



Melhores Práticas de **Arquitetura** **de Software** na era da Nuvem

Apoio:  **trybe**
Um novo curso para sua vida

Melhores Práticas de Arquitetura de Software na era da Nuvem

Melhores Práticas de Arquitetura de Software na era da Nuvem

Trybe, Francisco Isidro Massetto, Elder Moraes, Sérgio Lopes, Sandro Giacomozi, Leandro Domingues, Mauricio Salatino, Karina Varela e Otavio Santana

Esse livro está à venda em
<http://leanpub.com/manual-arquitetura-software>

Essa versão foi publicada em 2021-06-01



Esse é um livro [Leanpub](#). A Leanpub dá poderes aos autores e editores a partir do processo de Publicação Lean. [Publicação Lean](#) é a ação de publicar um ebook em desenvolvimento com ferramentas leves e muitas iterações para conseguir feedbacks dos leitores, pivotar até que você tenha o livro ideal e então conseguir tração.

© 2021 Trybe, Francisco Isidro Massetto, Elder Moraes, Sérgio Lopes, Sandro Giacomozi, Leandro Domingues, Mauricio Salatino, Karina Varela e Otavio Santana

Conteúdo

A importância de aprender conceitos ao invés de novos frameworks	11
Por que estudar Análise de Algoritmos, Estruturas de Dados, Sistemas Operacionais, Arquitetura de Computadores, Grafos, entre outros?	12
A importância de um estudo mais aprofundado em disciplinas teóricas	14
O ponto-chave: usar frameworks, fazer as tarefas manualmente, ou melhor, criar seu próprio?	20
Tenho lido sobre DDD, para onde devo ir depois?	23
Java na nuvem	24
Evoluindo seu monolito na prática, usando DDD	27
Resumindo	66
Clean code	68
Modelos Ricos	68
Lombok: problema ou solução?	87
Clean Architecture	89
Granularidade de camadas	92
Conclusão	94
Refatoração	95
Medo de alterar o código que não é seu	96

CONTEÚDO

A importância dos testes automatizados na hora refatorar	98
Refatoração contínua do código	100
Qual sua motivação para refatorar o código?	101
Conclusão	102
NoSQL vs. SQL	103
NoSQL	105
NoSQL e suas classes	108
Conclusão	120
Arquitetura de microsserviços	121
Arquitetura monolítica	121
Microsserviços	124
Migração de um monolito para microservices	128
Os erros mais comuns com microservices	128
Conclusão	131
Cloud	133
Cloud-Native ou Cloud-Enabled?	134
A jornada cloud-native	150
IaaS, PaaS e SaaS: uma perspectiva arquitetural	151
Kubernetes - quando usar e quando não usar	157
Kubernetes Vanilla e seus sabores	159
Quando não usar Kubernetes	167
Conclusão	169
Precisamos falar sobre atualizações	170
Com que frequência você entrega alguma coisa para seu (sua) usuário (a)?	171
Os problemas de não atualizar a versão de JVM	173
Utilidade versus Hype	181
Destrinchando performance de aplicações	184
Introdução e conceitos	184
Agilidade versus Performance	185

CONTEÚDO

Como medir a performance?	186
Entendendo e separando os componentes	191
Monitorando a performance por componente	193
Monitorando a performance em sistemas distribuídos	195
Mapeamento Objeto Relacional	196
Conclusão	198
Apêndice A: Segurança	199
Práticas de Segurança	199
Bibliografia	204



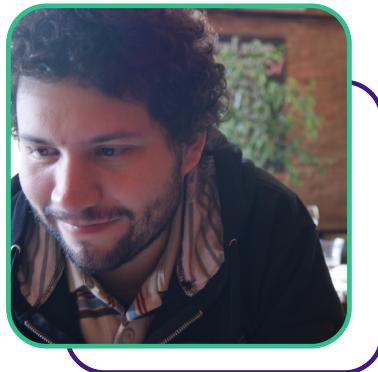
- **Nome:** Otavio Santana
- **Cargo:** Staff Engineer
- **Bio:** Capacitando pessoas desenvolvedoras em todo o mundo a entregar softwares melhores, com mais rapidez e escalabilidade na nuvem. Otavio é um engenheiro de software apaixonado por foco em nuvem e tecnologia Java. Ele tem experiência principalmente em aplicações de persistência poliglotas e de alto desempenho em finanças, mídia social e e-commerce. Otavio é membro de Grupos de Especialistas e Líder Especialista em vários comitês executivos JCP e JSRs. Ele está trabalhando em vários projetos da Apache e Eclipse Foundation, como Apache Tamaya, MicroProfile, Jakarta EE, onde está liderando a primeira especificação em Jakarta EE com Jakarta NoSQL. Líder do JUG e palestrante global nas conferências JavaOne e Devoxx, Otavio recebeu reconhecimento por suas contribuições para OSS, como o JCP Outstanding Award, membro do ano e JSR inovador, Duke's Choice Award e Java Champion Award, para citar alguns.



- **Nome:** Sérgio Lopes
- **Cargo:** Especialista em TI no Banco Itaú S/A
- **Bio:** Desenvolvedor C++, desenvolvedor Java, engenheiro de software, especialista em TI em back-end. Trabalhando em um dos mais robustos Internet Bankings da América Latina. Atuando na criação do framework de desenvolvimento chamado *Universal*, que é o sistema central do Internet Banking Itaú desde 2001 e até os dias de hoje ainda em operação, trabalhando no desenvolvimento de ferramentas de suporte para esse framework, suas especificações e sua manutenção, em C/C++. Desenvolvimento do Framework Java, que veio substituir o framework *Universal*, atuando nele desde 2014. Criação de mecanismos de coexistência entre o legado e este novo Framework, responsável por atender às necessidades transversais dos canais digitais, fornecendo ferramentas e serviços para acelerar o trabalho de outros engenheiros. Trabalhando com tecnologias como: JAVA, Spring Framework, Spring Boot, Docker, Windows/Linux, SQL, JavaScript, Hibernate, Application Server: JBoss EAP.6.X, Testes automatizados com (JUnit4, DBUnit, JMock, Mockito). Eclipse IDE, Maven, Git, Jenkins, Nexus, C++, Visual Studio, Artifactory, Sonar.



- **Nome:** Karina Varela
- **Cargo:** Gerente Técnico Principal de Marketing de Produtos, Red Hat
- **Bio:** Karina M. Varela tem experiência de mais de dez anos em TI, trabalhando em funções como desenvolvedora de aplicações, arquiteta de software, consultora, líder de tecnologia e gerenciamento de marketing de produto. Com uma sólida formação em desenvolvimento de software, ela tem experiência profissional em planejamento, arquitetura, entrega e solução de problemas de software crítico em ambientes empresariais de diferentes setores ao redor do mundo. A partir de 2020, Karina está trabalhando com a Unidade de Negócios de Serviços de Aplicativos da Red Hat como Gerente de Marketing Técnico, especialista no assunto de Automação de Negócios. Ela é membro da comunidade SouJava, tem paixão por ajudar as comunidades e gosta especialmente de projetos e iniciativas de código aberto. Outro hobby é falar em conferências e já palestrou em conferências como Campus Party, TDC e Women Who Code.



- **Nome:** Mauricio Salatino (Salaboy)
- **Cargo:** Engenheiro de Software
- **Bio:** Mauricio é engenheiro de software na Camunda (<http://www.camunda.com>) e instrutor na LearnK8s (<http://learnk8s.io>). Trabalha com o Kubernetes há cinco anos, treinando equipes e desenvolvendo aplicações nativas da nuvem. Em sua jornada, participou de vários projetos de código aberto, incluindo Zeebe, Jhipster, Spring Cloud e Jenkins X. Anteriormente, Mauricio trabalhou na Red Hat/JBoss, no departamento de Engenharia de Automação de Negócios. Apresentou-se na Kubecon 2019 San Diego e na Kubecon 2020 Amsterdam. Está atualmente envolvido com a CD Foundation (<http://cd.foundation>) e com o projeto Jenkins X (como membro do comitê de direção de bootstrap).



- **Nome:** Elder Moraes
- **Cargo:** Developer Advocate, Red Hat
- **Bio:** Elder ajuda pessoas desenvolvedoras Java a trabalhar em grandes projetos, orientando-os(as) sobre como construir e entregar aplicações seguras, disponíveis e rápidas do lado do servidor. Ele é o autor do Jakarta EE Cookbook e membro do conselho do SouJava, um dos maiores JUGs do mundo. Como defensor do desenvolvedor, ele compartilha experiências e melhores práticas por meio de conteúdo online e em eventos internacionais como JavaOne, The Developers Conference, QCon, Oracle Code One, Campus Party e Devnexus.



- **Nome:** Sandro Giacomozi
- **Cargo:** Engenheiro de Software, TOTVS
- **Bio:** Sandro ajuda pessoas desenvolvedoras Java que trabalham em aplicações corporativas a se tornarem especialistas em Java e DevOps, praticando as habilidades certas. Voluntário e palestrante. Seu objetivo na indústria de software é tornar as organizações e as pessoas mais ágeis por meio de processos e ferramentas. Entregas mais rápidas, eficientes e de qualidade. Pessoas e tecnologia alinhadas à satisfação e entrega ao cliente.



- **Nome:** Francisco Isidro
- **Cargo:** Professor, Pesquisador da Universidade Federal do ABC
- **Bio:** Professor Isidro é professor de Ciência da Computação e pesquisador com foco em Ensino de Fundamentos de Programação, Desenvolvimento de *Game Engines* e Computação em Nuvem. Isidro mantém um canal no Youtube que oferece conteúdo gratuito sobre Estruturas de Dados, Sistemas Operacionais, Desenvolvimento Web, Jogos e outros assuntos para todos os profissionais e estudantes que desejam aprimorar seus conhecimentos e entender os fundamentos da Ciência da Computação e Desenvolvimento de Software. Palestrante na Campus Party, The Developer's Conference, QCon e outras conferências, Isidro está sempre ajudando as comunidades de desenvolvedores oferecendo conteúdo técnico e orientação profissional.



- **Name:** Leandro Domingues
- **Job Title:** Fundador da Cluster Consultoria, MongoDB Champion / Consulting Engineer, Microsoft Data Platform MVP, Community Manager, Speaker
- **Bio:** MongoDB Champion, Microsoft Data Platform MVP, Top 50 Neo4j Certified com grande experiência em manutenção de grandes volumes de dados e aplicações críticas, após vários anos trabalhando com bancos de dados relacionais, voltei minha carreira para bancos NoSQL nos últimos 7 anos. Com toda a experiência acumulada em bancos de dados relacionais, hoje atuo ajudando pessoas e empresas a entrarem ou se manterem no universo de bancos NoSQL, participando desde a idealização de projetos até a manutenção de ambientes replicados e escaláveis, passando pela modelagem de dados até aumento de performance. Como desenvolvedor minha carreira foi trilhada com base em .NET, porém nos últimos anos me dedico a tecnologias OpenSource, me especializei em NodeJS e Python, manutenção de ambientes Linux, etc. Participo constantemente de eventos nacionais e internacionais em busca de atualização e conhecimento principalmente na área de OpenSource. Ministro treinamentos in-company e

turmas abertas de MongoDB e Neo4j.

- **Bio:** Professor Isidro é professor de Ciência da Computação e pesquisador com foco em Ensino de Fundamentos de Programação, Desenvolvimento de *Game Engines* e Computação em Nuvem. Isidro mantém um canal no Youtube que oferece conteúdo gratuito sobre Estruturas de Dados, Sistemas Operacionais, Desenvolvimento Web, Jogos e outros assuntos para todos os profissionais e estudantes que desejam aprimorar seus conhecimentos e entender os fundamentos da Ciência da Computação e Desenvolvimento de Software. Palestrante na Campus Party, The Developer's Conference, QCon e outras conferências, Isidro está sempre ajudando as comunidades de desenvolvedores oferecendo conteúdo técnico e orientação profissional.



Equipes realmente inovadoras contratam talentos de tecnologia na Trybe.

E o melhor: elas não pagam nada por isso

A Trybe é uma escola focada em resolver um dos principais problemas das empresas do Brasil: a falta de pessoas qualificadas para trabalhar com desenvolvimento de software.

Contrate conosco



A importância de aprender conceitos ao invés de novos frameworks

Escrever este capítulo pode não ser uma das tarefas mais fáceis, uma vez que o público-alvo deste material é o de desenvolvedores(as) com alto nível de experiência (*Senior Developers*). A intenção, portanto, deste texto não é de um acadêmico dizer o que profissionais devem estudar, mas, principalmente, gerar uma autocritica sobre muitos assuntos que poderiam ser explorados com maior profundidade (e até mesmo a aplicação da famosa expressão “ligar os pontos”) enquanto estávamos em processo de formação (na universidade).

Pois bem, como iniciar? Basicamente, a formação de profissionais de desenvolvimento tem como base seu *background* em um curso de Ciência (ou Engenharia) da Computação (ou outro bacharelado na área). Muitas dessas respostas podem estar em disciplinas já esquecidas por nós, como estudantes, e mesmo por profissionais.

O grande “mistério” que podemos tentar revelar aqui neste capítulo pode ser extremamente simples (ou tornar-se simples) a partir do momento em que identificamos o seguinte: as diferentes disciplinas que estudamos estão sempre interconectadas, mesmo que não pareça.

Por que estudar Análise de Algoritmos, Estruturas de Dados, Sistemas Operacionais, Arquitetura de Computadores, Grafos, entre outros?

Podemos interconectar disciplinas de Análise de Algoritmos com Programação Orientada a Objetos e Bancos de dados? Sim! E de uma forma mais comum do que se pode imaginar:

Considere o exemplo de uma busca de objetos em um banco através de um ORM. Dependendo da aplicação, compartilhamento de recursos, escalabilidade e também concorrência, temos que pensar em eficiência de algoritmos mesmo antes de sua implementação.

Ter consciência de que um Lazy Fetch com Hibernate, por exemplo pode gerar uma complexidade assintótica muito maior do que um Eager Fetch é fundamental para senior developers. Isso porque, por exemplo, em um sistema de recuperação de cabeçalhos de pedidos e seus respectivos itens, ocorre uma sequência de acessos ao banco de dados muito volumosa na primeira abordagem (*lazy*). Acessos para recuperar todos os cabeçalhos de pedidos e, para cada pedido, mais uma sequência de acessos ao banco de dados. Se observarmos um algoritmo com essas características, veremos algo próximo ao pseudocódigo a seguir:

```
1 recuperar conjunto de pedidos e armazenar em listaP  
2 para cada pedido P em listaP faça  
3     recuperar itens I[] do pedido P a partir de seu ID  
4 fim-para
```

De forma bastante geral, a execução das instruções pode ser descrita da forma abaixo:

- 1 1 execução da linha 1
- 2 N+1 execuções da linha 2 (for sempre tem o teste que retorna falso)
- 3 N execuções da linha 3

Desse modo, a complexidade desse algoritmo é de ordem $O(n)$. Isso significa que, à medida que o conjunto de valores recuperados na linha 1 aumenta, o tempo computacional aumenta proporcionalmente.

Podemos extrapolar ainda esse algoritmo e, para cada item, recuperar toda a lista de impostos, que pode gerar um algoritmo de complexidade quadrática $O(n^2)$. Observe:

- 1 recuperar conjunto de pedidos e armazenar em listaP
- 2 para cada pedido P em listaP[] faça
 - 3 recuperar itens I[] do pedido P a partir de seu ID
 - 4 para cada item I de P, faça
 - 5 recuperar lista L de impostos de I[]
 - 6 fim-para
- 7 fim-para

Nesse caso, analisando a complexidade assintótica, temos:

- 1 1 execução da linha 1
- 2 N+1 execuções da linha 2
- 3 N execuções das linhas de 3 a 5, o que implica:
 - 4 1 execução da linha 3
 - 5 N+1 execuções da linha 4
 - 6 N execuções da linha 5

Portanto, se somarmos todas as execuções, temos $1 + (N+1) + N * (1 + (N+1) + N)$, o que resulta num polinômio cujo maior expoente é 2 (N^2). Logo, nosso algoritmo de busca de pedidos, itens e impostos pode ser considerado um algoritmo de ordem quadrática. Isso porque apenas pensamos na recuperação da informação. Nem pensamos ainda na iteração desses valores, o que pode elevar ainda mais o tempo computacional. Agora imagine esse cenário em um sistema Web com algumas dezenas de milhares de pessoas conectadas. Se fizermos acessos ao disco (que é um dispositivo bastante lento, se comparado ao acesso à memória) de forma ineficiente para uma única requisição, como será o desempenho da aplicação para diversas requisições simultâneas?

A que conclusão podemos chegar depois de toda essa discussão? Talvez não seja tarefa do dia a dia de senior developers realizar a análise assintótica de algoritmos através da obtenção dos polinômios. Porém, para quem irá trabalhar com problemas mais complexos que a maioria dos casos do dia a dia, identificar complexidades (mesmo que de forma mais superficial, inicialmente) de algoritmos é fundamental para ter uma visão criteriosa de quais decisões tomar.

A importância de um estudo mais aprofundado em disciplinas teóricas

Sabemos que Estruturas de Dados é um assunto que, para muitos(as) estudantes em formação, é tenebroso e extremamente abstrato. Porém é necessário e fundamental termos conhecimento das estruturas básicas e avançadas para também entendermos as ferramentas que utilizamos em sistemas atuais.

Quer exemplos?

- **Filas:** Apache Kafka, Rabbit MQ, MQTT, entre outros, são exemplos de ferramentas que utilizam os conceitos de filas em suas implementações. Não apenas filas únicas, mas filas com prioridades, que também são objeto de estudo no período da graduação.
- **Pilhas:** Sua IDE faz uso de pilhas a todo momento para conferir se as chaves abertas foram fechadas corretamente (o parser da linguagem faz uso de pilhas quase que constantemente). Se os parênteses que definem as instruções em Clojure estão em conformidade. Isso apenas para citar exemplos imediatos.

Além desses exemplos mais “simples” (e destaco a importância de se colocar o termo entre aspas), temos também exemplos mais complexos, que geram muita diferença de desempenho nas aplicações. Por exemplo: qual a diferença entre utilizarmos Listas (sejam elas Vetores ou listas ligadas) e Hash? A principal diferença está no desempenho da busca. Observe este pequeno exemplo de benchmark em Java:

```
1      ArrayList<Produto> lista;
2      lista = new ArrayList<Produto>();
3      for (int i = 0; i < 100000; i++) {
4          Produto p = new Produto(i + 1, "Produto " + (i + 1), \
5          0, 0);
6          lista.add(p);
7      }
8      int quantosAchei=0; // numero de ocorrencias encontradas
9      // inicio da medicao do tempo
10     long inicio = System.currentTimeMillis();
11     Produto busca;
12     for (int cont = 0; cont < 10000; cont++) {
13         for (int i = 0; i < lista.size(); i++) {
14             Produto tmp = lista.get(i);
15             if (tmp.getId() == -1) { // forcando buscar um I\
```

```
16 D que nao existe na lista -> o pior caso
17             busca = tmp;
18             quantosAchei++;
19             break;
20         }
21     }
22 }
23 long fim = System.currentTimeMillis();
24 // fim da medicao do tempo
25 System.out.println("Achei = " + quantosAchei +" em "+(f\
26 im-inicio));
```

Esse simples algoritmo popula uma lista com 100.000 objetos do tipo produto e realiza 10.000 buscas de um produto inexistente nessa lista. Como é uma estrutura linear (e voltamos à análise de algoritmos), a busca obrigatoriamente tem que passar por todos os objetos até concluir que ele não existe na lista. Um algoritmo deste pode levar alguns bons segundos para executar (um teste em uma máquina comum pode levar entre 2 e 5 segundos para executar). É possível melhorar o desempenho dessa busca? Claro que sim. Poderíamos usar, ao invés de busca linear, um algoritmo de busca binária. Entretanto, para essa estratégia, nosso conjunto precisaria estar previamente ordenado.

Agora, se pensarmos em outra estrutura, como um mapa Hash, qual a vantagem? Vamos observar este código.

```
1      HashMap<Integer, Produto> mapa;
2      mapa = new HashMap<Integer, Produto>();
3      for (int i = 0; i < 1000000; i++) {
4          Produto p = new Produto(i + 1, "Produto " + (i + 1), \
5          0, 0);
6          mapa.put(p.getId(), p);
7      }
8      int quantosAchei=0;
9
10     // inicio da medicao
11     long inicio = System.currentTimeMillis();
12     for (int cont=0; cont< 10000; cont++) {
13         Produto busca = mapa.get(-1); // novamente forcando a b\
14 usca de um elemento que nao existe
15         if (busca != null) {
16             quantosAchei++;
17         }
18     }
19     long fim = System.currentTimeMillis();
20     // fim da medicao
21     System.out.println("Achei = "+quantosAchei+ " em "+(fim-i\
22 nicio));
```

Pela própria definição de Hash, há um cálculo para determinar, através de um atributo-chave do objeto, qual a posição de memória que ele irá ocupar. Uma vez determinada essa posição, o acesso é direto ao objeto (ou à inexistência dele). Portanto, o tempo computacional de acesso de um objeto em um mapa hash é $O(1)$, ou seja, constante!

Desse modo, independentemente do tamanho do conjunto, o tempo de acesso sempre será o mesmo (o que traz um desempenho excepcional, se comparado ao desempenho de busca em uma lista).

Não somente Hash, como também Árvores binárias, AVLs, ou

mesmo B-Trees, para que dev séniors tenham condições mínimas de definir estratégias de índices em tabelas de bancos de dados relacionais.

Agora, estou buscando valorizar bastante a Análise de Algoritmos, correto? Quer um exemplo de onde a Análise de Algoritmos pode não ser suficiente para uma solução?

Um algoritmo bastante comum na formação durante a graduação é o percurso e preenchimento de matrizes. Esse tipo de algoritmo, independente de o preenchimento ser via Linhas x Colunas ou Colunas x Linhas, na visão única e exclusiva da Análise de Algoritmos é irrelevante, pois ambos são de ordem assintótica $O(n^2)$. Observe estes fragmentos de código Java:

Fragmento 1: preenchimento Linha x Coluna

```
1 for (int i=0; i < TAM; i++)
2     for (int j=0; j < TAM; j++)
3         matriz[i][j] = valor;
```

Fragmento 2: preenchimento Coluna x Linha

```
1 for (int i=0; i < TAM; i++)
2     for (int j=0; j < TAM ; j++)
3         matriz[j][i] = valor;
```

O aspecto interessante nesses códigos é que, na prática, o desempenho desses dois algoritmos é muito diferente, se a dimensão da matriz for considerável (aqui, para considerável, vamos pensar em valores acima de 5.000).

