 EVOLUÇÕES DO HTTP COMO ALTERNATIVAS AO WEB SOCKET	55
5.10 Comparativo de Protocolos de Transporte.....	56
5.11 Comparativo de Protocolos de Dados	57
5.12 Comparativo de Táticas de APIs	58
6 Arquiteturas de Eventos	59
6.1 Conceitos	59
6.2 Conceitos Básicos	59
6.3 Padrões para Arquitetura de Eventos – Broker.....	60
6.4 Padrões para Arquitetura de Eventos – Mediator	61
6.5 Diferenças entre o Broker e o Mediator	62
6.6 Processamento de Mensagens e Fluxos	63
6.7 Comparativos	64
6.7.1 COMPARATIVO DE PADRÕES	64
6.7.2 COMPARATIVO DE FERRAMENTAS	64
6.7.3 PROCESSAMENTO DE FILA DE MENSAGENS COM BROKER AWS SQS	65
6.7.4 PROCESSAMENTO DE STREAMS COM BROKER APACHE KAFKA	66
6.7.5 PROCESSAMENTO DE FILA DE MENSAGENS COM MEDIATOR APACHE CAMEL	67
6.7.6 PROCESSAMENTO DE FILA DE MENSAGENS COM MEDIATOR APACHE CAMEL	67
7 Topologia de Servidores Web	67
7.1 Servidores Web Baseados em Processos (1º Geração)	68
7.2 Servidores Web Baseados em Threads (2º Geração)	68
7.3 Servidores Web de Aplicações (3º Geração).....	68
7.4 Servidores Web de Micro contêineres (4º Geração)	69
7.5 Comparativo de Servidores Web	70
7.6 Distribuição Física de Aplicações Web.....	70
7.7 Distribuição Mínima	71
7.8 Distribuição com Servidor Web Dedicado	71
7.9 Distribuição com Servidor de Aplicação Dedicado	71
7.10 Distribuição com Máquina para Conteúdo Estático	73
7.11 Distribuição com Arquiteturas Elásticas.....	73
7.12 Comparativo de Topologias Web	74
8 Tecnologias Web.....	75
8.1 Tecnologias Web de Visão	75

8.1.1	HTML 5.....	75
8.1.2	TECNOLOGIAS DE CSS	75
8.1.3	TECNOLOGIAS DE VISÃO MVC (SERVER SIDE)	76
8.1.4	TECNOLOGIAS DE VISÃO MVVM BASEADAS EM JAVASCRIPT	76
9	Servidores Web Baseados em Java EE.....	77
9.1	JSF (Java Server Faces)	79
9.2	Servlets.....	79
9.3	Serviços Web (JAX-WS e JAX-RS)	79
9.4	JAAS (Java Authentication and Authorization Service).....	80
9.5	EJB (Enterprise Java Beans) Session Beans.....	80
9.6	JPA (Java Persistence API)	80
9.7	JCA (Java Connector Architecture).....	81
9.8	Considerações de Desenho Arquitetural.....	81
9.8.1	ESCOLHA DE SERVIDORES.....	81
9.8.2	PRODUTIVIDADE.....	81
9.8.3	PROCESSAMENTO DE REQUISIÇÕES WEB	82
9.8.4	NAVEGAÇÃO.....	82
9.8.5	DESENHO DE PÁGINAS	82
9.8.6	AUTENTICAÇÃO	82
9.8.7	AUTORIZAÇÃO	83
9.8.8	CACHE.....	83
9.8.9	CONTROLE TRANSACIONAL	83
9.8.10	AUDITORIA (LOGS)	83
9.8.11	INSTRUMENTAÇÃO	83
9.8.12	GERÊNCIA DE SESSÃO WEB.....	84
9.8.13	VALIDAÇÃO	84
9.8.14	DESENHO DE SERVIÇOS WEB.....	84
9.8.15	CAMADA DE NEGÓCIO.....	84
9.8.16	ACESSO A DADOS	84
9.9	Riscos e Oportunidades para o Arquiteto Web Java EE.....	85
9.9.1	CURVA DE APRENDIZADO ALTA.....	85
9.9.2	BAIXA PRODUTIVIDADE	85
9.9.3	CONSUMO DE MEMÓRIA	85
9.9.4	DESCONTINUAÇÃO TECNOLÓGICA DE FRAMEWORKS	85
9.10	Para Saber Mais	86
10	Servidores Web Baseados em ASP.NET.....	87
10.1	ASP.NET Web Forms	90
10.2	ASP.NET MVC	91
10.3	WCF.....	91
10.4	ASP.NET Core 1	92
10.5	ASP.NET CORE 2 e 3	93
10.6	ASP.NET Web API	93
10.7	Entity Framework e LINQ	94
10.8	MSMQ.....	94
10.9	Service Fabric.....	94
10.10	Considerações de Desenho Arquitetural	95
10.10.1	CONFIGURAÇÃO DE SERVIDORES	95
10.10.2	PRODUTIVIDADE	96

10.10.3	PROCESSAMENTO DE REQUISIÇÕES WEB	96
10.10.4	NAVEGAÇÃO.....	97
10.10.5	DESENHO DE PÁGINAS	97
10.10.6	AUTENTICAÇÃO	97
10.10.7	AUTORIZAÇÃO	97
10.10.8	CACHE.....	98
10.10.9	CONTROLE TRANSACIONAL	98
10.10.10	AUDITORIA (LOGS)	98
10.10.11	INSTRUMENTAÇÃO	98
10.10.12	GERÊNCIA DE SESSÃO WEB.....	98
10.10.13	VALIDAÇÃO	99
10.10.14	DESENHO DE SERVIÇOS WEB.....	99
10.10.15	ACESSO A DADOS	99
10.11	Riscos e Oportunidades para o Arquiteto Web ASP.NET	99
10.11.1	ESTRUTURAÇÃO DE BONS MODELOS DE DOMÍNIO E BONS CÓDIGOS	99
10.11.2	CONSUMO DE MEMÓRIA EM APLICAÇÕES ASP.NET WEBFORMS	99
10.11.3	A NOVA ARQUITETURA ASP.NET CORE	100
10.11.4	MAPEAMENTO DE TECNOLOGIAS JAVA EE E .NET	100
10.12	Passado, Presente e Futuro da Plataforma .NET Core.....	100
10.13	Para Saber Mais.....	102

11 Servidores Web Baseados em JavaScript..... 103

11.1	Aceleradores, Bibliotecas e Frameworks CSS	109
11.2	Linguagens Aceleradoras sobre JavaScript	110
11.3	Linguagem ECMA Script (JavaScript 6, 7, 8, 9 e 10).....	112
11.4	Bibliotecas de Componentes JavaScript	112
11.5	Frameworks MVVM JavaScript	112
11.6	Ferramentas de Suporte JavaScript.....	114
11.6.1	GERENCIADORES DE DEPENDÊNCIAS.....	114
11.6.2	EXECUTORES DE TAREFAS	114
11.6.3	AUTOMAÇÃO DE TESTES COM JAVASCRIPT.....	115
11.7	JavaScript Servidor e o Node.JS	115
11.8	Mapeamento de Tecnologias JavaScript, Java EE Web e ASP.NET	117
11.9	Riscos e Oportunidades para o Arquiteto JavaScript.....	117
11.10	Para Saber Mais	118

12 Documentação de Arquiteturas 119

12.1	Passo 1 - Visualização de Negócio	119
12.2	Passo 2 – Condutores Arquiteturais.....	120
12.2.1	ACESSIBILIDADE E USABILIDADE	120
12.2.2	INTEROPERABILIDADE	120
12.2.3	SEGURANÇA	121
12.2.4	ESCALABILIDADE	121
12.2.5	MANUTENIBILIDADE	121
12.3	Passo 3 – Estilos Arquiteturais Web	121
12.4	Passo 4 – Escolha da Plataforma Tecnológica	121
12.5	Passo 5 – Visualização Lógica.....	122
12.6	Passo 6 – Visualização da Topologia Física.....	123
12.7	Passo 7 – Visualização da Persistência de Dados	123
12.8	Passo 8 – Visualização da Apresentação (Usabilidade e Acessibilidade).....	125
12.9	Passo 9 – Visualização de Segurança	126

12.10	Passo 10 – Visualização de Interoperabilidade	128
12.11	Passo 11 – Visualização de Manutenibilidade	129
12.12	Passo 12 – Alocação de Componentes aos Nodos Físicos	130
12.13	Passo 13 – Determinar Provas de Conceito	131
12.14	Modelos de um Documento de Arquitetura de Software	132
13	Aceleração de Arquiteturas com Práticas DevOps	133
13.1	DevOps para Aceleração da Entrega de Produtos	133
13.2	DevOps para Criar Progresso Real nos Projetos	134
13.3	Práticas DevOps	135
13.3.1	COMUNICAÇÃO TÉCNICA AUTOMATIZADA	135
13.3.2	QUALIDADE CONTÍNUA DO CÓDIGO	135
13.3.3	CONFIGURAÇÃO COMO CÓDIGO	135
13.3.4	GESTÃO DOS BUILDS	136
13.3.5	AUTOMAÇÃO DOS TESTES	136
13.3.6	TESTES DE CARGA	136
13.3.7	GESTÃO DE CONFIGURAÇÃO	136
13.3.8	AUTOMAÇÃO DOS RELEASES	137
13.3.9	AUTOMAÇÃO DA MONITORAÇÃO DE APLICAÇÕES	137
13.3.10	TESTES DE ESTRESSE	137
13.3.11	INTEGRAÇÃO CONTÍNUA (<i>CONTINUOUS INTEGRATION</i>)	138
13.3.12	IMPLANTAÇÃO CONTÍNUA (<i>CONTINUOUS DEPLOYMENT</i>)	138
13.3.13	ENTREGA CONTÍNUA (<i>CONTINUOUS DELIVERY</i>)	138
13.3.14	IMPLANTAÇÕES CANÁRIOS	139
13.3.15	INFRAESTRUTURA COMO CÓDIGO (<i>IAC</i>)	139
13.3.16	AMBIENTES SELF-SERVICE	140
13.3.17	INJEÇÃO DE FALHAS	140
13.3.18	TELEMETRIA	141
13.3.19	PLANEJAMENTO DE CAPACIDADE	141
13.4	Ferramentas DevOps	141
13.4.1	PLATAFORMAS DE COLABORAÇÃO E COMUNICAÇÃO	141
13.4.2	FERRAMENTAS DE ANÁLISE DE CÓDIGO FONTE	141
13.4.3	FERRAMENTAS DE GERÊNCIA DE CÓDIGO FONTE (SCM)	142
13.4.4	FERRAMENTAS DE AUTOMAÇÃO DE BUILDS	142
13.4.5	FERRAMENTAS DE INTEGRAÇÃO CONTÍNUA	142
13.4.6	FERRAMENTAS DE GESTÃO DE CONFIGURAÇÃO E PROVISIONAMENTO	142
13.4.7	FERRAMENTAS DE CONTEINERIZAÇÃO	143
13.4.8	FERRAMENTAS DE GESTÃO DE REPOSITÓRIOS	143
13.4.9	FERRAMENTAS DE AUTOMAÇÃO DE TESTES	143
13.4.10	FERRAMENTAS DE TESTES DE PERFORMANCE, CARGA E ESTRESSE	144
13.4.11	FERRAMENTAS DE INJEÇÃO DE FALHAS	144
13.4.12	PLATAFORMAS DE NUvens	144
Bibliografia	146	

1 Como Arquitetar Sistemas de Software

1.1 O Que são arquiteturas de software

A arquitetura de software é uma disciplina da engenharia de software que tem por objetivo suportar a tomada das decisões técnicas mais importantes em um projeto e garantir que essas decisões sejam corretamente implementadas ao longo da construção desse projeto. A arquitetura de software de um sistema Web reduz os riscos de um projeto de software e aumenta as chances de que este seja bem-sucedido e alinhado às necessidades das organizações.

Definição de Arquitetura de Software: "A Software architecture is the set of design decisions which, if made incorrectly, may cause your project to be cancelled." (ROZANSKI; WOODS, 2011).

Uma arquitetura de sistema de software é formada por:

- Uma coleção de necessidades dos principais interessados sobre o que um sistema precisa;
- Uma coleção de softwares e componentes do sistema, suas conexões e restrições;
- A lógica que demonstra que os componentes, conexões e restrições que definem um sistema, se implementadas, satisfaria a coleção de necessidades que o sistema apresenta.

Uma arquitetura de software é composta por um conjunto de decisões significativas sobre a organização de um sistema de software, a seleção dos seus elementos estruturais e suas interfaces. Além disso, a arquitetura inclui o comportamento como especificado nas colaborações entre esses elementos, a composição desses estruturais e elementos comportamentais em subsistemas maiores, em conformidades com o estilo arquitetônico que guia essa organização.

Mary Shaw e David Garlan (SHAW; GARLAN, 1996) sugerem, ainda no começo dos anos 90, que a arquitetura de software é um tipo de desenho (design) que vai além das questões dos algoritmos e estruturas de dados da computação. A concepção e especificação da estrutura geral do sistema emergem, então, como um novo tipo de problema.

Essas questões estruturais incluem:

- Qual o **estilo do software** a ser construído (Web 1.0, Web 2.0, API Web, Microsserviços)?
- Qual a **plataforma** do software a ser construído (Java EE Web, ASP .NET, PHP, NodeJS)?
- Quais as principais **restrições e premissas** que devem ser observadas?
- Quais os **principais requisitos arquiteturais** que devem ser atendidos (usabilidade, performance, confiabilidade)?
- Quais as **principais frameworks e tecnologias** que irão ser usados?
- Qual a **modularização lógica e física** dos componentes e regras de negócio do software a ser construído.

Além disso, a arquitetura também deve endereçar elementos não técnicos, chamados de sociotécnicos, que incluem:

- O contexto ambiental da empresa para onde o software está sendo entregue;
- As competências do time de desenvolvimento que irá trabalhar no projeto;
- Restrições financeiras;
- Restrições temporais, entre outras.
- Uma arquitetura de software fornece os seguintes benefícios para projetos de software.
- A arquitetura de software promove a geração de valor dentro de um projeto através da resolução dos cenários de negócio mais importantes e complexos, habilitando os times gerenciais para a construção do projeto com riscos reduzidos;
- Em nível organizacional, a arquitetura de software promove o reuso de software entre projetos e promove o alinhamento das diretrizes técnicas de um projeto com as diretrizes da organização.

1.2 A Função dos Arquitetos de Software

O arquiteto de software é ainda um papel recente na comunidade brasileira de software. Portanto, muita confusão ainda existe sobre o papel que realiza dentro de um desenvolvimento de software. Atenção! Um arquiteto não é um desenvolvedor sênior. Um desenvolvedor é especialista e tático. Já um arquiteto é um generalista-especialista com visão estratégica.

Isso não significa, entretanto, que o arquiteto viva em uma torre de marfim, criando decisões desconectadas da realidade. É esperado que um arquiteto tenha habilidades de desenvolvimento e acompanhe os times de desenvolvimento no dia a dia do desenvolvimento de software. A colaboração diária entre arquitetos e desenvolvedores é um dos maiores fatores de sucesso de arquiteturas em projetos.

De acordo com Phillippe Kruchten (KRUCHTEN, 2003), o dia típico de um arquiteto tem a seguinte natureza de ocupação do tempo:

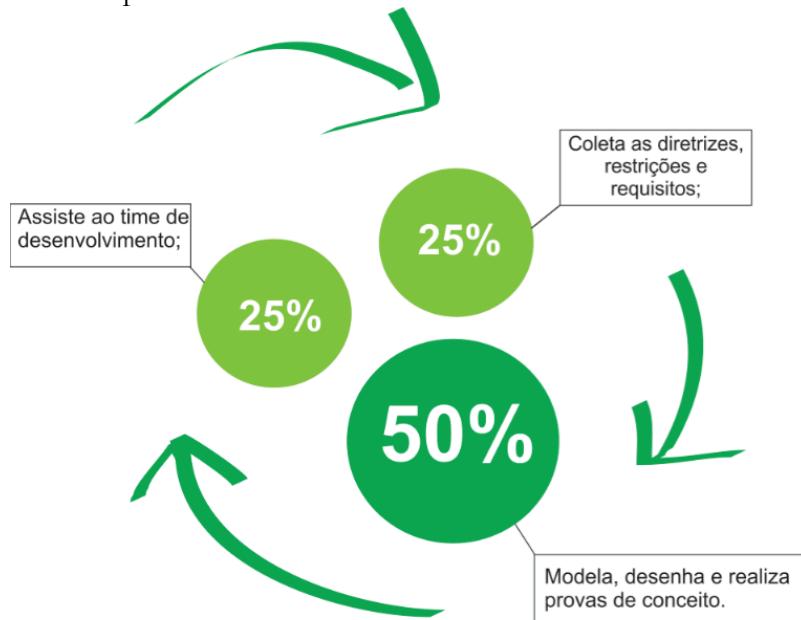


Figura 1: A agenda de um arquiteto de aplicações Web.

Um arquiteto deve trabalhar em intensa e forte colaboração com a equipe, apoiando o time na investigação dos pontos de relevância técnica de um projeto. Um arquiteto deve atuar como um líder técnico, realizando a identificação dos mecanismos arquiteturais relevantes, motivando o time para a investigação e resolução desses mecanismos e apoiando o time do início ao fim do projeto.

Um bom arquiteto Web deve possuir as seguintes características:

- Possuir liderança técnica;
- Ser hábil negociador;
- Possuir conhecimentos de desenho e programação em tecnologias Web;
- Possuir conhecimentos do domínio da aplicação;
- Ser capaz de tomar decisões em condições de imprecisão e conduzir times de projetos.

Um arquiteto de software deve conhecer também outras disciplinas (gerência de projetos e análise de negócio) ou domínios (hardware, dados ou segurança). Além disso, um arquiteto também deve possuir excelente capacidade de conduzir debates e realizar comparações entre as arquiteturas candidatas dos problemas do dia a dia. A liderança técnica e a capacidade de apoiar os times de projetos também são bem-vindas dentro do perfil de um arquiteto.

Dada a dificuldade dessa tarefa, é comum que empresas formem o que chamamos de time de arquitetura, composta por um conjunto de pessoas com bom domínio de cada um dos itens acima e com excelente capacidade de conduzir discussões e realizar apoio aos times de projetos.

O time de arquitetura ou arquiteto são peças fundamentais para a garantia de sucesso de projetos, e tem como principais atividades:

- Identificação, análise, mitigação e contingência de riscos técnicos;
- Definição, coordenação e execução das estratégias técnicas dos projetos;
- Treinamento e acompanhamento da equipe nas principais tecnologias envolvidas em um projeto;
- Garantia de que projetos sejam repassados para o ambiente de produção sem grandes contratemplos.

1.3 Um Processo Mínimo para Arquitetar Sistemas

Projetos de sistemas triviais podem ser executados sem a figura de um arquiteto Web. Mas para projetos não triviais, é importante que uma pessoa ou um time assuma o papel do arquiteto. Ao executar esse papel, é importante seguir os passos de um processo mínimo de trabalho.

O primeiro passo é garantir a presença de um arquiteto no início do projeto. Alguns gerentes alocam equipes técnicas de projetos no meio de projetos, adiando riscos técnicos e perdendo oportunidades de aumento de produtividade.

No início do projeto, o arquiteto deve realizar entre entrevistas com os usuários-chave para coletar os **condutores e requisitos arquiteturais**. Os condutores e requisitos arquiteturais irão fornecer a agenda de trabalho do time de arquitetura. Um catálogo desses requisitos é apresentado no Capítulo 3.

A partir dos condutores e requisitos arquiteturais, o time de arquitetura e o time de gerência estabelecem o **escopo da arquitetura**, que é um subconjunto dos condutores e requisitos elicitados.

Com o escopo do projeto em mãos, o time de arquitetura trabalha na definição do **estilo e plataforma arquitetural**.

A partir do estilo e plataforma, o arquiteto deve trabalhar nas soluções táticas da arquitetura, tais como a segurança, integrações, usabilidade, desempenho ou confiabilidade.

O estilo, plataforma e mecanismos podem ser desenhados em linguagens de forma livre ou mesmo em linguagens formais com a UML2 ou Archimate. Estes desenhos são chamados de **visualizações** e permitem expressar a arquitetura para os interessados. Visualizações usadas incluem a lógica (diagrama de pacotes), implementação (diagrama de componentes) ou implantação (diagrama de implantação).

O arquiteto deve então identificar e mitigar os riscos advindos das escolhas tecnológicas. Alguns destes riscos devem ser explorados através de código executável, chamados de **provas de conceito**. A profundidade e extensão da prova de conceito é proporcional ao risco identificado pelo time de arquitetura. O arquiteto junto com o time de desenvolvimento **executa as provas de conceito**. Em adição, o time escolhe também cenários de negócio ponta a ponta para execução sobre a arquitetura. A arquitetura é refinada ao longo deste trabalho a partir dos resultados de passo.

Por último, o arquiteto **acompanha e suporta o trabalho de desenvolvimento**. Os requisitos devem ser construídos a partir das diretrizes arquiteturais e aprendizado obtido até o momento no projeto.

Os passos desse processo são resumidos na Figura 2.

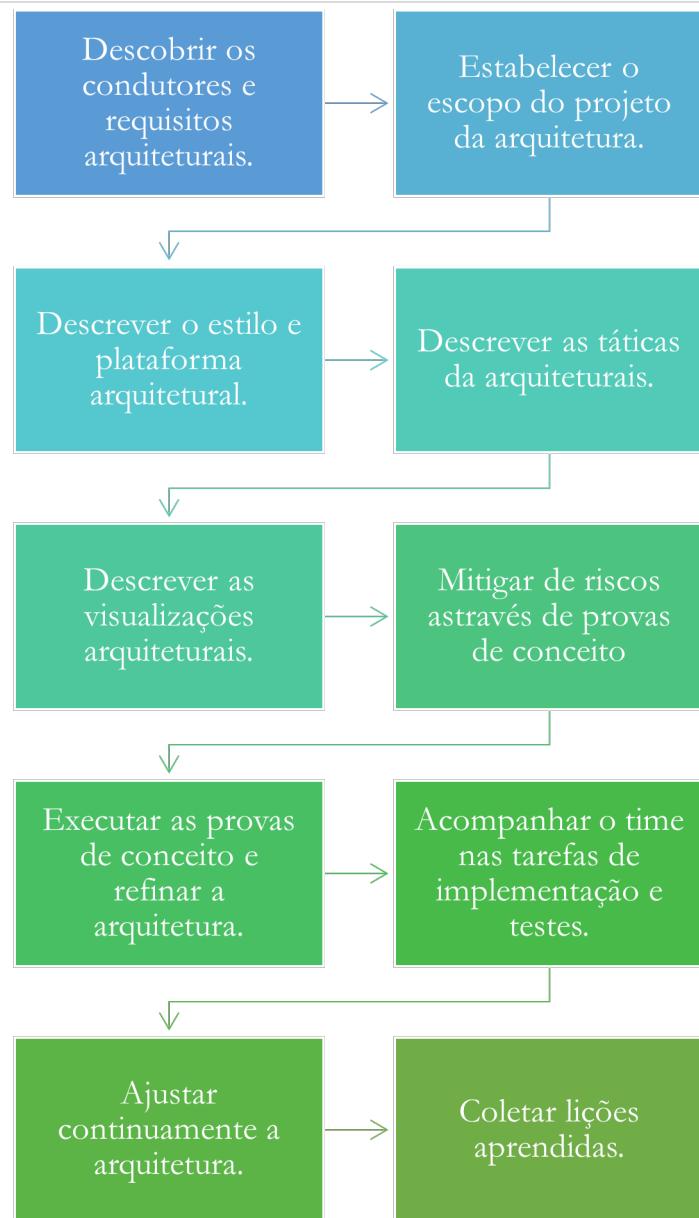


Figura 2: Um processo mínimo para desenvolver arquiteturas Web.

1.4 A Relação do Arquiteto com o Time do Projeto

Analista de Requisitos

O time de arquitetura avalia todos os requisitos funcionais do sistema para a elaboração da arquitetura. Além disso, o arquiteto apoia os analistas de negócio e requisitos a definirem os requisitos não-funcionais. O arquiteto trabalha junto com time de negócios para estabelecer que requisitos afetam o trabalho da arquitetura.

Desenvolvedores

Todo código do sistema deve ser baseado nas decisões realizadas pelo time de arquitetura e desenvolvimento. Por exemplo, em um sistema 3 camadas, o código da camada Web não deve invocar o código da camada de acesso a dados.

Quando novas decisões de codificação afetam a arquitetura, elas devem ser discutidas com o time de arquitetura que podem refinar o trabalho da arquitetura. É esperado que arquitetos e desenvolvedores mantenham colaboração diária e comunicação constante.

Analistas Projetistas

Algumas empresas formalizam o detalhamento dos requisitos em modelos detalhados. Para essas organizações, a arquitetura fornece as diretrizes fundamentais para que os modelos de realização dos requisitos sejam elaborados. É esperado que o arquiteto mantenha forte colaboração com o projetista. Em algumas situações o arquiteto também pode assumir o papel do projetista.

Analista de Testes

A arquitetura fornece toda a agenda dos testes técnicos, como por exemplo os testes de segurança, acessibilidade, desempenho ou maturidade. Em algumas organizações, até existe um papel chamado de engenheiro ou arquiteto de testes que estabelece uma comunicação com o time de arquitetura.

Gestão de Projetos

A arquitetura gera toda a agenda de riscos técnicos do projeto e também apoia a gestão na priorização dos cenários que devem ser realizados nas fases preliminares do projeto. Conforme o estilo adotado pelo projeto, o próprio time que será alocado pelo gerente de projeto deve variar. O arquiteto e gerente também deve possuir forte colaboração no projeto.

Gestor de Configuração

Um componente da arquitetura pode ser gerido de forma autônoma no sistema de gestão de configuração. Nesse sentido um diagrama de componentes elaborado por um arquiteto fornece uma agenda para o gestor de configuração estabelecer módulos, troncos de desenvolvimento e políticas para a promoção de código entre esses troncos.

A Relação do Arquiteto com a Cultura DevOps

É escopo da arquitetura capturar atributos de qualidade importantes como a configurabilidade, manutenibilidade, testabilidade, implantabilidade ou recuperabilidade. Estes atributos estão ligados a métricas como o tempo de ciclo, tempo de recuperação e % de retrabalho, que são elementos centrais das práticas DevOps. Portanto, existe uma relação forte do trabalho de arquitetura com a implantação de práticas e ferramentas DevOps (ver capítulo 11).

1.5 Para Saber Mais

O tema de arquitetura de software é bastante extenso. Um bom arquiteto deve contar com bons livros de referência para projetar boas arquiteturas em projetos complexos Web. Indico para você o enxoval da noiva do arquiteto.

- Um primeiro livro é ***Software Architecture in Practice*** (Bass, Clements, & Kazman, 2012). Esse clássico sobre arquitetura de *software* cobre aspectos técnicos e gerenciais necessários para um bom projeto de arquitetura e apresenta excelentes exemplos e casos reais da aplicação dos conceitos aos projetos.
- Um segundo livro é o ***Evaluating Software Architectures – Methods and Case Studies*** (Clements, Kazman, & Klein, 2001), que lida com o aspecto de avaliar a qualidade de sistemas legados e novos projetos. Em particular, o método ATAM (*Architecture Tradeoff Analysis Method*), usado para avaliar arquiteturas de *software*, é explicado e exemplificado com bastante clareza.
- Um terceiro livro é sobre como documentar arquiteturas de software. O livro ***Documenting Software Architectures*** (Clements et al., 2010) fornece valiosos conselhos sobre como expressar arquiteturas de *software* para analistas, gerentes, técnicos, testadores e outros interessados no produto. Esse último livro conta a colaboração de Paulo Merson, arquiteto Brasileiro que trabalha no SEI.
- Um quarto livro é a chave para ligar a arquitetura à gestão de projetos. É o livro ***Architecture Centric Software Project Management: A Practical Guide*** (Paulish, 2012), que mostra como arquitetos podem (e devem) apoiar o trabalho do gerente de projetos no sentido de criar estruturas de projeto centradas em arquitetura.
- Por último, mas não menos importante temos o livro de Eoin Woods, chamado ***Software Systems Architectures: Working With Stakeholders Using Viewpoints and Perspectives***, que é uma referência mais leve, porém abrangente sobre a temática de arquitetura de softwares. (Rozanski & Woods, 2005) e também uma leitura obrigatória para arquitetos.

2 Requisitos Arquiteturais

2.1 O que são Requisitos Arquiteturais

Um requisito expressa uma condição ou uma capacidade que um sistema deve oferecer a seus usuários. Alguns tipos comuns de requisitos incluem requisitos funcionais e não-funcionais.

Um outro tipo de requisito é o chamado **requisito arquitetural**. Um requisito arquitetural é aquele que tem impacto na arquitetura de um software. Podemos definir um requisito como arquitetural se ele satisfaz aos dois princípios abaixo:

- Alto valor para o negócio;
- Complexidade tecnológica;

Um arquiteto, de posse dos requisitos funcionais e não-funcionais de um sistema, pode usar o esquema apresentado na Figura 3 para definir a agenda de arquitetura.

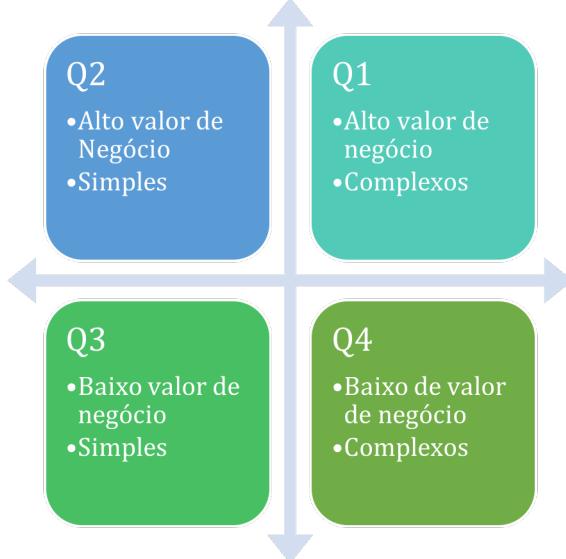


Figura 3: Tipos de requisitos em sistemas Web.

Atenção! Nem todo requisito não-funcional é um requisito arquitetural. Em projetos de sistemas alguns requisitos podem ser complexos, mas se não apresentarem valor de negócio alto não serão alvo da arquitetura não serão requisitos arquiteturais.

Dica: Requisitos arquiteturais são contextuais, i.e., são selecionados baseado no contexto específico de cada projeto e suas condições de execução. Um mesmo requisito pode ser arquitetural em um projeto e não o ser em outro projeto ou se trabalhado por outro time.

2.2 Como Descrever Bons Requisitos Arquiteturais

Uma vez que um arquiteto tenha selecionado a sua agenda de requisitos, ele precisa expressá-lo com precisão. É comum que usuários ou analista de requisitos tragam requisitos com uma escrita imprecisa e subjetiva. Exemplos incluem: “O carrinho de compras deve ser amigável” ou “O tempo de resposta deve ser bom”.

Uma técnica que arquitetos podem usar para refinar requisitos arquiteturais é a SMART. Requisitos SMART são aqueles que atendem a cinco critérios:

- *Specific* (Específicos). Um requisito deve ser específico. Exemplos de requisitos específicos indicam o caso de uso, tela ou módulo a que eles se referem.
- *Mensurable* (Mensurável). Requisitos arquiteturais devem ser mensuráveis, sempre que possível. Isso permite que ele seja testado. Sem dúvida este é um dos pontos mais complexos do modelo SMART.
- *Attainable* (Atingível). Esta característica verifica se um requisito é atingível, independente dos recursos e tempo disponível. Pode parecer que todo requisito é atingível mas existem contraexemplos. Se escrevêssemos que a tela de cadastro de alunos precisa ter disponibilidade de 100%, estariámos prometendo algo que não pode ser cumprido. Todo programa computacional opera sobre máquinas, que apresentam tempos médios entre falhas (MTBF) em suas especificações.
- *Realizable* (Realizável). Vários requisitos são atingíveis, mas nem todos o são no contexto de um projeto. Projetos têm tempo limitado e recursos físicos e financeiros limitados. Se, por exemplo, buscássemos que um sistema Web operasse em um ambiente tolerante a falhas, mas não tivéssemos orçamento para buscar máquinas de redundância, esse requisito não seria realizável.
- *Traceable* (Rastreável). Determina a origem e validade sobre o requisito. Alguns estudos mostram quase 50% dos requisitos de softwares em produção são pouco usados. Um dos motivos para isso é a introdução de requisitos nos softwares as vezes ocorre de forma indisciplinada e as vezes pelos próprios desenvolvedores, sem autorização dos usuários. Desta forma, é importante saber a origem de cada requisito.

Vamos observar a técnica SMART em ação a partir de um exemplo ambíguo.

- Requisito ambíguo: “*O sistema deve ser rápido e capaz de processar grandes quantidades de requisições simultâneas.*”
- Requisito SMART: “*A tela de cadastro de usuários deve possuir um tempo de resposta menor que 8 segundos e suportar 20 usuários simultâneos em horários de pico (15:00 às 19:00).*”

2.3 Requisitos de Acessibilidade e Usabilidade

Requisitos ligados à universalização do acesso a páginas Web em diferentes dispositivos e diferentes perfis de usuários. Um exemplo seriam páginas que funcionem em grandes monitores, tablets ou telefones celulares. Um outro exemplo seriam sítios Web que podem ser operados por pessoas com limitações de visão e até mesmo cegueira.

O W3C mantém uma especificação dedicada a esse tema, chamada WCAG (Guia de Acessibilidade de Conteúdo Web) -

“O cumprimento destas diretrizes fará com que o conteúdo se torne acessível a um maior número de pessoas com incapacidades, incluindo cegueira e baixa visão, surdez e baixa audição, dificuldades de aprendizagem, limitações cognitivas, limitações de movimentos, incapacidade de fala, fotossensibilidade bem como as que tenham uma combinação destas limitações. Seguir estas diretrizes fará também com que o conteúdo Web se torne mais usável aos utilizadores em geral.”, WCAG 2.0¹.

O governo federal brasileiro desenvolveu uma iniciativa similar, chamada eMAG, que foi normatizada através de uma portaria governamental.

“A primeira versão do eMAG foi disponibilizada para consulta pública em 18 de janeiro de 2005 e a versão 2.0 já com as alterações propostas, em 14 de dezembro do mesmo ano. A terceira versão do Modelo de Acessibilidade em Governo Eletrônico (eMAG 3.0) foi lançada em 21 de setembro de 2013, no evento Acessibilidade Digital – um direito de todos, trazendo uma seção chamada “Padronização de acessibilidade nas páginas do governo federal” com o intuito de uniformizar os elementos de acessibilidade que devem existir em todos os sítios e portais do governo. A versão 3.1 apresenta diversas melhorias no conteúdo do

¹ <https://www.w3.org/Translations/WCAG20-pt-PT/>

texto para torná-lo mais comprehensível, com destaque para o subitem “O processo para desenvolver um sítio acessível”, que ganhou um capítulo próprio. Também foram inseridos novos exemplos, inclusive com o uso de HTML5 e WAI-ARIA para determinadas recomendações.”, eMAG.²

Recomendação Arquitetural: O WCAG e eMAG permitem avaliar o nível de acessibilidade de um sítio Web em três níveis: A, AA ou AAA. O W3C mantém uma lista de utilitários para isso aqui³.

2.4 Requisitos de Autenticação e Autorização

A autenticação permite identificar quais os usuários válidos para usar uma determinada aplicação. A autorização (ou controle de acesso) permite identificar que páginas, recursos ou dados um determinado usuário autenticado pode acessar.

De forma geral, os requisitos de autenticação podem ter grande variabilidade dentro de uma aplicação e podem incluir:

- Uso de dois ou mais fatores de autenticação (ex. Senhas/*tokens* ou senhas/biometria);
- Uso de mecanismos de validação que impeçam o acesso por robôs, chamados de CAPTCHAs;
- Garantia de uso de senhas fortes;
- Senhas únicas para acesso a várias aplicações (SSO – *Single Sign On*).
- Como estruturar listas de acesso de controle.

Um sítio com muitas informações para suporte ao trabalho arquitetural de autenticação em aplicações Web é o OWASP (*Open Web Application Security Project*). Alguns guias de requisitos arquiteturais Web para autenticação e autorização podem ser encontrados aqui:

- Dicas práticas para autenticação de aplicações Web⁴
- Guia para autenticação em aplicações Web⁵
- Guia para autorização em aplicações Web⁶
- Guia de controle de acesso em aplicações Web⁷

2.5 Requisitos de Confidencialidade e Integridade

A integridade em sistemas Web diz respeito a garantia que as mensagens não são adulteradas. Juntos esses atributos garantem um transporte seguro das informações. Embora o HTTPS seja um exemplo prático da implementação de transporte seguro, não é suficiente para um arquiteto Web apenas recomendar o uso desse protocolo e esquecer de possíveis problemas associados ao transporte de informações Web. Exemplos recentes de vulnerabilidade em implementações SSL publicados pela mídia indicam a gravidade do assunto⁸.

Conhecer problemas associados à criptografia em aplicações Web é importante para o arquiteto Web. Alguns desses aspectos incluem:

- Algoritmos fracos de criptografia;
- Chaves fracas;
- Transmissão insegura.

O OWASP mantém algumas boas referências para consulta sobre criptografia Web, tais como:

- Guia de criptografia Web⁹

³ <https://www.w3.org/WAI/ER/tools/>

⁴ https://www.owasp.org/index.php/Authentication_Cheat_Sheet

⁵ https://www.owasp.org/index.php/Guide_to_Authentication

⁶ https://www.owasp.org/index.php/Guide_to_Authorization

⁷ https://www.owasp.org/index.php/Access_Control_Cheat_Sheet

⁸ O sítio <https://drownattack.com> apresenta em detalhes um exemplo de vulnerabilidade encontrada no SSL2 e como se prevenir.

⁹ https://www.owasp.org/index.php/Guide_to_Cryptography

- Dicas Práticas de criptografia Web¹⁰

2.6 Requisitos Amplos de Segurança

A segurança Web vai além da autenticação, autorização ou transporte seguro. Aspectos como a gerência de sessão de usuários, auditoria, entrada maliciosa de dados (ex. *SQL injection*), tratamento de falhas, entre outros. Nessa direção, é importante que o arquiteto Web possa contar com uma boa referência de requisitos arquiteturais Web. O OWASP mantém uma lista de verificação de requisitos arquiteturais ao longo de quatro níveis de maturidade e 15 dimensões de segurança¹¹. Para aplicações tradicionais, ela pode ser usada como uma lista de verificação rápida para capturar omissões. Para aplicações Web de segurança crítica, como Home-Banking e afins, ela pode ser usada como um critério formal para testes de segurança ao longo dos seus níveis de maturidade.

2.7 Requisitos de Alta Disponibilidade

Esse requisito lida com o tempo de permanência de uma aplicação Web em um ambiente de produção. A disponibilidade pode assumir vários níveis, conforme mostrado na Tabela 1.

Nível	SLA de Disponibilidade	Como comunicar	Recomendação de Adoção
1	50% em base diária	Horário comercial	Aplicações de Intranet/escritório.
2	90% em base diária	Não mais que 2 horas por dia de indisponibilidade.	Aplicações de Intranet para empresas com três turnos.
3	99% em base mensal	Não mais que 7 horas por mês de indisponibilidade.	Alta disponibilidade (portais de empresas)
4	99,9% em base mensal	Não mais que 1 hora por mês de indisponibilidade.	Altíssima disponibilidade. (e-commerce)
5	99,99% em base mensal	Não mais que 5 minutos por mês de indisponibilidade.	Missão crítica. (Bancos, Telecom)

Tabela 1: Níveis de disponibilidade em aplicações Web

Essa tabela pode ser útil para apresentar e discutir a disponibilidade de uma aplicação Web. Os níveis 1 e 2 podem ser implementados com soluções de baixo custo técnico. Mas as soluções de nível 3, 4 e 5 apresentam custos altos de infraestrutura, caso ela seja montada dentro da própria empresa. Com o advento de nuvens, é possível ter soluções de nível 3 ou 4 com custo razoável até mesmo para pequenas empresas.

2.8 Requisitos de Tolerância a Falhas

Esse requisito lida com o tempo de recuperação de uma aplicação Web em um ambiente de produção. A recuperabilidade pode assumir vários níveis, conforme mostrado na Tabela 2.

Nível	SLA de Recuperabilidade	Recomendação de Adoção
1	Até 1 dia	Aplicações de Intranet/escritório.

¹⁰ https://www.owasp.org/index.php/Cryptographic_Storage_Cheat_Sheet

¹¹ A versão 3.0 do *Application Security Verification Standard*, publicada em Outubro de 2015, está disponível em <https://www.owasp.org/images/6/67/OWASPAplicationSecurityVerificationStandard3.0.pdf>

2	Até 4 horas	Aplicações de Intranet para empresas com três turnos.
3	Até 1 hora	Aplicações de Internet com alta disponibilidade
4	Menor que 5 minutos	Aplicações de Internet de missão crítica. (Bancos, Telecom)

Tabela 2: Níveis de recuperabilidade em aplicações Web

2.9 Requisitos de Performance

Esse requisito lida com tempo de resposta de aplicações Web. A performance pode assumir vários níveis, conforme o tipo de requisito funcional em um Sistema Web

A performance pode assumir vários níveis, conforme mostrado na Tabela 3.

Nível	SLA de Resposta	Tempo de Recomendação de Adoção
1	Até 20 segundos	Relatórios online
2	Até 6 segundos	Consultas e entradas de dados para aplicações de escritório.
4	Até 2 segundos	Consultas e entradas de dados para aplicações comerciais (PDVs, Call-centers)
5	Até 0,1 segundos Instantâneo	Consultas e entradas de dados para aplicações comerciais críticas

Tabela 3: Níveis de tempo de resposta em aplicações Web

O arquiteto Web pode usar essa tabela para capturar o nível de recuperabilidade Web desejado.

2.10 Para Saber Mais

Um artigo seminal e obrigatório sobre o tema é o *Capturing Architectural Requirements* (Eeles, 2005), que apresenta uma tabela de requisitos para suporte ao trabalho do arquiteto. O método de elicitação de atributos de qualidade do SEI chamado QAW (Barbacci et al., 2003) é uma boa fonte para que arquitetos se aprofundem no processo de captura de requisitos arquiteturais.

3 Estilos Arquiteturais Básicos

Um estilo arquitetural é uma escolha crucial que o time de arquitetura deve fazer em um projeto. Ele representa uma abordagem estratégica para projetar, implementar e distribuir uma aplicação. Dentro do campo da arquitetura de software para TI, existem diversos estilos arquiteturais disponíveis, cada um com suas próprias vantagens e desvantagens. É importante que um arquiteto esteja familiarizado com esses estilos e seja capaz de avaliá-los de acordo com o contexto específico do projeto. É importante notar que não existem estilos arquiteturais universais que sejam melhores ou piores em todas as situações. A escolha do estilo arquitetural deve ser feita com base no contexto específico do projeto e na análise dos requisitos do usuário.

3.1 Estilo Arquitetural em Camadas

O estilo arquitetural em camadas é uma abordagem popular para projetar e construir aplicações de software. Como o nome sugere, essa abordagem divide a aplicação em camadas, cada uma com uma função específica.

Essas camadas podem ser físicas, ou seja, existir em diferentes máquinas ou ambientes, ou lógicas, existindo dentro de uma mesma máquina.

Considere um exemplo simples em três camadas lógicas. A camada mais baixa é responsável por lidar com tarefas relacionadas aos recursos físicos, como acesso a banco de dados e comunicação com dispositivos externos. A camada intermediária é responsável por lidar com tarefas de negócios e lógicas de aplicação. A camada superior é responsável pela interface com o usuário e geralmente é a camada mais visível para o usuário final. Cada camada é isolada das outras, o que permite uma maior flexibilidade e escalabilidade na construção e manutenção da aplicação. Além disso, essa abordagem também facilita a identificação e correção de problemas, uma vez que as responsabilidades de cada camada são claramente definidas. O estilo arquitetural em camadas é amplamente utilizado em aplicações de vários setores, como financeiro, saúde, comércio eletrônico, entre outros.

Essa abordagem é altamente escalável e flexível, tornando-a uma das mais populares entre os arquitetos de software.

3.1.1 Princípio Aberto Fechado

O princípio aberto-fechado é um princípio fundamental da orientação a objetos e é amplamente aplicado na arquitetura de software. Ele afirma que as classes, módulos ou componentes devem ser projetados de tal forma que possam ser estendidos para adicionar novas funcionalidades, mas não modificados para corrigir bugs ou alterar comportamentos existentes.

No contexto do estilo arquitetural de camadas, o princípio aberto-fechado é aplicado para garantir que cada camada seja isolada das outras e possa ser modificada sem afetar o funcionamento geral do sistema. Por exemplo, a camada de acesso a banco de dados pode ser modificada para suportar uma nova base de dados sem afetar as outras camadas. Da mesma forma, a camada de interface com o usuário pode ser modificada para suportar novos dispositivos sem afetar as outras camadas.

O princípio aberto-fechado também é importante para garantir a escalabilidade e flexibilidade do sistema, pois permite que novas funcionalidades sejam adicionadas sem precisar modificar as camadas existentes. Isso torna mais fácil manter e evoluir o sistema ao longo do tempo.

3.1.2 Camadas Físicas

Uma aplicação web com arquitetura em camadas físicas pode usar esse modelo para distribuir as camadas em diferentes servidores. A camada de apresentação pode ser executada em um servidor web, a camada de negócios em outro servidor e a camada de dados em um banco de dados. Isso permite escalabilidade, segurança e fácil manutenção.

Outro uso comum da arquitetura em camadas físicas é aplicações distribuídas. Essas aplicações podem usar a camada de apresentação em diferentes dispositivos, como computadores, smartphones ou tablets, e as camadas de negócios e dados em outro lugar, como nuvem ou data center. Isso permite que a aplicação seja acessível de qualquer lugar e em diferentes dispositivos.

Em resumo, a arquitetura em camadas físicas é uma abordagem comum para projetar aplicações de software, onde cada camada é responsável por uma funcionalidade específica e essas camadas existem em diferentes máquinas ou ambientes, permitindo escalabilidade, segurança e fácil manutenção.

3.2 Estilo MVC

O estilo arquitetural MVC (Model-View-Controller) é amplamente utilizado na construção de aplicações que precisam desacoplar a lógica de interface gráfica dos controles e domínios. Ele é um exemplo de estilo em camadas.

MVC – Model View Controller

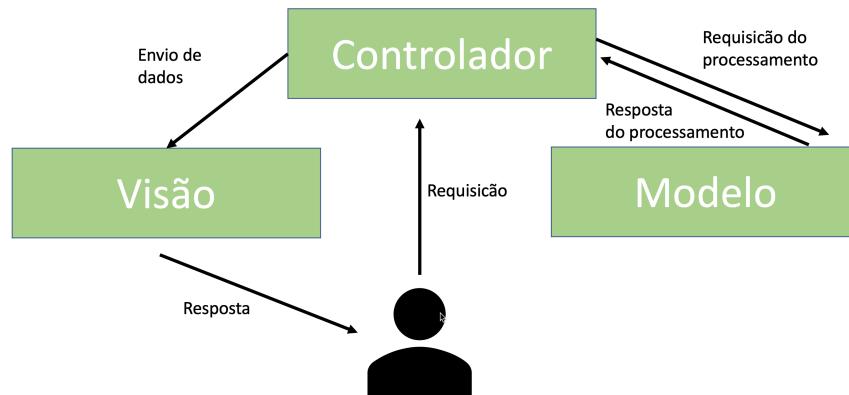


Figura 4: Padrão Arquitetural MVC

Vamos ver cada um deste componente em detalhes:

- **Modelo** - O componente modelo armazena dados e lógica relacionada. Ele representa dados que estão sendo transferidos entre componentes do controlador ou qualquer outra lógica de negócio relacionada. Por exemplo, um objeto Controlador ajuda você a recuperar as informações do cliente do banco de dados. Ele manipula dados e os envia de volta para o banco de dados ou usá-los para renderizar os mesmos dados.
- **Visão** - A Visão é aquela parte do Aplicativo que representa a apresentação de dados. As visualizações são criadas pelos dados coletados a partir dos dados do modelo. Uma visualização solicita que o Modelo dê informações para que ele renderize a saída para o usuário. A Visão também representa os dados de gráficos, diagramas e tabelas. Por exemplo, qualquer visualização do cliente incluirá todos os componentes de interface do usuário, como caixas de texto, dropdowns, etc.

