

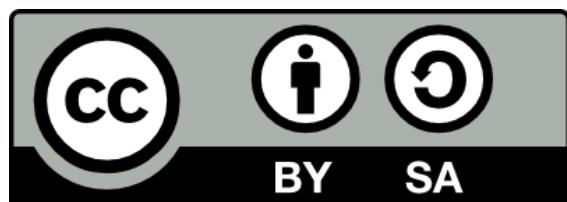
Arquitetura de Sistemas Web

Princípios, Práticas e Tecnologias

Edição 2019

Marco Mendes

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



Edição 2019
Versão 1.7.0

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 em Métodos Ágeis, DevOps e Arquitetura de Sistemas pela empresa Arkhi.

Possuo 26 anos de experiência profissional no Brasil e no exterior, sendo especialista em arquiteturas corporativas, arquitetura de aplicações Web, arquitetura Java EE, .NET, APIs, SOA, microserviços, engenharia de software e métodos ágeis.

Mantenho um blog com informações complementares ao livro em <http://marco-mendes.com>.

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

1 Fundamentos de Arquiteturas Web.....	12
1.1 Introdução	12
1.2 Como navegar pelo livro	13
1.3 Arquitetura de Referência de um Sistema Web	14
1.4 O Protocolo HTTP	14
1.5 Segurança, Cache e Idempotência nos Métodos HTTP	16
1.6 A Nomeação de Recursos na Web - URIs.....	18
1.7 A linguagem XML	21
1.8 A notação JSON	21
1.9 Folhas de Estilo em Cascata - CSS.....	21
1.10 A Linguagem JavaScript	21
1.11 Desafios Arquiteturais Web.....	22
1.12 O Papel das Arquiteturas e Arquitetos Web	24
2 Como Arquitetar Sistemas Web	25
2.1 O Que são arquiteturas de software.....	25
2.2 A Função dos Arquitetos de Software	26
2.3 Um Processo Mínimo para Arquitetar Sistemas	27
2.4 A Relação do Arquiteto com o Time do Projeto.....	28
2.5 Para Saber Mais	29
3 Requisitos Arquiteturais Web	31
3.1 O que são Requisitos Arquiteturais	31
3.2 Como Descrever Bons Requisitos Arquiteturais.....	31
3.3 Requisitos de Acessibilidade e Usabilidade	32
3.4 Requisitos de Autenticação e Autorização	33
3.5 Requisitos de Confidencialidade e Integridade	33
3.6 Requisitos Amplos de Segurança	34
3.7 Requisitos de Alta Disponibilidade	34
3.8 Requisitos de Tolerância a Falhas.....	35
3.9 Requisitos de Performance	35
3.10 Para Saber Mais	36
4 Estilos Arquiteturais Web	37
4.1 Introdução.....	37
4.2 Web 1.0	37
4.3 Web 2.0	37
4.4 SPA (<i>Single Page Application</i>)	38
4.5 API Web RESTful	39
4.6 Microsserviços Web.....	43
4.7 Computação sem servidor (<i>Serverless</i>).....	46
4.8 Comparativo dos Estilos Arquiteturais.....	50
4.9 Para Saber Mais	51
5 Topologia de Servidores Web.....	53
5.1 Servidores Web Baseados em Processos (1º Geração)	53

5.2	Servidores Web Baseados em Threads (2º Geração)	53
5.3	Servidores Web de Aplicações (3º Geração).....	54
5.4	Servidores Web de Micro contêineres (4º Geração)	54
5.5	Comparativo de Servidores Web	55
5.6	Distribuição Física de Aplicações Web.....	56
5.7	Distribuição Mínima.....	56
5.8	Distribuição com Servidor Web Dedicado.....	56
5.9	Distribuição com Servidor de Aplicação Dedicado.....	57
5.10	Distribuição com Máquina para Conteúdo Estático	60
5.11	Distribuição com Arquiteturas Elásticas	60
5.12	Comparativo de Topologias Web.....	61
5.13	Para Saber Mais	62
6	Servidores Web Baseados em Java EE.....	63
6.1	JSF (Java Server Faces)	65
6.2	Servlets.....	65
6.3	Serviços Web (JAX-WS e JAX-RS)	65
6.4	JAAS (Java Authentication and Authorization Service).....	66
6.5	EJB (Enterprise Java Beans) Session Beans.....	66
6.6	JPA (Java Persistence API)	67
6.7	JCA (Java Connector Architecture).....	67
6.8	Considerações de Desenho Arquitetural.....	67
6.8.1	ESCOLHA DE SERVIDORES	67
6.8.2	PRODUTIVIDADE	68
6.8.3	PROCESSAMENTO DE REQUISIÇÕES WEB	68
6.8.4	NAVEGAÇÃO	69
6.8.5	DESENHO DE PÁGINAS.....	69
6.8.6	AUTENTICAÇÃO	69
6.8.7	AUTORIZAÇÃO.....	69
6.8.8	CACHE	69
6.8.9	CONTROLE TRANSACIONAL.....	69
6.8.10	AUDITORIA (LOGS)	70
6.8.11	INSTRUMENTAÇÃO.....	70
6.8.12	GERÊNCIA DE SESSÃO WEB	70
6.8.13	VALIDAÇÃO	70
6.8.14	DESENHO DE SERVIÇOS WEB	71
6.8.15	CAMADA DE NEGÓCIO	71
6.8.16	ACESSO A DADOS	71
6.9	Riscos e Oportunidades para o Arquiteto Web Java EE	71
6.9.1	CURVA DE APRENDIZADO ALTA	71
6.9.2	BAIXA PRODUTIVIDADE.....	72
6.9.3	CONSUMO DE MEMÓRIA	72
6.9.4	DESCONTINUAÇÃO TECNOLÓGICA DE FRAMEWORKS.....	72
6.10	Para Saber Mais	72
7	Servidores Web Baseados em ASP.NET.....	74
7.1	ASP.NET Web Forms	76
7.2	ASP.NET MVC	76

7.3	ASP.NET Core.....	77
7.4	WCF.....	78
7.5	ASP.NET Web API	79
7.6	Entity Framework e LINQ	79
7.7	MSMQ	80
7.8	Service Fabric	80
7.9	Considerações de Desenho Arquitetural	81
7.9.1	CONFIGURAÇÃO DE SERVIDORES.....	81
7.9.2	PRODUTIVIDADE	82
7.9.3	PROCESSAMENTO DE REQUISIÇÕES WEB	82
7.9.4	NAVEGAÇÃO	82
7.9.5	DESENHO DE PÁGINAS.....	83
7.9.6	AUTENTICAÇÃO	83
7.9.7	AUTORIZAÇÃO.....	83
7.9.8	CACHE	83
7.9.9	CONTROLE TRANSACIONAL.....	84
7.9.10	AUDITORIA (LOGS).....	84
7.9.11	INSTRUMENTAÇÃO.....	84
7.9.12	GERÊNCIA DE SESSÃO WEB	84
7.9.13	VALIDAÇÃO	85
7.9.14	DESENHO DE SERVIÇOS WEB	85
7.9.15	ACESSO A DADOS	85
7.10	Riscos e Oportunidades para o Arquiteto Web ASP.NET	85
7.10.1	ESTRUTURAÇÃO DE BONS MODELOS DE DOMÍNIO E BONS CÓDIGOS	85
7.10.2	CONSUMO DE MEMÓRIA EM APLICAÇÕES ASP.NET WEBFORMS.....	85
7.10.3	A NOVA ARQUITETURA ASP.NET CORE	86
7.10.4	MAPEAMENTO DE TECNOLOGIAS JAVA EE E .NET	86
7.11	Para Saber Mais	87

8 Servidores Web Baseados em JavaScript 88

8.1	Aceleradores, Bibliotecas e Frameworks CSS	90
8.2	Linguagens Aceleradores sobre JavaScript	91
8.3	Linguagem ECMA Script 2015 (JavaScript 6)	93
8.4	Bibliotecas de Componentes JavaScript	93
8.5	Framework MV* JavaScript.....	94
8.6	Ferramentas de Suporte JavaScript.....	95
8.6.1	GERENCIADORES DE DEPENDÊNCIAS	95
8.6.2	EXECUTORES DE TAREFAS	96
8.6.3	AUTOMAÇÃO DE TESTES COM JAVASCRIPT	96
8.7	JavaScript Servidor e o Node.JS	96
8.8	Mapeamento de Tecnologias JavaScript, Java EE Web e ASP.NET	98
8.9	Riscos e Oportunidades para o Arquiteto JavaScript	99
8.10	Para Saber Mais	99

9 Arquitetura de Sistemas Móveis..... **Erro! Indicador não definido.**

9.1	Componentes da Arquitetura	Erro! Indicador não definido.
9.2	Segurança.....	Erro! Indicador não definido.
9.3	Cache.....	Erro! Indicador não definido.

9.4	Comunicação de Rede	Erro! Indicador não definido.
9.5	Portabilidade	Erro! Indicador não definido.
9.6	Testabilidade	Erro! Indicador não definido.
9.7	Gestão de Energia	Erro! Indicador não definido.
9.8	Usabilidade	Erro! Indicador não definido.
9.9	Tecnologias Móveis	Erro! Indicador não definido.

10 Documentação de Arquiteturas.....101

10.1	Passo 1 - Visualização de Negócio.....	102
10.2	Passo 2 – Condutores Arquiteturais	102
10.2.1	ACESSIBILIDADE E USABILIDADE.....	103
10.2.2	INTEROPERABILIDADE	103
10.2.3	SEGURANÇA.....	103
10.2.4	ESCALABILIDADE	103
10.2.5	MANUTENIBILIDADE.....	103
10.3	Passo 3 – Estilos Arquiteturais Web	103
10.4	Passo 4 – Escolha da Plataforma Tecnológica	104
10.5	Passo 5 – Visualização Lógica	104
10.6	Passo 6 – Visualização da Topologia Física	106
10.7	Passo 7 – Visualização da Persistência de Dados.....	106
10.8	Passo 8 – Visualização da Apresentação (Usabilidade e Acessibilidade)	108
10.9	Passo 9 – Visualização de Segurança	110
10.10	Passo 10 – Visualização de Interoperabilidade.....	111
10.11	Passo 11 – Visualização de Manutenibilidade.....	112
10.12	Passo 12 – Alocação de Componentes aos Nodos Físicos	113
10.13	Passo 13 – Determinar Provas de Conceito	115
10.14	Modelos de um Documento de Arquitetura de Software	115

11 Aceleração de Arquiteturas com Práticas DevOps.....117

11.1	DevOps para Aceleração da Entrega de Produtos.....	117
11.2	DevOps para Criar Progresso Real nos Projetos.....	118
11.3	Práticas DevOps Básicas	119
11.3.1	COMUNICAÇÃO TÉCNICA AUTOMATIZADA	119
11.3.2	QUALIDADE CONTÍNUA DO CÓDIGO.....	119
11.3.3	CONFIGURAÇÃO COMO CÓDIGO.....	120
11.3.4	GESTÃO DOS BUILDS	120
11.3.5	AUTOMAÇÃO DOS TESTES	120
11.3.6	TESTES DE CARGA	120
11.3.7	GESTÃO DE CONFIGURAÇÃO	121
11.3.8	AUTOMAÇÃO DOS RELEASES.....	121
11.3.9	AUTOMAÇÃO DA MONITORAÇÃO DE APLICAÇÕES	121
11.4	Práticas DevOps Avançadas	122
11.4.1	TESTES DE ESTRESSE	122
11.4.2	INTEGRAÇÃO CONTÍNUA (<i>CONTINUOUS INTEGRATION</i>)	122
11.4.3	IMPLANTAÇÃO CONTÍNUA (<i>CONTINUOUS DEPLOYMENT</i>)	122
11.4.4	ENTREGA CONTÍNUA (<i>CONTINUOUS DELIVERY</i>)	123
11.4.5	IMPLANTAÇÕES CANÁRIOS	123
11.4.6	INFRAESTRUTURA COMO CÓDIGO (<i>IAC</i>)	124

11.4.7	AMBIENTES SELF-SERVICE	125
11.4.8	INJEÇÃO DE FALHAS	125
11.4.9	TELEMETRIA.....	125
11.4.10	PLANEJAMENTO DE CAPACIDADE.....	126
11.5	Ferramentas DevOps.....	126
11.5.1	PLATAFORMAS DE COLABORAÇÃO E COMUNICAÇÃO	126
11.5.2	FERRAMENTAS DE ANÁLISE DE CÓDIGO FONTE	126
11.5.3	FERRAMENTAS DE GERÊNCIA DE CÓDIGO FONTE (SCM).....	126
11.5.4	FERRAMENTAS DE AUTOMAÇÃO DE BUILDS.....	127
11.5.5	FERRAMENTAS DE INTEGRAÇÃO CONTÍNUA	127
11.5.6	FERRAMENTAS DE GESTÃO DE CONFIGURAÇÃO E PROVISIONAMENTO.....	127
11.5.7	FERRAMENTAS DE CONTEAINERIZAÇÃO	128
11.5.8	FERRAMENTAS DE GESTÃO DE REPOSITÓRIOS	128
11.5.9	FERRAMENTAS DE AUTOMAÇÃO DE TESTES	128
11.5.10	FERRAMENTAS DE TESTES DE PERFORMANCE, CARGA E ESTRESSE	129
11.5.11	FERRAMENTAS DE INJEÇÃO DE FALHAS	129
11.5.12	PLATAFORMAS DE NUVENS	129
	Bibliografia.....	131

1 Fundamentos de Arquiteturas Web

1.1 Introdução

Os sistemas Web surgiram em 1990 quando Tim Berners Lee fez a publicação do primeiro servidor Web, primeiro navegador e de um conjunto mínimo de páginas com hipertextos. Nos últimos vinte e sete anos a *World Wide Web* se popularizou e está presente na vida diária de bilhões de pessoas do mundo. Em janeiro de 2017, números publicados pelo portal NetCraft¹ indicavam a existência de 6 milhões de servidores Web servindo quase 2 bilhões de sítios Web. Na área de TI, por consequência, os sistemas Web se tornaram o padrão dominante para o desenvolvimento de aplicações corporativas.

Sistemas Web são baseados em princípios distintos de sistemas tradicionais como cliente servidor ou aplicativos móveis nativos. Resumo estes princípios abaixo.

1. As requisições a páginas Web não mantém estado. Essa propriedade, chamada de *stateless*, facilita o desenho de servidores Web pois os mesmos não precisam armazenar a conversação de cada um dos seus clientes. Ao mesmo tempo, essa característica traz desafios para o projetista em aplicações que precisam manter uma conversação com seus usuários finais.
2. Os documentos trafegados são baseados em estruturas de hipertextos e hipermídias. O efeito dessa diferença é que uma requisição a uma única página Web pode desencadear a requisições a dezenas de arquivos de texto, imagens ou vídeos.
3. Sistemas Web tem mais variabilidade na sua carga de trabalho do que sistemas convencionais cliente-servidor e isso pode trazer consequências de performance, escalabilidade e segurança para os seus usuários e administradores.
4. Sistemas Web podem ser desenvolvidos em um grande número de tecnologias, bibliotecas e frameworks. Saber analisar, comparar e escolher as tecnologias mais apropriadas é crítico para termos arquiteturas eficientes e robustas.

Arquitetar sistemas Web é uma tarefa desafiante e exige conhecimento especialista de sistemas, protocolos, padrões, tecnologias, ambientes de desenvolvimento, processos e até mesmo informações financeiras de produtos. E com o objetivo de facilitar este caminho para arquitetos e desenvolvedores, compilei aqui livro os princípios, práticas e tecnologias centrais que um especialista deveria conhecer, considerando o contexto do ano de 2017.

¹ <http://news.netcraft.com/>

1.2 Como navegar pelo livro

Este livro pode ser lido de forma linear pelo leitor. Ao mesmo tempo, ele também pode ser lido de forma fragmentada para você ter contato mais imediato e específico com um aspecto de interesse particular.

Capítulo 1 – Apresenta uma **arquitetura de referência** de sistemas Web, seus principais **protocolos e os princípios de desenho** de aplicações RESTful.

Capítulo 2 – Apresenta um **processo mínimo de desenvolvimento de arquiteturas Web**.

Capítulo 3 – Apresenta como identificar e capturar **requisitos arquiteturais Web**. Esses requisitos determinam a agenda de interesse técnico ao longo de um projeto.

Capítulo 4 – Apresenta os principais **estilos arquiteturais Web**, tais como MVC, MVVM, APIs, microserviços ou *serverless*. Apresentamos aqui suas vantagens, desvantagens e contextos de uso.

Capítulo 5 – Apresenta os principais tipos de tipos de **servidores Web e suas topologias físicas** com vantagens, desvantagens e contextos de uso.

Capítulo 6 – Apresenta as principais **tecnologias Java EE Web**, considerações de desenho em aplicações Java, riscos e oportunidades.

Capítulo 7 – Apresenta as principais **tecnologias ASP.NET**, considerações de desenho em aplicações .NET, riscos e oportunidades.

Capítulo 8 – Apresenta as principais **tecnologias JavaScript** e o uso de arquiteturas híbridas que combinem frameworks cliente JavaScript com serviços em servidores .NET, Java EE e PHP.

Capítulo 9 – Apresenta um panorama geral de **tecnologias móveis**, com foco no consumo de serviços Web realizado em servidores Node.JS, .NET, Java EE, PHP ou Python.

Capítulo 10 – Apresenta um **caso prático** de como um arquiteto poderia documentar esse caso com as técnicas de arquitetura e tecnologias apresentadas nesse livro.

Capítulo 11 – Apresenta como uma arquitetura de software pode ser ter a sua implementação acelerada com práticas e ferramentas **DevOps**.

1.3 Arquitetura de Referência de um Sistema Web

Programe você em ASP.NET, PHP, Java ou outra tecnologia, o esquema de um sistema Web é similar. A isso chamamos de modelo de referência. A figura apresenta um modelo para sistemas de informação Web.

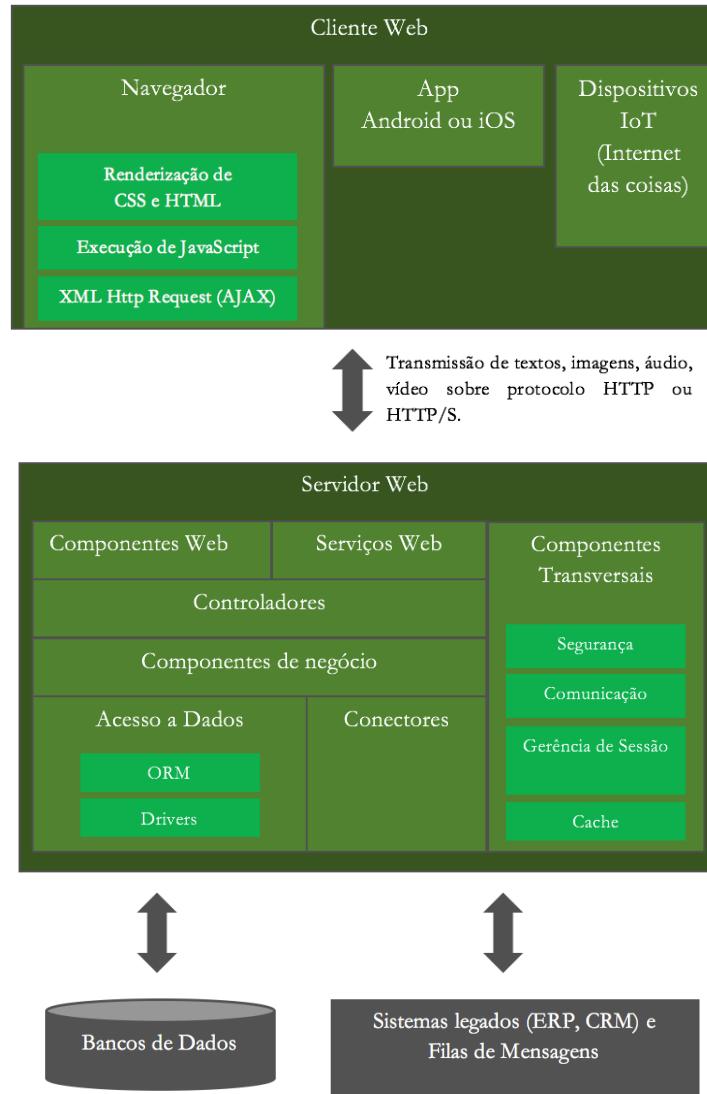


Figura 1: Arquitetura de referência de um sistema Web

1.4 O Protocolo HTTP

HTTP é um protocolo que permite a obtenção de recursos, tais como documentos HTML. É a base de qualquer troca de dados na Web e um protocolo cliente-servidor, o que significa que as requisições são iniciadas pelo destinatário, geralmente um navegador da Web. Um documento completo é reconstruído a partir dos diferentes subdocumentos obtidos, como por exemplo texto, descrição do layout, imagens, vídeos, scripts e muito mais.

Esse protocolo é tão fundamental que podemos definir o que é um servidor Web a partir dele.

Definição: Um software servidor Web é qualquer aplicação que suporte requisições baseadas no protocolo HTTP.

Projetado no início da década de 1990, o HTTP é um protocolo extensível que evoluiu ao longo do tempo. É um protocolo de camada de aplicação que é enviado sobre TCP, ou em uma conexão TCP criptografada com TLS, embora qualquer protocolo de transporte confiável possa, teoricamente, ser usado. Devido à sua extensibilidade, ele é usado para não apenas buscar documentos de hipertexto, mas também imagens e vídeos ou publicar conteúdo em servidores, como nos resultados de formulário HTML.

A implicação dessa definição é que a operação de um servidor Web pode ser operacionalizada em poucas linhas de código. A mostra um código Node.js que implementa um servidor Web mínimo que ecoa as mensagens que recebe de seus navegadores.

```

1 // Carga de modulos em Javascript. Aqui estou carregando o modulo HTTP, que dar
2 //suporte a requisicoes em protocolo HTTP
3 var http = require('http');
4
5 // Nesta linha eu crio um servidor Web, que irá ouvir requisições no porto 8080.
6 // Toda conexao a esse servidor será tratado por uma função criada pelo desenvolvedor.
7 // Esta função gera um cabeçalho HTTP 200 (OK) e então ecoa uma mensagem para
8 //o navegador que originou a requisição.
9 http.createServer(function (req, res) {
10   res.writeHead(200, {'Content-Type': 'text/plain'});
11   res.end('Meu primeiro programa Node.js!');
12 }).listen(8080);
13
14 // Nesta linha eu faço um log no console do servidor onde o Node.js foi iniciado
15 console.log('Servidor rodando em http://localhost:8080/');

```

Figura 2: Código de um servidor Web mínimo.

O protocolo HTTP opera sobre o protocolo TCP-IP e é baseado no conceito de objetos e métodos. Um objeto denota o recurso a ser endereçado dentro de um servidor Web. Exemplos de objetos podem ser páginas HTML, XML ou imagens. Uma requisição HTTP começa com uma solicitação do navegador no cliente e uma resposta do servidor. A requisição cliente é enviada em um formato semelhante ao mostrado na linha 1 da figura 3. E a resposta do servidor Web é mostrada nas listas 2-10.

```

1 GET /index.html HTTP/1.1
2 Host: www.exemplo.com
3 HTTP/1.1 200 OK
4 Date: Mon, 23 May 2016 22:38:34 GMT
5 Server: Apache/1.3.27 (Unix) (Red-Hat/Linux)
6 Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
7 Etag: "3f80f-1b6-3e1cb03b"
8 Accept-Ranges: bytesContent-Length: 438
9 Connection: close
10 Content-Type: text/html; charset=UTF-8

```

Figura 3: Invocação de um método GET em um servidor Web

O protocolo HTTP possui diversos métodos, sendo o GET o método mais comum de manipulação de objetos. No exemplo mostrado, ele é usado para solicitar uma página chamada index.html. O servidor recebe esse pedido e

devolve o objeto ao cliente, precedido por metadados como por exemplo o código da resposta ou a data e hora da resposta.