Mas por que a diferença de desempenho? Por conta do número de acessos ao cache. Se aprofundarmos um pouco mais o estudo e também levarmos em conta a Arquitetura de Computadores, veremos que uma matriz é alocada em memória como

um grande array (a notação de matriz que as linguagens de programação utilizam é apenas uma abstração). Por conta disso (e levando em consideração o conceito da localidade referencial - negligenciado no estudo de Sistemas Operacionais), obviamente, se uma linha é trazida para o cache, é muito mais rápido preencher seus elementos adjacentes até que haja a necessidade de acesso a outras páginas de memória (gerando assim um *page fault*) do que ficar buscando elementos em regiões “distantes” da memória, aumentando - e muito - o volume de paginação.

Aliás, pode não parecer algo muito próximo do desenvolvimento Web atual levantarmos aspectos de Sistemas Operacionais e Arquitetura de Computadores, mas muitos desses conceitos de gerenciamento de memória e algoritmos de substituição de páginas estão, sim, presentes em ferramentas de desenvolvimento Web. Ferramentas de Cache (como REDIS, por exemplo) precisam definir uma política de atualização desses dados que serão expostos. Conhecer os algoritmos como o *MRU - Most Recently Used*, *LRU - Least Recently Used*, *MFU - Most Frequently Used*, entre outros, pode ser de fundamental importância para entender como uma ferramenta dessas trabalha, ou mesmo (na ausência de algo pronto) para implementar sua própria ferramenta de cache.

E o estudo de teoria dos Grafos? Sem os algoritmos de otimização vistos em Grafos (como o famoso *Algoritmo de Dijkstra*), não teríamos aplicações como Waze, por exemplo. Só que o ponto principal não se limita a apenas reproduzir o algoritmo, mas principalmente em encontrar um modelo que represente o que arestas e vértices significam no contexto do problema a ser resolvido. A distância entre dois vértices pode ser interpretada através da diferença efetiva em metros (ou quilômetros) ou, se baseada na velocidade dos automóveis e no tempo para se trafegar entre dois pontos, infere-se uma distância que pode ser dinâmica e, por consequência, atualizar o valor das arestas.

Não apenas Grafos, mas Autômatos e Compiladores, duas disciplinas fundamentais hoje para extração de textos através de expressões regulares (100% implementadas através de Autômatos Finitos Determinísticos - os famosos AFDs) e também Compiladores para a construção de novas linguagens (se observarmos a quantidade de linguagens que surgiram desde o ano 2000 até hoje, isso é muita coisa).

Mas você, como dev, pode se perguntar: “E quando eu farei uma nova linguagem?”. Talvez a resposta seja “dificilmente”, para não dizer “nunca”; mas, olhando um pouco além do dia a dia, desenvolvedores Front-End já devem ter utilizado frameworks como Angular ou React, correto? Pois bem, tais frameworks precisam fazer uso de análise de arquivos HTML com *tags* específicas que substituem valores para objetos. A leitura dessas tags (para posterior geração do código final da sua *single page application*) é feita por um *parser* que, a partir dos componentes criados, gera todo o arquivo HTML com as bibliotecas Javascript para que seu Aplicativo consiga funcionar. Talvez isso ajude a responder à questão que motiva este capítulo, na seção a seguir.

O ponto-chave: usar frameworks, fazer as tarefas manualmente, ou melhor, criar seu próprio?

Difícil de responder, porém com uma tendência a buscar desafios. Profissionais de desenvolvimento tornam-se sênior com mais rapidez quando entendem profundamente como as tecnologias, bibliotecas e frameworks funcionam nos bastidores. Portanto, tentaremos responder a essa pergunta (ou talvez até uma provocação) analisando-a por três diferentes visões.

Visão do conhecimento: Obviamente, você pode (e deve!) criar seu próprio framework pelo menos uma vez na vida, mesmo

que a única pessoa a usar esse framework seja você. Qual o motivo? Este mesmo: conhecimento. Silvio Meira (um dos principais influenciadores em Computação do Brasil e do mundo) já disse em uma palestra que “*o profissional de computação que nunca desenvolveu um compilador não pode se considerar um profissional completo, pois um compilador exige que você conheça toda a grade de um curso de Ciência da Computação*”. E ouso complementar: desenvolva também uma *Game Engine*, pois existe a parte de Álgebra Linear e Computação Gráfica que é pouco aplicada em um compilador. Um compilador, assim como uma *Game Engine*, exige que um dev *Ligue os Pontos* conceituais que sempre ficaram negligenciados.

Visão Corporativa: Muitas empresas podem adotar frameworks feitos pela comunidade? Sim, claro! Entretanto, existem decisões estratégicas dentro das corporações para que seus produtos não dependam de tecnologias de terceiros. Portanto, é muito comum haver times de desenvolvimento de frameworks (nos grandes *players*, principalmente) para uso interno em seus produtos. Isso proporciona controle total sobre a tecnologia que empregam. Quando esses frameworks estão maduros o suficiente, podem ficar disponíveis para a comunidade (em licenças *Open Source*) para que a comunidade faça bom uso e os popularize. Ou simplesmente ficam para uso interno e exclusivo, sem acesso pelas comunidades de desenvolvimento.

Visão da Necessidade: Este pode ser um agente motivador mais imediato, pois criar um framework tem por objetivo aumentar a produtividade no desenvolvimento de software. Quando os processos internos de desenvolvimento de equipe são bem definidos, é óbvio que os times começam a enxergar as necessidades comuns e gerar soluções de uso geral e propósito específico, sejam eles frameworks Back ou Front end, ORM ou mesmo em mais baixo nível para geração ou otimização de código. O principal objetivo de se construir uma tecnologia como essa é justamente investir tempo criando uma ferramenta que irá otimizar

mizar o tempo de desenvolvimento de uma aplicação (compare o tempo de desenvolvimento de uma camada de integração de Java com Bancos de Dados utilizando JDBC puro ou JPA, se ainda restam dúvidas).

Portanto, e para concluir, à medida que profissionais se tornam naturalmente seniores, seu conhecimento sobre as mais diversas áreas do conhecimento dentro da Ciência da Computação tende a tornar-se mais profundo e ao mesmo tempo mais amplo. Não basta conhecer um pouco sobre várias coisas e especializar-se em apenas uma delas. À medida que os problemas difíceis passam a ser mais comuns, profissionais tendem a ser expostos a situações que exigem muito conhecimento em muitas disciplinas diferentes, bem como a maneira como esses conhecimentos estão relacionados para a resolução do problema. O famoso “*ligar os pontos*”.

Tenho lido sobre DDD, para onde devo ir depois?

Você já codifica Java há muitos anos, já leu sobre Domain-Driven Design (DDD) e deseja aplicar isso a um projeto da vida real. Como isso funciona? O que realmente significa aplicar DDD no ecossistema de hoje? Realmente vale a pena o tempo investido?

O DDD nos fornece uma base teórica e padrões muito bons para construir um software robusto, que agregue valor real ao negócio. Para fazer isso, o DDD ajuda a fornecer uma estrutura para organizar o software construído e otimizar equipes independentes, que possam trabalhar juntas consumindo APIs bem definidas. No final das contas, o DDD fornece os princípios de que você precisa para traduzir para Java ou qualquer outra linguagem de sua escolha. Mas, ao fazer isso, você precisará tomar algumas decisões difíceis sobre como esses componentes/serviços/aplicações Java serão executados e interagirão.

Este capítulo aborda o lado prático de como esses conceitos podem ser mapeados em um exemplo existente, executado no Kubernetes, oferecendo dicas práticas sobre como os conceitos DDD podem ser mapeados para um stack tecnológico (pilha de tecnologia) concreto. Claro, existem milhares de opções diferentes para se escolher hoje, mas você pode tomar isso como um exemplo do tipo de coisas a que você precisará se atentar ao passar por esta jornada.

INFO: é importante destacar que este capítulo não é sobre os conceitos básicos de DDD. Portanto, se você é novo no DDD, os seguintes livros são recomendados: *Implementing DDD* e *DDD Distilled*.

Este capítulo está dividido em duas seções principais:

- [Introdução aos tópicos relacionados a Java e nuvem](#)
- [De Monolith até K8s usando DDD](#)

Java na nuvem

Existem muitos frameworks por aí com o objetivo de fornecer uma experiência mais fácil para os desenvolvedores que criam microsserviços. Exemplos disso são Spring Boot, Quarkus, Micronaut, Helidon etc. Em geral, o objetivo desses frameworks é criar um arquivo JAR autônomo, que possa ser executado, fornecendo a você um executável livre de dependências e que requer apenas a Java Virtual Machine (JVM) para rodar.

Isso vai contra o que nós (como comunidade Java) estávamos fazendo cinco anos atrás, e alguns de nós ainda fazem, que era fazer o deploy de nossas aplicações Java dentro de um servidor de aplicação ou um Servlet Container como o Tomcat.

Enquanto costumávamos ter um monolito com todos os recursos de nossas grandes aplicações integradas, agora buscamos ter um conjunto de serviços com funcionalidades bem definidas. Esses novos (micro) serviços compartilham as seguintes características:

- Tendem a localizar e versionar todos os artefatos que são necessários para ir do código-fonte até um serviço em execução em um ambiente.
- Cada serviço é construído, mantido, desenvolvido e implantado por uma equipe diferente.
- Cada serviço tem seu próprio ciclo de lançamento.
- Cada serviço expõe um conjunto bem definido de APIs.

Construir um serviço hoje com endpoints REST é uma tarefa bastante fácil se você estiver usando um desses frameworks mencionados anteriormente. Você tem um modelo de programação baseado em anotação que permite mapear métodos Java para terminais REST e mecanismos avançados de serialização/deserialização que lidarão com todo o boilerplate de análise de solicitações HTTP.

DICA: Para mais detalhes sobre a arquitetura de microserviços, consulte o capítulo [Microsserviços](#).

O verdadeiro problema surge quando você começa a ter mais do que um punhado de serviços. A execução de cada serviço em sua própria JVM irá forçá-lo a executar cada serviço em uma porta diferente e cuidar dos problemas quando essas JVMs travarem. Por isso, a indústria saltou rapidamente para containers por volta de 2015.

Containers & organização de containers

Seguindo a mesma linha de ficar livre de dependências, como os frameworks Java mencionados na seção anterior, containers nos ajudam a executar nosso software em qualquer lugar. Isso significa que vamos um passo adiante e agora não queremos nem mesmo depender da Java Virtual Machine sendo instalada em nossa máquina host, onde os serviços serão executados.

Docker (um container runtime) nos ajudou com isso. Podemos encapsular nosso JAR autônomo junto com a JVM e os arquivos de configuração para torná-lo um container que possa ser executado em qualquer lugar onde o Docker Container Runtime estiver instalado.

Cada container Docker tem seu próprio tempo de execução isolado, o que nos permite isolar as falhas nos limites do container Docker.

Quando você tem um Bounded Context (contexto delimitado) e alguns Serviços, provavelmente está tudo bem apenas executar esses microsserviços Java usando um script (ou docker-compose). No entanto, mesmo para as práticas diárias de desenvolvimento, você notará que cada um desses serviços exigirá uma porta dedicada, e será necessário manter o controle dessas portas para garantir que estejam disponíveis para cada um dos serviços.

Quando o número de serviços aumenta, isso se torna incontrolável. Por esse motivo, os Container Orchestrators (orquestradores de container) se tornaram populares nos últimos anos, e o Kubernetes está liderando o caminho. O Kubernetes é responsável por lidar com a criação desses container runtimes, como escaloná-los quando há carga e como lidar com containers que apresentem mau comportamento ou falhem.

DICA: Para obter mais detalhes sobre containers e ferramentas de orquestração, consulte o capítulo sobre [Cloud](#).

O sucesso do Kubernetes é baseado no fato de que cada grande provedor na nuvem fornece um serviço Kubernetes gerenciado, tornando-o o padrão de fato para suporte a multicloud. Em outras palavras, não importa qual provedor você escolha, você sempre pode confiar que haverá uma API Kubernetes exposta para você interagir e provisionar seus serviços.

Capitalizando os benefícios do DDD

Se você seguir o caminho do DDD, precisará primeiramente aproveitar algumas das promessas que o DDD lhe deu e certificar-se de que está colhendo os benefícios. Se não podemos fornecer continuamente novas versões de nossos serviços sem interromper a aplicação inteira, estamos apenas tornando nossa

vida mais complicada para nada. Se não estamos entregando valor de negócio concreto como resultado do DDD, todas as mudanças sugeridas neste capítulo não valem o esforço ou o tempo.

Eu recomendo o seguinte artigo: “[The Business Value of Using DDD](#)”, que dá uma visão geral de alto nível dos benefícios de adotar o DDD, não para você como desenvolvedor(a), mas para o seu negócio.

A próxima seção explora um exemplo que eu criei com base na minha experiência durante a rearquitetura de aplicações monolíticas Java para uma abordagem mais distribuída. O exemplo é fictício, qualquer semelhança com a realidade é mera coincidência :) Nós encorajamos você a abstrair os conceitos e padrões do cenário de exemplo e mapeá-los para seu próprio domínio. No final das contas, este é apenas um exemplo, embora complexo e totalmente funcional.

Evoluindo seu monolito na prática, usando DDD

Esta seção cobre um cenário de exemplo que nos ajuda a explicar alguns dos conceitos em ação. Você pode mapear esses conceitos para seu domínio de negócios e copiar a solução técnica real do exemplo para alguns dos desafios apresentados.

Como esperado, criar uma aplicação completa é um trabalho árduo e geralmente requer muito tempo. Por esse motivo, o exemplo a seguir é fornecido como um conjunto de repositórios de código aberto, e você pode contribuir para melhorá-lo.

- [Repositório De Monolith para K8s Github](#)

DICA: Como uma aplicação real, o exemplo evoluirá com o tempo, agregando mais ferramentas e melhores práticas, por isso convidamos você a participar dessa jornada em que todos podemos aprender e compartilhar informações valiosas juntos.

Começamos nossa jornada com uma aplicação Java monolítica. O cenário que iremos abordar pertence a uma empresa que se encarrega de fornecer uma plataforma para a criação de sites de conferências. Imagine que cada um de nossos clientes exija que hospedemos e escalonemos seu site de conferências. Todos nós vimos grandes aplicações Java Web e, neste cenário, a aplicação se parece com isto:



Imagen 02_01: Aplicação monolítica para gerenciamento de conferências.

O Facade (fachada) “Customer Management” se encarrega de isolar os diferentes clientes uns dos outros. Em algumas empresas, isso é definido como uma plataforma ou aplicação multi-tenant (multilocatário). Infelizmente, isso é bastante comum no espaço Java. Por razões históricas, as implementações acabaram crescendo em monolitos grandes e assustadores, que vieram com muitos problemas de escalabilidade, bem como com desafios de isolamento de tráfego e dados. Não importa o quanto sofisticada seja a aparência de nossa plataforma, ela está apenas rodando em uma única JVM. Você não tem como dimensionar cada cliente individualmente - você dimensiona tudo ou nada.

Como você pode ver na caixa vermelha, cada Site de Conferência conterá um monte de módulos, dependendo da seleção de cada cliente, mas, na realidade, em tempo de execução, todo o código estará lá para cada conferência.

Observe que, se você tiver uma plataforma como esta e ela fizer o trabalho que deve fazer, você **NÃO deve** alterá-la. A menos que você esteja tendo problemas para gerenciar ou escalar essa “plataforma”, você não deve rearquitetar a coisa toda.

Agora, essa arquitetura de monolito tem algumas desvantagens claras e, para esse cenário, podemos considerar o seguinte motivo para rearquitetá-la em uma plataforma adequada nativa da nuvem:

- Clientes não podem ser escalados independentemente.
- O tráfego de clientes é todo gerenciado pela mesma aplicação.
- Ponto Único de Falha na JVM.
- Se os dados são armazenados em um banco de dados, os dados são compartilhados entre os(as) clientes. O código da aplicação precisa lidar com o isolamento dos dados de cada cliente. O banco de dados também se torna um gargalo, pois todos os dados dos(as) clientes estão no mesmo banco de dados.
- Cada mudança na plataforma requer que toda a aplicação seja reiniciada.
- Cada desenvolvedor(a) envolvido(a) com a aplicação trabalha com a mesma base de código, o que torna um lançamento e uma fusão de recursos uma grande tarefa, com muitos riscos envolvidos. Isso geralmente pode ser feito por alguém que entenda toda a aplicação.

DICA: Se você já tem essa aplicação instalada e funcionando e tem clientes usando a plataforma, terá um bom entendimento de quais recursos são essenciais e como começar a reformulá-los.

Como Martin Fowler descreve no post [Monolith First](#), é o caminho a tomar. Por ter um monolito, você já entende a solução que precisa construir, tornando mais fácil estimar como a nova arquitetura resolverá os problemas da versão existente. Em outras palavras, se você não tem um monolito existente, não comece com uma arquitetura distribuída do zero. Crie um monolito primeiro e depois divida, se necessário.

O próximo passo em nossa jornada é decidir por onde começar. Em minha experiência, vi três padrões comuns se repetindo:

- **Iniciar novas funcionalidades como serviços separados:** isso geralmente é recomendado se você puder manter o monolito como está. Novos serviços não resolverão os problemas já existentes, mas ajudarão suas equipes de desenvolvedores a se acostumarem a trabalhar com uma mentalidade de microserviços.
- **Dividir a funcionalidade existente do monolito** (e lentamente desativar o código antigo): se você tiver problemas urgentes com o monolito, pode avaliar a ramificação de algumas das funcionalidades externas para um novo serviço. Isso pode resolver alguns dos problemas existentes, mas não trará nenhum valor de negócios imediatamente. Isso também aumenta a complexidade das operações do dia a dia, pois você pode acabar executando duas soluções para o mesmo problema por um longo período de tempo. Além do mais, isso pode ser usado para entender o quanto complexa e cara uma rearquitetura central pode ser.
- **Reestruturar o núcleo da plataforma como microserviços** (para resolver os problemas existentes): mais cedo ou

mais tarde, se você estiver tendo problemas para manter e dimensionar seu monolito, precisará repensar e redesenhar as partes principais de sua plataforma, certificando-se de focar em resolver os problemas atuais de escalabilidade e manutenção. Isso pode ser uma iniciativa cara, mas pode ser feito de forma totalmente isolada de seus ambientes de produção. Várias vezes eu vi como isso é feito como uma Prova de Conceito para demonstrar que é realmente possível e também para garantir que os membros de sua equipe entendam as implicações de um negócio (vantagens) e do ponto de vista técnico (novo stack tecnológico, novas ferramentas).

Neste capítulo, irei cobrir a última dessas opções (**Reestruturar o núcleo da plataforma como microsserviços**) para destacar a solução para nossos problemas existentes com a aplicação monolítica, mas você pode explorar as outras duas, se elas forem mais apropriadas para sua situação.

DICA: Mais informações sobre estratégias e práticas sobre como migrar um monólito existente para a arquitetura de microsserviços podem ser encontradas no capítulo [Microsserviços](#).

É aqui que os conceitos e padrões DDD se tornam realmente úteis para definir como dividir as funcionalidades do monolito e como organizar nossas equipes em torno dos novos serviços. Nas seções a seguir, exploraremos alguns desses conceitos em ação.

Suposições e simplificações

As plataformas são muito complexas. Tentar reestruturar algo grande vai lhe dar mais dores de cabeça do que soluções. Por

esse motivo, você deve tentar simplificar o escopo do problema para enfrentar diferentes desafios de forma controlada.

Para o nosso cenário, isso pode significar que, em vez de tentar a rearquitetura de toda a plataforma, primeiro tentaremos resolver a arquitetura e a forma de um simples site de conferências. Isso significa que, em vez de focar a plataforma, primeiro nos concentraremos no que o cliente espera de seu site de conferência. Ao compreender a forma de um único site de conferência, você e suas equipes têm um conjunto bem definido de desafios a serem resolvidos, o que pode agregar valor imediato aos negócios. Posteriormente, automatizaremos como cada um desses sites de conferências pode ser provisionado.

Do ponto de vista arquitetônico, isso pode significar um site de conferência monolítico ou um site de conferência construído com diferentes serviços distribuídos. Se os sites de conferência forem complexos o suficiente e quisermos reutilizar os módulos para todos eles, podemos considerar uma abordagem distribuída. Com base na minha experiência, esse tipo de plataforma aproveita os serviços compartilhados na maioria das vezes, então faz sentido arquitetá-los pensando na reutilização.

DICA: Essa estratégia leva a vários serviços desde o primeiro dia, então isso é algo com o qual você e suas equipes devem se acostumar.

Nossos sites de conferências independentes serão semelhantes a este:

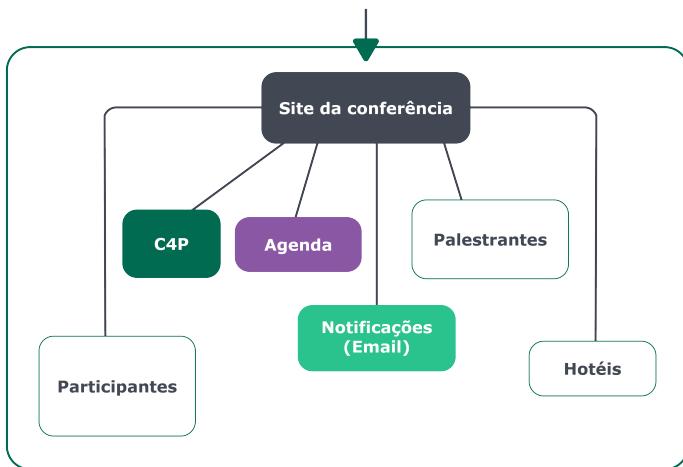


Imagen 02_02: Evolução da aplicação de conferências monolítica para uma arquitetura com serviços distribuídos.