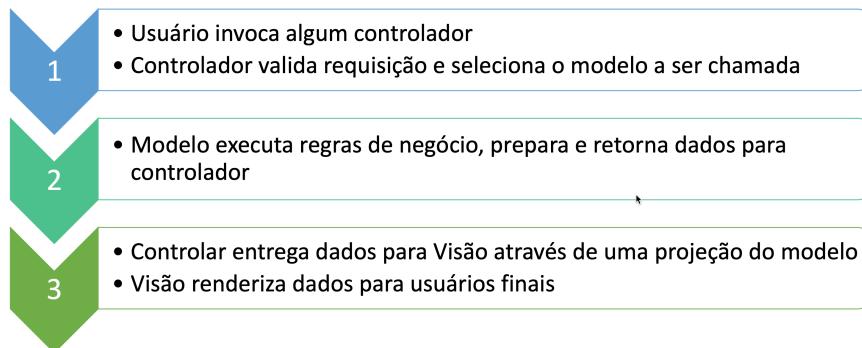
- **Controlador** - O Controlador é aquela parte do aplicativo que lida com a interação do usuário. O Controlador interpreta as entradas do mouse e do teclado do usuário, informando o Modelo e o View para mudar conforme apropriado.

No coração do MVC, e a ideia que foi a mais influente para estruturas posteriores, está a separação da apresentação. A ideia por trás disso é fazer uma divisão clara entre objetos de domínio que modelem nossa percepção do mundo real e objetos de apresentação que são os elementos de GUI que vemos na tela. Os objetos de domínio devem ser completamente independentes e funcionar sem referência à apresentação, eles também devem ser capazes de suportar várias apresentações, possivelmente simultaneamente. Essa abordagem também foi uma parte importante da cultura Unix, e continua hoje permitindo que muitos aplicativos sejam manipulados através de uma interface gráfica e de linha de comando.

A parte de apresentação do MVC é feita dos dois elementos restantes: visão e controlador. O trabalho do controlador é pegar a entrada do usuário e descobrir o que fazer com ele. O controlador, então, tem papel de mediação. Com a requisição em mãos, ele define qual modelo chamar. Dentro do modelo teremos o recheio da aplicação (regras de negócio e dados)

No MVC, o elemento de domínio é referido como o modelo (Modelo). Os modelos de objetos são completamente ignorantes da interface do usuário. O modelo contém regras de negócio locais as entidades e a manipulação dos dados ali contidos. Observe que não existe apenas uma visão e um controlador, você tem um par de controlador de visualização para cada elemento da tela, cada um dos controles e a tela como um todo.

A partir da figura anterior, tipicamente temos a seguinte ordem de execução em chamadas MVC.



Essas três camadas trabalham juntas para garantir que a aplicação seja escalável e flexível. A camada de modelo é isolada da camada de visualização, o que permite que elas possam ser modificadas independentemente uma da outra. O controlador gerencia a lógica de comunicação entre as camadas, garantindo que a aplicação siga a arquitetura MVC.

Alguns exemplos de frameworks web que usam o estilo MVC (Model-View-Controller) incluem:

- Ruby on Rails
- Laravel (PHP)
- Django (Python)
- ASP.NET MVC (C#)
- JSF

3.3 Estilo MVVM

O estilo arquitetural MVVM (Model-View-ViewModel) é uma variação do estilo MVC que é amplamente utilizado na construção de aplicações web mais modernas e é também um exemplo de estilo em camadas.

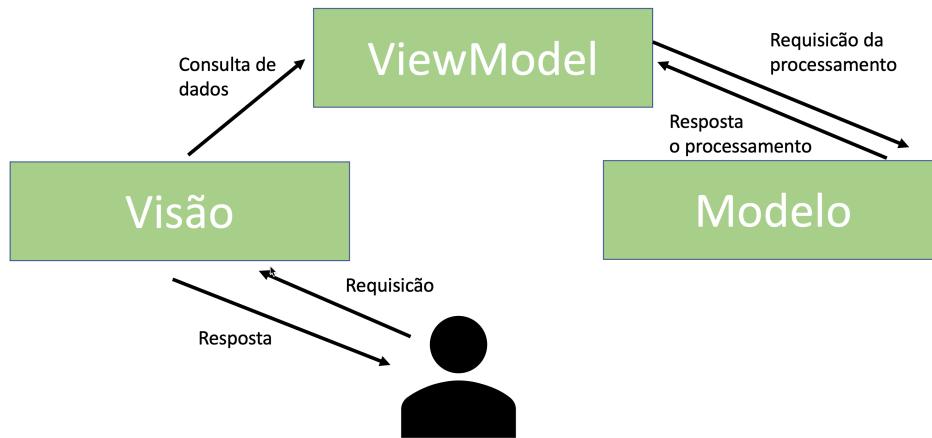


Figura 6: Padrão Arquitetural MVVM

O padrão MVVM (Model-View-ViewModel), similar ao MVC, ajuda a separar corretamente a lógica de negócios e de apresentação de um aplicativo da interface do usuário. Manter uma separação clara entre a lógica do aplicativo e a interface do usuário ajuda a resolver inúmeros problemas de desenvolvimento e pode facilitar o teste, a manutenção e a evolução de um aplicativo.

A arquitetura MVVM oferece vinculação de dados bidirecional entre o modelo de exibição e o modelo de exibição. Ele também ajuda você a automatizar a propagação de modificações dentro do View-Model para a exibição. O modelo de visão faz uso do padrão de observador para fazer alterações no modelo de exibição.

Vamos ver um ao outro neste componente em detalhes:

- **Modelo** - O modelo armazena dados e lógica relacionada. Ele representa dados que estão sendo transferidos entre componentes do controlador ou qualquer outra lógica de negócio relacionada.
- **Visão** – A Visão significa componentes de interface do usuário como HTML, CSS, jQuery, etc.
- **Modelo da visão (View Model)** - O modelo de visualização é responsável por apresentar funções, comandos, métodos, para apoiar o estado do View. Também é responsável por operar o modelo e ativar os eventos no View.

O MVVM foi desenvolvido a partir dos anos 90 para a escrita de aplicações Desktop. Ele foi popularizado por modelos visuais tais como o Java Swing, Microsoft Silverlight e afins. Nos últimos anos, com a crescente complexidade das interfaces Web, ele foi adotado em frameworks tais como Angular, React ou Vue.js.

A ordem de chamadas no MVVM é descrita na figura abaixo.

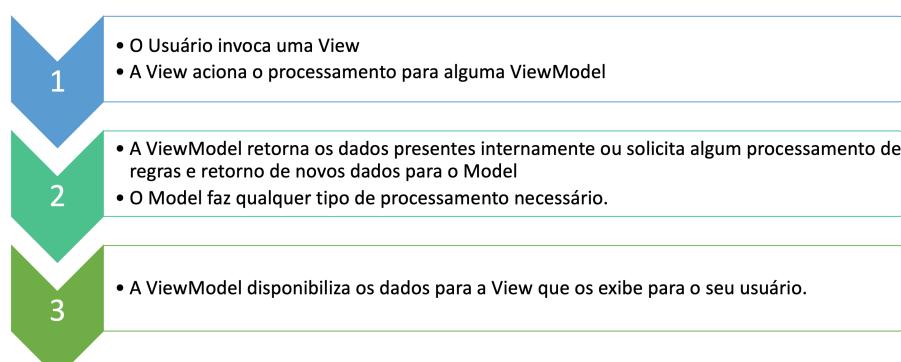


Figura 7: Ordem de Chamada no MVVM

Alguns exemplos de frameworks web que usam o estilo MVVM (Model-View-ViewModel) incluem:

- AngularJS (JavaScript)
- Vue.js (JavaScript)
- React.js (JavaScript)
- Xamarin.Forms (C#)
- WPF (C#)

3.4 DDD – Desenho Dirigido por Domínios

3.4.1 O que é o DDD

O Domain Driven Design é um estilo popularizado por Eric Evans para habilitar a criação de código realmente orientado por objetos e que forneça melhor manutenibilidade e testabilidade de aplicações. Ele fornece, antes de tudo, **comunicação entre desenvolvedores, analistas de negócio e clientes**. No DDD, a modelagem e implementação andam juntas.

O DDD (Domain-Driven Design) é uma abordagem arquitetural que se concentra no desenvolvimento de um modelo de domínio rico e expressivo, que captura os conceitos, regras e processos de negócios de um sistema. Ele enfatiza a separação clara das responsabilidades entre as diferentes camadas da aplicação, com uma ênfase especial no domínio.

3.4.2 DDD versus MVC e MVVM

Abordagem	Área de foco	Componentes principais	Separação de responsabilidades
DDD	Design de software	Interface, aplicação, domínio e infraestrutura	Separação clara das responsabilidades, com ênfase especial no domínio
MVC	Interface de usuário	Modelo, visualização e controlador	Separação das responsabilidades entre a lógica de apresentação e a lógica de negócios
MVVM	Interface de usuário	Modelo, visualização e view model	Separação das responsabilidades entre a lógica de apresentação e a lógica de negócios, com um intermediário para acessar e manipular os dados do modelo

3.4.3 Conceitos DDD

O DDD fornece uma estrutura para o desenvolvimento de software que estabelece um mecanismo para a interação entre o desenvolvedor e o negócio. O foco não é apenas na tecnologia, mas também na compreensão do domínio do negócio e na tradução desses conhecimentos em uma solução de software que atenda às necessidades do negócio. Continuar reading DDD é uma abordagem de arquitetura de software que visa desenvolver um projeto de software em torno de um modelo de domínio.

Especialistas do domínio (usuários, analistas e outros especialistas da área), juntamente com os desenvolvedores e arquitetos trabalham de mãos dadas com um objetivo em comum: construir software guiado pelo domínio para atender as necessidades do cliente. Para fazer isso, em primeiro lugar, é necessário que todos usem uma linguagem em comum e que não haja tradução na comunicação entre os membros do time. O time desenvolve, portanto, uma linguagem ubíqua (geral e universal) que expressa o conhecimento dos especialistas de negócio para o modelo de domínio (para o código).

O DDD se baseia em uma série de princípios, como o princípio da linguagem ubíqua, que diz que todos os participantes de um projeto de software devem conversar sobre o problema comum usando a mesma língua. Isso possibilita uma compreensão mais profunda da solução do problema, pois todos os envolvidos têm um entendimento comum do domínio.

Outro princípio importante é o da modelagem de domínio, que continua reading que diz que as soluções de software devem ser modeladas de acordo com os requisitos do negócio. Isso significa que os desenvolvedores devem se concentrar na criação de modelos que descrevam o domínio do negócio e não na criação de código. Essa abordagem é importante para garantir que o código seja simples e escalável.

Além disso, o DDD também se preocupa com o uso de design patterns, ou seja, com as melhores práticas para a implementação de software. Esses padrões ajudam os desenvolvedores a implementar soluções de software de maneira eficiente e consistente.

3.4.4 Linguagem Ubíqua no DDD

A linguagem ubíqua (também conhecida como linguagem onipresente ou linguagem do domínio) é uma linguagem comum compartilhada por todos os membros de uma equipe de desenvolvimento de software e pelos usuários do sistema em questão. A linguagem ubíqua é projetada para ajudar a equipe de desenvolvimento a entender o domínio do problema (o ambiente em que o software é usado e as regras de negócios que regem sua operação) e para ajudar a garantir que todos os membros da equipe estejam falando a mesma língua. A linguagem ubíqua é uma parte importante da abordagem de Design Orientado a Domínio (DDD) e é essencial para a criação de um modelo de domínio compartilhado.

A linguagem ubíqua é composta de termos específicos do domínio, jargões e expressões que descrevem os conceitos e processos dentro do domínio do problema. Esses termos são escolhidos para representar os conceitos do domínio de forma clara e concisa e para que possam ser compreendidos facilmente por todos os membros da equipe. A linguagem ubíqua deve ser usada tanto na documentação quanto no código do software.

Ao desenvolver um software, é importante que todos os membros da equipe entendam os termos e conceitos do domínio para que possam trabalhar juntos de forma eficiente e eficaz. A linguagem ubíqua ajuda a criar um modelo de domínio compartilhado que pode ser usado para comunicar com precisão os requisitos de negócios e definir as funcionalidades do software.

Além disso, a linguagem ubíqua ajuda a evitar ambiguidades e confusões que podem surgir quando diferentes membros da equipe usam termos diferentes para se referir aos mesmos conceitos. A linguagem ubíqua ajuda a garantir que todos os membros da equipe tenham uma compreensão clara dos requisitos de negócios e das funcionalidades do software, e isso pode ajudar a reduzir a probabilidade de erros e problemas de comunicação.

3.4.5 Exemplo de Linguagem Ubíqua

No contexto de e-commerce, a linguagem ubíqua poderia incluir os seguintes termos:

1. Produto - Um item que pode ser comprado no site.
2. Carrinho de compras - O local onde os produtos são armazenados antes de serem comprados.
3. Checkout - O processo de finalização da compra.
4. Frete - A taxa cobrada para entrega do produto.
5. Pagamento - O processo de pagamento pelos produtos comprados.
6. Cupom de desconto - Um código que pode ser inserido para receber um desconto na compra.
7. Endereço de entrega - O endereço onde o produto será entregue.
8. Estoque - A quantidade de um produto disponível para venda.
9. Avaliação do produto - Uma classificação ou comentário dado por um cliente sobre um produto.
10. Cliente - Uma pessoa que compra produtos no site.
11. Pedido - Uma solicitação de compra de um ou mais produtos.
12. Entrega expressa - Uma opção de entrega mais rápida, mas com custo adicional.
13. Favoritos - Uma lista de produtos salvos pelo cliente para comprar mais tarde.
14. Troca ou devolução - O processo de retornar ou trocar um produto comprado.
15. Avaliação do cliente - Uma classificação ou comentário dado pelo cliente sobre a experiência de compra no site.

3.4.6 Modelo de Domínio e o DDD

O modelo de domínio uma representação estruturada e organizada das entidades, regras e comportamentos que compõem o domínio de negócio de uma aplicação. Ele é um elemento central do padrão arquitetural Domain-Driven Design (DDD), que tem como objetivo criar software que reflita de forma precisa o domínio de negócio de uma empresa.

Ele é construído por meio da identificação e mapeamento de entidades, agregados, eventos, serviços e regras de negócio. Essas entidades representam objetos do mundo real ou abstrações do sistema, enquanto os agregados são grupos de entidades que compartilham um ciclo de vida e consistência transacional.

Os eventos são usados para registrar a ocorrência de uma ação ou mudança de estado no domínio, enquanto os serviços representam operações que não pertencem a uma única entidade ou agregado e podem envolver várias entidades. Por fim, as regras de negócio são definidas como restrições ou condições que devem ser satisfeitas para garantir a consistência e a integridade do sistema.

Ele é implementado por meio de classes, interfaces e outros elementos da linguagem de programação escolhida. Ele deve ser projetado de forma independente da tecnologia ou infraestrutura utilizada, permitindo que a lógica de negócio seja facilmente adaptável e modificada sem afetar a arquitetura geral da aplicação.

Ele é uma ferramenta valiosa para a comunicação e colaboração entre membros da equipe de desenvolvimento e os stakeholders do projeto, incluindo especialistas do domínio e usuários finais. Ele ajuda a garantir que o software desenvolvido atenda às necessidades do negócio e seja capaz de evoluir ao longo do tempo.

Com um modelo do domínio conseguimos:

- O modelo **serves como algo comum e palpável a todos os membros do time**, que, junto com a linguagem ubíqua, permite que todos possam participar ativamente da construção progressiva do mesmo.
- O modelo (desde que feito corretamente) **garante que aquilo que está sendo especificado é o que está sendo implementado**.
- O modelo é **o meio de comunicação usado pelo time**. Graças ao vínculo entre o modelo e a implementação, os desenvolvedores podem falar na linguagem do software ao comunicarem-se com os especialistas do domínio (sem ter que traduzir a mensagem).

- O modelo é **o conhecimento destilado** — é o modo como o time concorda em estruturar o conhecimento extraído do domínio.
- O modelo é evolutivo: **A cada iteração entre especialistas de domínio e a equipe técnica, o modelo se torna mais profundo e expressivo, mais rico, e os desenvolvedores transferem essa fonte de valor para o software.**
- Assim, o modelo vai sendo gradualmente enriquecido com a sabedoria dos especialistas do domínio destilado pelos desenvolvedores, **fazendo com que o time ganhe cada vez mais sabedoria sobre o negócio** e que esse conhecimento seja transferido para o modelo (para o código) através dos blocos de construção do DDD. Quando novas regras de negócio são adicionadas e/ou regras existentes são alteradas ou removidas, a implementação é refatorada para refletir essas alterações do modelo no código. No final, o modelo (que em última instância será o software) vai expressar com riqueza de conhecimento o negócio.

O DDD coloca um monte de conhecimento no modelo que reflete profundamente o domínio. E como o desenvolvimento é iterativo, essa colaboração continua durante todo o andamento do projeto. Ou seja, o Domain-Driven Design nos leva a construir softwares guiados pelo conhecimento e modelagem do negócio antes de qualquer apelo por tecnologia. E, por isso, o DDD nos conduz através dos seus blocos de construção a utilizar alguns princípios de arquitetura e *design patterns* consagrados, entre eles destaco:

- Isolamento do domínio com arquitetura em camadas.
- Representação do modelo através de artefatos de software bem definidos (*entities, value objects, services, factories, repositories, specs, modules, etc.*).
- Gerenciamento do ciclo de vida de objetos do domínio com o conceito de *aggregates*.

Cinco dicas para termos um modelo de domínio efetivo são:

- **Vincular o modelo com a implementação:** esse vínculo é feito desde o início, quando o modelo ainda é primitivo e será mantido até o fim. Esse vínculo é profundo, **a implementação deve refletir 100% o modelo.**
- **Cultivar uma linguagem baseada no modelo:** no início será necessário que os desenvolvedores e os especialistas no domínio entendam cada um os termos do outro, mas depois ambos falarão a mesma linguagem, organizando as sentenças da comunicação numa estrutura consistente com o modelo e sem ambiguidades.
- **Desenvolver um modelo rico em conhecimento:** objetos têm dados e comportamentos associados. O modelo não deve ser apenas uma estrutura de dados (modelo anêmico), ele deve capturar o conhecimento do domínio para resolver os problemas do domínio.
- **Destilar o modelo:** o modelo deve ser refinado. Assim como conceitos importantes devem ser adicionados, conceitos que não tem relevância devem ser removidos. **A cada iteração o modelo ficará mais rico e terá mais valor.**
- **Brainstorming e experimentação: a interação direta entre os desenvolvedores e domain experts,** através de brainstorms e diagramas feitos na hora, transformam as discussões em **laboratórios do modelo**, no qual diversas variações de experimentos podem ser exercitadas e o resultado pode ser usado se mostrar valor ou descartado caso contrário.
- Se afastar de modelos de classes anêmicas. Uma classe anêmica é uma classe com atributos apenas e sem comportamentos (métodos).

3.4.7 Exemplo de Modelo de Domínio

Vamos utilizar o exemplo de um sistema de gerenciamento de vendas de uma loja de roupas online para ilustrar o modelo de domínio com termos da linguagem ubíqua:

- Entidades: Cliente, Produto, Pedido, Carrinho de Compras, Pagamento
- Agregados: Pedido (composto por Carrinho de Compras, Produto e Cliente), Pagamento (composto por Pedido)
- Eventos: Produto Adicionado ao Carrinho, Produto Removido do Carrinho, Pedido Realizado, Pagamento Confirmado
- Serviços: Processamento de Pagamento, Consulta de Pedidos, Cálculo de Descontos

- Regras de Negócio: Produto não pode ser adicionado com quantidade zero, Desconto de 10% aplicado para pedidos acima de R\$ 100,00, Pagamento somente é confirmado após autorização da operadora de cartão.

Nesse modelo de domínio, as entidades representam os objetos do mundo real (cliente, produto, pedido, etc.) que são gerenciados pelo sistema. Os agregados ajudam a controlar o ciclo de vida das entidades e garantir a consistência transacional do sistema. Os eventos são usados para registrar a ocorrência de ações e mudanças de estado no domínio, enquanto os serviços representam operações que envolvem várias entidades. As regras de negócio são utilizadas para garantir a integridade do sistema e proteger os interesses da loja e dos clientes.

Com esse modelo de domínio, a equipe de desenvolvimento pode trabalhar de forma mais eficiente e colaborativa, pois todos têm uma visão clara dos objetos, comportamentos e regras que compõem o sistema. Isso ajuda a garantir que o software desenvolvido atenda às necessidades do negócio e seja capaz de evoluir ao longo do tempo.

3.4.8 Camadas Típicas do DDD

O esquema conceitual básico do DDD, conforme proposto por Eric Evans, é resumido na seguinte figura.

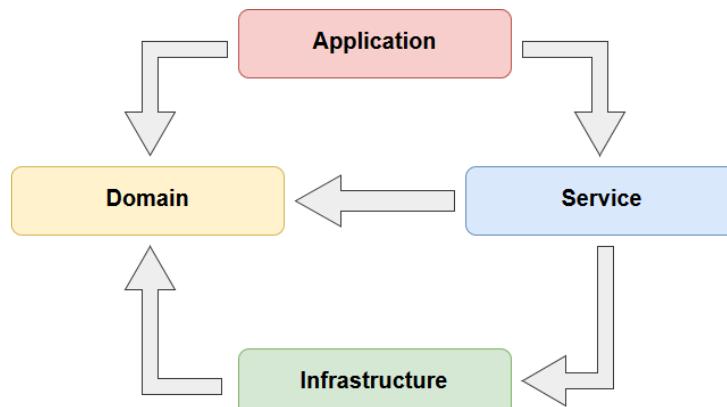


Figura 8: Camadas do DDD

A camada de Interface de Usuário (UI) é responsável por renderizar a interface gráfica para o usuário. Esta camada também é responsável por acionar as ações e eventos necessários para atender às solicitações do usuário. Esta camada depende da camada de Aplicação.

A camada de Aplicação é responsável por receber as solicitações do usuário através da camada de Interface de Usuário e traduzir estas solicitações para a camada de Domínio. Esta camada também é responsável por validar os dados e controlar os fluxos de trabalho. Esta camada depende da camada de Domínio.

A camada de Domínio é responsável por realizar o processamento de negócios, gerenciar as regras de negócio e manter os dados. Esta camada também é responsável por fornecer os recursos necessários para que a camada de Aplicação possa executar as solicitações do usuário. Esta camada não depende de outras camadas.

Além dessas três camadas, o DDD também prevê a camada de Infraestrutura, responsável por fornecer os serviços necessários para o funcionamento do sistema, como o acesso a banco de dados, serviços de filas de mensagens, segurança e outros. A infraestrutura cuida de toda a tecnologia. Ela depende da camada de domínio. Mas a camada de domínio não a conhece.

3.4.9 Dependências entre Camadas no DDD

As relações de dependências entre as camadas podem ser resumidas da seguinte forma:

- A camada de interface depende da camada de aplicação para obter as informações do modelo de domínio e executar as operações de negócio.
- A camada de aplicação depende da camada de domínio para executar as regras de negócio e gerenciar as entidades e objetos de valor.
- A camada de domínio é a camada central do sistema, que contém as entidades, objetos de valor e regras de negócio do modelo de domínio. Ela não depende diretamente de nenhuma outra camada.
- A camada de infraestrutura é responsável por fornecer as implementações das interfaces de infraestrutura do domínio, como os repositórios, serviços de mensageria, serviços de autenticação, etc. A camada de infraestrutura depende da camada de domínio para definir essas interfaces e é responsável por implementá-las.

Essas relações de dependências ajudam a garantir a modularidade e independência das camadas, permitindo que cada camada evolua de forma independente, sem afetar as outras camadas. Por exemplo, a camada de infraestrutura pode ser facilmente substituída por uma implementação diferente, sem afetar a lógica de negócios ou a camada de domínio.

3.4.10 O domínio é ignorante

No DDD, existem dois padrões relacionados à separação de preocupações: Ignorância de Persistência e Ignorância de Infraestrutura.

A Ignorância de Persistência refere-se à ideia de que o modelo de domínio não deve estar ciente do mecanismo de armazenamento de dados usado pelo sistema. Em vez disso, essa responsabilidade é delegada a uma camada de infraestrutura, que é responsável por fornecer uma implementação para os repositórios do domínio. Essa abordagem ajuda a reduzir o acoplamento entre a lógica de negócios e a camada de persistência, permitindo que cada uma evolua de forma independente.

Já a Ignorância de Infraestrutura diz respeito à ideia de que o modelo de domínio não deve estar ciente dos detalhes técnicos da infraestrutura usada pelo sistema, como o sistema de arquivos, protocolos de rede ou bancos de dados específicos. Em vez disso, essa responsabilidade é delegada a uma camada de infraestrutura, que é responsável por fornecer uma implementação para as interfaces de infraestrutura do domínio. Essa abordagem ajuda a tornar o modelo de domínio mais independente da infraestrutura, facilitando a evolução e manutenção do sistema.

Esses padrões ajudam a manter o modelo de domínio focado nas regras de negócios e responsabilidades relacionadas, reduzindo o acoplamento e aumentando a modularidade do sistema.

3.4.11 Exemplo de Código

Compilo abaixo um código DDD que modela a relação entre um carrinho de compras que contém vários produtos. O código é exibido dentro das classes das camadas do DDD: Infraestrutura, Domínio e Aplicação.

Domínio – Produto

```
class Produto {
    id: number;
    nome: string;
    preco: number;

    constructor(id: number, nome: string, preco: number) {
        if (preco < 0) {
            throw new Error('O preço não pode ser negativo');
        }
        this.id = id;
        this.nome = nome;
        this.preco = preco;
    }
}
```

Domínio – Carrinho

```
// Classe Carrinho que representa o carrinho de compras
class Carrinho {
    private produtos: Produto[] = [];

    adicionarProduto(produto: Produto): void {
        this.produtos.push(produto);
    }

    listarProdutos(): Produto[] {
        return this.produtos;
    }

    // Método que calcula o total dos preços dos produtos no carrinho
    calcularTotal(): number {
        let total = 0;
        for (const produto of this.produtos) {
            total += produto.preco;
        }
        return total;
    }
}
```

```
// Método que aplica um desconto de 10% no total se for maior que R$100
aplicarDesconto(): number {
    const total = this.calcularTotal();
    if (total > 100) {
        return total * 0.9; // Aplica desconto de 10%
    }
    return total;
}
```

Infraestrutura – Repositório de Produto

```
// Interface para o Repositório de Produtos
interface RepositorioProduto {
    obterPorId(id: number): Produto | undefined;
    listarTodos(): Produto[];
}

// Implementação do Repositório de Produtos
class RepositorioProdutoImpl implements RepositorioProduto {
    private produtos: Produto[] = [
        new Produto(1, "Produto 1", 50.0),
        new Produto(2, "Produto 2", 70.0),
        new Produto(3, "Produto 3", 80.0),
    ];

    obterPorId(id: number): Produto | undefined {
        return this.produtos.find((produto) => produto.id === id);
    }

    listarTodos(): Produto[] {
        return this.produtos;
    }
}
```

Aplicação – Serviço de Carrinho de Compras

```
// Classe de Serviço de Carrinho de Compras
class CarrinhoServiço {
    constructor(private repositórioProduto: RepositórioProduto) {}

    adicionarProduto(carrinho: Carrinho, idProduto: number): void {
        const produto = this.repositórioProduto.obterPorId(idProduto);
        if (produto) {
            carrinho.adicionarProduto(produto);
        } else {
            throw new Error(`Produto com ID ${idProduto} não encontrado`);
        }
    }

    listarProdutos(carrinho: Carrinho): Produto[] {
        return carrinho.listarProdutos();
    }

    // Método que calcula o total dos preços dos produtos no carrinho
    calcularTotal(carrinho: Carrinho): number {
        return carrinho.calcularTotal();
    }

    // Método que aplica um desconto de 10% no total se for maior que R$100
    aplicarDesconto(carrinho: Carrinho): number {
        return carrinho.aplicarDesconto();
    }
}
```

Controlador – Carrinho de Compras

```

import express, { Request, Response } from 'express';
import { CarrinhoService } from './carrinhoService';
import { ProdutoDTO } from './produtoDTO';

export class CarrinhoController {
    private readonly carrinhoService: CarrinhoService;

    constructor(carrinhoService: CarrinhoService) {
        this.carrinhoService = carrinhoService;
    }

    public registerRoutes(app: express.Application): void {
        app.get('/carrinho', this.listCarrinho.bind(this));
        app.post('/carrinho/produtos', this.addProduto.bind(this));
    }

    private listCarrinho(req: Request, res: Response): void {
        const carrinho = this.carrinhoService.getCarrinho();
        res.json(carrinho);
    }

    private addProduto(req: Request, res: Response): void {
        const produtoDTO: ProdutoDTO = req.body;
        const produto = this.carrinhoService.addProduto(produtoDTO);
        res.status(201).json(produto);
    }
}

```

3.4.12 Padrões de Desenho no DDD

A seguir, listamos e explicamos alguns dos principais padrões de projeto do DDD:

1. Entidade (Entity): é uma classe que representa um objeto de negócio no domínio da aplicação. Cada entidade possui uma identidade única e atributos que a caracterizam. Além disso, as entidades são mutáveis, ou seja, seus atributos podem ser alterados ao longo do tempo. É importante que as entidades sejam independentes de outros objetos da aplicação, e que possuam comportamentos e lógicas de negócio encapsulados dentro delas.
2. Objeto de valor (Value Object): é um objeto imutável que representa um valor dentro do domínio da aplicação. Diferente das entidades, os objetos de valor não possuem uma identidade única e são comparados pela igualdade de seus atributos. Os objetos de valor são geralmente utilizados para

representar tipos primitivos mais complexos, como datas, horários e endereços, e podem ser compostos de outros objetos de valor ou entidades.

3. Serviço de domínio (Domain Service): é uma classe responsável por coordenar as ações entre as entidades e objetos de valor do domínio da aplicação. Os serviços de domínio são responsáveis por implementar as regras de negócio mais complexas da aplicação, e geralmente recebem como parâmetros as entidades e objetos de valor que serão manipulados. É importante que os serviços de domínio sejam independentes da camada de aplicação e infraestrutura, e que possuam uma interface bem definida.
4. Repositório (Repository): é uma classe responsável por armazenar e recuperar as entidades da aplicação. Os repositórios são responsáveis por abstrair o acesso aos dados da aplicação, de forma que a camada de domínio possa utilizar as entidades sem se preocupar com a forma como elas são persistidas. É importante que os repositórios sejam independentes da camada de infraestrutura, e que possuam uma interface bem definida.
5. Camada de aplicação (Application Layer): é uma camada intermediária entre a camada de domínio e a camada de infraestrutura. A camada de aplicação é responsável por receber as requisições do usuário, orquestrar a interação entre as entidades e objetos de valor do domínio, e gerar as respostas adequadas. É importante que a camada de aplicação seja independente da camada de infraestrutura, e que possua uma interface bem definida.
6. Camada de infraestrutura (Infrastructure Layer): é a camada responsável por prover os recursos de baixo nível necessários para a execução da aplicação. Isso inclui o acesso ao banco de dados, serviços externos, sistema de arquivos, entre outros. É importante que a camada de infraestrutura seja independente das outras camadas da aplicação, e que possua interfaces bem definidas para se comunicar com elas.
7. Agregado (Aggregate): é um conjunto de entidades e objetos de valor que são tratados como uma única unidade no domínio da aplicação. Cada agregado possui uma entidade raiz (root entity) que é responsável por garantir a consistência das demais entidades e objetos de valor que fazem parte do agregado. Os agregados são utilizados para garantir a consistência do domínio da aplicação e para facilitar a manipulação dos objetos de negócios.
8. Evento de Domínio (Domain Event): é uma classe que representa um evento que ocorreu dentro do domínio da aplicação. Os eventos de domínio são utilizados para notificar outras partes da aplicação sobre mudanças que ocorreram no estado do domínio. Por exemplo, um evento de domínio pode ser utilizado para notificar a camada de aplicação sobre uma atualização em uma entidade, ou para notificar a camada de infraestrutura sobre a necessidade de persistir os dados.
9. Especificação (Specification): é uma classe que representa uma regra de negócio que pode ser avaliada como verdadeira ou falsa para um determinado objeto de negócio. As especificações são utilizadas para encapsular as regras de negócio mais complexas da aplicação, e para facilitar a escrita de testes automatizados. Por exemplo, uma especificação pode ser utilizada para verificar se um cliente é elegível para receber um desconto em uma compra.
10. Fábrica (Factory): é uma classe responsável por criar e inicializar objetos de negócio dentro do domínio da aplicação. As fábricas são utilizadas para encapsular a lógica de criação dos objetos, e para garantir que eles sejam criados de forma consistente. Por exemplo, uma fábrica pode ser utilizada para criar um novo cliente com todos os atributos preenchidos corretamente.

11. Serviço de Aplicação (Application Service): é uma classe responsável por orquestrar as operações de negócio da aplicação. Os serviços de aplicação são responsáveis por receber as requisições do usuário, utilizar os serviços de domínio e os repositórios para manipular as entidades e objetos de valor do domínio, e gerar as respostas adequadas. É importante que os serviços de aplicação sejam independentes da camada de infraestrutura, e que possuam uma interface bem definida.
12. Anti-Corruption Layer (ACL): é uma camada responsável por traduzir e adaptar os dados de sistemas externos para o formato utilizado dentro da aplicação. O ACL é utilizado para garantir a integridade dos dados e para isolar a aplicação das mudanças que podem ocorrer nos sistemas externos. Por exemplo, um ACL pode ser utilizado para traduzir os dados recebidos de um sistema legado para um formato que possa ser utilizado pela aplicação.

3.4.13 Riscos no DDD

Apesar de ser uma abordagem que traz diversos benefícios para o desenvolvimento de software, o DDD também apresenta alguns riscos que devem ser considerados. Seguem três possíveis riscos do uso do DDD

1. Complexidade: O DDD pode levar a uma arquitetura mais complexa e abstrata, especialmente se a equipe não tiver familiaridade com os conceitos e padrões do DDD. Isso pode dificultar a manutenção e a evolução do sistema, especialmente para equipes com menos experiência em desenvolvimento de software.
2. Sobrecarga de abstração: O DDD incentiva a criação de entidades, objetos de valor, agregados, serviços de domínio e outras abstrações para representar o modelo de negócios do sistema. Embora essa abordagem ajude a garantir a coesão e a expressividade do modelo de domínio, pode levar a uma sobrecarga de abstração se a equipe não conseguir encontrar o equilíbrio certo entre o nível de detalhe do modelo e a complexidade do código.
3. Dificuldades na integração com sistemas legados: Se o sistema estiver integrado com sistemas legados ou de terceiros, pode ser difícil aplicar os conceitos do DDD de forma consistente e efetiva. Isso pode levar a uma arquitetura híbrida, com diferentes padrões de design e estruturas de código, o que pode aumentar a complexidade e a dificuldade de manutenção do sistema.

4 Estilos Arquiteturais Avançados

4.1 Pipelines

O estilo arquitetural baseado em pipelines, também conhecido como arquitetura de fluxo de trabalho, é uma abordagem para projetar aplicações de software onde os dados ou tarefas são processados através de uma série de etapas ou pipelines. Cada pipeline é composta por uma série de tarefas ou componentes que são executadas em sequência, transformando os dados de entrada em dados de saída.

Uma das principais características desse estilo arquitetural é a separação clara entre as diferentes etapas do pipeline. Cada etapa é responsável por uma tarefa específica, como validação, transformação ou persistência de dados, e essas etapas são executadas de forma independente, permitindo escalabilidade e manutenção fácil.

Outra característica importante é a capacidade de adicionar, remover ou substituir etapas do pipeline sem afetar as outras etapas. Isso permite que a aplicação seja facilmente adaptada às mudanças de requisitos sem a necessidade de mudanças significativas na arquitetura.

Um exemplo de uso desse estilo arquitetural é em sistemas de processamento de transações, onde os dados de transações são processados através de um pipeline que inclui etapas como validação, autorização, contabilidade e persistência. Outro exemplo é em sistemas de processamento de imagens, onde os dados de imagem são processados através de um pipeline que inclui etapas como redimensionamento, filtragem e compressão.

Mais um exemplo de uso desse estilo arquitetural é no contexto de ETL (Extração, Transformação e Carga). Nesse caso, os dados são extraídos de fontes diversas, passam por processos de transformação, validação e limpeza, e são carregados em um banco de dados de destino. Cada uma dessas etapas é implementada como um componente separado, que pode ser escalado e substituído facilmente.

Outro exemplo é o uso da arquitetura baseada em pipelines com o uso de AWS Lambda Step Functions. Nessa abordagem, as etapas do pipeline são implementadas como funções AWS Lambda e gerenciadas pelo serviço Step Functions. Isso permite que os desenvolvedores construam fluxos de trabalho complexos, escaláveis e altamente disponíveis, sem se preocupar com a infraestrutura subjacente. Além disso, as funções Lambda podem ser facilmente integradas com outros serviços AWS, como SQS, SNS, DynamoDB e muito mais.

4.1.1 Pipes e Filters

Em arquitetura baseada em pipelines, os "pipes" são componentes que são responsáveis por transportar dados de uma etapa para outra. Esses componentes geralmente são implementados como canais de comunicação entre as etapas, como por exemplo, filas de mensagens, fluxos de dados ou canais de comunicação assíncronos.

Os "filters" são componentes que são responsáveis por realizar operações específicas em cima dos dados enquanto eles passam pelo pipeline. Esses componentes são geralmente implementados como funções ou módulos que são aplicados aos dados de entrada, processando-os e produzindo dados de saída.

Os pipes e filters trabalham juntos para construir o pipeline, onde os pipes são os canais que transportam os dados e os filters são os componentes que processam os dados. Essa arquitetura permite que os desenvolvedores construam sistemas escaláveis e flexíveis, pois os pipes e filters são geralmente independentes entre si, e podem ser substituídos ou escalados facilmente.

4.2 MicroKernel

O estilo arquitetural baseado em microkernel é uma abordagem que se concentra na separação dos componentes de um sistema em núcleos pequenos e modularizados, conhecidos como "microkernels". Esses microkernels são responsáveis por fornecer serviços básicos e funcionalidades ao sistema, enquanto os outros componentes, conhecidos como "módulos", são adicionados ao sistema para fornecer funcionalidades específicas.

Uma das principais vantagens deste estilo arquitetural é a flexibilidade. Como os módulos são independentes e podem ser adicionados ou removidos facilmente, o sistema pode ser adaptado facilmente às mudanças de requisitos. Além disso, os microkernels são projetados para serem estáveis e confiáveis, o que geralmente resulta em sistemas mais robustos e seguros.

Outra vantagem é a escalabilidade. Como os módulos são independentes e não estão acoplados ao núcleo, eles podem ser facilmente escalados horizontalmente, adicionando mais instâncias do módulo para lidar com aumentos na carga de trabalho. Além disso, como os microkernels são projetados para serem leves, eles geralmente consomem menos recursos, o que também contribui para a escalabilidade do sistema.

Uma outra vantagem é a segurança. Como o núcleo é responsável por fornecer apenas os recursos básicos, é menos propenso a falhas e vulnerabilidades. Além disso, como os módulos são isolados uns dos outros, uma falha em um módulo não afeta o funcionamento geral do sistema.

Um exemplo comum de um sistema baseado em microkernel é o sistema operacional de código aberto Linux. O núcleo do Linux, ou o "microkernel", fornece serviços básicos como gerenciamento de memória e gerenciamento de processos, enquanto os módulos adicionais, como o gerenciador de arquivos ext4, fornecem funcionalidades específicas. Isso permite que o sistema seja adaptado facilmente às necessidades do usuário final, como suporte a diferentes tipos de armazenamento ou protocolos de rede.

Outro exemplo de uso desse estilo no contexto do ERP SAP é a arquitetura do SAP ECC. O núcleo do sistema é composto pelos componentes de gerenciamento de banco de dados, gerenciamento de transações e gerenciamento de usuários, enquanto os módulos são responsáveis por fornecer funcionalidades adicionais, como finanças, recursos humanos e vendas. Isso permite que o sistema seja facilmente personalizado de acordo com as necessidades específicas de cada empresa.

É importante notar que a implementação de um microkernel pode ser desafiadora, pois requer uma boa compreensão da funcionalidade requerida e do sistema. Além disso, a complexidade do núcleo pode aumentar com o tempo à medida que novos recursos são adicionados, o que pode afetar a escalabilidade e a manutenção do sistema.

O Kubernetes é um exemplo de um sistema operacional baseado em microkernel. Ele fornece uma estrutura básica para gerenciar e escalar aplicativos em contêineres. O núcleo do Kubernetes é composto por componentes essenciais, como o controlador, o etcd e o kubelet, que fornecem as funcionalidades básicas de gerenciamento de recursos, armazenamento de estado e gerenciamento de contêineres. Além disso, o Kubernetes também possui uma série de plugins e extensões, como o Kubernetes Dashboard, que podem ser adicionados ao núcleo para fornecer funcionalidades adicionais. Esses plugins são desenvolvidos e mantidos por uma comunidade ativa e podem ser facilmente integrados ao núcleo do Kubernetes, sem afetar sua estabilidade e segurança.

Em resumo, o Kubernetes é uma boa exemplo de como uma arquitetura baseada em microkernel pode ser usada para criar sistemas flexíveis e escaláveis, permitindo que novos recursos e funcionalidades sejam adicionados sem afetar a estabilidade e segurança do sistema.

4.3 Serviços

O estilo arquitetural baseado em serviços é uma abordagem que se concentra na decomposição de um sistema em componentes independentes chamados de "serviços". Esses serviços são projetados para serem reutilizáveis, escaláveis e independentes de outros componentes do sistema. Eles são expostos através de uma interface de programação de aplicação (API) que é utilizada para comunicar-se com outros serviços e componentes do sistema.

Esse estilo arquitetural é baseado na ideia de que os serviços devem ser projetados para serem usados por múltiplos clientes e sistemas, e não apenas para atender a uma única aplicação. Dessa forma, os serviços podem ser reutilizados em diferentes contextos e projetos, aumentando a escalabilidade e a flexibilidade do sistema.

Os serviços também são projetados para serem independentes e isolados uns dos outros. Isso significa que eles não dependem de outros serviços para funcionar, o que aumenta a estabilidade e a segurança do sistema. Além disso, essa independência permite que os serviços sejam escalados e gerenciados de forma independente, o que aumenta a flexibilidade do sistema.

O Service-Oriented Architecture (SOA) é uma implementação desse estilo arquitetural. O SOA é uma abordagem para construir aplicações como um conjunto de serviços que se comunicam através de protocolos padronizados, como o SOAP ou o REST. Os serviços podem ser implementados em diferentes linguagens e plataformas, desde que sigam as especificações de comunicação.

4.4 Microsserviços

Uma aplicação tradicional é distribuída em produção em um ou poucos componentes executáveis (DLL ou Executáveis). Na prática isso leva que tenhamos dezenas de casos de uso em um único código executável, o que limita a velocidade de implantação de novas funcionalidades em ambientes de produção, a implantação de práticas de entrega contínua (*Continuous Delivery*) ou mesmo a adoção de novas tecnologias.

Em oposição a esse conceito, o estilo de microsserviços lida com a criação de pequenos serviços autônomos. Cada microsserviço implementa uma pequena função de negócio e pode ser implantado e removido dos ambientes de produção de forma independente. Em termos técnicos, cada microsserviço pode ser escrito em uma linguagem de implementação distinta, possui o seu próprio banco de dados e se comunica com outros serviços através de chamadas REST.

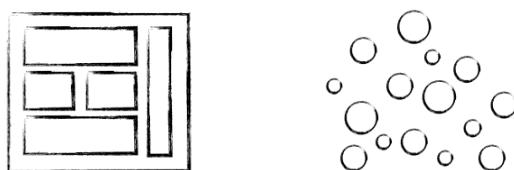


Figura 9: Estilo Arquitetural Monolítico versus Microsserviços. Cada microsserviço opera de forma distribuída, possui o seu próprio banco de dados e expõe uma API RESTful para os seus clientes.

Esse estilo foi pensado para aplicações modernas de Internet que podem ser implementadas e distribuídas em pequenas partes. Não é estilo apropriado para aplicações tradicionais com bases de dados enormes como por exemplo ERPs Web.

É comum que microsserviços utilizem o estilo DDD e API, apresentados nesse capítulo.

Os princípios que orientam este estilo incluem:

1. **Governança Descentralizada.** Uma das consequências da governança centralizada é a tendência de padronização de plataformas tecnológicas corporativas. A experiência mostra que esta abordagem pode ser restritiva e limitante para acomodar evoluções tecnológicas. Ao dividir os componentes de um monólito em microsserviços, temos possibilidades novas de escolha durante a criação de cada um deles. Você pode usar tecnologias distintas como Node.js, JavaScript ou C# para contextos específicos na sua organização.
2. **Gestão de Dados Descentralizada.** Com a abordagem de microsserviços, cada serviço gera o seu próprio banco de dados, diferentes instâncias da mesma tecnologia de banco de dados, ou sistemas diferentes de banco de dados. Esta abordagem é chamada de abordagem chamada persistência poliglota.
3. **Operação Distribuída.** Cada microsserviço opera de forma independente, com o seu próprio banco de dados e operar em seu próprio ambiente de execução chamado de micro-contêiner.
4. **Exposição de APIs.** Microsserviços expõe suas funcionalidades através de APIs bem estruturadas para a camada de apresentação. E microsserviços se comunicam através de REST/HTTP para operações síncronas e através de filas de mensagens para operações assíncronas.
5. **Uso de tecnologias simples**, como o ASP.NET Core, ASP.NET Web API, Spring Boot ou Node.js. Não usamos barramentos de serviços (os antigos ESBs) para operar arquitetura de microsserviços.

Algumas das vantagens deste estilo incluem:

1. **Cada microsserviço é pequeno, se comparado com uma aplicação tradicional.** O código é compreendido com facilidade pelo desenvolvedor. A baixa quantidade de código não torna a IDE lenta, tornando os desenvolvedores mais produtivos. Além disso, cada serviço inicia mais rápido que uma grande aplicação monolítica o que torna os desenvolvedores mais produtivos e agiliza as implantações. A arquitetura de microsserviços facilita escalar o desenvolvimento. Pode-se organizar o esforço de desenvolvimento em vários times pequenos com o uso de métodos ágeis e práticas DevOps. Cada equipe é responsável pelo desenvolvimento e implantação de um único serviço ou um conjunto de serviços relacionados e pode desenvolver, implantar e escalar seus serviços, independente de todos os outros times.
2. **Cada serviço pode ser implantado com independência de outros serviços.** Caso os desenvolvedores responsáveis por um serviço necessitem implantar uma modificação para um determinado serviço que não impacte a API deste serviço, não há necessidade de coordenação com outros desenvolvedores. Pode-se implantar as modificações. A arquitetura de microsserviços torna viável a integração e a implantação contínua.
3. **Além disso, cada serviço pode ser escalado de forma independente de outros serviços através da duplicação ou particionamento.** Além disso, cada serviço pode ser implantado em

um hardware mais adequado para as exigências de seus recursos. Situação bem diferente da utilização de uma arquitetura monolítica, que possui componentes com diferentes necessidades, e.g. uso de CPU versus uso de disco, são implantados em conjunto.

4. A arquitetura de microsserviços também melhora o isolamento de falhas. Por exemplo, um vazamento de memória em um serviço afeta apenas aquele serviço. Outros serviços irão continuar a receber requisições. Em contrapartida, em uma arquitetura monolítica, um componente com comportamento inadequado irá comprometer todo o sistema.

5. A arquitetura de microsserviços elimina compromissos de longo prazo com a pilha tecnológica. Em princípio, ao desenvolver um novo serviço, os desenvolvedores são livres para escolher os *frameworks* e linguagem adequados para aquele serviço. Embora em muitas organizações as escolhas possam ter restrições, o ponto principal é a independência sobre as decisões tomadas. E é mais simples migrar microsserviços que estejam em uma tecnologia legada ou descontinuada por um fornecedor.

Existem algumas desvantagens e pontos de atenção.

1. **Os times de desenvolvedores devem lidar com a complexidade adicional de desenvolvimento e testes de sistemas distribuídos.** Os desenvolvedores devem implementar um mecanismo de comunicação entre processos. A implementação de casos de uso que abrangem vários serviços sem o uso de transações distribuídas é difícil. IDEs e outras ferramentas de desenvolvimento tem o foco na construção de aplicações monolíticas e não oferecem suporte direto para o desenvolvimento de aplicações distribuídas. Escrever testes automatizados para vários serviços também é um desafio.
2. **A arquitetura de microsserviços introduz uma complexidade operacional significativa.** Existem mais elementos (múltiplas instâncias de diferentes serviços) que devem ser gerenciados em produção. Para alcançar o sucesso necessita-se de um alto nível de automação, seja por código desenvolvido pela própria equipe ou tecnologias PAAS como o Microsoft Azure Service Fabric, Netflix Asgard e o Pivotal Cloud Foundry
3. **Além disso, a implantação de funcionalidades que abrangem vários serviços requer uma coordenação cuidadosa entre as várias equipes de desenvolvimento.** É preciso criar um plano ordenado de implantações de serviços com base nas dependências entre serviços. Entretanto, implantar atualizações em uma arquitetura monolítica é mais simples, pois é executado de forma atômica, onde apenas um artefato precisa ser implantado.

Para uso efetivo desse estilo, é necessário o uso intenso de automação por plataformas PAAS em produção, que identificam e extraem os metadados dos microsserviços lá implantados como o contrato de operações REST ou dependências para outros serviços. Essas plataformas também monitoram SLAs de disponibilidade ou performance e também agregam funções básicas como segurança ou auditoria.