Outros métodos HTTP comuns são:

- HEAD: Similar ao GET, mas não retorna o recurso solicitado. O seu uso é apenas para obter metadados de objetos do servidor para fins diversos como por exemplo a indexação de servidores por robôs;
- POST: Usado para enviar dados para processamento pelo servidor. É o método mais usado para submeter dados de formulários HTML e pode ser usado para submeter textos e anexos binários diversos;
- PUT: Edita as informações de um determinado recurso Web;
- DELETE: Remover um determinado recurso Web.

A versão atual do protocolo HTTP, a 1.1, ainda prevê outros métodos mais específicos como o TRACE, OPTIONS e CONNECT usado para ecoar o pedido, saber que métodos um servidor Web específico aceita e para o estabelecimento de túneis seguros SSL.

Os dados trafegados sobre HTTP operam em canal aberto, sem confidencialidade e integridade. No entanto, existem extensões a esse protocolo que adicionam o suporte a transporte seguro. A extensão HTTP/S é a mais usada em servidores Web que desejem estabelecer conversações seguras. Essa extensão utiliza criptografia assimétrica baseada em protocolos como por exemplo o RSA, onde o número de bits da chave de criptografia determina o nível de robustez associada à mensagem. Embora quanto maior o número de bits da chave de criptografia, melhor se torna a proteção, existe também um efeito sobre o desempenho do servidor devido ao processo de criptografia de cada mensagem trocada entre o cliente e o servidor Web. Arquitetos devem prover um balanço apropriado entre as necessidades de performance e segurança através da seleção apropriada da chave de criptografia e o seu número de bits. O protocolo HTTP é mantido pelo *World Wide Consortium (W3C)* e a sua documentação oficial é mantida na especificação IETF RFC 7231².

1.5 Métodos HTTP

O protocolo HTTP define um conjunto de **métodos de requisição** responsáveis por indicar a ação a ser executada para um dado recurso. Embora esses métodos possam ser descritos como substantivos, eles também são comumente referenciados como **HTTP Verbs (Verbos HTTP)**. Cada um deles implementa uma semântica diferente, mas alguns recursos são compartilhados por um grupo deles, como por exemplo, qualquer método de requisição pode ser do tipo safe, idempotent ou cacheable.

GET

O método GET solicita a representação de um recurso específico. Requisições utilizando o método GET devem retornar apenas dados.

HEAD

O método HEAD solicita uma resposta de forma idêntica ao método GET, porém sem conter o corpo da resposta.

POST

O método POST é utilizado para submeter uma entidade a um recurso específico, frequentemente causando uma mudança no estado do recurso ou efeitos colaterais no servidor.

PUT

O método PUT substitui todas as atuais representações do recurso de destino pela carga de dados da requisição.

² <https://tools.ietf.org/html/rfc7231>

DELETE

O método DELETE remove um recurso específico.

CONNECT

O método CONNECT estabelece um túnel para o servidor identificado pelo recurso de destino.

OPTIONS

O método OPTIONS é usado para descrever as opções de comunicação com o recurso de destino.

TRACE

O método TRACE executa um teste de chamada *loop-back* junto com o caminho para o recurso de destino.

PATCH

O método PATCH é utilizado para aplicar modificações parciais em um recurso.

A documentação original dos métodos é trazida aqui - <https://tools.ietf.org/html/rfc7231#section-4>.

1.6 Segurança, Cache e Idempotência nos Métodos HTTP

Os métodos de solicitação são considerados "seguros" se sua semântica é de leitura apenas. Ou seja, o cliente não solicita e não espera qualquer alteração de estado no servidor Web como resultado da aplicação de um método seguro a um recurso de destino. Da mesma forma, não se espera que o uso de um método seguro cause qualquer dano e perda de propriedade e aumentos no servidor de origem. Dentro do protocolo HTTP, os métodos GET, HEAD, OPTIONS e TRACE são considerados seguros.

Os métodos de solicitação HTTP são considerados "cacheáveis" se a sua semântica permite indicar que suas respostas possam ser armazenadas para reuso futuro. O objetivo desta propriedade é permitir a escalabilidade de servidores Web. Dentro da especificação HTTP, os métodos GET, HEAD e POST são "cacheáveis".

Uma propriedade importante nos métodos de solicitação HTTP é a idempotência. Um método é chamado de idempotente se o efeito pretendido no servidor de múltiplas solicitações idênticas com esse método é o mesmo que o efeito para uma única solicitação desse tipo. Dentro da especificação HTTP, os métodos "seguros" e os métodos PUT e DELETE apresentam esta propriedade. A idempotência é importante na construção de uma API tolerante a falhas. Suponha que um cliente Web deseja atualizar um recurso por meio do POST. Como o POST não é um método idempotente chamá-lo várias vezes pode resultar em atualizações erradas. O que aconteceria se você enviou o pedido POST para o servidor, mas você obtém um tempo limite. O recurso está atualizado? O tempo limite ocorreu durante o envio do pedido para o servidor, ou a resposta ao cliente? Podemos repetir com segurança ou precisamos descobrir primeiro o que aconteceu com o recurso? Ao usar métodos idempotentes, como por exemplo o PUT, não precisamos responder a essa pergunta. Podemos reenviar a solicitação de forma segura até que possamos obter uma resposta do servidor.

Para facilitar a referência, a tabela abaixo fornece uma comparação das propriedades para os métodos HTTP mais comuns.

Método HTTP	Seguro	Suporta cache	Idempotente
GET	Sim	Sim	Sim
PUT	Não	Não	Sim
POST	Não	Sim	Não
DELETE	Não	Não	Sim

Tabela 1: Propriedades de métodos HTTP (conforme RFC 7231)

1.7 Requisições HTTP

Requisições HTTP são mensagens enviadas pelo cliente para iniciar uma ação no servidor. Suas linhas iniciais contêm três elementos:

1. Um *método TTP*, um verbo (como [GET](#), [PUT](#) ou [POST](#)) ou um nome (como [HEAD](#) ou [OPTIONS](#)), que descrevem a ação a ser executada. Por exemplo, GET indica que um recurso deve ser obtido ou POST significa que dados são inseridos no servidor (criando ou modificando um recurso, ou gerando um documento temporário para mandar de volta).
2. O *alvo da requisição*, normalmente um [URL](#), ou o caminho absoluto do protocolo, porta e domínio são em geral caracterizados pelo contexto da requisição. O formato deste alvo varia conforme o método HTTP. Pode ser
 - Um caminho absoluto, seguido de um '?' e o texto da consulta. Esta é a forma mais comum, conhecida como a *forma original*, e é usada com os métodos GET, POST, HEAD, e OPTIONS.
POST / HTTP/1.1
GET /background.png HTTP/1.0
HEAD /test.html?query=alibaba HTTP/1.1
OPTIONS /anypage.html HTTP/1.0
 - Uma URL completa, conhecida como a *forma absoluta*, usada principalmente com GET quando conectado a um proxy.
GET http://developer.mozilla.org/en-US/docs/Web/HTTP/Messages HTTP/1.1
 - O componente autoridade de um URL, que consiste no nome do domínio e opcionalmente uma porta (prefixada por ':'), chamada de *forma autoridade*. Só usada com CONNECT ao estabelecer um túnel HTTP.
CONNECT developer.mozilla.org:80 HTTP/1.1
 - A *forma asterisco*, um simples asterisco ('*'), usada com OPTIONS. Representa o servidor como um todo.
OPTIONS * HTTP/1.1
3. A *versão HTTP*, que define a estrutura do restante da mensagem, atuando como um indicador da versão esperada para uso na resposta.

1.8 HTTP/2

O HTTP 2 é baseado no protocolo SPDY, um protocolo introduzido pelo Google em 2009 e adotado por algumas tecnologias incluindo o próprio navegador do Google Chrome, Mozilla Firefox, Internet Explorer da Microsoft, muitos sites como o Facebook, e alguns dos softwares que fornece páginas da Web para browsers. O SPDY foi projetado para acelerar o carregamento de páginas da web e a experiência de navegação dos usuários online.

O SPDY e o HTTP2 utilizam “header field compression” e “multiplexing” para permitir que os navegadores possam fazer várias solicitações para servidores web através de uma única conexão. O novo protocolo usa o “multiplexing” para permitir que muitas mensagens sejam intercaladas em uma conexão ao mesmo tempo, de modo que as respostas que levam um longo tempo para o servidor carregar não bloqueie as outras que estão sendo executadas paralelamente. O HTTP/2 possibilita um uso mais eficiente dos recursos de rede e uma percepção reduzida da latência ao incluir a compressão dos campos de cabeçalho e permitir diversas trocas simultâneas na mesma conexão. Especificamente, ele permite intercalar mensagens de solicitação e resposta na mesma conexão e usa uma codificação eficiente para os campos de cabeçalho HTTP. Além disso, oferece priorização de solicitações, permitindo que as mais importantes sejam concluídas mais rapidamente, o que melhora ainda mais o desempenho.

No centro de todas as melhorias de desempenho do HTTP/2 está a nova camada de frame binário, que determina como as mensagens HTTP são encapsuladas e transferidas entre cliente e servidor.

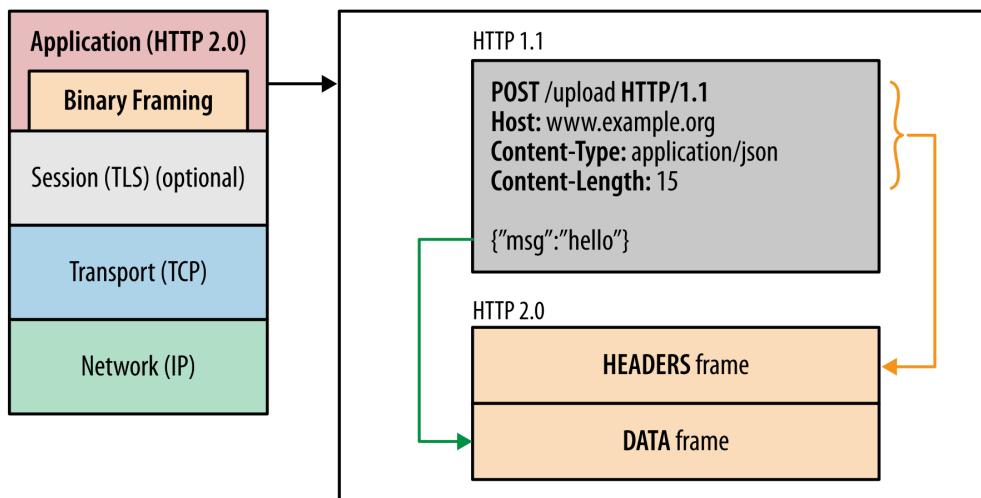


Figura 4: Operação do HTTP 2

O conceito de "Camada" se refere a uma escolha de projeto usada para introduzir um novo mecanismo de codificação otimizado entre a interface do soquete e a API de HTTP mais importante exposta aos aplicativos: a semântica HTTP, como verbos, métodos e cabeçalhos, não mudou, mas como ela é codificada durante o trânsito, sim. Diferentemente do protocolo HTTP/1.x de texto simples delimitado por linha nova, toda a comunicação do HTTP/2 é dividida em mensagens e frames menores, e cada um deles é codificado em formato binário.

Como resultado, o cliente e o servidor devem usar o novo mecanismo de codificação binária para entendimento mútuo: um cliente HTTP/1.x não consegue entender um servidor exclusivamente HTTP/2, e vice-versa. Por sorte,

nossos aplicativos continuam operando felizes e sem saber de todas essas alterações, já que o cliente e o servidor realizam todo o trabalho de frame necessário para nós.

Com o novo mecanismo de frame binário, o HTTP/2 não precisa mais de diversas conexões TCP para multiplexar streams em paralelo. Cada stream é dividido em vários frames que podem ser intercalados e priorizados. Isso faz com que todas as conexões HTTP/2 sejam persistentes e somente se exija uma conexão por origem, o que oferece inúmeros benefícios de desempenho. A maioria das transferências HTTP é curta e súbita, enquanto que o TCP é otimizado para transferências de dados em lote e de longa duração. Ao reutilizar a conexão, o HTTP/2 pode fazer o uso mais eficiente de cada conexão TCP, além de reduzir significativamente a sobrecarga geral do protocolo. Além disso, o uso de menos conexões reduz a área de ocupação de memória e de processamento ao longo de todo o caminho da conexão (ou seja, cliente, intermediários e servidores de origem). Isso reduz o custo operacional total e aumenta a utilização e a capacidade da rede. Como consequência, a mudança para HTTP/2 não deve apenas reduzir a latência de rede, mas também ajudar a aumentar a produtividade e diminuir os custos operacionais.

Outra modificação é relacionada à segurança através da substituição da tecnologia de criptografia TLS (Transport Layer Security, anteriormente chamado SSL Secure Sockets) em HTTP/2.

1.9 A Nomeação de Recursos na Web - URIs

Um identificador uniforme de recursos, ou URI, é usado para identificar objetos e sítios Web. Uma URI pode ser um localizador URL, usado para designar um endereço de um sítio Web como por exemplo <http://www.google.com>. Uma URI também pode ser uma URN, usada para designar um objeto dentro de um sítio Web, como por exemplo /imagens/logo.gif. Requisições HTTP no cliente usam o esquema de URIs para denotar como acessar recursos no servidor. Uma aplicação prática dos métodos HTTP e a nomeação de recursos URI reside no estilo Web chamado REST, descrito no capítulo 4. A especificação de URIs é armazenada no na especificação IETF RFC 3986³.

A linguagem de marcação de hipertextos, ou HTML, é a linguagem central usada para criar páginas em servidores Web. Ela permite organizar textos, imagens ou mesmo outras mídias como sons e vídeos. Páginas HTML são armazenadas dentro do servidor Web, mas são processadas dentro do navegador cliente. Os tempos necessários para baixar o arquivo HTML processar a estrutura de árvore desse arquivo são os dominantes na arquitetura de sistemas Web.

A especificação HTML é mantida pelo W3C e a versão 5.0 lançada em 2014 trouxe muitas novidades tais como:

- Controle nativo de objetos multimídia como vídeos;
- API para renderização de gráficos bidimensionais;
- Aprimoramento do uso off-line com cache de objetos;
- Facilidades para depuração de erros.

Dica: O HTML 5 tornou obsoletas tecnologias como o Adobe Flash e Java Applets. Nessa direção, desde 2015 o Google Chrome não suporta mais Java Applets. O Adobe Flash caiu em desuso nos últimos anos e Oracle (mantenedora do Java) anunciou o fim do suporte a Applets Java a partir do JDK 9.

O sítio do W3C mantém as informações oficiais sobre o HTML ⁴. O anúncio do fim de suporte a Applets é encontrado aqui ⁵.

³ <https://tools.ietf.org/html/rfc3986>

⁴ <http://www.w3.org/TR/html5/>

⁵ Anúncio realizado pela Oracle do fim do suporte a Applets a partir do JDK 9, previsto para 2017 - https://blogs.oracle.com/java-platform-group/entry/moving_to_a_plugin_free

1.10 A linguagem XML

A linguagem de marcação extensível, ou XML, é uma linguagem de marcação para estruturação, armazenamento e troca de informações. Ela é usada para armazenar e trocar informações em ambientes Web. Devido aos seus mecanismos de validação estrutural (DTD), ele permite maior segurança sobre o formato dos dados enviados. Devido à flexibilidade para composição de novos esquemas de dados, ele é ainda usado para estruturar protocolos de dados na interoperabilidade entre organizações. No Brasil, vemos uma grande adoção do XML no setor público e no segmento bancário.

O XML é mantido pela W3C e as suas informações oficiais são mantidas aqui⁶.

1.11 A notação JSON

A notação de objetos JavaScript, ou JSON, é um formato leve para o armazenamento e troca de informações em ambiente Web. Embora o JSON seja um subconjunto da notação de objeto JavaScript, ele opera de forma independente de implementações JavaScript. O JSON é um padrão descrito no IETF no RFC 4627⁷.

Nos últimos anos, o JSON ganhou grande popularidade na construção de sistemas Web por ser mais simples de interpretar e manipular do que arquivos XML. Uma outra vantagem é o seu tamanho reduzido, se comparado a XMLs, uma vez que seus arquivos não carregam *tags*.

Dica: Arquivos JSON utilizam 30% menos de banda de passagem que arquivos XML. Isso o leva a ser o formato mais recomendável para a maior parte dos requisitos de troca de informações em aplicações Web.

1.12 Folhas de Estilo em Cascata - CSS

As folhas de estilo em cascata, ou CSS, é uma linguagem para definir a apresentação de documentos escritos em HTML, XML ou JSON. Ela permite desacoplar a informação de um documento do seu formato. Um exemplo prático do uso de CSS são páginas responsivas, que podem ser construídas uma única vez e exibidas em computadores, *tablets* ou *smartphones* de diferentes resoluções.

O CSS é mantido pelo W3C e a sua última especificação, a 3.0, trouxe um conjunto rico de novidades que agregam na experiência de uso e riqueza visual de aplicações Web. Em conjunto com especificações ricas de usabilidade Web, como o Google Material Design⁸ ou o Microsoft UWP⁹, eles permitem criar páginas modernas e ricas baseadas no padrão de usabilidade *Single Page Application (SPA)*.

Dica: Use aceleradores para trabalhar com CSS. Um primeiro acelerador é uso da metodologia blocos, elementos e modificadores CSS chamada BEM – <http://getbem.com/introduction/>. Um segundo acelerador é uso de bibliotecas CSS. Exemplos dessas bibliotecas incluem o LESS, Foundation, Bootstrap ou o MaterializeCSS.

1.13 A Linguagem JavaScript

O JavaScript surgiu em 1996 como uma linguagem usada para fornecer interatividade a páginas. Ela permite manipular o HTML e o CSS e assim criar interações mais ricas com os usuários dessas páginas. Um exemplo simples do uso do JavaScript ocorre na página principal de pesquisa do Google. À medida que o usuário começa a digitar

⁶ <http://www.w3schools.com/xml/>.

⁷ <http://www.w3schools.com/xml/>.

⁸ <https://design.google.com>

⁹ <https://developer.microsoft.com/en-us/windows/design>

um termo de pesquisa na caixa de texto, o navegador mostra os resultados mais prováveis daquele termo em uma cortina dinâmica logo abaixo dessa caixa.

Quando um navegador Web acessa e baixa uma página HTML, ele monta um objeto interno que é a representação em árvore dessa página. Essa estrutura de dados, chamada DOM (Modelo de Objeto de Documento), pode ser manipulada de forma dinâmica através da linguagem chamada JavaScript.

Embora o uso de JavaScript seja obrigatório em qualquer página, o seu uso traz algumas sutilezas. A primeira é que a execução de código trava a construção de um objeto DOM. Portanto, se um desenvolvedor Web cria uma página cujo JavaScript usa muita CPU ou está aguardando pelo retorno de uma resposta servidora, a página trava ou se torna lenta para uso. Isso cria um efeito de usabilidade ruim. Isso pode ser evitado através do uso de chamadas assíncronas, que é o padrão dominante de uso em JavaScript. Todos os navegadores possuem o suporte a um objeto XML XMLHttpRequest, que permite a execução assíncrona de requisições JavaScript. O termo AJAX significa o uso assíncrono de JavaScript e XML e é, na prática, habilitado pela tecnologia XHR – XML XMLHttpRequest.

A linguagem JavaScript não deve ser confundida com a linguagem Java. Embora a sua especificação original lançada pela Netscape em 1996 tenha se inspirada na linguagem Java, ela seguiu um curso distinto ao longo dos últimos 20 anos. A linguagem JavaScript se chama ECMAScript e é padronizada pelo instituto europeu ECMA¹⁰. A sua última especificação, chamada de JavaScript 7 é também conhecida como ES7 ou ECMA Script 2016.

Nos últimos anos, a linguagem JavaScript cresceu e se popularizou através de um rico conjunto de aceleradores. Exemplos incluem:

- Bibliotecas de componentes gráficos tais como o jQuery, ExtJS ou Telerik;
- Frameworks MV* de organização de eventos cliente e interações com o servidor tais como o AngularJS, Ember, Backbone, AureliaJS ou Meteor¹¹.
- Linguagens como ClojureScript, TypeScript e CoffeScript e transpiladores que compilam o código dessas linguagens em código JavaScript ES5 ou ES6;
- Bibliotecas para execução de código JavaScript em ambiente servidor como o Node.js, Express, Koa, Meteor ou Sails.js.
- Bibliotecas gerenciamento de pacotes como o Bower e npm.
- Bibliotecas de automação de testes como o Mocha, Jasmine, Karma, Selenium ou Phantom.js.
- Bibliotecas de automação de tarefas como o Grunt, Gulp, Webpack ou Sourcemap.

Dica: A versão 6 da linguagem JavaScript (ES6 ou ECMAScript 2015) trouxe muitas melhorias na linguagem. Em novos projetos Web, busque usar esta versão da linguagem.

1.14 Desafios Arquiteturais Web

A massiva expansão da Web nas últimas duas décadas culminou em um movimento de convergência global chamado de *Nexus das Forças* pelo Gartner Group e de *Plataforma Aberta 3.0* pelo *Open Group* (THE OPEN GROUP, 2014). Este movimento é descrito na **Figura 4**.

¹⁰ <http://www.ecmascript.org>

¹¹ MV* é um termo usado para indicar padrões arquiteturais como o MVC, MVVM ou MVP. O site <http://todomvc.com> apresenta comparativos de frameworks Javascript MV*.

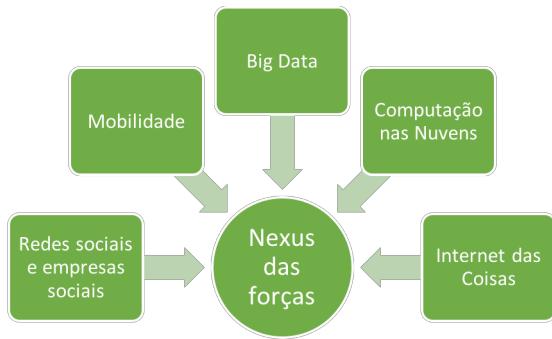


Figura 4: Nexus das Forças

Estas forças convergentes estão redefinindo o mundo como conhecemos, habilitando novos modelos de negócio para empresas tradicionais e governos em todas as suas esferas, criando empresas digitais e ofertando um modelo de consumo em escala global como jamais foi observado na história da humanidade. Este modelo de convergência nos fornece uma nova plataforma arquitetural, descrita na **Figura 5**, que deve responder em 2020 por 40% das receitas da indústria de TI e telecomunicações.