Como você pode ver no diagrama anterior, está bastante claro que há mudanças arquitetônicas importantes. É bastante comum ter a Interface do Usuário, neste caso, a caixa “Conference Site”, separada dos serviços principais. Na maioria das vezes, esse componente voltado para o(a) usuário(a) agirá como um roteador, encaminhando solicitações do site de conferência para serviços que não são expostos diretamente a usuários(as).

Mas, antes de entrar em detalhes técnicos, como priorizamos e nos certificamos de que começamos com o pé direito? Como podemos definir o escopo desses serviços corretamente (nem muito macro, nem muito micro)? Os conceitos DDD podem nos ajudar, fornecendo algumas respostas e orientações.

Contextos limitados para começar a dividir seu monolito

Se você está planejando uma rearquitetura para o núcleo de sua plataforma, precisa ter cuidado e escolher com sabedoria quais componentes serão rearquitetados primeiramente e por quê. Na minha experiência, você deve tentar lidar com os componentes principais primeiro. Escolha componentes que você entende e sabe que agregam valor ao negócio onde você está.

No caso do nosso cenário, lidar com a Chamada de Propostas (Call for Proposals) é fundamental para iniciar uma conferência. Há um trabalho substancial para aceitar propostas, revisá-las e garantir que a conferência tenha uma agenda interessante.

Se estivermos confiantes de que a implementação da funcionalidade de Chamada de Propostas primeiro nos dará vitórias imediatas, precisamos estimar o quanto grande e complexo é o esforço.

O DDD propõe o conceito de contexto limitado como um conjunto bem definido de funcionalidades que se encaixam. Essas funcionalidades geralmente mapeiam como um especialista de domínio (especialista no assunto) fará o trabalho se não houver software disponível. Para planejar, projetar e implementar essas funcionalidades, uma equipe é montada com especialistas em domínio, que trabalharão lado a lado com engenheiros de software. Na maioria das vezes, um especialista de domínio saberá quais contextos delimitados já existem e pelo que eles são responsáveis.

INFO: De uma perspectiva DDD, é extremamente importante não tornar esses Bounded Contexts um limite técnico que é imposto aos especialistas em domínio.

Um Bounded Context irá expor um conjunto bem definido de APIs para que outras equipes de diferentes contextos delimitados possam consumir e interagir com a funcionalidade de Chamada de Propostas.

O Bounded Context Call for Proposals permitirá que uma equipe implemente todas as funcionalidades necessárias, independentemente das outras equipes. Essa equipe será responsável por todo o ciclo de vida do projeto, desde a concepção e implementação até a execução para o resto da empresa e para que seus(suas) clientes o consumam.

Assim que começa a projetar a funcionalidade Call for Proposals, você percebe que precisará consumir e interagir com outras equipes. Logo no início, os seguintes Bounded Contexts são identificados:

Chamada de Propostas

Agenda da conferência

Notificações

Imagen 02_03: Bounded contexts da aplicação de conferências.

Cada um desses Bounded Contexts deve pertencer a equipes diferentes, e precisamos ter certeza de que eles têm autonomia suficiente para progredir, criar novas versões com novos recursos e implantar componentes de software concretos para os ambientes de nossos(as) clientes.

Do lado prático, cada Bounded Context será implementado como um ou um conjunto de serviços que implementam os

recursos do contexto. Não há necessidade de ser estrito quanto ao número de serviços que irão compor um Bounded Contexts, mas geralmente há um que se encarrega de expor um conjunto de APIs para o mundo exterior.

Para o propósito deste exemplo, um único serviço implementará toda a lógica do Bounded Contexts Call for Proposals, e a equipe por trás desse serviço será responsável por projetar suas APIs, escolher as estruturas que usarão e realmente implantá-las em um ambiente ao vivo.

Aprofundando os detalhes práticos, existem algumas das melhores práticas compartilhadas por muitas empresas e ferramentas:

- Um Repositório / Um Serviço + Entrega Contínua
- APIs abertas

Um Repositório / Um Serviço + Entrega Contínua

Normalmente, é recomendável manter todos os recursos técnicos necessários para executar um serviço próximos ao código-fonte. Isso facilita a manutenção e a carga cognitiva de compreensão de como executar nossos serviços em um ambiente real.

Hoje em dia, é bastante comum ter um serviço, digamos, escrito em Spring Boot versionado em um provedor git como o GitHub, onde podemos encontrar um Dockerfile para criar uma versão em container de nosso serviço, que pode ser executado em qualquer lugar onde um Docker daemon esteja presente.

Com o aumento da popularidade do Kubernetes, também é comum encontrar o Manifesto do Kubernetes (arquivos YAML) descrevendo como nosso serviço pode ser implantado em um cluster do Kubernetes. Finalmente, tendemos a usar pipelines de integração contínua para realmente construir todos esses componentes de software, portanto, é bastante comum também encontrar a definição de pipeline perto da fonte que precisa ser construída.

Você, como desenvolvedor(a) que visa ao Kubernetes como sua plataforma de implantação, agora é responsável por vários artefatos, não apenas pelo código-fonte do serviço Java.



Imagen 02_04: Boas práticas em um processo deploy de aplicações no Kubernetes.

Para fazer o deploy do seu código no Kubernetes:

- Você precisará construir e testar seu código-fonte. Se for Java, você pode usar, por exemplo, Maven ou Gradle para fazer isso;
- Isso resultará em um arquivo JAR que você pode querer enviar para um repositório como Nexus ou Artifactory. Esse arquivo JAR já terá uma versão nele, e se você estiver usando Maven ou Gradle, esse JAR será identificado por seu GAV (Grupo/Artefato/Versão).
- A partir desse JAR, você cria uma imagem Docker definindo um Dockerfile que entende como executar sua aplicação Java com uma JVM fornecido.
- Por fim, se quiser fazer o deploy do seu container para um cluster Kubernetes, você precisará de alguns manifestos YAML.
- Opcionalmente, se você estiver criando muitos serviços, convém usar o Helm para empacotar e liberar esses arquivos YAML. Todos esses artefatos precisam ser versionados adequadamente, o que significa que, quando você constrói uma nova versão do seu arquivo JAR, um novo container precisa ser construído, e um novo gráfico do Helm precisa ser lançado.

INFO: O Helm fornece a ideia de gráficos (pacotes) que mapeiam um a um a forma como lida-

mos com nossos artefatos Maven. Se você estiver trabalhando com gráficos do Helm, esses gráficos geralmente também são enviados/liberados para um repositório de gráficos, como o Chart Museum.

Neste ponto, se você está pensando “isso é muito trabalho”, você está 100% certo. Se você está pensando “eu não quero fazer tudo isso”, você está absolutamente certo. Eu também não quero fazer isso. Se você quer que isso funcione, precisa usar ferramentas especializadas que já entregam todas essas funcionalidades de forma automatizada. Você deve tentar automatizar cada etapa, e a indústria usa pipelines de integração contínua para conseguir isso. Vamos falar sobre como entregar pipelines com Jenkins X.

INFO: Neste exemplo, estamos visando ao Kubernetes e optamos por usar Jenkins X neste conjunto de ferramentas. Jenkins X traz CI/CD (Continuous Integration/Continuous Delivery) para o Kubernetes e faz parte da Continuous Delivery Foundation.

Como você pode notar, o Jenkins X não é apenas sobre integração contínua, mas também sobre entrega contínua. Ao cobrir a entrega contínua, o pipeline não para quando esses componentes são construídos. O pipeline é responsável por construir, testar e também implantar nossos artefatos em um ambiente ativo, onde serão executados para atender nossos(as) clientes. A parte “contínua” faz referência ao fato de que você deseja ter certeza de que a implantação de uma nova versão do seu serviço é fácil e que você terá como objetivo implantar novas versões em um curto período de tempo.

INFO: Para alcançar a entrega contínua, o Jenkins X usa um conjunto de convenções para permitir que os

desenvolvedores se concentrem na construção de valor de negócios. Essas convenções não são exclusivas do Jenkins X e fazem parte das melhores práticas coletadas de diferentes setores e profissionais. Projetos como o Jenkins X são os catalisadores para milhares de membros da comunidade, que são especialistas em CI/CD, o que resulta nas melhores práticas e ferramentas que as aplicam.

Uma das convenções usadas pelo Jenkins X é chamada “Trunk Based Development”. Basicamente, significa que cada alteração aplicada (merged) à branch master irá gerar uma nova versão de nossos artefatos. Na maioria das vezes, isso não é confortável para os desenvolvedores, já que muitas dessas práticas são comumente definidas em cada empresa e tendem a variar bastante. A principal motivação para usar algo como o Trunk Based Development é garantir que as equipes não gastem tempo definindo essas práticas. Ao trabalhar com essa convenção, você pode se concentrar na escrita do código e, quando o código estiver pronto e mesclado ao master, uma nova versão é criada e implantada em algum tipo de ambiente de teste para validações adicionais.

DICA: Recomendo fortemente que, se você estiver iniciando um novo projeto, verifique as vantagens do Trunk Based Development, bem como do livro Accelerate, pois foi usado como base para a criação de ferramentas como [Jenkins X](#).

No fim das contas, Jenkins X usa ambas as convenções, “Um Repositório / Um Serviço” mais “Trunk Based Development”, para levar seu serviço do código-fonte para uma instância em execução dentro de um Cluster Kubernetes.

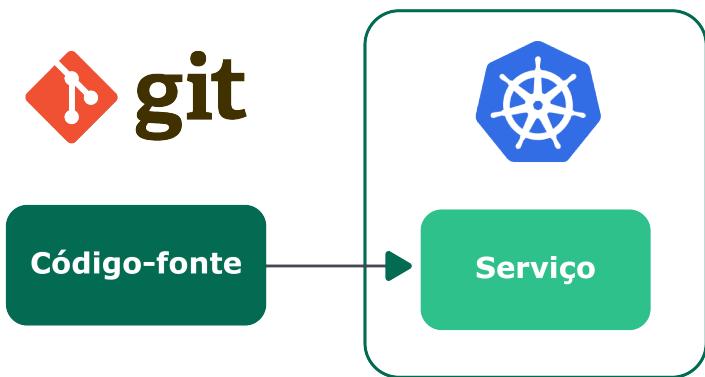


Imagen 02_05: Com a prática proposta, um repositório representa um serviço que será implantado.

Em nosso exemplo, os links a seguir demonstram todos esses conceitos em ação.

- Pipeline
- DockerFile
- Gráficos do Helm
- Versões contínuas

Você pode encontrar o mesmo setup para todos os projetos dentro da Demonstração do Site de Conferência.

APIs abertas

Se você já está implementando um Bounded Context, logo no início você precisará projetar e especificar que tipo de interface irá expor para outro Bounded Context e serviços terceiros que possam estar interessados na funcionalidade que seu contexto fornece. Uma maneira popular de implementar essas APIs são os terminais REST.

INFO: Como você provavelmente está familiarizado com endpoints REST, esta seção se concentra na [Especificação de API aberta](#)

Conforme definido no texto das especificações “*a especificação OpenAPI remove as suposições ao chamar um serviço.*” Hoje em dia, frameworks populares como Spring Boot vêm com integração pronta para uso com API aberta e ferramentas de API aberta.

Apenas adicionando uma extensão/iniciador Spring Boot, você permite que sua aplicação exponha uma interface de usuário que serve como documentação e navegador ao vivo de suas APIs.

Se você está usando Spring Boot padrão (Tomcat), será preciso adicionar ao seu arquivo pom.xml:

```
1 <dependency>
2   <groupId>org.springdoc</groupId>
3   <artifactId>springdoc-openapi-ui</artifactId>
4   <version>${springdoc-openapi-ui.version}</version>
5 </dependency>
```

Se você estiver usando o Webflux, a pilha reativa que você precisa adicionar é:

```
1 <dependency>
2   <groupId>org.springdoc</groupId>
3   <artifactId>springdoc-openapi-webflux-ui</artifactId>
4   <version>${springdoc-openapi-ui.version}</version>
5 </dependency>
```

- <https://github.com/salaboy/fmtok8s-email/blob/master/pom.xml#L40>

Em projetos da vida real, essas interfaces de usuário e documentos de especificação de API podem ser usados por outras equipes para entender com detalhes concretos como interagir com seus serviços. Quanto mais cedo você expõe uma API, mais cedo outras equipes podem começar a aproveitar seu serviço.

A captura de tela a seguir mostra a interface de usuário da API aberta, a qual é fornecida apenas incluindo a dependência anterior. Essa tela pode ser acessada apontando seu navegador para `host:port/swagger-ui.html`. Ela fornece um cliente simples para interagir com seus serviços, entender quais terminais estão expostos e quais dados esses terminais esperam e retornam.

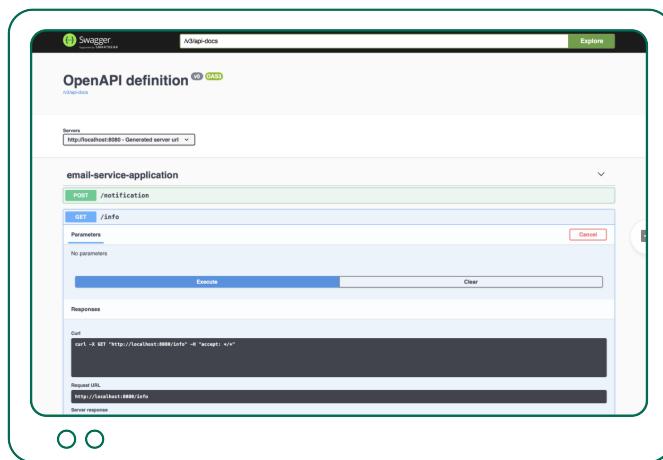


Imagen 02_06: Interface da Swagger API

Sinta-se à vontade para clonar um dos serviços deste exemplo e executá-lo com o comando `mvn spring-boot:run` para explorar as definições de APIs de cada serviço. Por padrão, cada serviço iniciará na porta 8080, portanto, você deve apontar seu navegador para <http://localhost:8080/swagger-ui.html>

Mapa de contexto para entender as interações técnicas e da equipe

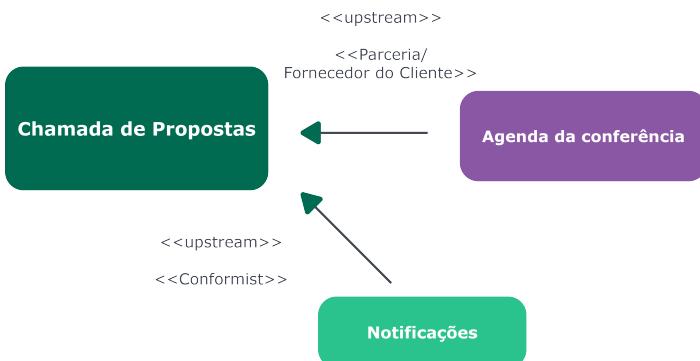
Bounded Contexts são ótimos para entender um conjunto bem definido de funcionalidades que precisam ser fornecidas juntas. Quando temos vários desses contextos, precisamos entender como eles irão interagir uns com os outros e seus relacionamentos. É aí que o conceito de Mapas de Contexto realmente ajuda. Com mapas de contexto, você pode mapear as relações entre o bounded contexts e o que eles realmente precisam para interagir. O mapeamento de contexto também fornece visibilidade sobre como as equipes responsáveis por cada contexto limitado irão interagir com outras equipes.

Do lado prático, trata-se de integrações de sistema. Como nossos serviços ou os serviços que expõem algum tipo de API conversam entre si. Como eles transformam e movem dados e como eles sabem quais serviços estão disponíveis para consumo.

Como você pode imaginar, as APIs são extremamente importantes, mas entender quem vai consumir nossas APIs, o que se espera dessas APIs e quem realmente depende de nós é igualmente crítico.

Mapas de contexto bem definidos ajudam muito a planejar e compreender como esses bounded contexts “isolados” e as equipes que trabalham neles irão interagir no dia a dia.

Para nosso exemplo, o seguinte mapa de contexto faria sentido:



Esse diagrama descreve as relações entre o bounded context simples que temos para nossa aplicação do site de conferência. Aqui podemos ver que existe uma relação **Cliente/Fornecedor** entre a Call for Proposals e o bounded context da Agenda da Conferência. A Call for Proposals é **um consumidor** da Agenda da Conferência do serviço upstream. Entre essas duas equipes, existe também uma relação de **Parceria(Partnership)**, pois elas precisam colaborar para fazer as coisas. Isso significa que a comunicação entre essas duas equipes é importante e que elas devem ser capazes de influenciar o roteiro uma da outra.

Por outro lado, a relação com o serviço de Notificação é diferente. A Call for Proposals tem uma relação ascendente com o Bounded Context de Notificação (Notification), mas vai **confortar** com seus contratos. Isso significa que, da perspectiva da equipe de Call for Proposals, eles não podem influenciar ou alterar as APIs do Bounded Context de notificação. Isso acontece muito quando temos sistemas legados ou quando esse bounded context é externo à nossa empresa.

DICA: Pulando para o lado prático, embora as integrações de sistema sejam um tópico muito amplo, esta seção se concentra em uma

recomendação muito prática: “Você deve aprender sobre testes de contrato orientados ao consumidor (Consumer-Driven Contract)”. Mais uma vez, Martin Fowler tem um artigo, publicado em 2006, sobre isso: <https://martinfowler.com/articles/consumerDrivenContracts.html>.

Embora o tópico em si não seja novo, existem ferramentas muito atualizadas para realmente implementar isso em seus projetos, como [Spring Cloud Contracts](#).

Bônus: Implementação de contratos com Spring Cloud Contracts

Com Spring Cloud Contracts, a história é assim: primeiro você define um contrato para suas APIs. Isso basicamente significa que tipo de solicitação o consumidor deve enviar e que tipo de resposta precisamos fornecer como serviço.

Um contrato é semelhante a isto: <https://github.com/salaboy/fmtok8s-c4p/blob/no-workflow/src/test/resources/contracts/shouldAcceptPostProposal.groovy>

```
1 Contract.make {
2     name "should accept POST with new Proposal"
3     request{
4         method 'POST'
5         url '/'
6         body([
7             "title": $(anyNonEmptyString()),
8             "description": $(anyNonEmptyString()),
9             "author": $(anyNonEmptyString()),
10            "email": $(anyEmail())
11        ])
12        headers {
```

```
13             contentType('application/json')
14         }
15     }
16     response {
17         status OK()
18         headers {
19             contentType('application/json')
20         }
21         body(
22             "id": $(anyUuid()),
23             "title": $(anyNonEmptyString()),
24             "description": $(anyNonEmptyString()),
25             "author": $(anyNonEmptyString()),
26             "email": $(anyEmail())
27         )
28     }
29 }
```

Este contrato define a interação para o envio de uma nova Proposta ao Serviço Call for Proposal. Como você pode ver, ele envolve uma solicitação POST e um body com algumas propriedades predefinidas, incluindo um header com um Content Type muito específico. Esse contrato também define que o retorno para o consumidor agregará às informações enviadas uma propriedade `id` com formato UUID.

Agora, esse contrato pode ser usado para gerar um teste para realmente verificar se o seu serviço está funcionando conforme o esperado, do ponto de vista do consumidor. Portanto, se você definiu qualquer contrato em seu projeto ao construir e testar esse projeto, os contratos serão executados em uma instância real de seu serviço. Isso nos permite ter certeza de que quebraremos a construção se um contrato for quebrado. Para criar e executar automaticamente esses testes, você só precisa adicionar uma dependência e um plug-in ao seu projeto maven: <https://github.com/salaboy/fmtok8s-c4p/blob/no>

workflow/pom.xml#L50

```
1 <dependency>
2   <groupId>org.springframework.cloud</groupId>
3   <artifactId>spring-cloud-starter-contract-verifier</art\>
4   ifactId>
5   <version>${spring.cloud.contract}</version>
6   <scope>test</scope>
7 </dependency>
```

E na seção `<build><plugins>`, o seguinte plugin: <https://github.com/salaboy/fmtok8s-c4p/blob/no-workflow/pom.xml#L88>

```
1 <plugin>
2   <groupId>org.springframework.cloud</groupId>
3   <artifactId>spring-cloud-contract-maven-plugin</artifac\>
4   tId>
5   <version>${spring.cloud.contract}</version>
6   <extensions>true</extensions>
7   <configuration>
8     <packageWithBaseClasses>com.salaboy.conferences.c4p</\>
9   packageWithBaseClasses>
10    <testMode>EXPLICIT</testMode>
11  </configuration>
12 </plugin>
```

Finalmente, dependendo da forma do serviço que você testará, pode ser necessária alguma confirmação: <https://github.com/salaboy/fmtok8s-c4p/blob/no-workflow/src/test/java/com/salaboy/conferences/c4p/ContractVerifierBase.java#L16>

```
1 @RunWith(SpringRunner.class)
2 @SpringBootTest( webEnvironment = SpringBootTest.WebEnvir\ 
3 onment.RANDOM_PORT,
4         properties = "server.port=0")
5 public abstract class ContractVerifierBase {
6
7     @LocalServerPort
8     int port;
9
10    ...
11 }
```

Se você precisar adicionar qualquer código de inicialização, esta é a classe que todo teste gerado automaticamente deve herdar.

Agora que o contrato foi validado com cada build, também podemos usar o contrato para gerar um Stub, que é um serviço que se comporta como o serviço real, mas com dados fictícios. Os dados fictícios também são gerados automaticamente, pois também podem ser fornecidos pela definição do contrato. Esse stub é um artefato por si só que pode ser distribuído para outros serviços, por exemplo, aqueles que consomem o serviço “real” para teste.

Isso basicamente significa que agora, toda vez que você construir seu serviço, **dois** arquivos JAR serão criados. Um é o JAR da aplicação Spring Boot real, e o outro é o Stub de Serviço. Esse stub de serviço pode ser enviado automaticamente para o repositório do seu artefato (por exemplo, Nexus ou Artifactory) e viverá no mesmo grupo e nome do artefato da sua aplicação JAR.