4.5 Arquiteturas Baseadas em Eventos

A Arquitetura de Eventos (Event Driven Architecture - EDA) é uma abordagem para a construção de sistemas que se baseia no fluxo de eventos. Nessa arquitetura, os componentes do sistema se comunicam entre si através de eventos, em vez de chamadas diretas. Um evento é um pacote de informações que descreve algo que aconteceu no sistema, como uma atualização de dados ou uma solicitação de serviço.

A vantagem da EDA é que ela é altamente escalável e tolerante a falhas, pois os componentes do sistema são independentes entre si e podem ser adicionados ou removidos sem afetar o funcionamento geral do sistema. Além disso, a EDA permite que os componentes do sistema evoluam independentemente uns dos outros, o que facilita a manutenção e o desenvolvimento contínuo.

Exemplos de tecnologias e plataformas que utilizam a arquitetura de eventos incluem o Apache Kafka, o RabbitMQ e o AWS Kinesis.

4.5.1 Brokers e Mediators

A arquitetura de Broker é baseada no uso de um componente central que funciona como um ponto de contato para todas as comunicações entre os componentes. Esse componente, conhecido como Broker, é responsável por encaminhar os eventos (mensagens) entre os diferentes componentes. Ele geralmente mantém uma lista de assinantes interessados em determinado evento e encaminha a mensagem para esses assinantes. Um exemplo de Broker é o Apache Kafka.

Já a arquitetura de Mediator é baseada no uso de um componente central que atua como intermediário entre os componentes, mas com a função adicional de realizar algum tipo de processamento nas mensagens antes de encaminhá-las. Esse componente, conhecido como Mediator, é responsável por decidir qual componente deve receber a mensagem e também pode fazer modificações na mensagem antes de encaminhá-la. Um exemplo de Mediator é o Apache Camel.

Vamos olhar para um exemplo na arquitetura de eventos na AWS é baseada em serviços como o Amazon SNS (Simple Notification Service) e o Amazon SQS (Simple Queue Service). Esses serviços permitem que diferentes componentes de uma aplicação se comuniquem através de eventos assíncronos.

Um exemplo de uso dessa arquitetura pode ser uma aplicação de e-commerce. Quando um cliente realiza uma compra, um evento de "compra realizada" é gerado e enviado ao Amazon SNS. Esse evento é então transmitido a diferentes componentes da aplicação, como o sistema de pagamento, o sistema de estoque e o sistema de envio. Cada componente responde ao evento de forma assíncrona, processando sua parte da compra. Por exemplo, o sistema de pagamento pode verificar se o pagamento foi aprovado, o sistema de estoque pode atualizar o estoque e o sistema de envio pode enviar uma notificação de confirmação de envio para o cliente.

Isso permite uma maior escalabilidade e flexibilidade na aplicação, já que cada componente pode ser escalado e gerenciado independentemente. Além disso, a arquitetura de eventos também facilita a manutenção e atualização da aplicação, pois novos componentes podem ser adicionados ou removidos sem afetar o funcionamento dos outros componentes.

4.6 Arquitetura Baseada em Espaços

A maioria dos aplicativos de negócios baseados na Web segue o mesmo fluxo de solicitação geral: uma solicitação de um navegador atinge o servidor da Web, depois um servidor de aplicativos e, finalmente, o servidor de banco de dados. Embora esse padrão funcione muito bem para um pequeno conjunto de usuários, os gargalos começam a aparecer à medida que a carga do usuário aumenta, primeiro na camada do servidor da Web, depois na camada do servidor de aplicativos e, finalmente, na camada do servidor de banco de dados. A resposta usual para gargalos com base em um aumento na carga do usuário é expandir os servidores web. Isso é relativamente fácil e barato e, às vezes, funciona para resolver os problemas de gargalo. No entanto, na maioria dos casos de alta carga de usuários, a expansão da camada do servidor da Web apenas move o gargalo para o servidor de aplicativos.

O dimensionamento de servidores de aplicativos pode ser mais complexo e caro do que os servidores da Web e, geralmente, apenas transfere o gargalo para o servidor de banco de dados, que é ainda mais difícil e caro de dimensionar. Mesmo se você puder dimensionar o banco de dados, o que acabará sendo uma topologia em forma de triângulo, com a parte mais larga do triângulo sendo os servidores da Web (mais fáceis de dimensionar) e a menor parte sendo o banco de dados (mais difícil de dimensionar).

Em qualquer aplicativo de alto volume com uma grande carga de usuários simultâneos, o banco de dados geralmente será o fator limitante final em quantas transações você pode processar simultaneamente. Embora várias tecnologias de cache e produtos de dimensionamento de banco de dados ajudem a resolver esses problemas, o fato é que dimensionar um aplicativo normal para cargas extremas é uma proposta muito difícil.

A arquitetura baseada em espaço adquire seu nome a partir do conceito de *tuple space*, técnica que utiliza múltiplos processadores paralelos comunicando-se através de memória compartilhada. A alta escalabilidade, elasticidade e desempenho são alcançados removendo o banco de dados central como uma restrição síncrona no sistema e, em vez disso, aproveitando grades de dados replicadas em memória. Os dados da aplicação são mantidos na memória e replicados entre todas as unidades de processamento ativas. Quando uma unidade de processamento atualiza os dados, ela envia assincronamente os dados para o banco de dados, geralmente via mensagens com filas persistentes. As unidades de processamento iniciam e encerram dinamicamente à medida que a carga de usuários aumenta e diminui, endereçando a escalabilidade variável. Como não há um banco de dados central envolvido no processamento transacional padrão da aplicação, os gargalos do banco de dados são removidos, fornecendo uma escalabilidade quase infinita dentro da aplicação.

Existem vários componentes de arquitetura que compõem uma arquitetura baseada em espaço: uma unidade de processamento contendo o código da aplicação, middleware virtualizado utilizado para gerenciar e coordenar as unidades de processamento, bombas de dados para enviar dados atualizados assincronamente para o banco de dados, escritores de dados que realizam as atualizações a partir das bombas de dados, e leitores de dados que leem os dados do banco de dados e os entregam às unidades de processamento no início.

5 Estilo Arquitetural de APIs

5.1 O que é uma API

API significa "Application Programming Interface" (Interface de Programação de Aplicativos, em português) e é um conjunto de padrões, protocolos e ferramentas que permitem que diferentes aplicativos se comuniquem e interajam uns com os outros. Em outras palavras, uma API é um conjunto de regras que define como um software deve interagir com outro software.

As APIs permitem que desenvolvedores de software criem aplicativos que se integrem a outras plataformas e serviços, sem precisar conhecer os detalhes internos dessas plataformas e serviços. Isso torna mais fácil para os desenvolvedores criarem novos aplicativos e serviços, sem precisar reinventar a roda e reescrever código complexo para interagir com outras plataformas.

As APIs são usadas em uma ampla variedade de aplicativos e serviços, desde aplicativos de desktop e mobile até plataformas em nuvem e sistemas de gerenciamento de dados.

Por exemplo, o Google Maps oferece uma API que permite que os desenvolvedores integrem mapas interativos em seus próprios aplicativos, e o Twitter oferece uma API que permite que os desenvolvedores criem aplicativos.

5.2 Plataforma de APIs como Aceleradores

Uma plataforma de API para suportar o ciclo de vida completo de desenvolvimento de APIs pode incluir várias ferramentas e recursos para auxiliar em cada fase do processo.

Aqui estão alguns exemplos:

- Desenho: uma plataforma de API pode incluir ferramentas para criar e modelar APIs, como o Swagger ou o RAML. Essas ferramentas permitem que os desenvolvedores visualizem e projetem as APIs de maneira eficiente, definindo a estrutura, os parâmetros, as respostas e as operações que serão suportadas.
- Implementação: depois de projetar a API, os desenvolvedores precisam criar o código real para implementá-la. Uma plataforma de API pode oferecer suporte para diversas linguagens de programação e frameworks, como o Node.js ou o Ruby on Rails, além de fornecer orientações e exemplos de código para ajudar os desenvolvedores a começar.
- Testes: uma vez que a API foi implementada, é importante testá-la para garantir que ela funcione corretamente e atenda aos requisitos esperados. Uma plataforma de API pode incluir ferramentas para testar APIs de forma automatizada, como o Postman ou o SoapUI. Essas ferramentas permitem que os desenvolvedores criem scripts de teste e executem testes automatizados para verificar se a API está funcionando corretamente.
- Publicação: depois de testar a API, é hora de publicá-la para que outros desenvolvedores possam começar a usá-la. Uma plataforma de API pode incluir recursos para publicar APIs em diferentes ambientes, como produção ou sandbox, além de fornecer documentação e exemplos de uso para ajudar outros desenvolvedores a entender como usar a API.
- Hospedagem: uma plataforma de API também pode incluir recursos para hospedar as APIs publicadas, incluindo escalabilidade e alta disponibilidade. A hospedagem da API deve ser segura e confiável para garantir que ela esteja sempre disponível para os usuários. Produtos como API Gateways suportam essa etapa do processo.

- Engajamento de desenvolvedores: por fim, uma plataforma de API deve incluir recursos para engajar desenvolvedores que desejam usar a API. Isso pode incluir fóruns de discussão, suporte técnico, exemplos de código e documentação clara e precisa. Além disso, a plataforma pode incluir recursos de monitoramento para permitir que os desenvolvedores acompanhem o desempenho e a utilização da API.

5.3 O API Gateway

Um API Gateway é um componente de software que funciona como uma porta de entrada para um conjunto de APIs, permitindo que os usuários accessem e gerenciem várias APIs de maneira centralizada. Em outras palavras, o API Gateway é um ponto único de entrada para todas as APIs de uma plataforma, oferecendo recursos como autenticação, gerenciamento de tráfego, segurança, monitoramento e análise.

Quando um cliente faz uma solicitação para uma API, ele primeiro interage com o API Gateway, que encaminha a solicitação para a API relevante. O API Gateway gerencia o tráfego entre as diferentes APIs, garantindo que as solicitações sejam processadas corretamente e distribuídas de forma equilibrada entre os recursos disponíveis.

O API Gateway também oferece recursos de segurança, como autenticação e autorização, para proteger as APIs contra ataques maliciosos e garantir que apenas usuários autorizados possam acessar os recursos protegidos. Além disso, o API Gateway pode ser configurado para oferecer recursos de análise e monitoramento, permitindo que os desenvolvedores acompanhem o desempenho e a utilização das APIs, identifiquem problemas e otimizem a performance.

O API Gateway é um componente importante de plataformas de API, pois ajuda a simplificar e gerenciar o acesso às APIs. Ele fornece um ponto único de entrada para todas as APIs da plataforma, o que torna mais fácil para os usuários encontrarem e usar as APIs de que precisam. Além disso, o API Gateway ajuda a proteger as APIs contra ameaças de segurança e oferece recursos avançados de análise e monitoramento que ajudam a melhorar o desempenho e a qualidade das APIs.

Algumas plataformas de API, como o Amazon API Gateway, são construídas especificamente para fornecer um gateway de API. Outras plataformas de API podem ter um gateway de API integrado como parte de suas soluções mais amplas de gerenciamento de API. Em ambos os casos, o API Gateway é um componente fundamental para garantir que as APIs sejam gerenciadas de forma eficiente, segura e escalável.

5.4 APIs Baseadas em REST

Uma API baseada em REST é uma API que segue as restrições e princípios da arquitetura REST (Representational State Transfer). Essa arquitetura é baseada no uso do protocolo HTTP e em uma abordagem de recursos e representações, onde cada recurso é identificado por um URI (Uniform Resource Identifier) e pode ser acessado através de verbos HTTP como GET, POST, PUT e DELETE.

As APIs baseadas em REST são caracterizadas por serem simples, escaláveis e interoperáveis. Elas oferecem uma maneira padronizada de acessar recursos através de uma interface uniforme e sem estado, permitindo que diferentes sistemas e plataformas possam se comunicar de forma eficiente.

As APIs baseadas em REST têm as seguintes características:

1. Recursos: as APIs baseadas em REST são construídas em torno de recursos, que são objetos de dados que podem ser acessados através de um URI.

2. Verbos HTTP: as APIs baseadas em REST usam verbos HTTP para definir as operações que podem ser realizadas em um recurso. Os verbos HTTP mais comuns são GET, POST, PUT e DELETE.
3. Representações: cada recurso pode ter várias representações, que são formatos diferentes para a mesma informação. Por exemplo, um recurso pode ser representado em JSON, XML ou outro formato.
4. Interface uniforme: a interface de uma API baseada em REST é uniforme e consistente, tornando-a fácil de entender e usar.
5. Sem estado: as APIs baseadas em REST não armazenam informações sobre as transações anteriores, tornando-as escaláveis e fáceis de manter.

As APIs baseadas em REST são amplamente utilizadas na indústria, pois são flexíveis, escaláveis e fáceis de integrar com outras plataformas e sistemas. Elas são usadas para fornecer acesso a dados e recursos em aplicativos móveis, sites, IoT e outras aplicações que requerem comunicação com outras plataformas.

5.4.1 Modelo de Maturidade de Richardson

O modelo de maturidade de Richardson, também conhecido como Maturity Model for REST (MMR), é um modelo que descreve uma série de níveis de maturidade para o desenvolvimento de APIs baseadas em REST, proposto por Leonard Richardson em 2008. O modelo é composto por quatro níveis, cada um representando uma etapa de evolução da implementação de uma API RESTful.

Os quatro níveis do modelo de maturidade de Richardson são:

1. Nível 0 - Padrões de URL: Neste nível, as APIs não são construídas seguindo nenhum padrão ou convenção de URL. Cada recurso é acessado através de uma URL diferente e não há consistência na forma como as URLs são estruturadas.
2. Nível 1 - Recursos: Neste nível, as APIs são construídas em torno de recursos e cada recurso é identificado por uma URL única. No entanto, a implementação não segue completamente os princípios REST, pois os verbos HTTP são usados de forma inadequada, como por exemplo, utilizando apenas o verbo GET para todas as operações.
3. Nível 2 - Verbos HTTP: Neste nível, as APIs usam os verbos HTTP corretamente para operações específicas em um recurso, como GET para recuperar um recurso, POST para criar um novo recurso, PUT para atualizar um recurso existente e DELETE para excluir um recurso.
4. Nível 3 - HATEOAS: Neste nível, as APIs adicionam o recurso HATEOAS (Hypermedia As The Engine Of Application State), que permite que o cliente navegue pela API usando links em cada resposta, em vez de depender de documentação explícita. Isso aumenta a flexibilidade e a escalabilidade da API e simplifica a integração do cliente com a API.

O modelo de maturidade de Richardson é útil para avaliar a qualidade de uma API RESTful e identificar áreas para melhoria. Ele incentiva o uso de práticas recomendadas e o cumprimento dos princípios REST, tornando as APIs mais eficientes, escaláveis e fáceis de usar e manter.

5.4.2 Exemplo de API REST no Nível 2

Este é um exemplo simples de uma API RESTful para cadastro de clientes:

1. Para listar todos os clientes cadastrados, faça uma requisição HTTP GET para o endpoint /clientes. A resposta será uma lista de objetos JSON, onde cada objeto representa um cliente.
2. Para cadastrar um novo cliente, faça uma requisição HTTP POST para o endpoint /clientes, com um objeto JSON contendo as informações do novo cliente no corpo da requisição. Se o cadastro for bem sucedido, a API retornará o código de status HTTP 201 (Created). Se os dados enviados forem inválidos, a API retornará o código de status HTTP 400 (Bad Request).

Este exemplo segue os princípios do nível 2 do modelo de maturidade de Richardson, utilizando os verbos HTTP de forma adequada para cada operação.

5.4.3 Exemplo de API REST no Nível 3

Aqui está um exemplo de uma API RESTful para cadastro de clientes que segue o nível 3 do modelo de maturidade de Richardson, utilizando HATEOAS:

Para obter a lista de todos os clientes cadastrados, faça uma requisição HTTP GET para o endpoint /clientes.

A resposta será um objeto JSON que contém a lista de clientes, juntamente com links para as operações permitidas em cada cliente, como atualização ou exclusão. Por exemplo:

```
{
  "clientes": [
    {
      "id": 1,
      "nome": "João Silva",
      "email": "joao.silva@teste.com",
      "telefone": "(11) 98765-4321",
      "endereco": {
        "rua": "Rua dos Testes",
        "cidade": "São Paulo",
        "estado": "SP",
        "cep": "01234-567"
      },
      "links": [
        {
          "rel": "self",
          "href": "/clientes/1"
        },
        {
          "rel": "update",
          "href": "/clientes/1",
          "method": "PUT"
        },
        {
          "rel": "delete",
          "href": "/clientes/1",
          "method": "DELETE"
        }
      ]
    },
  ],
}
```

Observe que cada objeto cliente contém um array de links, que representam as operações permitidas no cliente. Os links contêm informações sobre o método HTTP correspondente, como PUT ou DELETE, e a URL para acessar a operação correspondente. Além disso, o objeto de resposta contém links adicionais para criar um novo cliente.

Para criar um novo cliente, basta fazer uma requisição HTTP POST para o endpoint /clientes, com um objeto JSON que contém as informações do novo cliente no corpo da requisição. A resposta será um objeto JSON que representa o novo cliente, juntamente com links para as operações permitidas no cliente, como atualização ou exclusão.

Para atualizar ou excluir um cliente existente, basta seguir o link correspondente no objeto JSON de resposta da lista de clientes e fazer uma requisição HTTP PUT ou DELETE para o endpoint indicado, com o objeto JSON atualizado, no caso da atualização. A resposta será um objeto JSON que representa o cliente atualizado ou uma mensagem de sucesso, no caso da exclusão.

Este exemplo segue os princípios do nível 3 do modelo de maturidade de Richardson, utilizando HATEOAS.

5.5 APIs Baseadas em RPC

API RPC (Remote Procedure Call) é um estilo de arquitetura de software que permite que um aplicativo chame funções ou procedimentos em um servidor remoto por meio de uma API. Em outras palavras, é um modelo de comunicação em que o cliente chama uma função ou método em um servidor e aguarda a resposta.

Neste modelo, as operações são descritas em termos de funções que o servidor oferece. O cliente envia uma solicitação para o servidor indicando qual função deseja executar, juntamente com os argumentos necessários para a execução dessa função. O servidor processa a solicitação e retorna uma resposta que contém o resultado da operação.

Em uma API RPC, as chamadas podem ser síncronas ou assíncronas. Em chamadas síncronas, o cliente aguarda a resposta do servidor antes de continuar a executar sua rotina. Em chamadas assíncronas, o cliente pode enviar uma solicitação e continuar executando sua rotina, sem aguardar a resposta do servidor imediatamente.

Embora as APIs RPC possam ser úteis em certos cenários, elas geralmente são consideradas menos flexíveis do que outros estilos de arquitetura de software, como as APIs RESTful, que seguem o princípio de usar métodos HTTP para representar as operações. Isso ocorre porque as APIs RPC tendem a ser mais rígidas em termos de como as funções são chamadas e os tipos de dados que podem ser transmitidos, enquanto as APIs RESTful permitem uma maior flexibilidade na definição das operações.

5.6 gRPC

Uma API gRPC é baseada no protocolo RPC (Remote Procedure Call) e é projetada para oferecer alto desempenho e eficiência de comunicação entre clientes e servidores. Ela permite que os desenvolvedores criem sistemas distribuídos em grande escala que podem ser executados em ambientes heterogêneos.

Em uma API gRPC, os serviços são descritos usando um arquivo de definição de serviço chamado "Protocol Buffers". Esse arquivo define os métodos de serviço disponíveis, seus parâmetros de entrada e saída e seus tipos de dados. O arquivo é compilado em uma linguagem específica, gerando classes que podem ser usadas para invocar os métodos do serviço.

O protocolo gRPC usa o protocolo HTTP/2 como base para a comunicação entre o cliente e o servidor.

As solicitações e respostas são codificadas em formato binário e compactadas usando o algoritmo de compressão gRPC para reduzir o tamanho da mensagem e melhorar a eficiência da comunicação.

Uma das principais vantagens do gRPC é a sua capacidade de trabalhar com vários idiomas de programação. Ele suporta várias linguagens, incluindo Java, Python, Go, C++, Ruby, entre outras. Isso significa que os desenvolvedores podem criar clientes e servidores em diferentes linguagens de programação, e eles podem se comunicar facilmente entre si usando a API gRPC.

Além disso, o gRPC também suporta streaming bidirecional, o que significa que o cliente e o servidor podem enviar e receber fluxos de mensagens ao mesmo tempo. Isso torna possível a criação de aplicativos em tempo real, como chats, jogos multiplayer e transmissões de vídeo ao vivo.

Em resumo, uma API gRPC é uma poderosa ferramenta de comunicação cliente-servidor baseada no protocolo RPC, que oferece alto desempenho, eficiência e suporte a várias linguagens de programação. Ele usa o protocolo HTTP/2 para comunicação e oferece suporte a streaming bidirecional para a criação de aplicativos em tempo real.

Existem várias alternativas ao gRPC que também usam o protocolo RPC para a comunicação entre o cliente e o servidor. Algumas das alternativas mais populares incluem:

- Apache Thrift: O Thrift é um framework de desenvolvimento de software que suporta vários idiomas de programação e é usado para desenvolver sistemas escaláveis e interoperáveis. Ele usa um formato binário compacto para comunicação entre o cliente e o servidor, o que o torna mais eficiente em termos de desempenho em relação às APIs RESTful.
- Apache Avro: O Avro é um sistema de serialização de dados que suporta RPC. Ele é projetado para ser compacto, eficiente e interoperável entre diferentes linguagens de programação. O Avro usa um formato binário compacto para comunicação entre o cliente e o servidor, o que o torna mais eficiente em termos de desempenho em relação às APIs RESTful.
- JSON-RPC: O JSON-RPC é um protocolo simples que usa JSON (JavaScript Object Notation) para serializar dados e RPC para comunicação entre o cliente e o servidor. Ele é projetado para ser fácil de usar e suporta uma ampla variedade de linguagens de programação.
- XML-RPC: O XML-RPC é um protocolo simples que usa XML (Extensible Markup Language) para serializar dados e RPC para comunicação entre o cliente e o servidor. Ele é projetado para ser fácil de usar e suporta uma ampla variedade de linguagens de programação.

5.7 REST versus RPC

	REST	RPC
Abordagem	Baseado em recursos	Baseado em procedimentos
Protocolos	HTTP, HTTPS	HTTP, HTTPS, TCP, UDP
Formatos de dados	JSON, XML, outros	Qualquer formato
Performance	Menos eficiente do que o RPC, devido à sobrecarga adicional de transferência de dados de metadados	Mais eficiente do que o REST, devido à ausência de sobrecarga adicional de transferência de dados de metadados
Facilidade de implementação	Fácil	Mais difícil do que o REST
Compatibilidade	Compatível com a maioria das plataformas	Pode não ser compatível com todas as plataformas
Suporte a operações complexas	Não muito adequado	Pode lidar com operações complexas
Escalabilidade	Fácil de escalar	Pode ser menos escalável do que o REST
Cache	Pode ser facilmente cacheado	Cache menos eficiente do que o REST

5.8 GraphQL

GraphQL é uma linguagem de consulta de dados que foi desenvolvida pelo Facebook em 2012 e lançada como um projeto de código aberto em 2015. É uma alternativa ao REST, que permite que os desenvolvedores definam a estrutura dos dados necessários para um aplicativo específico e, em seguida, solicitem esses dados por meio de uma única consulta.

Em vez de ter que fazer várias chamadas para obter diferentes partes de dados de uma API, o GraphQL permite que os desenvolvedores especifiquem exatamente quais dados precisam em uma única consulta, reduzindo assim o tráfego de rede e melhorando o desempenho do aplicativo.

Aqui estão alguns conceitos chave do GraphQL:

- Schema: O Schema é a estrutura de tipos de dados que define o conjunto de dados disponíveis para consulta na API. É semelhante a um contrato entre o servidor e o cliente.
- Queries: As queries são a maneira como os clientes solicitam dados do servidor. As queries especificam quais dados são necessários e como esses dados devem ser organizados.
- Mutations: As mutations são usadas para modificar os dados na API. Elas são usadas para criar, atualizar ou excluir dados.
- Resolvers: Os resolvers são funções que resolvem as queries e mutations, retornando os dados solicitados pelo cliente.

- Fragments: Os fragments são usados para reutilizar partes de queries.

Aqui está um exemplo simples de uma query GraphQL:

```
query {
  author(id: 1) {
    name
    books {
      title
    }
  }
}
```

O retorno dessa consulta seria o seguinte.

```
{
  "data": {
    "author": {
      "name": "F. Scott Fitzgerald",
      "books": [
        {
          "title": "O Grande Gatsby"
        },
        {
          "title": "Suave é a Noite"
        }
      ]
    }
  }
}
```

Observe como os dados solicitados na consulta GraphQL são retornados exatamente como especificados, sem informações adicionais desnecessárias. Isso ajuda a reduzir o tamanho da resposta e a melhorar a eficiência da comunicação entre o servidor e o cliente.

Aqui está um exemplo de mutation GraphQL:

```

mutation {
  createBook(
    title: "O Grande Gatsby",
    authorId: 1
  ) {
    id
    title
  }
}

```

Esta mutation cria um novo livro com o título "O Grande Gatsby" e o associa ao autor com o ID 1.

5.9 WebSocket

O WebSocket é uma tecnologia de comunicação bidirecional em tempo real entre um navegador e um servidor da web. Ele permite que os desenvolvedores criem aplicativos da web que enviam e recebem dados em tempo real sem a necessidade de consultas de atualização constantes, como ocorre com o HTTP.

O WebSocket foi padronizado em 2011 e é suportado pelos principais navegadores da web. Ao contrário do HTTP 1.1, que usa uma conexão de curta duração para enviar uma solicitação e receber uma resposta do servidor, o WebSocket usa uma conexão persistente que permite que o servidor envie dados para o cliente a qualquer momento, sem a necessidade de uma solicitação explícita. Essa conexão é mantida aberta enquanto o aplicativo da web estiver em execução e pode ser usada para enviar uma quantidade ilimitada de dados.

O WebSocket usa um protocolo binário de baixa sobrecarga para minimizar o tamanho dos dados enviados pela rede. Ele também usa criptografia SSL/TLS para garantir que a comunicação entre o navegador e o servidor da web seja segura.

Aqui está um exemplo de como um aplicativo da web pode usar o WebSocket para enviar e receber mensagens em tempo real:

1. O cliente estabelece uma conexão WebSocket com o servidor da web.
2. O servidor da web envia uma mensagem de boas-vindas para o cliente através da conexão WebSocket.
3. O cliente envia uma mensagem de resposta ao servidor da web através da conexão WebSocket.
4. O servidor da web recebe a mensagem do cliente e envia uma nova mensagem para o cliente através da conexão WebSocket.

O WebSocket é usado em uma ampla variedade de aplicativos da web em tempo real, como jogos online, bate-papos em grupo, ferramentas de colaboração em tempo real e muito mais.

Um exemplo mínimo é mostrado no código a seguir.

```
const socket = new WebSocket('wss://example.com');

socket.addEventListener('open', function (event) {
  socket.send('Olá, servidor!');
});

socket.addEventListener('message', function (event) {
  console.log('Mensagem do servidor: ', event.data);
});

socket.addEventListener('close', function (event) {
  console.log('Conexão fechada.');
});
```

Este código cria uma nova conexão WebSocket com o servidor em "wss://example.com". Quando a conexão é estabelecida com sucesso, o cliente envia a mensagem "Hello, server!" para o servidor. Quando o servidor envia uma mensagem de volta para o cliente, o evento "message" é acionado e o cliente exibe a mensagem recebida no console. Quando a conexão é fechada, o evento "close" é acionado e o cliente exibe uma mensagem de que a conexão foi fechada.

5.9.1 Evoluções do HTTP como alternativas ao Web Socket

O HTTP/1.1 foi a versão mais usada do protocolo HTTP até recentemente, e apesar de ter sido projetado para suportar a maioria das necessidades da Web, ele apresentou algumas limitações que levaram a evoluções do protocolo, incluindo:

1. HTTP/2: Esta versão foi projetada para reduzir o tempo de carregamento de páginas, reduzir a latência e melhorar a segurança. Ele introduziu o multiplexação de fluxo, onde várias solicitações podem ser enviadas simultaneamente em uma única conexão, eliminando a necessidade de múltiplas conexões para solicitações paralelas. O HTTP/2 também suporta compressão de cabeçalho e TLS (Transport Layer Security) como padrão.
2. HTTP/3: Esta versão é a mais recente evolução do protocolo HTTP e está atualmente em fase de padronização. O HTTP/3 foi projetado para melhorar ainda mais a latência, segurança e confiabilidade em comparação com as versões anteriores. Ele é baseado no protocolo QUIC (Quick UDP Internet Connection), que foi projetado para superar as limitações do TCP (Transmission Control Protocol), que é usado pelo HTTP/1.1 e HTTP/2. O HTTP/3 oferece benefícios como conexões mais rápidas, menor latência e menor consumo de energia.

Embora o WebSocket seja uma tecnologia popular para comunicação bidirecional em tempo real, o HTTP/2 e HTTP/3 também podem oferecer suporte a recursos semelhantes, como servidor push e streaming. Além disso, o HTTP/3 também tem a capacidade de suportar comunicação bidirecional em tempo real, o que pode torná-lo uma alternativa ao WebSocket no futuro.

No entanto, é importante notar que cada tecnologia tem suas próprias vantagens e desvantagens e pode ser mais adequada para diferentes casos de uso. Por exemplo, o WebSocket ainda pode ser a melhor opção para aplicativos que exigem comunicação bidirecional em tempo real e suporte para conexões persistentes.

5.10 Comparativo de Protocolos de Transporte

Protocolo	Tipo	Camada de Transporte	Modelo de Transferência	Formato de Mensagem	Uso Comum
HTTP 1.1	Web	TCP	Requisição/Resposta	Texto (HTML, XML, JSON)	Navegador web, APIs REST
HTTP 2.0	Web	TCP, TLS	Requisição/Resposta	Binário (H2)	Navegador web, APIs REST
HTTP 3	Web	UDP, TLS	Requisição/Resposta	Binário (H3)	Navegador web, APIs REST
gRPC	RPC	TCP, TLS	Bidirecional	Binário (Protobuf)	Microsserviços, Comunicação inter-servidor
Apache Thrift	RPC	TCP, TLS	Bidirecional	Binário (Protocolo próprio)	Microsserviços, Comunicação inter-servidor
MQTT	Mensagem	TCP, TLS	Publish/Subscribe	Binário (MQTT)	Internet das Coisas (IoT)
ZigBee	Rede	IEEE 802.15.4	Publish/Subscribe	Binário (ZCL)	Domótica, Internet das Coisas (IoT)
SOAP	Web	TCP, HTTP	Requisição/Resposta	XML	Integração de sistemas

5.11 Comparativo de Protocolos de Dados

Protocolo	Tamanho (em relação ao XML)	Facilidade de uso	Popularidade	Vantagens	Desvantagens
XML	100%	Média	Média	Humanamente legível, amplo suporte em linguagens e ferramentas	Verboso, alto overhead em relação ao tamanho dos dados
JSON	25%	Fácil	Muito alta	Leve, fácil de ler e escrever, suporte em linguagens e ferramentas	Limitado a tipos de dados simples, sem compressão
ProtoBuffer	10-20%	Difícil	Alta	Compacto, rápida serialização e desserialização, suporta atualizações de esquema	Não legível para humanos, requer definição de esquema, necessidade de compilador específico

5.12 Comparativo de Táticas de APIs

Tecnologia	Vantagens	Desvantagens
REST	Fácil de entender e usar; Ampla adoção e suporte; Confiável e escalável.	Limitado em termos de manipulação de dados; Requer várias solicitações HTTP para operações complexas.
RPC	Desempenho rápido; Alta eficiência na comunicação; Estrutura simples.	Menor flexibilidade em comparação com outras tecnologias; Maior complexidade de gerenciamento em ambientes distribuídos.
GraphQL	Seleção flexível de dados; Redução de tráfego de rede; Melhorias no desempenho da aplicação.	Maior curva de aprendizado; Requer mais recursos do servidor; Pode ser vulnerável a ataques de negação de serviço (DoS).
WebSocket	Comunicação bidirecional em tempo real; Latência reduzida; Protocolo de baixa sobrecarga; Suporte a conexões persistentes; Facilidade de implementação de notificações em tempo real.	Pode ser difícil de depurar e testar; Requer servidores dedicados.

6 Arquiteturas de Eventos

6.1 Conceitos

Uma arquitetura orientada por eventos é um estilo de arquitetura de software em que os componentes do sistema se comunicam através de eventos assíncronos. Nesse estilo arquitetural, as ações do sistema são representadas por eventos que ocorrem em momentos específicos e são registrados por um mecanismo de eventos.

Os eventos podem ser gerados internamente pelo próprio sistema ou por sistemas externos, e cada evento contém informações sobre o que aconteceu, quando ocorreu e quaisquer outros dados relevantes. Esses eventos são processados por um ou mais componentes do sistema que estão interessados neles e que podem tomar ações ou tomar decisões com base nesses eventos.

Uma arquitetura orientada por eventos é altamente escalável e flexível, pois os componentes do sistema podem ser adicionados ou removidos conforme necessário, sem afetar o funcionamento do sistema como um todo. Além disso, a arquitetura é capaz de lidar com grandes volumes de eventos assíncronos, o que é ideal para sistemas que precisam lidar com grande quantidade de dados em tempo real, como sistemas de comércio eletrônico, sistemas de IoT (Internet das Coisas), sistemas de monitoramento de segurança, entre outros.

Uma arquitetura orientada por eventos é composta por diversos componentes, como geradores de eventos, processadores de eventos, armazenamento de eventos e mecanismos de entrega de eventos. Alguns exemplos de tecnologias usadas em arquiteturas orientadas por eventos incluem Apache Kafka, Amazon Kinesis, RabbitMQ e Azure Event Grid.

6.2 Conceitos Básicos

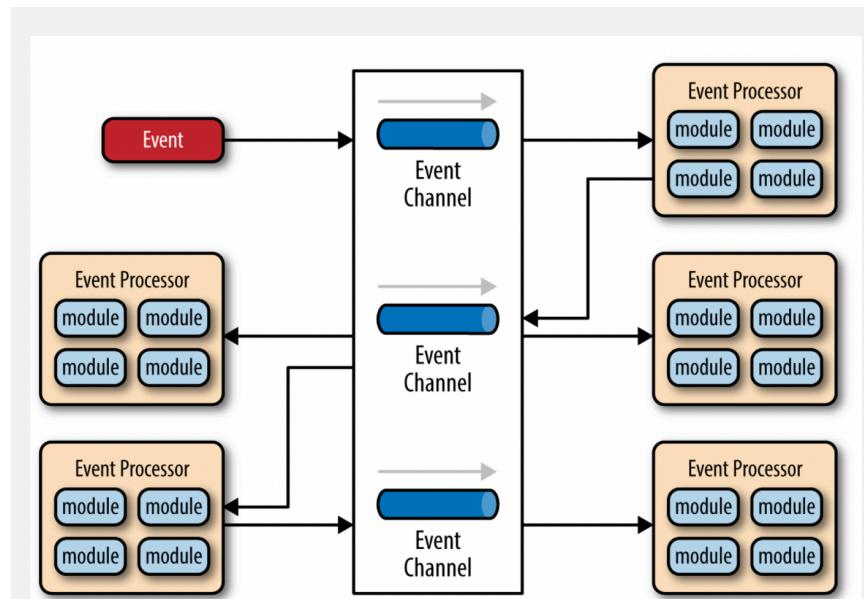
1. Mensagem: Uma mensagem é uma unidade básica de comunicação entre componentes de um sistema de eventos. Geralmente, uma mensagem é um objeto que contém informações específicas que precisam ser transmitidas para outro componente do sistema. As mensagens podem ser enviadas de forma síncrona ou assíncrona, dependendo das necessidades do sistema.
2. Evento: Um evento é uma ocorrência significativa que acontece dentro do sistema e que pode ser interessante para outros componentes. Os eventos podem ser gerados em resposta a ações de usuários, operações do sistema ou até mesmo de outros eventos. Cada evento geralmente contém informações relevantes sobre a ocorrência, como data e hora, identificador único e detalhes sobre a ação realizada.
3. Fila: Uma fila é uma estrutura de dados que armazena mensagens em uma ordem específica. As mensagens são adicionadas à fila em uma extremidade e removidas na outra extremidade, em uma ordem conhecida como "primeiro a entrar, primeiro a sair" (FIFO). As filas garantem que as mensagens sejam entregues em ordem e que cada mensagem seja processada exatamente uma vez. As filas são geralmente usadas quando um componente precisa processar mensagens em um ritmo mais lento do que o ritmo em que as mensagens são recebidas.
4. Tópico: Um tópico é uma estrutura de dados que permite que os eventos sejam transmitidos a um ou mais destinatários, conhecidos como assinantes. Os tópicos podem ser organizados em hierarquias, permitindo que os assinantes se inscrevam em tópicos específicos de acordo com suas necessidades. Os tópicos não garantem que cada mensagem seja processada exatamente uma vez, mas permitem que os assinantes recebam todos os eventos que são relevantes para suas necessidades. Os tópicos são

geralmente usados quando um componente precisa processar mensagens em um ritmo mais rápido do que o ritmo em que as mensagens são recebidas.

6.3 Padrões para Arquitetura de Eventos – Broker

O padrão Broker é um modelo em que um componente central, chamado de Broker, é responsável por receber eventos de vários produtores e torná-los disponíveis para um ou mais consumidores interessados.

O Broker é responsável por facilitar e garantir a entrega de eventos para todos os consumidores registrados e pode incluir recursos para filtrar eventos e roteá-los para o destinatário correto.



Fonte: Fundamentals of Software Architecture – Neal Ford

Vamos compreender esse padrão em um caso simples de comércio eletrônico, ao ser aplicado ao processo de compra e envio de um pedido:

1. Pedido Recebido: Quando um cliente faz um pedido no site de comércio eletrônico, o sistema de comércio eletrônico publica um evento de "pedido recebido" em um tópico específico. Esse evento pode incluir informações como o número do pedido, os itens comprados e as informações do cliente.
2. Pagamento: Em seguida, o sistema de processamento de pagamentos pode se inscrever nesse tópico para receber informações sobre os pagamentos. Quando o pagamento do pedido é processado com sucesso, o sistema de processamento de pagamentos publica um evento de "pagamento processado" em outro tópico específico.
3. Verificação de Estoque: O sistema de gerenciamento de estoque pode se inscrever nesse tópico para receber notificações sobre quais itens precisam ser enviados e para onde. Quando o pedido é feito, o sistema de gerenciamento de estoque verifica se há estoque suficiente dos itens solicitados. Se houver estoque disponível, o sistema de gerenciamento de estoque publica um evento de "estoque verificado" em outro tópico específico.
4. Processamento do Pedido: O sistema de gerenciamento de pedidos pode se inscrever nesse tópico para receber informações sobre os pedidos. Quando os eventos de "pedido recebido", "pagamento

processado" e "estoque verificado" são recebidos, o sistema de gerenciamento de pedidos pode processar o pedido, gerando um evento de "pedido processado" em outro tópico específico.

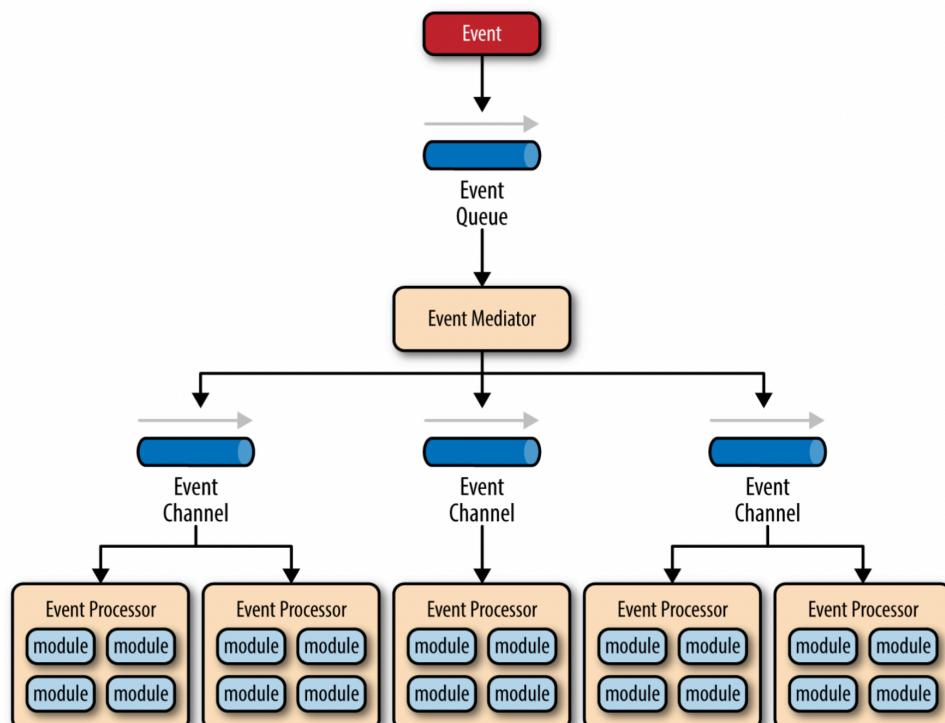
5. Preparação para envio: Quando o pedido é processado, o sistema de gerenciamento de envio pode se inscrever no tópico "pedido processado" para receber informações sobre o pedido. O sistema de gerenciamento de envio pode, em seguida, preparar o pedido para envio, gerando um evento de "pedido preparado para envio" em outro tópico específico.
6. Despacho: Finalmente, quando o pedido é preparado para envio, o sistema de gerenciamento de envio pode gerar um evento de "pedido despachado" em outro tópico específico, indicando que o pedido foi despachado para o cliente.

O Broker é responsável por garantir que todos os sistemas que precisam receber informações sobre os pedidos as recebam. Além disso, o Broker pode incluir recursos para filtrar eventos e roteá-los para os destinatários corretos. Esse padrão torna o processo de compra e envio mais eficiente, permitindo que os diferentes sistemas trabalhem de forma independente e reduzindo a necessidade de acoplamento entre eles.

6.4 Padrões para Arquitetura de Eventos – Mediator

O padrão Mediator é um modelo em que um componente central, chamado de Mediator, atua como intermediário entre os componentes do sistema.

Em vez de se comunicarem diretamente, os componentes enviam mensagens para o Mediator, que então encaminha as mensagens para os destinatários corretos. O Mediator pode incluir lógica para filtrar mensagens, gerenciar o fluxo de mensagens e coordenar ações entre os componentes.



Fonte: Fundamentals of Software Architecture – Neal Ford

Com base no exemplo de um site de comércio eletrônico, podemos descrever como o padrão Mediator pode ser aplicado ao processo de compra e envio de um pedido:

Pedido Recebido: Quando um cliente faz um pedido no site de comércio eletrônico, o sistema de comércio eletrônico notifica o Mediator sobre o evento "pedido recebido", fornecendo informações como o número do pedido, os itens comprados e as informações do cliente. **Pagamento:** O Mediator recebe a notificação do pagamento processado e encaminha as informações para o sistema de gerenciamento de pedidos.

Verificação de Estoque: O sistema de gerenciamento de estoque envia uma notificação ao Mediator informando o estoque disponível para os itens do pedido. O Mediator recebe a notificação e encaminha as informações para o sistema de gerenciamento de pedidos. **Processamento do Pedido:** O sistema de gerenciamento de pedidos recebe as informações do Mediator sobre o pedido, pagamento e estoque, e processa o pedido, gerando um evento de "pedido processado". **Preparação para envio:** O sistema de gerenciamento de envio é notificado pelo Mediator sobre o evento de "pedido processado", e prepara o pedido para envio, gerando um evento de "pedido preparado para envio". **Despacho:** O sistema de gerenciamento de envio envia uma notificação para o Mediator indicando que o pedido foi despachado, gerando um evento de "pedido despachado".

Diferentemente do padrão Broker, no Mediator não há um componente central que gerencia as comunicações. Em vez disso, o Mediator atua como um intermediário entre os diferentes sistemas, permitindo que eles se comuniquem de forma independente. O Mediator é responsável por encaminhar as notificações de um sistema para outro e garantir que todas as partes interessadas recebam as informações necessárias. Isso permite que os sistemas se comuniquem de forma mais flexível e com menos acoplamento.

6.5 Diferenças entre o Broker e o Mediator

O Mediator e o Broker são ambos padrões de arquitetura de eventos que podem ser usados para simplificar a comunicação entre componentes em um sistema distribuído. No entanto, existem algumas diferenças fundamentais entre eles.

O Mediator é um componente centralizado que atua como intermediário entre os outros componentes. Em vez de cada componente se comunicar diretamente com os outros, eles se comunicam apenas com o Mediator, que gerencia as interações e coordena as atividades necessárias para processar uma mensagem ou evento. Cada componente só precisa se preocupar em se comunicar com o Mediator, que coordena todo o processo. Essa abordagem pode tornar o sistema mais modular e flexível, pois cada componente pode ser alterado ou substituído sem afetar os outros componentes.

Já o Broker é um componente que atua como intermediário entre os componentes que produzem e consomem mensagens ou eventos. Ele recebe as mensagens dos produtores e as envia para os consumidores, que se inscrevem em tópicos relevantes para receber as mensagens que precisam processar. Cada componente precisa se preocupar em se inscrever nos tópicos relevantes para receber as mensagens que precisa processar. O Broker pode oferecer escalabilidade para lidar com grandes volumes de mensagens e pode ser configurado para garantir alta disponibilidade e resiliência.

Em resumo, a principal diferença entre o Mediator e o Broker é que o Mediator atua como um componente centralizado que gerencia as interações entre os componentes, enquanto o Broker é responsável por encaminhar as mensagens entre os componentes sem coordená-las. O Mediator pode simplificar a comunicação entre os componentes e tornar o sistema mais modular e flexível, enquanto o Broker pode oferecer escalabilidade e resiliência para lidar com grandes volumes de mensagens em sistemas distribuídos.

6.6 Processamento de Mensagens e Fluxos

A arquitetura orientada por eventos tem como base a troca de mensagens entre diferentes sistemas e serviços, permitindo que eles se comuniquem de forma assíncrona e desacoplada. Nesse contexto, existem duas tecnologias principais: processamento de mensagens e fluxos (streams) de eventos.

O processamento de mensagens é baseado em filas, onde as mensagens são armazenadas em uma ordem específica e processadas por um único consumidor por vez. Quando uma mensagem é lida da fila, ela é removida e processada pelo consumidor. Isso garante que cada mensagem seja processada uma vez e em uma ordem específica. Exemplos de tecnologias de processamento de mensagens incluem Apache ActiveMQ, RabbitMQ e Amazon SQS.

Essas tecnologias são otimizadas para a entrega de mensagens confiável e garantida, incluindo a capacidade de lidar com mensagens perdidas ou falhas de sistema. Além disso, essas tecnologias geralmente têm recursos avançados, como controle de transações, filtragem de mensagens, roteamento de mensagens, entrega baseada em prioridade e monitoramento de mensagens.

Por outro lado, o fluxo de eventos é uma abordagem baseada em streams, em que os eventos são transmitidos continuamente para todos os consumidores que estão inscritos em um fluxo (*stream*). Os consumidores podem processar esses eventos em tempo real, em paralelo, sem esperar que uma mensagem específica seja processada antes de processar outra. Isso permite que os sistemas respondam rapidamente às mudanças nos eventos, bem como escalabilidade e tolerância a falhas. Exemplos de tecnologias de fluxo de eventos incluem Apache Kafka, Amazon Kinesis e Azure Event Hubs.

Em resumo, enquanto o processamento de mensagens se concentra em garantir que cada mensagem seja processada em uma ordem específica e por apenas um consumidor, o fluxo de eventos permite que os eventos sejam processados em paralelo e em tempo real, sem se preocupar com a ordem em que eles foram recebidos. A escolha entre essas tecnologias depende dos requisitos específicos de cada sistema e do tipo de aplicação que está sendo construída.

6.7 Comparativos

6.7.1 Comparativo de Padrões

	Processamento de Mensagens	Fluxos
Brokers	Gerenciam filas de mensagens e atuam como intermediários entre os produtores e consumidores de mensagens. Possuem recursos avançados para roteamento e filtragem de mensagens.	Podem processar grandes volumes de dados em tempo real, distribuir dados para vários consumidores, e fornecer recursos de transformação e processamento de fluxo.
Mediators	Permitem a comunicação entre componentes independentes de um sistema, reduzindo o acoplamento e aumentando a flexibilidade e escalabilidade do sistema. Os mediadores gerenciam a comunicação e coordenação entre componentes.	Permitem a comunicação e transformação de dados entre diferentes fontes de dados em tempo real. Podem ser usados para integrar sistemas e processos heterogêneos.