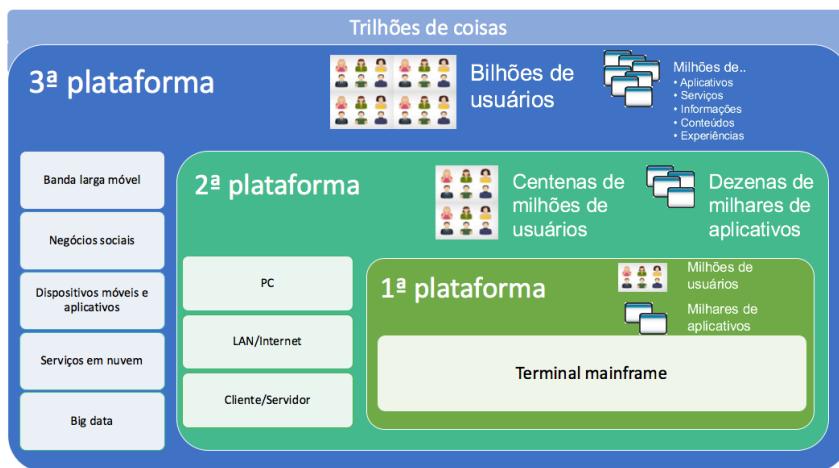


Figura 5: Terceira Plataforma para Crescimento e Inovação, IDC.

O Nexus das Forças, apesar de fascinante, traz enormes desafios sobre a comunidade técnica de TI na evolução dos padrões Web já existentes e na criação de novos padrões que suportem trilhões de dispositivos (internet das coisas) sendo operados por bilhões de pessoas em bilhões de interações sociais e comerciais em base diária. Não por acaso, os principais grupos de trabalho do W3C estão orientados ao longo destas novas plataformas. Podemos destacar, neste contexto, as seguintes linhas de governança no W3C.

- **Desenho de Aplicações Web** - Envolve as normas para a construção e renderização de páginas da Web, incluindo HTML, CSS, AJAX, SVG e outras tecnologias tais como *Web Apps* em dispositivos móveis. Esta linha também inclui informações sobre como tornar as páginas acessíveis a pessoas com qualquer tipo deficiência visual (WCAG), a internacionalização de páginas e desenhos responsivos e fluidos.
- **Arquiteturas Web** – Esta linha se concentra nas tecnologias de base e os princípios que sustentam a Web, incluindo URIs e HTTP. Um resultado concreto deste trabalho é o interesse ativo da comunidade no padrão REST para a interoperabilidade de aplicações e o trabalho conjunto da especificação do HTTP 2.0.
- **Web Semântica** – Esta linha está ajudando a construir um conjunto de tecnologias para apoiar a *Web de Dados*, que é uma evolução sobre a tradicional *Web de Documentos* presente nos dias atuais. O objetivo final da “Web de dados” é permitir que os computadores possam apoiar interações confiáveis na rede e o suporte a decisão a seres humanos. O termo “Web Semântica” refere-se à visão do W3C da Web dos dados

vinculados. As tecnologias da Web Semântica permitem às pessoas criar repositórios de dados na Web, construir vocabulários controlados, ontologias e escrever regras para manipulação de dados. Dados vinculados são habilitados por tecnologias como RDF, SPARQL, OWL e SKOS, entre outras.

- **Web de Serviços** – Esta linha refere-se ao desenho baseado em mensagens encontrado na Web e em softwares corporativos. A rede de serviços é baseada em tecnologias como HTTP, XML, SOAP, WSDL, SPARQL e outros.
- **Web de Dispositivos** - Esta linha se concentra em tecnologias para permitir o acesso à Web em qualquer lugar, a qualquer hora, usando qualquer dispositivo. Isso inclui o acesso à Web a partir de telefones celulares e outros dispositivos móveis, bem como o uso da tecnologia Web em eletrônicos de consumo, impressoras 2D e 3D, televisão interativa e carros conectados.

1.15 O Papel das Arquiteturas e Arquitetos Web

Empresas e órgãos do governo estão desenvolvendo novas aplicações Web. Nesse processo de desenvolvimento, algumas questões de conhecimento que surgem incluem:

- Qual a agenda técnica trazida pelos principais condutores de negócio de um determinado projeto ou produto?
- Como identificar, avaliar e selecionar tecnologias dentro do contexto de um projeto?
- Como compor padrões, tecnologias, bibliotecas, frameworks, componentes e afins dentro de linhas de base tecnológicas confiáveis?
- Como usar com segurança e produtividade as tecnologias do ecossistema Web em projetos?
- Como apoiar times de projeto para garantir um uso correto e produtivo das tecnologias selecionadas para uso?
- Como ter produtividade e retornos financeiros no uso dessas tecnologias Web?

Essas perguntas fazem parte da agenda do arquiteto de aplicações Web. Através de um processo leve e disciplinado de trabalho, o arquiteto Web lidera o trabalho de seleção e uso de tecnologias e busca responder a essas questões de conhecimento através de artefatos de desenho e implementação. Os capítulos seguintes são organizados ao longo dessa agenda de trabalho do arquiteto, incluindo princípios (capítulo 2), práticas (capítulos 3-5) e tecnologias (capítulos 6-8).

2 Como Arquitetar Sistemas Web

2.1 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, Node.JS)?
- 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.

2.2 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:

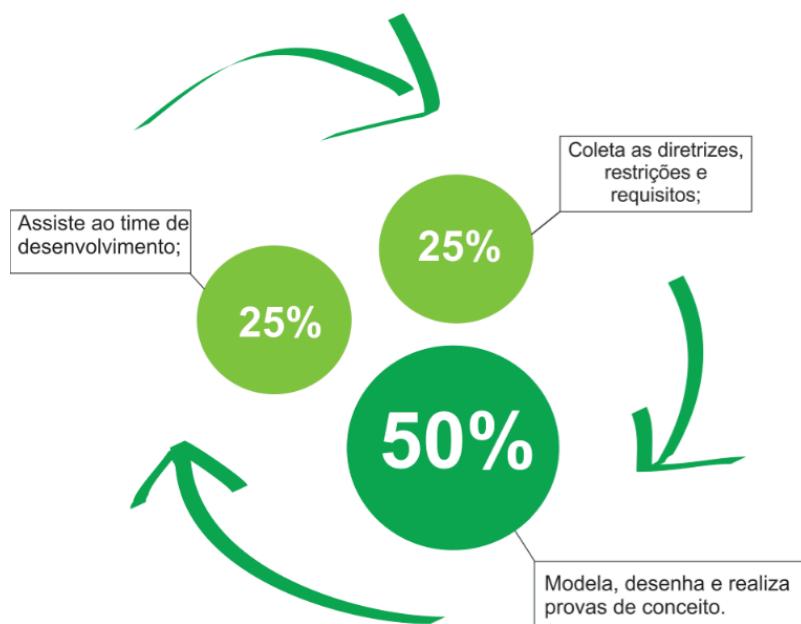


Figura 6: A agenda de um arquiteto de aplicações Web.

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.

2.3 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. O estilo é a **principal decisão arquitetural** de um projeto e a plataforma é a **principal decisão técnica** de um projeto. O Capítulo 4 apresenta os estilos arquiteturais Web mais comuns. Os Capítulos 6, 7 e 8 apresentam algumas plataformas técnicas Web e o capítulo 5 apresenta como organizar a topologia física dessa plataforma técnica.

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 7.

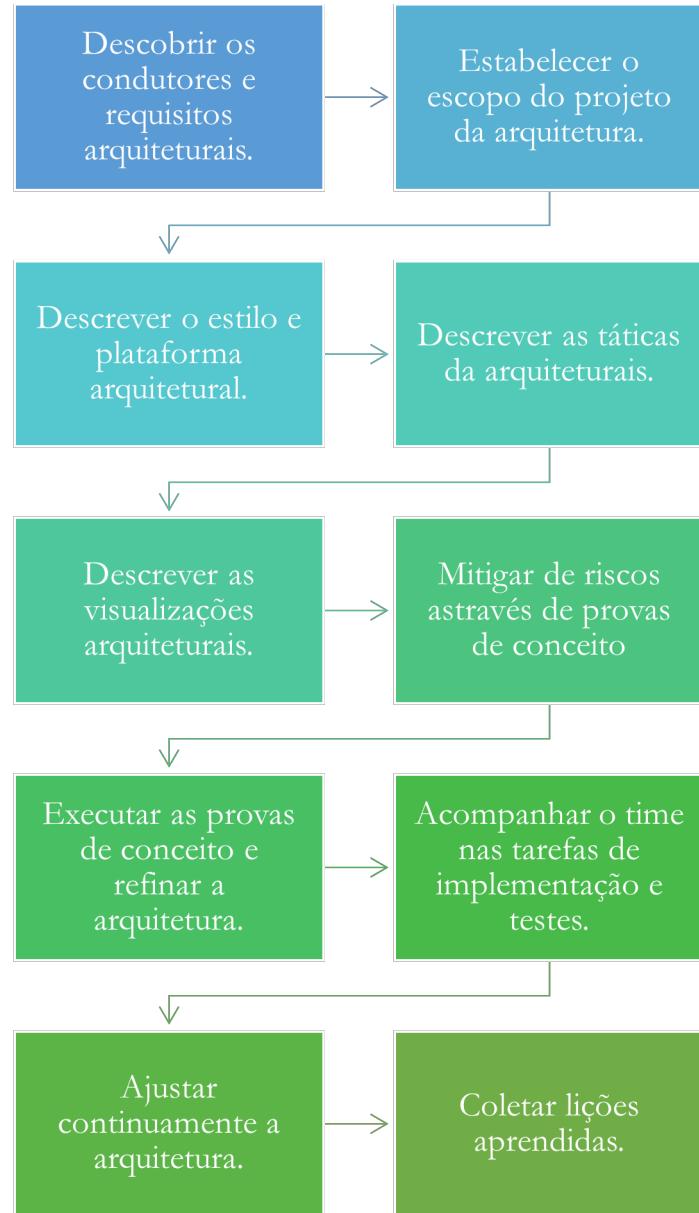


Figura 7: Um processo mínimo para desenvolver arquiteturas Web.

2.4 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 negócio 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 no 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).

2.5 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.

3 Requisitos Arquiteturais Web

3.1 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 8 para definir a agenda de arquitetura.

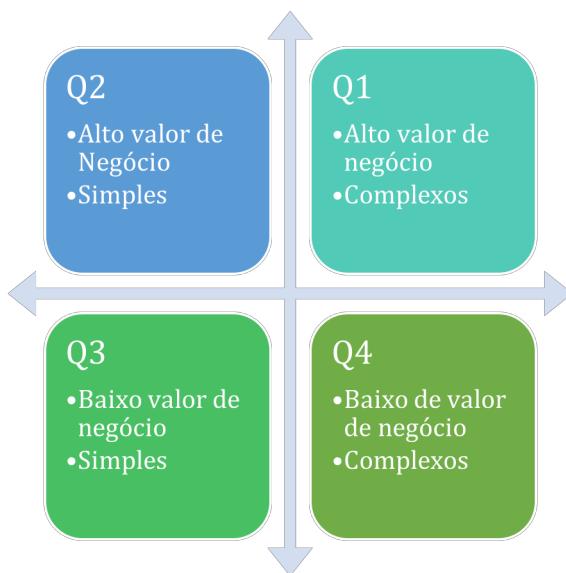


Figura 8: Tipos de requisitos em sistemas Web.

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.

3.2 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%, estaríamos 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).”

3.3 Requisitos de Acessibilidade e Usabilidade

Requisitos ligados a 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 2.0¹².

O governo federal brasileiro desenvolveu uma iniciativa similar, chamada de 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 –

¹² <https://www.w3.org/Translations/WCAG20-pt-PT/>

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 texto para torná-lo mais compreensível, 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.¹³

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 aqui¹⁴.

3.4 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 Web¹⁵
- Guia para autenticação em aplicações Web¹⁶
- Guia para autorização em aplicações Web¹⁷
- Guia de controle de acesso em aplicações Web¹⁸

3.5 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 assunto¹⁹.

¹⁴ <https://www.w3.org/WAI/ER/tools/>

¹⁵ https://www.owasp.org/index.php/Authentication_Cheat_Sheet

¹⁶ https://www.owasp.org/index.php/Guide_to_Authentication

¹⁷ https://www.owasp.org/index.php/Guide_to_Authorization

¹⁸ https://www.owasp.org/index.php/Access_Control_Cheat_Sheet

¹⁹ O site <https://drownattack.com> apresenta em detalhes um exemplo de vulnerabilidade encontrada no SSL2 e como se prevenir.

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 Web²⁰
- Dicas Práticas de criptografia Web²¹

3.6 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 segurança²². 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.

3.7 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 2.

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 2: Níveis de disponibilidade em aplicações Web

²⁰ https://www.owasp.org/index.php/Guide_to_Cryptography

²¹ https://www.owasp.org/index.php/Cryptographic_Storage_Cheat_Sheet

²² 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>

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.

3.8 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 3.

Nível	SLA de Recuperabilidade	Recomendação de Adoção
1	Até 1 dia	Aplicações de Intranet/escritório.
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 3: Níveis de recuperabilidade em aplicações Web

3.9 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 4.

Nível	SLA de Tempo de Resposta	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 4: Níveis de tempo de resposta em aplicações Web

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

3.10 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.

4 Estilos Arquiteturais Web

4.1 Introdução

Um estilo arquitetural é a primeira e grande decisão arquitetural realizada pelo time de arquitetura em um projeto. Um estilo arquitetural representa uma abordagem de desenho, implementação e distribuição de uma aplicação Web. Dentro de arquiteturas Web, existem diversos estilos arquiteturais. Cada um desses estilos é mais apropriado a um certo contexto e um arquiteto deve conseguir enumerá-los, compará-los e recomendá-los conforme o problema em sua mão.

Não existem estilos melhores ou piores em termos absolutos. O contexto é fundamental em arquitetura.

4.2 Web 1.0

Este é um estilo legado, que foi dominante na primeira década Web - 1990 a 2000. Ele se caracteriza por formulários simples, com baixa interatividade, uso limitado de JavaScript e componentes visuais simples. Ele é ainda usado em aplicações cadastrais de Intranet ou por alguns geradores de código Web criados no começo do século.

Algumas das tecnologias de mercado que representaram bem esse estilo incluem:

- Microsoft ASP
- Microsoft ASP.NET WebForms
- Java EE JSP e Servlets
- PHP
- Ruby
- Python

Em 2017, não existem motivos para usar este estilo arquitetural que se tornou peça tecnológica legada.

4.3 Web 2.0

A partir de 2005, o AJAX se popularizou e a quantidade de código de JavaScript começou a aumentar dentro das aplicações Web. Isso trouxe um movimento de enriquecimento de componentes Web e produção de páginas mais ricas e interativas.

Este estilo é o dominante hoje na produção de portais corporativos e aplicações de *e-commerce* por toda a Internet. Algumas das tecnologias que representam esse estilo incluem:

- Microsoft ASP.NET MVC
- Java EE JSF (PrimeFaces, RichFaces, ZK)
- Cake PHP
- Ruby on Rails
- Django/Python

Essas tecnologias são usadas com bibliotecas JavaScript para prover componentes Web mais ricos e interativos. Um exemplo desse tipo de biblioteca é o jQuery.

Em termos de organização de camadas, observamos o uso do padrão arquitetural como MVC Web com padrão dominante.

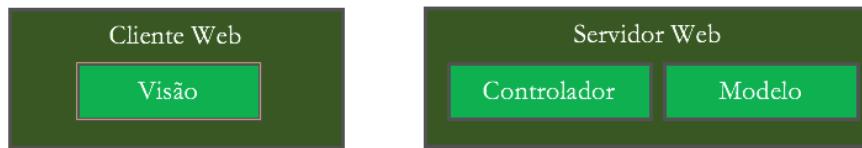


Figura 9: Padrão MVC Web em Arquiteturas Web.

4.4 SPA (*Single Page Application*)

A partir de 2010 os *Web Designers* começaram a trazer ainda mais inovação e riqueza nas suas páginas Web. Essa riqueza trouxe alguns desafios na programação Web cliente em termos de desempenho das suas aplicações. Aspectos que impactam o desempenho em páginas com muita interação incluem:

- Tempo de carga de uma árvore DOM para renderizar uma página HTML;
- Grande número de interações entre a camada de visão e a camada de controle, onde cada interação gera uma ou mais requisições HTTP.

Alguns projetistas Web propuseram algumas modificações no desenho dessas aplicações, que incluem:

- Manter uma única página física para toda aplicação e similar a navegação entre “telas” através da carga de partes de elementos DOM dentro do DOM da página central.
- Movimentar a execução do controlador do servidor para o cliente, reduzindo a quantidade de requisições HTTP entre o cliente e o servidor Web. Essa movimentação é batizada no dialeto como *MVVM (Model View ViewModel)* ou *MVP (Model View Presenter)*.

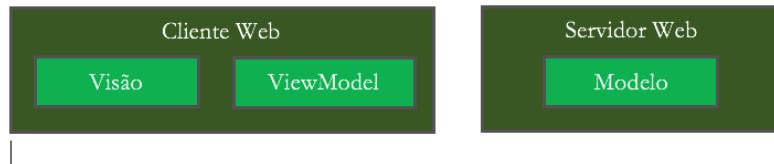


Figura 10: Padrão MVVM em Arquiteturas Web.

Portanto, sítios Web que usam apenas uma única página física e fazem a manipulação dinâmica de uma árvore DOM para simular a navegação são chamadas de SPA ou *Single Page Application*.

Algumas tecnologias que facilitam a implementação nesse modelo incluem:

- Google AngularJS
- Ember
- Facebook
- ReactJS
- Backbone
- AureliaJS

Uma consequência do uso deste tipo de estilo arquitetural é que o código controlador passa a ser todo escrito em JavaScript. O servidor Web se torna um servidor de serviços Web, seja em tecnologia .NET, Java ou até mesmo JavaScript servidor.

4.5 API Web RESTful

Muitas empresas foram desafiadas a expandir a sua presença Web em outros meios. O movimento de aplicações móveis é uma dessas forças. Com o movimento da internet das coisas (IoT), essa pressão continuará a aumentar nos próximos anos. Nesse sentido, algumas empresas têm realizado uma migração digital de portais Web para uma presença mais ampla. Isso é acompanhado através da criação de uma API de serviços Web.

O exemplo da **Figura 11**, extraído do portal de desenvolvedores CNova, mostra um exemplo de uma API de Lojista no tema de gerenciamento de pedidos.

orders : Recurso para gerenciamento de pedidos			
		Show/Hide	List Operations
		Expand Operations	
GET	/orders	Recupera detalhes de todos os pedidos	
POST	/orders	Operação para criar um pedido (SANDBOX)	
GET	/orders/status/approved	Recupera uma lista de pedidos Aprovados	
GET	/orders/status/canceled	Retorna uma lista de pedidos Cancelados	
GET	/orders/status/delivered	Recupera uma lista de pedidos Entregues	
GET	/orders/status/new	Recupera uma lista de pedidos Novos	
GET	/orders/status/partiallyDelivered	Retorna uma lista de pedidos Parcialmente Entregues	
GET	/orders/status/partiallySent	Retorna uma lista de pedidos Parcialmente Enviados	
GET	/orders/status/sent	Recupera uma lista de pedidos Enviados	
GET	/orders/lorderIdl	Recupera detalhes do pedido informado	
GET	/orders/lorderIdl/items/{skuSellerId}	Recupera detalhes de um item específico do pedido	
POST	/orders/lorderIdl/trackings/cancel	Operação para confirmar o cancelamento de um item de pedido	
POST	/orders/lorderIdl/trackings/delivered	Registra uma nova operação de tracking de entrega	
POST	/orders/lorderIdl/trackings/exchange	Operação para confirmar a troca de um item de um pedido	
POST	/orders/lorderIdl/trackings/return	Operação para confirmar devolução (reembolso) de item do pedido	
POST	/orders/lorderIdl/trackings/sent	Registra uma nova operação de tracking de envio	
PUT	/orders/status/approved/lorderIdl	Operação para realizar a aprovação de um pedido (SANDBOX)	

Figura 11: Exemplo de API Web RESTful

Esta API apresenta funcionalidades expostas através de métodos HTTP tais como GET, POST ou PUT e pode ser consumida em qualquer linguagem moderna, seja por aplicativos Web, dispositivos móveis ou dispositivos da Internet das coisas. Esta visão é representada na **Figura 12**.

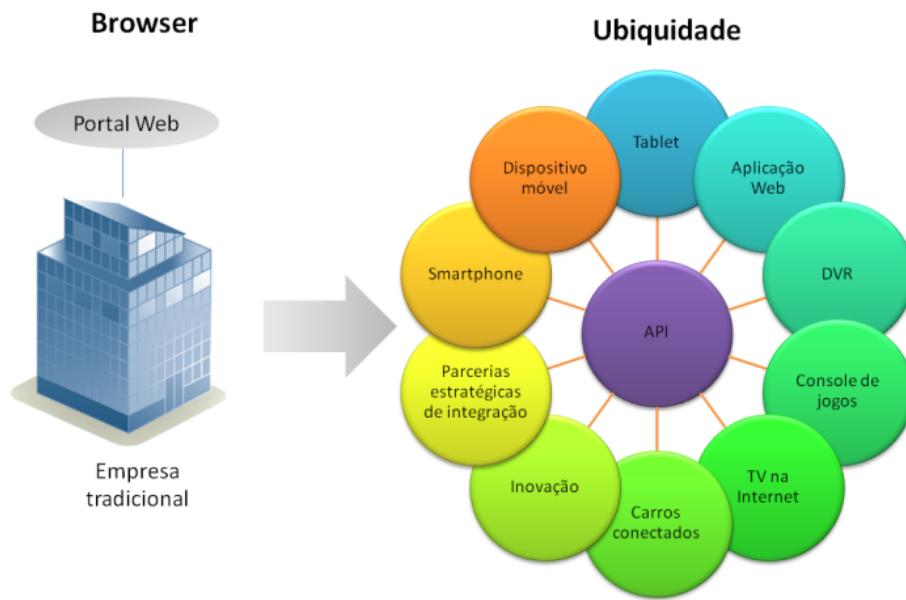


Figura 12: Estilo arquitetural de Web API

APIs modernas são criadas sobre o conceito de REST, que é um conjunto de princípios para expor serviços em sistemas distribuídos.

As origens do REST: O termo REST (*Representational State Transfer*) foi cunhado por Roy Fielding em 2000 na sua tese de doutorado. O autor, que foi um dos autores do protocolo HTTP e também cocriador do servidor Web Apache, criou um conjunto de princípios para orientar o bom desenho de sistemas distribuídos tais como performance, simplicidade, escalabilidade, portabilidade, confiabilidade e modificabilidade. Os princípios REST não tem relação obrigatória com HTTP, embora o mercado utilize o termo de forma mais livre do que na visão original do autor.

O REST, ou Transferência de Estado Representacional, se baseia no seguinte conjunto de princípios:

- Separação clara entre cliente e servidor. O servidor fornece acesso a um conjunto de serviços e ouve por requisições em portos pré-estabelecidos.
- Ausência de estado. Cada mensagem contém toda a informação necessária para compreender o pedido. Como resultado, nem o cliente e nem o servidor necessitam gravar nenhum estado das comunicações entre mensagens.
- Suporte a cache. Este princípio propõe que cada resposta a uma requisição possa ser marcada como cacheável (quando aplicável).
- Interface uniforme. A interface de acesso dos componentes do servidor deve ser uniforme. Esta propriedade permite que diferentes tipos de clientes possam interagir de forma facilitada com os componentes do servidor.
- Um sistema baseado em REST é baseado no conceito de recursos. Uma página Web, um vídeo ou uma imagem são exemplos de recursos. Um recurso define o tipo de informação que será transferida e as ações relacionadas. A estrutura de descrição de um recurso REST possui quatro propriedades:
 - Representação – Um ou mais formatos para representação da informação. Exemplos incluem XML, JSON ou formato binário.
 - Identificador – Uma URL que recupera um recurso específico em um dado momento.
 - Metadados – O tipo de conteúdo, a hora da última modificação, entre outros.
 - Dados de controle – Controle de cache, permissão de modificação, entre outros.