Finalmente, um Serviço X projetado para consumir seu serviço pode criar testes que iniciarão o stub gerado antes, localmente, para evitar a necessidade de uma instância real ou uma configuração de ambiente inteira. Você pode facilmente iniciar o stub antes de seus testes usando as seguintes anotações:

```
1 @RunWith(SpringRunner.class)
2 @SpringBootTest(webEnvironment = SpringBootTest.WebEnviro\
3 nment.MOCK)
4 @AutoConfigureMockMvc
5 @AutoConfigureJsonTesters
6 @AutoConfigureStubRunner(stubsMode = StubRunnerProperties\
7 .StubsMode.REMOTE, repositoryRoot = "<your nexus repository\
8 ry>", ids = "com.salaboy.conferences:fmtok8s-c4p")
9 public class C4PApisTests {
10     @StubRunnerPort("fmtok8s-c4p")
11     int producerPort;
12
13     ...
14 }
```

<https://github.com/salaboy/fmtok8s-api-gateway/blob/master/src/test/java/com/salaboy/conferences/site/C4PApisTests.java#L29>

Isso baixa automaticamente a versão mais recente do stub e o executa antes que seu teste comece, usando uma porta aleatória que você pode obter via injection, usando `@StubRunnerPort`.

É importante observar que tanto o Serviço quanto os contratos são versionados juntos, como parte da mesma base de código. Isso implica que o Stub gerado e o próprio Serviço terão a mesma versão. Um serviço de consumidor, para executar seus testes, pode depender do Stub, pois nunca deve depender do próprio serviço. Assim que o consumidor tiver testado por meio do Stub de Serviço do produtor, você pode reconhecer rapidamente quando um contrato é quebrado ou quando uma nova versão do contrato não é mais compatível com os consumidores, pois os testes usando os Stubs serão interrompidos quando versões novas e incompatíveis forem lançadas. Neste momento, os consumidores se deparam com uma decisão simples: ficar dependendo dos contratos antigos com uma versão fixa, ou

atualizar para a versão mais recente do contrato. Isso pode exigir que você execute várias versões do seu serviço ao mesmo tempo. Felizmente para nós, o Kubernetes foi criado para oferecer suporte a esses cenários. Você pode ler sobre versões canário (Canary Releases) se estiver interessado em aspectos de multi-version deployments.

DICA: O capítulo [Cloud](#) cobre Canary Releases, bem como outras estratégias de implantação.

Ambos, bounded contexts e mapas de contexto, são ótimas ferramentas conceituais para entender como estruturar suas equipes e seu software, mas, mais importante, esses conceitos ajudam você a se concentrar no valor do negócio.

Foco no valor do negócio

Ao trabalhar com aplicativos reais, você precisa se concentrar no que vai agregar mais valor e resolver mais problemas para o seu negócio. Sendo muito prático, tendo a analisar casos de uso e como o bounded context, em conjunto com os mapas de contexto, fornece um fluxo de negócios de ponta a ponta.

Para este exemplo específico, o cenário de conferência, estamos analisando o fluxo básico da Call for Proposals, que se parece com isto:

1. O palestrante em potencial envia uma proposta pelo site da conferência
2. O Conselho/Comitê analisa a proposta
3. Se a proposta for aceita
 1. Um novo item da agenda é criado e publicado na página da agenda
4. Uma notificação é enviada por e-mail para propostas aceitas e rejeitadas

Você precisa prestar atenção às suas interações humanas, pois essas interações tendem a exigir comportamentos assíncronos, como lembretes, notificações, alertas, bem como interfaces de usuário, que devem ser cuidadosamente projetadas.

DICA: Como engenheiros, tendemos a simplificar e subestimar a quantidade de trabalho e iterações que podem exigir a criação de uma boa experiência do(da) usuário(a).

A interface de usuário que cobre esse cenário simples é assim:

- A página principal dentro do site da conferência exibe a agenda dividida por dias. Os itens da pauta são os que já estão confirmados e foram aprovados pelo comitê.

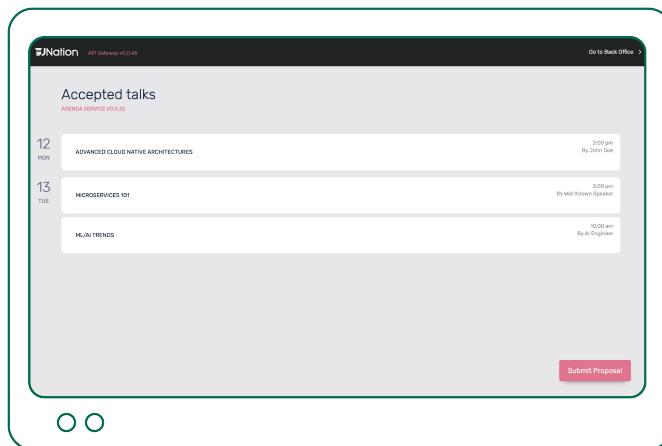


Imagen 02_08: Página Principal do site da aplicação de conferências com lista de itens confirmados e aprovados.

- A página principal também permite que palestrantes em potencial enviem propostas, preenchendo um formulário:

New Proposal
C4P SERVICE V0.0.42

TITLE
Cloud Events Orchestration

AUTHOR
Salaboy

EMAIL
salaboy@mail.com

ABSTRACT

This presentation covers Cloud Events and how they can be orchestrated using a Workflow Engine.|

SEND

Imagen 02_09: Formulário de Call for Papers

- Assim que a proposta for enviada, o palestrante em potencial precisará aguardar a aprovação ou rejeição do comitê. Os membros do comitê têm uma página de back-office, onde podem aprovar ou rejeitar cada proposta enviada:

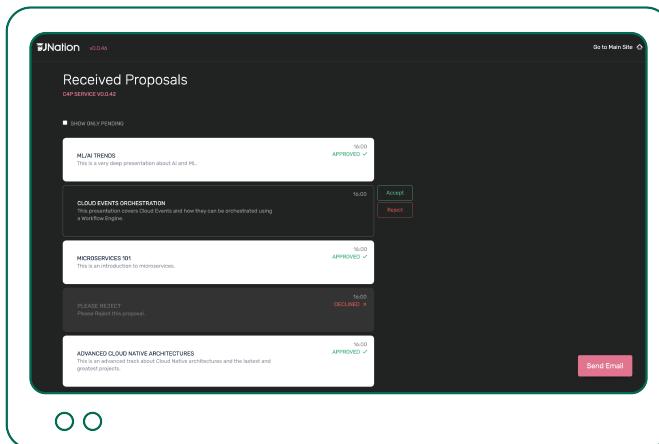


Imagen 02_10: Página de aprovação acessada pelo comitê do evento

- A página de back-office também oferece aos membros do comitê a opção de enviar notificações por e-mail aos palestrantes em potencial.

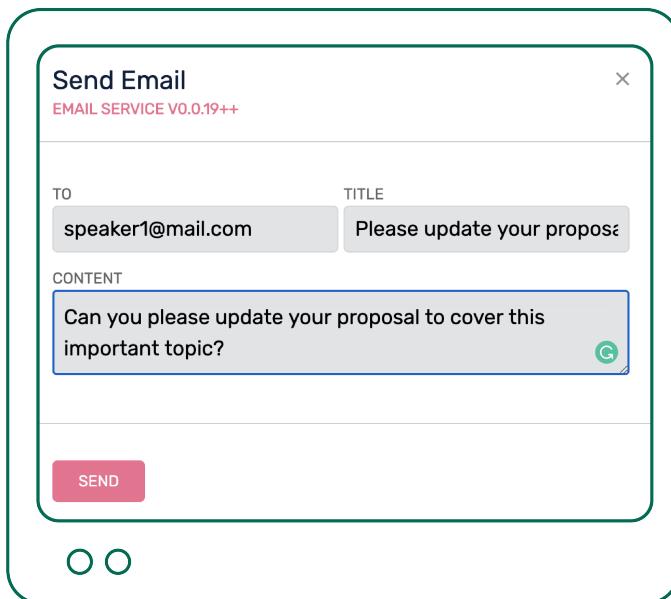


Imagen 02_10: Página de utilizada pela organização para contactar potenciais palestrantes.

Mais uma vez, você pode notar a simplificação proposital desse cenário, para estabelecer um conjunto básico de funcionalidades, iterar rapidamente e fazê-lo funcionar e então expandir os requisitos.

Arquitetura e Serviços

De uma perspectiva arquitetônica, parece mais assim:

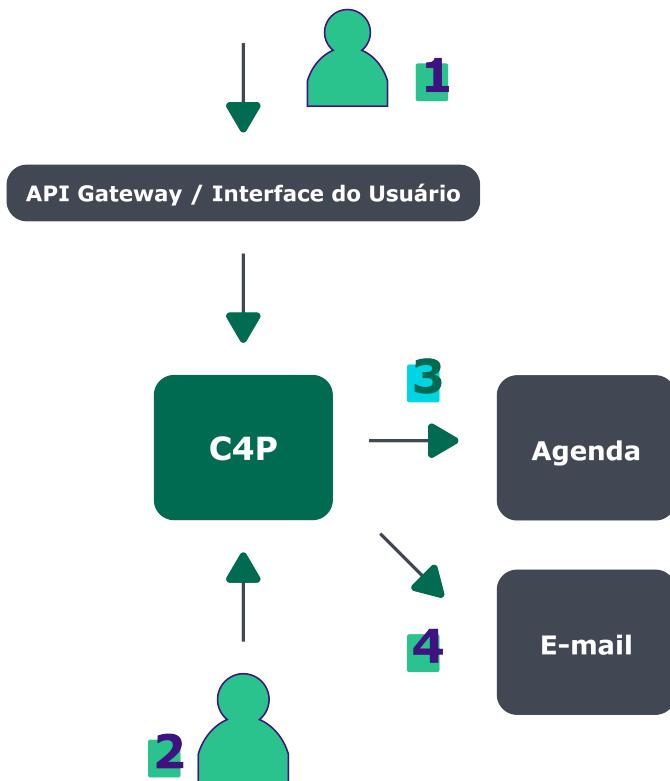


Imagen 02_12: Arquitetura e usuários da aplicação de conferências.

A Interface do Usuário com alguma capacidade de roteamento é necessária para encaminhar solicitações ao Serviço Call for Proposals (C4P) para o Serviço de Agenda ou Emails. Neste exemplo, todas as comunicações acontecem por meio de invocações HTTP/Rest.

Gateway de API / Interface do usuário

Na maioria das vezes, um Gateway de API também é usado para ocultar acesso direto a todos os outros serviços. É bastante comum ver esse serviço delegando autorização e autenticação a um provedor OAuth ou SAML, servindo como uma barreira de segurança para o mundo externo. O exemplo usa o [Spring Cloud Gateway](#), que fornece o mecanismo de roteamento para encaminhar solicitações de entrada para o restante dos serviços. O Spring Cloud Gateway nos permite transformar qualquer aplicação Spring Boot em um roteador de solicitação com recursos avançados.

INFO: É importante observar que o Spring Cloud Gateway oferece a flexibilidade de realmente adicionar programaticamente qualquer transformação que você deseja/precisa nas solicitações (requests) de entrada. Esse poder e liberdade vêm com a desvantagem de que depende de você manter, testar e corrigir o bug. Em grandes projetos, você pode querer avaliar um gateway de API de terceiros (como Kong, 3Scale, Apigee etc.) com base nos requisitos do seu projeto.

O módulo API Gateway / interface do usuário pode ser encontrado neste repositório: <https://github.com/salaboy/fmtok8s-api-gateway>

O que adiciona às suas dependências maven:

```

1 <dependency>
2   <groupId>org.springframework.cloud</groupId>
3   <artifactId>spring-cloud-starter-gateway</artifactId>
4 </dependency>
```

E configure as rotas padrão para nossos serviços dentro do arquivo application.yaml: <https://github.com/salaboy/fmtok8s-api-gateway/blob/master/src/main/resources/application.yaml#L4>

```

1 spring:
2   cloud:
3     gateway:
4       routes:
5         - id: c4p
6           uri: ${C4P_SERVICE:http://fmtok8s-c4p}
7           predicates:
8             - Path=/c4p/**
9             filters:
10               - RewritePath=/c4p/(?<id>.*), /$\\{id}
11         - id: email
12           uri: ${EMAIL_SERVICE:http://fmtok8s-email}
13           predicates:
14             - Path=/email/**
15             filters:
16               - RewritePath=/email/(?<id>.*), /$\\{id}
```

Essas rotas definem um caminho para o gateway como /c4p/**, que irá encaminhar automaticamente a solicitação para o serviço <http://fmtok8s-c4p>.

INFO: A interface de usuário do site pode ser encontrada aqui: <https://github.com/salaboy/fmtok8s-api-gateway/tree/master/src/main/resources/templates> O Controller que

busca os dados dos serviços de back-end, aqui: <https://github.com/salaboy/fmtok8s-api-gateway/blob/master/src/main/java/com/salaboy/conferences/site/DemoApplication.java>

Como estamos executando no Kubernetes, podemos usar o nome do serviço Kubernetes em vez de apontar para um pod específico. Esse mecanismo de roteamento nos permite expor apenas os Endpoints do Gateway de API para o mundo externo, deixando todos os outros serviços atrás de uma rede segura.

Eventos de domínio e o serviço Call for Proposals

Como o fluxo em análise é fundamental para o Bounded Context Call For Proposals, não é surpresa que a lógica central pertença ao Serviço Call For Proposals, mais concretamente, às duas funções a seguir: Envio de Proposta e Decisão Tomada pelo Conselho (Proposal Submission and Decision Made By the Board).

INFO: O Serviço Call for Proposals pode ser encontrado aqui: <https://github.com/salaboy/fmtok8s-c4p/>

O endpoint de Envio de Proposta aceita uma proposta da interface do usuário e a armazena em um banco de dados ou storage. Este é um passo importante, precisamos garantir que não perderemos propostas. Observe que podemos estar interessados em emitir um evento de domínio DDD neste momento, pois outros sistemas/aplicações podem estar interessados em reagir sempre que uma proposta for recebida.

INFO: Verifique uma implementação real aqui: <https://github.com/salaboy/fmtok8s-c4p/blob/no->

[workflow/src/main/java/com/salaboy/conferences/c4p/C4PController.java#L37](https://github.com/salaboy/fmtok8s-c4p/blob/no-workflow/src/main/java/com/salaboy/conferences/c4p/C4PController.java#L37)

Mais importante, o endpoint Decision Made by the Board registra uma decisão feita pela diretoria, mas também define as etapas a seguir com base nessa decisão. Na vida real, essa decisão afetará o curso da ação. Na maioria das vezes, esses pontos de decisão e as ações derivadas deles são essenciais para administrar um negócio eficiente e com boa relação custo-benefício. <https://github.com/salaboy/fmtok8s-c4p/blob/no-workflow/src/main/java/com/salaboy/conferences/c4p/C4PController.java#L60>

```

1  @PostMapping(value = "/{id}/decision")
2      public void decide(@PathVariable("id") String id, @Re\
3 questBody ProposalDecision decision) {
4          emitEvent("> Proposal Approved Event ( " + ((deci\
5 sion.isApproved() ? "Approved" : "Rejected") + ")");
6          Optional<Proposal> proposalOptional = proposalSto\
7 rageService.getProposalById(id);
8          if (proposalOptional.isPresent()) {
9              Proposal proposal = proposalOptional.get();
10
11             // Apply Decision to Proposal
12             proposal.setApproved(decision.isApproved());
13             proposal.setStatus(ProposalStatus.DECIDED);
14             proposalStorageService.add(proposal);
15
16             // Only if it is Approved create a new Agenda It\
17 em into the Agenda Service
18             if (decision.isApproved()) {
19                 agendaService.createAgendaItem(proposal);
20             }
21
22             // Notify Potential Speaker By Email

```

```
23         emailService.notifySpeakerByEmail(decision, p\  
24  roposal);  
25     } else {  
26         emitEvent(" Proposal Not Found Event (" + id \  
27  + ")");  
28     }  
29  
30 }
```

Como você provavelmente pode imaginar, esse método contém a lógica de todo o fluxo. Começa por verificar se o Id da proposta pode ser encontrado; se estiver presente, aplicará a decisão ao objeto da proposta (Approved ou Rejected); e, somente se for aprovado, chame o Serviço Agenda para criar um novo item da agenda. Não importa se foi aceito ou não, um e-mail deve ser enviado ao potencial palestrante para notificá-lo sobre a decisão. Ambos `agendaService` e `emailService` estão simplesmente encapsulando chamadas REST simples.

Este exemplo de método simples, em cenários da vida real, nunca é simples. Devido à complexidade inerente aos desafios da vida real, você deve esperar que métodos simples como o discutido acima se tornem monstros reais. A próxima seção aborda algumas das principais armadilhas e considerações que você deve ter em mente ao escrever a lógica de negócios que é a chave para seu domínio. A próxima seção também tenta compartilhar recursos, abordagens e projetos valiosos que podem ajudar você a tomar melhores decisões no início para evitar erros comuns.

Armadilhas comuns

As aplicações da vida real são complexas, e essa complexidade tende a vir da complexidade inerente aos problemas que estamos tentando resolver. Podemos tentar reduzir essa complexi-

dade usando uma abordagem de melhoria contínua e certificando-nos de que não estamos reinventando rodas desnecessárias, que não estão fornecendo qualquer diferenciação nos negócios.

Vamos começar com algo que você pode ter enfrentado no passado: comunicações REST para REST podem ser desafiadoras.

Comunicações REST para REST podem ser desafiadoras

Se você olhar o exemplo fornecido na seção anterior, os serviços de Agenda e de Email são chamados a partir do Serviço Call for Proposals usando uma [chamada REST](#). Como discutimos antes, essas interações representam uma parte fundamental de nosso fluxo de negócios, então você precisa ter certeza de que essas interações ocorram conforme planejado 100% das vezes. É vital que nosso aplicativo não acabe em um estado inconsistente - por exemplo, uma proposta é aprovada e publicada na agenda, mas nunca enviamos uma notificação ao palestrante. Como está codificado neste exemplo, se os Serviços de Agenda ou de Email ficarem inativos, a solicitação HTTP silenciosamente falhará.

Este requisito básico, ao trabalhar com sistemas distribuídos onde não existe estado compartilhado entre os serviços, torna-se um desafio com várias soluções possíveis.

Uma solução comum é escrever dentro de nosso código de serviço para tentar novamente em caso de falha, o que torna cada chamada muito mais complicada. No passado, havia várias bibliotecas que forneciam auxiliares para tais situações. É importante notar que o Kubernetes, por padrão, não lida com esse tipo de falha de forma alguma.

Outra solução pode ser usar um mecanismo de mensagem ou pub/sub para comunicar nossos serviços que, prontos para uso, oferecem mais garantias sobre a entrega das mensagens enviadas. Mudar para o uso de mensagens como RabbitMQ ou Kafka

apresenta outro conjunto de desafios, na maioria das vezes relacionado a lidar com outra peça complexa de infraestrutura que precisa ser mantida ao longo do tempo. No entanto, o sistema de mensagens provou ser robusto e a única opção para determinados cenários em que essas garantias são necessárias e o volume de interações é alto.

Finalmente, uma abordagem mais recente são as Service Meshes, em que delegamos a responsabilidade de tentar novamente, por exemplo, à infraestrutura. O Service Mesh usa proxies para inspecionar cargas HTTP e códigos de erro, de forma que novas tentativas automáticas possam ser feitas em caso de falha.

DICA: Você deve dar uma olhada em Istio, Gloo e LinkerD se quiser entender mais sobre como funcionam Service Meshes e quais são suas vantagens. Mais detalhes sobre as Service Meshes são compartilhados no capítulo [Cloud](#).

Fluxo enterrado no código

É bastante comum encontrar lógicas de negócios complexas escondidas dentro de nossos serviços, de certa forma obscurecidas por todos os padrões necessários para lidar com erros técnicos, buscar dados de diferentes fontes e transformar dados em diferentes formatos. Em projetos da vida real, fica muito difícil para os especialistas em Domínio entenderem de fato o código que implementa seus fluxos de negócios.

O exemplo discutido neste capítulo se tornaria difícil de ler se adicionássemos o código para lidar com outros aspectos, como:

- Caminhos infelizes e casos excepcionais: como o palestrante que apresentou a proposta desapareceu e não está respondendo a e-mails.

- Eventos, lembretes e restrições com base no tempo (Time-Based): por exemplo, programe um lembrete para os membros do conselho analisarem uma proposta antes de três dias após o envio. Cancele o lembrete se a decisão foi tomada antes do prazo.
- Novos requisitos são adicionados, o que leva as pessoas desenvolvedoras a mudar a sequência do fluxo, como, por exemplo, um e-mail que precisa ser enviado com um link de confirmação para o palestrante antes de publicarmos e aprovarmos a palestra para a agenda. Quanto mais requisitos, mais código precisamos adicionar, e mais ele vira um código espaguete.
- Relatórios e análises: por exemplo, seu gerente quer saber quantas propostas recebemos por dia e quanto tempo em média leva para aprovar ou rejeitar propostas. Você pode ficar tentado a adicionar novos endpoints para lidar com esses relatórios dentro do mesmo serviço.

Não existem soluções milagrosas para enfrentar esses desafios, mas eu gostaria de mencionar algumas coisas que podem, em seus cenários, ajudar a reduzir a complexidade e fornecer visibilidade sobre como seus serviços estão funcionando.

Os eventos de domínio (Domain Events) são introduzidos no DDD para externalizar o estado da aplicação que pode ser relevante para o consumo de outros serviços. Do lado prático, esses eventos podem ser criados usando [Cloud Events](#), que fornece um formato independente de transporte para a troca de eventos.

As ferramentas de Orquestração de Serviço podem ser usadas em conjunto com Cloud Events para externalizar a lógica de negócios enterrada em nossos serviços.

DICA: Verifique projetos como Zeebe, jBPM ou Kogito para entender mais sobre como essas ferramentas

podem ajudar você. Além disso, você pode achar este post sobre Cloud Events muito útil.

Finalmente, conforme discutido no livro *Implementing DDD*, patterns (padrões) como CQRS (Command/Query Responsibility Segregation [Segregação de Responsabilidade de Consulta e Comando]) podem ajudá-lo muito ao lidar com relatórios e análises. Você quer evitar a execução de relatórios caros ou rotinas intensas de processamento de dados no banco de dados de serviço. Ao aplicar o CQRS, você externaliza os dados que está interessado em relatar em um armazenamento separado que possui um formato otimizado para indexar, pesquisar e resumir dados. Uma abordagem popular é enviar dados para ElasticSearch para a indexação completa do texto. Então, na sua aplicação, se você deseja pesquisar entre milhares de propostas, você não consulta o Serviço Call for Proposals. Em vez disso, você usa os índices ElasticSearch, descarregando o serviço Call for Proposals, para que ele possa continuar aceitando propostas para suas conferências.