6.7.2 Comparativo de Ferramentas

	Processamento de Mensagens	Fluxos
Brokers	RabbitMQ, ActiveMQ, SQS, ZeroMQ	Apache Kafka, AWS Kinesis, Google Cloud Pub/Sub
Mediators	Azure Service Bus, MuleSoft Anypoint, Apache Camel	Apache Flink, Apache Storm, Apache Beam

6.7.3 Processamento de Fila de Mensagens com Broker AWS SQS

Fragmento de código em Node.js com o uso do AWS SQS

```
const AWS = require('aws-sdk');
const sqs = new AWS.SQS({ region: 'us-east-1' }); // substitua pela região

const params = {
  QueueUrl: 'URL_DA_FILA', // substitua pela URL da sua fila
  MaxNumberOfMessages: 1 // define o número máximo de mensagens para receber
};

sqs.receiveMessage(params, function(err, data) {
  if (err) {
    console.log('Erro ao receber mensagem da fila', err);
  } else if (data.Messages) {
    console.log('Mensagem recebida:', data.Messages[0].Body);

    // deleta a mensagem da fila depois de processá-la
    const deleteParams = {
      QueueUrl: 'URL_DA_FILA',
      ReceiptHandle: data.Messages[0].ReceiptHandle
    };
    sqs.deleteMessage(deleteParams, function(err, data) {
      if (err) {
        console.log('Erro ao deletar mensagem da fila', err);
      } else {
        console.log('Mensagem deletada da fila', data);
      }
    });
  } else {
    console.log('Nenhuma mensagem encontrada na fila');
  }
});
```

6.7.4 Processamento de Streams com Broker Apache Kafka

Fragmento de código em Node.js com o uso do Kafka

```
const kafka = require('kafka-node');
const ConsumerGroupStream = kafka.ConsumerGroupStream;

const consumerOptions = {
  kafkaHost: 'localhost:9092',
  groupId: 'my-group',
  autoCommit: true,
  autoCommitIntervalMs: 5000,
  sessionTimeout: 30000,
  protocol: ['roundrobin'],
  fetchMaxBytes: 1024 * 1024,
};

const topics = ['my-topic'];

const consumerGroupStream = new ConsumerGroupStream(consumerOptions, topics)
```

```
consumerGroupStream
  .pipe(
    // Transforma a mensagem em JSON
    new Transform({
      objectMode: true,
      decodeStrings: true,
      transform: function (message, encoding, callback) {
        try {
          const json = JSON.parse(message.value);
          callback(null, json);
        } catch (error) {
          callback(error);
        }
      },
    }),
    // Filtra apenas as mensagens com um determinado campo
    filter((message) => message.field === 'value'),
    // Processa a mensagem
    map((message) => {
      console.log(`Processando mensagem: ${JSON.stringify(message)}`);
      return message;
    })
  )
  .on('error', function (error) {
    console.error(error);
  });
}
```

6.7.5 Processamento de Fila de Mensagens com Mediator Apache Camel

Neste fragmento de código em XML e Java, estamos configurando uma rota que lê mensagens de uma fila do Amazon SQS chamada "myqueue". A configuração usa um cliente do Amazon SQS para se conectar à fila.

```
<bean id="amazonSQSClient" class="com.amazonaws.services.sqs.AmazonSQSClient">
    <constructor-arg>
        <bean class="com.amazonaws.auth.BasicAWSCredentials">
            <constructor-arg value="your_access_key"/>
            <constructor-arg value="your_secret_key"/>
        </bean>
    </constructor-arg>
    <property name="region" value="us-west-2"/>
</bean>

<route>
    <from uri="aws-sqs://myqueue?amazonSQSClient=#amazonSQSClient"/>
    <to uri="log:received-message"/>
</route>
```

```
public static void main(String[] args) throws Exception {
    CamelContext context = new DefaultCamelContext();
    context.addRoutes(new MyRouteBuilder());
    context.start();
    Thread.sleep(5000);
    context.stop();
}
```

6.7.6 Processamento de Fila de Mensagens com Mediator Apache Camel

7 Topologia de Servidores Web

Servidores de software Web são componentes arquiteturais que suportam o protocolo HTTP. Ao mesmo tempo, essas aplicações devem ser também distribuídas sobre máquinas para operar em ambientes de produção.

É esperado que o arquiteto de software Web saiba analisar, comparar e escolher os tipos de servidores de software e servidores físicos mais apropriados ao contexto do seu problema. Isso afeta requisitos arquiteturais tais como:

- Performance;
- Escalabilidade;

- Tolerância a Falhas;
- Manutenibilidade;
- Implantabilidade;
- Segurança da informação.

Esse capítulo é organizado ao longo dessas escolhas. A primeira parte esclarece as opções, vantagens e desvantagens de cada tipo de servidor Web. A segunda parte apresenta possíveis configurações de distribuição física chamada de topologias.

7.1 Servidores Web Baseados em Processos (1º Geração)

O primeiro servidor Web foi desenvolvido a partir do utilitário *inetd* do Linux, que é um programa utilitário que responde requisições de um cliente em um certo porto e faz o despacho da requisição para um programa. No começo dos anos 90, um programa específico foi desenvolvido para lidar com requisições http. Esse programa se chama httpD (Http Daemon), desenvolvido pela NCSA (*National Center for Supercomputing Applications* da Universidade de Illinois).

Como aplicações Web tendem a gerar um tráfego alto composto por múltiplas requisições de tempo de vida curto, esse tipo de mecanismo não é usado para fins profissionais. Com o tempo, peças específicas de software foram desenvolvidas para tratar com escalabilidade e segurança requisições HTTP. Ao mesmo tempo, é importante destacar o servidor Web mais popular do planeta, o Apache HTTP Server, foi desenvolvido a partir do utilitário NCSA httpD.

7.2 Servidores Web Baseados em Threads (2º Geração)

A história desses servidores se confunde com a história do Apache HTTP Server. Em linhas gerais, o Apache HTTP Server¹² surgiu a partir de utilitários simples para responder a requisições HTTP e foi evoluído para incluir características hoje fundamentais em aplicações Web tais como suportar:

- Múltiplos clientes simultâneos através de *multi-threading*;
- APIs de extensibilidade para a construção e distribuição de novos módulos;
- Transporte seguro (SSL) e mecanismos de autenticação e autorização de páginas.

Esses servidores se tornaram dominantes na Web ainda nos anos 90 e exemplos de servidores dessa categoria incluem o Microsoft IIS¹³ e NGINX¹⁴. Enquanto o primeiro servidor é dominante para aplicações desenvolvidas em ASP e ASP.NET, o segundo foi desenvolvido como uma opção mais performática do Apache HTTP Server.

7.3 Servidores Web de Aplicações (3º Geração)

A tecnologia Java EE era no final dos 90 o esforço mais sofisticado de organização de plataformas servidoras, inspirados por modelos hoje legados como o CORBA. Servidores Java EE trazem, por especificação, uma enorme coleção de serviços embutidos (*out of the box*), tais como linguagens de páginas dinâmicas, gerência de memória, operação clusterizada, controle transacional distribuído, modelos de componentes distribuídos e conectores com plataformas legados, entre outros). Como consequência dessa enormidade de serviços, empresas como IBM, BEA, SUN, TIBCO, Fujitsu, Oracle e JBOSS, entre outras, começaram a desenvolver servidores Web com esteroides. Essas peças foram apelidadas de “servidores de aplicação” e são servidores Web que foram desenvolvidos para rodar aplicações servidoras. No mundo Microsoft, a combinação do IIS, .NET Framework, MSMQ e Windows pode ser vista, com alguma liberdade arquitetural, com um servidor de aplicação Microsoft que hospeda e roda aplicações .NET

¹² <https://httpd.apache.org>

¹³ <http://www.iis.net>

¹⁴ <https://www.nginx.com>

A história do Java EE e .NET se confundem com esses tipos de servidores Web. Em 2016, alguns servidores de aplicações populares incluem:

- IBM Websphere Application Server¹⁵
- Oracle Internet Applicaton Server¹⁶ (antigo BEA Weblogic)
- Redhat Wildfly¹⁷ (JBoss Application Server)
- Apache TomEE¹⁸
- Microsoft IIS + MSQM + .NET Framework¹⁹
- Microsoft BizTalk

7.4 Servidores Web de Micro contêineres (4º Geração)

A partir de 2010, um movimento de minimalismo começou a tomar conta da comunidade de desenvolvimento Web. Os motivos estão ligados a problemas de escalabilidade e o peso de várias soluções dos servidores de terceira geração. Alguns desses servidores exigem pelo menos 1GB de memória para funcionamento, ocupam dezenas de gigabytes de espaço em disco, requerem processadores de última geração para performar e alocam centenas de *threads* quando são instanciados. Um exemplo de servidor Web de quarta geração é Express²⁰ do Node.JS. Ele é um servidor minimalista que opera junto da própria aplicação JS que está sendo executada. Ele ocupa um espaço mínimo de memória (entre 10 a 20 megabytes), poucos megabytes de espaço disco e usa recursos mínimos de CPU.

As próprias comunidades Java EE e .NET começam a desenvolver soluções minimalistas para servidores Web. No mundo Java EE, a Spring (hoje Pivotal) entregou soluções minimalistas como o Spring Boot²¹. O Eclipse Jetty²² e o KumuluzEE²³ são outras soluções nesse sentido. No mundo .NET, a versão mais recente do ASP.NET (ASP.NET 5, rebatizado de ASP.NET Core) e o projeto .NET Core são exemplos nesse sentido. Aplicações ASP.NET Core podem rodar sem a necessidade de servidores como o IIS. Essa nova geração de servidores Web elimina o modelo tradicional e empacotamento e distribuição de aplicações (*assemble & deploy*). Ao invés, a própria aplicação Java, C ou JS servidor é executada como um servidor Web em um modelo chamado de aplicação auto hospedada (*self-host application*). Essa nova geração é útil para o desenvolvimento no estilo arquitetural de microsserviços.

¹⁵ <http://www-03.ibm.com/software/products/pt/appserv-was>

¹⁶ <http://www.oracle.com/technetwork/middleware/ias/overview/index.html>

¹⁷ <http://wildfly.org>

¹⁸ <http://tomee.apache.org/apache-tomee.html>

¹⁹ <https://www.microsoft.com/net/default.aspx>

²⁰ <http://expressjs.com/pt-br>

²¹ <http://projects.spring.io/spring-boot/>

²² <http://www.eclipse.org/jetty/>

²³ <https://ee.kumuluz.com>

7.5 Comparativo de Servidores Web

	Baseados em Processos 1º Geração	Baseados em Threads 2º Geração	Baseados em Servidores de Aplicação 3º Geração	Embarcados em Aplicações Web 4º Geração
Objetivo	Provar o protocolo HTTP no início dos anos 90.	Habilitar requisições HTTP em larga escala para aplicações comerciais.	Habilitar aplicações Web e serviços de valor agregado Web.	Habilitar aplicações Web minimalistas e serviços extensíveis sob demanda
Exemplos	NCSA httpD	Apache HTTP Server Microsoft IIS Apache Tomcat	IBM WAS Oracle IAS JBoss Wildfly Apache TomEE Microsoft BizTalk PHP Zend PHP Symfony	Node Express.JS HTTP.sys Kestrel Eclipse Jetty Pivotal Spring Boot PHP Silex
Carga inicial na máquina	Leve	Média	Pesada	Leve
Administração	Simples	Média	Complexa	Simples
Escalabilidade	Ruim	Muito boa	Muito boa	Excelente
Estilos Arquiteturais Web	Web 1.0	Web 1.0 Web 2.0 SPA	Web 1.0 Web 2.0 SPA API Web	Web 2.0 SPA API Web Microsserviços
Padrões Arquiteturais Comuns	N/A	MVC	MVC	MVC/MVVM/MVP API Gateway
Linguagens Comuns	C, C++, Python	PHP, Python, Ruby, ASP, ASP.NET, JSP, JSF	JSF/Java ASP.NET/C#	JavaScript/Node C# Java com Spring Boot Python PHP

Tabela 4: Tipos de Servidores de Software Web

7.6 Distribuição Física de Aplicações Web

Uma vez escolhido o servidor de software Web e montada a aplicação, é importante deliberar como a aplicação será distribuída. Essa distribuição é denominada de topologia e é realizada através de um esforço conjunto entre o time de arquitetos Web e os arquitetos de infraestrutura.

A distribuição física deve ser escolhida conforme os condutores arquiteturais Web definidos para o projeto. Atenção! Não existe uma topologia ótima, mas a melhor topologia dentro do contexto do produto sendo construído.

7.7 Distribuição Mínima

Nessa distribuição temos uma máquina física hospedando o servidor Web, aplicativo Web e banco de dados da aplicação.

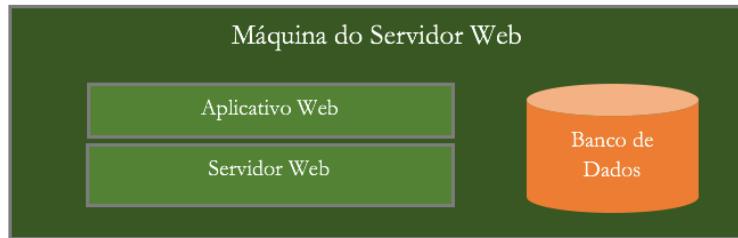


Figura 10: Distribuição Mínima de uma Aplicação Web

Esse cenário é recomendado para ambientes de desenvolvimento pois permite o trabalho *off-line* e também para aplicações de produção de pequeno porte que usem servidores Web com micro com bancos leves ou bancos Non-SQL, ou mesmo aplicações *CPU-bound* (centrada em CPU).

Aplicações MEAN (MongoDB, Express, AngularJS e Node.js) para sistemas de informação simples são muitas vezes distribuídas nessa topologia.

7.8 Distribuição com Servidor Web Dedicado

Essa distribuição pode ser vista na maior parte das aplicações LAMP, Java e .NET do mercado. Aqui existe uma máquina dedicada para o serviço Web. O servidor de banco de dados é apartado em uma outra máquina, separando assim a carga de trabalho HTTP da carga de trabalho de transações SQL.

Essa configuração é útil quando o sistema de informação apresenta um tráfego de dados, volume transacional e apresenta natureza *IO-bound* (centrada em entrada e saída de dados).

Ela também apresenta uma vantagem de segurança adicional. A máquina do banco de dados pode ser administrada de forma independente por um DBA. Uma outra possibilidade é podermos colocar *firewalls* entre o servidor e o servidor de banco de dados. Em ambientes de empresas de porte médio e grande, essas necessidades de administração e segurança se fazem presente na maior parte dos casos.

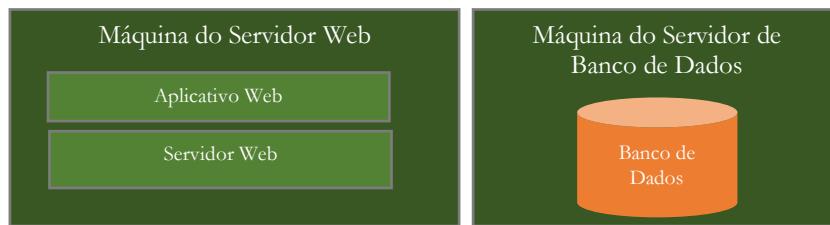


Figura 11: Distribuição com Servidor Web Dedicado

7.9 Distribuição com Servidor de Aplicação Dedicado

Algumas aplicações Web fazem uso extensivo de processamento de CPU, com exigências altas de tráfego e controle transacional. Aplicações bancárias e de Telecom são exemplos nesse sentido. Nesse tipo de cenário temos a presença de uma terceira máquina física para cuidar apenas do processamento das transações de negócios.



Figura 12: Distribuição com Servidor de Aplicação Dedicado

Esse modelo de distribuição permite vantagens adicionais tais como:

- Controle da carga da aplicação para os processos de dados e processos transacionais de negócio;
- Segurança aumenta com a introdução de firewalls entre o servidor Web e o servidor de aplicação e entre o servidor de aplicação e o servidor de banco de dados. Somado ao firewall que fica entre a Internet pública e o servidor Web, podemos ter três firewalls até o banco de dados.
- Poder de administração independente de cada máquina (DBAs ou Arquitetos de Aplicação).

Ao mesmo tempo, essa topologia traz alguns desafios tais como:

- Necessidade de criação de uma API remota de serviços entre a camada do servidor de aplicação e camada Web. Isso pode ser realizado com o uso de tecnologias como JAX-WS, JAX-RS, EJB, WCF ou ASP.NET WebAPI;
- Maior complexidade para implantabilidade e administração;
- Custos maiores devido ao conjunto de máquinas necessárias para implementar essa topologia.
- Distribuição com Cluster de Servidores

Aplicações de missão crítica requerem alta disponibilidade (99% ou mais) bem como tolerância a falhas em seus componentes. A arquitetura baseada em clusters de máquinas físicas podem ser um auxílio nesse sentido, conforme Figura 13.



Figura 13: Distribuição com Clusterização

Essa arquitetura promove dois aspectos centrais:

- Tolerância a falhas. Se uma máquina do servidor Web, servidor de aplicação ou servidor de banco de dados falhar, ainda existe uma outra máquina capaz de responder.
- Escalabilidade. As requisições podem ser distribuídas ao longo das máquinas do cluster através de um balanceamento de carga e com isso suportar picos de escalabilidade.

O uso de clusters não vem sem custos e diversos aspectos devem ser considerados:

Eventuais modificações na aplicação para que ela consiga operar em modo de clusterização. Isso pode envolver o uso de componentes preparados para esse fim com EJB ou mesmo a aquisição de produtos especializados como SQL Server Enterprise ou Oracle RAC.

Inclusão de平衡adores de carga, seja através de roteadores ou máquinas dedicadas e software específicos para esse fim (ex. Linux LVS ou Microsoft Network Load Balancing²⁴).



Figura 14: Balanceador de carga para clusterização

7.10 Distribuição com Máquina para Conteúdo Estático

Em algumas configurações de clusters, é possível que certas máquinas fiquem reservadas para lidar apenas com o conteúdo estático (HTML estático, imagens ou vídeos). Nesse tipo de configuração, uma máquina específica é alocada para esse fim.



Figura 15: Balanceador de carga com máquina para conteúdo estático

7.11 Distribuição com Arquiteturas Elásticas

A virtualização e a computação nas nuvens permitem que máquinas sejam alocadas conforme parâmetros definidos pelo arquiteto Web. Um exemplo nesse sentido é a tecnologia Amazon EC2 com o uso dos produtos AutoScale e Elastic Load Balancer (ver Figura 16). Outros fornecedores de nuvem, como a Microsoft Azure, Digital Ocean ou IBM já fornecem também esse tipo de serviço nativamente na sua estrutura de serviço.

²⁴

LVS

NLB - <https://msdn.microsoft.com/en-us/library/bb742455.aspx>

<http://www.linuxvirtualserver.org/whatis.html>

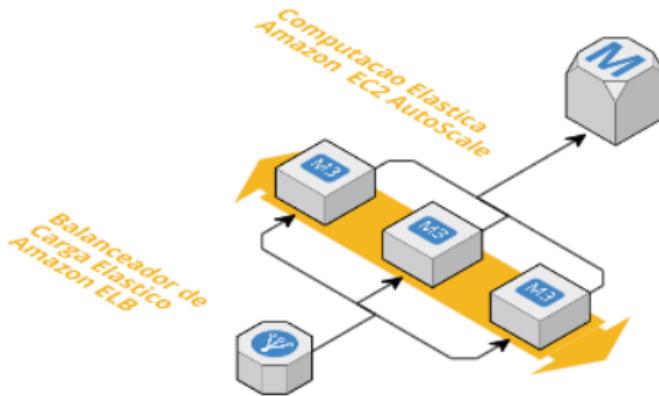


Figura 16: Cluster elástico em ambiente de nuvem. O ELB (balanceador elástico) analisa as cargas de trabalho e solicita que o componente AutoScale aloque ou desaloque máquinas de forma dinâmica.

Ela permite que um conjunto mínimo de máquinas sejam alocadas para servir uma aplicação. Se o consumo de recursos aumentar além de um certo ponto, o ambiente aloca mais máquinas e cuida do平衡amento de carga da aplicação. Da mesma forma, se o consumo reduzir abaixo de um certo valor, o ambiente desaloca máquinas para evitar um custo de aluguel desnecessário.

7.12 Comparativo de Topologias Web

A Tabela 5 apresenta um resumo dos tipos de topologia conforme requisitos arquiteturais comuns em plataformas Web.

	Simplicidade	Facilidade para Implantar	Segurança Física	Escalabilidade Horizontal	Tolerância a Falhas
Distribuição Mínima	😊	😊	😐	😢	😢
Servidor Web dedicado	😊	😐	😐	😐	😢
Servidor de Aplicação dedicado	😢	😢	😊	😊	😢
Clusters de Servidores	😢	😢	😊	😊	😊
Clusters com Servidor para Conteúdo Estático	😢	😢	😊	😊	😊
Cluster Elástico em Nuvens	😢	😐	😊	😊	😊

Tabela 5: Tipos de Topologia Versus Condutores Arquiteturais Web.

Essa tabela indica que não existem topologias ótimas. Como outras decisões arquiteturais, a “melhor topologia” é sempre uma decisão de projeto, tomada pelos arquitetos através de cuidadosa análise ambiental dos condutores do seu projeto.

8 Tecnologias Web

8.1 Tecnologias Web de Visão

8.1.1 HTML 5

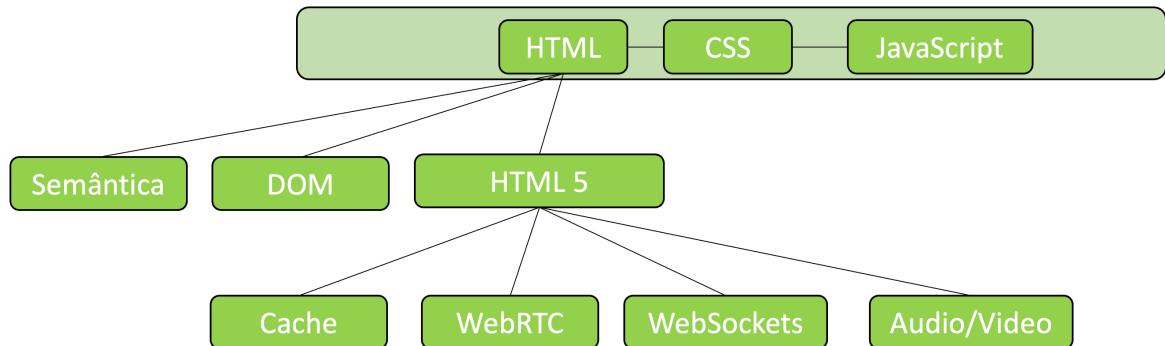


Figura 17: Tecnologias HTML 5

<https://developer.mozilla.org/pt-BR/docs/Web/HTML/HTML5>

8.1.2 Tecnologias de CSS

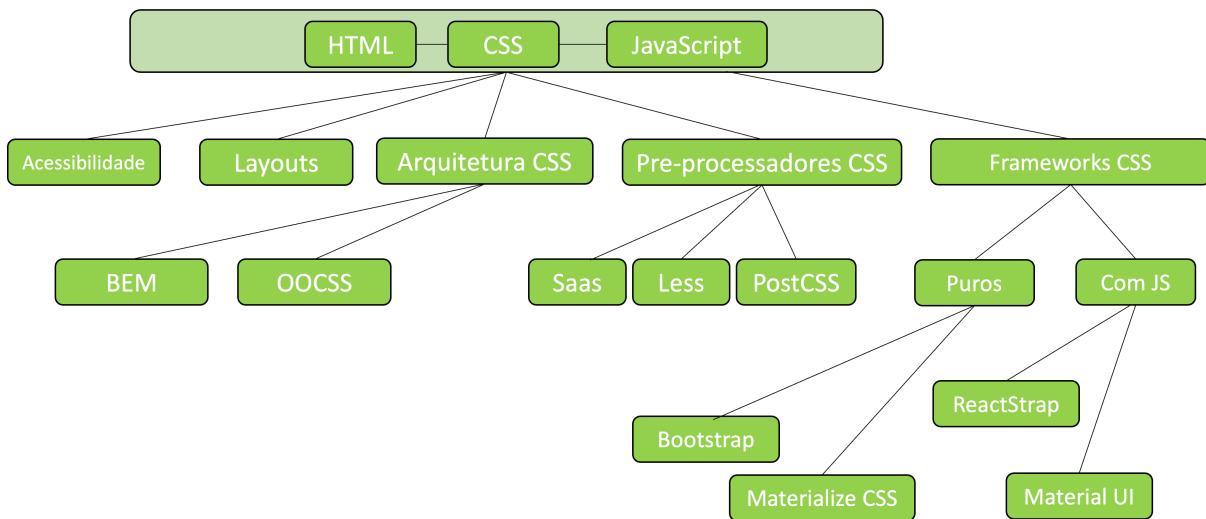


Figura 18: Tecnologias CSS

<https://developer.mozilla.org/pt-BR/docs/Web/CSS>

8.1.3 Tecnologias de Visão MVC (Server Side)

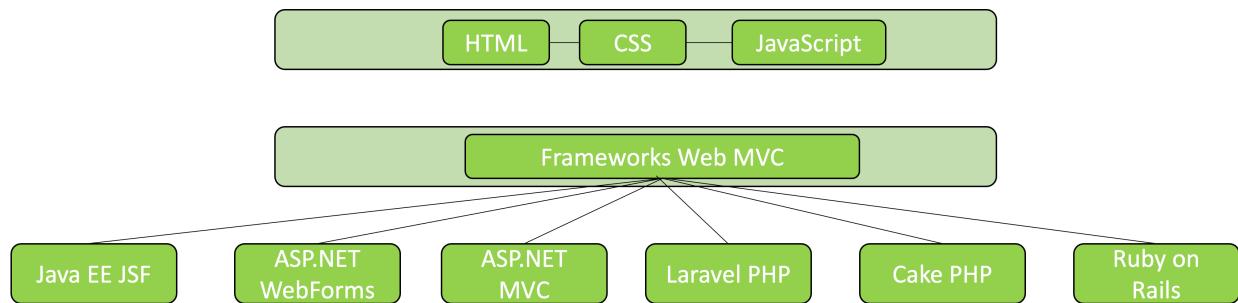


Figura 19: Tecnologias Web MVC

8.1.4 Tecnologias de Visão MVVM Baseadas em Javascript

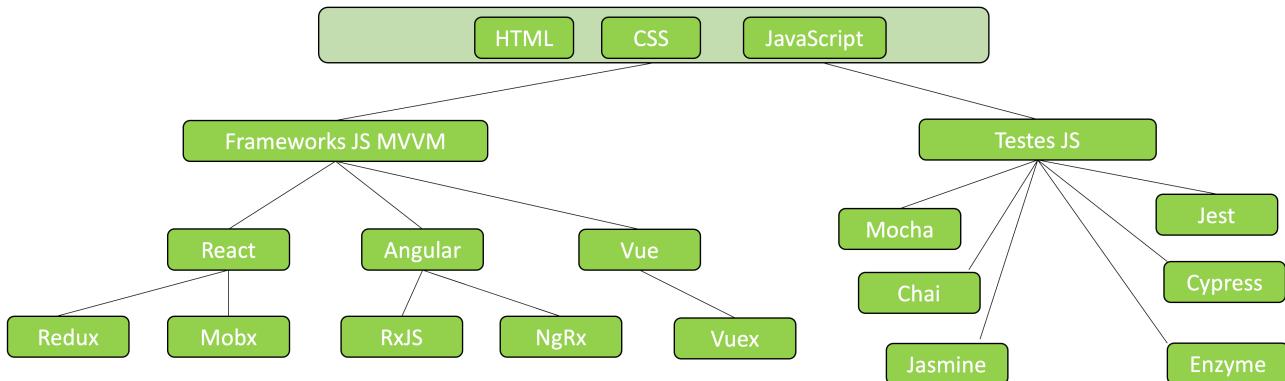


Figura 20: Tecnologias Web MVVM

9 Servidores Web Baseados em Java EE

O Java EE (*Java Enterprise Edition*) é uma especificação de tecnologias coordenadas cuja proposta é reduzir o custo e a complexidade do desenvolvimento, implantação e gerenciamento de aplicações corporativas complexas. Ele é construído sobre a plataforma Java SE (JVM) e oferece um conjunto de APIs para o desenvolvimento e execução de aplicativos portáteis, robustos, escaláveis, confiáveis e seguros no lado do servidor.

O Java EE é mantido pela comunidade JCP²⁵, que inclui dezenas de empresas tais como a Oracle, IBM ou Redhat. A partir das especificações do JCP, diferentes fornecedores lançam servidores Web que implementam essa especificação. Exemplos incluem:

- Apache Tomcat
- Redhat JBOSS Wildfly (JBoss Application Server)
- IBM WebSphere Application Server
- Oracle Internet Application Server

O Java EE apresenta como principais vantagens arquiteturais:

- Operação em sistemas operacionais distintos como Windows, Linux ou Z/OS (mainframes);
- Facilidades para componentização de aplicações;
- Serviços de suporte como controle transacional distribuído, segurança, cache e escalabilidade;
- Facilidades para escalabilidade horizontal e vertical e tolerância a falhas;
- Grande conjunto de conectores para a comunicação com sistemas legados;
- Diversidade de fornecedores.

Representamos os principais componentes arquiteturais Java EE Web na figura a seguir.

²⁵ <http://jcp.org>

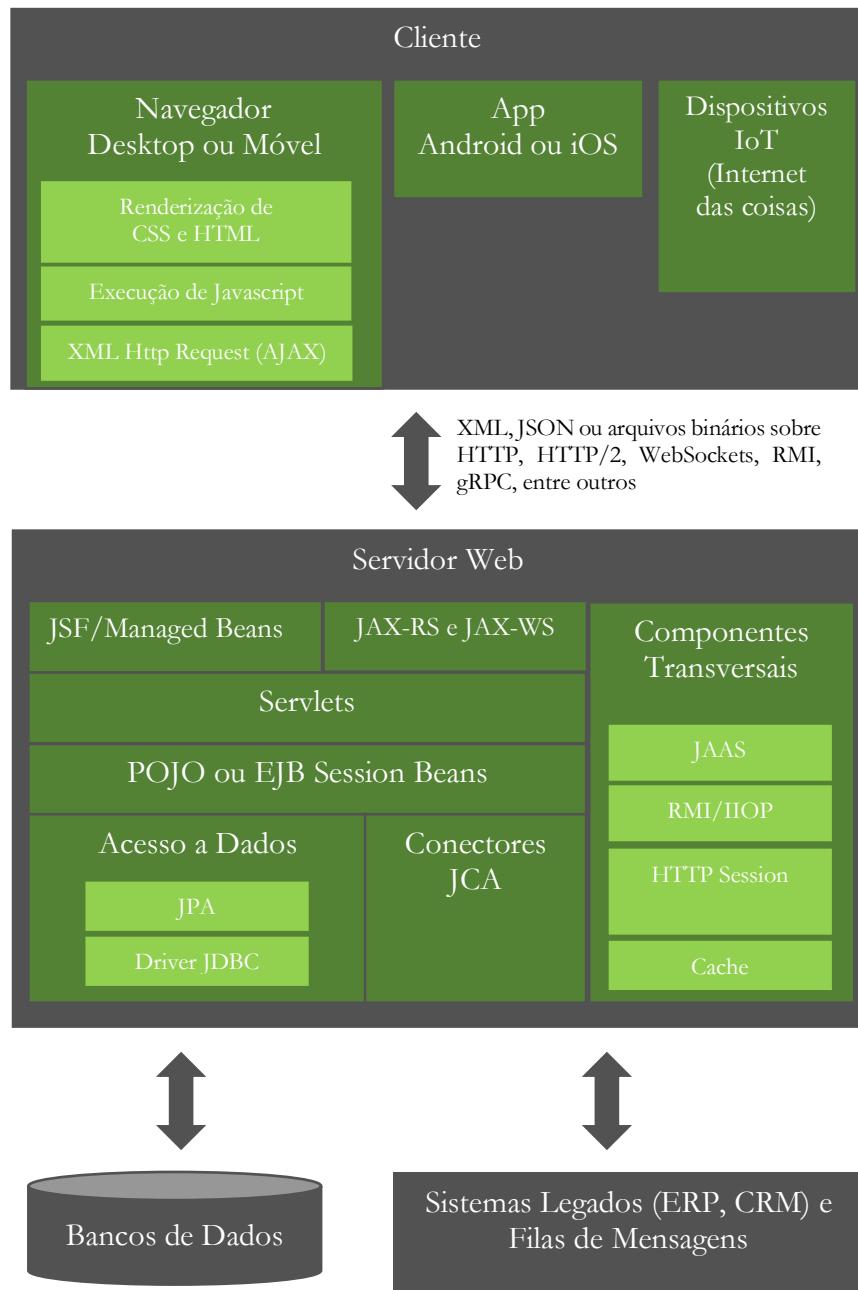


Figura 21: Componentes arquiteturais Java EE Web

9.1 JSF (Java Server Faces)

O JSF é um framework de componentes Web servidor. Em termos práticos, ela tem por objetivo facilitar a criação de componentes Web ricos, encapsulando eventos e código JavaScript. O JSF é construído sobre a tecnologia de páginas dinâmicas chamada de JSP. Algumas bibliotecas de componentes JSF incluem

- PrimeFaces²⁶
- JBOSS RichFaces²⁷
- Apache MyFaces²⁸

Uma página JSF é montada como um arquivo XHTML com uma linguagem de marcação de visão chamada de Facelets²⁹. Os componentes declarados em uma página JSF têm um suporte rico de eventos, que são ligados a classes Java chamada de *beans* gerenciados (*Managed Beans*). Esses beans são classes Java comuns e permitem com isso fazer o tratamento dos eventos Web em linguagem Java.

O JSF, similar ao ASP.NET, tem por premissa trazer o controle de eventos Web para o servidor e com isso reduzir a quantidade de código HTML e JavaScript escrito pelo desenvolvedor. O custo disso é um consumo maior de recursos do servidor Web. Aplicações JSF são conhecidas por fazer uso intenso de memória e esse é um ponto de atenção para arquitetos Web.

Um tutorial introdutório ao JSF está presente no capítulo 7 do guia oficial do Java EE 7, disponível aqui³⁰. Os capítulos de 8 a 16 desse mesmo tutorial apresentam os detalhes de uso dessa tecnologia e exemplos práticos.

9.2 Servlets

Uma servlet é uma classe da linguagem Java que é usada para suportar requisições em protocolos de rede. As servlets mais usadas são a de suporte ao protocolo HTTP e podem ser usadas pelo desenvolvedor para implementar controladores de requisições HTTP em aplicações MVC. Em aplicações Java EE que usam componentes JSF, as servlets são usadas para fazer o controle de requisições, mediando o acesso entre JSF e os componentes de negócio da aplicação.

Um erro arquitetural comum, que deve ser evitado pelo arquiteto Java EE, é usar *servlets* para conter regras de negócio. Regras de negócio em Java EE devem ficar apenas em POJOs ou EJB SessionBeans. Servlets são componentes leves e permitem um controle mais fino dos recursos de um servidor Web, tais como a gerência de sessão (HTTP Session), segurança (JAAS) ou auditoria. Um tutorial introdutório a Servlets está disponível aqui³¹.

9.3 Serviços Web (JAX-WS e JAX-RS)

Algumas aplicações Web Java EE não fazem uso de componentes Web JSF e expõe serviços Web para clientes em outras plataformas cliente. Exemplos incluem:

- Aplicações híbridas JavaScript/Java EE, onde o controlador é feito em linguagem JavaScript. Um exemplo são aplicações HTML/AngularJS/Java EE Web.
- Aplicações que expõe serviços em protocolo SOAP.
- Aplicações que expõe serviços em protocolo HTTP.
- Implementação de serviços SOA ou microsserviços Web.
- Implementação de APIs Web

O Java EE possui duas especificações para a criação de serviços. A primeira e mais antiga especificação é centrada em WS-* (SOAP, WSDL, UDDI e afins) e é chamada de JAX-WS. Algumas implementações JAX-WS incluem:

²⁶ <http://primefaces.org>

²⁷ <http://richfaces.jboss.org>

²⁸ <https://myfaces.apache.org>

²⁹ <https://docs.oracle.com/javaee/7/tutorial/jsf-facelets.htm>

³⁰ <https://docs.oracle.com/javaee/7/tutorial/jsf-intro.htm>

³¹ <https://docs.oracle.com/javaee/7/tutorial/servlets.htm>

- JBOSS WebServices³²
- Apache CXF³³
- Metro³⁴

A segunda especificação, lançada no Java EE 6, é o JAX-RS e permite implementar serviços baseado no estilo arquitetural RESTful. Algumas implementações JAX-RS incluem:

- Apache CXF
- RESTEasy³⁵
- Jersey³⁶

O capítulo 27 do Java EE Tutorial apresenta um panorama de WebServices em Java EE, sendo que o capítulo 28 apresenta o JAX-WS e os capítulos 29 a 31 apresenta o JAX-RS em mais detalhes. Esse material pode ser encontrado aqui³⁷.

Aplicações Web Java EE fazem uso de outras APIs de suporte que, embora não implementem protocolos Web, fornecem serviços úteis para alguns tipos de sistemas de informação.

9.4 JAAS (Java Authentication and Authorization Service)

O serviço de autenticação e autorização é mandatório em todo servidor Web Java EE. Ele fornece para o arquiteto Web:

- Autenticação;
- Autorização;
- Integridade e confidencialidade para o transporte seguro de informações.

O JAAS traz como vantagens o isolamento do código Java dos repositórios de segurança tais como banco de dados, servidores LDAP, Kerberos, OpenID e OAuth e também a configuração declarativa das permissões sobre o código. Essa especificação é explicada em mais detalhes, com exemplos, aqui³⁸.

Algumas implementações de terceiros têm ganhado popularidade e rivalizado com a especificação JAAS. Exemplos incluem o Apache Shiro³⁹ ou o PicketBox⁴⁰.

9.5 EJB (Enterprise Java Beans) Session Beans

Esses são os componentes para a implementação de fachadas de negócio, controlar transações e implementar alguns tipos de regras de negócio. EJBs possuem facilidades para escalabilidade vertical e horizontal e contam com suporte transacional embutido através do JTS (Java Transaction Server).

Esses componentes já foram complexos de usar no modelo EJB 2, mas foram simplificados no EJB 3 e retomaram o seu papel no desenvolvimento de aplicações Web. Exemplos de uso e uma explicação mais detalhada são encontradas aqui⁴¹.

9.6 JPA (Java Persistence API)

A API de persistência Java provê facilidades de mapeamento objeto relacional para gerenciar dados relacionais em aplicações Java, i.e., reduz a quantidade de código SQL escrito e aumenta a produtividade em sistemas de informação. Ou seja, o JPA é um framework de mapeamento objeto relacional (ORM) e implementa facilidades para a paginação de objetos, cache de primeiro e segundo nível, controle transacional e recuperação e gravação de lotes de objetos. O JPA faz uso obrigatório da API de persistência

³² <http://jbossws.jboss.org>

³³ <http://cxf.apache.org>

³⁴ <https://metro.java.net>

³⁵ <http://resteasy.jboss.org>

³⁶ <https://jersey.java.net>

³⁷ <https://docs.oracle.com/javaee/7/tutorial/partwebsvcs.htm>

³⁸ <https://docs.oracle.com/javaee/7/tutorial/partsecurity.htm>

³⁹ <http://shiro.apache.org>

⁴⁰ <http://picketbox.jboss.org>

⁴¹ <https://docs.oracle.com/javaee/7/tutorial/partentbeans.htm>

Java chamada JDBC (*Java Database Connectivity*), que consiste da implementação de drivers que isolam o código Java de banco de dados relacionais. Algumas implementações JPA incluem:

- Hibernate ORM⁴²
- Eclipse Link⁴³

A documentação oficial do JPA é encontrada nos capítulos 37 a 44 do Java EE Tutorial. O sítio da documentação pode ser acessado aqui⁴⁴.

9.7 JCA (Java Connector Architecture)

Em cenários de grandes empresas, aplicações Java EE precisam interoperar com recursos legados como o SAP ECC, CICS, Cobol ou Natural. O JCA é uma tecnologia de apoio para este tipo de cenário. Informações e exemplos de uso podem ser encontrados nos capítulos 51 a 56 do Java EE Tutorial, disponível aqui⁴⁵.

- JMS (Java Messaging Services)

A API JMS permite que aplicações criem, enviem, recebam ou leiam mensagens sobre sistemas de filas de mensagens. Servidores Web Java EE não precisam implementar essa especificação, que é encontrada em servidores de aplicação Java EE como o JBOSS Wildfly. Além disso, existem produtos específicos de mercado que implementam essa tecnologia como por exemplo:

- Apache RabbitMQ⁴⁶
- JBOSS HornetMQ⁴⁷
- IBM WebSphere MQ⁴⁸

O Java EE Tutorial explica o JMS nos capítulos 45 e 46, com exemplos e cenários de uso. A documentação pode ser encontrada aqui⁴⁹.

9.8 Considerações de Desenho Arquitetural

9.8.1 Escolha de servidores

Escolha os servidores mais simples que possam atender ao problema em questão. Já observamos arquitetos que escolheram peças complicadas para o cenário sendo atacado, escolhendo servidores de aplicação onde servidores Web mais simples como o Apache Tomcat poderiam ser selecionados.

Em cenários mais simples ou em estilos de microsserviços, considere tecnologias Java Web alternativas como por exemplo o Spring Boot⁵⁰, que encapsula várias tecnologias Java EE em um modelo de uso mais simples.

Reserve tempo no projeto para ajustar o servidor Web ou servidor de aplicação. As configurações de fábrica e excesso de mensagens de log tornam os mesmos pouco performáticos.

9.8.2 Produtividade

As tecnologias Java EE não vêm combinadas e empacotadas em uma IDE simples. Uma tarefa arquitetural Java EE crítica é pesquisar, testar, selecionar combinar os frameworks em um ambiente de produtividade para o time de desenvolvimento.

O uso de aceleradores de desenvolvimento como geradores de código para casos de uso simples e IDEs apropriadas para o desenvolvimento Web como o Eclipse Java EE⁵¹ ou JetBrains IntelliJ⁵² é importante

⁴² <http://hibernate.org/orm/>

⁴³ <http://www.eclipse.org/eclipselink/#jpa>

⁴⁴ <https://docs.oracle.com/javaee/7/tutorial/partpersist.htm>

⁴⁵ <https://docs.oracle.com/javaee/7/tutorial/partsupporttechs.htm>

⁴⁶ <http://camel.apache.org/rabbitmq.html>

⁴⁷ <http://hornetq.jboss.org>

⁴⁸ <http://www-03.ibm.com/software/products/pt/ibm-mq>

⁴⁹ <https://docs.oracle.com/javaee/7/tutorial/partmessaging.htm>

⁵⁰ <http://projects.spring.io/spring-boot/>

⁵¹ <http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/mars2>

⁵² <https://www.jetbrains.com/idea/>

na arquitetura de aplicações Java EE. Devido a diversidades de bibliotecas e frameworks, o uso de ambientes de automação de tarefas e controle de dependências automação como o Maven⁵³ e o Gradle⁵⁴ é um aspecto importante na construção de sistemas Java EE.

9.8.3 Processamento de Requisições Web

No modelo JSF, o navegador se comunica com o servidor através de eventos nos componentes JSF. Esse modelo fornece uma experiência de desenvolvimento centrada em componentes Web que cuidam da renderização HTML, gestão do estado e a lógica de interação com o navegador. Embora conveniente, esse modelo vem com um custo associado no uso de memória e recursos do servidor. Páginas JSF apresentam uma escalabilidade limitada e devem ser avaliadas com cautela pelo time de arquitetura em conformidade com os requisitos arquiteturais.

A alternativa ao modelo JSF, até o Java EE 7, é o uso de uma abordagem RESTful com o uso de controladores em JavaScript. Nesse modelo, o time de desenvolvimento programa a visão e controles em bibliotecas e frameworks JavaScript como o Angular ou React. A camada Web Java EE cuida da exposição dos serviços (em Servlets) e da lógica de regras de negócio em POJOs e EJBs. Essa alternativa permite um controle mais fino do uso de recursos de servidores Web.

Seja com o uso do JSF, Servlets ou o Java MVC, algumas boas práticas para o controle de requisições Web em aplicações Java EE incluem:

- Centralizar os passos Web de pré-processamento e o pós-processamento em Servlets para promover reuso entre páginas. Exemplos incluem auditorias específicas ou autenticação baseada em IPs. Em termos técnicos, existe uma modalidade de Servlets chamada de *FilterServlets* que podem ser usadas para esse fim.
- Sempre separe as responsabilidades Web com o uso de um padrão arquitetural como o MVC, MVP ou similar.
- Não coloque regras de negócio em Servlets. Deixe isso para os SessionBeans e POJOs.
- Use o suporte do JAAS ou outras soluções como o Spring Security e Apache Shiro para proteger dados sensíveis com o uso do protocolo SSL/TLS.

9.8.4 Navegação

- Mantenha a navegação simples para promover uma melhor experiência de uso.
- Use menus e outras estruturas de apoio para minimizar a quantidade de navegações entre páginas em uma aplicação JSF.
- E como regra geral, não use URIs Web dentro de páginas JSF. Isso acopla a navegação e afeta a manutenibilidade. Use o recurso de navegação JSF, que permite gerar regras de navegação que associam URIs Web reais a nomes virtuais. Esses nomes virtuais, cadastrados no arquivo de configuração JSF, são então usados para fazer a navegação entre páginas.

9.8.5 Desenho de Páginas

- Mantenha as páginas simples.
- Reuse blocos de informação com o recurso de *JSF Templates*.
- Use *layouts* padronizados para melhorar a experiência de uso para os seus usuários.
- Use componentes ricos para interações visuais que lidem com tabelas e grades.
- Mantenha todo e qualquer elemento de configuração visual em arquivos CSS.

9.8.6 Autenticação

⁵³ <https://maven.apache.org>

⁵⁴ <http://gradle.org>

- Garanta o uso de práticas de proteção de contas como travamento de contas, tamanho mínimo de senhas, regras para expiração e alteração de senhas.
- Se as senhas estiverem armazenadas em banco de dados, armazena-as criptografadas.
- Para aplicações que requerem maior nível de proteção, não use banco de dados para armazenar os dados de contas (JDBC Realm do JAAS). Use as facilidades de domínios modulares e empilháveis do JAAS tais como o LDAP Realm para usar sistemas de segurança de terceiros como o Microsoft AD ou OpenLDAP.

9.8.7 Autorização

Estabeleça a priori o modelo de autorização a ser usado, tais como *User Based Access Control*, *Role Based Access Control* ou *Resource Based Access Control*, que fornecem níveis crescentes de controle de acesso a aplicações.

Considere que o JAAS foi desenhado para implementar o modelo baseado em papéis (Role Based). Para aplicações que requeiram um controle de acesso ainda mais granular, considere o uso do padrão baseado em recursos com o uso de frameworks como o Apache Shiro.

9.8.8 Cache

O uso de cache pode aumentar bem o desempenho das suas aplicações Java EE Web. O JPA possui cache de nível 1 (controlado através de configuração) e de nível 2 (controlado através de uma API de programação). Considere o uso apropriado do cache JPA para reduzir o tráfego SQL em aplicações Java. Servidores Web Java EE tem suporte a cache de conteúdo estático, que pode ser usado para evitar o acesso a disco para o carregamento de arquivos, imagens e outros conteúdo estáticos.

9.8.9 Controle Transacional

Sempre que possível, use transações locais ao banco de dados com o JPA. Elas fornecem maior desempenho e escalabilidade. Use transações distribuídas XA como exceção.

Estabeleça um ponto único na aplicação para a demarcação transacional. SessionBeans ou servlets podem ser usados como esses pontos de controle de transação.

9.8.10 Auditoria (logs)

Uma boa auditoria garante confiabilidade para a sua aplicação Java e deve ser realizada em todas as camadas da sua aplicação. Ao mesmo tempo, existem facilidades nos ambientes e servidores que podem tornar essa tarefa mais simples. Ao realizar a auditoria Java EE Web:

Use o suporte embutido de auditoria dos servidores Web, que permitem registrar eventos sobre vários tipos de componentes e camadas Web.

Para o controle personalizado de auditoria, considere o uso de frameworks aceleradores como por exemplo o SLF4J⁵⁵, que fornecem boas APIs para o controle de logs.

Controle o acesso aos arquivos de logs, que podem registrar informações sensíveis e de caráter administrativo apenas.

9.8.11 Instrumentação

Use os recursos apresentados pelos servidores Web e aplicação, que monitoram memória e CPU; Sempre que necessário, considere o uso de instrumentadores da máquina virtual Java. O JConsole, em particular, já vem com o JDK e pode ser configurado para monitorar aplicações Web. Um roteiro básico de uso dessa ferramenta pode ser encontrado aqui⁵⁶.

⁵⁵ <http://www.slf4j.org>

⁵⁶ <http://docs.oracle.com/javase/7/docs/technotes/guides/management/jconsole.html>

O Java EE conta com uma infraestrutura de monitoração chamado de JMX, que pode ser usada para monitoração mais fina de componentes Web. Se necessário, é possível criar seus próprios componentes personalizados JMX. Um tutorial disponibilizado pela própria Oracle pode ser encontrado aqui⁵⁷.

9.8.12 Gerência de Sessão Web

Ao usar de JSF, seja conservador com o tempo de sessão Web devido ao alto consumo de memória. Tempos menores promovem economia de memória enquanto aumentam a segurança do seu sistema. Considere com cautela que dados precisam ser armazenados na sessão Web. Sempre que possível, use escopo de requisição para as variáveis Web.