O protocolo HTTP e o padrão JSON podem ser usados para implementar os princípios REST. Quando estes princípios são observados temos um sistema ou API RESTful.

Os métodos HTTP permitem manipular recursos em servidores Web para operações de criação, atualização, remoção e pesquisa de informações. Os métodos HTTP GET e DELETE são intuitivos e são usados para as operações de leitura e remoção de recursos Web. Já os métodos HTTP POST e PUT merecem mais atenção. Em linhas gerais, usamos o método PUT para a atualização de um recurso específico e o método POST para a criação de um novo recurso. Considere o exemplo abaixo.

```

1  PUT /produto/1234 HTTP/1.1
2  <produto>
3    <titulo>Cerveja Backer Tommy Gun Double IPA</titulo>
4    <preco dinheiro="brl">9.50</preco>
5  </produto>
6
7  POST /produtos HTTP/1.1
8  <produto>
9    <titulo>Cerveja KSB Living The Dream Double IPA</titulo>
10   <preco dinheiro="brl">13.50</preco>
11  </produto>
12 HTTP 1.1 201 Created
13 Location: /produtos/1789

```

Figura 13: POST para criação de recurso e PUT para atualização

No primeiro exemplo, sabemos a localização do recurso (produto de código 1234). O servidor, ao receber a atualização, fará a atualização dos dados do produto 1234 com o conteúdo passado no corpo do documento. Já no segundo exemplo não sabemos a localização de um novo artigo, que ainda vai ser criado. E assim usamos o método POST para deixar que o servidor decida onde o artigo será criado.

Também possível usar o método PUT para a criação de novos recursos se a API tiver este suporte. Considere o exemplo abaixo.

```

15  PUT /produtos/minha-nova-cerveja HTTP/1.1
16  <produto>
17    <titulo>Cerveja Dogma Azzaca Lover</titulo>
18    <preco dinheiro="brl">8.50</preco>
19  </produto>
20 HTTP 1.1 201 Created
21 Location: /produtos/minha-nova-cerveja
22

```

Figura 14: PUT para atualização de recursos

Importante lembrar que o método PUT é idempotente, conforme estudado no capítulo 1, e o método POST não o é. Esta e outras nuances tornam o desenho de uma boa API REST um trabalho que merece atenção e tempo. Atenção! Acreditar que POST e PUT podem ser sempre mapeados para ações de criação e atualização pode ser ingênuo. O artigo apresentado neste sítio²³ traz algumas considerações importantes para um bom desenho REST. E o modelo de maturidade de Richardson, descrito no quadro da próxima página, permite avaliar o grau de

²³ <https://www.thoughtworks.com/insights/blog/rest-api-design-resource-modeling>

maturidade de uma API REST. Hoje existem guias diversos para o desenho de APIs, como por exemplo o Microsoft API Guidelines²⁴. Eles podem ser usados para você iniciar a definição de um padrão de API na sua organização. Existem diversas tecnologias que suportam esse estilo, tais como:

- Microsoft ASP.NET Web API
- Microsoft WCF
- Java EE JAX-RS
- Node.js Express
- Laravel Routing (PHP)²⁵

Para referência, um fragmento de código com o uso do ASP.NET Web API é mostrado abaixo.

```

1 [RoutePrefix("api/books")]
2 public class ControladorLivros : ApiController
3 {
4     [Route("api/livros")]
5     public IEnumerable<Livro> ObterLivros() { ... }
6
7     [Route("api/livros/{id:int}")]
8     public Livro ObterLivro(int id) { ... }
9
10    [Route("api/livros")]
11    [HttpPost]
12    public HttpResponseMessage CriarLivro(Livro livro) { ... }
13 }
14
15 class Livro { ... }

```

Figura 15: Classe em C# para expor uma API RESTful de manipulação de livros.

Dica: O modelo de maturidade de Richardson é usado na comunidade REST para avaliar o grau de maturidade de uma aplicação e possui quatro níveis. Use-o para avaliar o grau de maturidade da sua API. Os níveis são descritos abaixo.

0. Pântano de objetos XML. O nível 0 usa o protocolo HTTP como protocolo de transporte RPC apenas. Aqui o implementador usa apenas um único recurso (ex. <http://exemplo.org/produto>) e o corpo da mensagem define a intenção do uso (ex. listagem dos produtos ou criação de um novo produto).

1. Recursos. Aqui a API faz a distinção entre recursos (ex. <http://exemplo.org/produto/1> e <http://exemplo.org/produto/2>). Mas neste nível apenas o protocolo HTTP.

2. Verbos HTTP. Aqui a API usa os métodos POST, PUT, DELETE e GET para indicar as intenções sobre os recursos HTTP mantidos no servidor Web. Neste nível os códigos de resposta HTTP também são usados para controlar o estado da aplicação e notificar erros.

3. Neste nível temos o uso de controles de hipertexto. De forma simples, ela indica que o conteúdo da resposta de uma requisição já traz dentro do objeto JSON as URIs necessárias para realizar as próximas ações.

Mais informações sobre o modelo de maturidade de Richardson em:

<https://martinfowler.com/articles/richardsonMaturityModel.html>.

²⁴ <https://github.com/Microsoft/api-guidelines>

²⁵ <https://laravel.com/docs/5.3/routing>

Apresento abaixo algumas orientações de desenho para você criar boas APIs nos seus projetos, independente da linguagem.

1. **Organize APIs ao longo de recursos.** As URIs expostas por um serviço REST devem ser baseados em substantivos (os dados aos quais a API da Web fornece acesso) e não em verbos (o que uma aplicação pode fazer com os dados).
2. **Evite APIs anêmicas.** Evite projetar uma interface REST que espelhe ou dependa da estrutura interna dos dados que ela expõe. O REST é mais do que implementar operações CRUD simples (*Create, Retrieve, Update, Delete*) sobre tabelas separadas em um banco de dados relacional. O objetivo do REST é mapear as entidades de negócios e as operações que um aplicativo pode executar nessas entidades para a implementação física dessas entidades, mas um cliente não deve ser exposto a esses detalhes físicos.
3. **Padronize as suas APIs.** Adote uma convenção de nomenclatura consistente nas URIs. Em geral, é útil usar substantivos no plural para URIs que fazem referência a coleções.
4. **Crie APIs simples.** Evite criar URIs de recursos mais complexos do que coleção/item/coleção.
5. **Considere a atualização em lote para operações complexas.** Considere implementar operações HTTP PUT em massa que possam atualizar em lotes vários recursos de uma coleção de dados. A solicitação PUT deve especificar o URI da coleção e o corpo da solicitação deve especificar os detalhes dos recursos a serem modificados. Esta abordagem pode ajudar a reduzir ineficiências do protocolo HTTP e melhorar o desempenho da sua aplicação.
6. **Trate erros do servidor.** Se os dados fornecidos por uma solicitação PUT ou POST forem inválidos, o servidor Web deve responder com uma mensagem com o código de status HTTP 400 (Pedido inválido). O corpo da mensagem pode conter informações adicionais sobre o problema com o pedido e os formatos esperados, ou pode conter um link para um URL que fornece mais detalhes.
7. **Se você precisar receber datas e horas nas API, use o padrão ISO 8601.** APIs que trafegam datas ou horas devem usar o padrão ISO 8601 para garantir interoperabilidade - <https://www.w3.org/TR/NOTE-datetime>.

Dica: Uma boa forma de começar a usar APIs é com o uso da tecnologia Swagger. Esta tecnologia é usada para o desenho e documentação de APIs e contém as seguintes ferramentas.

- **Swagger Editor** – Editor visual para a criação de APIs
- **Swagger CodeGen** – Ferramenta de desenvolvimento centrado em APIs
- **Swagger UI** – Portal para publicação de APIs

O site para elas está aqui: <http://swagger.io/>

4.6 Microsserviços Web

Uma aplicação Web tradicional é organizada ao longo de um padrão como MVC e distribuída em produção em um ou poucos componentes executáveis (DLL ou WARs). 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.

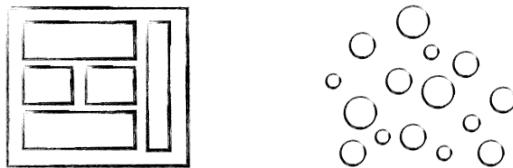


Figura 16: Estilo Arquitetural Monolítico versus Microsserviços. Cada microsserviço opera de forma distribuída, possui o seu próprio banco de dados e expõe uma API RESTful para os seus clientes.

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.

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 as 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.

Vamos examinar um exemplo simples de codificação e execução de um microsserviço com o Spring Boot.

```

1
2 @RestController
3 public class AloMundoMicrosservico {
4     @RequestMapping("/alo")
5     public void alo() {
6         // Método controlador invocado a partir do endereço http://localhost:8080/alo
7         // Método pode invocar as regras de negócio daqui e manipular o seu próprio banco de dados.
8     }
9 }
10
11 @SpringBootApplication
12 public class Microsservico {
13
14     public static void main(String[] args) {
15         SpringApplication.run(Microsservico.class, args);
16     }
17 }
18
19
20 // Na linha de comando o seguinte comando é executado. depois que o JAR (executável Java) foi gerado.
21 java -jar build/libs/Microsservico.jar
22
23 // O JAR contém os códigos binários das classes Microsserviço (ponto de entrada) e AloMundoMicrosservico (que expõe recurso REST)
24

```

Figura 17: Fragmento de código de um microsserviço em Java com o Spring Boot

Neste exemplo quando o utilitário de linha de comando Java é executado, ele instancia um contêiner leve sobre a máquina virtual Java. Ele contêiner tem a capacidade de responder requisições HTTP e suporta diversos serviços do Java EE. Observe que não existe aqui a implantação de código em um servidor Web. Este código roda como um processo apartado no sistema operacional e manipula seus próprios recursos, como por exemplo o seu banco de dados.

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.

4.7 Computação sem servidor (*Serverless*)

As arquiteturas sem servidor permitem que você crie e execute aplicativos e serviços sem ter que gerenciar a infraestrutura. Com este tipo de computação, a sua aplicação ainda é executada em servidores, mas todo o gerenciamento do servidor é feito por uma infraestrutura de nuvem. Você não precisa mais provisionar, escalar e manter servidores para executar seus aplicativos, bancos de dados e sistemas de armazenamento. Com o conceito da computação sem servidor, você desenvolve um código em Java, C# ou Node.JS e o código é executado pela nuvem em resposta a um evento (chamada HTTP, mensagem em uma fila ou mudança em um dado do banco de dados).

Considere um primeiro exemplo conceitual, descrito na **Figura 26**.

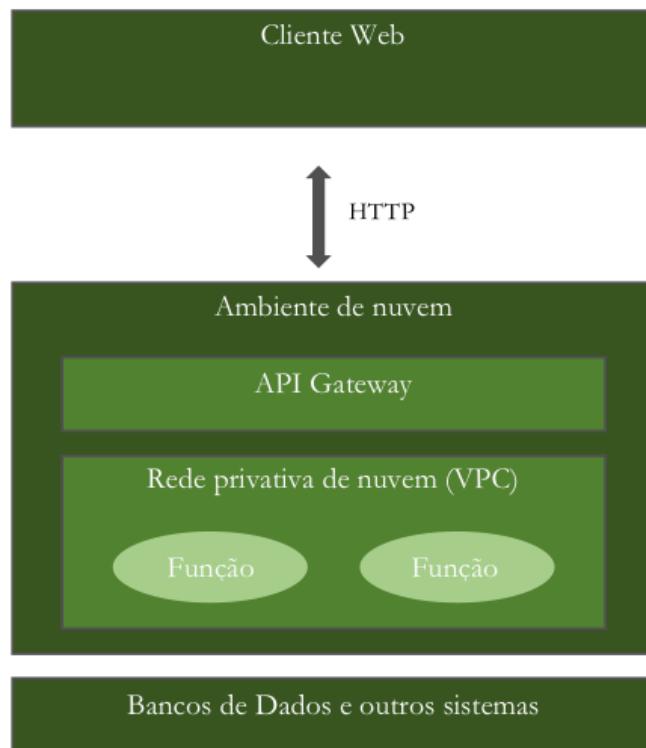


Figura 18: Estilo Arquitetural de Computação sem Servidor (*Serverless*), também chamado de Função como Serviço (FaaS)

Vemos neste exemplo um ambiente de nuvem um API Gateway, que é uma fachada para a exposição e acesso a API de um aplicativo. E logo após a API vemos o conceito de funções. Estas funções são códigos que são executados pelo ambiente de nuvem, mas não residem em um sistema operacional tradicional como Windows ou Linux. A execução de um código ocorre quando um evento configurado pelo desenvolvedor é ativo. No exemplo anterior isso poderia ser a chamada a um método do API Gateway. Em resposta a este evento, o ambiente de nuvem carrega e executa a função. A função pode fazer o que qualquer código faria, como por exemplo chamar o banco de dados ou um outro recurso existente. Como neste tipo de computação você codifica implanta funções (e não sistemas operacionais e servidores), ela é também chamada de função como serviço (FaaS) ou *Backend as a Service* (BAAS)²⁶.

Vamos examinar um exemplo prático para observar como isso funciona. Para isso examine o código abaixo.

```

1 module.exports = function (context, myQueueItem) {
2     context.log('Código serverless JavaScript disparado por evento de fila');
3     context.log('Logando aqui o pacote de entrada', myQueueItem);
4     context.log('E aqui poderia fazer legal como chamar o banco de dados');
5     context.done();
6 };
7

```

Figura 19: Código JavaScript para criar uma função Azure.

Esta função é implantada em ambiente Web nas nuvens, mas não sobre uma máquina virtual. E este código é disparado por um evento de chegada de um pacote de dados em uma fila. Quando o novo pacote chega na fila, o evento é disparado e temos então a execução do código. O log abaixo mostra o resultado da execução.

²⁶ <https://martinfowler.com/articles/serverless.html>

```

2017-03-19T17:18:00.134 Function started (Id=78d439e1-604e-493c-85bb-e2e1ce9c1e82)
2017-03-19T17:18:00.134 Código serverless JavaScript disparado por evento de fila
2017-03-19T17:18:00.134 Logando aqui o pacote de entrada DADOS DE ENTRADA AQUI!
2017-03-19T17:18:00.134 E aqui poderia fazer legal como chamar o banco de dados
2017-03-19T17:18:00.134 Function completed (Success, Id=78d439e1-604e-493c-85bb-
e2e1ce9c1e82)

2017-03-19T17:19:02 No new trace in the past 1 min(s).

2017-03-19T17:20:02 No new trace in the past 2 min(s).

```

Figura 20: Resultado da execução de uma Azure Function. O log foi extraído do console de administração do ambiente de testes em <https://functions.azure.com/try>.

Alguns cenários potenciais para este tipo de computação podem incluir:

- Realizar processamento de eventos (ex. processamento de um boleto quando um evento de faturamento chegar)
- Realizar um processamento temporizado (ex. agendador que irá limpar tabelas temporárias em um banco de dados)
- Execuções de páginas Web (ex. listagem de pedidos abertos para a força de venda em uma empresa)
- Processamentos de tempo real (ex. rastreamento de eventos de carros).

O conceito de computação sem servidor foi popularizado com o produto Amazon Lambda²⁷ e já existem já implementações como o Microsoft Azure Functions²⁸, o Google Cloud Functions²⁹, o Auth0 Web Task³⁰ o IBM OpenWhisk³¹.

Vamos agora estudar um exemplo conceitual um pouco mais elaborado, agora com o uso de produtos da nuvem da Amazon.

²⁷ <https://aws.amazon.com/pt/lambda/>

²⁸ <https://azure.microsoft.com/pt-br/services/functions/>

²⁹ <https://cloud.google.com/functions/>

³⁰ <https://webtask.io>

³¹ <https://www.ibm.com/cloud-computing/bluemix/pt/openwhisk>

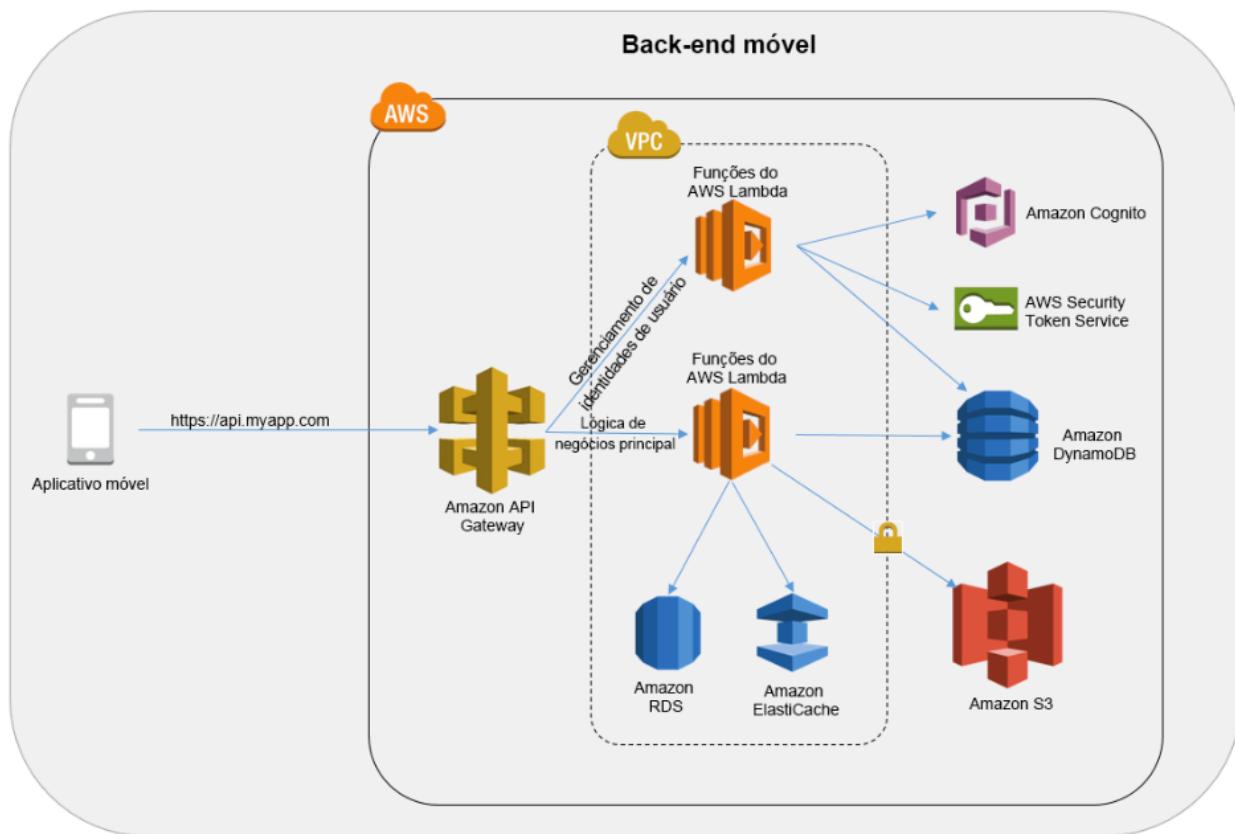


Figura 21: Computação sem Servidor (*Serverless*) com funções AWS Lambda

Nessa figura, temos a figura central das funções Lambda. Uma função Lambda é um código escrito em Java, Node.js ou Python. O código é implantado através de um console de administração e neste momento o desenvolvedor especifica parâmetros como a memória disponível para a função ou o tempo máximo de execução permitido. O produto chamado API Gateway invoca as funções lambda em resposta a requisições cliente. E estas execuções consomem tempo de processamento e memória da infraestrutura virtual. E este consumo de ciclos de CPU e de memória usada pelas funções é usada para calcular o custo de utilização.

Observe que este estilo é apropriado para cenários arquiteturais orientados a tarefas, como por exemplo o processamento de boletos. Nesse cenário, você poderia trocar as necessidades de provisionamento de máquinas virtuais e sistemas operacionais pela pura execução de função. E o pagamento ocorre apenas pela quantidade de ciclos de CPU e memória consumida. A promessa de vários fornecedores é que este tipo de computação pode ter um custo menor do que a computação convencional com o provisionamento de máquinas virtuais. Em um exemplo disponível do Amazon Lambda, a execução de 30 milhões de funções com tempo médio de 200 milissegundos por execução e consumo médio de 128 megabytes de memória geraria uma conta mensal de menos de seis dólares. (<https://aws.amazon.com/pt/lambda/pricing>).

Dica: Faça simulações para avaliar se este modelo de computação é apropriado e viável na ótica financeira para o seu cenário específico de negócio.

Em perspectiva, algumas das potenciais vantagens deste estilo incluem:

- Abstração do sistemas operacionais e máquinas virtuais, com redução de custos indiretos da administração destes servidores;
- Redução de riscos de subprovisionamento ou superprovisionamento da computação;
- Escalabilidade sob demanda;
- Redução de custos para aplicativos orientados a tarefas.

Existem algumas desvantagens que devem ser consideradas também, tais como:

- Aprisionamento a soluções de fornecedores de nuvens;
- Latência de execução, que pode variar de 10 a 100 milissegundos para funções pequenas em Python ou JavaScript até alguns segundos para funções baseadas em Java e executadas com baixa frequência.
- Duração da execução, que é limitada pelos fornecedores. E isso pode implicar em refatoração arquitetural de código legado que faça algum tipo de processamento em lote de longa duração.
- Ausência da gestão de estado. Funções *FaaS* não mantém estado.

Em 2017, este modelo é ainda bastante incipiente e deve haver cautela para a adoção do mesmo. Embora já existam casos reais no Brasil, observo que o uso ainda é restrito para alguns cenários pontuais em algumas grandes empresas.

4.8 Comparativo dos Estilos Arquiteturais

Os estilos arquiteturais Web 2.0, SPA, API Web, Microsserviços e computação sem servidor (FaaS) tem vantagens e desvantagens, como qualquer tática arquitetural. E para ajudar você a escolher o melhor estilo conforme as necessidades do seu projeto, elaborei uma tabela comparativa destes estilos.