Adaptadores (Adapters) para sistemas legados

Uma breve nota sobre Sistemas Legados: tente abstraí-los para que você tenha controle sobre suas APIs. Para o exemplo visto neste capítulo, um serviço chamado Email foi introduzido para expor via endpoints HTTP a funcionalidade de um servidor de email. Isso foi feito propositalmente para destacar as vantagens de fornecer um adaptador para um servidor que não podemos alterar (um servidor SMTP). Nesse adaptador, podemos incluir funções auxiliares, modelos e funcionalidades específicas de domínio que são exigidas por nossos casos de uso.

O serviço Email fornecido não inclui uma conexão SMTP, mas expõe um conjunto de APIs que são fáceis de consumir e não exigem que outros serviços incluam clientes SMTP.

INFO: O código-fonte desse serviço pode ser encontrado aqui: <https://github.com/salaboy/fmtok8s-email>

Considere a criação de adapters para seus serviços legados. Lembre-se de que, dentro do Kubernetes, mesmo que os adapters sejam criados em containers separados, esses containers podem ser executados dentro do mesmo host, evitando um salto (hop) extra de rede.

Resumindo

Este capítulo abordou uma ampla gama de ferramentas e princípios referentes a uma aplicação de exemplo. Algumas dessas ferramentas farão sentido para o seu cenário, outras não. O que é importante tirar daqui pode ser resumido nos seguintes pontos:

- Otimize as decisões sobre a construção ou integração de software de terceiros para resolver questões específicas ou cruzadas (cross-cutting concerns). Para ganhar agilidade, é fundamental ter um processo claro para avaliar as ferramentas em comparação com a construção de software internamente para desafios que não são essenciais para o seu negócio.
- Mantenha suas equipes atualizadas com treinamentos. A transferência de conhecimento é um grande problema quando o stack tecnológico é amplo e complexo. Aprenda a identificar quais são os principais tópicos com os quais suas equipes têm dificuldade e encontre treinamento que possa ajudar a difundir o conhecimento entre elas.
- Use convenções em vez de definições internas, aproveite as comunidades de código aberto, que são ótimos lugares

para encontrar as melhores práticas aplicadas, inovações e tendências. Não tenha medo de participar, envolver-se e compartilhar seus aprendizados.

- Considere o uso de provedores SaaS (Software as a Service) em vez de hospedagem interna, quando possível. Se você já está rodando em um provedor na nuvem, precisa seriamente considerar o conjunto de serviços que eles oferecem. Provedores na nuvem e ofertas de SaaS economizam um tempo valioso ao configurar e manter peças-chave de sua infraestrutura. Mesmo um(a) desenvolvedor(a) podendo executar Kafka, ElasticSearch ou qualquer outra ferramenta de terceiros usando containers, não significa que ele(ela) esteja disposto a manter, atualizar e fazer backup desses serviços para toda a empresa.

Clean code

São claros e reconhecidos os grandes benefícios obtidos por meio da utilização de boas práticas no código. Algo que logo se nota ao se trabalhar com um código limpo e fluido é a melhor legibilidade de código e a maior facilidade de manutenção. Mas, ao construir aplicações e discutir práticas de arquitetura, há um outro ponto que não podemos deixar de lado: a integridade dos dados que serão manipulados. Será que as boas práticas de código refletem de forma positiva na integridade desses dados? Um dos tópicos primordiais tópicos cobertos pelo livro [Clean Code](#) é que, diferentemente da programação estruturada, a orientação a objetos expõe comportamento escondendo os dados. Com isso em mente, neste capítulo discorreremos sobre os benefícios da utilização de boas práticas de código e as vantagens obtidas ao se implementarem modelos ricos.

Modelos Ricos

Levando em consideração uma aplicação que adota práticas de clean code, os ganhos de performance são consideráveis, uma vez que, por não trafegar e validar dados em bancos de dados, há economia tanto de processamento de rede quanto de hardware. Durante o desenvolvimento, há vários desafios em se “blindar” o código, como os inúmeros conceitos técnicos que são aplicados e a junção de negócio com o interesse na criação de uma linguagem ubíqua. Que tal aprofundar um pouco mais nesse conceito através da criação de um exemplo?

Vamos criar uma aplicação de gestão de jogadores de futebol.

Teremos o conceito de time, e dentro de um time teremos as seguintes informações:

- O nome do jogador, representado pelo atributo `name` da classe `Player`;
- A posição (`position` da classe `Player`) do jogador (goleiro, ataque, defesa e meio de campo);
- O ano em que o jogador entrou no time, representado pelo atributo `start` da classe `Player`;
- O ano em que o jogador saiu do time, representado pelo atributo `end` da classe `Player`;
- O número de gols que o jogador realizou no time, representado pelo atributo `goals` da classe `Player`;
- O salário do jogador, representado pelo atributo `salary` da classe `Player`;
- O email para contato, no atributo `email`;
- A relação com o time, representada pela classe `Team`;
- Lembrando a regra, *um time não deve ter mais que vinte membros.*

Com base nas informações citadas, a primeira versão do modelo é mostrada a seguir:

```
1 import java.math.BigDecimal;
2
3 public class Player {
4
5     String name;
6
7     Integer start;
8
9     Integer end;
10
11    String email;
```

```
12     String position;
13
14     Integer goals;
15
16     BigDecimal salary;
17 }
18
19
20 public class Team {
21
22     String name;
23
24     List<Player> players;
25 }
```

À primeira vista, podemos notar possíveis melhorias no código. Note que o atributo `position` é do tipo `String`, o que não faz sentido, uma vez que as posições de jogadores serão sempre as mesmas: goleiro (goalkeeper), defesa (defender), meio de campo (midfielder) e atacante (forward). Podemos melhorar esse cenário aplicando o conceito de [Value Objects](#) através da utilização de um `Enum`:

```
1 public enum Position {
2     GOALKEEPER, DEFENDER, MIDFIELDER, FORWARD;
3 }
```

Uma vez que definimos o modelo inicial, nosso próximo passo é analisar segurança e encapsulamento dos objetos. Esta é umas das bases para um bom código orientado a objetos: a possibilidade de “esconder” os dados e expor apenas os comportamentos. A criação dos métodos assessores (`setter`) deve ser considerado como último recurso para acesso ao objeto. Um outro ponto é: deve-se avaliar a necessidade de esses métodos

serem públicos, ou seja, considere criá-los como `default` ou `protected`, caso seja possível.

Sob essa perspectiva, vamos analisar algumas regras de negócio da nossa aplicação de exemplo:

- Os jogadores não mudam de e-mail, nome e de posição, e só é possível marcar um gol por vez. Repare que a criação de métodos setters não é importante para esses atributos.
- O ano de saída pode estar vazio, porém, quando preenchido, deverá ser posterior à data de entrada;
- Apenas o time (representado pela classe `Team`) é responsável por gerenciar os jogadores (`Player`), ou seja, será necessário criar métodos para adicionar jogadores ao time.

Vamos trabalhar na classe `Team`. Teremos de criar um método para adicionar vários players (`Player`) no `Team`. Também precisaremos de um método (`get`) que retorne os players. É importante validar a entrada de players, uma vez que não faz sentido adicionar um player nulo. Devemos também garantir que apenas a classe `Team` adicione/remova players. Para isso, devemos assegurar que essa classe retorne uma lista apenas de leitura, do contrário teremos problemas com encapsulamento. Uma maneira de resolver isso seria retornar uma lista como no exemplo a seguir:

```
1 import java.util.ArrayList;
2 import java.util.Collections;
3 import java.util.List;
4 import java.util.Objects;
5
6 public class Team {
7
8     private static final int SIZE = 20;
9
10    private String name;
11
12    private List<Player> players = new ArrayList<>();
13
14    @Deprecated
15    Team() {
16    }
17
18    private Team(String name) {
19        this.name = name;
20    }
21
22    public String getName() {
23        return name;
24    }
25
26    public void add(Player player) {
27        Objects.requireNonNull(player, "player is require\\
28 d");
29
30        if (players.size() == SIZE) {
31            throw new IllegalArgumentException("The team \
32 is full");
33        }
34        this.players.add(player);
35    }
```

```
36
37     public List<Player> getPlayers() {
38         return Collections.unmodifiableList(players);
39     }
40
41     public static Team of(String name) {
42         return new Team(Objects.requireNonNull(name, "name is required"));
43     }
44 }
45 }
```

INFO: Muitos frameworks precisam que o construtor padrão exista por questão de realizar a criação de uma instância a partir da API de Reflection. Como o objetivo é desencorajar o uso do construtor padrão ao invés do uso do método de construção, o construtor será anotado com [Deprecated](#). A anotação Deprecated indica que esse método não deve ser utilizado.

Com relação à classe Player, todos os atributos terão getters padrão, com exceção do atributo end, que terá um tratamento especial: o getEnd retornará um Optional, uma vez que o end pode ser nulo. Outro ponto é o método setEnd, que só será íntegro caso o último ano seja igual ou maior que o ano de início do player, ou seja, se ele começou a jogar em 2004, não faz sentido ele ter terminado de jogar em 2002. Desse modo, o setter terá de fazer a validação no momento do acesso.

```
1 import java.math.BigDecimal;
2 import java.util.Objects;
3 import java.util.Optional;
4
5 public class Player {
6
7     private String name;
8
9     private Integer start;
10
11    private Integer end;
12
13    private String email;
14
15    private Position position;
16
17    private BigDecimal salary;
18
19    private int goal = 0;
20
21
22    public String getName() {
23        return name;
24    }
25
26    public Integer getStart() {
27        return start;
28    }
29
30    public String getEmail() {
31        return email;
32    }
33
34    public Position getPosition() {
35        return position;
```

```
36     }
37
38     public BigDecimal getSalary() {
39         return salary;
40     }
41
42     public Optional<Integer> getEnd() {
43         return Optional.ofNullable(end);
44     }
45
46     public void setEnd(Integer end) {
47         if (end != null && end <= start) {
48             throw new IllegalArgumentException("the last \
49 year of a player must be equal or higher than the start."\
50 );
51         }
52         this.end = end;
53     }
54 }
55
56     public int getGoal() {
57         return goal;
58     }
59
60     public void goal() {
61         goal++;
62     }
```

Uma vez definidos os métodos de acesso, o próximo passo está na criação das instâncias de Team e Player. Como boa parte das informações é obrigatória para se criar uma instância válida, o primeiro movimento natural seria a criação de um método construtor. Isso é válido com objetos simples, como o Team, porém a class Player tem mais complexidades, como:

- A quantidade de parâmetros: pode gerar um construtor

[polyadic](#) (construtor com mais de três argumentos).

- A complexidade das validações: não faz sentido um player começar a jogar antes de 1863, uma vez que o esporte nasceu nesse ano.

Para resolver esses problemas, executaremos dois passos: utilização de tipos customizados e aplicação do padrão Builder.

Tipos customizados

A primeira estratégia é a criação de um tipo. Essa estratégia faz sentido quando um objeto tem grande complexidade, como por exemplo objetos que lidam com dinheiro e data. Trazer essa complexidade para a entidade pode quebrar o princípio da responsabilidade única. Existe um artigo muito bom escrito por Martin Fowler, [When to Make a Type](#), que explica as vantagens de tais recursos. Também não queremos reinventar a roda, portanto, para representar ano e dinheiro utilizaremos as APIs de Date/Time que nasceram do Java 8 e a Money-API. O único tipo que precisaremos criar é o tipo e-mail, como mostra o código a seguir:

```
1 import java.util.Objects;
2 import java.util.function.Supplier;
3 import java.util.regex.Pattern;
4
5 public final class Email implements Supplier<String> {
6
7     private static final String EMAIL_PATTERN =
8         "[_A-Za-z0-9-\\\\+](\\.[_A-Za-z0-9-]+)*@"
9         + "[A-Za-z0-9-]+(\\. [A-Za-z0-9-]+)*(\\\\
10 . [A-Za-z]{2,})$";
11
12     private static final Pattern PATTERN = Pattern.compile \\
```

```
13 e(EMAIL_PATTERN);
14
15     private final String value;
16
17     @Override
18     public String get() {
19         return value;
20     }
21
22     private Email(String value) {
23         this.value = value;
24     }
25
26     @Override
27     public boolean equals(Object o) {
28         if (this == o) {
29             return true;
30         }
31         if (o == null || getClass() != o.getClass()) {
32             return false;
33         }
34         Email email = (Email) o;
35         return Objects.equals(value, email.value);
36     }
37
38     @Override
39     public int hashCode() {
40         return Objects.hashCode(value);
41     }
42
43     @Override
44     public String toString() {
45         return value;
46     }
47
```

```
48     public static Email of(String value) {
49         Objects.requireNonNull(value, "o valor é obrigató\
50         rio");
51         if (!PATTERN.matcher(value).matches()) {
52             throw new IllegalArgumentException("Email não\
53             válido");
54         }
55
56         return new Email(value);
57     }
58 }
```

Agora iremos utilizar esses tipos na classe Player. Ela ficará com o seguinte formato:

```
1 import javax.money.MonetaryAmount;
2 import java.time.Year;
3 import java.util.Objects;
4 import java.util.Optional;
5
6 public class Player {
7
8     private String id;
9
10    private String name;
11
12    private Year start;
13
14    private Year end;
15
16    private Email email;
17
18    private Position position;
19
20    private MonetaryAmount salary;
```

```
21  
22 // ...  
23  
24 }
```

Builder Pattern

O segundo e último passo será utilizar o padrão Builder para fazer com que as regras de validação estejam dentro dessa classe. Ao adotar esse padrão, garantiremos que o objeto só será instanciado quando os dados forem realmente válidos, além de colocar a responsabilidade dessa criação em uma classe. Esse padrão é muito interessante porque, além de garantir a responsabilidade única, diminui a chance de um dos parâmetros ser trocado accidentalmente.

Um ponto importante é que muitos dos frameworks de mapeamento, como o Hibernate, OpenJPA etc., requerem getters e setters, além do construtor padrão. Uma solução para isso seria criar um construtor com todos os parâmetros necessários em privado e um outro como default e com a anotação Deprecated, deixando claro para a pessoa desenvolvedora que aquele construtor não é bem-visto para uso. Veja no exemplo abaixo que o Builder fica como inner class da classe Player:

```
1 import javax.money.MonetaryAmount;  
2 import java.time.Year;  
3 import java.util.Objects;  
4 import java.util.Optional;  
5  
6 public class Player {  
7  
8     static final Year SOCCER_BORN = Year.of(1863);  
9 }
```

```
10     //hide
11
12     private Player(String name, Year start, Year end, Email email, Position position, MonetaryAmount salary) {
13         this.name = name;
14         this.start = start;
15         this.end = end;
16         this.email = email;
17         this.position = position;
18         this.salary = salary;
19     }
20
21
22     @Deprecated
23     Player() {
24     }
25
26     public static PlayerBuilder builder() {
27         return new PlayerBuilder();
28     }
29
30     public static class PlayerBuilder {
31
32         private String name;
33
34         private Year start;
35
36         private Year end;
37
38         private Email email;
39
40         private Position position;
41
42         private MonetaryAmount salary;
43
44         private PlayerBuilder() {
```

```
45      }
46
47      public PlayerBuilder withName(String name) {
48          this.name = Objects.requireNonNull(name, "name is required");
49          return this;
50      }
51
52
53      public PlayerBuilder withStart(Year start) {
54          Objects.requireNonNull(start, "start is required");
55          if (Year.now().isBefore(start)) {
56              throw new IllegalArgumentException("you cannot start in the future");
57          }
58          if (SOCCER_BORN.isAfter(start)) {
59              throw new IllegalArgumentException("soccer was not born on this time");
60          }
61          this.start = start;
62          return this;
63      }
64
65
66      public PlayerBuilder withEnd(Year end) {
67          Objects.requireNonNull(end, "end is required");
68      };
69
70
71      if (start != null && start.isAfter(end)) {
72          throw new IllegalArgumentException("the last year of a player must be equal or higher than the start.");
73      }
74
75      if (SOCCER_BORN.isAfter(end)) {
76          throw new IllegalArgumentException("soccer
```

```
80    r was not born on this time");
81        }
82        this.end = end;
83        return this;
84    }
85
86    public PlayerBuilder withEmail(Email email) {
87        this.email = Objects.requireNonNull(email, "e\
88 mail is required");
89        return this;
90    }
91
92    public PlayerBuilder withPosition(Position positio\
93 on) {
94        this.position = Objects.requireNonNull(positi\
95 on, "position is required");
96        return this;
97    }
98
99    public PlayerBuilder withSalary(MonetaryAmount sa\
100 lary) {
101        Objects.requireNonNull(salary, "salary is req\
102 uired");
103        if (salary.isNegativeOrZero()) {
104            throw new IllegalArgumentException("A pla\
105 yer needs to earn money to play; otherwise, it is illegal\
106 .");
107        }
108        this.salary = salary;
109        return this;
110    }
111
112    public Player build() {
113        Objects.requireNonNull(name, "name is require\
114 d");
```

```
115         Objects.requireNonNull(start, "start is requi\\
116     red");
117         Objects.requireNonNull(email, "email is requi\\
118     red");
119         Objects.requireNonNull(position, "position is\\
120     required");
121         Objects.requireNonNull(salary, "salary is req\\
122     uired");
123
124     return new Player(name, start, end, email, po\\
125     sition, salary);
126 }
127
128 }
129 }
```

Dessa forma, teremos a certeza de que, quando a instância de um jogador (Player) for criada, ela será consistente e à prova de falhas.

```
1   CurrencyUnit usd = Monetary.getCurrency(Locale.US);
2   MonetaryAmount salary = Money.of(1_000_000, usd);
3   Email email = Email.of("marta@marta.com");
4   Year start = Year.now();
5   Year end = start.plus(-1, ChronoUnit.YEARS);
6
7   Player marta = Player.builder().withName("Marta")
8       .withEmail(email)
9       .withSalary(salary)
10      .withStart(start)
11      .withPosition(Position.FORWARD)
12      .build();
```

Podemos aplicar esse mesmo princípio na criação de um time (Team):

```
1   Team bahia = Team.of("Bahia");
2   Player marta = Player.builder().withName("Marta")
3       .withEmail(email)
4       .withSalary(salary)
5       .withStart(start)
6       .withPosition(Position.FORWARD)
7       .build();
8
9   bahia.add(marta);
```

Utilizando Bean Validation

Uma outra maneira de garantir a validação dos dados é utilizando recursos do [Bean Validation](#). O Bean Validation é uma

especificação Java cujo objetivo é garantir que os atributos dentro da classe sejam válidos através da utilização exclusiva de anotações, ou seja, de uma maneira simples e reaproveitável.

É importante salientar que todas as regras citadas anteriormente continuam válidas com o uso dessa API e que é importante que sejam mantidas todas as boas práticas de orientação a objetos, como encapsulamento. Assim, o uso do Bean Validation pode ser visto como uma dupla verificação ou para que o Builder execute as validações oriundas do framework, de modo que ele só retorne a instância caso todas as validações sejam bem-sucedidas.

```
1 import javax.money.MonetaryAmount;
2 import javax.validation.constraints.NotBlank;
3 import javax.validation.constraints.NotNull;
4 import javax.validation.constraints.PastOrPresent;
5 import javax.validation.constraints.PositiveOrZero;
6 import java.time.Year;
7 import java.util.Objects;
8 import java.util.Optional;
9
10 public class Player {
11
12     static final Year SOCCER_BORN = Year.of(1863);
13
14     @NotBlank
15     private String name;
16
17     @NotNull
18     @PastOrPresent
19     private Year start;
20
21     @PastOrPresent
22     private Year end;
23 }
```

```
24     @NotNull
25     private Email email;
26
27     @NotNull
28     private Position position;
29
30     @NotNull
31     private MonetaryAmount salary;
32
33     @PositiveOrZero
34     private int goal = 0;
35
36     //continue
37
38 }
39
40 import javax.validation.constraints.NotBlank;
41 import javax.validation.constraints.NotNull;
42 import javax.validation.constraints.Size;
43 import java.util.ArrayList;
44 import java.util.Collections;
45 import java.util.List;
46 import java.util.Objects;
47
48 public class Team {
49
50     static final int SIZE = 20;
51
52     @NotBlank
53     private String name;
54
55     @NotNull
56     @Size(max = SIZE)
57     private List<Player> players = new ArrayList<>();
58 }
```

```
59     //continue  
60  
61 }
```

O código fonte desse exemplo está disponível no repositório:
<https://github.com/soujava/bulletproof>.

TIP: Lembre-se da importância dos testes de unidade em todo o processo de desenvolvimento!

Com a criação desse exemplo, demonstramos que apenas ao utilizar conceitos de orientação a objetos você estará criando um código à prova de falhas. Até esse momento, todas as práticas funcionam de maneira agnóstica ao banco de dados, ou seja, podemos utilizar essas boas práticas independentemente da tecnologia de persistência que será adotada.

Lombok: problema ou solução?

De uma maneira geral, o projeto Lombok é uma biblioteca famosa e conhecida por reduzir a quantidade de linhas de código através da utilização de anotações. Essa ferramenta tem seus benefícios, uma vez que a redução de código pode facilitar a leitura. Um exemplo é a anotação `Builder`, que permite a criação de classes mais intuitivas e no padrão `Builder`. Por outro lado, também vemos alguns problemas no uso desse projeto, itens que estão listados a seguir:

- O código é gerado pelo projeto Lombok e não é visualizado pela IDE. Qualquer exceção que envolva essas classes tenderá a ser difícil de ser seguida pela pilha de exceção;
- Existem anotações como `@Data` que permitem a quebra do encapsulamento das classes. Com isso, você deixará

de programar utilizando o paradigma de programação orientada a objetos. É importante salientar as informações trazidas no capítulo 7 do livro Clean Code: a maior diferença entre OOP e programação estruturada é que, na primeira opção, nós escondemos os dados para expor o comportamento;

- Boa parte dos códigos como getters e setters podem ser gerados pela IDE;
- Os poderes das anotações são muito tentadores, porém encapsulamento não é sobre ter o atributo privado e com getter e setter públicos, mas sim sobre garantir que atributos sejam acessados com a menor visibilidade possível.