9.8.13 Validação

Não confie nos dados recebidos. Valide todos os dados que chegam à sua aplicação Java. Ao mesmo tempo, use validadores cliente JSF para aumentar a experiência de uso e reduzir a quantidade de requisições HTTP.

Para maior manutenibilidade do seu código, use a especificação do Bean Validation, que permite criar marcações de validação sobre os seus atributos das classes. O capítulo 21⁵⁸ do Java EE Tutorial apresenta um bom material a respeito.

9.8.14 Desenho de Serviços Web

Sempre que possível, use serviços baseados em JAX-RS. Eles são mais simples e mais performáticos que servidores baseados em SOAP. Para o transporte de dados, de preferência a JSON por serem mais leves que XML.

Use melhores práticas de padronização de serviços para criar APIs robustas e reusáveis.

9.8.15 Camada de negócio

Nem toda aplicação Java EE precisa fazer uso de EJBs, mas ele pode ser um facilitador para controle transacional e escalabilidade horizontal.

Se usar EJBs, reduza o uso de SessionBean com estado, a não ser em situações excepcionais. O uso de Session Beans Stateless (sem estado) deve ser preferido por não estressar a memória do servidor e promover maior escalabilidade.

9.8.16 Acesso a Dados

Em sistemas de informação, uma boa parte do processamento é gasta no acesso às fontes de dados. Por isso, o uso adequado do JPA e drives JDBC é crítico para um bom desempenho da sua aplicação. Algumas recomendações incluem:

Saber comparar e selecionar o tipo de driver JDBC mais apropriado ao seu cenário. Os drives JDBC, classificados como tipo 1, 2, 3 e 4, possuem vantagens e desvantagens que devem ser reconhecidos pelos arquitetos Java EE. Como regra geral, escolha o driver do tipo 2 (nativo) para aplicações que requerem mais performance. Escolha o driver do tipo 4 (100% puro Java) para aplicações que requerem maior portabilidade entre sistemas operacionais. O driver tipo 1 é hoje considerado legado e o driver do tipo 3 foi desenhado para o uso como middlewares de banco de dados e caiu em desuso nos dias atuais.

Usar com extrema atenção as anotações JPA, que tem impacto direto na performance das queries SQL. Sempre configurar a memória reservada para o cache nível 1 JPA. Quando necessário, considerar o uso de cache nível 2 JPA.

⁵⁷ <https://docs.oracle.com/javase/tutorial/jmx/>

⁵⁸ <https://docs.oracle.com/javaee/7/tutorial/bean-validation.htm>

9.9 Riscos e Oportunidades para o Arquiteto Web Java EE

A arquitetura Java EE fornece muitas possibilidades de uso e se tornou bastante popular no mercado mundial nas últimas duas décadas. No Brasil, vemos o seu uso com intensidade nos órgãos governamentais, universidades, centros de pesquisa e empresas de grande porte.

Arquitetar uma aplicação Java EE é crítico e pode ser considerado um campo minado. Talvez a arquitetura Java EE Web seja a mais complexa, quando comparada a ASP.NET, LAMP e Node.js. Por isso, apresentamos aqui uma lista de riscos comuns em aplicações Java EE e estratégias de mitigação.

9.9.1 Curva de Aprendizado Alta

A tecnologia Java EE possui detalhes e requer conhecimentos estabelecidos na plenitude apenas em cursos de ciência da computação em boas universidades. Portanto, o arquiteto Java EE deve:

- Usar arquiteturas Java EE simples e provadas;
- Ter cuidado no uso de recomendações de terceiros que não foram provadas no seu contexto;
- Reservar tempo para treinar o seu time durante o projeto;
- Sempre fornecer um modelo real de um tipo de caso de uso do projeto (ex. relatório simples, cadastro, mestre-detalhe, pesquisa).

9.9.2 Baixa Produtividade

Quando comparado ao LAMP ou o ASP.NET, o Java EE puro possui menor produtividade. Em termos de IDE, o Java EE não tem um ambiente tão produtivo como o Visual Studio para .NET. Portanto, o arquiteto Java EE deve considerar:

- Uso de poucas camadas arquiteturais e frameworks simples;
- Uso de componentes ricos Web e boas bibliotecas JSF.
- Uso de aceleradores arquiteturais como por exemplo o Play Framework⁵⁹;
- Usar linguagens de mais alto nível que rodam no topo de máquinas virtuais como por exemplo o Scala⁶⁰, Groovy⁶¹ e *scaffoldings* como o Grails⁶²;
- Realizar provas de conceito de produtividade antes que o projeto comece (ou no início dele);
- Trabalhar as expectativas de produtividade com os gestores. O Java EE é mais uma tecnologia de robustez e segurança do que um ambiente de desenvolvimento rápido de aplicações.

9.9.3 Consumo de Memória

Aplicações JSF podem ser grandes consumidoras de memória e degradar a escalabilidade de aplicações Java. Para evitar isso, considere:

- Testar a performance das suas aplicações JSF;
- Manter as páginas simples;
- Não usar JSF e usar ao invés frameworks JavaScript controladores e servlets para exposição de serviços RESTful.

9.9.4 Descontinuação Tecnológica de Frameworks

Diversos frameworks Java EE Web sofrem obsolescência. O leitor que já trabalha há algum tempo com Java EE desde o início do século já ouviu falar de frameworks como Struts, Tapestry, Velocity ou Wicket,

⁵⁹ <https://www.playframework.com>

⁶⁰ <http://www.scala-lang.org>

⁶¹ <http://www.groovy-lang.org>

⁶² <https://grails.org>

entre dezenas de outros frameworks que foram populares e depois esquecidos. Mesmo algumas bibliotecas JSF, outrora populares, se tornaram obsoletas em um outro momento. O arquiteto Java EE deve gerir expectativas com o seu corpo gestor e estar preparado para precisar evoluir a arquitetura entre 5 a 10 anos. Apesar da complexidade e riscos aqui apresentados, algumas pesquisas no Brasil mostram que desenvolvedores e arquitetos Java EE ganham, em média, 20 a 30% mais que arquitetos .NET e JavaScript. Além disso, esse ambiente apresenta oportunidades permanentes de aprendizado, que aumenta a empregabilidade dos arquitetos que trabalham nessa tecnologia.

9.10 Para Saber Mais

A literatura de Java EE é extensa e é formada na sua maioria por livros de tecnologias específicas. Mas se queremos nos aprofundar a arquitetura de sistemas Java EE, uma boa referência é o livro de Adam Bien (Bien, 2012), que apresenta o Java EE 6 e o padrão arquitetural ECB. O Java EE 7 Tutorial (Oracle, 2013) também tem uma boa cobertura dos padrões e especificações do Java EE.

Se você é um adepto de Java EE e quer saber das últimas novidades pelo canal oficial da Oracle, acompanhe a evolução da JSR 366 (Java EE 8) a partir deste sítio⁶³.

⁶³ <https://jcp.org/en/jsr/detail?id=366>

10 Servidores Web Baseados em ASP.NET

O ASP.NET é o modelo oficial de desenvolvimento de aplicações Web da Microsoft. Ele evoluiu a partir da primeira geração de páginas dinâmicas da Microsoft, O ASP (*Active Server Pages*). A Microsoft o mantém hoje como uma tecnologia aberta. Ela não possui custos de propriedade e o código fonte já é disponibilizado para a comunidade no portal do GitHub.

Lançado ainda em 2002, o ASP.NET teve evolução e possui um conjunto amplo de APIs para o desenvolvimento de aplicações Web. Algumas dessas APIs incluem:

- ASP.NET Web Forms
- ASP.NET MVC (versões 1 a 5)
- ASP.NET Web API
- ASP.NET Core (ASP.NET MVC 6 e ASP.NET Web API 2)

Desenvolvedores Web podem construir aplicações ASP.NET sobre qualquer linguagem que opere sobre a máquina virtual do .NET Framework. Vemos o uso dominante da linguagem C#, seguido da linguagem VB.NET em algumas empresas.

- O ASP.NET apresenta como principais vantagens arquiteturais:
- Mão de obra de desenvolvimento Web. Em 2016 o ASP.NET é a tecnologia Web servidora mais usada no mercado brasileiro.
- Aceleradores de produtividade, com destaque para o Visual Studio.
- Facilidades para componentização de aplicações;
- Serviços de suporte como controle transacional distribuído, segurança, cache e escalabilidade;
- A partir do ASP.NET Core, suporte multiplataforma em Linux e OS/X;
- A partir do ASP.NET Core, operação no IIS ou como aplicações auto hospedadas;
- Diversidade de fornecedores.

A

Figura 22 mostra os principais componentes da arquitetura Web Microsoft.

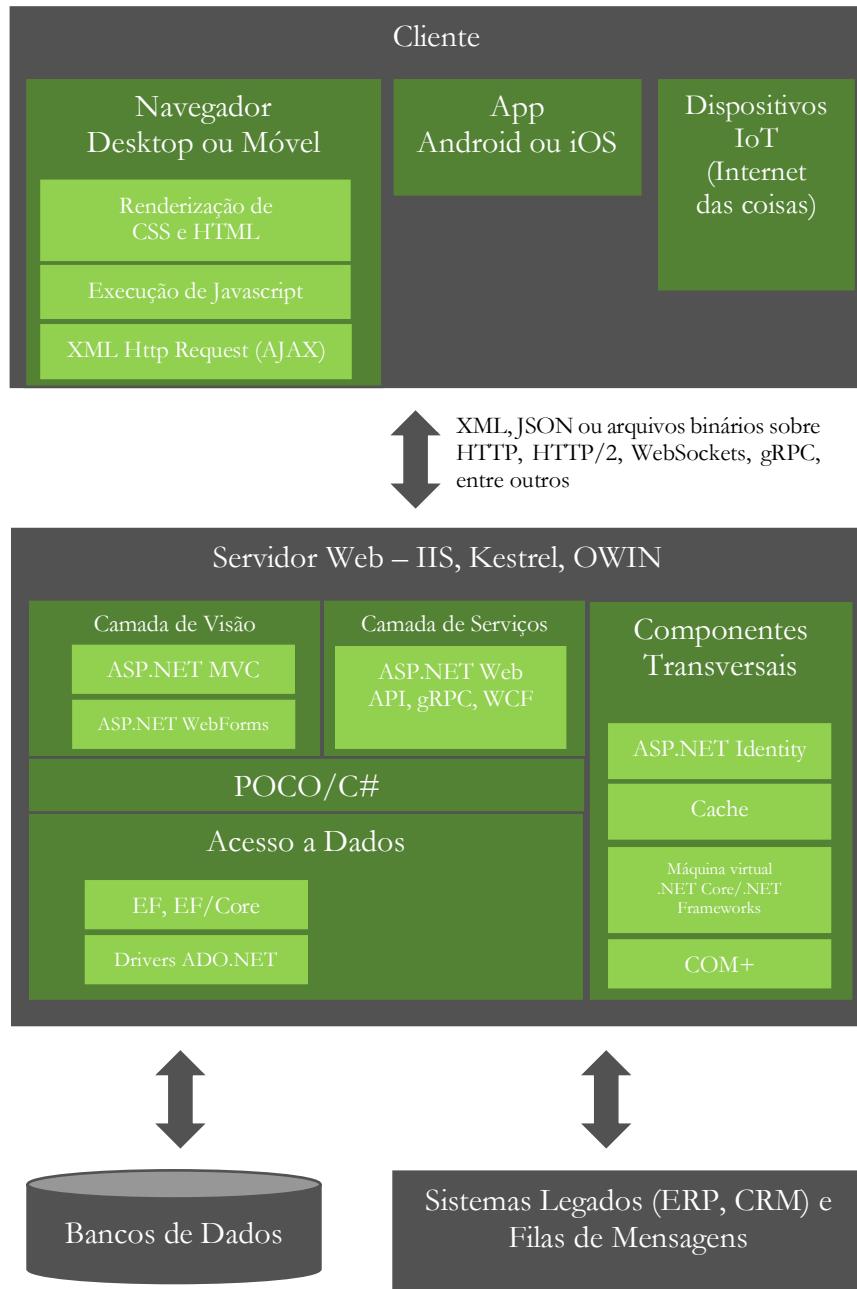


Figura 22: Componentes arquiteturais ASP.NET

Toda aplicação ASP.NET é uma aplicação que roda sobre uma máquina virtual (.NET Framework ou .NET Core). O .NET Framework opera apenas em ambiente Windows. Mas existem versões também para outros sistemas operacionais com o projeto Xamarin Mono⁶⁴ e o Microsoft .NET Core⁶⁵.

O .NET Framework/.NET Core oferece os serviços de base para uma aplicação .NET em dois componentes arquiteturais:

- Common Language Runtime (máquina virtual)
- Biblioteca de classes

⁶⁴ <http://www.mono-project.com>

⁶⁵ <https://dotnet.github.io>

O **Common Language Runtime**, ou CLR, é um agente que gerencia o código em tempo de execução. Ele provê serviços essenciais como gerenciamento de memória, gerenciamento de threads e comunicação remota. Ele também aplica segurança de tipos quem promove segurança para sua aplicação .NET em ambientes Web. Ele é similar ao JVM para aplicações Java e Java EE.

Já a **biblioteca de classes** do .NET Framework é uma coleção de tipos reutilizáveis que se integram com o *Common Language Runtime*. A biblioteca de classes é orientada à objetos, fornecendo tipos que seu próprio código gerenciado pode derivar. Além disso, componentes de terceiros podem se integrar com classes do .NET Framework. Nas seções seguintes são descritas as principais tecnologias Web Microsoft.

10.1 ASP.NET Web Forms

O ASP.NET Web Forms⁶⁶ foi a primeira tecnologia ASP.NET lançada pela Microsoft, sucedendo o modelo ASP.NET. Essa tecnologia possui componentes Web ricos que encapsulam a aparência e os eventos Web. Ou seja, o desenvolvedor não precisa de conhecimento mais avançado de programação HTML, JavaScript e CSS. Isso é ainda potencializado pelo suporte de desenvolvimento rápido de aplicações do Visual Studio. Aqui o desenvolvedor trabalha da seguinte forma:

- Arrasta e solta os componentes da paleta;
- Liga desses componentes com as fontes de dados;
- E então programa os eventos necessários para dar vida aos componentes.

O modelo de desenvolvimento do Windows Forms⁶⁷ foi o modelo inspirador para o ASP.NET Web Forms. Ele apresenta como vantagem a produtividade. Desenvolvedores Web com experiência limitada conseguem produzir aplicações com boa velocidade. Podemos dizer que esse modelo de programação foi fundamental na popularidade atual do ASP.NET no Brasil.

Páginas ASP.NET WebForms usam o modelo de *postback*. Aqui o desenvolvedor precisa primeiro criar a página com o formulário rico. Depois ele cria uma classe que irá tratar os eventos que ocorrem nessa página. Essa classe é o *code behind*. Quando uma interação é originada através de um cliente, o servidor Web em resposta irá criar uma instância da classe do *code behind*. O estado dos controles visuais da página associada ao código fica então disponível. Com isso o desenvolvedor pode manipular os controles da página conforme necessário.

Esse modelo apresenta algumas desvantagens, descritas abaixo:

- **Baixa portabilidade.** Códigos rodam apenas no IE (Internet Explorer) e servidor Web IIS (Internet Information Services), pois o controle da geração do HTML e JavaScript é realizado pelos servidores Web e motor do ASP.NET. Usar o ASP.NET Web Forms implica em limitar a sua aplicação ao navegador IE. Até mesmo o navegador Microsoft Edge, disponível no Windows 10, não tem suporte a esse tipo de aplicação.
- **Baixa escalabilidade e alto consumo de memória.** O ASP.NET Web Forms tem um consumo razoável de CPU e memória dos servidores Web. Não é incomum empresas que possuam aplicações ASP.NET WebForms tenham problemas de vazamento de memória, baixo desempenho e indisponibilidades nos seus ambientes de produção.
- **Baixa separação de camadas arquiteturais.** O ASP.NET WebForms não enfatiza a criação de modelos de domínio com classes OO relacionadas devido ao seu modelo RAD – *Rapid Application Development* com componentes de tela ligados a componentes de dados (DataSets). Embora isso seja produtivo, um domínio de classes estável não emerge e a aplicação se torna centrada em telas e dados (2 camadas).

Podemos já considerar esse modelo de desenvolvimento legado e recomendado apenas em cenários de Intranet, onde podemos ter controle do navegador, com equipes internas de desenvolvimento que tenham baixo conhecimento OO.

⁶⁶ <http://www.asp.net/web-forms>

⁶⁷ [https://msdn.microsoft.com/en-us/library/dd30h2yb\(v=vs.110\).asp](https://msdn.microsoft.com/en-us/library/dd30h2yb(v=vs.110).asp)

10.2 ASP.NET MVC

O ASP.NET MVC surgiu como um modelo de desenvolvimento alternativo ao ASP.NET Web Forms com os seguintes objetivos arquiteturais:

- Fornecer uma separação mais clara entre visão, controladores e modelo, permitindo que códigos OO mais robustos com o uso do padrão arquitetural de modelo de domínio (*Domain Model*⁶⁸) emergissem.
- Maior controle da geração do HTML e JavaScript, permitindo assim portabilidade real entre navegadores Web.

O ASP.NET MVC possui diversas versões, sendo as principais:

- ASP.NET MVC 1 – Lançada em março de 2009 no framework ASP.NET 1.0 e .NET Framework 1.0.
- ASP.NET MVC 2 – Lançada em março de 2010 no framework ASP.NET 2 e .NET Framework 2.
- ASP.NET MVC 3 – Lançada em janeiro de 2011 no framework ASP.NET 3 e .NET Framework 3.
- ASP.NET MVC 4 – Lançada em agosto de 2012 no framework ASP.NET 4 e .NET Framework 4.
- ASP.NET MVC 5 – Lançada em outubro de 2013 no framework ASP.NET 4.5 e .NET Framework 4.5.

A partir do ASP.NET MVC 3, diferentes motores de visualização foram incorporados na tecnologia, tais como o Razor⁶⁹ (páginas .cshtml ou .vbhtml) ou o WebForms View Engine (páginas com extensão .aspx). Em termos práticos, o Razor oferece a possibilidade de um código de visualização mais simples.

As aplicações ASP.NET MVC (versões 1 a 5) são baseadas em Windows e rodam dentro do servidor Web IIS. Já o ASP.NET MVC 6, que faz parte do modelo de desenvolvimento Web multiplataforma chamado de ASP.NET Core é explicado mais a frente nesse capítulo.

Talvez a principal diferença do ASP.NET MVC é o seu mecanismo de roteamento de requisições. Ao invés do modelo de *postbacks*, essas aplicações possuem o seguinte ciclo de vida:

1. Uma requisição cliente aciona um controlador. Esse controlador é uma classe C# ou VB.NET que tem a capacidade de responder requisições dentro de uma URL específica, com estado representacional REST.
2. O controlador aciona as classes de modelo e faz acesso às APIs de persistência conforme necessário.
3. Após receber os dados e processamento de regras de negócio, o controlador despacha a requisição para uma visão (página ASP.NET com motor Razor ou WebForms), que exibe os dados vindos do modelo para o cliente.

Esse modelo traz algumas consequências arquiteturais:

- Mais flexibilidade na aparência e comportamento Web. Em termos práticos, o ASP.NET MVC viabilizou aplicações Web 2 e SPA dentro do mundo Microsoft.
- Capacidade de geração de código OO, com modelos de domínio robustos e frameworks de mapeamento objeto relacional como por exemplo o Entity Framework ou o nHibernate.

10.3 WCF

O WCF (Windows Communication Foundation) é um modelo unificado de programação para o desenvolvimento de serviços e microsserviços. Embora ele não seja parte integrante do ASP.NET, ele é

⁶⁸ <http://martinfowler.com/eaaCatalog/domainModel.html>

⁶⁹ <http://www.asp.net/web-pages/overview/getting-started/introducing-razor-syntax-c>

muitas vezes usado para suportar serviços em aplicações Web Microsoft. Ele foi inspirado em modelos de componentes de objetos como o COM⁷⁰, COM+⁷¹, com facilidades adicionais para controle dos protocolos de transporte e dados.

O WCF pode ser comparado a EJBs dentro do mundo Java EE. Ou seja, ele pode ser usado para a montagem de componentes transacionais, escaláveis e com segurança integrada. Ele pode operar sobre o IIS ou de forma independente em aplicações auto hospedadas que operam sobre o .NET Framework.

Serviços WCF foram desenhados para serem expostos em protocolos SOAP com contratos WSDL (WS-*), mas podem ser configurados para expor serviços em outros protocolos como HTTP, TCP ou Named Pipes. Os formatos de dados também são configuráveis sem necessidades de programação e incluem arquivos textos XML, JSON, binários e MTOM⁷². Uma lista completa dos protocolos ofertados pelo WCF está disponível nesse sítio⁷³.

Em termos arquiteturais, o WCF ainda é a API recomendada para o arquiteto criar e dispor serviços Web baseados em protocolo WSDL/SOAP.

A última versão do WCF, em maio de 2016, é a 4.5. Uma documentação de apoio da Microsoft pode ser encontrada aqui⁷⁴.

10.4 ASP.NET Core 1

O ASP.NET Core, cujo primeiro nome era ASP.NET 5, foi uma reescrita completa da arquitetura de aplicações Web da Microsoft. A figura abaixo mostra essa perspectiva, comparando com o modelo oficial do ASP.NET 4.6.

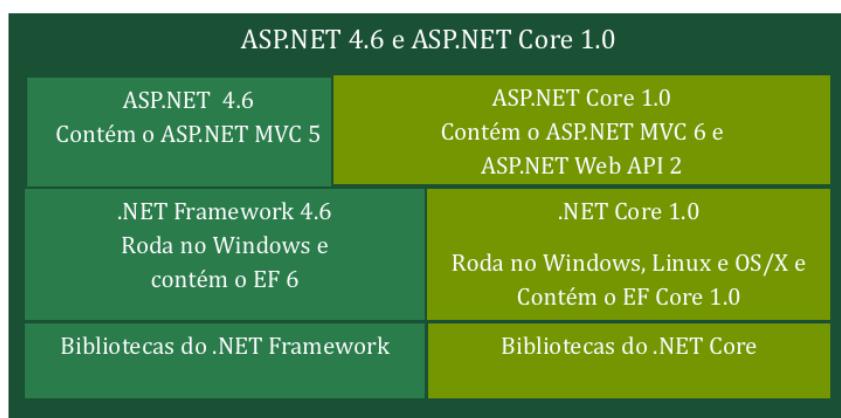


Figura 23: ASP.NET Core 1.0 versus o ASP.NET 4.6

O primeiro aspecto a observar é que o ASP.NET Core opera sobre o ASP.NET 4.6 e também sobre o .NET Framework 4.6. Em termos práticos, você pode desenvolver aplicações ASP.NET Core em Linux e OS/X com o VSCode ou trabalhar no Windows no VSCode ou no Visual Studio 2015.

Ao mesmo tempo, veja que as aplicações ASP.NET Core não tem mais dependência obrigatória do .NET Framework, que funciona em ambiente Windows. Você pode rodar essas aplicações sobre uma nova máquina virtual multiplataforma, chamada de .NET Core 1.0. Essa máquina virtual está disponível em ambiente Linux e OS/X. Similar a uma JVM Java, ela permite que agora que você rode aplicações .NET em outros sistemas operacionais e fora do servidor Web IIS.

Em termos de desenvolvimento, a API de desenvolvimento do ASP.NET Core é bastante similar a do ASP.NET MVC. As principais diferenças foram:

- A descontinuidade do ASP.NET Web Forms, que não é mais suportado nessa plataforma devido à sua dependência Windows;

⁷⁰ <https://www.microsoft.com/com/default.mspx>

⁷¹ [https://msdn.microsoft.com/library/ms685978\(VS.85\).aspx](https://msdn.microsoft.com/library/ms685978(VS.85).aspx)

⁷² https://en.wikipedia.org/wiki/Message_Transmission_Optimization_Mechanism

⁷³ [https://msdn.microsoft.com/en-us/library/ms731092\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/ms731092(v=vs.110).aspx)

⁷⁴ [https://msdn.microsoft.com/en-us/library/dd456779\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/dd456779(v=vs.110).aspx)

- A incorporação do ASP.NET Web API 2.0 como parte integrante desse framework. O Web API é uma biblioteca usada para a montagem de aplicações com estilo arquitetural de APIs Web;
- O lançamento de uma IDE paralela ao Visual Studio, chamada de VS Code, mais leve e centrada em edição e refatoração de código. O VS Code⁷⁵ é uma IDE multiplataforma e tem semelhanças com IDEs leves e populares de desenvolvimento como o Atom⁷⁶ e o Sublime⁷⁷.

O .NET Core já tem suporte oficial da Microsoft e está na versão 3. A Microsoft disponibiliza uma trilha rica de tutoriais e recursos de apoio para trabalhar com esse framework. Um bom ponto de partida está neste sítio: <https://docs.asp.net/en/latest/>.

A despeito disso, arquitetos devem ainda usar o ASP.NET 4.6, IIS e Windows como ambientes oficiais em produção até que esse novo framework ganhe maturidade e estabilidade.

10.5 ASP.NET CORE 2 e 3

O ASP.NET Core evoluiu nos últimos anos e teve dois lançamentos centrais (ASP.NET Core 2 em 2017 e ASP.NET Core 3 em 2019). Detalhamos aqui o estado tecnológico do ASP.NET Core 3.

O [ASP.NET](#) Core oferece os seguintes benefícios, além daqueles originalmente introduzidos em 2015:

- O [Razor Pages](#) torna a codificação de cenários focados em página mais fácil e produtiva.
- O [Blazor](#) permite que você use C# no navegador junto com o JavaScript. Compartilhe a lógica de aplicativo do lado do cliente e do servidor toda escrita com o .NET.
- Suporte para hospedagem de serviços RPC (chamada de procedimento remoto) usando [gRPC](#).
- Um [sistema de configuração](#) pronto execução em nuvem.
- Capacidade de hospedagens nos seguintes ambientes
 - [Kestrel](#)
 - [IIS](#)
 - [HTTP.sys](#)
 - [Nginx](#)
 - [Apache](#)
 - [Docker](#)

10.6 ASP.NET Web API

Em um movimento global de desenvolvimento de software, houve uma maior adoção de padrões REST para o desenvolvimento de serviços Web, em oposição a serviços Web WS-*. Os motivos foram simplicidade, facilidade de desenvolvimento e manutenção. Dentro do modelo de desenvolvimento da Microsoft, isso levou ao desenvolvimento uma nova API mais simples, orientada a REST, para o desenvolvimento de serviços Web. O modelo de programação do ASP.NET Web API⁷⁸ é simples e requer que o desenvolvedor crie classes com os métodos de negócio e acesso a dados e faça anotações nessas classes. Essas anotações irão expor os métodos na Web dentro de URLs bem formados REST para invocações GET, POST, PUT, PATCH e DELETE, entre outros métodos HTTP. O ASP.NET Web API 2 foi incorporado dentro do ASP.NET Core 1.0 e será evoluído em conjunto as novas versões do ASP.NET MVC. Um bom conjunto de tutoriais e documentações de apoio pode ser encontrado aqui⁷⁹. Em termos arquiteturais, o ASP.NET Web API é o componente recomendado para a criação de APIs REST e até mesmo microsserviços.

⁷⁵ <https://code.visualstudio.com>

⁷⁶ <https://atom.io>

⁷⁷ <https://www.sublimetext.com>

⁷⁸ <http://www.asp.net/web-api>

⁷⁹ <http://www.asp.net/web-api/overview/getting-started-with-aspnet-web-api/tutorial-your-first-web-api>

10.7 Entity Framework e LINQ

Aplicações modernas ASP.NET MVC precisam de acesso simplificado e facilitado a bancos de dados relacionais. A Microsoft desenvolveu para este propósito uma API de mapeamento objeto relacional chamada Entity Framework (EF). As versões do Entity Framework são:

- Entity Framework 1.0, lançada no .NET Framework 3.5 em Agosto de 2008
- Entity Framework 4.0, lançada no .NET Framework 4.0 em Abril de 2010.
- Entity Framework 4.1, com suporte ao modelo *Code First* em Abril de 2011.
- Entity Framework 5.0, lançada no .NET Framework 4.5 em Agosto de 2012
- Entity Framework 6.0, desvinculado do .NET Framework. Foi lançada em Outubro de 2013.
- Entity Framework Core 1.0, lançada com o ASP.NET Core 1.0 no começo de 2016.

O EF tem o seu acesso acelerado por um suporte funcional na linguagem C#, que é o LINQ⁸⁰, que reduz a quantidade de código que precisa ser escrito para manipular o banco de dados.

O EF possui algumas abordagens de desenvolvimento, que incluem:

- *Database First*⁸¹
- *Model First*⁸²
- *Code First*⁸³

Escolha o modelo *Code First* para novos desenvolvimentos por fornecer maior versatilidade no controle do mapeamento, a menos que você esteja fazendo engenharia direta (Model First) ou reversa (Database First) do banco de dados. Vale lembrar que a ferramenta de modelagem gráfica do Entity Framework funciona com os modelos *Model First* e *Database First*. Existe também o recurso do EF de migrações de código (*Code First Migrations*), que facilita o trabalho de refatorações contínuas no seu código e modelo de dados.

Como o projeto do EF demorou a evoluir em termos arquiteturais, durante os últimos anos um framework independente ganhou também popularidade no mundo Microsoft. Esse framework é o nHibernate⁸⁴, baseado no já popular framework ORM Java Hibernate.

Ambos esses frameworks (EF e nHibernate) tem o seu trabalho também facilitado por um acelerador de configuração chamado FluentAPI⁸⁵. No contexto do Entity Framework, ele é usado para projetos EF Code First.

10.8 MSMQ

Esta é uma tecnologia da Microsoft para o suporte a filas de mensagens, que é um importante acelerador arquitetural para trabalhar em cenários que dependem de escalabilidade e tolerância a falhas. O MSMQ (Microsoft Message Queue) é similar ao JMS do Java e suporta filas (canais 1 para 1) e tópicos (canais 1 para muitos). A Microsoft possui também extensa documentação^{86 87} sobre esta tecnologia, que vem instalada em todo servidor Windows.

10.9 Service Fabric

Nos últimos anos, as plataformas de nuvens se tornaram populares. E com o recente advento de

⁸⁰ <https://msdn.microsoft.com/pt-br/library/bb397906.aspx>

⁸¹ <https://www.asp.net/mvc/overview/getting-started/database-first-development/setting-up-database>

⁸² <https://msdn.microsoft.com/en-us/data/j205424.aspx>

⁸³ <https://msdn.microsoft.com/en-us/data/jj193542>

⁸⁴ <http://nhibernate.info>

⁸⁵ <https://msdn.microsoft.com/en-us/data/jj591617.aspx>

⁸⁶ [https://msdn.microsoft.com/en-us/library/ms711472\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/ms711472(v=vs.85).aspx)

⁸⁷ <https://msdn.microsoft.com/en-us/library/bb969123.aspx>

plataformas de serviços e microsserviços, a Microsoft possui também um produto específico para a criação de plataforma de microsserviços em nuvens públicas ou privativas, chamado de Malha de Serviços (Service Fabric).

O Azure Service Fabric⁸⁸ é uma plataforma de sistemas distribuídos que facilita o empacotamento, implantação e gerenciamento de microsserviços escalonáveis e confiáveis.

Muitas das atuais tecnologias de nuvem da Microsoft já são criadas sobre esta malha de serviços, incluindo o Banco de Dados SQL do Azure, Azure DocumentDB, Cortana, Microsoft Power BI, Microsoft Intune, Hubs de Eventos do Azure, Hub IoT do Azure e o Skype for Business.

A arquitetura de referência desta malha de serviços é apresentada abaixo.

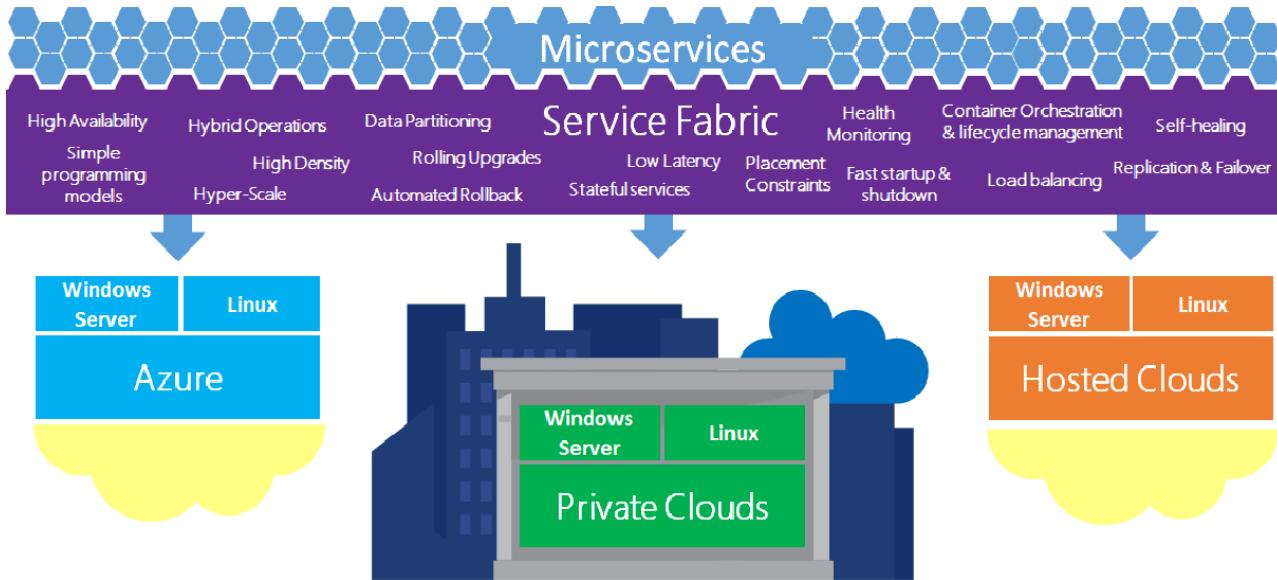


Figura 24: Arquitetura de Referência do Service Fabric

Fonte: <https://docs.microsoft.com/pt-br/azure/service-fabric/service-fabric-overview>

O Service Fabric é um orquestrador de microsserviços em um cluster de computadores (locais ou na nuvem do Azure). Os microsserviços podem ser desenvolvidos de várias maneiras usando os modelos de programação do Service Fabric. O Service Fabric também pode implantar serviços de imagens de contêiner com o uso de tecnologias com o Docker. É importante observar que você pode misturar serviços em processos e serviços em contêineres no mesmo aplicativo. Se você quiser apenas implantar e gerenciar imagens de contêiner em um cluster de computadores, o Service Fabric é uma opção interessante.

O Service Fabric permite compilar aplicativos que consistam em microsserviços. Os microsserviços sem estado (como gateways de protocolo e proxies Web) não mantêm um estado mutável fora de uma solicitação e sua resposta do serviço. As funções de trabalho dos Serviços de Nuvem do Azure são um exemplo de serviço sem estado. Os microsserviços com estado (como contas de usuário, bancos de dados, dispositivos, carrinhos de compra e filas) mantêm um estado mutável e autorizativo além da solicitação e sua resposta. Os aplicativos em escala da Internet de hoje consistem em uma combinação de microsserviços com e sem estado.

10.10 Considerações de Desenho Arquitetural

10.10.1 Configuração de servidores

Reserve tempo no projeto para ajustar o IIS. As configurações de fábrica e excesso de mensagens de log tornam os mesmos pouco performáticos.

⁸⁸ <https://docs.microsoft.com/pt-br/azure/service-fabric/service-fabric-overview>

Tenha especial cuidado na configuração da sua aplicação ASP.NET no IIS. O IIS tem componentes que podem ser usados em conjunto ou de forma isolada. O guia de arquitetura do IIS⁸⁹ da Microsoft é um excelente ponto de partida para entender a arquitetura desse servidor e seus pontos de configuração. Com o advento do ASP.NET Core, é possível operar aplicações ASP.NET fora do IIS, rodando-a direto sobre o .NET Framework e sistema operacional da máquina. Isto é recomendado em cenários mais simples e aplicações de microsserviços. Para cenários clássicos e empresas que requerem uma publicação em um servidor Web com segurança centralizada, devemos fazer o processo tradicional de publicações de aplicações no IIS⁹⁰.

10.10.2 Produtividade

O Visual Studio é um potente acelerador de produtividade e é disponibilizado em versão sem custo, também excelente. A última versão, chamada de VS Community 2020⁹¹, pode ser baixada daqui.

Times .NET tem se beneficiado de uma poderosa ferramenta de suporte a refatoração de código, chamada ReSharper⁹², que opera como um plug-in para o Visual Studio.

Alguns times também tem usado o suporte a personalização de *scaffolding*⁹³ (andaimes) e DSLs (Domain Specific Language)⁹⁴ do Visual Studio para acelerar tarefas repetitivas como cadastros e aumentar a produtividade de seus times.

Times .NET que possuem o Visual Studio Ultimate⁹⁵ podem usar os seus facilitadores para avaliar as suas arquiteturas físicas com a engenharia reversa de suas aplicações.

E para quem prefere o uso de ferramentas abertas, a grande novidade é o uso do VSCode que é uma IDE gratuita, multiplataforma e multilinguagem que se tornou nos últimos 5 anos a IDE mais usada no planeta.

10.10.3 Processamento de Requisições Web

No modelo ASP.NET Web Forms, o navegador se comunica com o servidor através de eventos nos componentes. Embora conveniente e produtivo, esse modelo vem com um custo associado no uso de memória e recursos do servidor. Páginas ASP.NET Web Forms, assim como JSF, apresentam uma escalabilidade limitada e devem ser avaliadas com cautela pelo time de arquitetura em conformidade com os requisitos arquiteturais.

Use o modelo do ASP.NET MVC com o motor de visualização do Razor para garantir melhor performance, portabilidade e integração facilitada com frameworks JavaScript e CSS.

Algumas boas práticas para ASP.NET incluem:

- Centralizar os passos Web de pré-processamento e o pós-processamento para promover reuso entre páginas. Exemplos incluem logs, auditorias ou autenticação baseada em IPs.
- Sempre separar as responsabilidades Web com o uso de um padrão arquitetural como o MVC, MVP ou similar.
- Proteja os dados trafegados e uso as APIs de Data Protection⁹⁶ do ASP.NET para facilitar esse trabalho.

⁸⁹<http://www.iis.net/learn/get-started/introduction-to-iis/introduction-to-iis-architecture#Components>

⁹⁰<https://docs.asp.net/en/latest/publishing/iis.html>

⁹¹<https://www.visualstudio.com/pt-br/products/visual-studio-community-vs.aspx>

⁹²<https://www.jetbrains.com/resharper/>

⁹³<https://msdn.microsoft.com/pt-br/magazine/dn745864.aspx>

⁹⁴<https://msdn.microsoft.com/en-us/library/ee943825.aspx>

⁹⁵<https://msdn.microsoft.com/en-us/library/57b85fsc.aspx>

⁹⁶<https://docs.asp.net/en/latest/security/data-protection/index.html>

10.10.4 Navegação

- Mantenha a navegação simples para promover uma melhor experiência de uso.
- Use menus e outras estruturas de apoio para minimizar a quantidade de navegações entre páginas em uma aplicação ASP.NET.
- Como regra geral, não use URIs Web dentro de páginas ASP.NET. Isso acopla a navegação e afeta a manutenibilidade. Use o recurso de navegação do ASP.NET (Site Navigation⁹⁷), que permite gerar regras de navegação que associam URIs Web reais a nomes virtuais.

10.10.5 Desenho de Páginas

Mantenha as páginas simples.

- Use *layout Razor*⁹⁸ padronizados para melhorar a experiência de uso para os seus usuários.
- Use componentes ricos para interações que possam usar tabelas e grades. A Microsoft disponibiliza componentes ricos na sua API⁹⁹, mas existem outros fornecedores de mercado que possuem componentes ainda mais poderosos. Exemplos incluem o Kendo-UI¹⁰⁰, da Telerik, ou o Ext.NET¹⁰¹, da Object.Net.
- Mantenha todo e qualquer elemento de configuração visual em arquivos CSS.

10.10.6 Autenticação

Garanta o uso de práticas de proteção de contas como travamento de contas, tamanho mínimo de senhas, regras para expiração e alteração de senhas. Se as senhas estiverem armazenadas em banco de dados, armazene-as criptografadas. A Microsoft mantém em seu site boas práticas para gestão de senhas¹⁰² no ASP.NET.

Recomenda-se usar as facilidades já presentes no framework ASP.NET Identity¹⁰³, que já tem suporte para contas individuais, contas organizacionais em repositórios LDAP ou contas do próprio Windows para aplicações de Intranet. Aplicações de maior criticidade de segurança devem usar métodos com autenticação de dois fatores¹⁰⁴, também presente no ASP.NET Identity.

Para aplicações de Internet e nuvem que possam usar autenticação aberta, considere o suporte do ASP.NET Identity com o OAuth¹⁰⁵.

10.10.7 Autorização

Escolha a priori o seu modelo de autorização, que pode ser baseado em visão, recursos, políticas, dados ou uma combinação destes.

O ASP.NET possui vários modelos de autorização de aplicações (*Simple Authorization, Role Based Authorization, Claims Based Authorization, Resource Bases Authorization, View Based Authorization e Custom Policy-Based Authorization*)¹⁰⁶. Esses modelos permitem um controle fino dos recursos da sua aplicação, enquanto mantém mínima intervenção no código fonte.

⁹⁷ <https://msdn.microsoft.com/en-us/library/e468hxky.aspx>

⁹⁸ <http://www.asp.net/web-pages/overview/ui-layouts-and-themes/3-creating-a-consistent-look>

⁹⁹ <https://docs.asp.net/en/latest/mvc/views/view-components.html>

¹⁰⁰ <http://www.telerik.com/aspnet-mvc>

¹⁰¹ <http://ext.net>

¹⁰² <http://www.asp.net/identity/overview/features-api/best-practices-for-deploying-passwords-and-other-sensitive-data-to-aspnet-and-azure>

¹⁰³ <http://www.asp.net/visual-studio/overview/2013/creating-web-projects-in-visual-studio#auth>

¹⁰⁴ <http://www.asp.net/identity/overview/features-api/two-factor-authentication-using-sms-and-email-with-aspnet-identity>

¹⁰⁵ <https://azure.microsoft.com/en-us/documentation/articles/web-sites-dotnet-deploy-aspnet-mvc-app-membership-oauth-sql-database/>

¹⁰⁶ <https://docs.asp.net/en/latest/security/authorization/introduction.html>

10.10.8 Cache

O uso de cache pode aumentar bem o desempenho das suas aplicações ASP.NET. Considere o uso da API de gerenciamento de memória para otimizar o acesso às suas páginas ASP.NET¹⁰⁷.

O EF possui cache de nível 1 (controlado através de configuração) e de nível 2 (controlado através de uma API de programação). Considere o uso apropriado do cache de nível 2¹⁰⁸ do EF para reduzir o tráfego SQL em aplicações ASP.NET.

Servidores Web, como o IIS, tem suporte a cache de conteúdo estático, que pode ser usado para evitar o acesso a disco para o carregamento de arquivos, imagens e outros conteúdo estáticos. O Cache de Output do IIS pode ser importante no desempenho e escalabilidade do seu site e pode ser configurado conforme as instruções disponíveis aqui¹⁰⁹.

10.10.9 Controle Transacional

Sempre que possível, use transações locais ao banco de dados. Elas fornecem maior desempenho e escalabilidade. Use transações distribuídas XA apenas como exceção. O ASP.NET fornece esse suporte no pacote System.Transactions¹¹⁰. Estabeleça um ponto único na arquitetura lógica da sua aplicação para a demarcação transacional.

10.10.10 Auditoria (logs)

Uma boa auditoria garante confiabilidade para a sua aplicação ASP.NET e deve ser realizada em todas as camadas da sua aplicação. Ao mesmo tempo, existem facilidades nos ambientes e servidores que podem tornar essa tarefa mais simples. Ao realizar a auditoria em .NET, considere:

- Usar o suporte embutido de auditoria dos servidores Web, que permitem registrar eventos sobre vários tipos de componentes e camadas Web.
- Controlar o acesso aos arquivos de logs, que podem registrar informações sensíveis e de caráter administrativo apenas.

10.10.11 Instrumentação

Use as facilidades de instrumentação do seu ambiente e não reinvente a roda. O Windows possui uma excelente ferramenta nativa de monitoração, o Performance Monitor¹¹¹. Ela pode ser usada para observarmos o estado do disco, memória, CPU, rede, servidor IIS, bancos de dados e também das aplicações que foram criadas pelo arquiteto .NET.

Use os recursos apresentados pelos servidores Web e aplicação, que monitoram memória e CPU. O IIS, em particular, possui facilidades diversas para instrumentação de aplicação e contadores específicos de monitoração de desempenho¹¹². O próprio ASP.NET possui uma infraestrutura nativa para a monitoração¹¹³ de aplicações, diagnóstico de problemas e log de eventos.

10.10.12 Gerência de Sessão Web

Seja restritivo com o tempo de sessão Web devido ao potencial consumo de memória. Tempos menores promovem economia de memória enquanto aumentam a segurança do seu sistema. Considere com cautela que dados precisam ser armazenados na sessão Web. Sempre que possível, use escopo de requisição para as variáveis Web.

¹⁰⁷ <https://docs.asp.net/en/latest/performance/caching/index.html>

¹⁰⁸ <https://msdn.microsoft.com/en-us/magazine/hh394143.aspx>

¹⁰⁹ <http://www.iis.net/learn/manage/managing-performance-settings/configure-iis-7-output-caching>

¹¹⁰ [https://msdn.microsoft.com/en-us/library/ms254973\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/ms254973(v=vs.110).aspx)

¹¹¹ [https://technet.microsoft.com/pt-br/library/cc749249\(v=ws.11\).aspx](https://technet.microsoft.com/pt-br/library/cc749249(v=ws.11).aspx)

¹¹² <http://www.solarwinds.com/topics/microsoft-iis-monitor>

¹¹³ [https://msdn.microsoft.com/en-us/library/ms178703\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/ms178703(v=vs.85).aspx)

10.10.13 Validação

Não confie nos dados recebidos. Valide todos os dados que chegam à sua aplicação ASP.NET. Ao mesmo tempo, use validadores cliente ASP.NET¹¹⁴ para aumentar a experiência de uso e reduzir a quantidade de requisições HTTP.

10.10.14 Desenho de Serviços Web

Sempre que possível, use serviços baseados em ASP.NET Web API (ao invés de WCF). Eles são mais simples e mais performáticos que servidores baseados em SOAP. Para o transporte de dados, dê preferência a JSON por serem mais leves que XML. Use melhores práticas de padronização de serviços para criar APIs robustas e reusáveis.

Se a sua intenção é avançar de forma massiva para uma arquitetura de microsserviços, considere o uso da estrutura do Service Fabric. Esta solução oferta uma solução profissional para lidar com a gestão de microsserviços em ambiente de nuvens.

10.10.15 Acesso a Dados

Em sistemas de informação, uma boa parte do processamento é gasta no acesso às fontes de dados. Por isso, o uso adequado do acesso a dados em ASP.NET é crítico para um bom desempenho da sua aplicação. Algumas recomendações incluem:

- Saber comparar e selecionar o driver de banco de dados ADO.NET mais apropriado ao seu cenário.
- Usar as facilidades do LINQ, que melhora a manutenibilidade e até mesmo a performance da sua aplicação.
- Usar com muita atenção as anotações de mapeamento do nHibernate ou Entity Framework, que tem impacto direto na performance das queries SQL.
- Sempre configurar a memória reservada para o cache nível 1. Quando necessário, considerar o uso de cache nível 2 do nHibernate ou Entity Framework.

10.11 Riscos e Oportunidades para o Arquiteto Web ASP.NET

A arquitetura .NET é a que oferece o maior número de novos projetos de desenvolvimento de software no Brasil nessa década. Isso tem aumentado a demanda por arquitetos .NET, em projetos que não apenas envolvem o ASP.NET, mas outros elementos da arquitetura e produtos Microsoft como o CRM Dynamics, portal Sharepoint e o SQL Server. Alguns riscos e oportunidades nesse campo incluem.

10.11.1 Estruturação de bons modelos de domínio e bons códigos

A IDE do Visual Studio fornece tantos aceleradores que alguns desenvolvedores descuidam do desenho de classes. Um aspecto importante para arquitetos é garantir que boas práticas de separação de código, uso de padrões de desenho, organização de classes e suas relações e a prática da refatoração contínua dentro do ambiente Microsoft.

10.11.2 Consumo de Memória em Aplicações ASP.NET WebForms

Aplicações WebForms podem ser grandes consumidoras de memória e degradar a escalabilidade do IIS. Para evitar isso, considere:

- Testar a performance das suas aplicações ASP.NET Web Forms;
- Manter as páginas simples;

¹¹⁴ <https://msdn.microsoft.com/en-us/library/bwd43d0x.aspx>

- Usar ASP.NET MVC sempre que possível.