	Vantagens	Desvantagens e outras considerações
Web 2.0	<p>Bem disseminado, apresenta muitas tecnologias disponíveis no mercado e possui baixo risco técnico.</p> <p>Em 2017, é o padrão dominante para o desenvolvimento de aplicativos Web.</p>	<p>Pode apresentar problemas de desempenho se todo o código controlador ficar no servidor. (Ex. ASP.NET MVC ou MVC do Java JSF).</p> <p>Pode ser implementado com o padrão MVC ou MVVM.</p>
SPA (Single Page Application)	Performático para aplicações responsivas e já conta com bom suporte tecnológico de frameworks JavaScript.	<p>Requer um trabalho mais elaborado de usabilidade e ainda não possui uma mão de obra tão ampla no Brasil.</p> <p>Requer o uso do padrão MVVM.</p>
Web API	Simples e apropriado para criar reuso de código servidor em ambientes Web e móveis. Também conta com amplo suporte tecnológico de fornecedores.	<p>Requer atenção a padrões de desenho de APIs e pede em médio prazo a introdução de um API Gateway.</p> <p>Requer atenção também a detalhes de usabilidade da API para promover reuso, controle de versões, performance para operações em lote e também segurança.</p>
Microsserviços	<p>Apropriado para acelerar o desenvolvimento, reduzir a complexidade do código de aplicações e prover escalabilidade sob demanda em ambientes de produção.</p> <p>Permite a evolução tecnológica</p>	<p>Traz complexidades operacionais em produção e apresenta testes mais desafiantes devido à sua natureza distribuída. A mão de obra que sabe operar neste estilo é bastante escassa ainda.</p> <p>Quando se usa microsserviços, é esperado também o uso do estilo API Web e um produto de API Gateway.</p>

Computação sem servidor (<i>Serverless</i>)	Apropriado para execução de tarefas sem demanda e para a redução do custo de provisionamento e administração de máquinas virtuais	Traz algum aprisionamento a plataformas de fornecedores como Google, Amazon ou Microsoft. A mão de obra que sabe operar neste estilo é bastante escassa ainda. Quando se usa computação sem servidor, é esperado também o uso do estilo API Web e um produto de API Gateway.
--	---	---

Tabela 5: Comparativo de estilos arquiteturais Web

4.9 Para Saber Mais

O conceito de arquétipos apresentado em Meier (Meier, 2009) é similar ao de estilos apresentados nesse capítulo e se fundamenta então como uma excelente referência de estudo para aprofundamento no conceito de estilos arquiteturais.

Para os interessados no estilo de microsserviços, uma boa introdução ao tema é encontrado no livro *Building Microservices* (Newman, 2015).

Já o livro *API For Dummies* é uma introdução leve e apropriado para conhecer os fundamentos do estilo arquitetural de APIs Web (APIGEE, 2014).

5 Topologia de Servidores Web

Embora já existam perspectivas de computação sem servidor como visto no capítulo anterior, a esmagadora maioria das aplicações Web operam sobre servidores de software Web, que 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.

5.1 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 Illionis).

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*.

5.2 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 Server³² 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.

³² <https://httpd.apache.org>

Esses servidores se tornaram dominantes na Web ainda nos anos 90 e exemplos de servidores dessa categoria incluem o Microsoft IIS³³ e NGINX³⁴. 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.

5.3 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

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 Server³⁵
- Oracle Internet Applicaton Server³⁶ (antigo BEA Weblogic)
- Redhat Wildfly³⁷ (JBOSS Application Server)
- Apache TomEE³⁸
- Microsoft IIS + MSQM + .NET Framework³⁹
- Microsoft BizTalk

5.4 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 performer e alocam centenas de *threads* quando são instanciados. Um exemplo de servidor Web de quarta geração é Express⁴⁰ do Node.js. Ele é um servidor minimalistico 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 Boot⁴¹. O Eclipse Jetty⁴² e o KumuluzEE⁴³ 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

³³ <http://www.iis.net>

³⁴ <https://www.nginx.com>

³⁵ <http://www-03.ibm.com/software/products/pt/appserv-was>

³⁶ <http://www.oracle.com/technetwork/middleware/ias/overview/index.html>

³⁷ <http://wildfly.org>

³⁸ <http://tomee.apache.org/apache-tomee.html>

³⁹ <https://www.microsoft.com/net/default.aspx>

⁴⁰ <http://expressjs.com/pt-br>

⁴¹ <http://projects.spring.io/spring-boot/>

⁴² <http://www.eclipse.org/jetty/>

⁴³ <https://ee.kumuluz.com>

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.

5.5 Comparativo de Servidores Web

	Baseados em Processos	Baseados em Threads	Baseados em Servidores de Aplicação	Embarcados em Aplicações Web
	1º Geração	2º Geração	3º Geração	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 2.0 SPA	Web 1.0 Web 2.0 SPA	Web 1.0 Web 2.0 SPA	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	JavaScript/Node C# Java Python PHP

Tabela 6: Tipos de Servidores de Software Web

5.6 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.

5.7 Distribuição Mínima

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

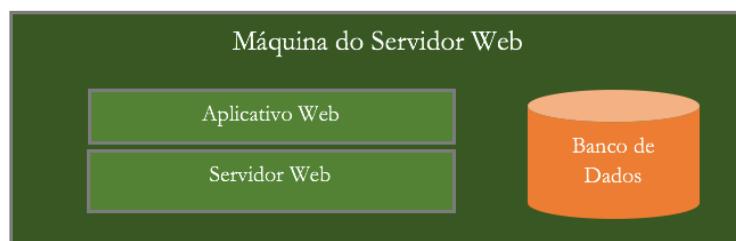


Figura 22: Distribuição Mínima de uma Aplicação Web

Esse cenário é recomendado para ambientes de desenvolvimento pois permite o trabalho *off-line* e também para aplicações 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 NodeJS) para produtos de Internet são muitas vezes distribuídas nessa topologia.

5.8 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 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. A

Figura 23 apresenta o esquema desse modelo.

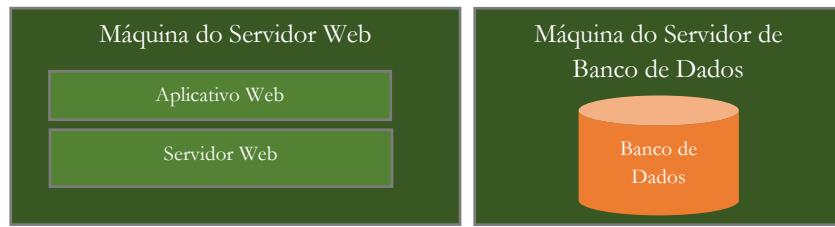


Figura 23: Distribuição com Servidor Web Dedicado

5.9 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 24: 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, 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 25.



Figura 25: 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 Balancing⁴⁴).



Figura 26: Balanceador de carga para clusterização

⁴⁴ LVS - <http://www.linuxvirtualserver.org/whatis.html>

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

5.10 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. (Ver a

Figura 27).



Figura 27: Balanceador de carga com máquina para conteúdo estático

5.11 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 28**).

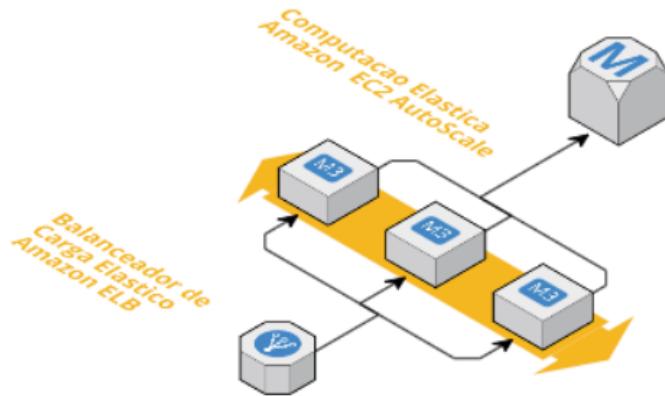


Figura 28: Cluster elástico em ambiente de nuvem. O ELB (balanceador elástico) analisa as cargas de trabalho e solicita que o componente AutoScale aloque ou desaloque máquinas de forma dinâmica.

Ela permite que um conjunto mínimo de máquinas sejam alocada para servir uma aplicação. Se o consumo de recursos aumentar além 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.

5.12 Comparativo de Topologias Web

A Tabela 7 apresenta um resumo dos tipos de topologia conforme requisitos arquiteturais comuns em plataformas Web.

	Simplicidade	Facilidade para Implantar	Segurança Física	Escalabilidade	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 7: 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.

5.13 Para Saber Mais

O livro de JD Meier (Meier, 2009) possui um bom tratamento sobre topologias Web, com exemplos centrados em tecnologias .NET.

Para um aprofundamento em topologias de nuvens, existem muitas referências e livros já publicados. Uma boa referência para leitura é o livro de Cloud Computing de Thomas Erl (Erl, 2013).

6 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 JCP⁴⁵, 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:

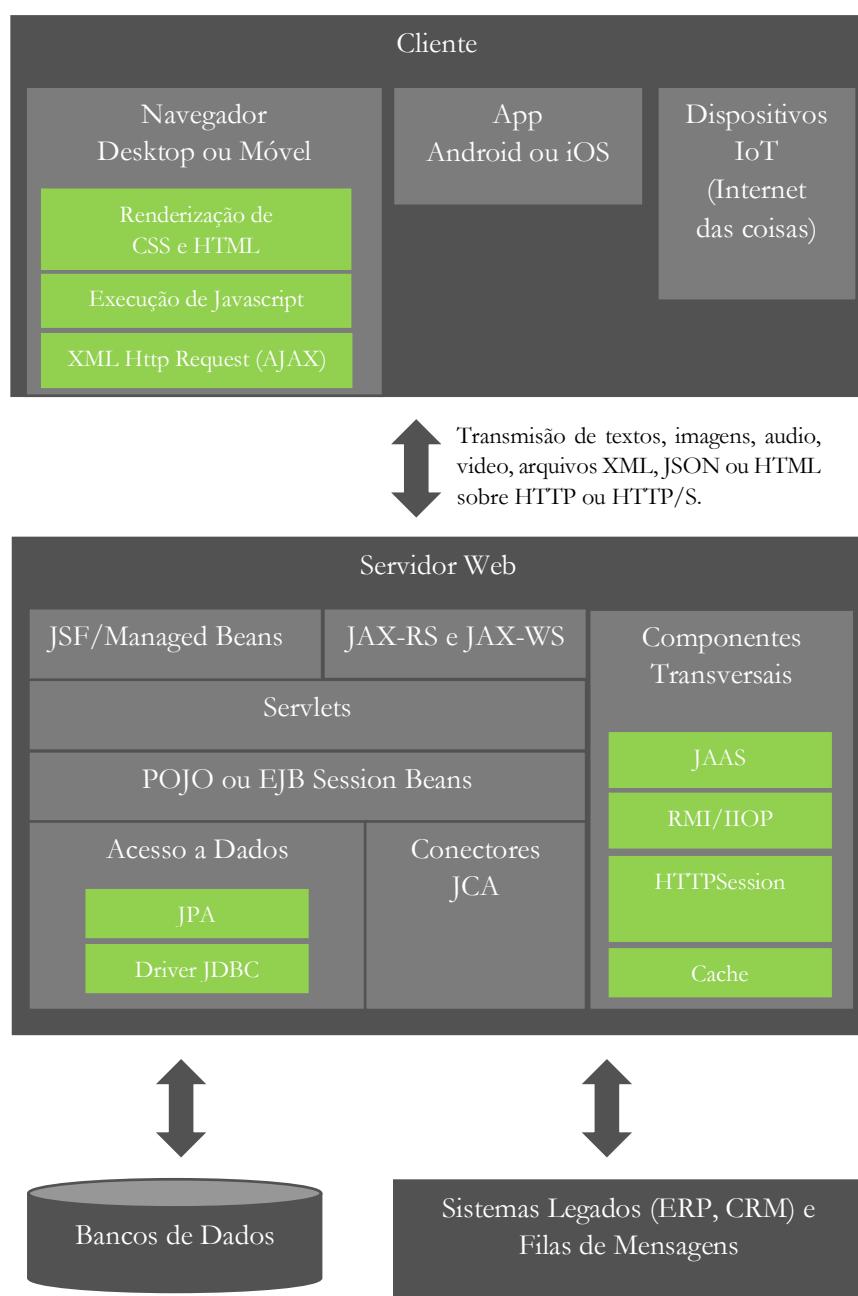
- Apache Tomcat
- Redhat JBOSS Wildfly (JBoss Application Server)
- IBM WebSphere Application Server
- Oracle Internet Application Server

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

⁴⁵ <http://jcp.org>

Figura 29.**Figura 29:** Componentes arquiteturais Java EE Web

6.1 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:

- PrimeFaces⁴⁶
- JBOSS RichFaces⁴⁷
- Apache MyFaces⁴⁸

Uma página JSF é montada como um arquivo XHTML com uma linguagem de marcação de visão chamada de Facelets⁴⁹. 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 aqui⁵⁰. Os capítulos de 8 a 16 desse mesmo tutorial apresentam os detalhes de uso dessa tecnologia e exemplos práticos.

6.2 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 negócios 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 aqui⁵¹.

6.3 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.

⁴⁶ <http://primefaces.org>

⁴⁷ <http://richfaces.jboss.org>

⁴⁸ <https://myfaces.apache.org>

⁴⁹ <https://docs.oracle.com/javaee/7/tutorial/jsf-facelets.htm>

⁵⁰ <https://docs.oracle.com/javaee/7/tutorial/jsf-intro.htm>

⁵¹ <https://docs.oracle.com/javaee/7/tutorial/servlets.htm>

- 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:

- JBOSS WebServices⁵²
- Apache CXF⁵³
- Metro⁵⁴

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
- RESTEasy⁵⁵
- Jersey⁵⁶

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 aqui⁵⁷.

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.

6.4 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, aqui⁵⁸.

Algumas implementações de terceiros têm ganhado popularidade e rivalizado com a especificação JAAS. Exemplos incluem o Apache Shiro⁵⁹ ou o PicketBox⁶⁰.

6.5 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).

⁵² <http://iboss.ws.iboss.org>

⁵³ <http://cxf.apache.org>

⁵⁴ <https://metro.java.net>

⁵⁵ <http://resteasy.iboss.org>

⁵⁶ <https://jersey.java.net>

⁵⁷ <https://docs.oracle.com/javaee/7/tutorial/partwebsvcs.htm>

⁵⁸ <https://docs.oracle.com/javaee/7/tutorial/partsecurity.htm>

⁵⁹ <http://shiro.apache.org>

⁶⁰ <http://picketbox.iboss.org>

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 aqui⁶¹.

6.6 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 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 ORM⁶²
- Eclipse Link⁶³

A documentação oficial do JPA é encontrada nos capítulos 37 a 44 do Java EE Tutorial. O sítio da documentação pode ser acessado aqui⁶⁴.

6.7 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 aqui⁶⁵.

- 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 RabbitMQ⁶⁶
- JBOSS HornetMQ⁶⁷
- IBM WebSphere MQ⁶⁸

O Java EE Tutorial explica o JMS nos capítulos 45 e 46, com exemplos e cenários de uso. A documentação pode ser encontrada aqui⁶⁹.

6.8 Considerações de Desenho Arquitetural

6.8.1 Escolha de servidores

Escolha os servidores mais simples que possam atender ao problema em questão. Já observamos arquitetos que

⁶¹ <https://docs.oracle.com/javaee/7/tutorial/partentbeans.htm>

⁶² <http://hibernate.org/orm/>

⁶³ <http://www.eclipse.org/eclipselink/#jpa>

⁶⁴ <https://docs.oracle.com/javaee/7/tutorial/partpersist.htm>

⁶⁵ <https://docs.oracle.com/javaee/7/tutorial/partsupporttechs.htm>

⁶⁶ <http://camel.apache.org/rabbitmq.html>

⁶⁷ <http://hornetq.jboss.org>

⁶⁸ <http://www-03.ibm.com/software/products/pt/ibm-mq>

⁶⁹ <https://docs.oracle.com/javaee/7/tutorial/partmessaging.htm>

escolheram peças complicadas para o cenário sendo atacado, escolhendo servidores de aplicação onde servidores Web mais simples como o ApacheTomcat poderiam ser selecionados.

Em cenários mais simples ou em estilos de microsserviços, considere tecnologias Java Web alternativas como por exemplo o Spring Boot⁷⁰, 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.

6.8.2 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 EE⁷¹ ou JetBrains IntelliJ⁷² é importante 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 Maven⁷³ e o Gradle⁷⁴ é um aspecto importante na construção de sistemas Java EE.

6.8.3 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 ReactJS, VueJS, Ember, Backbone, AureliaJS ou Angular. 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.

⁷⁰ <http://projects.spring.io/spring-boot/>

⁷¹ <http://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/mars2>

⁷² <https://www.jetbrains.com/idea/>

⁷³ <https://maven.apache.org>

⁷⁴ <http://gradle.org>

6.8.4 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.

6.8.5 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.

6.8.6 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, 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.

6.8.7 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 requiram um controle de acesso ainda mais granular, considere o uso do padrão baseado em recursos com o uso de frameworks como o Apache Shiro.

6.8.8 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.

6.8.9 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.

6.8.10 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 SLF4J⁷⁵, 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.

6.8.11 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 aqui⁷⁶.

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 aqui⁷⁷.

6.8.12 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.

6.8.13 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 21⁷⁸ do Java EE Tutorial apresenta um bom material a respeito.

⁷⁵ <http://www.slf4j.org>

⁷⁶ <http://docs.oracle.com/javase/7/docs/technotes/guides/management/jconsole.html>

⁷⁷ <https://docs.oracle.com/javase/tutorial/jmx/>

⁷⁸ <https://docs.oracle.com/javaee/7/tutorial/bean-validation.htm>

6.8.14 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.

6.8.15 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.

6.8.16 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.

6.9 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.

6.9.1 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).

6.9.2 Baixa Produtividade

Quando comparado ao LAMP ou o ASP.NET, o Java EE Web 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 Framework⁷⁹;
- Usar linguagens de mais alto nível que rodam no topo de máquinas virtuais como por exemplo o Scala⁸⁰, Groovy⁸¹ e *scaffoldings* como o Grails⁸²;
- 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.

6.9.3 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.

6.9.4 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, 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.

6.10 Para Saber Mais

A literatura de Java EE é extensa e é formada na sua maioria por livros de tecnologias específicas. Mas se queremos

⁷⁹ <https://www.playframework.com>

⁸⁰ <http://www.scala-lang.org>

⁸¹ <http://www.groovy-lang.org>

⁸² <https://grails.org>

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 sítio⁸³.

⁸³ <https://jcp.org/en/jsr/detail?id=366>

7 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 30 mostra os principais componentes da arquitetura Web Microsoft.

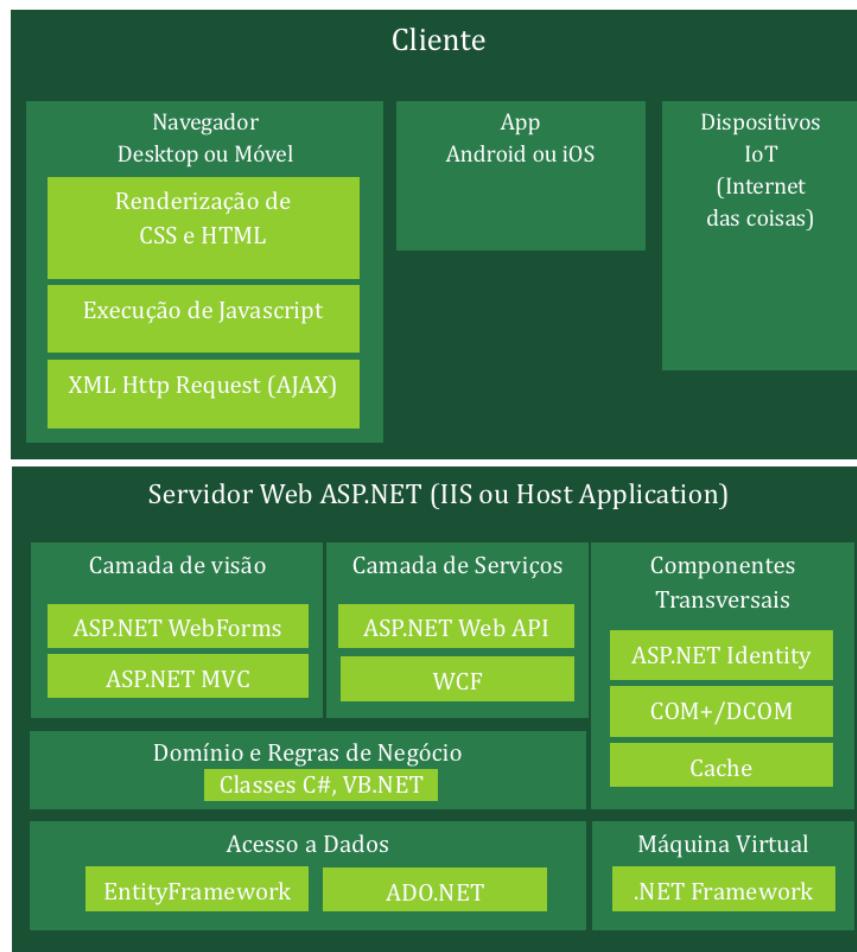


Figura 30: Componentes arquiteturais ASP.NET

Toda aplicação ASP.NET é uma aplicação que roda sobre o .NET Framework. O .NET Framework opera apenas em ambiente Windows. Mas existem versões também para outros sistemas operacionais com o projeto Xamarin Mono⁸⁴ e o recente Microsoft .NET Core⁸⁵.

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

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

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.

⁸⁴ <http://www.mono-project.com>

⁸⁵ <https://dotnet.github.io>

7.1 ASP.NET Web Forms

O ASP.NET Web Forms⁸⁶ 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 Forms⁸⁷ 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.

7.2 ASP.NET MVC

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

⁸⁶ <http://www.asp.net/web-forms>

⁸⁷ [https://msdn.microsoft.com/en-us/library/dd30h2yb\(v=vs.110\).asp](https://msdn.microsoft.com/en-us/library/dd30h2yb(v=vs.110).asp)

- 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*⁸⁸) 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.

O ASP.NET MVC Core (ASP.NET MVC 6), em maio de 2016, se encontra em versão RC2 – *Release Candidate* e foi lançado no framework ASP.NET Core 1.0 (que foi antes chamada de ASP.NET 5).

A partir do ASP.NET MVC 3, diferentes motores de visualização foram incorporados na tecnologia, tais como o Razor⁸⁹ (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.

7.3 ASP.NET Core

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.

⁸⁸ <http://martinfowler.com/eaaCatalog/domainModel.html>

⁸⁹ <http://www.asp.net/web-pages/overview/getting-started/introducing-razor-syntax-c>

ASP.NET 4.6 e ASP.NET Core 1.0	
ASP.NET 4.6 Contém o ASP.NET MVC 5	ASP.NET Core 1.0 Contém o ASP.NET MVC 6 e ASP.NET Web API 2
.NET Framework 4.6 Roda no Windows e contém o EF 6	.NET Core 1.0 Roda no Windows, Linux e OS/X e Contém o EF Core 1.0
Bibliotecas do .NET Framework	Bibliotecas do .NET Core

Figura 31: 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;
- 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 Code⁹⁰ é uma IDE multiplataforma e tem semelhanças com IDEs leves e populares de desenvolvimento como o Atom⁹¹ e o Sublime⁹².

O NET Core já tem suporte oficial da Microsoft e está na versão 1.1. 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.

7.4 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 muitas vezes usado para

⁹⁰ <https://code.visualstudio.com>

⁹¹ <https://atom.io>

⁹² <https://www.sublimetext.com>

suportar serviços em aplicações Web Microsoft. Ele foi inspirado em modelos de componentes de objetos como o COM⁹³, COM+⁹⁴, 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 MTOM⁹⁵. Uma lista completa dos protocolos ofertados pelo WCF está disponível nesse sítio⁹⁶.

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 aqui⁹⁷.

7.5 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 API⁹⁸ 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 aqui⁹⁹.

Em termos arquiteturais, o ASP.NET Web API é o componente recomendado para a criação de APIs REST e até mesmo microsserviços.

7.6 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.