Queremos deixar claro que o objetivo deste tópico não é classificar o Lombok e seu relacionamento ou não com as boas práticas de programação. A intenção é mostrar que, apesar de possuir suas vantagens - expostas em diversos sites -, é importante ter em mente os problemas acarretados com a adoção dessa tecnologia.

Clean Architecture

É interessante como os conceitos de Clean Architecture podem ser relacionados em diversos aspectos com o livro de Domain Driven Design, de Eric Evans. Podemos exemplificar esta relação quando, no livro de DDD, é citada diversas vezes a proposta de criação de uma linguagem próxima do negócio: a linguagem ubíqua. Já no livro Clean Architecture, de Robert C. Martin Series, se fala sobre separar o código de negócio do que importa, ou seja, não amarrar regra de negócio com a tecnologia escolhida.

INFO: Clean Architecture é um livro muito bom e faz parte de uma “trilogia” cuja leitura é recomendada: Clean Code: a Handbook of Agile Software Craftsmanship (Robert C. Martin Series), The Clean Coder: a Code of Conduct for Professional Programmers (Robert C. Martin Series) e, finalmente, Clean Architecture: a Craftsman’s Guide to Software Structure and Design (Robert C. Martin Series).

Assumimos que o leitor tenha realizado leitura prévia do livro Clean Architecture. Este capítulo não tem o objetivo de falar sobre o livro, mas expor como utilizamos e aplicamos seus conceitos com uma visão prática.

Iniciaremos falando sobre a estratégia de dividir e conquistar. Essa estratégia possui muitas vantagens, e vamos citar duas dentre elas:

- **Testes:** Uma das grandes vantagens nessa separação, certamente, se encontra na facilidade de testes, principalmente os testes de unidade. Uma maior separação das

camadas facilita, por exemplo, mockar as camadas de infraestrutura e fazer com que o teste seja barato - em outras palavras, fácil de criar e rápido de executar. Por consequência, esses testes tendem a ser executados constantemente, com a facilidade e maior cobertura. É comum vivenciar projetos em que testes levam horas para serem executados, trazendo consequências negativas. Testes que falham constantemente tendem a ser ignorados, e testes não confiáveis farão com que o time nunca saiba se houve um problema de regressão ou se simplesmente é aquele “erro amigo”.

TIP: Vale salientar que não estamos criticando outros tipos de testes; porém, testes unitários têm vantagens. Ao considerarmos que tecnologias como Hibernate, Java e JPA possuem seus próprios testes, lembre-se de que o que você de fato testar são as regras do seu negócio.

- **Baixo impacto em mudanças:** Uma separação maior do negócio da tecnologia acarreta menores impactos em casos de mudança de tecnologia, como, por exemplo, uma troca entre vendors de banco de dados.

TIP: Na área de arquitetura, o pragmatismo é uma característica crucial. O maior foco de um arquiteto é resolver um problema usando a tecnologia como meio, não como fim. Deste modo, usuários(as) finais não precisam, e muitas vezes, não querem saber qual o banco ou linguagem estão sendo utilizados.

Observe a imagem a seguir:

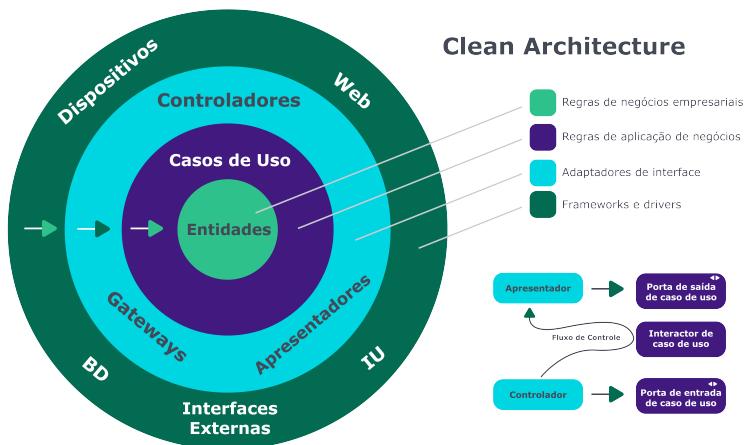


Imagen 04_01: As diferentes camadas da aplicação e como as práticas do Clean Architecture pregam que devem ocorrer as interações entre estas camadas.

Fonte: <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>

Usaremos a imagem acima para discorrer sobre pontos de atenção que vale a pena aplicar em sua arquitetura:

- Tenha uma estratégia de teste eficaz, que siga a pirâmide de testes;
- As estruturas devem ser isoladas em módulos individuais. Quando (não se) mudarmos de ideia, precisaremos fazer mudança em apenas um lugar;
- “Arquitetura Gritante” também conhecida como uso pretendido. Ao olhar para a estrutura do pacote, percebe-se imediatamente o que o aplicativo faz, e não seus detalhes técnicos. É semelhante a quando se utiliza package by layer ao invés de package by feature.
- Toda a lógica de negócios deve estar em um caso de uso, portanto, será fácil encontrar e não duplicar em nenhum outro lugar;

- Será um bom monolito com casos de uso claros que você poderá dividir em microserviços mais tarde, depois de aprender mais sobre eles.

Granularidade de camadas

Algo bastante discutido em livros de arquitetura em geral é a granularidade de camadas. Com o tempo se percebe que as camadas podem ajudar tanto em abstração e separação de responsabilidade de negócio quanto no aumento da complexidade do código.

TIP: Avalie sempre ao colocar mais camadas em uma aplicação e tenha atenção para que elas não se tornem uma arma de destruição ao invés de um item de ajuda.

A estratégia descrita pelo livro Clean Architecture é uma visão de “fora para dentro”, ou seja, a camada framework acessa a camada de adaptação seguindo a linha do princípio da dependência. Descreveremos a seguir, de uma maneira geral, as camadas.

Entidades

Caso você venha do DDD, não verá muita novidade nessa camada e nos conceitos. Ela é responsável por encapsular o domínio do negócio. O ponto principal é que essa camada é o core, ou seja, é a razão de fazer a aplicação em si. Ou seja, é nela que se concentram as regras de negócio e não deve mudar de acordo com itens externos, como, por exemplo, mudança de banco de dados.

TIP: Particularmente, não achamos “crime federal” caso existam tecnologias que tendem a não mudar, por exemplo, bibliotecas utilitárias usadas por toda a empresa. Em uma aplicação menor, podem ser apenas interfaces e classes que tenham que ser utilizadas em todas as camadas, como a interface de um repositório.

Casos de uso

Nesta camada se concentram as ações da regra de negócios. Uma maneira de pensar é que esta seja uma continuação da camada de entidade. Muitas regras que envolvem as entidades ficam muito grandes para caberem apenas na entidade, isso sem mencionar o clássico problema de responsabilidade única que tanto falamos no [SOLID](#).

Interface de adaptação

Um ponto de vista interessante é que essa camada é uma grande implementação do padrão de projeto [Adapter](#). O seu maior objetivo é deixar as camadas de entidades e Casos de Uso mais transparentes. Por exemplo, uma interface repositório pode ter diversas implementações, seja uma base de dados relacional, seja não relacional.

O maior objetivo dessa camada é garantir que as mudanças de tecnologia não impactem as outras camadas. Afinal, para o(a) usuário(a) não importa se o banco de dados é um Cassandra ou um PostgreSQL, mas, num nível técnico, é importante pensar nas diferentes estratégias de modelagem.

Frameworks

Esta é a camada que “não importa” para o negócio. Em outras palavras, é a camada para o “meio” e não para o “fim”. Ela é composta por ferramentas e tecnologias como banco de dados. O maior ponto para a estratégia dessa camada é evitar que ela passe para o menor número possível de camadas. É importante salientar que quanto menos código nessa camada melhor, ou seja, ele terá o necessário para interligar as tecnologias.

Conclusão

O livro Clean Architecture traz uma boa referência para se aplicar em arquiteturas maduras e uma melhor estratégia de como utilizar e comunicar entre as camadas. Um ponto importante: não existe bala de prata, e o livro em questão também não é um. O material é bastante rico e cheio de detalhes, porém, lembre-se de que camadas tendem a aumentar a complexidade do seu código, pois quanto mais camadas são criadas, mais camadas são mantidas. Como diria o livro **fundamentos de arquitetura de software**, tudo é um trade-off, e o bom senso ainda é a melhor ferramenta para que o(a) arquiteto(a) saiba *quando* e *como* aplicar conceitos e práticas.

Refatoração

Após alguns anos de estrada, quem desenvolve percebe que passa grande parte do seu tempo lendo código e, na maioria das vezes, código escrito por outras pessoas. Sem muito esforço, percebe-se que esse tempo é superior ao tempo gasto escrevendo novas linhas de código, afinal de contas, é preciso entender o funcionamento do código atual, onde deverá fazer suas mudanças, e, principalmente, quais classes ou arquivos serão impactados por elas.

Não à toa, uma simples alteração no código, como adicionar um `if`, pode levar horas ou mesmo dias. Geralmente, esse alto custo para manter o software é consequência de uma péssima qualidade do código que foi escrito, afinal, quanto maior a dificuldade em ler e entender um trecho de código, maior será o tempo gasto para alterá-lo. E não se engane, mesmo um(a) dev que implementou a funcionalidade no sistema pode ter dificuldades para ler o código após alguns meses.

Para diminuir o custo nas alterações do sistema, é necessário **investir na qualidade e na clareza do código produzido**, seja no código atual, seja na introdução de novas linhas de código. Em outros termos, deve-se melhorar o código já pronto e que funciona em produção sem mudar o seu comportamento. Essa técnica, conhecida por grande número de profissionais, é chamada de **Refatoração** (ou, em inglês, Refactoring).

Uma das definições mais aceitas na indústria para “Refatoração” é a de **Martin Fowler**, em seu livro **Refactoring: Improving the Design of Existing Code**, segundo a qual:

“Refatoração é uma técnica controlada para

reestruturar um trecho de código existente, **alterando sua estrutura interna sem modificar seu comportamento externo.** Consiste em uma série de pequenas transformações que preservam o comportamento inicial. Cada transformação (chamada de refatoração) reflete em uma pequena mudança, mas uma sequência de transformações pode produzir uma significante reestruturação. Como cada refatoração é pequena, é menos provável que se introduza um erro. Além disso, o sistema continua em pleno funcionamento depois de cada pequena refatoração, reduzindo as chances de o sistema ser seriamente danificado durante a reestruturação.” – *Martin Fowler*

Com base nas palavras de Fowler, podemos entender que refatoração é o processo de modificar um trecho de código já escrito, executando pequenos passos (**baby steps**) sem modificar o comportamento atual do sistema. É uma técnica utilizada para melhorar algum aspecto do código, entre os quais podemos citar melhorias na clareza do código para facilitar a leitura, ou também ajustes no design das classes a fim de trazer maior flexibilidade ao sistema.

Medo de alterar o código que não é seu

Se você trabalha com desenvolvimento, em algum momento da sua carreira, entrou ou entrará no meio de um projeto de software. Isso quer dizer que você chegou (ou chegará) de parquedas para desenvolver, corrigir e manter funcionalidades em cima de uma base de código existente, que pode ter desde 6 meses até 20 anos de vida.

Essa base de código provavelmente já passou pela mão de diversas outras pessoas, desde quem tem mais experiência até quem tem nível mediano e quem faz estágio, geralmente sob pressão e prazos surreais e apertados. Para piorar, dificilmente a equipe que iniciou o projeto e tomou decisões importantes de arquitetura e de design estará ainda na empresa. Ou seja, um sistema com 5 anos de idade provavelmente sofreu trocas no time completo 2 ou 3 vezes - no mínimo.

Essa alta rotatividade de profissionais traz inúmeros prejuízos à empresa, em especial ao código do sistema. Isso porque certos trechos de código críticos foram (e são) alterados, mantidos e evoluídos por diversos profissionais, desde quem domina o negócio de ponta a ponta a quem nem tanto; desde quem tem maestria nas tecnologias e frameworks utilizados a quem não tem a mínima ideia de como eles funcionam; e há também quem tenha utilizado o projeto como laboratório no curto tempo que permaneceu na empresa. Não é de se espantar que boa parte desse código seja repleto de gambiarras e duplicação de rotinas, abarrotada de comentários desatualizados, pouca clareza nas lógicas de negócio, alto acoplamento e baixa coesão. Não se assuste ao ter que manter uma classe ou um método com mais de 5 mil linhas de código macarrônico e totalmente ilegível.

Situações como essa são mais comuns do que se pode imaginar. Lidar com sistemas problemáticos dessa magnitude traz frustração e, principalmente, **medo** para um(a) dev que acaba de entrar na empresa, que terá que trabalhar no código “dos outros”, programando no escuro, sem ter a mínima noção se sua última alteração impacta noutras partes do sistema ou, pior ainda, qual o tamanho desse impacto.

A importância dos testes automatizados na hora refatorar

Quanto menos conhecimento sobre sistema o(a) desenvolvedor(a) tem, maior é sua insegurança na hora de alterar código e menores são as garantias de que suas alterações não causarão mais estragos ao sistema, que já se encontra frágil pelo tempo. Mesmo pequenas refatorações com o intuito de melhorar a clareza ou a simplicidade do código podem gerar novos bugs, reintroduzir bugs antigos no sistema ou impactar o sistema de forma negativa.

Se refatorar código traz riscos, como se pode garantir que erros não serão introduzidos nas suas refatorações?

A verdade é que refatorar código sem uma garantia concreta de que o comportamento atual não vá mudar é um ato imprudente, pois não adianta nada melhorar o código se algo que já existe é quebrado. Essa garantia pode ser obtida de diversas maneiras, como uma equipe de QA (testers) ou um ambiente de homologação para que clientes possam validar a mudança, mas sem dúvida uma das melhores alternativas a um preço justo é através do uso de **Testes Automatizados**. Uma bateria de testes de regressão mostraria rapidamente se uma determinada mudança (ou refatoração) no código mudou o comportamento da funcionalidade ou mesmo impactou outras partes do sistema. Com isso, qualquer erro introduzido seria imediatamente apontado, facilitando a correção a cada passo da refatoração de maneira imediata.

Como dito por Fowler, refatorar é o ato de aplicar pequenas transformações no código existente sem mudar seu comportamento externo. Assim como muitas técnicas de engenharia de software, a teoria é mais fácil do que a prática. Em um processo complexo, como desenvolvimento de software, temos que verificar frequentemente onde estamos e, se necessário, fazer corre-

ções no percurso. No caso da refatoração, a cada transformação precisamos verificar que tudo continua funcionando como esperado para só então partirmos para o passo seguinte. O contrário também é verdade: precisamos identificar se nosso último passo quebrou algo no sistema para então desfazê-lo e adotar uma estratégia diferente. Esse ciclo frequente de verificação em que aprendemos a cada passo recebe o nome de **Feedback Loop**. No mundo de metodologias ágeis, o *Feedback* funciona como a engrenagem motora para todos os quatro valores do Manifesto Ágil e suas derivações encontradas no mercado.

Quanto mais curto e frequente o feedback loop, mais natural o processo de refatoração se torna. Mas como encurtá-lo? Bem, existem diversas práticas e ferramentas que podem ajudar, como pair programming, servidor de integração contínua (CI) ou code review, mas entre elas eu quero destacar a cobertura de testes. Quando temos código coberto por testes, esse ciclo é encurtado de tal forma que fica natural para quem desenvolve repeti-lo a todo momento. Basta um teste de unidade quebrar que sabemos de imediato o que foi feito de errado. Perceba que os testes funcionam como uma rede de segurança para encorajar quem desenvolve a refatorar código e, principalmente, a manter a motivação para tornar essa prática uma constante no seu dia a dia.

Não vou mentir, é possível fazer refatorações sem uma linha de teste automatizado, muitas empresas e equipes fazem isso; contudo, não se pode ignorar que há grandes riscos envolvidos nessa prática que podem introduzir bugs ou gerar prejuízos para a empresa ou cliente final. Refatorar código sem testes torna tudo mais difícil e arriscado, consome mais tempo do que o necessário, desencoraja melhorias no código e ainda exige muito mais da pessoa desenvolvedora e da equipe. Para refatorar sem testes, a experiência de quem desenvolve conta bastante, seja para fazer mudanças com passos pequenos e seguros, analisar e mensurar o impacto de suas (possíveis) mudanças, seja para

simplesmente decidir que não vale a pena refatorar determinado trecho de código.

Apesar de a experiência de quem desenvolve ser um fator importante na hora de refatorar código não coberto por testes, dependendo da empresa e dos processos adotados por ela, as chances são de que gestores não permitam modificar um código que funciona e já se encontra em produção.

O tema testes automatizados é muito amplo e merece um capítulo completo ou mesmo um livro - e por sinal existem alguns muito bons. Mas para não alongar neste tópico, deixo o seguinte questionamento: **na hora de refatorar, quanto longe podemos ir, ou quanto ousados podemos ser sem uma boa cobertura de testes?**

Refatoração contínua do código

Um ponto importante é que dificilmente a empresa ou stakeholder permitirá que se crie uma tarefa inteira para realizar a refatoração de um projeto. Afinal, para o(a) usuário(a) final, pouco impacta a tecnologia ou a qualidade de código que o projeto está usando. Uma boa estratégia é a famosa técnica de escoteiro. Essa técnica se baseia em refatorar os códigos relacionados que precisam de melhoria e clareza, ao passar por uma história que agrupa valor para o produto. Sempre salientando que, por segurança, é importante adicionar testes antes de fazer qualquer refatoração.

Qual sua motivação para refatorar o código?

Existem diversas motivações para realizar uma refatoração do código, e listamos aqui alguns dos principais:

- **Refatorando para legibilidade e flexibilidade:** Legibilidade traz diversos benefícios para o ciclo de vida de uma aplicação, seja como agilidade na manutenabilidade, seja como facilidade de depurar, além do fator de bem-estar dentro do time. Vale ressaltar que essa manutenção dificilmente entrará como uma história ou atividade, porém, se abster totalmente nessa atividade acarreta o fator “Janelas partidas”, ou seja, à medida que o código for piorando a legibilidade, maior é a tendência de que a próxima pessoa que realizar manutenção abrirá mão da qualidade para se livrar dessa atividade o mais rápido possível até que o código seja um risco pelo fator da falta de legibilidade.
- **Refatorando para performance:** Performance é sempre importante, porém, deve-se considerar se essa melhoria resultará em maior complexidade ou algum outro eventual problema de consistência de dados. Como diria Donald Knuth, “a otimização prematura é a raiz de todo o mal”. Assim, toda melhoria de performance é importante, porém leve em consideração os seus impactos e se isso faz sentido nesse momento.
- **Refatorando para remover duplicidade:** Reduzir código dentro do projeto é uma boa motivação. Quando somos juniores tendemos a ficar felizes quando adicionamos código, e quando ficamos mais experientes ficamos felizes com a redução de código de que precisamos. Uma das estratégias de reduzir o código é, justamente, evitar a duplicação de código. Isso te garantirá um único ponto para refatoração: facilidade de teste, além de performance, prin-

cipalmente se você trabalha dentro da JVM, uma vez que existe o fator JIT.

- **Refatorando para usar uma biblioteca:** O melhor código, certamente, é aquele que não escrevemos. Reduzir a quantidade de código significa reduzir a complexidade do seu lado, além de diminuir os pontos que darão erro. Isso é excelente, principalmente quando essa biblioteca já esteja disponível dentro do seu projeto. Vale salientar que excesso de dependência também pode ser perigoso. Lembre-se de que, quanto maiores as suas dependências, maior a estratégia para atualizar as bibliotecas (além do conhecido problema de “Jar Hell”).

Conclusão

Com isso concluímos o tópico sobre refatoração do código. Abordamos pontos importantes, como a motivação de se refatorar o código além dos seus riscos. Como bons profissionais de desenvolvimento, é sempre importante colocar na balança as motivações e evitar problemas futuros de manutenabilidade e legibilidade do código. Como sempre, o bom senso será sua melhor ferramenta.

NoSQL vs. SQL

Muitas das vezes, quando iniciamos um debate sobre SQL (Structured Query Language) e NoSQL no meio técnico, já podemos esperar o surgimento de discussões acaloradas. Isso se deve ao fato, principalmente, de que existe uma ideia de que há uma guerra, uma rixa etc. que precisa ser vencida entre quem faz uso dessas tecnologias. Já queremos deixar claro, não existe guerra alguma! Vamos desfazer esse mito?

Para começar, precisamos falar um pouquinho de história. Com isso, entenderemos os detalhes sobre o NoSQL e suas diferenças para o SQL.

Assim como sempre ocorre no universo da tecnologia, nós criamos uma solução para resolver um problema específico. Vamos voltar mais de quarenta anos no tempo para entender o problema existente no contexto dos bancos de dados:

- Para que pudessem ser utilizados posteriormente, os dados produzidos pelos computadores deveriam ficar armazenados e não poderiam ser perdidos quando as máquinas fossem desligadas.

Porém, naquela época, a tecnologia de armazenamento estava engatinhando, e os discos rígidos eram muito caros. Foi necessário então estruturar e normalizar os dados que seriam gravados nesses discos. Com a adoção dessa prática, os dados armazenados utilizariam menos espaço e, consequentemente, trariam um aproveitamento melhor de recursos, gerando economia ou a possibilidade de se armazenar maior quantidade de dados em um mesmo espaço.

Esse foi o contexto em que os bancos de dados relacionais foram criados, inclusive implementando todos os conceitos detalhados no capítulo anterior, como formas normais e estrutura de dados bem definidas.

O tempo passou, nossa tecnologia evoluiu e, com essa evolução, mais um problema surgiu: produzimos dados em uma quantidade muito grande, muitas vezes de forma desestruturada e também descentralizada, com sistemas cada vez mais distribuídos.

TIP: Esses dados são chamados de *desestruturados* por terem origem em diversas fontes, como sensores IOT (Internet of Things, i.e. geladeiras conectadas à internet, relógios inteligentes e carros autônomos), imagens e documentos não catalogados, dentre outros exemplos.

Estruturar, isto é, organizar os dados provenientes de fontes como estas era (e é) possível, porém iria requerer muito tempo. Esse tempo extra impactaria o processo de desenvolvimento e entrega de software e, consequentemente, levaria as empresas a perderem o time to market da solução sendo criada. Esse problema precisava ser resolvido e, assim, nasceram os bancos de dados NoSQL!