10.11.3 A Nova Arquitetura ASP.NET Core

ASP.NET Core 1.0 marca a entrada firme da Microsoft no mundo Linux em termos de ambientes e linguagens de programação. Isso traz muitas oportunidades para arquitetos pois o conhecimento já sedimentado no Windows e usado no passado em aplicações .NET terá que ser evoluído ou recriado para cenários multiplataforma. Esse aumento da complexidade ambiental da plataforma .NET é um terreno fértil de oportunidades para desenvolvedores que queiram trilhar também o caminho da arquitetura de aplicações.

10.11.4 Mapeamento de Tecnologias Java EE e .NET

Este capítulo se encerra com uma comparação arquitetural das plataformas Java EE e .NET, estudadas neste e no capítulo anterior. Uma análise arquitetural conjunta de .NET e Java EE permite estabelecer correlações, fazer comparações ou mesmo se apropriar de uma outra arquitetura para desenvolvedores que queiram evoluir suas habilidades arquiteturais. Para ligar as informações apresentadas neste e no capítulo anterior,

Tabela 6 fornece um comparativo arquitetural .NET e Java EE para táticas usadas no mundo Web.

Tática Arquitetural	.NET	Java EE
Páginas Web de primeira geração	ASP	JSP
Páginas Web de segunda geração	ASP.NET	JSF
Controladores MVC	Classes C#	Servlets
Motores de visualização	Razor ou WebForms	JSTL
APIs RESTful	ASP.NET Web API	JAX-RS
APIs WS-*	WCF	JAX-WS
Mapeamento Objeto Relacional	Entity Framework	JPA
Acesso a banco de dados	ADO.NET	JDBC
Segurança	ASP.NET Identity	JAAS
Filas de Mensagens	MSMQ ou ServiceFabric	JMS
Microsserviços	ASP.NET Web API ou Service Fabric	JMS
Máquina Virtual	.NET Framework .NET Core .NET Core for Docker Service Fabric	JVM

Tabela 6: Comparativo arquitetural entre Java EE e .NET

10.12 Passado, Presente e Futuro da Plataforma .NET Core

A primeira versão principal do .NET Core foi focada em aplicações web e em microsserviços de alto desempenho. Um ano depois, com o anúncio da [versão 2.0](#), vários componentes e APIs foram adicionados, para facilitar a migração de aplicativos web para o .NET Core. O InfoQ discutiu o [lançamento do .NET Core 2.0](#) e o seu futuro, com desenvolvedores veteranos da comunidade. As principais vantagens apontadas foram o reconhecimento do .NET Core como plataforma estável, pronta para o desenvolvimento de novos aplicativos, além de benefícios de desempenho significativos em relação ao .NET Framework.

Já o destaque do .NET Core 3.0 foi o [suporte a aplicativos desktop no Windows](#), com foco em Windows Forms, Windows Presentation Framework (WPF) e UWP XAML. Na época desta versão, o .NET Standard foi posicionado como base comum para o Windows Desktop Apps e o .NET Core. Além disso, o .NET Core foi apresentado como parte de uma composição contendo o ASP.NET Core, o Entity

Framework Core e o [ML.NET](#). O suporte ao desenvolvimento e ao porte de aplicativos desktop do Windows para o .NET Core seria fornecido pelos "Windows Desktop Packs" (componentes adicionais para plataformas Windows).

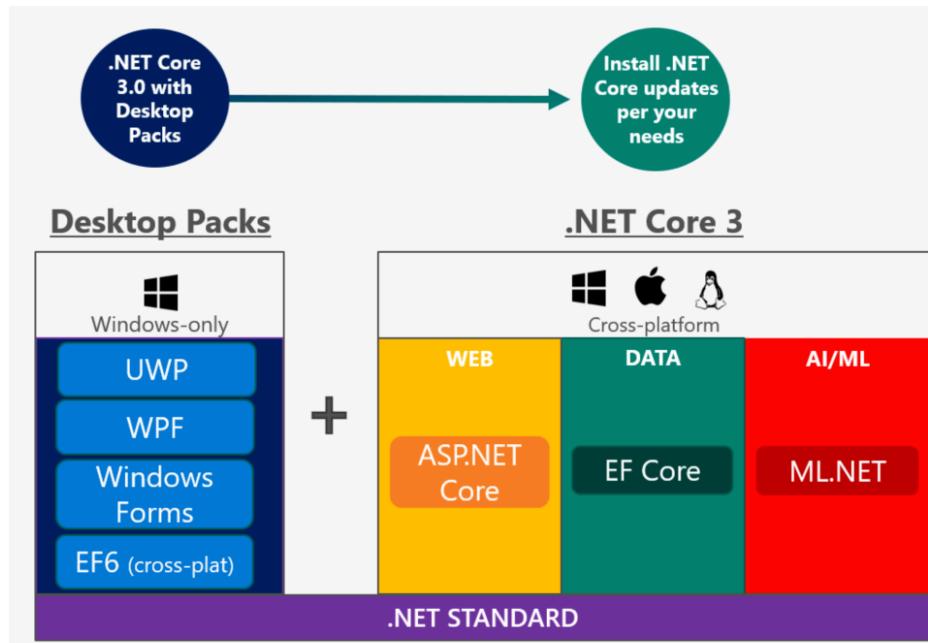


Figura 25: Arquitetura de Referência do .NET 5

Fonte: [Infoq.Com](#)

Para os próximos anos, a plataforma irá evoluir uma versão chamada .NET 5 (ou vNext) descrita na figura abaixo.

.NET – A unified platform



Figura 26: Arquitetura de Referência do .NET Core 5 (Vnext)

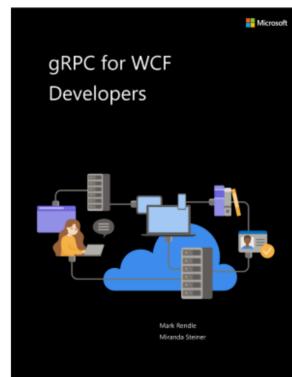
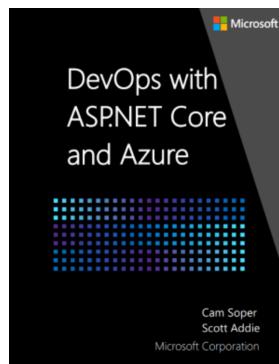
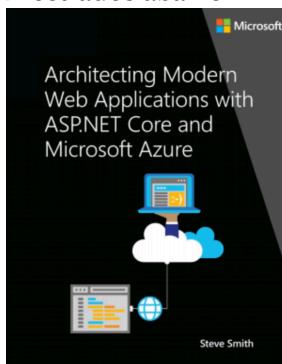
Fonte: [Infoq.Com](#)

A Microsoft posicionou o .NET 5 como plataforma unificadora para aplicativos e tecnologias desktop, web, em nuvem, IoT e mobile, além de para jogos e inteligência artificial — com integração explícita a todas as edições do Visual Studio e também à interface de linha de comandos (CLI). O objetivo da nova

versão .NET é produzir um único runtime multiplataforma do .NET, integrando os melhores recursos das tecnologias .NET Core, .NET Framework, Xamarin e [Mono](#) (a implementação multiplataforma original do .NET).

10.13 Para Saber Mais

Uma excelente fonte é o portal de arquitetura da Microsoft - <https://dotnet.microsoft.com/learn/dotnet/architecture-guides> que possui excelentes livros sobre os diversos componentes da arquitetura. Alguns dos livros disponíveis gratuitamente nesse portal são mostrados abaixo.



11 Servidores Web Baseados em JavaScript

A linguagem JavaScript foi criada pela empresa Netscape em 1995 e foi incorporada no seu navegador, o Netscape Navigator, que durante um bom período dos anos 90 foi o mais popular da Internet. Ainda nos anos 90, a linguagem JavaScript teve uma função limitada no desenvolvimento Web. Como as páginas tinham pouca interatividade, o JavaScript era usado para tarefas simples validações de dados no lado do cliente, pequenos efeitos visuais e manipulação da árvore DOM. Além disso, a Microsoft tinha a sua própria linguagem de script, o VBScript, e tínhamos nesse momento do tempo um mercado dividido para a programação cliente.

No começo do século XXI a crença dominante era que qualquer programação séria na Web deveria ser realizada no servidor em tecnologias como PHP, Python, Ruby, JSF e ASP.NET. Mesmo depois que a Web 2.0 se popularizou, a partir de 2005, as tecnologias servidoras Web continuaram a manter o seu domínio arquitetural. Entre 2005 e 2010 as aplicações Web foram ainda mais desafiadas em interatividade e escalabilidade. A premissa de cuidar da geração dos elementos HTML, CSS e JavaScript gerados por componentes do lado servidor era ainda dominante e algumas tecnologias levaram isso ao extremo, como os hoje legados Google Web Toolkit (GWT) e Adobe Flex (Flash), que eliminavam quase na totalidade a necessidade de conhecimento JavaScript, CSS e HTML.

A partir de 2010, com a popularização dos ambientes de nuvens, ambientes móveis e estilos Web como o SPA as aplicações Web começaram a requerer ainda mais usabilidade, portabilidade, manutenibilidade e escalabilidade. Em resposta a isso, vários desenvolvedores começaram a potencializar o uso do JavaScript. A partir de bibliotecas utilitárias como o JQuery, o ecossistema JavaScript cresceu entre 2010 e 2015. Ferramentas de produtividade frameworks MVC, de testes de unidade, gerenciadores de dependências e outras já comuns em ASP.NET, Java EE e LAMP foram reproduzidas para JavaScript. O lançamento oficial da linguagem HTML 5 e do CSS 3 em 2014 potencializou ainda mais esse movimento. Em 2016 a linguagem JavaScript já é a mais usada para desenvolver aplicações Web no mundo, sendo usada inclusive para o desenvolvimento de código servidor. Podemos olhar a extensão desse panorama no diagrama arquitetural da

Figura 27.

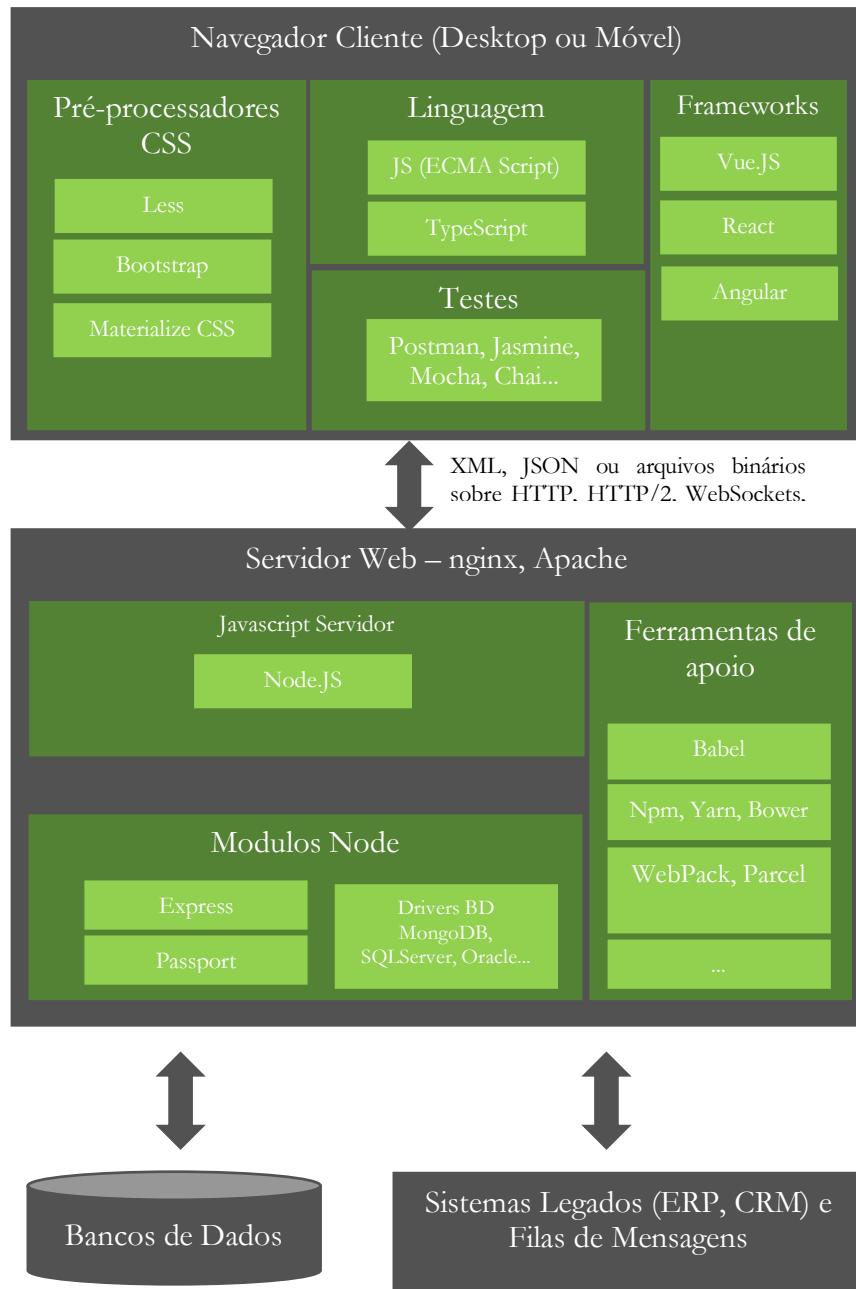


Figura 27: Componentes de uma arquitetura baseada em JavaScript, CSS e HTML

Os componentes dessa figura serão explicados em detalhes ao longo desse capítulo. De forma simplificada podemos ver que o ecossistema JavaScript/CSS é amplo e inclui os seguintes tipos de frameworks:

- Pré-processadores e bibliotecas CSS, como por o Bootstrap. Esse tipo de biblioteca facilita a organização visual das páginas e suas principais funções envolvem aumentar a qualidade gráfica das páginas HTML e aumentar portabilidade entre navegadores e portabilidade em dispositivos móveis com telas pequenas.
- Linguagens baseadas em JavaScript, como por o Microsoft TypeScript. Essas linguagens têm por objetivo aumentar a produtividade da escrita de código cliente e fornecer abstrações sobre o JavaScript tradicional.
- Bibliotecas de componentes JavaScript, que tem por objetivo facilitar a manipulação de eventos, tratamento da árvore DOM, animações e fornecer componentes visuais mais ricos. O jQuery, criado ainda em 2006, é um exemplo de biblioteca nesse sentido.

- Frameworks MV*, que introduzem uma mudança arquitetural significativa ao trazer o controlador para execução no cliente. Um exemplo é o Google AngularJS. Com esse tipo de framework, os componentes de visão e controlador (VC) são processados no navegador. Apenas o modelo (M) é executado no servidor através de serviços. Isso torna a aplicação mais leve, pois muitas requisições HTTP entre o cliente e o servidor são eliminadas. Além disso, esses frameworks também trazem *data-bindings* entre as visões e os dados que ficam disponibilizados nos controladores cliente, rebatizados de *View Models*. Isso acelera bastante o desenvolvimento pois evita código JavaScript desnecessário.
- Ferramentas de testes de unidade como o Jasmine, baseadas em tecnologias como o PHPUnit, JUnit ou VSUnit, melhoraram a manutenibilidade de aplicações JavaScript ao introduzir a automação de testes para esse tipo de código.
- Ferramentas de suporte para o gerenciamento de dependências como o npm, yarn, bower ou gerenciamento de tarefas como o Grunt.
- Componentes para execução de código JavaScript no servidor, que podem substituir a necessidade do uso de programação em C#, Java ou PHP.

Somados a IDEs JavaScript de primeira linha, como por exemplo o JetBrains WebStorm¹¹⁵, VS Code¹¹⁶, Atom¹¹⁷ ou Sublime¹¹⁸, os desenvolvedores Web tem hoje um rico ambiente para o desenvolvimento de aplicações Web.

A pilha descrita na

¹¹⁵ <https://www.jetbrains.com/webstorm/>

¹¹⁶ <https://code.visualstudio.com>

¹¹⁷ <https://atom.io>

¹¹⁸ <https://www.sublimetext.com>

Figura 27 não precisa ser adotada na sua totalidade. A

Figura 28 mostra essa configuração híbrida, com código Java EE, ASP.NET ou LAMP e código JavaScript no cliente.

Sobre essa figura, devemos observar que existem alguns conflitos e zonas de sombra, que devem ser avaliados pelos arquitetos Web. O uso de frameworks MV* é um exemplo. O AngularJS, por exemplo, procura fazer as mesmas funções de controladores em ASP.NET e JSF e, portanto, conflita com esses frameworks. Ou seja, se você usa o AngularJS ou outro framework MV* em JavaScript, o servidor ASP.NET ou Java EE irá conter apenas serviços Web em tecnologias como o ASP.NET Web API ou JAX-RS.

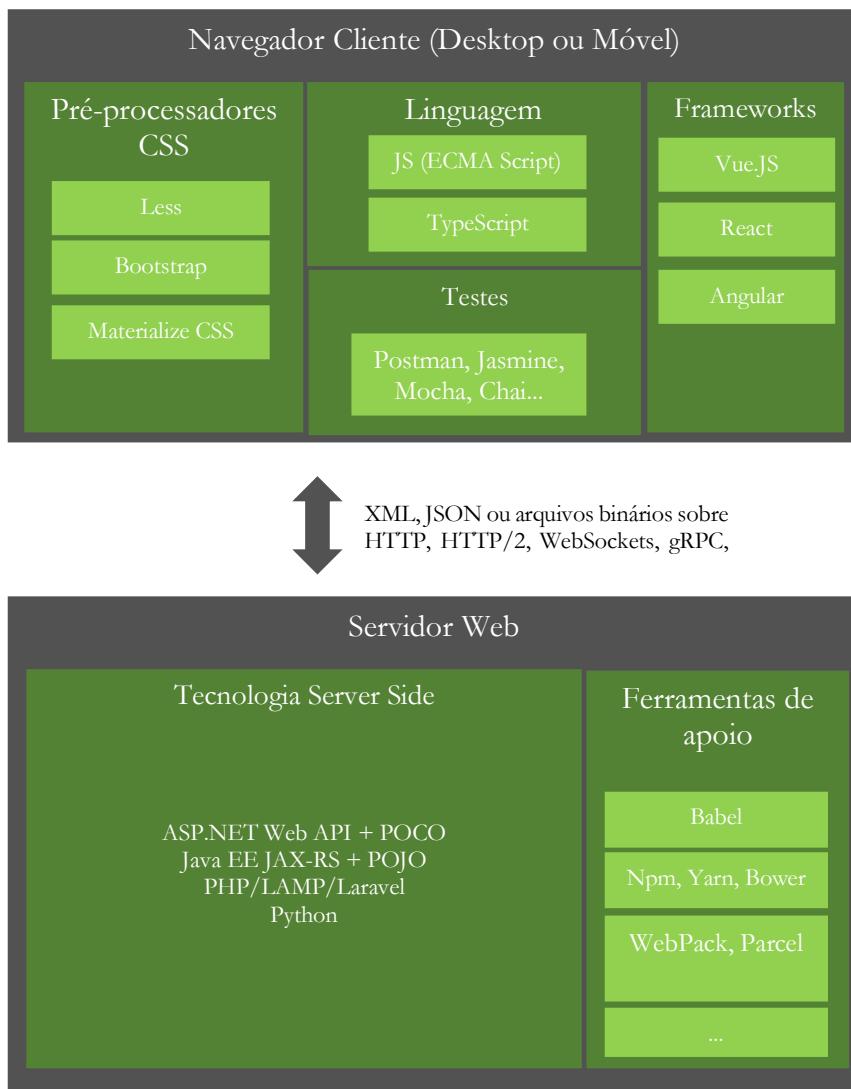


Figura 28: Componentes de uma arquitetura híbrida JavaScript com JavaEE, .NET ou LAMP

As seções seguintes detalham os componentes arquiteturais CSS e JavaScript e as suas implicações para o arquiteto Web.

11.1 Aceleradores, Bibliotecas e Frameworks CSS

As folhas de estilo em cascata¹¹⁹, ou CSS, são uma especificação da W3C. O seu uso está associado ao controle da aparência da apresentação de páginas HTML. Elas permitem separar a estrutura de componentes Web (que fica no HTML) da sua forma de apresentação em termos de tipografia, cores organização espacial ou tamanho do dispositivo.

O CSS parece ter pouco interesse para o arquiteto e embora o seu uso avançado seja mais importante para o *Web Designer*, esta é uma tecnologia que tem implicações arquiteturais importantes.

Vimos na seção **Error! Reference source not found.** do capítulo 3 que existem hoje estilos arquiteturais Web centrados em aspectos estéticos de interação. Nesse contexto o CSS ganha maior atenção do arquiteto, sendo importante adotar as seguintes recomendações:

Uso de materiais de estilo conceitual para desenho Web responsivo se os requisitos de acessibilidade graves. O Material Design¹²⁰ do Google é um projeto nesse sentido. Inspirado no trabalho de desenho

¹¹⁹ <http://www.w3schools.com/css/>

¹²⁰ <https://material.google.com>

Web feito pelo Google nos aplicativos Android a partir versão Lollipop (5.0), este sítio fornece um guia de estilo para aplicações responsivas. A Microsoft também possui um projeto semelhante, chamado de UWP – Universal Windows Platform Design.

Uso de bibliotecas CSS para aumentar a produtividade dos times. Algumas bibliotecas já implementam o conceito de Material Design, como por exemplo o MaterializeCSS¹²¹ ou Material-UI¹²². Algumas dessas bibliotecas, como o Less¹²³ ou o Bootstrap¹²⁴, já são populares entre desenvolvedores Web cliente.

Uso de padrões para a organização de código CSS e melhorar a manutenibilidade de aplicações. Um padrão interessante a respeito é o BEM (*Block, Element and Modifier*), que separa objetos HTML (blocos), funções e modificadores de blocos¹²⁵.

11.2 Linguagens Aceleradoras sobre JavaScript

A linguagem JavaScript, que foi apresentada no capítulo 1, manipula a página Web através de uma representação conceitual do HTML mantida pelo navegador, chamada de modelo de objeto de documentos (DOM¹²⁶). O conteúdo da árvore DOM pode ser manipulado à vontade pelo JavaScript, o que permite a interatividade em páginas Web. Ao mesmo tempo, chamadas JavaScript ao DOM são blocantes. Isso poderia inviabilizar o desempenho da construção das suas páginas e por isso JavaScript faz uso intenso de comunicações assíncronas, possibilitada pelo mecanismo conhecido como AJAX (JavaScript assíncrono e XML, potencializado pela tecnologia XHR – XML HTTP Request¹²⁷).

Um exemplo mínimo é mostrado a seguir, onde uma função JavaScript faz uma chamada ao servidor através de um método GET. Essa chamada opera de forma assíncrona, i.e., não trava qualquer renderização do cliente. Enquanto os dados são retornados pelo servidor, a resposta é exibida na seção do HTML marcada pelo campo myDiv.

```

1  <script type="text/javascript">
2      function carregaDadosCliente() {
3          var xmlhttp = new XMLHttpRequest();
4
5          xmlhttp.onreadystatechange = function() {
6              if (xmlhttp.readyState == XMLHttpRequest.DONE ) {
7                  if(xmlhttp.status == 200){
8                      document.getElementById("myDiv").innerHTML =
9                          xmlhttp.responseText;
10                 }
11             else if(xmlhttp.status == 400) {
12                 alert('Veio um erro 400');
13             }
14             else {
15                 alert('Algum outro erro vindo do servidor');
16             }
17         };
18     };
19
20     xmlhttp.open("GET", "dados_cliente.json", true);
21     xmlhttp.send();
22 }
23 </script>
24
25

```

Figura 29: Código JavaScript com exemplo AJAX

O código acima representa bem o funcionamento assíncrono de uma aplicação Web. Quando a função é chamada através de algum evento na tela, uma chamada assíncrona é realizada ao servidor. Nesse exemplo, os dados do servidor no recurso dados_cliente.json são retornados e então exibidos em uma seção myDiv do HTML.

¹²¹ <http://materializecss.com>

¹²² <http://www.material-ui.com>

¹²³ <http://lesscss.org>

¹²⁴ <http://getbootstrap.com>

¹²⁵ <https://www.toptal.com/css/introduction-to-bem-methodology>

¹²⁶ <http://www.w3.org/DOM/>

¹²⁷ <http://www.w3.org/TR/XMLHttpRequest2/>

Essa natureza assíncrona do JavaScript é benéfica para o desempenho e interatividade das aplicações, mas tornou a manutenibilidade um grande problema. É comum que códigos JavaScript sejam ilegíveis e isso se torna uma preocupação arquitetural em desenvolvimentos Web de larga escala. Nesse sentido, o mercado começou a observar alguns aceleradores nesse sentido, que incluem:

- Linguagens aceleradoras como CoffeScript¹²⁸, que reduzem a quantidade de linhas de código escrita. Um exemplo é mostrado no fragmento abaixo, onde duas funções idênticas são definidas nas duas linguagens. O código CoffeScript é mais conciso e simples de manter.

```

1  // Aqui é JavaScript puro
2  square = function(x) {
3      return x * x;
4  };
5
6
7  cubes = (function() {
8      var i, len, results;
9      results = [];
10     for (i = 0, len = list.length; i < len; i++) {
11         num = list[i];
12         results.push(math.cube(num));
13     }
14     return results;
15 })();
16
17 // Aqui é CoffeScript
18 square = (x) -> x * x
19
20 cubes = (math.cube num for num in list)
21

```

Figura 30: Fragmento de Código JavaScript versus CoffeScript

- Uso de linguagens OO como o TypeScript¹²⁹ da Microsoft. Essa linguagem, que conta já com suporte forte no VSCode e outras IDEs introduz facilidades de tipagem, interfaces e classes e torna a programação JavaScript mais próxima do C# e Java. Um exemplo mínimo é mostrado na Figura 31.

```

1  class Student {
2      fullName: string;
3      constructor(public firstName, public middleInitial, public lastName) {
4          this.fullName = firstName + " " + middleInitial + " " + lastName;
5      }
6
7
8  interface Person {
9      firstName: string;
10     lastName: string;
11 }
12
13 function greeter(person : Person) {
14     return "Alo, " + person.firstName + " " + person.lastName;
15 }
16
17 var user = new Student("Joao", "M.", "Silva");
18
19 document.body.innerHTML = greeter(user);
20
21 // Saida mostra: Alo, Joao Silva

```

Figura 31: Fragmento de Código TypeScript

Em termos técnicos, essas linguagens são compiladas ou traduzidas para JavaScript. Alguns autores chamam esse processo de *transilação (transpiling)*, que é suportado por ferramentas de traduzem JavaScript de mais alto nível para o JavaScript original. Um exemplo de uma ferramenta nesse sentido é o Babel¹³⁰

¹³¹

¹²⁸ <http://coffeescript.org>

¹²⁹ <https://www.typescriptlang.org>

¹³⁰ <https://github.com/thejameskyle/babel-handbook/blob/master/translations/pt-BR/user-handbook.md>

¹³¹ <https://github.com/addyosmani/es6-tools#transpilers>

Se você ainda não usa as especificações mais modernas do JavaScript (ECMA Script 2015, ECMA Script 2016 ou ECMA Script 2017), a recomendação é usar as facilidades dessas linguagens aceleradoras para melhorar a manutenibilidade do seu código.

11.3 Linguagem ECMA Script (JavaScript 6, 7, 8, 9 e 10)

A partir da sua popularização nos últimos anos, o JavaScript evoluiu e em 2015 o ECMA Script 2015 foi lançado. Esse é o nome oficial da linguagem JavaScript 6. Inspirado no TypeScript, essa linguagem introduziu muitas novidades nessa linguagem, com foco na manutenibilidade e testabilidade. Uma listagem rápida das diferenças e funcionalidades pode ser encontrado aqui¹³². Constantes, definição de módulos, classes e funções geradoras, entre outros conceitos sintáticos, foram introduzidos nessa linguagem.

O efeito arquitetural do JS6 é melhorar a manutenibilidade de aplicações Web. Embora ele tenha suporte quase completo apenas em alguns navegadores como o Chrome ou Safari¹³³, ele pode ser transformado em código JavaScript 5 com o uso de transpiladores em bibliotecas como o Babel.

A linguagem JS melhora em base manual e hoje a versão 10 foi lançada em Junho de 2019. A sua documentação oficial pode ser encontrada aqui¹³⁴.

11.4 Bibliotecas de Componentes JavaScript

Arquitetos Web podem ajudar desenvolvedores nesse sentido, com a investigação, avaliação e uso de bibliotecas de componentes JavaScript de alta produtividade.

Exemplos de framework nesse sentido incluem o:

- jQuery-UI¹³⁵
- Kendo-UI.¹³⁶
- Zino-UI¹³⁷
- EasyUI
- Wijmo¹³⁹
- extJS¹⁴⁰
- D3¹⁴¹

A grande vantagem desse tipo de biblioteca é possibilidade de fazer códigos com interatividade rica com pouco código. Como exemplo, o exemplo de Grid¹⁴² do Kendo-UI disponibiliza em 50 linhas de código uma tabela com ligação remota de dados, paginação, ordenação, filtros dinâmicos nas colunas e funcionalidades de um CRUD completo.

11.5 Frameworks MVVM JavaScript

Aqui iremos falar de frameworks como Vue.JS, React ou Angular.

As arquiteturas clássicas Web, como ASP.NET, PHP ou JSF, fazem todo o controle da requisição e navegação e processamento da página dinâmica no servidor. Isso pode ser observado na Figura 32.

¹³² <http://es6-features.org>

¹³³ <https://kangax.github.io/compat-table/es6/>

¹³⁴ <https://www.ecma-international.org/ecma-262/10.0/index.html#Title>

¹³⁵ <http://jqueryui.com>

¹³⁶ <http://demos.telerik.com/kendo-ui/>

¹³⁷ <http://zinoui.com>

¹³⁸ <http://www.jeasyui.com>

¹³⁹ <http://wijmo.com>

¹⁴⁰ <https://www.sencha.com/products/extjs/>

¹⁴¹ <https://d3js.org>

¹⁴² <http://demos.telerik.com/kendo-ui/grid/editing>

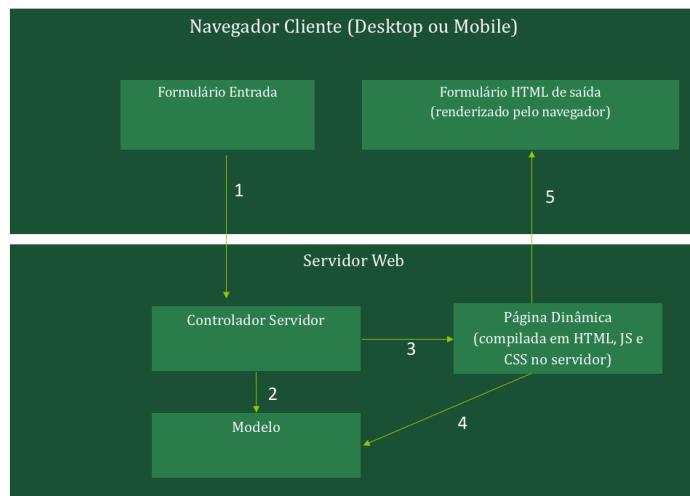


Figura 32: Modelo MVC Web clássico

Em linhas gerais no MVC tradicional Web, uma página origina a requisição HTTP (passo 1). Isso é processado em um controlador que fica no servidor em tecnologias como uma Servlet Java ou uma classe C#. Esse controlador faz acesso ao modelo de domínio para executar regras e acesso aos dados da aplicação (passo 2). O controlador faz então o despacho da requisição para uma página dinâmica que cuidará da resposta (passo 3), em tecnologias como JSF ou ASP.NET. Ao fazer isso, o controlador fornece acesso ao modelo para a página dinâmica (passo 4). A página dinâmica é compilada pelo servidor Web em HTML, JavaScript e CSS, que é então entregue para o navegador (passo 5).

Com o advento das páginas SPA (*Single Page Application*), um modelo alternativo de trabalho foi proposto conforme a Figura 33.

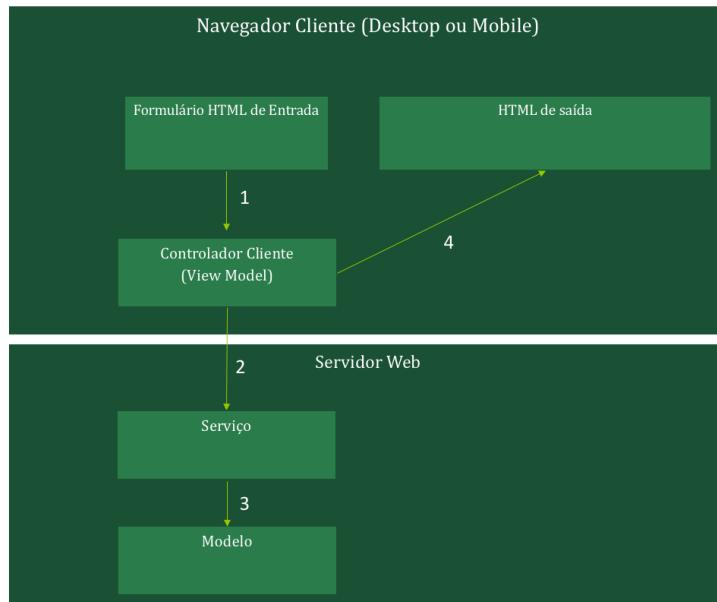


Figura 33: Modelo MV* (MVP/MVVM) Web

Nesse modelo, a requisição cliente ocorre ao um controlador Web (passo 1). De forma distinta do MVC clássico, esse controlador é escrito em JavaScript e, portanto, é executado no cliente. O controlador faz o acesso às regras e dados disponíveis no servidor (passo 2). Para isso, uma camada de serviços leves precisa ser desenvolvida em tecnologias como JAX-RS, ASP.NET Web API ou Node.js (passo 3). Com os dados que vieram do servidor em mãos o controlador passa esses dados para o formulário de saída. Note ainda que em aplicações SPA, por haver uma única página, o formulário de saída é o mesmo formulário de entrada.

Esse modelo foi popularizado pelo framework AngularJS e traz algumas vantagens sobre o MVC tradicional tal como:

- menor carga de trabalho no servidor Web, que se torna um servidor de serviços e não mais precise fazer toda compilação de páginas dinâmicas;
- mais performance, pois todas as requisições entre o a visão e o controlador ocorrem na camada cliente;
- menor tráfego de dados, visto que somente algumas porções da página são atualizadas com os dados retornados pelo modelo.

Nesse novo modelo o controlador também mantém os dados que serão manipulados pela visão e ganha um novo (*View Model* ou *Presenter*). No primeiro, existe uma ligação unidirecional da visão para o modelo. No segundo, existe uma ligação bidirecional entre a visão e o modelo. Ou seja, se o modelo for modificado, as visões associadas a ele são notificadas. Como exemplo, o sítio AngularHub¹⁴³ mantém online uma coleção de tutoriais do AngularJS e mostra essa dinâmica de interação de forma bastante didática.

A partir do AngularJS 1.0 e depois com o Angular e o React, outros frameworks com a mesma arquitetura Web surgiram e se popularizaram. A respeito desses frameworks, existem bons sítios que mantêm informações e comparativos como o disponibilizado aqui¹⁴⁴.

Em termos arquiteturais, o arquiteto Web deve avaliar com cautela o uso desses frameworks pois essa escolha implica na renúncia do uso de tecnologias como o JSF ou ASP.NET. Ao decidir pelo uso desses frameworks, o arquiteto deve investir no mentoreamento dos seus times em JavaScript e ferramentas de suporte.

11.6 Ferramentas de Suporte JavaScript

Com todo esse conjunto de técnicas e ferramentas JavaScript, é fácil se tornar desorganizado quando precisamos desenvolver aplicações em larga escala. No passado, desenvolvedores C, C++, Java e C# desenvolvedores ferramentas de suporte para o auxílio a construção e manutenção de códigos complexos. Essas incluem gerenciadores de dependências, montadores de executáveis, aceleradores de desempenho e documentação. Desenvolvedores JavaScript fizeram o mesmo nos últimos anos para profissionalizar a construção de código JavaScript em larga escala. Algumas dessas incluem:

11.6.1 Gerenciadores de dependências

Similar ao NuGet¹⁴⁵ para .NET ou o Maven para Java EE, a comunidade JavaScript desenvolveu ferramentas para gerir dependências. O Bower¹⁴⁶ é uma ferramenta bastante popular nesse sentido. O WebPack¹⁴⁷ e o npm¹⁴⁸ são também bastante utilizados. Nos últimos anos o Yarn¹⁴⁹ do Facebook também ganhou forte adesão do mercado. Uma outra biblioteca útil para o empacotamento de módulos e facilitar a gestão de dependências é o Browserify¹⁵⁰.

11.6.2 Executores de Tarefas

Similar ao Make¹⁵¹ para C, C++ e C# ou o Ant e Maven para Java EE, a comunidade JavaScript desenvolveu ferramentas como o Gulp¹⁵² e Grunt¹⁵³. Em linhas gerais, elas processam os arquivos de um sítio Web cliente e executam tarefas como minificação¹⁵⁴, compressão de imagens, análise do uso de boas

¹⁴³ <http://www.angularjshub.com/examples/>

¹⁴⁴ <http://todomvc.com>

¹⁴⁵ <https://www.nuget.org>

¹⁴⁶ <http://bower.io>

¹⁴⁷ <https://webpack.github.io>

¹⁴⁸ <https://www.npmjs.com>

¹⁴⁹ <https://yarnpkg.com/>

¹⁵⁰ <http://browserify.org>

¹⁵¹ <https://www.cs.umd.edu/class/fall2002/cmsc214/Tutorial/makefile.html>

¹⁵² <http://gulpjs.com>

¹⁵³ <http://gruntjs.com>

¹⁵⁴ Técnica de remoção de espaços em branco de arquivos HTML e JS e compactação de imagens que reduz o número de bytes transmitidos nas aplicações Web.

práticas de codificação, execução de testes de unidade automatizados, transpilação de JavaScript, distribuição e geração de aplicações prontas para homologação ou produção.

Um uso prático do Grunt, por exemplo, é permitir que desenvolvedores programem em JavaScript 6, que tem melhor manutenibilidade e produtividade, e executarem tarefas de pré-processamento com o uso de transpilador Babel para a tradução dos arquivos .JS em versão compatível com o JavaScript 5, garantindo assim portabilidade com navegadores mais antigos.

11.6.3 Automação de Testes com JavaScript

É papel de todo bom desenvolvedor escrever e manter testes automatizados que garantam o funcionamento do seu código. Com o aumento da codificação JavaScript em aplicações Web, é natural que frameworks de testes de unidade surjam também para JavaScript, similar ao JUnit¹⁵⁵, Visual Studio Unit Test¹⁵⁶ ou o NUnit¹⁵⁷. Algumas soluções nesse sentido incluem o QUnit¹⁵⁸ para testes de unidade e TDD, o Jasmine¹⁵⁹ para BDD (Behavior Driven Development) e Karma¹⁶⁰ para a execução de testes.

11.7 JavaScript Servidor e o Node.JS

Com o aumento da codificação JavaScript, alguns desenvolvedores foram ainda mais longe e propuseram o uso do JavaScript como linguagem servidora. Isso começou ainda em 2009 na conferência JSConf¹⁶¹, quando um desenvolvedor chamado Ryan Dahl apresentou um projeto em que estava trabalhando. Este projeto era uma plataforma que combinava a máquina virtual JavaScript V8 da Google e um laço de processamento de eventos. Esse projeto foi batizado de Node.js¹⁶².

Em linhas gerais, o Node.js é uma plataforma construída sobre o motor JavaScript do Google Chrome para construir aplicações de rede rápidas e escaláveis. O Node.js é um servidor de programas, ou seja, ele permite que escrevamos código JavaScript sobre o sistema operacional da máquina. O código da **Error! Reference source not found.** é construído sobre o Node.JS. Essa tecnologia usa uma abordagem minimalista, ou seja, ele vem pelado de fábrica. Ao mesmo tempo ele permite que centenas de módulos sejam instalados para adicionar novas funcionalidades. O npm (node package manager) é disponibilizado junto com o Node.js e faz esse papel.

Com o passar dos anos, a tecnologia Node.js ganhou popularidade e já rivaliza com tecnologias tradicionais com o Ruby, PHP, C# ou Java em ambientes mais dinâmicos como *startups*. Como o Node.JS não vem pronto para aplicações corporativas em larga escala, é importante conhecer alguns módulos usados em conjunto com o Node.JS. Alguns desses módulos incluem:

- ExpressJS¹⁶³ – Biblioteca para gerir requisições HTTP de forma facilitada a aplicativos móveis e Web.
- LoopBack^{164 165} - Framework para a montagem rápida de aplicações Node. Esse framework conta com suporte da IBM e StrongLoop e tem ferramentas visuais para a geração de modelos, APIs REST, scaffoldings e conectores com banco de dados.
- Restify¹⁶⁶ – Módulo para facilitar a criação de serviços REST.
- PassportJS¹⁶⁷ - Biblioteca para autenticação de usuários em aplicações Node. Conta já com mais de 300 estratégias de autenticação como por exemplo OpenID, OAuth1, OAuth2 ou SAML.
- Node-mysql¹⁶⁸ – Biblioteca para acesso a aplicações MySQL com Node.

¹⁵⁵ <http://junit.org>

¹⁵⁶ <https://www.visualstudio.com/en-us/docs/tfvc/create-and-run-unit-tests-vs>

¹⁵⁷ <http://www.nunit.org>

¹⁵⁸ <http://www.nunit.org>

¹⁵⁹ <http://jasmine.github.io/edge/introduction.html>

¹⁶⁰ <https://karma-runner.github.io/0.13/index.html>

¹⁶¹ <http://jsconf.com>

¹⁶² <https://nodejs.org/en/>

¹⁶³ <http://expressjs.com/pt-br/>

¹⁶⁴ <http://loopback.io>

¹⁶⁵ <http://loopback.io/resources/#compare>

¹⁶⁶ <http://restify.com>

¹⁶⁷ <http://passportjs.org>

¹⁶⁸ <https://www.npmjs.com/package/node-mysql>

- Node-OracleDB¹⁶⁹ – Driver mantido pela própria Oracle¹⁷⁰ para acesso aos bancos de dados Oracle em aplicações Node.
- Mssql¹⁷¹ – Driver para acesso a banco de dados SQL Server em aplicações Node.
- Mongoose¹⁷² – Framework para manipulação do banco de dados Non-SQL MongoDB.
- Azure¹⁷³ – Facilitador para implantação de aplicações Node em ambiente Microsoft Azure, mantido pela própria Microsoft.
- Mocha¹⁷⁴ – Biblioteca para testes de unidade para aplicações Node.
- Pm2¹⁷⁵ - Gerenciamento de processo de produção para aplicações Node, com suporte a clusters, balanceamento de carga e tolerância a falhas.
- NodeMon¹⁷⁶ – Biblioteca utilitária para facilitar a monitoração de mudanças no seu código fonte e recarregar a sua aplicação Web.
- Commander¹⁷⁷ – Módulo facilitador de execução de programas de linhas de comando em aplicações Node.
- Nodemailer¹⁷⁸ - Biblioteca para a manipulação de e-mails em aplicações Node.
- Request¹⁷⁹ – Biblioteca ainda mais simples que o Express para a manipulação de requisições HTTP.
- Hapi¹⁸⁰ - Um outro framework HTTP para desenvolvimento de aplicações Web em Node.
- Bluebird¹⁸¹ – Biblioteca para facilitar a escrita de programas concorrentes, com suporte à primitiva de programação *promise*¹⁸².
- Async¹⁸³ - Módulo utilitário para fornecer funções de manipulação de código assíncrono em JavaScript.
- Stomp-client¹⁸⁴ - Módulo utilitário para clientes do protocolo de fila de mensagens Stomp¹⁸⁵, suportado em implementações como o Apache ActiveMQ¹⁸⁶ e outros sistemas de filas de mensagem.
- Numerals.JS¹⁸⁷ - Biblioteca utilitária para manipular e formatar números.
- MomentJS¹⁸⁸ - Biblioteca utilitária para manipular datas.
- ShouldJS¹⁸⁹ – Biblioteca utilitária para auxiliar a escrita de testes BDD em JavaScript.
- Nock¹⁹⁰ - Biblioteca para criar mocks e simulações em testes de unidade JavaScript.

O portal do npmjs mantém uma lista¹⁹¹ de pacotes populares JavaScript e pode ser usado como fonte de referência para acompanhar novidades da comunidade. Importante citar que além do NodeJS existem também outras tecnologias servidoras JavaScript como por exemplo o Meteor¹⁹², potente framework para o desenvolvimento de aplicações JavaScript.

¹⁶⁹ <https://www.npmjs.com/package/oracledb>

¹⁷⁰ http://www.oracle.com/technetwork/database/database-technologies/scripting-languages/node_js/index.html

¹⁷¹ <https://www.npmjs.com/package/mssql>

¹⁷² <http://mongoosejs.com>

¹⁷³ <https://azure.microsoft.com/pr-br/develop/nodejs/>

¹⁷⁴ <http://mochajs.org>

¹⁷⁵ <http://pm2.keymetrics.io>

¹⁷⁶ <http://nodemon.io>

¹⁷⁷ <https://www.npmjs.com/package/commander>

¹⁷⁸ <https://www.npmjs.com/package/nodemailer>

¹⁷⁹ <https://www.npmjs.com/package/request>

¹⁸⁰ <http://hapijs.com>

¹⁸¹ <http://bluebirdjs.com/docs/why-promises.html>

¹⁸² https://en.wikipedia.org/wiki/Futures_and_promises

¹⁸³ <https://github.com/caolan/async/blob/v1.5.2/README.md>

¹⁸⁴ <https://www.npmjs.com/package/stomp-client>

¹⁸⁵ <http://stomp.github.io>

¹⁸⁶ <http://activemq.apache.org/stomp.html>

¹⁸⁷ <http://numeraljs.com/>

¹⁸⁸ <http://momentjs.com>

¹⁸⁹ <https://github.com/shouldjs/should.js>

¹⁹⁰ <https://github.com/node-nock/nock>

¹⁹¹ <https://www.npmjs.com/browse/star>

¹⁹² <https://www.meteor.com>

11.8 Mapeamento de Tecnologias JavaScript, Java EE Web e ASP.NET

Com tantos frameworks e bibliotecas, é fácil ficar confuso nesse mar de novas tecnologias. Para ajudar nesse processo,

Tabela 2 mostra um conjunto exemplo de tecnologias JavaScript que poderiam ser comparadas a equivalentes .NET e Java para táticas comuns em aplicações Web.

Tática Arquitetural	JavaScript	.NET	Java EE
Páginas Web de segunda geração	jQuery, extJS, Kendo-UI, Less, Bootstrap, MaterializeCSS	ASP.NET	JSF
Controladores MVC ou MMVM	Angular, React ou VueJS	Classes C#	Servlets
Motores de visualização	MustacheJS, UnderscoreJS, EmbeddedJS, HandleBarJS, Jade ¹⁹³	Razor ou WebForms	JSTL
APIs RESTful	Node Restify, Node Express LoopBack	ASP.NET Web API	JAX-RS
APIs WS-*	LoopBack	WCF	JAX-WS
APIs RPC	node gRPC	ASP.NET gRPC	Java gRPC
Mapeamento Objeto Relacional	LoopBack Meteor	Entity Framework	JPA
Acesso a banco de dados	Node node-oracledb, mssql, mongoose	ADO.NET	JDBC
Segurança	Node Passport	ASP.NET Identity	JAAS
Filas de Mensagens	stomp-client	MSMQ	JMS
Máquina Virtual	N/A	.NET Framework ou .NET Core	JVM
Contêineres e Orquestradores Comuns	Docker e Kubernetes	Docker e Kubernetes	Docker e Kubernetes

Tabela 7: Comparativo arquitetural entre Java Script, Java EE e .NET

11.9 Riscos e Oportunidades para o Arquiteto JavaScript

O massivo crescimento das tecnologias JavaScript traz riscos e também oportunidades para o arquiteto Web JavaScript. Esses riscos incluem:

- Grande número de bibliotecas e frameworks emergentes, que ainda disputam um lugar ao sol na comunidade Web.
- Nível de estabilidade incipiente de certas tecnologias JavaScript.
- Mão de obra de desenvolvimento no Brasil.

Arquitetos Web JavaScript podem endereçar esses riscos da seguinte forma:

¹⁹³ <http://www.creativebloq.com/web-design/template-engines-9134396>

- Experimentar as tecnologias. Fazer boas provas de conceito e testar as tecnologias em piloto antes de usá-las em projetos reais é importante. IDEs como o Orion Hub¹⁹⁴, por exemplo, permitem facilitar esse processo ao permitir que todo esse processo de codificação e testes nas nuvens, acelerando o processo de provas de conceito;
- Acompanhar como a comunidade avalia os frameworks JavaScript. Por exemplo, o sistema de reputação do GitHub chamado de GitHubStars¹⁹⁵ é bastante interessante. Quanto mais estrelas um projeto possui, maior aceitação ele tem da comunidade. Um exemplo desse sistema de reputação é mostrado na Figura 34.