⁹³ <https://www.microsoft.com/com/default.mspx>

⁹⁴ [https://msdn.microsoft.com/library/ms685978\(VS.85\).aspx](https://msdn.microsoft.com/library/ms685978(VS.85).aspx)

⁹⁵ https://en.wikipedia.org/wiki/Message_Transmission_Optimization_Mechanism

⁹⁶ [https://msdn.microsoft.com/en-us/library/ms731092\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/ms731092(v=vs.110).aspx)

⁹⁷ [https://msdn.microsoft.com/en-us/library/dd456779\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/dd456779(v=vs.110).aspx)

⁹⁸ <http://www.asp.net/web-api>

⁹⁹ <http://www.asp.net/web-api/overview/getting-started-with-aspnet-web-api/tutorial-your-first-web-api>

- 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 LINQ¹⁰⁰, 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*¹⁰¹
- *Model First*¹⁰²
- *Code First*¹⁰³

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 nHibernate¹⁰⁴, 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 FluentAPI¹⁰⁵. No contexto do Entity Framework, ele é usado para projetos EF Code First.

7.7 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^{106 107} sobre esta tecnologia, que vem instalada em todo servidor Windows.

7.8 Service Fabric

Nos últimos anos, as plataformas de nuvens se tornaram populares. E com o recente advento de 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 Fabric¹⁰⁸ 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.

¹⁰⁰ <https://msdn.microsoft.com/pt-br/library/bb397906.aspx>

¹⁰¹ <https://www.asp.net/mvc/overview/getting-started/database-first-development/setting-up-database>

¹⁰² <https://msdn.microsoft.com/en-us/data/jj205424.aspx>

¹⁰³ <https://msdn.microsoft.com/en-us/data/jj193542>

¹⁰⁴ <http://nhibernate.info>

¹⁰⁵ <https://msdn.microsoft.com/en-us/data/jj591617.aspx>

¹⁰⁶ [https://msdn.microsoft.com/en-us/library/ms711472\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/ms711472(v=vs.85).aspx)

¹⁰⁷ <https://msdn.microsoft.com/en-us/library/bb969123.aspx>

¹⁰⁸ <https://docs.microsoft.com/pt-br/azure/service-fabric/service-fabric-overview>

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

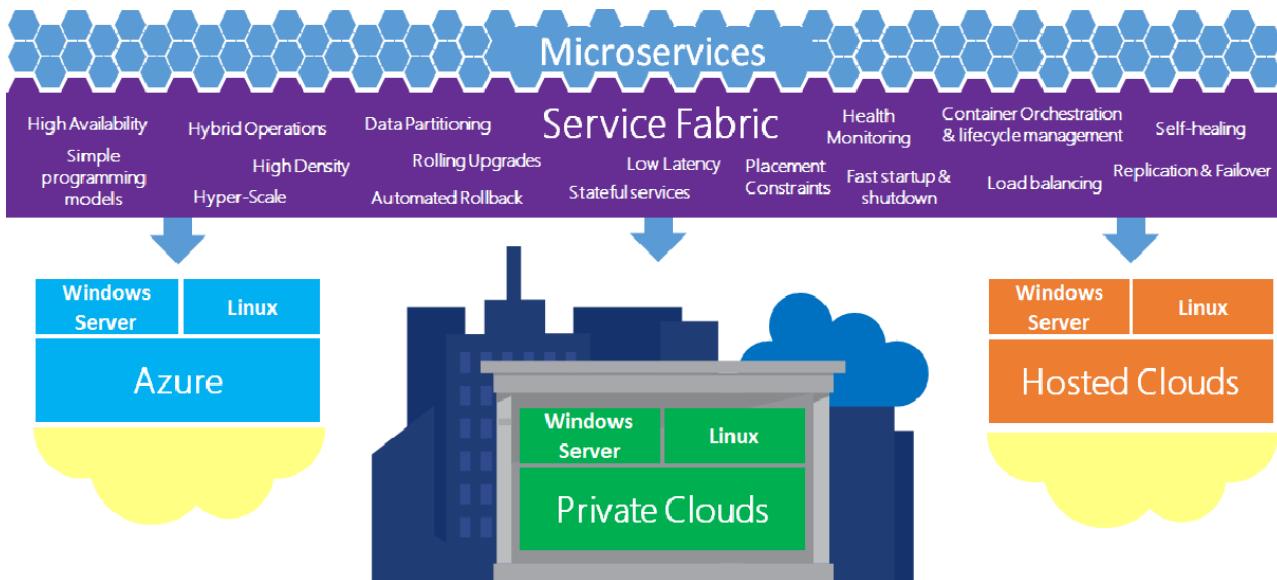


Figura 32: Arquitetura de Referência do Service Fabric

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.

7.9 Considerações de Desenho Arquitetural

7.9.1 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.

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 IIS¹⁰⁹ 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

¹⁰⁹<http://www.iis.net/learn/get-started/introduction-to-iis/introduction-to-iis-architecture#Components>

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 IIS¹¹⁰.

7.9.2 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 2017¹¹¹, pode ser baixada daqui.

Times .NET tem se beneficiado de uma poderosa ferramenta de suporte a refatoração de código, chamada Resharper¹¹², que opera como um plug-in para o Visual Studio.

Times .NET também tem usado o suporte a personalização de *scaffolding*¹¹³ (andaimes) e DSLs (Domain Specific Language)¹¹⁴ do Visual Studio para acelerar tarefas repetitivas como cadastros e aumentar a produtividade de seus times.

Times .NET que possuem o Visual Studio Ultimate¹¹⁵ podem usar os seus facilitadores para avaliar as suas arquiteturas físicas com a engenharia reversa de suas aplicações.

7.9.3 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.

Exceto por razões excepcionais, 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 Protection¹¹⁶ do ASP.NET para facilitar esse trabalho.

7.9.4 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 Navigation¹¹⁷), que permite gerar regras de navegação que associam URIs Web reais a nomes virtuais.

¹¹⁰ <https://docs.asp.net/en/latest/publishing/iis.html>

¹¹¹ <https://www.visualstudio.com/pt-br/products/visual-studio-community-vs.aspx>

¹¹² <https://www.jetbrains.com/resharper/>

¹¹³ <https://msdn.microsoft.com/pt-br/magazine/dn745864.aspx>

¹¹⁴ <https://msdn.microsoft.com/en-us/library/ee943825.aspx>

¹¹⁵ <https://msdn.microsoft.com/en-us/library/57b85fsc.aspx>

¹¹⁶ <https://docs.asp.net/en/latest/security/data-protection/index.html>

¹¹⁷ <https://msdn.microsoft.com/en-us/library/e468hxky.aspx>

7.9.5 Desenho de Páginas

- Mantenha as páginas simples.
- Use *layout Razor*¹¹⁸ 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 API¹¹⁹, mas existem outros fornecedores de mercado que possuem componentes ainda mais poderosos. Exemplos incluem o Kendo-UI¹²⁰, da Telerik, ou o Ext.NET¹²¹, da Object.Net.
- Mantenha todo e qualquer elemento de configuração visual em arquivos CSS.

7.9.6 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 senhas¹²² no ASP.NET.

Recomenda-se usar as facilidades já presentes no framework ASP.NET Identity¹²³, 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 fatores¹²⁴, 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 OAuth¹²⁵.

7.9.7 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 Based Authorization, View Based Authorization e Custom Policy-Based Authorization*)¹²⁶. Esses modelos permitem um controle fino dos recursos da sua aplicação, enquanto mantém mínima intervenção no código fonte.

7.9.8 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.NET¹²⁷.

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 2¹²⁸ do EF para reduzir o tráfego SQL em aplicações ASP.NET.

¹¹⁸ <http://www.asp.net/web-pages/overview/ui-layouts-and-themes/3-creating-a-consistent-look>

¹¹⁹ <https://docs.asp.net/en/latest/mvc/views/view-components.html>

¹²⁰ <http://www.telerik.com/aspnet-mvc>

¹²¹ <http://ext.net>

¹²² <http://www.asp.net/identity/overview/features-api/best-practices-for-deploying-passwords-and-other-sensitive-data-to-aspnet-and-azure>

¹²³ <http://www.asp.net/visual-studio/overview/2013/creating-web-projects-in-visual-studio#auth>

¹²⁴ <http://www.asp.net/identity/overview/features-api/two-factor-authentication-using-sms-and-email-with-aspnet-identity>

¹²⁵ <https://azure.microsoft.com/en-us/documentation/articles/web-sites-dotnet-deploy-aspnet-mvc-app-membership-oauth-sql-database/>

¹²⁶ <https://docs.asp.net/en/latest/security/authorization/introduction.html>

¹²⁷ <https://docs.asp.net/en/latest/performance/caching/index.html>

¹²⁸ <https://msdn.microsoft.com/en-us/magazine/hh394143.aspx>

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 aqui¹²⁹.

7.9.9 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.Transactions¹³⁰.

Estabeleça um ponto único na arquitetura lógica da sua aplicação para a demarcação transacional.

7.9.10 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.

7.9.11 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 Monitor¹³¹. 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 desempenho¹³².

O próprio ASP.NET possui uma infraestrutura nativa para a monitoração¹³³ de aplicações, diagnóstico de problemas e log de eventos.

7.9.12 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.

¹²⁹ <http://www.iis.net/learn/manage/managing-performance-settings/configure-iis-7-output-caching>

¹³⁰ [https://msdn.microsoft.com/en-us/library/ms254973\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/ms254973(v=vs.110).aspx)

¹³¹ [https://technet.microsoft.com/pt-br/library/cc749249\(v=ws.11\).aspx](https://technet.microsoft.com/pt-br/library/cc749249(v=ws.11).aspx)

¹³² <http://www.solarwinds.com/topics/microsoft-iis-monitor>

¹³³ [https://msdn.microsoft.com/en-us/library/ms178703\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/ms178703(v=vs.85).aspx)

7.9.13 Validação

Não confie nos dados recebidos. Valide todos os dados que chegam à sua aplicação ASP.NET. Ao mesmo tempo, use validadores cliente ASP.NET¹³⁴ para aumentar a experiência de uso e reduzir a quantidade de requisições HTTP.

7.9.14 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.

7.9.15 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.

7.10 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.

7.10.1 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.

7.10.2 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;

¹³⁴ <https://msdn.microsoft.com/en-us/library/bwd43d0x.aspx>

- Usar ASP.NET MVC sempre que possível.

7.10.3 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.

7.10.4 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, a Tabela 8 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 8: Comparativo arquitetural entre Java EE e .NET

7.11 Para Saber Mais

A literatura de .NET também é extensa, embora a maior parte dos livros cubra temas tecnológicos específicos. Ao mesmo tempo, existem bons livros de arquiteturas de aplicações .NET. O livro **Microsoft Architecture Application Guide** (Meier, 2009) é ainda uma excelente referência para arquitetos .NET.

Já o livro **C# 6 and .NET Core 1.0: Modern Cross-Platform Development** (Price, 2016) fornece um bom tratamento sobre as novas tecnologias da Microsoft - ASP.NET MVC 6 e Web API 2.

8 Servidores Web Baseados em JavaScript

A linguagem JavaScript foi criada pela extinta empresa Netscape ainda 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 por componentes 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 33.

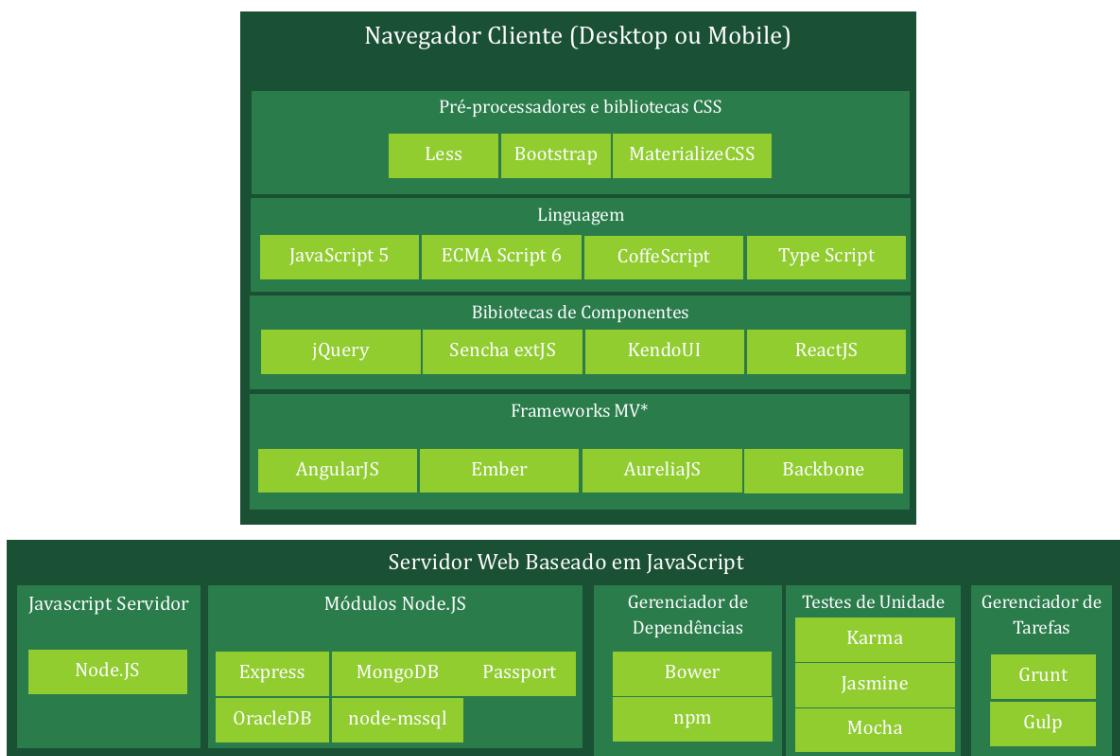


Figura 33: Componentes de uma arquitetura baseada em JavaScript, CSS e HTML

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:

- Preprocessadores 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 Bower e npm e 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 WebStorm¹³⁵, Visual Studio Code¹³⁶, Atom¹³⁷ ou Sublime¹³⁸, os desenvolvedores Web tem hoje um rico ambiente para o desenvolvimento de aplicações Web.

A pilha descrita na Figura 33 não precisa ser adotada na sua totalidade. A Figura 34 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.

¹³⁵ <https://www.jetbrains.com/webstorm/>

¹³⁶ <https://code.visualstudio.com>

¹³⁷ <https://atom.io>

¹³⁸ <https://www.sublimetext.com>

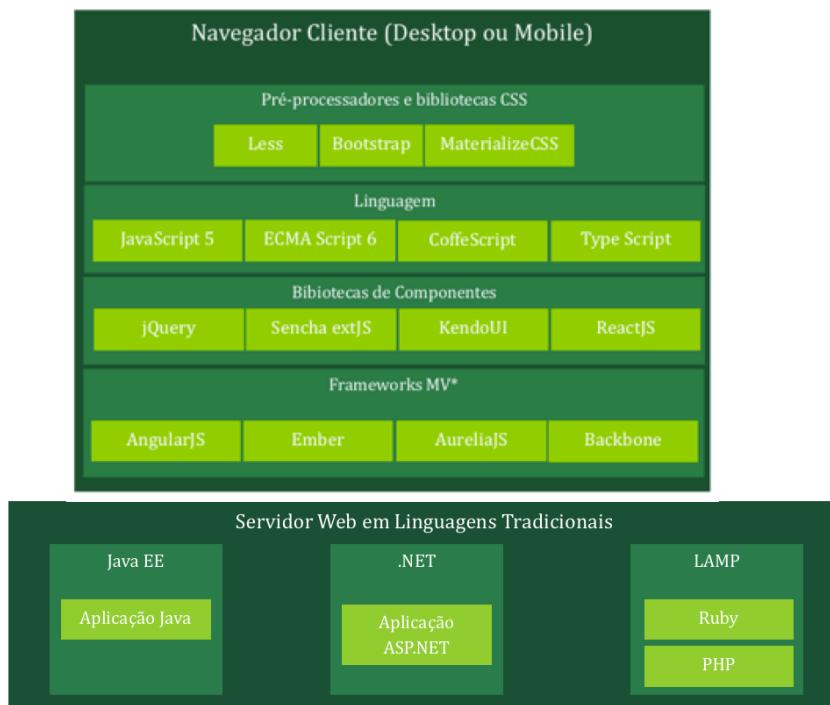


Figura 34: Componentes de uma arquitetura híbrida JavaScript com JavaEE, .NET ou LAMP

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

8.1 Aceleradores, Bibliotecas e Frameworks CSS

As folhas de estilo em cascata¹³⁹, 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 SPA (Single Page Application) 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 Design¹⁴⁰ do Google é um projeto nesse sentido. Inspirado no trabalho de desenho 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.

¹³⁹ <http://www.w3schools.com/css/>

¹⁴⁰ <https://material.google.com>

Uso de bibliotecas CSS para aumentar a produtividade dos times. Algumas bibliotecas já implementam o conceito de Material Design, como por exemplo o MaterializeCSS¹⁴¹ ou Material-UI¹⁴². Algumas dessas bibliotecas, como o Less¹⁴³ ou o Bootstrap¹⁴⁴, 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 blocos¹⁴⁵.

8.2 Linguagens Aceleradores 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 (DOM¹⁴⁶). 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 Request¹⁴⁷).

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 35: 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.

¹⁴¹ <http://materializecss.com>

¹⁴² <http://www.material-ui.com>

¹⁴³ <http://lesscss.org>

¹⁴⁴ <http://getbootstrap.com>

¹⁴⁵ <https://www.toptal.com/css/introduction-to-bem-methodology>

¹⁴⁶ <http://www.w3.org/DOM/>

¹⁴⁷ <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 desenvolvidos Web de larga escala. Nesse sentido, o mercado começou a observar alguns aceleradores nesse sentido, que incluem:

- Linguagens aceleradoras como CoffeScript¹⁴⁸, 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 36: Fragmento de Código JavaScript versus CoffeScript

- Uso de linguagens OO como o TypeScript¹⁴⁹ da Microsoft. Essa linguagem, que conta já com suporte forte no Visual Studio e outras IDEs como o Atom 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 37.

```

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 // Saída mostra: Alo, Joao Silva|

```

Figura 37: Fragmento de Código TypeScript

¹⁴⁸ <http://coffeescript.org>

¹⁴⁹ <https://www.typescriptlang.org>

Em termos técnicos, essas linguagens são compiladas ou traduzidas para JavaScript. Alguns autores chamam esse processo de *transpilação (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 Babel¹⁵⁰¹⁵¹.

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.

8.3 Linguagem ECMA Script 2015 (JavaScript 6)

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 encontrada aqui¹⁵². 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 Safari¹⁵³, ele pode ser transformado em código JavaScript 5 com o uso de transpiladores em bibliotecas como o Babel.

8.4 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-UI¹⁵⁴
- Kendo-UI¹⁵⁵
- Zino-UI¹⁵⁶
- EasyUI¹⁵⁷
- Wijmo¹⁵⁸
- extJS¹⁵⁹
- D3¹⁶⁰

A grande vantagem desse tipo de biblioteca é possibilidade de fazer códigos com interatividade rica com pouco código. Como exemplo, o exemplo de Grid¹⁶¹ 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.

¹⁵⁰ <https://github.com/thejameskyle/babel-handbook/blob/master/translations/pt-BR/user-handbook.md>

¹⁵¹ <https://github.com/addyosmani/es6-tools#transpilers>

¹⁵² <http://es6-features.org>

¹⁵³ <https://kangax.github.io/compat-table/es6/>

¹⁵⁴ <http://jqueryui.com>

¹⁵⁵ <http://demos.telerik.com/kendo-ui/>

¹⁵⁶ <http://zinoui.com>

¹⁵⁷ <http://www.jeasyui.com>

¹⁵⁸ <http://wijmo.com>

¹⁵⁹ <https://www.sencha.com/products/extjs/>

¹⁶⁰ <https://d3js.org>

¹⁶¹ <http://demos.telerik.com/kendo-ui/grid/editing>

8.5 Framework MV* JavaScript

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 38.

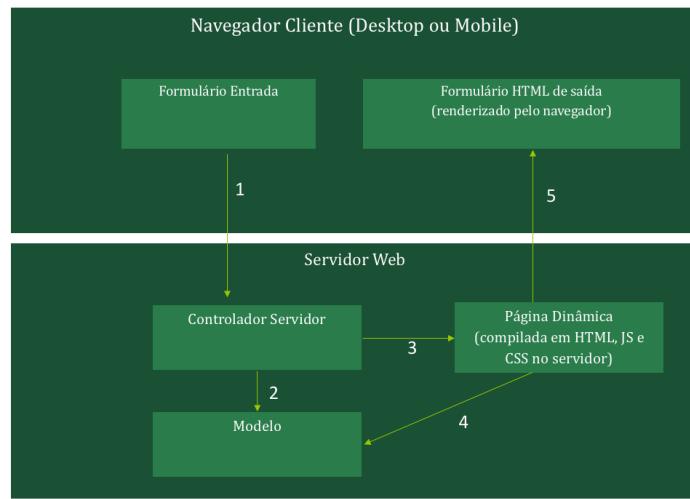


Figura 38: Modelo MVC Web clássico

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 39.

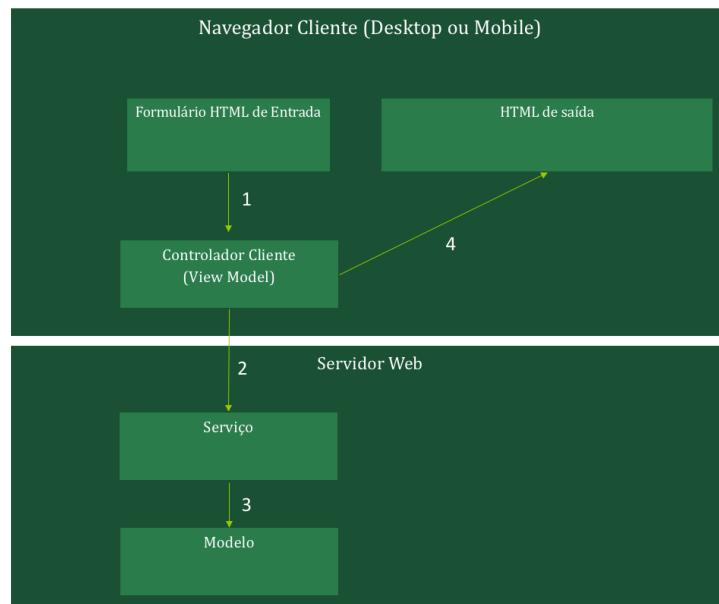


Figura 39: Modelo MV* (MVP/MVVM) Web

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 NodeJS (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 AngularHub¹⁶² mantém online uma coleção de tutoriais do AngularJS e mostra essa dinâmica de interação de forma bastante didática.

A partir do AngularJS 1.0, outros frameworks com a mesma arquitetura Web surgiram e se popularizaram. Exemplos incluem o Ember¹⁶³, Backbone.JS¹⁶⁴, AureliaJS¹⁶⁵, AngularJS 2.0¹⁶⁶. A respeito desses frameworks, existem bons sítios que mantêm informações e comparativos como o disponibilizado aqui¹⁶⁷.

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.

8.6 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:

8.6.1 Gerenciadores de dependências

Similar ao NuGet¹⁶⁸ para .NET ou o Maven para Java EE, a comunidade JavaScript desenvolveu ferramentas para gerir dependências. O Bower¹⁶⁹ é uma ferramenta bastante popular nesse sentido. O WebPack¹⁷⁰ e o npm¹⁷¹ são

¹⁶² <http://www.angularjshub.com/examples/>

¹⁶³ <http://emberjs.com>

¹⁶⁴ <http://backbonejs.org>

¹⁶⁵ <http://aurelia.io>

¹⁶⁶ <https://angular.io>

¹⁶⁷ <http://todomvc.com>

¹⁶⁸ <https://www.nuget.org>

¹⁶⁹ <http://bower.io>

¹⁷⁰ <https://webpack.github.io>

¹⁷¹ <https://www.npmjs.com>

também bastante utilizados. Uma outra biblioteca útil para o empacotamento de módulos e facilitar a gestão de dependências é o Browserify¹⁷².