O termo **NoSQL** foi originalmente criado em 1998 por Carlo Strozzi e, posteriormente, reintroduzido por Eric Evans em 2009, quando este participou da organização de um evento para discutir bancos de dados **open source** e **distribuídos**. E por falar em bancos distribuídos, esse é um conceito amplamente utilizado pelos bancos NoSQL: basicamente, os bancos NoSQL são bancos de dados que operam em computação distribuída, o que impulsiona um significativo grau de escalabilidade e performance.

TIP: Para entender um pouco mais sobre computa-

ção distribuída, recomendamos a leitura do seguinte artigo: [Paradigma da computação distribuída](#).

NoSQL

O que significa NoSQL?

Não existe uma definição, digamos, “oficial” para o que realmente esse termo significa, mas particularmente gosto da seguinte: **Not Only SQL**, ou seja, “Não Somente SQL”. Essa definição enfatiza que esses bancos podem utilizar linguagens semelhantes ao [SQL ANSI](#) para realizar consultas e demais operações e não somente o SQL em si.

Particularidades do NoSQL

Quando você estudou sobre bancos de dados relacionais, viu que esses bancos são baseados no conceito **ACID** (**A**tomicity, **C**onsistency, **I**solation, **D**urability). Os bancos de dados NoSQL, em sua grande maioria, baseiam-se em um outro conceito: o **BASE** (**B**ase Availability, **S**oft State and **E**ventually Consistent).

Antes de detalhar cada ponto do conceito **BASE**, você precisa entender a representação de um termo no contexto de banco de dados que verá ao longo deste capítulo: **cluster**. Um cluster, nesse contexto, se refere à capacidade de um conjunto de **servidores** (de banco) ou **instâncias** (de banco) se conectarem a um banco de dados. Uma **instância** é uma coleção de memória e processos que interagem com o banco de dados, que é o conjunto de arquivos físicos que efetivamente armazenam os dados.

Devemos destacar duas principais vantagens de um cluster, especialmente em ambientes de banco de dados de alto volume:

- **Tolerância a falhas** (*Fault Tolerance*): como há mais de um servidor ou instância para os usuários se conectarem, o cluster de bancos de dados oferece uma alternativa no caso de falha em um servidor. Quando se lida com dezenas de milhares de máquinas em um único *data center*, tais falhas são um problema presente;
- **Balanceamento de carga** (*Load Balancing*): o cluster geralmente é configurado para permitir que a aplicação cliente seja automaticamente alocada ao servidor com o mínimo de uso para que, assim, se otimize o uso da estrutura disponível para o banco.

Detalhando o conceito BASE

- Base Availability - **BA**
 - O banco de dados aparenta funcionar o tempo todo. Como existe a implementação do conceito de cluster, se um servidor falhar, o banco continuará funcionando por conta de outro servidor que suprirá essa falha;
- Soft State - **S**
 - Não é necessário estar consistente o tempo todo. Ou seja: com um banco distribuído em várias máquinas e todas sendo usadas com igual frequência para escrita e consulta, é possível que, em dado momento, uma máquina receba uma escrita e não tenha tido tempo de “repassar” essa escrita para as demais máquinas do banco. Assim, se uma aplicação consultar a máquina que já foi atualizada e outra o fizer numa máquina menos atualizada, os resultados, que deveriam ser iguais, serão diferentes. Imagine a sua *timeline* do **Facebook**: nela são exibidos os posts, porém nem todos os posts são exibidos exatamente ao mesmo tempo. Nesse caso, o que acontece é que a informação foi enviada ao banco de dados, mas nem todos

os servidores do cluster têm essa mesma informação ao mesmo tempo. Isso permite que o banco de dados possa gerenciar mais informações de escrita sem ter que se preocupar em replicá-las em uma mesma operação;

- Eventually Consistent - **E**

- O sistema se torna consistente em algum momento. Em outras palavras, ele eventualmente se tornará consistente. Como não temos a informação replicada “instantaneamente”, esse conceito se encarrega de deixar o banco consistente “ao seu tempo”. Isso porque, dependendo das configurações do cluster, essa replicação pode acontecer mais rapidamente ou não. Mas em algum momento as informações estarão consistentes e presentes em todos os servidores do cluster.

Para finalizar, uma outra particularidade marcante de bancos não estruturados é a ausência da feature de schema ou schema flexível. Isso quer dizer que não há necessidade de definição prévia do schema dos dados. Se, por um lado, isso torna mais dinâmico o processo de inclusão de novos atributos, por outro pode impactar a integridade desses dados. Não se preocupe: a seu tempo, todos esses conceitos ficarão bem mais claros.

NoSQL e suas classes

Os bancos de dados NoSQL podem ser categorizados em quatro tipos (que, no contexto de banco de dados, são chamados de **classes**):

- Chave / Valor - Key / Value
- Família de Colunas - Column Family
- Documentos - Document
- Grafos - Graph

Cada classe acima possui uma aplicação diferente, portanto, devemos entender as características de cada classe para tirar o melhor proveito de cada uma e saber quando escolhê-las.

Classe Key / Value

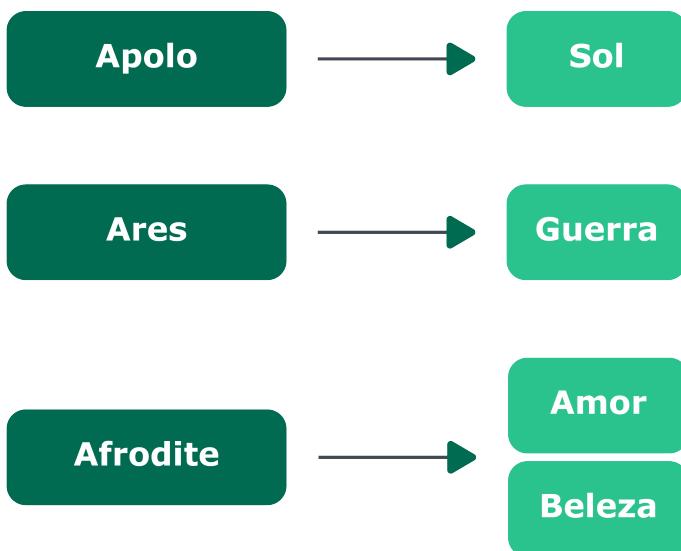


Imagen 06_01: Exemplo de estrutura de dados em um banco Key/Value

Os bancos do tipo chave-valor possuem uma estrutura similar à da classe `java.util.Map` do Java, ou seja, a *informação* (`value`) será recuperada apenas pela *chave* (`key`). Esse tipo de banco de dados pode ser utilizado, por exemplo, para gerenciar sessões de usuários logados. Outro exemplo de utilização é aliado a um DNS, onde a chave é o endereço, por exemplo, `www.google.com`, e o valor é o IP desse servidor.

Atualmente existem diversas implementações de banco de dados do tipo chave-valor, dentre os quais os mais famosos são:

- AmazonDynamo
- AmazonS3
- Redis
- Scalaris
- Voldemort

Comparando o banco de dados relacional com o do tipo chave-valor, podemos elencar alguns pontos de atenção. O primeiro é que a estrutura do chave-valor é bastante simples:

Estrutura relacional	Estrutura chave-valor
Table	Bucket
Row	Key/value pair
Column	--
Relationship	--

Nessa classe de banco, não é possível realizar operações como `join` entre os buckets, e o valor é composto por um grande bloco de informação, ao invés de ser subdividido em colunas, como ocorre na base de dados relacional.

Classe Column Family

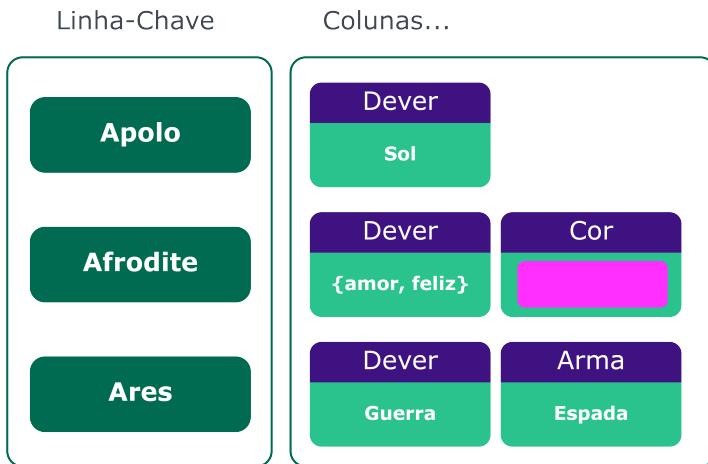


Imagen 06_02: Exemplo de estrutura de dados armazenadas em um banco Column Family.

Esse modelo se tornou popular através do paper “[Bigtable: a Distributed Storage System for Structured Data](#)”, criado por funcionários do Google, com o objetivo de montar um sistema de armazenamento de dados distribuído, projetado para ter um alto grau de escalabilidade e de volume de dados. Assim como o chave-valor, para realizar uma busca ou recuperar alguma informação dentro do banco de dados, é necessário utilizar o campo que funciona como um identificador único, que seria semelhante à chave na estrutura chave-valor. Porém as semelhanças terminam por aí.

Em bancos do tipo Column Family, as informações são agrupadas em colunas: uma unidade da informação (composta pelo nome) e a informação em si.

Esses tipos de bancos de dados são importantes quando se lida com um alto grau de volume de dados, e quando é necessário

distribuir essas informações entre diversos servidores. Mas vale salientar que a sua operação de leitura é bastante limitada, semelhante ao chave-valor, pois a busca da informação é definida a partir de um campo único ou uma chave. Existem diversos bancos de dados que utilizam essas estruturas. Podemos citar:

- Hbase
- Cassandra
- Scylla
- Clouddata
- SimpleDb
- DynamoDB

Dentre os tipos de bancos de dados do tipo família de coluna, o Apache Cassandra é o mais famoso. Caso uma aplicação necessite lidar com um grande volume de dados e com fácil escalabilidade, o Cassandra é certamente uma boa opção.

Ao contrapor o banco do tipo família de coluna com os bancos relacionais, é possível perceber que as operações, em geral, são muito mais rápidas. É mais simples trabalhar com grandes volumes de informações e servidores distribuídos em todo o mundo, porém isso tem um custo: a leitura desse tipo de banco de dados é bem limitada. Por exemplo, não é possível realizar uniões entre família de colunas como no banco relacional. A família de coluna permite que se tenha um número ilimitado de coluna, que por sua vez é composta por nome e a informação, exatamente como mostra a tabela a seguir:

Estrutura relacional	Estrutura de família de colunas
Table	Column Family
Row	Column
Column	Nome e valor da Column
Relacionamento	Não tem suporte

Classe Document

Os bancos de dados orientados a documento têm sua estrutura muito semelhante a um arquivo JSON ou XML. Eles são compostos por um grande número de campos, que são criados em tempo de execução, gerando grande flexibilidade, tanto para a leitura como para escrita da informação.

```
{  
    "name": "Diana",  
    "duty": [  
        "Hunt",  
        "Moon",  
        "Nature"  
    ],  
    "age": 1000,  
    "siblings": {  
        "Apollo": "brother"  
    }  
}
```

Imagen 06_03: Estrutura do dado armazenado em um banco orientado a documentos

Eles permitem que seja realizada a leitura da informação por campos que não sejam a chave. Algumas implementações, por

exemplo, têm uma altíssima integração com motores de busca, o que os torna cruciais para a realização de análise de dados ou logs de um sistema. Veja abaixo algumas implementações dos bancos de dados do tipo documento:

- AmazonSimpleDB
- ApacheCouchDB
- MongoDB
- Riak

Destes, o mais popular é o MongoDB. Ao comparar com uma base relacional, apesar de ser possível realizar uma busca por campos que não sejam o identificador único, os bancos do tipo documentos não têm suporte a relacionamento.

Estrutura relacional	Estrutura de documentos
Table	Collection
Row	Document
Column	Key/value pair
Relationship	-

Uma outra característica de bancos do tipo documento é que, no geral, são *schemaless*.

Classe Graph



Imagem 06_04: Como os dados são armazenados em um banco de classe Grafo.

O banco do tipo grafo é uma estrutura de dados que conecta um conjunto de vértices através de um conjunto de arestas. Os bancos modernos dessa categoria suportam estruturas de grafo multirrelacionais, em que existem diferentes tipos de vértices (representando pessoas, lugares, itens) e diferentes tipos de arestas. Os sistemas de recomendação que acontecem em redes sociais são o maior case para o banco do tipo grafo. Veja abaixo alguns exemplos desse tipo de banco:

- Neo4j
- InfoGrid
- Sones
- HyperGraphDB

Dos tipos de banco de dados mais famosos no mundo NoSQL, o grafo possui uma estrutura distinta com o relacional.

Multi-model database

Alguns bancos de dados possuem a comum característica de ter suporte de uma ou mais classes apresentadas. São exemplos:

- OrientDB
- Couchbase

Teorema do CAP

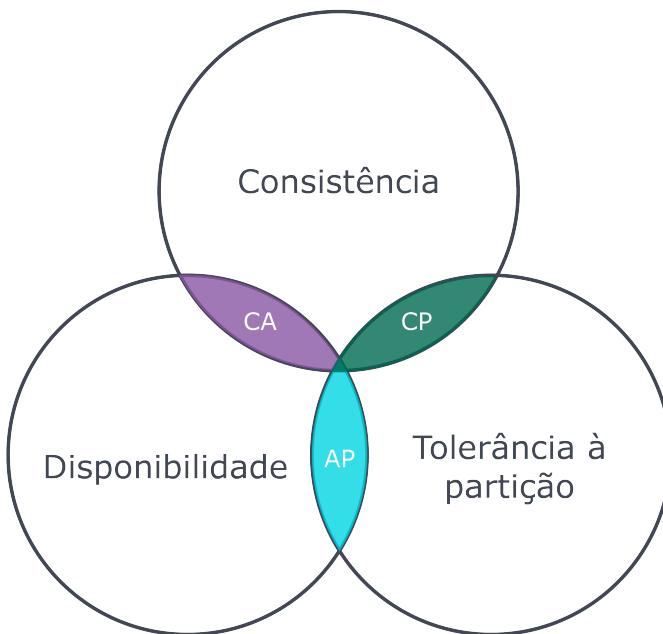


Imagen 06_05: Teorema do CAP.

Um dos grandes desafios dos bancos de dados NoSQL é que eles lidam com a persistência distribuída, ou seja, as informações ficam localizadas em mais de um servidor. Foram criados diversos estudos para ajudar nesse desafio de persistência distribuída, mas o mais famoso foi uma teoria criada em 1999, o Teorema do CAP.

Esse teorema afirma que é impossível que o armazenamento de dados distribuído forneça simultaneamente mais de duas das

três garantias seguintes:

- *Consistência*: uma garantia de que cada nó em um cluster distribuído retorna a mesma gravação mais recente e bem-sucedida. Consistência refere-se a cada cliente com a mesma visão dos dados.
- *Disponibilidade*: cada pedido recebe uma resposta (sem erro) - sem garantia de que contém a escrita mais recente.
- *Tolerância à partição*: o sistema continua a funcionar e a manter suas garantias de consistência, apesar das partições de rede. Os sistemas distribuídos que garantem a tolerância continuam operando, mesmo que aconteça alguma falha em um dos nós, uma vez que existe, pelo menos, um nó para operar o mesmo trabalho e garantir o perfeito funcionamento do sistema.

De uma maneira geral, esse teorema explica que não existe mundo perfeito. Quando se escolhe uma característica, perde-se em outra, como consequência. Em um mundo ideal, um banco de dados distribuído conseguiria suportar as três características, porém, na realidade, é importante saber o que se perderá quando escolher entre um e outro.

Por exemplo, o Apache Cassandra é AP, ou seja, sua arquitetura focará em tolerância a falha e disponibilidade. Existirão perdas na consistência, assim, em alguns momentos um nó retornará informação desatualizada.

Porém, o Cassandra tem o recurso de nível de consistência, de modo que é possível fazer com que algumas requisições ao banco de dados sejam enviadas a todos os nós ao mesmo tempo, garantindo consistência. Vale ressaltar que, fazendo isso, ele perderá o A, de *availability* do teorema do CAP, da disponibilidade.

Escalabilidade vs Complexidade

No mundo NoSQL, cada classe de banco tem o objetivo de resolver problemas particulares. Como o gráfico abaixo mostra, existe um balanço entre o modelo de complexidade: modelos que permitem mais complexidade em modelagem e busca resultam em menos escalabilidade. Como exemplo, temos o banco de classe chave-valor, que é mais escalável, porém permite menos complexidade, uma vez que as queries são baseadas apenas na chave.

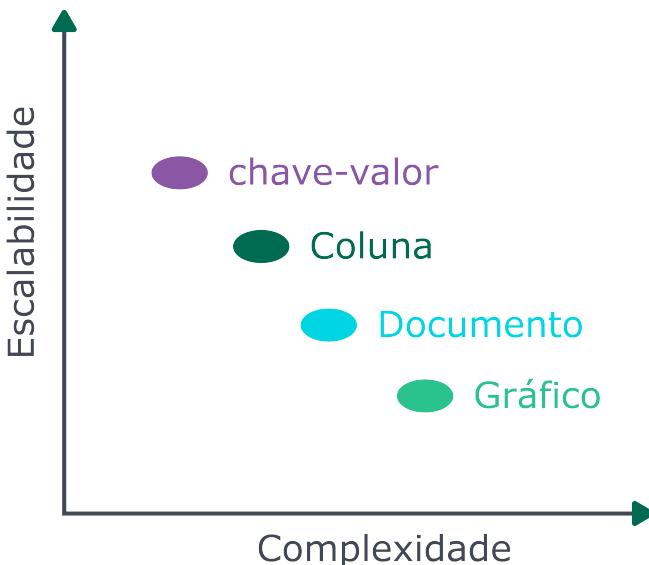


Imagen 06_06: Mapeamento dos tipos de banco de dados e a matriz Escalabilidade vs Complexidade.

Master/Slave vs Masterless

Em linha geral, a persistência no mundo NoSQL possui duas maneiras de comunicação entre os servidores:

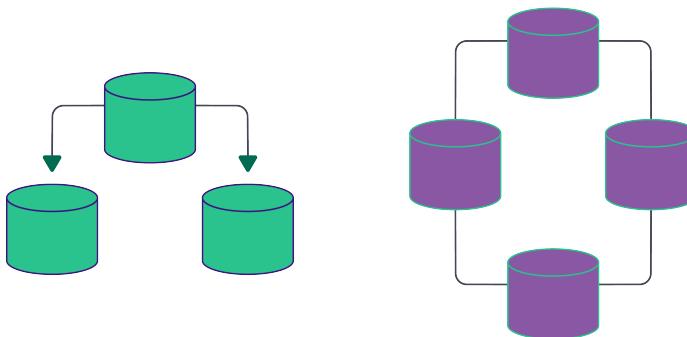


Imagen 06_07: Estrutura de banco Master/Slave e Masterless.

- *O Master/Slave:* é o modelo de comunicação que se caracteriza por um controle unidirecional de um ou mais dispositivos. Em linhas gerais, o nó master é utilizado para a escrita e para a replicação das informações para todos os nós escravos, que, por sua vez, são responsáveis por realizar a leitura das informações. Dessa maneira, é possível garantir maior consistência de dados. Como há um único ponto para a escrita, é possível ter suporte a comportamentos como, por exemplo, transação. Porém existe um ponto de falha: o master. Caso o servidor fique fora do ar, teremos problemas com a escrita. Em cenários como este, bancos de dados modernos conseguem realizar a eleição de um novo nó master de maneira automática.
- *Masterless:* é o modelo de comunicação que se caracteriza por um controle multidirecional por um ou mais dispositivos. Ou seja, não existe um único nó responsável por leitura ou escrita. Cada nó pode ser responsável pelas duas operações. Assim, não existe nenhum ponto de falha, a elasticidade acontece de maneira natural, porém a consistência da informação se torna mais difícil, uma vez que é necessário um certo tempo para que os nós tenham a informação mais atualizada.

Conclusão

Este capítulo teve como objetivo dar o pontapé inicial para os bancos de dados não relacionais. Foram discutidos os principais conceitos acerca de bancos não estruturados, as classes de bancos e suas estruturas. Com esse novo paradigma de persistência não estruturada, as portas se abrem para novas possibilidades e novos desafios na implementação de aplicações.

Os bancos de dados NoSQL vieram para enfrentar e sustentar a nova era das aplicações, na qual velocidade e menor tempo de resposta são um grande diferencial e fator decisivo na hora da escolha. Com este capítulo introdutório, você está apto (a) para seguir desbravando os bancos não relacionais, como o Cassandra, MongoDB, Neo4J e vários outros.

Arquitetura de microsserviços

Com a popularização da utilização de ambientes de cloud para entrega de software, a [arquitetura orientada a microsserviços](#) passou a ser cada vez mais adotada. O quanto confortável a comunidade de TI pode se sentir acerca da adoção dessa arquitetura?

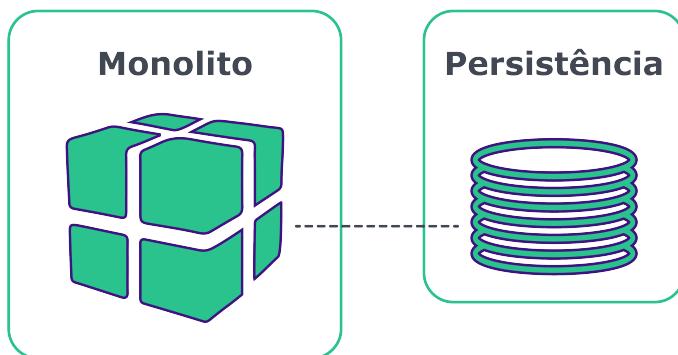
Com base na análise de [tendências de arquitetura e design de software realizada em Abril de 2020](#), podemos assumir que o conhecimento acerca dos benefícios e desafios da adoção desse modelo estão bem estabelecidos, uma vez que sua adoção já chegou à categoria de usuários(as) classificados(as) como “Late Majority”. Existem várias histórias de sucesso e de desastre de organizações que optaram pelo uso desse modelo, portanto, nosso ecossistema se encontra repleto de conhecimento como “por onde começar” e “lições aprendidas”.