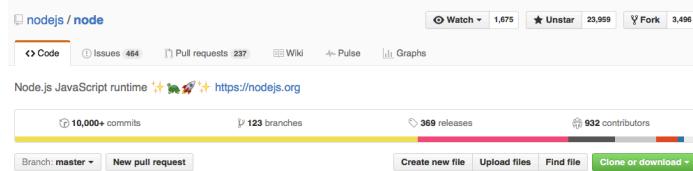


Figura 34: Sistema de reputação GitHub

11.10 Para Saber Mais

O fato de não haver uma pilha tecnológica única dominante para arquiteturas Web baseadas em JavaScript dificulta encontrar referências unificadas sobre o tema. Mas existem alguns livros que fornecem um tratamento mais em amplitude do que profundidade para JavaScript.

O livro de programação JavaScript de Eric Elliot (Elliot, 2014) é um bom ponto de partida para arquitetos Web. Já o livro de Stoyan Stefanov (Stefanov, 2010) dá um tratamento mais aprofundado do uso de padrões com essa linguagem.

¹⁹⁴ <https://orionhub.org/>

¹⁹⁵ <https://github.com/blog/1204-notifications-stars>

12 Documentação de Arquiteturas

Para projetos não triviais e empresas de médio e grande porte, a gestão do conhecimento técnico é fundamental. Um dos passos nesta direção é documentar, de forma leve e disciplinada, as decisões arquiteturais que foram discutidas nas fases iniciais de um determinado projeto. Ao longo deste capítulo, um exemplo é apresentado e os passos de sua organização e documentação são realizados.

O caso exemplo deste capítulo é inspirado em uma faculdade que possui um sistema acadêmico para os seus processos de negócio dos seus cursos de graduação. Esse sistema é baseado em tecnologia COBOL/CICS e usa banco de dados Oracle. Como consequência, o sistema é acessado pela equipe administrativa. Professores e alunos não tem acesso ao sistema e isso gera diversos problemas como sobrecarga de trabalho para a área administrativa e insatisfação para o corpo docente (professores) e corpo discente (alunos). Um projeto financiado pelo diretor de tecnologia tem como objetivo realizar uma modernização arquitetural nesse sistema, i.e., criar um portal Web de última geração para alunos e professores e permitir a migração gradativa de todo o código COBOL para uma nova plataforma ao longo de uma década. Uma reescrita completa do código antigo para uma nova plataforma seria caro e foi descartado.

Alguns desejos iniciais desse novo produto foram compilados em um pequeno texto e incluem:

- O novo sistema será implantado em módulos, com publicação semestral para manter o compasso de novas turmas.
- O sistema deve operar em vários navegadores e em telefones celulares dos alunos e professores.
- O desempenho e a escalabilidade são preocupações, devido aos períodos de pico de atividades como lançamento de notas, matrícula ou consultas de notas no fim do semestre.
- O sistema deve apresentar manutenção e testes facilitados pois a diretoria técnica quer ter o menor custo de propriedade com o produto após a sua operação.
- O sistema deve se comunicar com o antigo sistema já desenvolvido com tecnologia COBOL/CICS.
- O sistema deve operar em qualquer período do dia e da noite.
- Sistemas acadêmicos mantêm dados sensíveis e, portanto, a segurança é um aspecto crítico.

A partir do caso exemplo, iremos desenvolver uma documentação arquitetural que registre as decisões técnicas e o racional arquitetural.

12.1 Passo 1 - Visualização de Negócio

Recomendação Arquitetural: Comece o seu trabalho arquitetural com um desenho de alto nível, não técnico. Esse desenho é chamado de “marketecture” ou “visualização de negócio”. O objetivo desse desenho é facilitar o seu discurso e ancorar a sua comunicação com gerentes, clientes, analistas de teste e desenvolvedores. Não usamos a UML ou outras linguagens técnicas nesse momento. Estamos mais preocupados aqui com comunicação visual que precisão técnica.

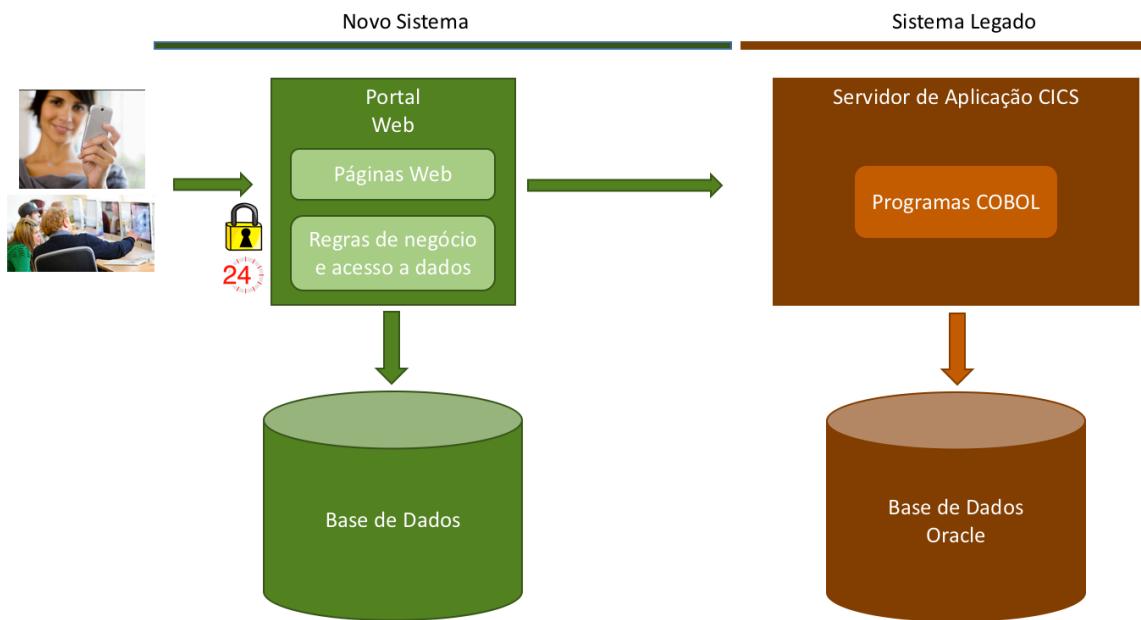


Figura 35: Visualização de Negócio do Sistema Acadêmico

Alguns pontos importantes que devem ser destacados em uma boa visualização de negócio incluem:

- Demarcar o que será construído (indicado em verde no desenho) e o que será integrado (indicando em marrom);
- Usar formas simples e humanizadas, como por exemplo os desenhos dos bancos de dados, o cadeado para indicar segurança da informação, o símbolo do relógio 24 horas para indicar alta disponibilidade e as fotos com alunos com computadores e celulares;
- Não introduzir tecnologias e decisões tecnológicas que ainda devem ser discutidas e deliberadas. Veja que o COBOL e Oracle aqui aparecem por serem informações do sistema já existente.

12.2 Passo 2 – Condutores Arquiteturais

O próximo passo é capturar e descrever os condutores arquiteturais, ou seja, qual a agenda técnica que deve guiar o trabalho do time de arquitetura. No contexto apresentado alguns condutores podem incluir:

12.2.1 Acessibilidade e Usabilidade

Racional: A acessibilidade é o atributo de qualidade que possibilita que um sistema seja universalizado e acessado pelo máximo de pessoas no maior número possível de dispositivos. Além disso, alunos entre 17 e 30 anos (gerações Y e Z) já se acostumaram a usar sistemas “bonitos” e com isso a usabilidade também se torna uma preocupação técnica.

12.2.2 Interoperabilidade

Racional: Durante toda uma década o sistema na nova tecnologia e o sistema legado irão conviver e, portanto, a interoperabilidade é um aspecto crítico. Além disso, o time de arquitetura também precisa interoperar com outros sistemas, como por exemplo:

- O sistema do Ministério da Educação que realiza o censo acadêmico;
- O sistema de segurança corporativa da faculdade baseado em Microsoft Active Directory;
- O sistema ERP que cuida do processamento da folha de pagamento e geração de boletos para os alunos.

12.2.3 Segurança

Racional: Sistemas acadêmicos são desafiados por *hackers*. Além disso, existe um conjunto de dados sigilosos como informações de milhares de alunos que irão circular em ambiente Web e dados como notas, faltas e comunicação entre professores. A agenda de autenticação, autorização, auditoria e transporte seguro ganha importância e deve ser trabalhada pelo time de arquitetura.

12.2.4 Escalabilidade

Racional: Faculdades possuem um calendário acadêmico rígido e os sistemas devem respeitar essa questão. Por exemplo, os períodos de lançamento de notas, consultas de notas e matrículas em disciplinas são críticos e geram sobrecargas imensas aos servidores. Dessa forma, a escalabilidade se torna um aspecto crítico e traz uma agenda de preocupação técnica para o time de arquitetura.

12.2.5 Manutenibilidade

Racional: O diretor da faculdade quer minimizar os custos de operação do produto. Além disso, sistemas acadêmicos tem regras de negócio bastante complicadas que requerem uma gestão bastante cuidadosa.

12.3 Passo 3 – Estilos Arquiteturais Web

No Capítulo 4 apresentamos os principais estilos arquiteturais Web (Web 1.0, Web 2.0, SPA, API Web e Micro Serviços Web). Cada estilo traz implicações distintas e é mais apropriado para um certo contexto. No exemplo trabalhado nesse capítulo, como existe uma necessidade de modernização tecnológica e desafios intensos de acessibilidade, o estilo Web 1.0 seria já descartado. O estilo Web 2.0 é um candidato natural, sendo que até um estilo SPA poderia ser usado. O uso de API Web pode ser usado para a exposição dos serviços legados, mas devido à arquitetura monolítica do CICS o uso de micro serviços para essa API não é viável.

Em resumo, uma recomendação do uso de Web 2.0 para o portal e uso de uma API Web REST para os serviços legados COBOL pode ser uma alternativa viável para o nosso problema.

12.4 Passo 4 – Escolha da Plataforma Tecnológica

Com os condutores e estilo arquitetural Web definidos, o arquiteto e seu time devem definir a plataforma a ser usada. Uma lista de opções de plataformas Web inclui:

- Java EE Web
- ASP.NET
- Centrada em JavaScript
- LAMP (PHP, Python ou Ruby)
- Híbridas (JavaScript MVVM + Java/ASP.NET/LAMP)

Os capítulos 6, 7 e 8 apresentaram em detalhes três das plataformas mais comuns para trabalharmos com arquiteturas na Web. No mundo real, recomendamos que o arquiteto não tome uma decisão baseada apenas na sua preferência pessoal. Ao invés, ele deve pesar os seguintes aspectos:

- Habilidade do time;
- Produtividade na tecnologia pelo time;
- Restrições para a organização alvo;
- Facilidade de encontrarmos profissionais com conhecimento da tecnologia para manter o produto;
- Custo de profissionais;
- Maturidade das tecnologias;
- Risco de descontinuidade e obsolescência das tecnologias.

Recomendação Arquitetural: Sempre use critérios racionais para selecionar a sua plataforma e documente essa decisão. Como essa é a decisão mais crítica na sua escolha arquitetural, você poderá ser cobrado por ela daqui a 2, 5 ou 10 anos. Explique porque você tomou essa decisão.

No exemplo aqui trabalhado, não temos essas variáveis contextuais para fazer uma escolha e iremos apresentar as soluções técnicas nas três plataformas apresentadas: Java EE Web, ASP.NET e JavaScript.

12.5 Passo 5 – Visualização Lógica

Após capturar os condutores, o arquiteto pode começar a expressar a sua solução em desenhos técnicos. Para isso ele pode usar desenhos UML e um primeiro diagrama a ser produzido é o de pacotes. Ele fornece uma visão de pássaro da arquitetura, mas permite que os módulos centrais do produto sejam especificados. O diagrama da

Figura 36 apresenta essa visualização.

Note que esse diagrama é representado em uma linguagem formal e os módulos que irão capturar as principais preocupações arquiteturais estão aqui desenhados.

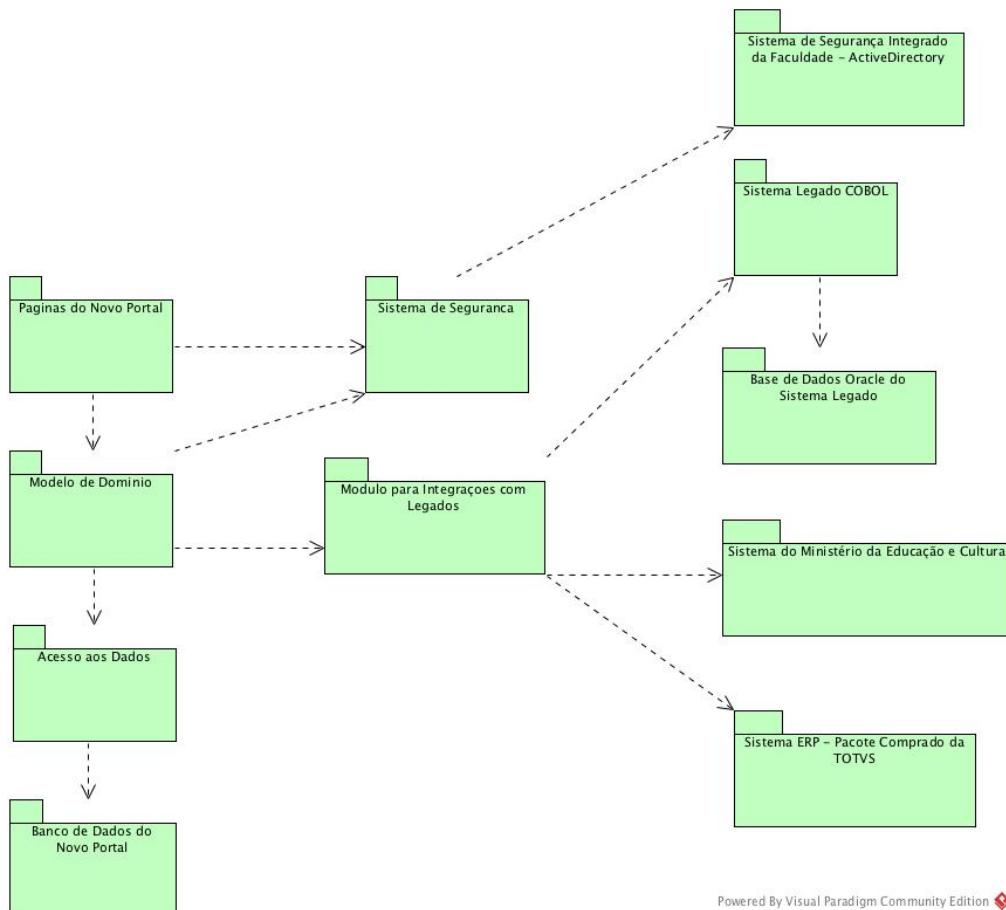


Figura 36: Visualização Lógica do Sistema Acadêmico

Algumas regras para um bom diagrama de pacotes incluem:

- Buscar a visão de amplitude;
- Mostrar as conexões entre os módulos arquiteturais;
- Não introduzir tecnologias;

Alguns arquitetos, para buscar uma melhor comunicação de negócio, também representam os módulos de negócio em um pacote especial chamado de modelo de domínio. Esse pacote pode ser refinado conforme mostrado a Figura 37. Em termos técnicos, cada pacote aqui mostrado se materializará como *package* em Java, um *namespace* em C# ou um *module* em JavaScript.

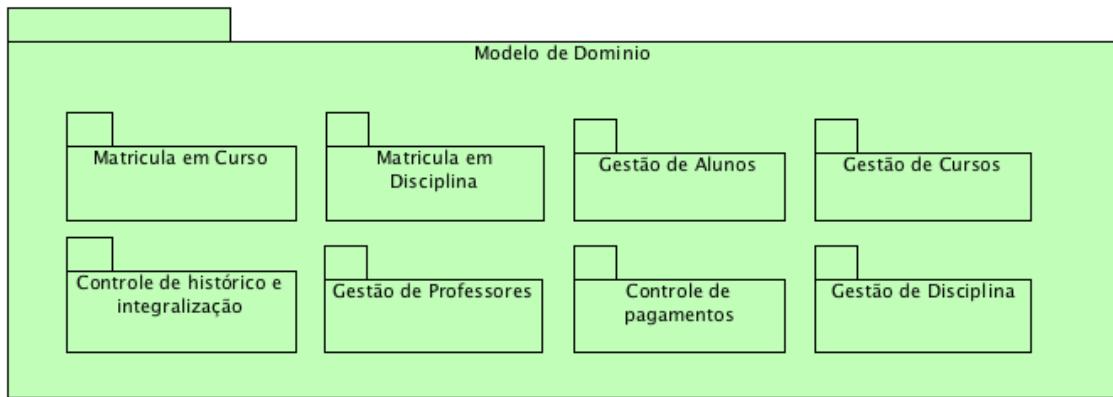


Figura 37: Modelo de Domínio do Sistema Acadêmico

12.6 Passo 6 – Visualização da Topologia Física

Tendo a visualização lógica em mãos, o arquiteto deve agora estruturar como a sua arquitetura será representada em termos físicos. A isso chamamos de topologia, ou representação da estrutura física de máquinas servidoras que irá hospedar a nossa aplicação.

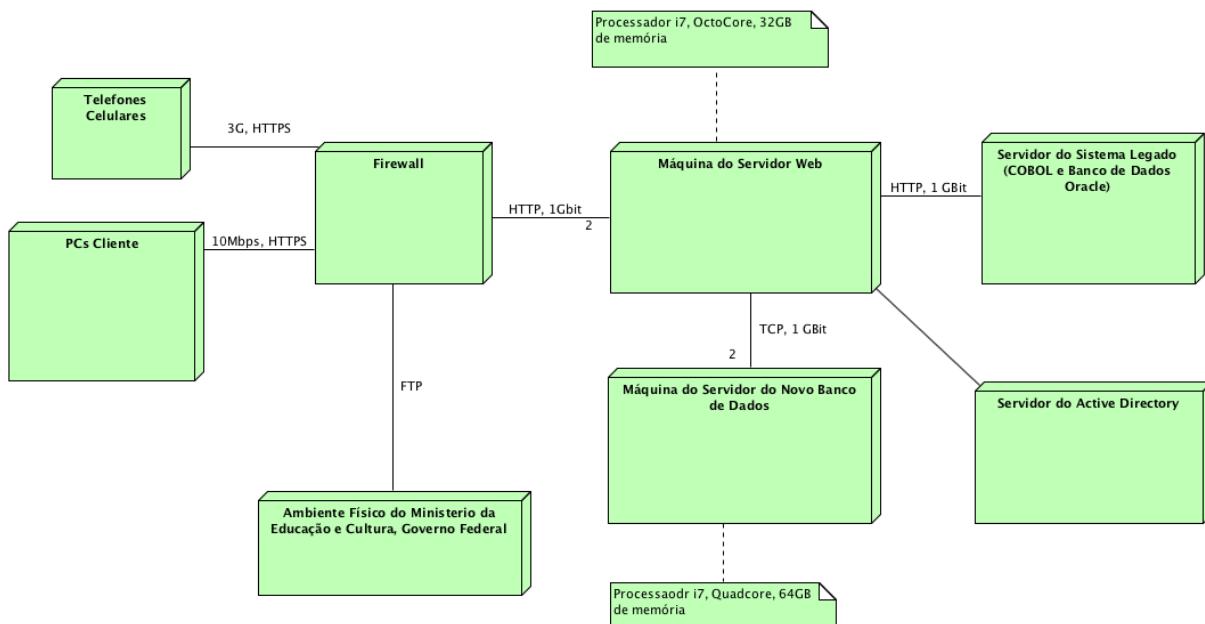


Figura 38: Topologia do Sistema Acadêmico

Neste diagrama, podemos representar:

- As máquinas que irão fazer parte da topologia, sejam elas clientes (ex. Telefones celulares) ou do ambiente da universidade (máquina do servidor Web);
- Protocolos de rede que a aplicação requer (ex. HTTPS, HTTP ou TCP);
- Bandas mínimas de passagem (ex. 3G ou 10Mbps);
- Clusters (note o número 2 na figura, indicado ao lado na máquina do servidor Web e do banco de dados);
- Máquinas externas à sua rede (ex. servidor do MEC);
- Máquinas já existentes (ex. Máquina do ActiveDirectory ou o servidor do sistema legado).

12.7 Passo 7 – Visualização da Persistência de Dados

Como a persistência é um aspecto crítico em sistemas Web, é recomendado que o arquiteto indique como

essa parte será resolvida pela arquitetura. Para isso, iremos precisar de um detalhamento técnico com o uso do diagrama de componentes UML.

Nesse passo, vamos assumir que a universidade já possui uma diretriz da arquitetura corporativa do uso de Oracle 12c e, portanto, o novo sistema de banco de dados também será Oracle. Quando esse cenário surge, isso se chama uma restrição arquitetural.

Como a persistência pode ser realizada de várias plataformas distintas, vamos fazer desenhos com o uso da plataforma Java EE, .NET e Node.JS. Os desenhos estão na Figura 39, Figura 40 e Figura 41. Cada caixa nesse diagrama representa um componente arquitetural, que em termos práticos é uma DLL, assembly, .JAR, biblioteca ou um módulo JavaScript.

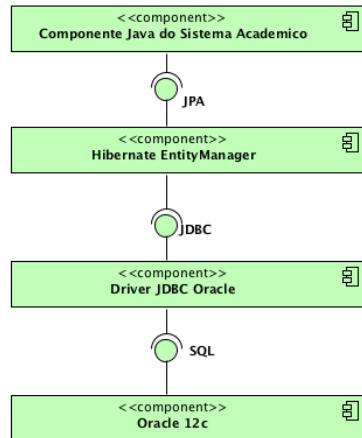


Figura 39: Visualização de Persistência em Java EE

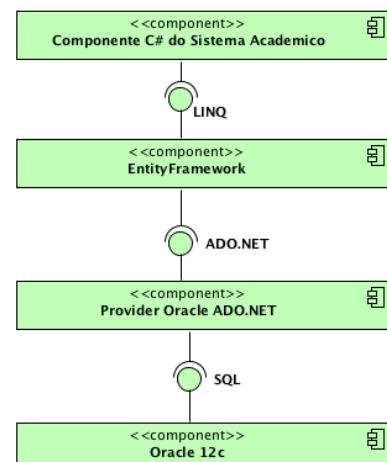


Figura 40: Visualização de Persistência em .NET

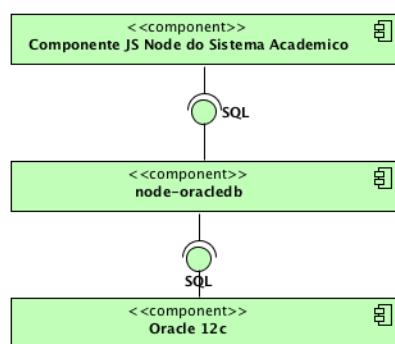


Figura 41: Visualização de Persistência em Node.js

Uma boa visualização de persistência deve capturar o seguinte:

- Banco de dados que serão usados (ex. Oracle 12c);

- Drivers de acesso a banco de dados (ex. node-oracledb ou Oracle JDBC Driver);
- Presença ou não de frameworks de mapeamento objeto relacional (ex. Entity Framework).

Quando necessário, especifique já até a versão dos drivers e frameworks usados e sítios de acesso para download. Por exemplo, em Junho 2016 uma especificação possível da Figura 39 poderia indicar os componentes banco de dados versão Oracle 12.1.0.2.0, driver JDBC versão ojdbc7¹⁹⁶ e versão do Hibernate ORM 5.2.0¹⁹⁷. É sempre válido testar se essas combinações interoperam com um teste rápido ou uma prova de conceito mais elaborada se o risco é maior.

12.8 Passo 8 – Visualização da Apresentação (Usabilidade e Acessibilidade)

Agora o arquiteto deve representar que bibliotecas e frameworks para a camada de apresentação. As figuras seguintes apresentam possibilidades para Java EE, .NET e AngularJS/Node.JS.

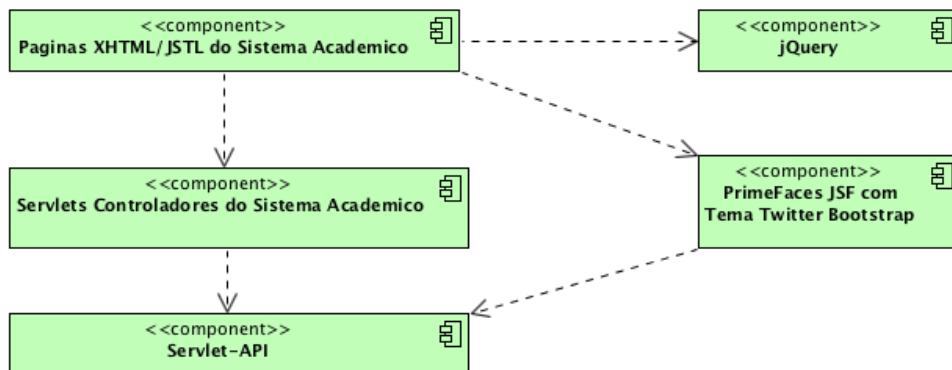


Figura 42: Visualização de Apresentação em Java EE

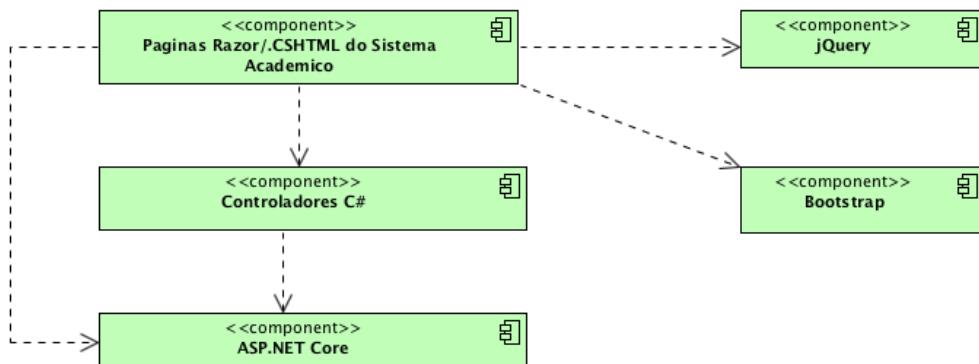


Figura 43: Visualização de Apresentação em ASP.NET Core

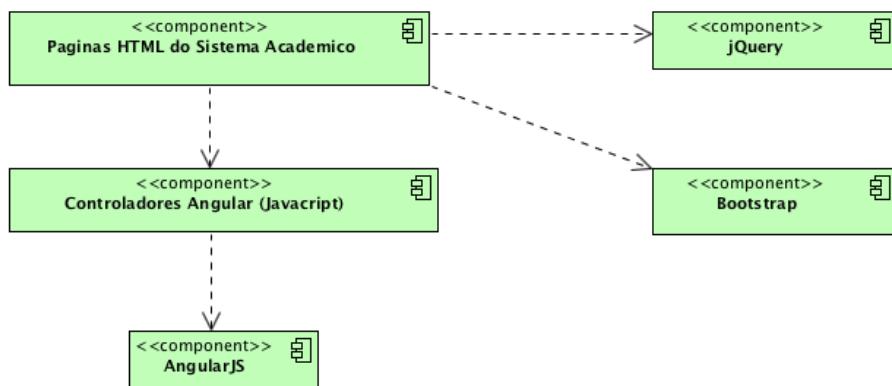


Figura 44: Visualização de Apresentação em AngularJS/Node.JS

¹⁹⁶ <http://www.oracle.com/technetwork/database/features/jdbc/jdbc-drivers-12c-download-1958347.html>

¹⁹⁷ <http://hibernate.org/orm/downloads/>

Uma boa visualização de apresentação deve indicar:

- Frameworks a serem usados e se necessário a versão de cada framework; (ex. Angular ou ASP.NET Core)
- Componentes da aplicação (ex. Páginas e controladores).

Como opção, arquitetos podem comunicar isso de forma alternativa à UML com o desenho da sua estrutura de projetos. Um exemplo nesse sentido é fornecido para um projeto ASP.NET Core com Bootstrap e jQuery.

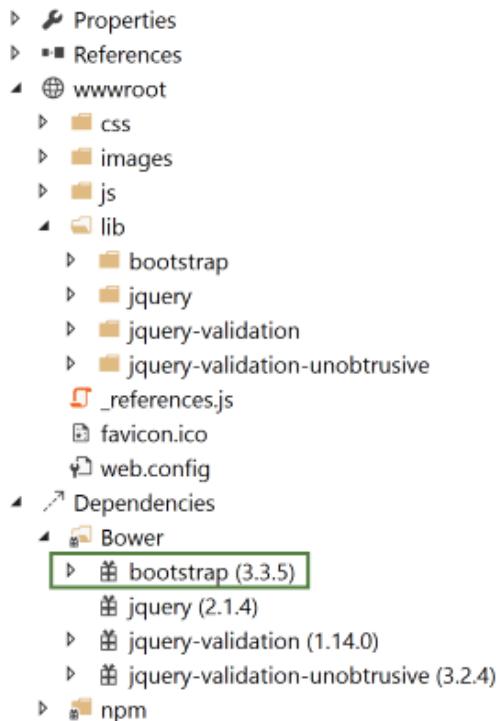


Figura 45: Visualização de Componentes de Apresentação em ASP.NET Core

Observe que a escolha de Bootstrap e jQuery foi arbitrária e usada como exemplo apenas. No mundo real, o arquiteto deve reservar tempo para buscar, comparar e selecionar as bibliotecas mais apropriadas para o seu contexto.

12.9 Passo 9 – Visualização de Segurança

O arquiteto deve também investir tempo para representar como a segurança será resolvida em sistemas Web. No exemplo sendo aqui trabalhado, em particular, a segurança foi colocada como um aspecto crítico e usaremos como premissa que a autenticação de usuários deve ser realizada sob controle de um ambiente controlado da faculdade, que é um servidor LDAP da Microsoft, o Active Directory.

Algumas soluções para Java EE, ASP.NET e Node.JS são mostradas nas figuras a seguir.

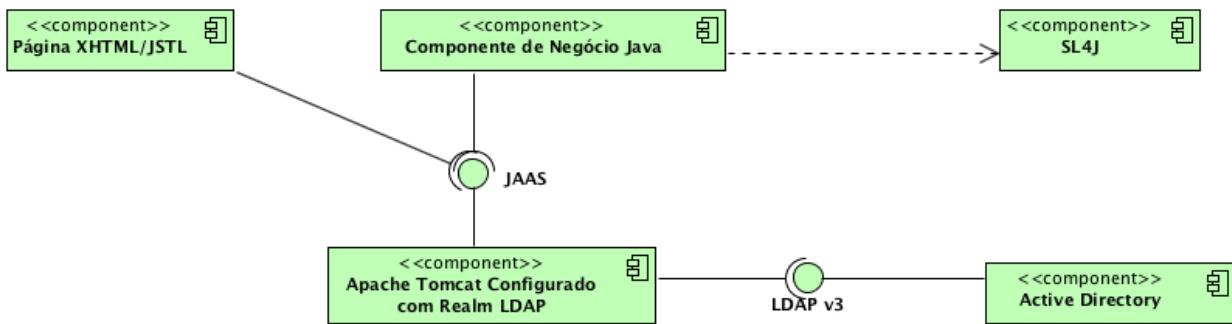


Figura 46: Visualização de Segurança em Java EE

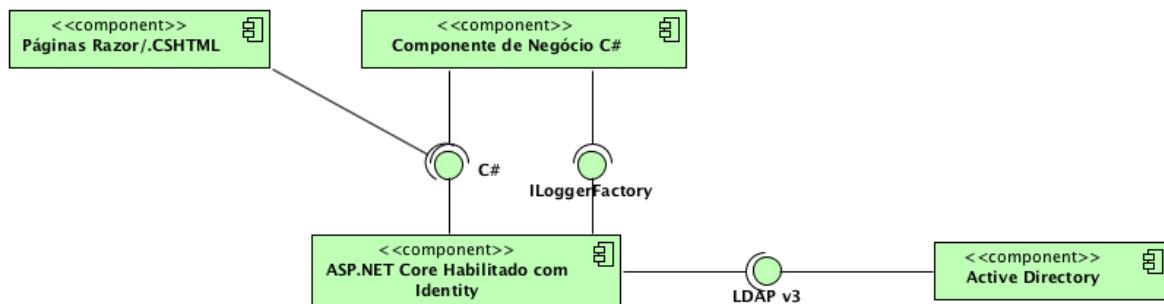


Figura 47: Visualização de Segurança em .NET

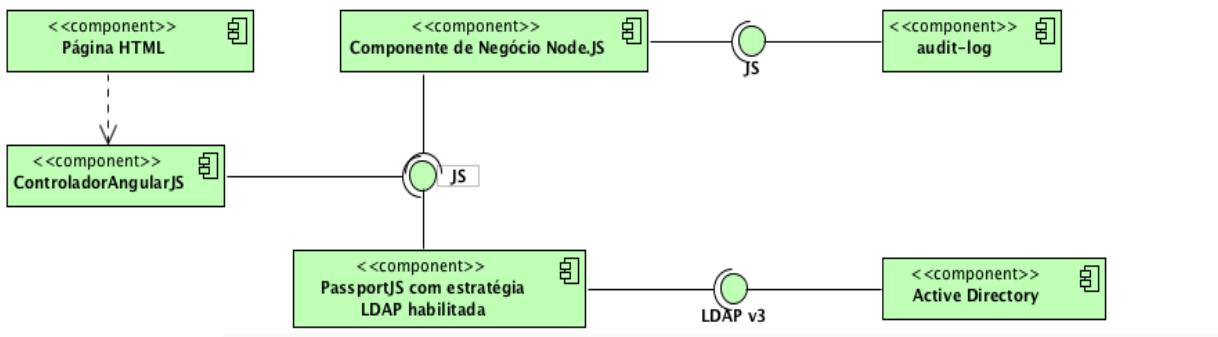


Figura 48: Visualização de Segurança em Node.js

Alguns pontos importantes que o arquiteto deve considerar nessa visualização incluem:

- Componentes extras a serem usados (ex. No exemplo do Node.js temos os componentes passport e audit-log);
- Configurações a serem realizadas (ex. ASP.NET Core habilitado como Identity ou Apache Tomcat configurado com LDAP)
- Protocolos relevantes (ex. LDAP ou JAAS)

Como aspectos de segurança podem envolver configurações específicas, é bom que a documentação arquitetural indique sítios auxiliares que contenham exemplos de como fazer essas configurações. Exemplos:

- Configuração do LDAP com AD no ApacheTomcat^{198 199}.
- Configuração e uso do Node Passport com AD²⁰⁰

¹⁹⁸ <https://tomcat.apache.org/tomcat-7.0-doc/realm-howto.html#JNDIRealm>

¹⁹⁹ <http://stackoverflow.com/questions/267869/configuring-tomcat-to-authenticate-using-windows-active-directory>

²⁰⁰ <https://www.npmjs.com/package/passport-ldapauth>

12.10 Passo 10 – Visualização de Interoperabilidade

Em sistemas não triviais ou em grandes empresas, a integração é sempre um ponto de atenção no desenho de arquitetural. Neste aspecto, é importante primeiro conhecermos os estilos mais comuns de integração antes de discutirmos opções tecnológicas nas plataformas Java, .NET ou Node.js. A Figura 49 apresenta estes estilos.



Figura 49: Estilos de integração

Embora o assunto de integração seja extenso e complexo, podemos dizer que:

- A troca de arquivos tende a ser a solução técnica mais simples, mas em sistemas complexos ela sofre de problemas de falta de controle transacional, falhas humanas e ambientes não confiáveis.
- Bancos de dados compartilhados tendem a ser simples também, mas sem uma governança adequada o seu uso indiscriminado pode trazer efeitos ruins com mudanças em uma aplicação gerando efeitos colaterais indesejados em dezenas de outras aplicações.
- Chamada de procedimentos remotos e filas de mensagens são mais complexos de serem entendidos pelo time, mas fornecem escala corporativa mais robusta. Ao mesmo tempo, também requerem governança técnica.
- Soluções de filas de mensagens são recomendadas quando buscamos tolerância a falhas e escalabilidade com baixo custo de hardware.

Vamos assumir no nosso cenário que o time que mantém o ambiente legado CICS/COBOL já conheça como expor programas COBOL através de uma API REST²⁰¹. Com essa premissa, podemos escolher a chamada de procedimentos remotos como paradigma de integração. Isso levaria à seguinte solução genérica, que envolve que o time COBOL crie uma fachada de serviços de negócio que será chamada pelos componentes de negócio do novo portal Web.

Vamos assumir também que o time do ministério da educação em Brasília mantém um servidor FTP seguro para receber os arquivos solicitados da faculdade, que é chamado de censo acadêmico. E vamos assumir também que o fornecedor do ERP interopere através de arquivos colocados em diretório e isso seja uma restrição pois não pode ser modificado em um grande custo como o fornecedor.

²⁰¹ https://www.ibm.com/developerworks/community/blogs/e4210f90-a515-41c9-a487-8fc7d79d7f61/entry/cics_atom_support_restful_service

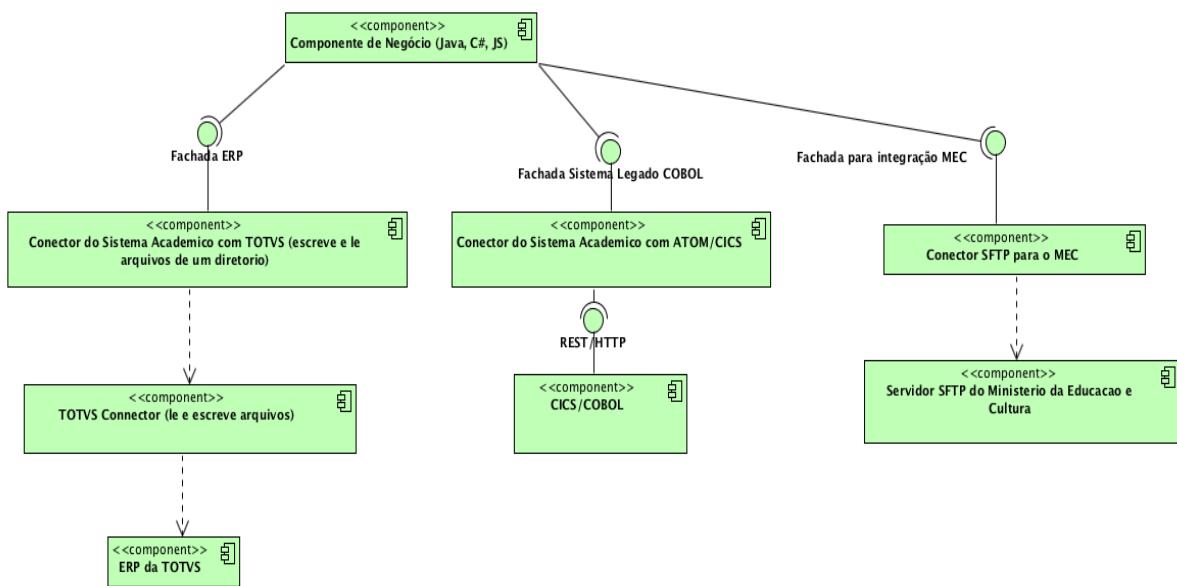


Figura 50: Visualização de Integração

Na solução apresentada, temos um conector (código Java, C# ou JS) específico por tecnologia. O primeiro conector usa as APIs de leitura e escrita de arquivos para fazer a comunicação. O segundo conector faz a chamada aos serviços REST/HTTP usando as APIs de programação dessas linguagens. O terceiro e último conector faz chamada via FTP ao servidor SFTP do governo federal.

12.11 Passo 11 – Visualização de Manutenibilidade

Dado que no nosso contexto a manutenibilidade e a testabilidade foram citados como aspectos críticos, queremos também demonstrar como isso poderá ser resolvido em cada plataforma. Isso leva a diagramas distintos nas várias tecnologias, conforme mostrado nas figuras.

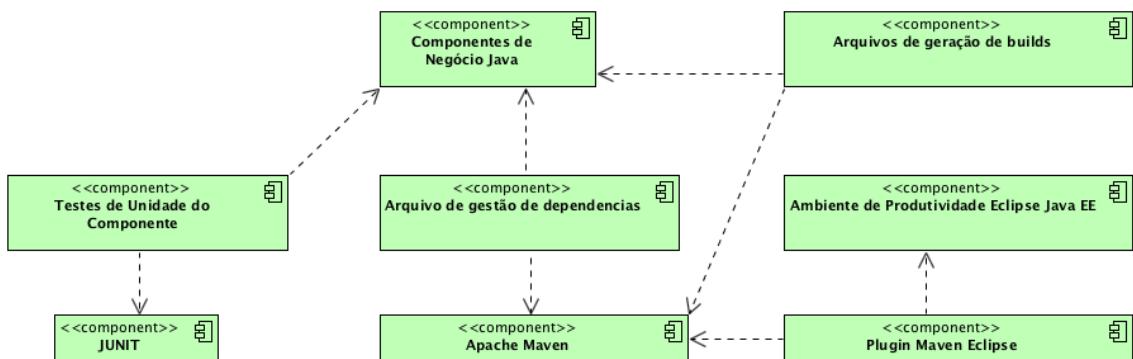


Figura 51: Visualização de Manutenibilidade em Java EE

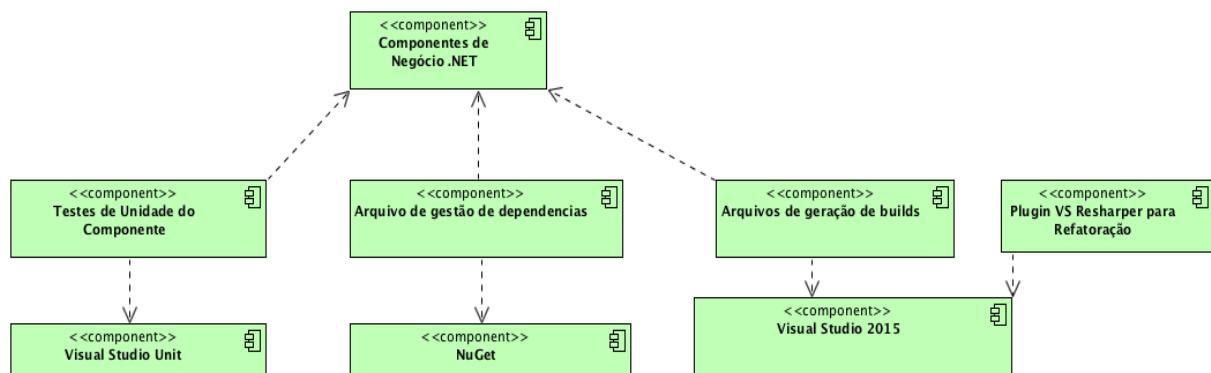
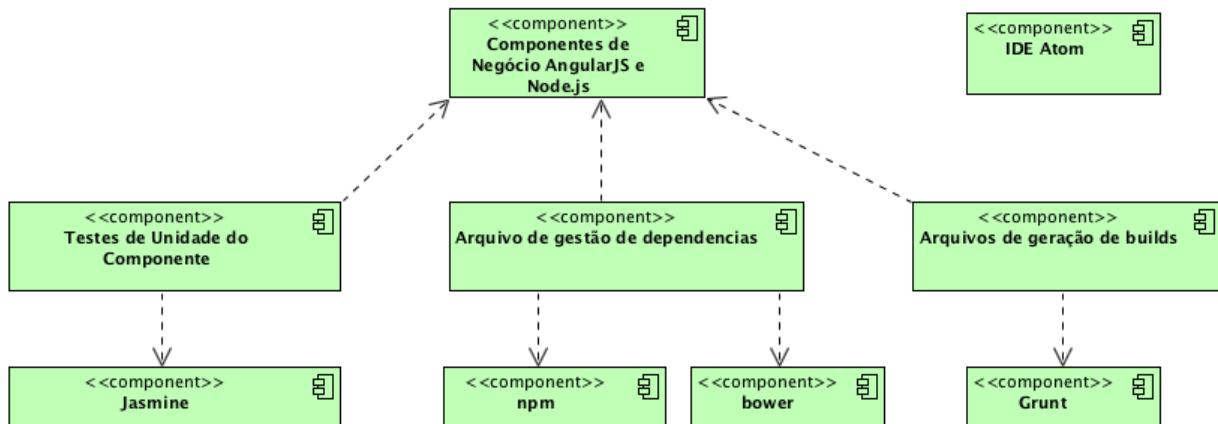
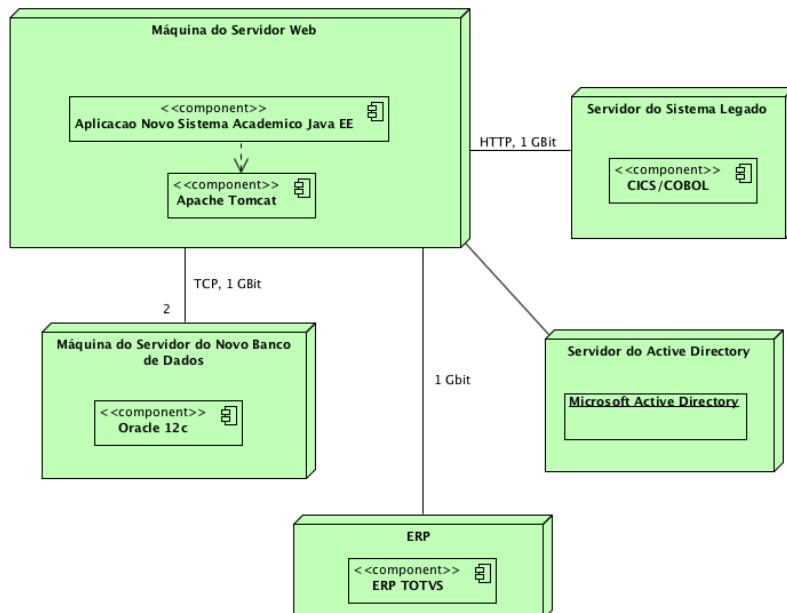


Figura 52: Visualização de Manutenibilidade em ASP.NET**Figura 53:** Visualização de Manutenibilidade em Node.js

Uma boa visão de manutenibilidade deve indicar que componentes de infraestrutura o arquiteto e seu time irá usar para apoiar os seus processos de manter e testar o seu código. Esses componentes não importam para o cliente, mas são críticos para o time de desenvolvimento em termos do seu processo técnico de desenvolvimento.

12.12 Passo 12 – Alocação de Componentes aos Nodos Físicos

O diagrama da Figura 38 mostra a topologia física. É possível também estendê-lo e mostrar como alguns componentes de software podem ser alocados a cada um desses hardwares. Embora isso seja mais útil em topologias bem mais complexas do que a trabalhada nesse exemplo, um exemplo é mostrado aqui para referência para o leitor.

**Figura 54:** Topologia Física com Componentes – Visão Java EE

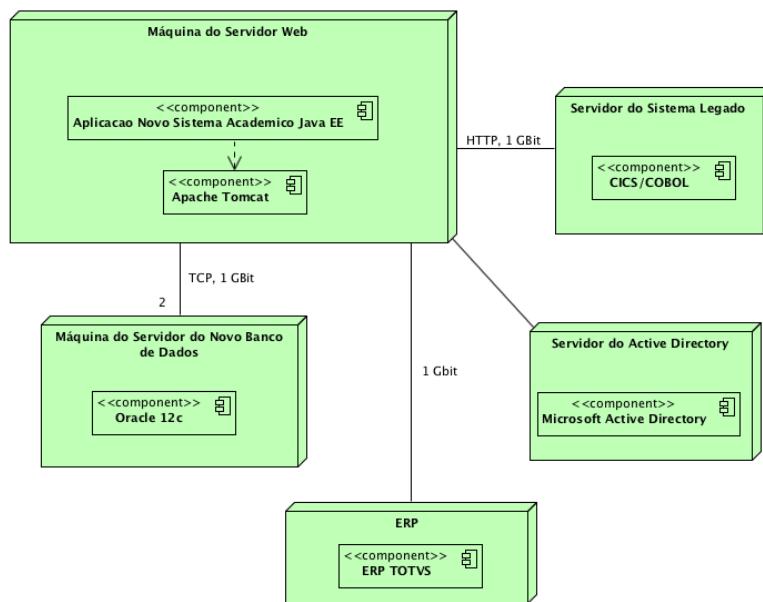


Figura 55: Topologia Física com Componentes – Visão .NET

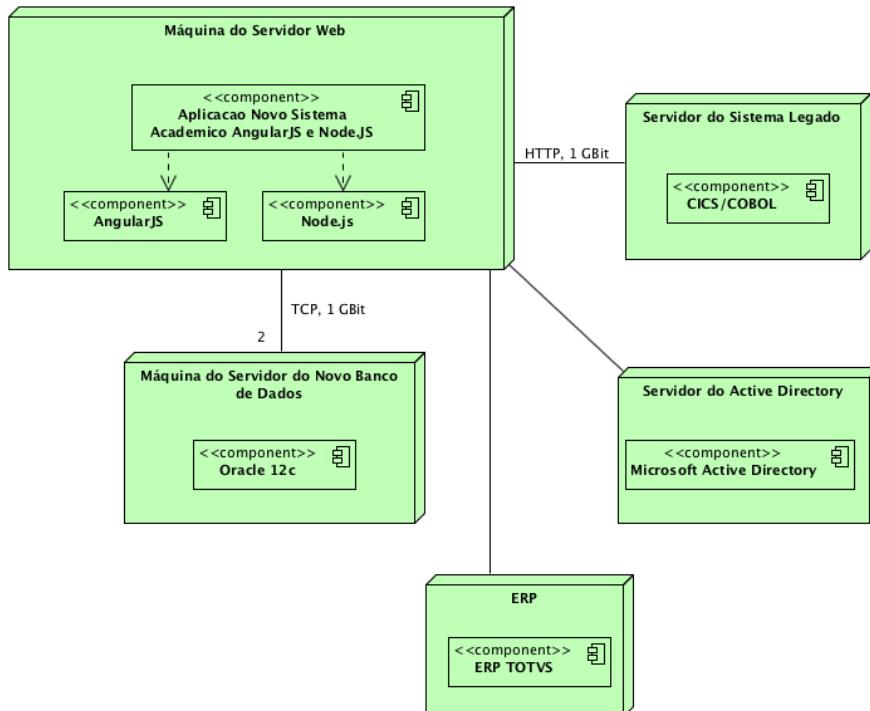


Figura 56: Topologia Física com Componentes – Visão Angular e Node.js

Recomendação Arquitetural: Utilize o conceito de pontos de vista para fazer múltiplos modelos sobre a sua arquitetura. Como pode ser observado nos passos 3-10, cada preocupação relevante do sistema acadêmico foi representada a partir de um modelo distinto. O uso de vários modelos, cada um dele pequeno e de fácil comunicação, é uma melhor prática arquitetural. Não exagere. Produza apenas documentação que tenha propósito e seja esperada pelo seu time.