8.6.2 Executores de Tarefas

Similar ao Make¹⁷³ para C, C++ e C# ou o Ant e Maven para Java EE, a comunidade JavaScript desenvolveu ferramentas como o Gulp¹⁷⁴ e Grunt¹⁷⁵. Em linhas gerais, elas processam os arquivos de um sítio Web cliente e executam tarefas como minificação¹⁷⁶, compressão de imagens, análise do uso de boas 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.

8.6.3 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 JUnit¹⁷⁷, Visual Studio Unit Test¹⁷⁸ ou o NUnit¹⁷⁹. Algumas soluções nesse sentido incluem o QUnit¹⁸⁰ para testes de unidade e TDD, o Jasmine¹⁸¹ para BDD (Behavior Driven Development) e Karma¹⁸² para a execução de testes.

8.7 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 JSConf¹⁸³, 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.js¹⁸⁴.

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 Figura 2 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

¹⁷² <http://browserify.org>

¹⁷³ <https://www.cs.umd.edu/class/fall2002/cmsc214/Tutorial/makefile.html>

¹⁷⁴ <http://gulpjs.com>

¹⁷⁵ <http://gruntjs.com>

¹⁷⁶ 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.

¹⁷⁷ <http://junit.org>

¹⁷⁸ <https://www.visualstudio.com/en-us/docs/tfvc/create-and-run-unit-tests-vs>

¹⁷⁹ <http://www.nunit.org>

¹⁸⁰ <http://www.nunit.org>

¹⁸¹ <http://jasmine.github.io/edge/introduction.html>

¹⁸² <https://karma-runner.github.io/0.13/index.html>

¹⁸³ <http://jsconf.com>

¹⁸⁴ <https://nodejs.org/en/>

aplicações corporativas em larga escala, é importante conhecer alguns módulos usados em conjunto com o Node.js. Alguns desses módulos incluem:

- ExpressJS¹⁸⁵ – Biblioteca para gerir requisições HTTP de forma facilitada a aplicativos móveis e Web.
- LoopBack¹⁸⁶ ¹⁸⁷ - 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.
- Restify¹⁸⁸ – Módulo para facilitar a criação de serviços REST.
- PassportJS¹⁸⁹ - 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-mysql¹⁹⁰ – Biblioteca para acesso a aplicações MySQL com Node.
- Node-OracleDB¹⁹¹ – Driver mantido pela própria Oracle¹⁹² para acesso aos bancos de dados Oracle em aplicações Node.
- MSSql¹⁹³ – Driver para acesso a banco de dados SQL Server em aplicações Node.
- Mongoose¹⁹⁴ – Framework para manipulação do banco de dados Non-SQL MongoDB.
- Azure¹⁹⁵ – Facilitador para implantação de aplicações Node em ambiente Microsoft Azure, mantido pela própria Microsoft.
- Mocha¹⁹⁶ – Biblioteca para testes de unidade para aplicações Node.
- Pm2¹⁹⁷ - Gerenciamento de processo de produção para aplicações Node, com suporte a clusters, balanceamento de carga e tolerância a falhas.
- NodeMon¹⁹⁸ – Biblioteca utilitária para facilitar a monitoração de mudanças no seu código fonte e recarregar a sua aplicação Web.
- Commander¹⁹⁹ – Módulo facilitador de execução de programas de linhas de comando em aplicações Node.
- Nodemailer²⁰⁰ - Biblioteca para a manipulação de e-mails em aplicações Node.
- Request²⁰¹ – Biblioteca ainda mais simples que o Express para a manipulação de requisições HTTP.
- Hapi²⁰² - Um outro framework HTTP para desenvolvimento de aplicações Web em Node.
- Bluebird²⁰³ – Biblioteca para facilitar a escrita de programas concorrentes, com suporte à primitiva de programação *promise*²⁰⁴.
- Async²⁰⁵ - Módulo utilitário para fornecer funções de manipulação de código assíncrono em JavaScript.
- Stomp-client²⁰⁶ - Módulo utilitário para clientes do protocolo de fila de mensagens Stomp²⁰⁷, suportado em implementações como o Apache ActiveMQ²⁰⁸ e outros sistemas de filas de mensagem.
- Numerals.js²⁰⁹ - Biblioteca utilitária para manipular e formatar números.

¹⁸⁵ <http://expressjs.com/pt-br/>

¹⁸⁶ <http://loopback.io>

¹⁸⁷ <http://loopback.io/resources/#compare>

¹⁸⁸ <http://restify.com>

¹⁸⁹ <http://passportjs.org>

¹⁹⁰ <https://www.npmjs.com/package/node-mysql>

¹⁹¹ <https://www.npmjs.com/package/oracledb>

¹⁹² http://www.oracle.com/technetwork/database/database-technologies/scripting-languages/node_js/index.html

¹⁹³ <https://www.npmjs.com/package/mssql>

¹⁹⁴ <http://mongoosejs.com>

¹⁹⁵ <https://azure.microsoft.com/pt-br/develop/nodejs/>

¹⁹⁶ <http://mochajs.org>

¹⁹⁷ <http://pm2.keymetrics.io>

¹⁹⁸ <http://nodemon.io>

¹⁹⁹ <https://www.npmjs.com/package/commander>

²⁰⁰ <https://www.npmjs.com/package/nodemailer>

²⁰¹ <https://www.npmjs.com/package/request>

²⁰² <http://hapijs.com>

²⁰³ <http://bluebirdjs.com/docs/why-promises.html>

²⁰⁴ https://en.wikipedia.org/wiki/Futures_and_promises

²⁰⁵ <https://github.com/caolan/async/blob/v1.5.2/README.md>

²⁰⁶ <https://www.npmjs.com/package/stomp-client>

²⁰⁷ <http://stomp.github.io>

²⁰⁸ <http://activemq.apache.org/stomp.html>

²⁰⁹ <http://numeraljs.com/>

- MomentJS²¹⁰ - Biblioteca utilitária para manipular datas.
- ShouldJS²¹¹ – Biblioteca utilitária para auxiliar a escrita de testes BDD em JavaScript.
- Nock²¹² - Biblioteca para criar mocks e simulações em testes de unidade JavaScript.

O portal do npmjs mantém uma lista²¹³ 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 Meteor²¹⁴, potente framework para o desenvolvimento de aplicações JavaScript.

8.8 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 3 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	AngularJS Ember Backbone	Classes C#	Servlets
Motores de visualização	Mustache, Underscore, EmbeddedJS, HandleBarJS, Jade ²¹⁵	Razor ou WebForms	JSTL
APIs RESTful	Node Restify, LoopBack	ASP.NET Web API	JAX-RS
APIs WS-*	LoopBack	WCF	JAX-WS
Mapeamento Objeto Relacional	LoopBack Meteor	Entity Framework	JPA
Acesso a banco de dados	Node node-oracledb, mssql, mongoose	ADO.NET	JDBC

²¹⁰ <http://momentjs.com>

²¹¹ <https://github.com/shouldjs/should.js>

²¹² <https://github.com/node-nock/nock>

²¹³ <https://www.npmjs.com/browse/star>

²¹⁴ <https://www.meteor.com>

²¹⁵ <http://www.creativeblog.com/web-design/template-engines-9134396>

Segurança	Node Passport	ASP.NET Identity	JAAS
Filas de Mensagens	stomp-client	MSMQ	JMS
Máquina Virtual	Docker com Node.JS	.NET Framework ou .NET Core	JVM

Tabela 9: Comparativo arquitetural entre Java Script, Java EE e .NET

8.9 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:

- 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 Hub²¹⁶, 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 GitHubStars²¹⁷ 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 40.

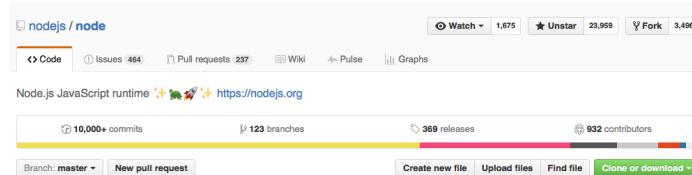


Figura 40: Sistema de reputação GitHub

8.10 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.

²¹⁶ <https://orionhub.org/>

²¹⁷ <https://github.com/blog/1204-notifications-stars>

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.

9 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.

9.1 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.

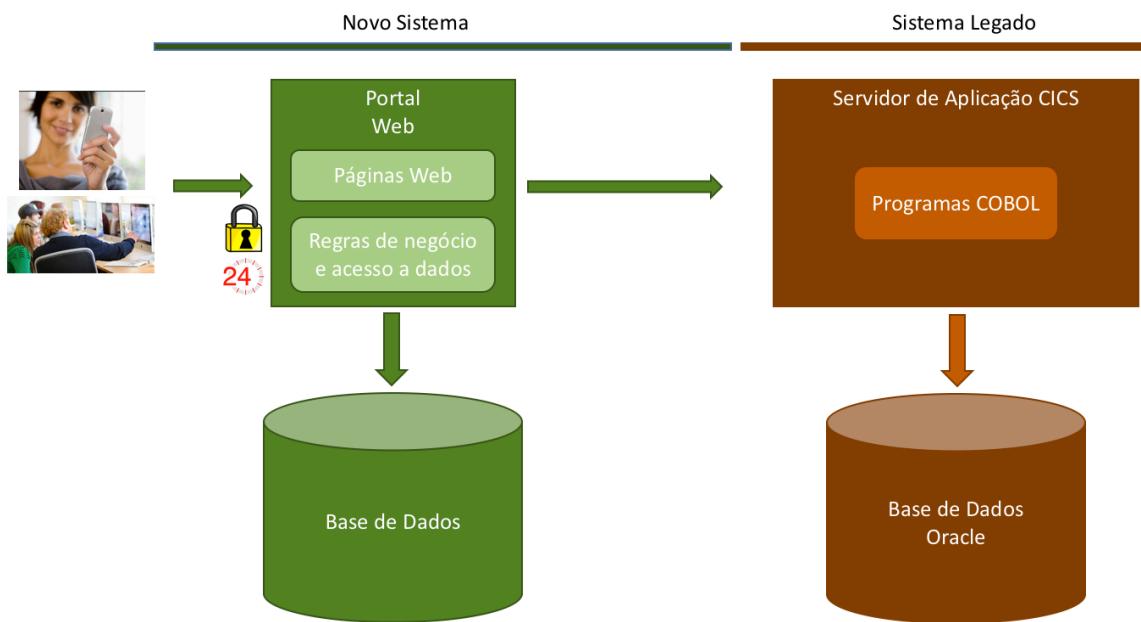


Figura 41: Visualização de Negócio do Sistema Acadêmico

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.

9.2 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:

9.2.1 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.

9.2.2 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.

9.2.3 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.

9.2.4 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.

9.2.5 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.

9.3 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.

9.4 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.

9.5 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 42 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.

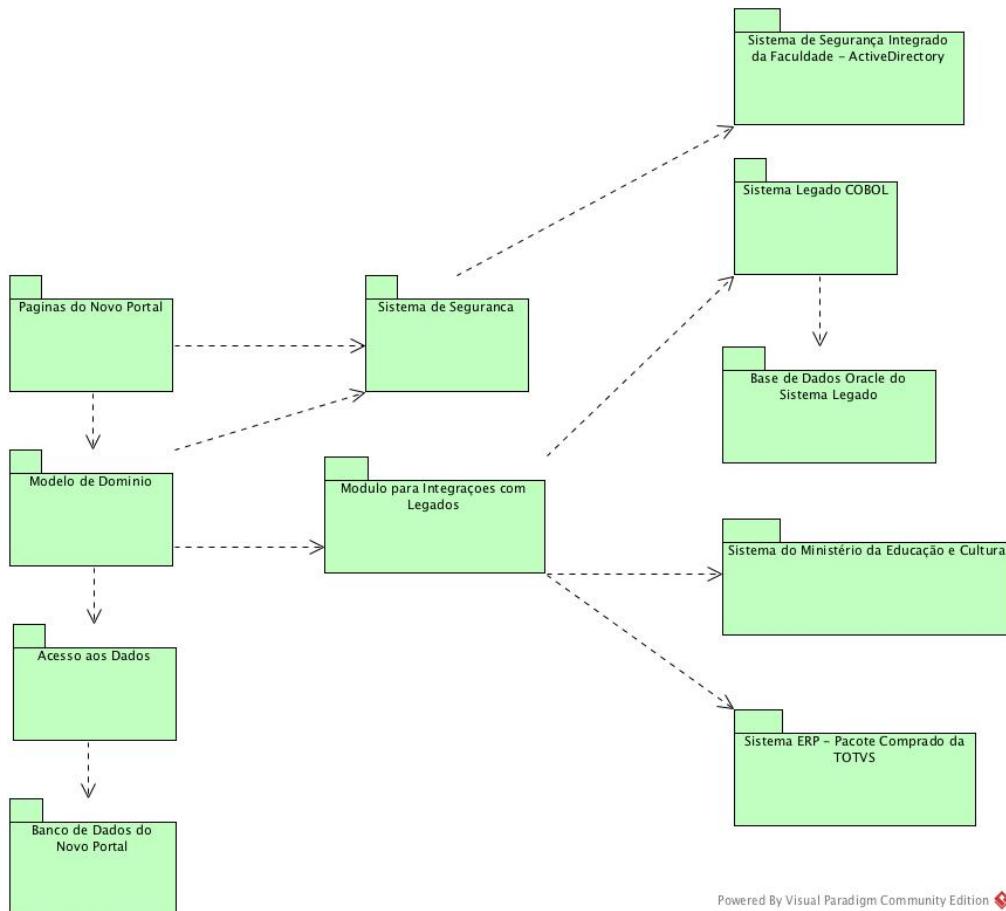


Figura 42: Visualização Lógica do Sistema Acadêmico

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 negócios em um pacote especial chamado de modelo de domínio. Esse pacote pode ser refinado conforme mostrado a Figura 43. Em termos técnicos, cada pacote aqui mostrado se materializará como *package* em Java, um *namespace* em C# ou um *module* em JavaScript.

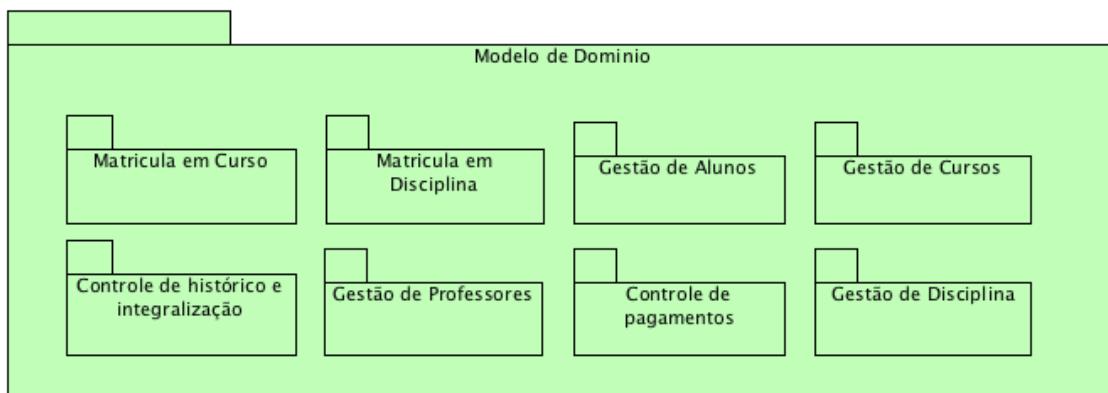


Figura 43: Modelo de Domínio do Sistema Acadêmico

9.6 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.

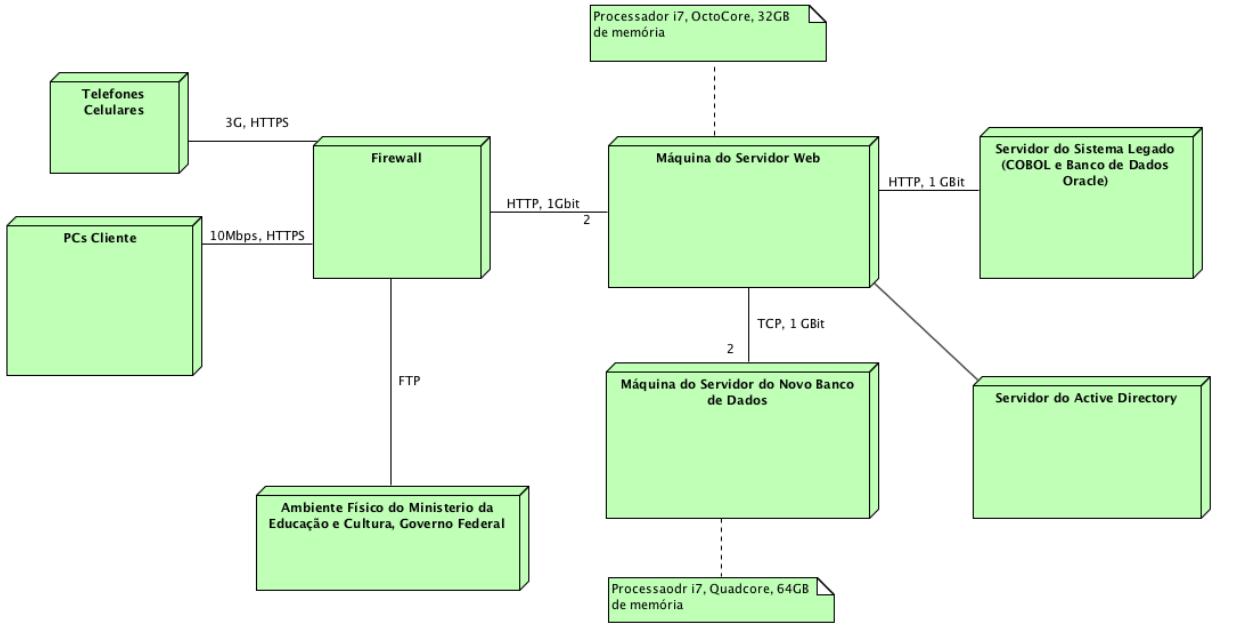


Figura 44: Topologia do Sistema Acadêmico

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).

9.7 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 45, Figura 46 e Figura 47. Cada caixa nesse

diagrama representa um componente arquitetural, que em termos práticos é uma DLL, assembly, JAR, biblioteca ou um módulo JavaScript.

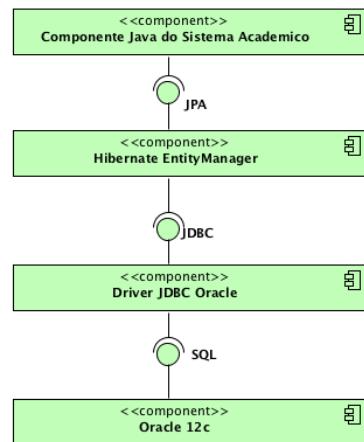


Figura 45: Visualização de Persistência em Java EE

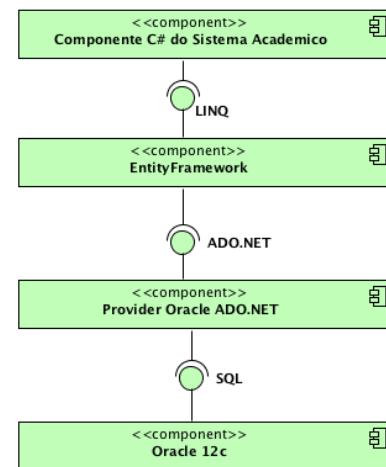


Figura 46: Visualização de Persistência em .NET

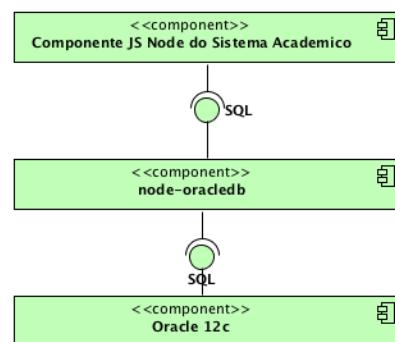


Figura 47: Visualização de Persistência em Node.JS

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 45 poderia indicar os componentes banco de dados versão Oracle 12.1.0.2.0, driver JDBC versão ojdbc7²¹⁸ e versão do Hibernate ORM 5.2.0²¹⁹. É sempre válido testar se essas combinações interoperam com um teste rápido ou uma prova de conceito mais elaborada se o risco é maior.

9.8 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.

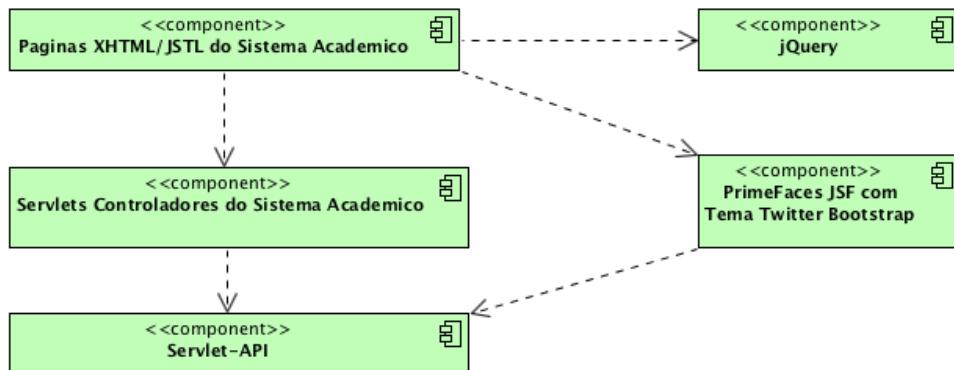


Figura 48: Visualização de Apresentação em Java EE

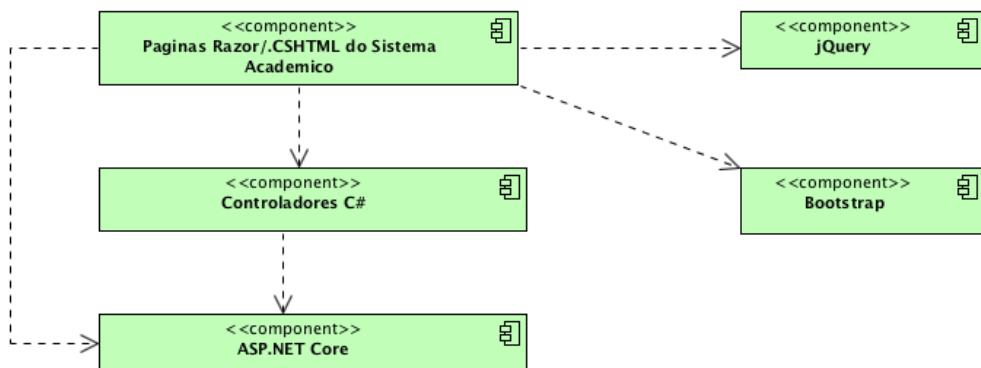


Figura 49: Visualização de Apresentação em ASP.NET Core

²¹⁸ <http://www.oracle.com/technetwork/database/features/jdbc/jdbc-drivers-12c-download-1958347.html>

²¹⁹ <http://hibernate.org/orm/downloads/>

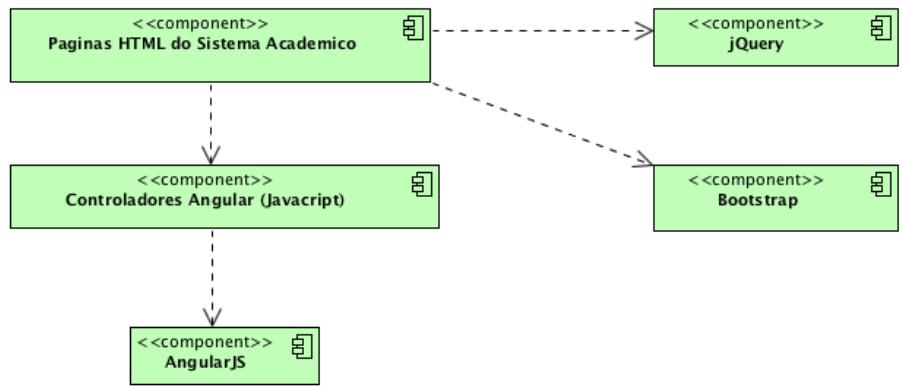


Figura 50: Visualização de Apresentação em AngularJS/Node.JS

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.