12.13 Passo 13 – Determinar Provas de Conceito

Quando a arquitetura traz riscos técnicos para o time, é prudente que o arquiteto organize provas de conceito. Quais as provas de conceito vão ser realizadas é uma informação contextual pois envolve a criticidade da tecnologia para o produto, conhecimento do time e momento do projeto.

Vamos assumir, por exemplo, que a plataforma do sistema acadêmico seja desenvolvida em Node.js e a equipe ainda não tenha experiência com o uso do modulo Passport com o servidor Microsoft AD. Nesse caso, podemos ter uma prova de conceito cujo objetivo é fazer um código mínimo em Node.js que realize a autenticação de usuários com o uso do Microsoft AD, conforme a estrutura da árvore LDAP já utilizada pela equipe de segurança da informação da faculdade.

12.14 Modelos de um Documento de Arquitetura de Software

Os processos de software como o RUP²⁰², OpenUP²⁰³, SEI V&B (Views and Beyond)²⁰⁴ trazem já bons modelos para a documentação de arquitetura de software.

Inspirados nesses e outros processos, uma sugestão de um documento de arquitetura de software leve e centrado em modelos visuais para sistemas Web poderia ter as seguintes seções.

- Visualização de Negócio
- Condutores Arquiteturais
- Estilo Arquitetural
- Plataforma Tecnológica
- Visualização Lógica
- Visualização Física
- Visualização de Persistência
- Visualização de Apresentação (Usabilidade e Acessibilidade)
- Visualização de Segurança
- Visualização de Interoperabilidade
- Visualização de Manutenibilidade
- Visualização dos Componentes Alocados aos Nodos Físicos
- Organização das Provas de Conceito

Ao mesmo tempo, se você e o seu time trabalham com métodos ágeis e não possuem necessidades formais de gestão documental, use quadro brancos e canetas para fazer os seus desenhos e gerar bons debates. Depois tire fotos com o seu telefone celular para capturar e disseminar o conhecimento.

²⁰² https://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf

²⁰³ <http://epf.eclipse.org/wikis/openuppt/>

²⁰⁴ <http://www.sei.cmu.edu/architecture/tools/document/viewsandbeyond.cfm>

13 Aceleração de Arquiteturas com Práticas DevOps

A cultura DevOps tem se estabelecido nos últimos anos para transformar a forma como desenvolvedores e times de operações desenvolvem e mantêm suas aplicações. Esta cultura é um forte aliado das práticas arquiteturais, pois busca conectar times de desenvolvimento, qualidade e operações com o uso de práticas e aceleradores de automação de ciclo de vida e monitoração de aplicações.

13.1 DevOps para Aceleração da Entrega de Produtos

Vimos no capítulo 3 que uma boa arquitetura deve endereçar atributos de qualidade que são críticos para o negócio. Nos tempos atuais é crítico reduzir o tempo de ciclo no desenvolvimento de aplicações, o tempo para recuperação de incidentes em produção e o esforço gasto em defeitos e retrabalhos. Em linguagem arquitetural, estas organizações estão buscando trabalhar condutores arquiteturais tais como manutenibilidade, testabilidade, implantabilidade, configurabilidade e recuperabilidade.

Em muitas empresas existem muitas dificuldades para que uma aplicação seja operacionalizada com rapidez e estabilidade para ambientes de produção. Nestas empresas, os times de desenvolvimento, qualidade e operações estão dispostos como silos e não mantém uma comunicação frequente e eficaz. Também vemos nestas empresas ciclos longos de entrega (trimestres, semestres ou até anos) e um alto índice de retrabalho em seus produtos.

Nos últimos anos, uma cultura de desenvolvimento que envolve pessoas, práticas e produtos emergiu sob o termo **DevOps**. Esta cultura de desenvolvimento tem suas raízes nos princípios Lean, práticas ágeis de desenvolvimento, processos ágeis de desenvolvimento com o XP (*Extreme Programming*) e melhores práticas de corpos de conhecimento com o ITIL. Embora o termo DevOps ainda represente um conjunto difuso de práticas técnicas e culturais, é também verdade que um conjunto comum de práticas tem trazido notáveis resultados de negócios para muitas organizações. Empresas como WalMart, Staples, Amazon, Uber, Netflix, Microsoft, IBM, Globo.com e Leroy Merlin, entre diversas outras, já possuem casos publicados do valor de negócios da adoção do DevOps dentro de suas TIs.

O valor de negócios do DevOps pode ser expresso em métricas como:

- **MTTD (Mean Time to Deliver).** Também chamado de tempo de ciclo ou *Lead Time* na comunidade Lean, ela mede o tempo gasto desde o momento do nascimento da demanda ou projeto até a sua disponibilização no ambiente de produção para os seus usuários.
- **MTTR (Mean Time to Recover).** Mede o tempo gasto pelo time de operações para recuperar o ambiente de produção de um incidente.
- **% de Retrabalho.** Mede o percentual do esforço do projeto gasto com atividades não planejadas e resolução de defeitos.

Segundo o relatório *The State of DevOps Report 2016²⁰⁵*, empresas de alta maturidade em práticas DevOps têm o tempo de entrega de aplicações (MTTD) acelerado em 2500 vezes, são 24 vezes mais eficientes para recuperar falhas (MTTR) e têm 3 vezes menos retrabalhos que empresas de baixa maturidade. Esse relatório cita o caso da Amazon, que hoje realiza 80 implantações por dia em ambiente de produção. O relatório *The Value of IT Automation* do *Gartner Group²⁰⁶* mostra como empresas como *Walmart*, e *Staples* aumentaram a eficiência operacional de suas TIs com o uso de práticas DevOps.

Podemos pensar através na cultura DevOps com a confluência de três fatores críticos no desenvolvimento e manutenção de software: pessoas, processos e produtos.

- **Pessoas:** Envolve aproximar de times que trabalharam separados (Desenvolvimento, Qualidade e Operações). A cultura DevOps coloca essas pessoas no mesmo compasso, trabalhando juntas e

²⁰⁵ <https://puppet.com/resources/white-paper/2016-state-of-devops-report>

²⁰⁶ <https://puppet.com/resources/analyst-report/value-of-it-automation>

com o objetivo de garantir ritmo nas entregas e aumentar o fluxo de valor da TI para as áreas de negócio.

- **Processos:** Envolve enxugar a burocracia e desperdícios nos processos tradicionais de fazer e manter software. A cultura DevOps traz as práticas Agile/Scrum e Lean para dentro do ciclo de montagem de arquiteturas e produtos de forma pragmática e acionável para os times de desenvolvimento, qualidade e produção.
- **Produtos:** Envolve usar ferramentas de ciclo de vida para enlaçar disciplinas importantes tais como qualidade contínua, gestão de configuração, automação de testes, gestão de builds, gestão de releases e infraestrutura como código dentro de processos simples e acionáveis. O [Microsoft Visual Studio Team Services](#), [IBM Jazz](#), [GitLab](#), [Puppet Enterprise](#), [Chef](#) ou [Atlassian Bamboo](#) são alguns exemplos destes produtos e serão explorados ao longo deste capítulo.

Nos ambientes competitivos atuais, as empresas não irão escolher se vão implantar as práticas DevOps, mas quando irão fazê-lo e em qual velocidade.

13.2 DevOps para Criar Progresso Real nos Projetos

A abordagem tradicional de gerenciamento de desenvolvimento de software adia para os momentos finais a verificação se uma funcionalidade está funcionando. Uma funcionalidade **pronta** deveria atender um conjunto significativo de critérios tais como:

- Operar no ambiente real (ou de homologação) do cliente;
- Operar em uma base de dados real (ou similar) a do cliente;
- Operar de forma integrada com os sistemas legados que ele deve conversar;
- Possui uma suíte de testes de automação para garantir bons testes de regressão nos aspectos funcionais e não-funcionais;
- Não gera débito técnico de manutenibilidade no código fonte;

Ao mesmo tempo, defeitos são inerentes no trabalho de se fazer software, que está sujeito a variabilidade do trabalho intelectual humano. E se o **critério de pronto** não é forte, o software irá acumular defeitos ao longo do seu ciclo de vida. E a consequência é que o % de progresso real é na prática menor que o % de progresso declarado em um cronograma desconectado da realidade. Quanto mais robusto o critério de pronto, mais sólido tende a se tornar o produto e maior será a medição real do progresso do produto. Quando times, por imaturidade ou preguiça estabelecem um critério de pronto fraco ou não cumprem o acordo de pronto, haverá trabalho não feito no produto. O trabalho não feito é a diferença entre o trabalho necessário para ir para produção e o trabalho realizado no projeto. Em muitas empresas, o trabalho não feito é tão grande que ele gera um enorme débito técnico no sistema, que se materializa com defeitos em homologação e produção.

A questão que surge é se teremos uma mecânica de trabalho que irá expor e resolver de imediato os defeitos ou se deixaremos que os defeitos permaneçam no sistema e gerem problemas na homologação, ou pior, no ambiente de produção. Times DevOps acreditam na primeira alternativa e para isso estabelecem uma cultura que não apenas reduz a fricção de desenvolvimento, como também garantem a entrega de produtos robustos em produção.

Para isso é necessário que o time estabeleça um critério de pronto sólido para cada funcionalidade. Por exemplo, um **critério de pronto sólido** poderia estabelecer que uma funcionalidade está pronta se ela foi:

- compilada em um ambiente limpo, diferente da máquina de desenvolvimento onde ela foi codificada;
- testada com uma suíte de testes de unidade;
- testada com robôs de automação de testes de telas;
- testada com robôs para automação de testes de segurança, usabilidade, performance, escalabilidade ou recuperabilidade;
- integrada com sucesso no tronco principal;
- montada em um build com um número de versão;
- promovida de forma automatizada para um ambiente de homologação;
- aprovada para ser colocada em produção após os testes exploratórios e sistêmicos do time de QA.

A cultura DevOps busca apoiar o estabelecimento de critérios de prontos sólido, através de práticas culturais, práticas técnicas e suporte de aceleradores de automação. Embora o restante do capítulo seja dedicado a práticas e ferramentas, é relevante reforçar que o primeiro pilar da cultura DevOps está nas pessoas. Se o time não se comprometer com a mudança cultural, não haverá sucesso ao introduzirmos uma sofisticada ferramenta de gestão de builds ou gestão de releases. Busque isso antes de instalar a configurar a sua primeira ferramenta DevOps.

13.3 Práticas DevOps

Embora não exista um corpo fechado de práticas técnicas DevOps, podemos propor uma lista de prática básicas para a implantação desta cultura. A lista aqui apresentada não possui uma ordem de prioridade, que depende da realidade de cada organização e da cultura já instalada nos seus times de desenvolvimento, qualidade e operações.

As práticas DevOps buscam endereçar a melhoria nos atributos arquiteturais de qualidade interna descritos no começo deste capítulo, tais como a manutenibilidade, testabilidade ou implantabilidade. Com estas práticas implementadas, teremos maior garantia que a arquitetura definida pelo time de arquitetura será transformada em código executável e que minimizaremos o débito técnico dos produtos de software sendo construídos.

13.3.1 Comunicação Técnica Automatizada

A cultura DevOps busca aproximar pessoas dos times de desenvolvimento, qualidade e operações. E essa aproximação pode ser facilitada com ferramentas de comunicação que aliam o melhor da mensagem instantânea e canais de times com alarmes técnicos automatizados. Por exemplo, o Slack ou HipChat permitem que times com pessoas de desenvolvimento, qualidade e operações possam:

- estabelecer conversas texto, áudio e vídeo em canais privativos e focados;
- receber notificações automáticas tais como a disponibilização de builds, aprovação de releases ou problemas em produção;
- disparar comandos através de *bots* para a abertura de defeitos, escalonamento de builds ou releases.

13.3.2 Qualidade contínua do código

É fácil que uma arquitetura definida pelo time de arquitetura seja violada pelo time de desenvolvimento. Um arquiteto não tem tempo e dedicação para vigiar o código fonte e realizar a aderência arquitetural do código ou observar o uso de práticas apropriadas de codificação. O efeito é que a arquitetura executável colocada em produção pode ser diferente da arquitetura imaginada pelo arquiteto.

No sentido de minimizar esta lacuna, esta primeira prática lida com a automação da verificação da qualidade de código por ferramentas. Existem opções sólidas que permitem avaliar o uso das melhores práticas de programação no seu ambiente (*Code Metrics Tools*) e avaliar a aderência do seu código a uma arquitetura de referência (*Architectural Analysis Tools*). E essas podem ser programadas para rodar a noite ou até mesmo durante o momento do checkin do código pelo desenvolvedor. Existem ainda outras que facilitam o processo de revisão por pares dos códigos fontes (*Code Reviews*), estabelecendo workflows automatizados e trilhas de auditoria. O recurso de *Pull Requests* do Git é um exemplo deste mecanismo. Com esta prática, temos robôs que atuam para facilitar a análise do código fonte, educar desenvolvedores e estabilizar ou mesmo reduzir o débito técnico instalado.

13.3.3 Configuração como Código

É comum que desenvolvedores façam muitas tarefas de forma manual. Exemplos incluem cópias de arquivos entre ambientes, configuração destes ambientes, configuração de senhas, geração de *release notes* ou a parametrização de aplicações. Esses trabalhos manuais são propensos a erros e podem consumir tempo valioso do seu time com tarefas braçais.

Times atentos devem observar quando algum tipo de trabalho manual e repetitivo começa a acontecer e buscar automatizar isso. A automação da configuração através da escrita de códigos de scripts é um instrumento DevOps importante para acelerar o trabalho de times de desenvolvimento, reduzir erros e garantir consistência do trabalho.

13.3.4 Gestão dos Builds

A gestão de builds (ou *Build Management*) é prática essencial para garantir que os executáveis da arquitetura sejam gerados de forma consistente, em base diária. Esta prática busca evitar o problema comum do código funcionar na máquina do desenvolvedor e ao mesmo tempo quebrar o ambiente de produção.

A automação de builds externaliza todas as dependências de bibliotecas e configurações feitas dentro de uma IDE para um script específico e que possa ser movida entre máquinas com consistência. Embora a automação de builds, na sua definição formal, lide apenas com a construção de um build, a prática comum de mercado é que builds devam executar um conjunto mínimo de testes de unidade automatizados para estabelecerem confiabilidade mínima ao executável sendo produzido.

Esta prática lida ainda com o estabelecimento de repositórios confiáveis de bibliotecas, o que garante governança técnica sobre o conjunto de versões específicas de bibliotecas que estejam sendo usadas para montar uma aplicação.

13.3.5 Automação dos Testes

Organizações de baixa maturidade em automação de testes tem mais retrabalho ao longo de um projeto e geram mais incidentes em ambientes de produção. Esta prática DevOps buscar melhorar a testabilidade de aplicações e envolve:

- a criação de testes de unidade de código para as regras de negócio não triviais do sistema e também pontos críticos do código fonte tais como repositórios de dados complexos, controladores de negócio e interfaces com outros sistemas e empresas;
- a automação da interação com telas com testes funcionais automatizados;
- testes automatizados de aspectos não-funcionais, como segurança, acessibilidade, performance ou carga.

13.3.6 Testes de Carga

Em aplicações Web, podem haver variações abruptas no perfil de carga da aplicação em produção. Isso se deve a natureza estocástica no comportamento de requisições Web e a natureza do protocolo HTTP. Não é incomum que picos aconteçam e gerem 10x mais carga de trabalho que o uso médio da aplicação. Os sintomas comuns nas aplicações de mercado são longos tempos de resposta e até mesmo indisponibilidade do servidor Web (erros 5xx).

Estes sintomas podem ser minimizados com o uso de testes de carga em aplicações Web. O teste de carga é uma prática que permite gerar uma carga controlada para uma aplicação em um ambiente de testes específico ou homologação e assim estabelecer se um determinado build é robusto e pode ser promovido para ambientes de produção.

13.3.7 Gestão de Configuração

Times já usam ferramentas de controle de versão para organizar o seu código fonte, tais como o SVN, GIT ou Mercurial. Ao mesmo tempo, existem outras preocupações associadas ao trabalho feito por times em uma mesma linha de código. A ausência de políticas automatizadas de gestão de configuração digo aumenta a chance que desenvolvedores desestabilizem os troncos principais dos repositórios e gerem fricção e retrabalho para seus pares.

Neste contexto, existem opções de gestão de configuração de código permitem que os repositórios sejam mantidos em estado confiável e que erros comuns sejam evitados através da automação de políticas. Tarefas de gestão de configuração de código tais como a criação de rótulos (*labels*), geração de versões,

mesclagem de troncos de desenvolvimento e a manutenção de repositórios podem ser aceleradas e tornadas mais consistentes com o uso destes recursos. Como exemplo, o Git suporta o conceito de *books*²⁰⁷, mecanismo extensível de automação de políticas de gestão de código. Outras como o GitLab, VSTS ou Atlassian Bamboo já possuem estes mecanismos embutidos.

13.3.8 Automação dos Releases

A automação dos releases (*Release Management*) é uma prática que buscar garantir que o processo de promoção do executável para os ambientes de testes, homologação e produção sejam automatizados e assim tornados consistentes. Isso é importante porque em muitas organizações é difícil colocar um produto em ambiente de produção. A demora para acesso aos ambientes, alto número de passos manuais, a complexidade e a dificuldade de analisar os impactos são comuns. Isso gera atritos, longas demoras e desgastes entre times de QA, desenvolvimento e operações. E no fim isso também provoca erros nos ambientes de produção.

Esta prática tem como principais benefícios:

- reduzir o tempo para entregar um novo *build* em ambiente de produção através da automação da instalação e configuração de ferramentas e componentes arquiteturais;
- reduzir erros em implantação causadas por parâmetros específicos que não foram configurados pelos times de desenvolvimento e operações;
- minimizar a fricção entre os times de desenvolvimento, QA e operações;
- prover confiabilidade e segurança no processo de implantar aplicações.

Um ponto de atenção é que a automação de releases não deve recompilar a aplicação. Para garantir consistência, ela deve garantir que o mesmo executável que foi montado na máquina do desenvolvedor (mesmo conjunto de bits) esteja operando em outros ambientes. Ou seja, a automação de releases faz a movimentação dos executáveis entre os ambientes e modifica apenas os parâmetros da aplicação e variáveis de ambiente. Este processo pode também envolver a montagem de máquinas virtuais em tempo de execução do script.

13.3.9 Automação da Monitoração de Aplicações

Ao colocarmos um build em ambiente de produção, devemos buscar que o mesmo não gere interrupções nos trabalhos dos nossos usuários. Muitas empresas ainda convivem com incidentes em ambientes de produção causados por falhas nos processos de entrega em produção e desconhecimento de potenciais problemas.

Uma forma de reduzir a chance de incidentes para os usuários finais é implementar a monitoração contínua de aplicações (*Application Performance Monitoring*). Esta prática permite que agentes sejam configurados para observar a aplicação em produção. Alarmes podem ser ativados para o time de operações se certas condições de uso forem alcançadas ou se erros inesperados surgirem. Isso permite ter ciência de eventuais incidentes com antecedência e tomar ações preventivas para restaurar o estado estável do ambiente de operações.

E, para melhorar a experiência, alguns times já fazem que estes alarmes sejam enviados através de *bots* para as ferramentas de comunicação tais como o HipChat ou Slack.

13.3.10 Testes de Estresse

O teste de estresse é um tipo de teste de carga onde queremos criar fraturas em uma aplicação dentro de um ambiente com parâmetros de hardware estabelecidos a priori. Ele consiste em aumentarmos a carga em uma aplicação para saber qual será o primeiro componente a falhar por sobrecarga (ex. Banco de dados, servidor Web ou fila de mensagem). Esta técnica permite que o time de operações possa priorizar a sua atenção em infraestruturas complexas. Ela também permite aos times de desenvolvimento estabelecer os limites de uso da sua aplicação para os seus clientes.

²⁰⁷ <https://git-scm.com/book/pt-br/v1/Customizando-o-Git-Um-exemplo-de-Pol%C3%ADtica-Git-Forcada>

13.3.11 Integração Contínua (*Continuous Integration*)

Depois que a automação dos builds acontece, ela pode ser programada para ser executada em base diária ou até mesmos várias vezes por dia. Quando esta maturidade for alcançada, podemos avançar para que ela seja executada sempre, i.e., toda vez que um *commit* acontecer em um código fonte.

É esperado que esta prática faça pelo menos o seguinte conjunto de passos:

- promova a recompilação do código fonte do projeto;
- execute as suítes de testes de unidade automatizados do projeto;
- gere o build do produto;
- crie um novo rótulo para o build;
- gere defeitos automatizados para o time se o build falhou por algum motivo.

A prática da integração continua promove as seguintes vantagens:

- detectar erros no momento que os mesmos acontecem;
- buscar um ambiente de gestão de configuração estável de forma continuada;
- estabelecer uma mudança cultural no paradigma de desenvolvimento, através de feedbacks contínuos para o time de desenvolvimento da estabilidade do build.

13.3.12 Implantação Contínua (*Continuous Deployment*)

Depois que um processo mínimo de integração e implantação estiver em curso, podemos avançar também para um processo contínuo de implantação nos ambientes intermediários (testes ou homologação). Este processo pode ser controlado por uma ferramenta e é ativado quando um novo build foi gerado pela ferramenta de automação de builds.

Em linhas gerais, a prática da implantação contínua garante que mesmo conjunto de bits do build é replicado no ambiente do desenvolvedor para ambientes controlados de desenvolvimento e homologação, garantindo a consistência do build em outros ambientes.

Em ambientes governados onde existam processos ITIL, DBAs e times independentes de QA, irão haver ciclo de pré-aprovações ou pós-aprovações para que as promoções de ambiente ocorram. O fato importante a notar na cultura DevOps é que o ser humano envolvido não copia arquivos ou parametriza aplicações. Ele examina a requisição, o build, parâmetros e as suas evidências de teste. Ao aprovar a solicitação de promoção, um robô irá fazer a movimentação dos builds e a parametrização apropriada da aplicação. Se ele não autorizar a promoção, um defeito é aberto para o solicitante no canal apropriado.

13.3.13 Entrega Contínua (*Continuous Delivery*)

Em termos simples, a entrega contínua é o processo de **implantação contínua em ambiente de produção**. Ela é ativada por uma ferramenta automatizada quando um novo build for publicado com sucesso no ambiente de homologação. A entrega contínua envolve:

- a requisição de aprovação de implantação para o responsável pelo ambiente de produção;
- a cópia dos arquivos do ambiente de homologação para o ambiente de produção;
- a verificação do estado da aplicação disponibilizada em produção.

Observe que a entrega contínua não significa que o ambiente de produção é modificado a todo instante. Apenas empresas de serviços de Internet podem e devem fazer isso. A entrega contínua implica que o ambiente de produção pode ser alterado se um novo build estiver disponível e as aprovações necessárias foram dadas para a promoção do build.

A entrega contínua envolve a execução de *smoke tests*, que são testes de sanidade da aplicação. Estes testes verificam a estabilidade mínima do build colocado em produção e testam cenários funcionais mínimos.

Integração, Implantação e Entrega Contínua – Um resumo

A *integração continua* (*Continuous Integration*) é o processo de compilar o código em ambiente limpo, rodar testes e outros processos de qualidade e gerar um build, disparado por qualquer modificação no código fonte.

A *implantação continua* (*Continuos Deployment*) é o processo de copiar o build gerado no processo de integração para ambientes intermediários como QA ou homologação. Mas a promoção para o ambiente de produção sempre ocorre com alguma processo de aprovação humana.

A *entrega continua* (*Continuous Delivery*) é o processo de implantação continua que busca promover os builds do ambiente de homologação para o ambiente de produção sem intervenção humana

13.3.14 Implantações Canários

As implantações canários são práticas úteis para empresas que praticam a entrega contínua e querem minimizar efeitos colaterais de novas funcionalidades na comunidade de usuários.

Quando um novo produto é colocado em produção pela automação dos releases, pode haver um risco de negócio em liberar as novas funcionalidades para toda a sua comunidade de usuários. Talvez seja necessário fazer um certo experimento de negócio para saber se aquela funcionalidade será útil e mantida no produto.

Estes experimentos controlados podem ser ativados com os testes canários. O termo é devido a uma prática que ocorria nas minerações na Europa há alguns séculos. Mineiros levavam canários em gaiolas para novos veios em suas minas. Eles começam a trabalhar e deixavam os canários perto deles ao longo do dia de trabalho. Se um canário morresse depois de um tempo pequeno naquele ambiente, era porque o ambiente não estava saudável para a atividade humana devido a gases tóxicos. Na cultura DevOps da TI, a implantação canário consiste em habilitar novas funcionalidades apenas para um grupo controlado de usuários (os canários). Ou seja, em ambiente de produção a aplicação opera de duas formas distintas para duas comunidades (A/B). Isso pode ser implementado através do padrão de desenho chamado *Feature Toggles*²⁰⁸ e permite estabelecer uma prática chamada HDD (*Hypothesis Driven Development*). Se os canários não gostam do ambiente, a funcionalidade pode ser removida do produto em ambiente de produção sem intervenção no código fonte.

13.3.15 Infraestrutura como Código (IAC)

Times de baixa performance DevOps ainda possuem processos manuais e morosos de acionamento entre desenvolvimento e operações. Um exemplo prático é a criação de uma nova máquina feita do time de desenvolvimento para o time de operações. Em muitas empresas, este tipo de requisição demora horas, dias ou até mesmo semanas para ser realizada.

Quando times alcançam boa maturidade na configuração de elementos como scripts, elas podem avançar e tratar até o mesmo o hardware como código. Através de tecnologias disseminadas nos últimos anos em ambientes Linux, Windows ou OS/X, é possível configurar os processos de automação de build e automação de releases para criar uma máquina virtual através de um código de script. A infraestrutura como permite estabelecer para o time de operações confiabilidade apropriada para as novas implantações realizadas em ambientes de produção.

A infraestrutura como código traz ainda como grande benefício estabelecer um protocolo comum entre os times de desenvolvimento e o time de operações. Esta prática elimina a necessidade da criação manual de ambientes físicos, que é moroso, propenso a erros e que causa atrito entre times. Ao automatizar esse processo, podemos reduzir o tempo de ciclo para entrega de aplicações em produção. A tecnologia do Docker²⁰⁹ é um excelente exemplo neste sentido.

Considere o exemplo do código a seguir. Nele vemos um arquivo em sintaxe Ansible, que é uma ferramenta de provisionamento de ambientes. Este arquivo define um estado de configuração desejado de um nodo de hardware em um ambiente. Ele verifica se o servidor Apache existe no cluster de máquinas denominado *webservers*. Se existir, ele baixa e instalar. Se já existir, ele avança o script. Após a instalação, ele verifica se o servidor está rodando. Se não estiver, ele sobe o servidor. Se ele já estiver rodando, ele não interfere no estado atual.

²⁰⁸ <http://martinfowler.com/articles/feature-toggles.html>

²⁰⁹ <http://docker.com>

```

---
- hosts: webservers
  vars:
    http_port: 80
    max_clients: 200
    remote_user: root
  tasks:
    - name: ensure apache is at the latest version
      yum: name=httpd state=latest
    - name: write the apache config file
      template: src=/srv/httpd.j2 dest=/etc/httpd.conf
      notify:
        - restart apache
    - name: ensure apache is running (and enable it at boot)
      service: name=httpd state=started enabled=yes
  handlers:
    - name: restart apache
      service: name=httpd state=restarted

```

Figura 57: Arquivo de script AnsibleFonte: <http://docs.ansible.com>

Através de utilitários como o Ansible, Power Shell DSC, Puppet, Chef, entre outras, estes arquivos de scripts podem ser executados em plataformas locais ou de nuvens como a Microsoft Azure ou Amazon EC2. A implantação deste arquivo cria uma máquina virtual com a exata especificação informada. Ou seja, todo o trabalho de criação manual de uma máquina virtual e sua tediosa configuração é eliminado. Ao invés, criamos e testamos um script em alguma linguagem e o ambiente subjacente se encarrega de fazer o provisionamento das máquinas virtuais no ambiente de produção.

13.3.16 Ambientes Self-Service

Em muitas empresas, é comum que um novato demore horas ou até dias para que consiga estabelecer um novo ambiente de trabalho. Isso é devido ao conjunto de passos manuais necessários, falta de procedimentos operacionais e dificuldades implícitas a montagem de ambientes Java EE ou .NET.

A prática de ambientes *self-service* permite que através de um código de script todo um ambiente de trabalho seja baixado, criado e disponibilizado para habilitar um desenvolvedor no seu trabalho em poucos minutos. Dado que um desenvolvedor tenha uma estação de trabalho com excelente memória RAM e uma rede veloz, é possível operacionalizar esta prática e salvar tempo precioso como o estabelecimento de ambientes de trabalho com confiabilidade e robustez. O Docker, em particular, é uma tecnologia que ganhou popularidade nos últimos anos para apoiar também esta prática.

13.3.17 Injeção de Falhas

Uma prática avançada DevOps é injetar defeitos no ambiente de produção de forma explícita. Por exemplo, podemos desligar o acesso ao banco de dados ou outros recursos críticos e forçar a aplicação a falhar. Isso pode parecer bizarro para alguns ou um contrassenso para outros. Mas pode fazer todo o sentido quando estamos buscando ambientes de alta disponibilidade e confiabilidade.

Através do uso de procedimentos controlados de desestabilização da aplicação, podemos verificar como a aplicação se recupera de uma falha em ambiente de produção. Algumas perguntas que o time de operações poderia investigar incluem:

- Ela se recupera e retorna para o estado original antes da falha?
- Ela emite alarmes apropriados para as partes interessadas?
- Ela fornece mensagens simples e explicativas para os usuários finais?

Esta prática começou a ganhar momento na TI depois que a Netflix publicou²¹⁰ o uso desta prática nos seus ambientes de produção e a disponibilização da sua ferramenta de injeção de falhas chamada *Simian*.

²¹⁰ <https://www.infoq.com.br/news/2012/08/netflix-chaos-monkey>

*Army*²¹¹. Ela permite estabelecer um mecanismo de antifragilidade²¹² na sua aplicação, tornando-a melhor ao longo do tempo à medida que ela seja estressada.

13.3.18 Telemetria

A telemetria é uma forma avançada de monitoração de aplicações em ambiente de produção. Ela permite conhecer os padrões de uso de uma aplicação, variações de carga, acesso, entre outras questões. A Telemetria conta também com um mecanismo intrínseco de capacidade de análise de uso (*analytics*) que permite aos times conhecer os padrões de acesso e assim evoluir o produto do ponto de vista técnico e de negócio.

13.3.19 Planejamento de Capacidade

O planejamento de capacidade envolve o uso de técnicas estatísticas e teoria de filas para conhecer, modelar e simular a carga de trabalho em aplicações e assim estabelecer o hardware mais apropriado para rodar uma aplicação, bem como ter ciência dos limites e potenciais problemas de operação.

Esta técnica pode ser implementada por ferramentas de testes de carga e performance e são úteis para empresas que trabalhem com cenários desafiantes de cargas de trabalho e busquem o uso de computação elástica em ambientes de nuvens.

13.4 Ferramentas DevOps

Práticas DevOps introduzem aceleradores arquitetural. Por isso organizamos nesta seção estas ferramentas por categoria para permitir que o leitor possa saber que problemas técnicos e condutores de negócio cada uma delas endereça.

13.4.1 Plataformas de Colaboração e Comunicação

A cultura DevOps pede aumento da comunicação entre os times de desenvolvimento, qualidade e operação. Ela também demanda um ambiente centralizado para a operação de ferramentas do ciclo de automação de práticas DevOps. O IBM Jazz, ServiceNow e Visual Studio Team Services são exemplos neste sentido, buscando fornecer um fluxo automatizado para o ciclo de vida de práticas DevOps. Já o Slack e HipChat permitem integrar o conceito de conversas entre pessoas e grupos com as tarefas de automação feita via bots.

13.4.2 Ferramentas de Análise de Código Fonte

A cultura DevOps requer que desenvolvedores escrevam código de qualidade. Nos últimos anos, foi possível quantificar o que é qualidade de código de forma prática. Termos vagos como complexidade ciclomática, modularidade, coesão alta e acoplamento fraco se tornaram métricas em ferramentas simples como o Visual Studio Code Review e o SonarQube. Esta última tem suporte para mais de 20 linguagens e se popularizou nos últimos anos na comunidade de TI. Além disso, práticas de refatoração em tempo de desenvolvimento se tornaram simples também com a IDE JetBrains IntelliJ, JetBrains ReSharper para o Visual Studio, Eclipse IDE ou Microsoft Visual Studio Code. E ferramentas como o Git suportam conceito de *pull requests*, que simplificam a revisão por pares entre desenvolvedores.

A recomendação para arquitetos aqui é incluírem estas ferramentas no ciclo de desenvolvimento e usarem a automação para emitir avisos, defeitos ou até mesmo impedirem o checkin de códigos em casos extremos.

²¹¹ <https://github.com/Netflix/SimianArmy>

²¹² A fragilidade significa que algo quebra sob algum estresse, como por exemplo um copo fino solto de certa altura. Já a robustez indica que algo resiste a um estresse sem alterar o seu estado. Mas a anti-fragilidade vai além da robustez e resiliência. Um organismo anti-frágil melhora o seu estado depois de submetido a um estresse. Por exemplo, exercícios físicos, até um certo nível, geram estresses em pessoas e a resposta do corpo delas é se tornar melhor com uma melhor densidade óssea e maior massa muscular. A anti-fragilidade é o oposto matemático da fragilidade. Enquanto a fragilidade é denotada como um número negativo -X, a robustez seria denotada como o número 0 e a anti-fragilidade seria denotada como um número positivo X. Este conceito é apresentado e discutido no livro Anti-Frágil – Coisas que se beneficiam com o caos, de Nicholas Taleb, publicado em 2012.

13.4.3 Ferramentas de Gerência de Código Fonte (SCM)

É impossível trabalhar com a cultura DevOps sem gerir o código fonte. E aqui estamos falando de ferramentas como o Subversion (SVN), Microsoft TFVC (*Team Foundation Version Control*), Mercurial ou Git. As duas primeiras operam de forma centralizada com um servidor de código, enquanto as duas últimas operam de forma distribuída. Em um SCM distribuído, existe um clone de cada repositório em cada máquina que tenha uma cópia do repositório de código.

Em linhas gerais, as ferramentas de SCM distribuídos são melhores pois facilitam a gestão de troncos, automação de política e tem performance melhor alternativas de SCM centralizados. O Git, que foi popularizado a partir da experiência da comunidade Linux em manter de forma distribuída o desenvolvimento do seu kernel, se tornou popular no Brasil nos últimos anos. Se possível na sua realidade opere um SCM distribuído, seja ele Git ou Mercurial.

Ao escolher um SCM, seja ele distribuído ou centralizado, é importante decidir se ele irá operar em ambiente local ou ambiente de nuvem. O GitHub, BitBucket e o GitLab são provedores de nuvem populares sobre os ambientes Git e Mercurial. Eles operam com planos sem custo de aquisição com funcionalidades e espaços limitados, mas permitem escalar para espaços maiores e mais funcionalidades com custos pequenos de entrada. Embora alguns gestores ainda temam as nuvens, observamos que a experiência de uso de ambientes de nuvens traz maior simplicidade, maior disponibilidade e menor custo de operação.

13.4.4 Ferramentas de Automação de Builds

Estas ferramentas são o coração da prática de gestão de builds e são específicas por tecnologia. A comunidade Java EE utiliza o Maven ou o Gradle para os seus processos de build. A comunidade Microsoft usa o MSBuild, que já vem incorporado na IDE do Visual Studio, Microsoft TFS (*Team Foundation Server*) e também no VSTS (TFS nas nuvens). A comunidade JavaScript usa com regularidade o Grunt e o Gulp. Outras populares para este fim incluem o Rake (Make do Ruby), Broccoli, SBT e Packer.

13.4.5 Ferramentas de Integração Contínua

Dão suporte ao escalonamento e gestão do processo de builds. Elas operam sobre as ferramentas de automação de builds e permitem que o desenvolver escalone ações automatizadas de build, sejam elas noturnas (*nightly builds*), em certos momentos do dia ou disparadas por *commits* nos códigos (prática de *Continuous Integration*). Elas também permitem rodar outras tarefas tais como execução de suítes de testes, geração de documentação de código e abertura automatizada de defeitos.

O Jenkins é uma ferramenta sem custo de aquisição e bem popular para este tipo de processo. Ela é usada por desenvolvedores Java, JavaScript e LAMP. Já a comunidade Microsoft faz uso intenso do Microsoft TFS (*Team Foundation Server*) ou o seu equivalente de nuvem - Microsoft VSTS (*Team Services*). O VSTS permite também incorporar builds de aplicações Java, Android, iOS, Xamarin, entre outros e se tornou uma ferramenta independente de ambiente Windows. Outras opções para CI incluem o Atlassian Bamboo, Travis CI, IBM Rational Team Concert, Code Ship, Thought Works CruiseControl, CodeShip, TeamCity, GitLab, SolanoCI, Continuum, ContinuaCI e Shippable. Elas já fornecem ambientes de nuvens e algumas permitem você baixar o servidor para operar no seu ambiente se necessário.

Adote na sua iniciativa DevOps uma ferramenta de CI que esteja bem integrada com as suas tecnologias de código fonte e a opere em nuvens se isso não ofender as políticas de segurança da sua organização. Observamos nos preços destes produtos um custo menor de entrada e propriedade ao adotarmos infraestruturas de nuvens para a execução das práticas de automação de builds e integração contínua.

13.4.6 Ferramentas de Gestão de Configuração e Provisionamento

Permitem automatizar todas as suas configurações como código e também provisionar hardwares, sendo também importantes para suportar releases de produtos (Release Management). Estas ferramentas são chamadas em alguns meios de CCA Tools (*Continuous Configuration Automation*).

Em linhas gerais, elas permitem:

- o escalonamento temporal do processo de release, que pode ser feito em base noturna (*nightly release*) ou ativado toda vez que um novo build estiver disponível (*Continuous Deployment* ou *Continuous delivery*);
- escrever a configuração de hardware através de códigos de script;
- provisionar de forma dinâmica os ambientes de hardwares;
- copiar os builds entre os ambientes de hardware provisionados;
- a conexão a plataformas de nuvens como Amazon EC2, Microsoft Azure, entre outras;
- configurar os parâmetros da aplicação copiada para os ambientes;
- estabelecer processos de aprovação antes e/ou depois das cópias dos builds;
- rodar *smoke tests*, gerar rótulos (labels) nos releases ou enviar notificações para os interessados no processo.

Na comunidade Microsoft, o VSTS incorporou um excelente módulo de gestão de liberações. Ele suporta scripts de infraestrutura como código em um dialeto chamado PowerShell DSC (*Desired State Configuration*). Em termos simples, ele permite que o desenvolvedor especifique um hardware em um script de código DSC (que é um arquivo JSON). O ambiente VSTS cria este ambiente de hardware no Microsoft Azure. Além disso, o VSTS Release Management controla o fluxo de aprovações, executa as distribuições necessárias dos builds nos ambientes, realiza a configuração dos parâmetros externalizados da aplicação nos ambientes provisionados e também permite rodar testes nos ambientes de produção.

Neste segmento estão também os maiores fornecedores de soluções e consultoria DevOps, como por exemplo as empresas Chef e Puppet Enterprises. Outras soluções comuns neste segmento incluem a Ansible, Salt, Vagrant, Terraform, Consul, CF Engine, BMC BladeLogic, BMC Release Process e Serena Release, entre outras.

13.4.7 Ferramentas de Conteinerização

Como operar com dezenas ou centenas de máquinas virtuais para o suporte a provisionamento pode ser caro, algumas ferramentas surgiram para facilitar a gestão de ambientes virtualizados. Elas operam permitindo que blocos de máquinas virtuais possam ser arranjados conforme necessário para criar novas configurações. Por exemplo, um time pode ter já um ambiente virtual com Oracle Database 11g, Apache Tomcat e Driver JDBC tipo 2 montado. Se um outro time precisa modificar apenas a versão do Driver JDBC para tipo 4 em um outro contexto, ele não precise recriar toda a máquina virtual do zero. Ele especifica as partes necessárias e ferramenta de conteinerização monta a nova máquina virtual a partir de fragmentos existentes no repositório. A ferramenta também inclui os novos fragmentos, conforme necessário. Isso gera uma tremenda economia de espaço em disco e tempo para a organização de máquinas virtuais.

O Docker é talvez a ferramenta de maior popularidade neste segmento e tem sido usado para tornar o provisionamento de hardware algo simples, barato e acionável. Outras populares neste contexto são o Mesos, Swarm, Kubernetes, Nomad e Rancher.

13.4.8 Ferramentas de Gestão de Repositórios

Permitem gerir os repositórios de código e bibliotecas para estabelecer ambientes controlados de desenvolvimento. Isso evita que componentes sejam introduzidos na aplicação de forma indisciplinada e através de cópia de arquivos no sistema de arquivos.

O Docker Hub é uma ferramenta neste sentido e disciplina o uso de máquinas virtuais Docker, nas nuvens ou na própria infraestrutura da empresa. No mundo Microsoft, o NuGet e o Package Management do VSTS são ferramentas usadas para este controle. Já a comunidade JavaScript e Node.js usa o NPM (Node Package Manager). Já a comunidade Java usar o Artifactory e o Nexus para estabelecer bases controladas de bibliotecas e componentes.

13.4.9 Ferramentas de Automação de Testes

E experiência mostra que builds requerem o uso de automação de testes para aumentar a robustez e confiabilidade do produto sendo gerado. Algumas destas ferramentas incluem:

- Automação para testes de unidade do código – JUnit, TestNG, NUnit, Visual Studio Unit Test, Jasmine, Mocha e QUnit.
- BDD (Behaviour Driven Development) – Cucumber, CucumberJS, e SpecFlow.
- Automação de testes funcionais – Selenium e Visual Studio Coded UI.
- Cobertura de Código – JaCoCo e Visual Studio

13.4.10 Ferramentas de Testes de Performance, Carga e Estresse

Em aplicações Web e móveis, onde a carga de trabalho pode variar com muita intensidade, é recomendado que usemos a automação de teste de performance, carga de trabalho e estresse. O JMeter é uma ferramenta popular na comunidade Java, embora também possa ser usada para testar recursos Web como conexões HTTP em qualquer tecnologia. O TestNG, VSTS Web Performance e o VSTS Load Test também são outros exemplos para este fim.

Depois de ter estabelecido um processo de automação de build e releases, pode ser apropriado inserir um ciclo de testes de performance e estresse no seu processo DevOps.

Uma transição suave para a operação pode ser facilitada pelo uso de monitoração de aplicações e telemetria. Em termos simples, elas fazem a análise dos elementos físicos dos ambientes de hardware (caixa preta), componentes da aplicação (caixa cinza) e até mesmo o comportamento do código fonte em produção (caixa branca).

A ferramenta NewRelic é um excelente exemplo neste sentido e se popularizou para a monitoração de aplicações Web. Outros exemplos incluem o Kibana, DataDog, Zabbix, VSTS Application Insights, ElasticSearch e StackState.

13.4.11 Ferramentas de Injeção de Falhas

Permitem injetar falhas em aplicações nos ambientes de homologação e produção, como por exemplo o Netflix *Simian Army*. Esta suíte²¹³, que foi disponibilizada no GitHub, contém utilitários diversas tais como:

- Chaos Monkey – Introduz falhas aleatórias em máquinas dos ambientes de produção.
- Latency Monkey – Introduz demoras artificiais nas comunicações RESTful para simular degradação do serviço e medir o seu efeito nas aplicações cliente.
- Conformity Monkey – Encontra instâncias que não estão aderentes a melhores práticas e as desligam. Por exemplo, uma melhor prática poderia ser que toda instância deveria pertencer a um grupo de escala (AutoScale) no ambiente AWS EC2 da Amazon. Se uma instância é encontrada e não obedece a esta política, ela é terminada.
- Doctor Monkey – Encontra instâncias que não estejam saudáveis e as desligam. Por exemplo, um sinal de falha na saúde é a CPU operar acima de 70% por mais de 1 hora. Se uma instância é encontrada em um estado não saudável, ela é terminada.
- Janitor Monkey – Busca e limpa recursos não usados nos ambientes de produção.
- Security Monkey – Busca vulnerabilidades em máquinas e desliga as instâncias que estejam ofendendo as políticas de segurança.
- 10-18 Monkey – Detecta problemas de configuração (i10n e 18n) em instâncias que servem a múltiplas regiões geográficas.
- Chaos Kong – Remove uma região inteira de disponibilidade da Amazon do ambiente de produção.

13.4.12 Plataformas de Nuvens

As plataformas de nuvens possuem um papel importante no ciclo DevOps. Sejam públicas ou privadas, elas permitem tratar a infraestrutura e redes como código e facilitam sobremaneira os processos de gestão

²¹³ <http://techblog.netflix.com/2011/07/netflix-simian-army.html>

de releases. A Amazon WebServices talvez seja o maior expoente deste segmento, com uma rica coleção de serviços de IAAS, PAAS e SAAS para suportar o desenvolvimento e operação de aplicações. Outras soluções populares incluem o Microsoft Azure, RackSpace, Digital Ocean e Google Cloud Platform.

Bibliografia

- APIGEE. (2014). *API for Dummies*. APIGEE. Retrieved from <https://pages.apigee.com/ebook-apis-for-dummies-reg.html>
- Barbacci, M., Ellison, R. J., Lattanze, A. J., Stafford, J., Weinstock, C., & Wood, W. (2003). Quality Attribute Workshops (QAWs). SEI.
- Bass, L., Clements, P., & Kazman, R. (2012). *Software Architecture in Practice* (3rd ed.). Addison-Wesley Professional.
- Bien, A. (2012). *Real World Java EE Patterns--Rethinking Best Practices* (1ed ed.). <http://press.adam-bien.com>.
- Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., ... Stafford, J. (2010). *Documenting Software Architectures: Views and Beyond* (2nd ed.). Addison-Wesley Professional.
- Clements, P., Kazman, R., & Klein, M. (2001). *Evaluating Software Architectures: Methods and Case Studies* (1st ed.). Addison-Wesley Professional.
- Eeles, P. (2005). Capturing Architectural Requirements. Retrieved from <http://www.ibm.com/developerworks/rational/library/4706.html>
- Elliot, E. (2014). *Programming JavaScript Applications: Robust Web Architecture with Node, HTML5, and Modern JS Libraries*. O'Reilly Media.
- Erl, T. (2013). *Cloud Computing: Concepts, Technology & Architecture*. Prentice Hall PTR.
- Meier, J. D. (2009). *Microsoft application architecture guide, Patterns and Practices*. Microsoft Press.
- Newman, S. (2015). *Building Microservices*. (O. Media, Ed.). O'Reilly Media.
- Oracle. (2013). Especificações Java EE 7. Retrieved from <http://www.oracle.com/technetwork/java/javaee/tech/index.html>
- Paulish, D. J. (2012). *Architecture-Centric Software Project Management: A Practical Guide* (1st ed.). Addison-Wesley Professional.
- Price, M. (2016). *C# 6 and .NET Core 1.0: Modern Cross-Platform Development*. Packt Publishing.
- Rozanski, N., & Woods, E. (2005). *Software Systems Architectures: Working With Stakeholders Using Viewpoints and Perspectives* (1st ed.). Addison-Wesley Professional. Retrieved from <http://www.amazon.com/dp/0321112296>
- Stefanov, S. (2010). *JavaScript Patterns*. O'Reilly Media.