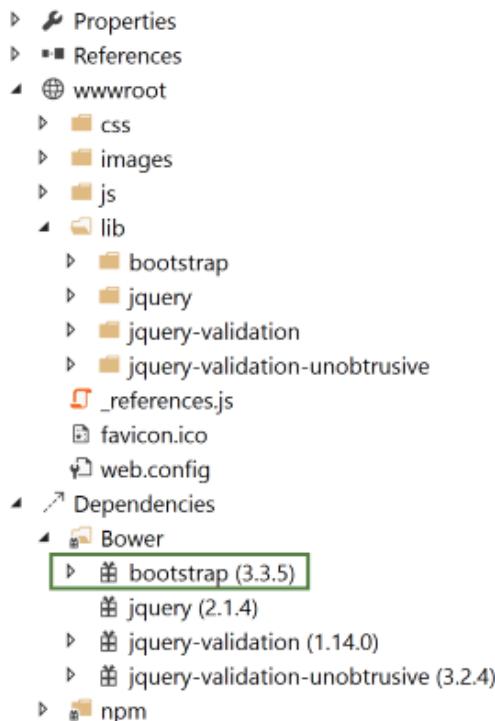


Figura 51: Visualização de Componentes de Apresentação em ASP.NET Core

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.

9.9 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.

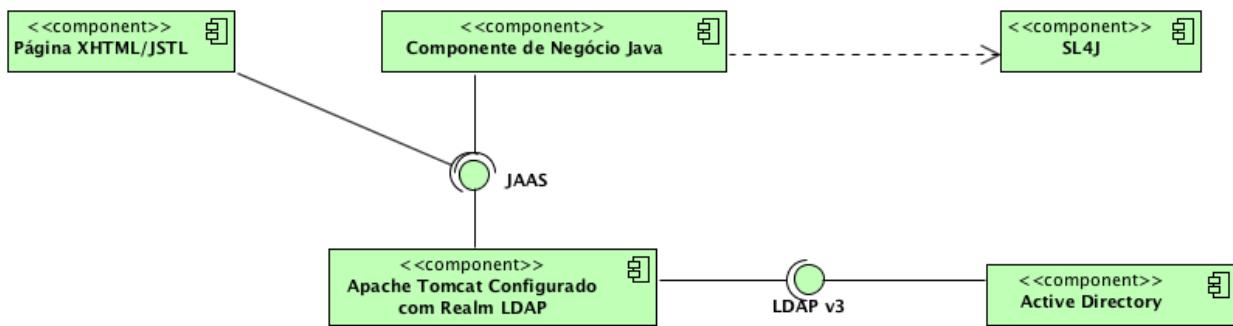


Figura 52: Visualização de Segurança em Java EE

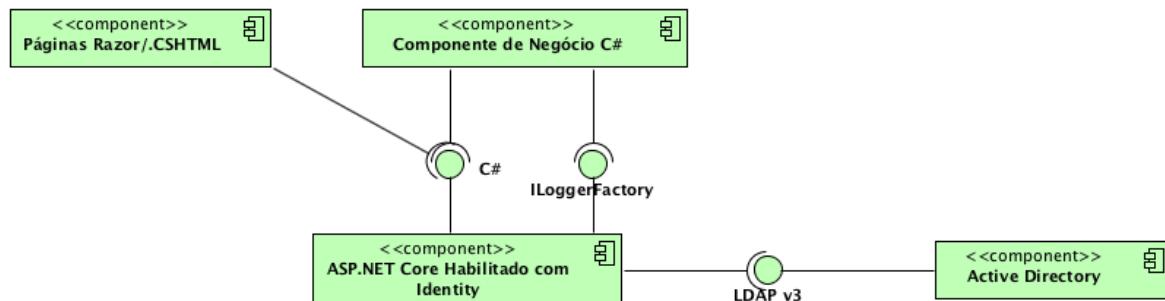


Figura 53: Visualização de Segurança em .NET

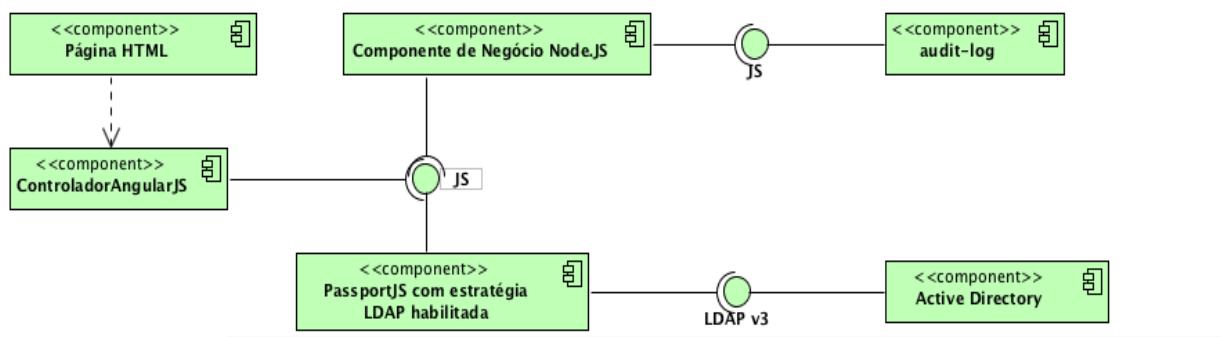


Figura 54: Visualização de Segurança em Node.js

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 ApacheTomcat²²⁰ ²²¹.
- Configuração e uso do Node Passport com AD²²²

9.10 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 55 apresenta estes estilos.



Figura 55: 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 REST²²³. Com essa premissa, podemos escolher a chamada de procedimentos remotos como paradigma de integração. Isso levaria à seguinte solução genérica, que envolve que o

²²⁰ <https://tomcat.apache.org/tomcat-7.0-doc/realm-howto.html#JNDIRealm>

²²¹ <http://stackoverflow.com/questions/267869/configuring-tomcat-to-authenticate-using-windows-active-directory>

²²² <https://www.npmjs.com/package/passport-ldapauth>

²²³ https://www.ibm.com/developerworks/community/blogs/e4210f90-a515-41c9-a487-8fc7d79d7f61/entry/cics_atom_support_restful_service

time COBOL crie uma fachada de serviços de negócio que será chamada pelos componentes de negócios 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.

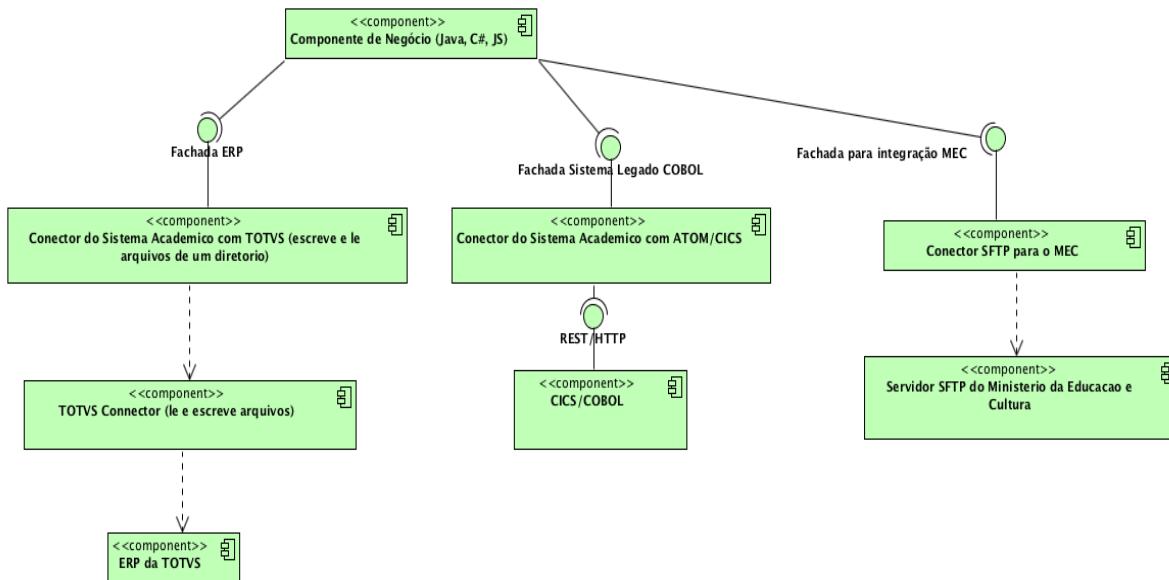


Figura 56: Visualização de Integração

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.

9.11 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.

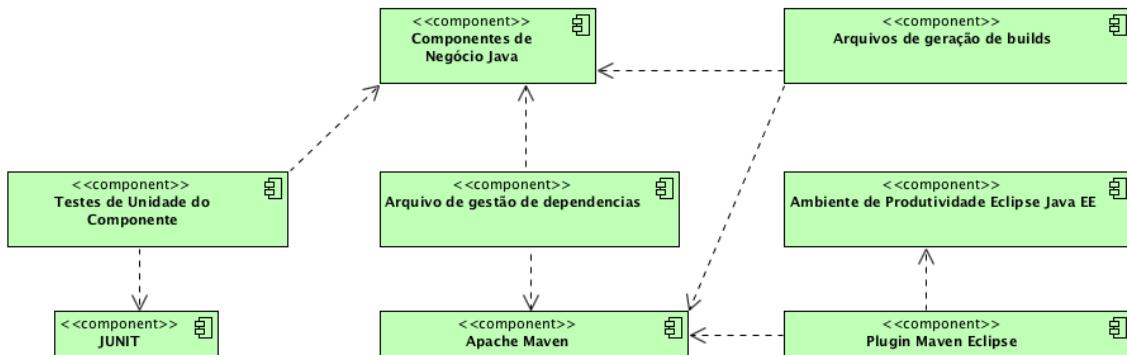


Figura 57: Visualização de Manutenibilidade em Java EE

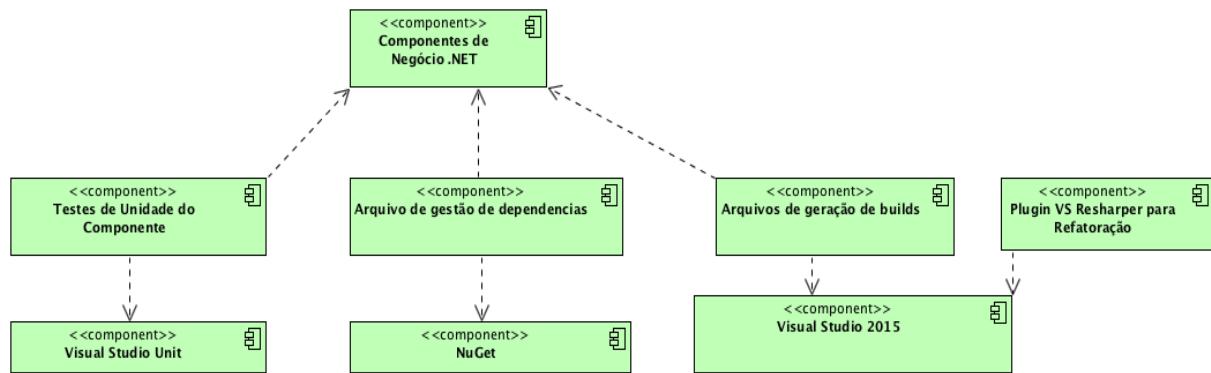


Figura 58: Visualização de Manutenibilidade em ASP.NET

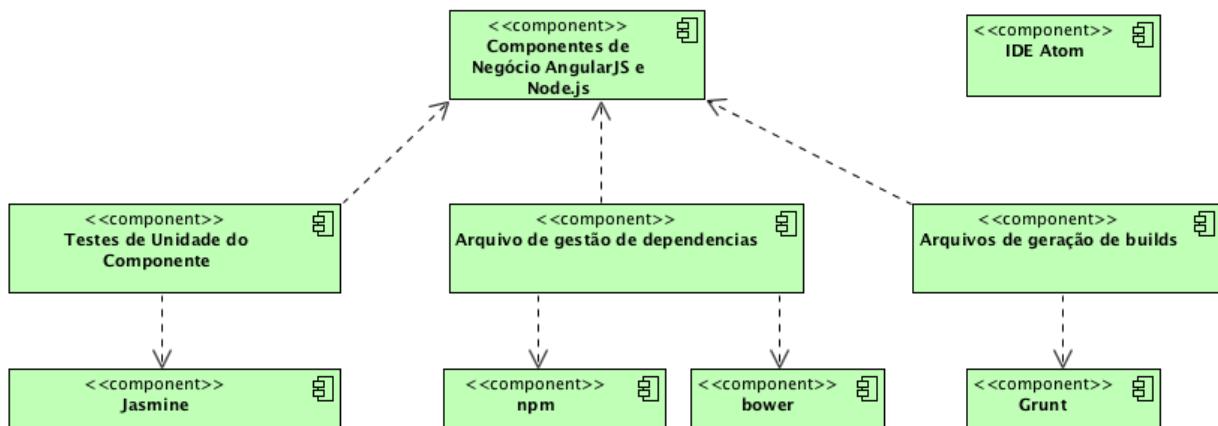


Figura 59: Visualização de Manutenibilidade em Node.js

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.

9.12 Passo 12 – Alocação de Componentes aos Nodos Físicos

O diagrama da Figura 44 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.

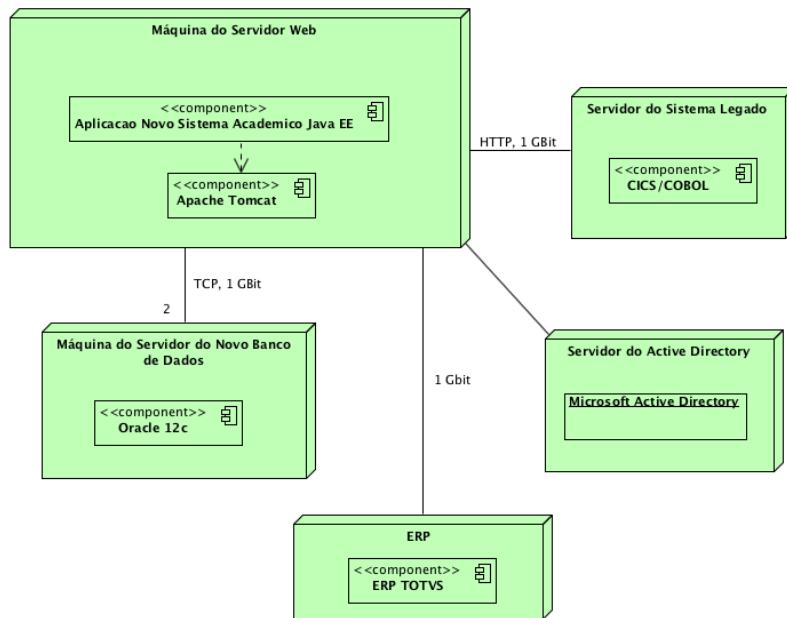


Figura 60: Topologia Física com Componentes – Visão Java EE

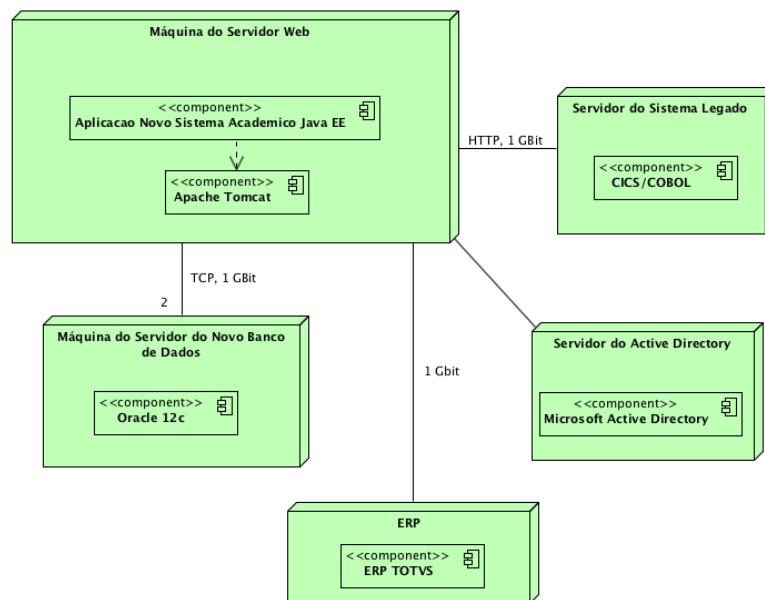


Figura 61: Topologia Física com Componentes – Visão .NET

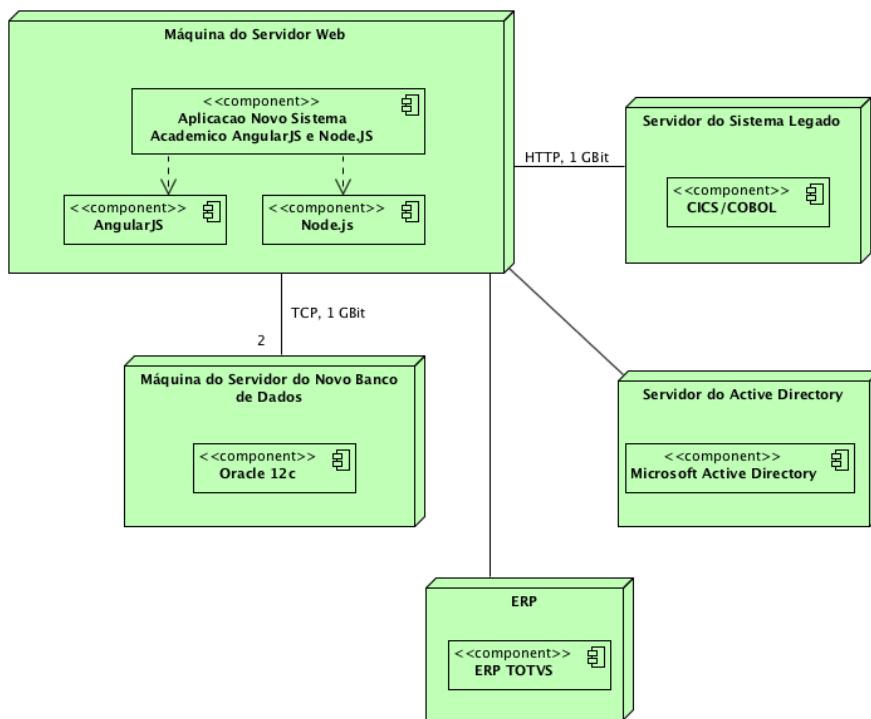


Figura 62: Topologia Física com Componentes – Visão Angular e Node.js

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.

9.13 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.

9.14 Modelos de um Documento de Arquitetura de Software

Os processos de software como o RUP²²⁴, OpenUP²²⁵, SEI V&B (Views and Beyond)²²⁶ 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.

²²⁴ https://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf

²²⁵ <http://epf.eclipse.org/wikis/openuppt/>

²²⁶ <http://www.sei.cmu.edu/architecture/tools/document/viewsandbeyond.cfm>

- 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.

10 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.

10.1 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 negócio 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 2016*²²⁷, empresas de alta maturidade em práticas DevOps tem o tempo de entrega de aplicações (MTTD) acelerado em 2500 vezes, são 24 vezes mais eficientes para recuperar falhas (MTTR) e tem 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 Group²²⁸ mostra como empresas como *Walmart*, e *Staples* aumentaram a eficiência operacional de suas TIs com o uso de práticas DevOps.

²²⁷ <https://puppet.com/resources/white-paper/2016-state-of-devops-report>

²²⁸ <https://puppet.com/resources/analyst-report/value-of-it-automation>

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 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.

10.2 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.

10.3 Práticas DevOps Básicas

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.

10.3.1 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.

10.3.2 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.

10.3.3 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.

10.3.4 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.

10.3.5 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.

10.3.6 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.

10.3.7 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*²²⁹, mecanismo extensível de automação de políticas de gestão de código. Outras como o GitLab, VSTS ou Atlassian Bamboo já possuem estes mecanismos embutidos.

10.3.8 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.

10.3.9 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

²²⁹ <https://git-scm.com/book/pt-br/v1/Customizando-o-Git-Um-exemplo-de-Pol%C3%ADtica-Git-Forcada>

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.

10.4 Práticas DevOps Avançadas

10.4.1 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.

10.4.2 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 contínua 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.

10.4.3 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.

10.4.4 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 contínua (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 contínua (Continuous Deployment)* é o processo de copiar o build gerado no processo de integração para ambientes intermediários como QA ou homologação. A cópia do build é também chamada de promoção por alguns autores.

A *entrega contínua (Continuous Delivery)* é o processo de implantação contínua que busca promover os builds do ambiente de homologação para o ambiente de produção.

10.4.5 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*²³⁰ 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.

10.4.6 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 adequada 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 Docker²³¹ 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.

```
---
- 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 63: Arquivo de script Ansible

Fonte: <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

²³⁰ <http://martinfowler.com/articles/feature-toggles.html>

²³¹ <http://docker.com>

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.

10.4.7 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.

10.4.8 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 publicou²³² o uso desta prática nos seus ambientes de produção e a disponibilização da sua ferramenta de injeção de falhas chamada *Simian Army*²³³. Ela permite estabelecer um mecanismo de antifragilidade²³⁴ na sua aplicação, tornando-a melhor ao longo do tempo à medida que ela seja estressada.

10.4.9 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.

²³² <https://www.infoq.com/br/news/2012/08/netflix-chaos-monkey>

²³³ <https://github.com/Netflix/SimianArmy>

²³⁴ 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.

10.4.10 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.

10.5 Ferramentas DevOps

Práticas DevOps introduzem aceleradores arquiteturais. 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.

10.5.1 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.

10.5.2 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.

10.5.3 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.

10.5.4 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.

10.5.5 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 desenvolvedor 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.

10.5.6 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.

10.5.7 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.

10.5.8 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.

10.5.9 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

10.5.10 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.

10.5.11 Ferramentas de Injeção de Falhas

Permitem injetar falhas em aplicações nos ambientes de homologação e produção, como por exemplo o *Netflix Simian Army*. Esta suíte²³⁵, 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 desliga. 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 desliga. 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.

10.5.12 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 de releases. A Amazon WebServices talvez seja o maior expoente deste segmento, com uma rica coleção de serviços de IAAS,

²³⁵ <http://techblog.netflix.com/2011/07/netflix-simian-army.html>

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.