Antes de entrar em mais detalhes sobre essa arquitetura, vamos brevemente recapitular os desafios da arquitetura monolítica que antecederam a criação desse novo modelo arquitetural.

Arquitetura monolítica

Em uma arquitetura monolítica encontramos uma aplicação cujos front-end e back-end são parte de um artefato único. Nesse artefato estão contidos todos componentes funcionais, que são compilados e disponibilizados em conjunto. A escalabilidade é impactada, uma vez que, sempre que é necessário escalar essa aplicação, é necessário também prover recursos para a

execução de todos os seus componentes - mesmo aqueles que não precisavam ser escalados. No cenário de persistência, é muito comum se encontrar a relação de um banco para um monolito, porém existem monolitos que trabalham acessando múltiplos bancos de dados (o que aumenta ainda mais o nível de complexidade de manutenção).



Vamos falar dos benefícios dessa arquitetura:

- Facilidade de manutenção, afinal, quanto menos camadas físicas, menos pontos para se verificar, especialmente quando ainda está pequeno;
- Facilidade de sincronizar os dados entre a pessoa usuária e o sistema - a melhor e mais rápida maneira de trocar informações é pela memória. Pensando num banco de dados relacional, por exemplo, a sincronização de dados acontece de maneira mais simples com os clássicos JOINs, ao invés de realizar todo o processo de sincronização ou orquestração entre serviços.
- Consistência de dados tende a ser facilitada em comparação a um sistema distribuído. Salientamos que sempre teremos o problema do CAP quando falamos de arquitetura distribuída. Para aplicações que precisam, por exemplo,

transação, no que no monolito seria um simples rollback, num sistema de microservices seria o caro e complexo padrão SAGA.

- Iniciar um projeto tende a ser muito simples com o monolito - é necessário planejar e arquitetar um ecossistema com menos camadas e componentes.
- O monitoramento de uma única aplicação é mais simples.

Com o passar dos ciclos de desenvolvimento, essas aplicações monolíticas tendem a crescer tanto em quantidade de linhas de código quanto em consumo de hardware, o que acarreta em:

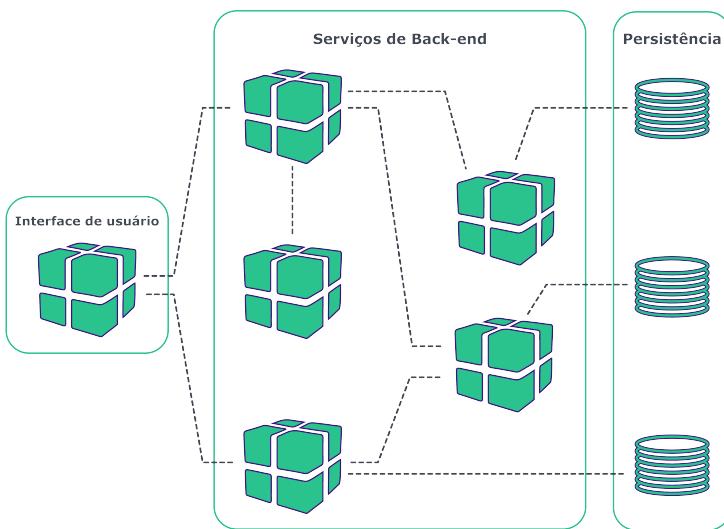
- As chances de se quebrar a aplicação ao realizar alterações pontuais passam a ser cada vez maiores;
- Quanto maior a aplicação, maior o número de testes unitários, o que exige um maior tempo de build sempre que se precisa compilar a aplicação;
- Um maior tempo de build provoca um processo de entrega mais longo;
- É comum que a área de negócio queira realizar testes de validação da aplicação ao liberar uma nova versão. Esses testes também requerem mais esforço, uma vez que - apesar de uma porção pequena ter sido alterada - toda a aplicação foi atualizada. Desta forma, os testes de homologação se tornam mais demorados;
- Evolução e migração dos dados requerem um maior planejamento prévio, uma vez que alterações em banco podem parar o funcionamento de toda a aplicação;
- A aplicação naturalmente passa a ocupar e consumir maior espaço em memória e possuir um maior tempo de start-up.
- Mudanças e entregas críticas exigem grande mobilização na organização e planejamento prévio para disponibilização em produção.

Notamos um dos maiores impasses: o acoplamento, em termos de código e de deploy. Quando a aplicação começa a se tornar muito complexa, com vários times de desenvolvimento e muitas funcionalidades, as adições ou alterações começam a se tornar cada vez mais custosas. Escalar um ambiente desses é desafiador.

É aqui que entra o conceito dos microsserviços, que começa a desacoplar esses serviços e dar responsabilidades únicas para os serviços. Nesta abordagem você pode alterar, disponibilizar e escalar de maneira independente todo o ecossistema, seguindo sempre a premissa de não afetar os outros microsserviços.

Microsserviços

A arquitetura orientada a microsserviços traz como preceito a criação de aplicações desacopladas entre si e modeladas conforme o domínio de negócios. Essas aplicações se integram através de diferentes protocolos, e os diversos padrões de comunicação (REST, GRPC, eventos assíncronos, entre outros) podem ser adotados. Com a adoção de uma arquitetura orientada a microsserviços, é possível promover entregas mais velozes e frequentes, além de trazer a quem desenvolve um ecossistema agnóstico de linguagem.



Como apontado por Sam Newman, em seu livro “Building Microservices”, estes são conceitos que estão implícitos em microserviços:

- São modelados levando-se em consideração o domínio do negócio;
- São altamente monitoráveis;
- Seu deploy pode ser feito de maneira independente dos outros serviços;
- Possui isolamento às falhas no ecossistema;
- Detalhes de implementação são “escondidos”;
- Automação é essencial em todos os níveis.

Com essas características, alcançamos uma arquitetura flexível e escalável.

TIP: Tenha em mente que as vantagens expostas não necessariamente nos levam a um mundo onde

as arquiteturas monolíticas não têm espaço. Na verdade, nós temos agora mais uma ferramenta em nossa caixa de ferramentas, uma forma diferente de entregar aplicações - que não necessariamente é a melhor forma para todos os cenários.

Vejamos algumas vantagens da utilização da abordagem de uma arquitetura orientada a microsserviços:

- **Escalabilidade vertical** - Devido à independência dos serviços, eles podem ser escalados conforme a necessidade, sem causar impactos nos demais serviços. Por serem menores, também requerem menos recursos de hardware.
- **Liberdade de selecionar tecnologias por projeto**: É possível escolher a melhor tecnologia para resolver determinado problema, mesmo que seja diferente das tecnologias utilizadas nos demais serviços.
- **Produtividade** - Esta abordagem suporta a existência de múltiplos times multidisciplinares (também conhecidos como squads) que podem atuar com um foco mais específico de negócio e, devido à independência técnica, entregar com maior velocidade. Em times grandes, a produtividade tende a tornar mais fáceis e menos problemáticas atividades como merge/rebase de um projeto se cada time trabalhar no seu próprio repositório. É muito importante pensar na organização e suas estruturas, que se refletem diretamente nos serviços e nas suas integrações. É o caso muito comum de que a estrutura do time impacta o modo como os softwares são organizados, muito bem explicado no Conway's law.
- **Agilidade** - Metodologias ágeis e suas diversas ramificações têm se mostrado cada vez mais populares. Ao se aliar esse estilo de gerenciamento de projeto com a arquitetura em microsserviços, permitimos tecnicamente a existência de ciclos completos e mais curtos para entrega de valor.

- **Reusabilidade** - Os componentes, como por exemplo um serviço que processe a lógica de negócio, podem ser consumidos via APIs, quando necessário, evitando assim a duplicidade de código e impactos na manutenibilidade.

Desafios da arquitetura de microsserviços

A comunicação entre componentes de uma aplicação monolítica ocorrem in-memory, ou seja, não possuem o overhead da **latência** existente na comunicação via rede, como ocorre no cenário de microsserviços. Quanto mais o número de serviços e a complexidade arquitetural aumentam, mais catastrófico pode ser esse problema. Lidar com o tempo de resposta do serviço invocado no(a) cliente em si é uma boa prática (como configuração de timeouts e retries), assim como manter em dia os serviços de monitoramento e alertas da sua rede.

Atente-se à separação por responsabilidade dos componentes de back-end e de front-end. O **front-end** deve ser implementado de forma que, na falha de um dos serviços de back-end, os demais itens funcionem normalmente - garantindo a melhor experiência possível.

A arquitetura de microsserviços é agnóstica a linguagens e frameworks. Com isso, seu ambiente pode se tornar **poliglota**. Do ponto de vista arquitetural e de liderança técnica, tenha parcimônia na avaliação das tecnologias a serem utilizadas para que não se depare com um cenário onde há um serviço sem profissionais capacitados para mantê-lo. A definição do escopo e tamanho de um microsserviço deve ser mensurada.

A definição do tamanho e escopo dos microsserviços pode ser uma tarefa que exija um pouco mais de esforço no início da jornada ao desacoplamento. Tenha em mente que o escopo e tamanho de um microsserviço devem ser mensurados a partir do princípio da responsabilidade única (**SOLID**).

Um dos desafios de governança é evitar a existência de aplicações órfãs em ambiente produtivo. Procure estabelecer times responsáveis por cada serviço, inclusive em sua fase produtiva. Desta forma, caso ocorra um problema inesperado ou uma nova solicitação de mudança, será mais fácil identificar quem poderá assumir as tarefas.

Cuidado com a granularização demasiada de projetos por repositórios, essa opção pode não ser a melhor quando existem mais repositórios que colaboradores(as) na empresa.

Migração de um monolito para microservices

Foram identificados padrões e técnicas que possibilitam a migração com sucesso de monolitos para arquitetura de microservices. Dois padrões bem populares para a realização dessa migração são [Strangler Fig Application \(Estrangulamento\)](#) e [UI Composition](#).

Caso você esteja em um cenário em que precisa migrar um monolito, é recomendado o estudo das estratégias presentes no livro [Monolith to Microservices, Sam Newman](#). Nesse livro, no capítulo 1, você também pode ver na prática um exemplo de migração de uma aplicação monolítica com o suporte do DDD.

Os erros mais comuns com microservices

É muito comum existir a lista dos erros que todo software tem, principalmente quando há uma mudança de paradigma. Por exemplo, quando houve a migração para os bancos de dados

NoSQL, certamente, o erro foi pensar em relacionamento em bancos de dados que não têm suporte a relacionamento, como [Cassandra](#). A lista a seguir apresenta os erros mais comuns que encontramos nos microservices:

- Quebra de domínio: o [DDD](#) trouxe vários benefícios, principalmente o de trazer o código para próximo do negócio com a linguagem ubíqua. Dentro do DDD temos o conceito de domínios, e quando movemos para microservices é muito normal quebrar de maneira errada o negócio no domínio. Esse tipo acontece, principalmente, quando fazemos a quebra de maneira precoce. É o mesmo caso da modelagem no banco de dados que fazemos, justamente, quando não temos muita informação do negócio. [Em seu artigo de definição de domínio, Martin Fowler](#) menciona o *bounded context*. Podemos ver isso num e-commerce, quando separamos o controle de estoque do produto; porém, o que acontece se a regra obrigar que o produto só possa ser exibido se tiver no estoque? Exato, toda vez que se consultar um produto, também precisará ser consultado o serviço de estoque, resultando num total acoplamento entre os dois serviços. Em outras palavras, o serviço de produto e o de estoque não deveriam estar em dois serviços nesse contexto.
- Não automatizar: uma das boas práticas existentes quando falamos de microservices, certamente, é o [CI/CD](#). Essas técnicas são realmente importantes, uma vez que existe uma grande quantidade de máquinas a serem gerenciadas.
- Diversidade de linguagens: essa decisão é uma das mais intrigantes. Até o momento, não conheço um único projeto cujo objetivo seja exibir alguma coisa no console, porém é muito comum ouvir de grandes nomes recomendações baseadas num “Hello World”. É importante ter muito cuidado com esse tipo de decisão; afinal, quanto maior o número

de linguagens dentro de uma empresa, mais e diversos campos o time terá que conhecer, ou existirão silos de conhecimento. Há diversas histórias que relatam que um sistema foi reescrito, simplesmente, por não ter um time técnico para manter ou porque a linguagem/framework foi descontinuada.

- **Sua aplicação não é grande suficiente para se tornar micro-service:** nem toda aplicação grande precisará ser migrada ou criada com o objetivo de se tornar um ambiente de microserviço. Um exemplo disso são as aplicações legadas e que atendem à necessidade de clientes.
- Um dos grandes argumentos para escolher microservices está na possibilidade de escolher escalar um componente individualmente. Porém, eis que surge a seguinte pergunta: realmente faz sentido escalar individualmente um componente?
- Microserviços precisam de informações e, como todo banco de dados distribuído, eles enfrentam a teoria do CAP. Dado um cenário no qual se realizam múltiplas atualizações em diversos serviços, é comum acrescentar um novo item na arquitetura: o padrão SAGA, resultando numa maior complexidade e pontos de testes no seu ambiente.
- **Começar o projeto já como microservices tende a ser um grande erro,** principalmente na instabilidade na definição dos domínios. Um erro na quebra dos serviços faz com que exista uma grande dependência e acoplamento entre eles. Considere um contexto em que os dados a serem utilizados estão armazenados em múltiplas bases de dados. Com pragmatismo, esse problema poderia ser facilmente resolvido em uma arquitetura monolítica, um join num banco de dados relacional, como MySQL ou PostgreSQL, ou um subdocumento num banco de dados NoSQL, como MongoDB.

- Utilizar microservices apenas porque grandes empresas utilizam esse tipo de arquitetura. No mundo de arquitetura de software, uma decisão não deve ser tomada apenas pela popularidade na solução. Como [Edson Yanaga](#) fala em seu [livro](#): “Certamente, sempre lemos grandes coisas sobre as arquiteturas de microservices implementadas por empresas como Netflix ou Amazon. Então, deixe-me fazer uma pergunta: quantas empresas no mundo podem ser Netflix e Amazon?”.

Conclusão

Como visto, a arquitetura de microsserviços traz muitos benefícios para o seu ambiente e lhe oferece a vantagem de deixar o desenvolvimento independente quando se tem vários times e funcionalidades, e essa independência se estende também para o deploy da aplicação. Ou seja, você dá velocidade e agilidade para seus times e consegue ter código de melhor qualidade, já que ele vai estar organizado ao redor da funcionalidade. Tem-se a vantagem de ser fácil de escalar apenas no ponto em que se precisa, e ainda poder ser aplicado na tecnologia que você tem mais domínio.

Mas, como já dito anteriormente, não é nenhuma bala de prata; há complexidades para o ambiente e novas preocupações em termos de segurança. Imagine um projeto gigante, com múltiplas instâncias e centenas de microsserviços; como você irá monitorar? Em caso de erro, como você vai encontrar, desviar ou mesmo tratar o erro?

Se usado da maneira correta, e tratados de perto os pontos de atenção, esse padrão de arquitetura tem muito a agregar nos seus projetos.

Microsserviços, como toda decisão de arquitetura, têm suas

vantagens e desvantagens. Como diz Martin Fowler:

Os microservices introduzem eventuais problemas de consistência, por causa de sua louvável insistência no gerenciamento de dados descentralizado. Com um monolito, podemos atualizar várias coisas juntas em uma única transação. Os microservices exigem vários recursos para atualizar, e as transações distribuídas são desaprovadas (por um bom motivo). Portanto, agora, os desenvolvedores precisam estar cientes dos problemas de consistência e descobrir como detectar quando as coisas estão fora de sincronia antes de fazer qualquer coisa de que o código se arrependa. - [Martin Fowler](#)

Cloud

Uma vez que conversamos sobre DDD, microservices, boas práticas, design de código e arquitetura de software, temos embasamento para prosseguir e abordar um dos temas mais discutidos do cenário de tecnologia: computação em nuvem (*cloud computing*).

Discorreremos sobre o que se considerar ao transpor arquitetura de aplicações para um ambiente de cloud, perspectivas populares a respeito de aplicações “cloud-native” e por que esse conceito é tão ligado a ferramentas como Kubernetes. Sêão também descritos padrões e funcionalidades esperadas em uma aplicação, para que sejam first-class citizens em um ambiente de cloud.

INFO: Este capítulo não tem como meta ensinar a fazer deploy de serviços em um cluster Kubernetes, configurar serviços na AWS ou criar aplicações cloud-native from scratch. A intenção deste capítulo é prover informações arquiteturais que embasarão suas decisões e modelagem dos seus serviços e plataformas. Com o conhecimento aqui fornecido, você estará preparado(a) e confiante para iniciar ou prosseguir sua jornada cloud-native, independentemente da solução ou linguagem adotada.

A buzz-word “cloud-native” começou a se estabelecer por volta de 2014, e sua crescente popularidade se mostra em seu melhor cenário. Para atingir maior espaço no mercado, empresas passaram a rotular seus produtos como cloud-native, quando, na verdade, são apenas tecnologias cloud-enabled.

TIP: Uma das formas de se avaliar a popularidade de um termo é validar a quantidade de buscas realizadas ao longo do tempo e as regiões interessadas.

<https://trends.google.com.br/trends/explore?date=all&q=cloud-native>

O entendimento desses conceitos te auxiliará no entendimento da situação em que suas aplicações estão e dará suporte ao planejamento de uma jornada para a cloud. Vamos discorrer sobre essas categorizações, seus conceitos e diferenças.

Cloud-Native ou Cloud-Enabled?

Uma aplicação pode ser categorizada conforme seu nível de adequação a um ambiente de cloud como sendo *cloud-enabled* ou *cloud-native* (a.k.a. *cloud-ready*).

Cloud-Enabled

É uma aplicação que foi containerizada e roda na cloud, mas que originalmente foi criada para rodar em ambiente tradicional, como por exemplo um data-center local, máquinas virtuais com cluster de servidores de aplicação tradicional. Essas aplicações podem ser categorizadas como *cloud-enabled* e têm maior consumo de recursos (cpu, memória, storage), em comparação a aplicações *cloud-native*.

Uma aplicação *cloud-enabled* passou por refatorações e ajustes para rodar em ambiente containerizado e também para suportar orquestração por plataformas como Kubernetes. Afinal de contas, não é mais o tradicional cluster de WildFly (a.k.a. JBoss EAP) ou GlassFish (Weblogic) clusterizados, que permitem que você

use a rede ou sistema de arquivos a seu bel-prazer. Agora, esses serviços rodam em pods, em contêineres efêmeros.

Apesar dos “contras” de se possuir uma aplicação cloud-enabled, o custo ou esforço de se refatorar toda a aplicação não são viáveis. Desta forma, a aplicação pode rodar em cloud, mas não pode usufruir de todos os benefícios existentes em um ambiente de cloud.

INFO: Kubernetes: É uma ferramenta open-source de orquestração de containers e trabalha muito bem com o [Docker](#). Atualmente, é a ferramenta mais popular na comunidade. Outros exemplos de ferramenta de orquestração de containers são Docker Swarm, Mesos e Amazon ECS.

Para entender melhor tudo o que uma aplicação cloud-enabled *não* é capaz de utilizar nativamente, vamos falar sobre o conceito cloud-native.

Perspectivas sobre o conceito cloud-native

No momento da escrita deste livro, não há um consenso ou definição exata acerca do termo. Portanto, vejamos posicionamentos:

“Cloud-native é uma abordagem para criar e executar aplicações que explora as vantagens do modelo de computação em nuvem. (...)"

—VMWare Tanzu (Pivotal)

“Cloud-native é uma maneira diferente de pensar e raciocinar sobre sistemas de software. Ele incorpora os seguintes conceitos: alimentado por infra-estrutura descartável, composta por limites, escala globalmente, adota a arquitetura descartável. (...)"

— Architecting Cloud Native Applications

“De maneira geral, “cloud-native” é uma abordagem para criar e executar aplicações que explora as vantagens do modelo de entrega de computação em nuvem. “Cloud-native” é sobre como as aplicações são criadas e implantadas, não onde. (...)"

— InfoWorld

“As tecnologias cloud-native capacitam as empresas a criar e executar aplicações escaláveis em ambientes modernos e dinâmicos, como públicos, privados e nuvens híbridas. Containers, service meshes, micro-services, infraestrutura imutável e APIs declarativas exemplificam essa abordagem. (...) “

— Cloud-Native Computing Foundation

‘Cloud native’ é um adjetivo que descreve as aplicações, arquiteturas, plataformas/infraestrutura e processos que, juntos, tornam o trabalho econômico de uma forma que nos permite melhorar nossa capacidade de responder rapidamente às mudanças e reduzir a imprevisibilidade

— Christian Posta

“Um conjunto de boas práticas para otimizar uma aplicação na nuvem por meio de: conteinerização, orquestração e automação.”

— Otávio Santana

Além dos conceitos acima, um conjunto de padrões bem recebido pela comunidade é o [12-factor](#). As excelentes práticas de arquitetura de software sugeridas nessa metodologia originam

do livro [Patterns of Enterprise Application Architecture](#), escrito por Martin Fowler e David Rice. Tendo em mente que o livro foi publicado em 2003, tempos em que ainda não se falava em “cloud-native”, podemos considerar que, ao adotar os conceitos do 12-factor, você estará não apenas criando uma aplicação cloud-native, mas também implementando boas práticas arquiteturais e culturais no desenvolvimento e entrega de software.

TIP: Nomes de projetos que serão detalhados a seguir podem evoluir e mudar com o tempo. Porém as definições e padrões esperados de uma aplicação cloud-native permanecerão. Portanto, recomendamos a leitura das referências citadas e aprofundamento no entendimento do conceito.

Capabilities for cloud

Veja alguns dos recursos que estão disponíveis e são comumente utilizados em suas aplicações que são concebidas para rodar na cloud:

- Gerência de configurações (Configuration Management)
- API Management
- Scheduling (of workloads)
- Distributed Tracing
- Service Security
- Centralized Metrics
- Auto Scaling / Self healing
- Service Discovery and Load Balancing
- Centralized Logging

Mais adiante discorreremos sobre um set de tecnologias disponíveis que implementam as features acima no ambiente. Neste momento, o que se deve ter em mente é que, para usufruir

dessas funcionalidades que existem no ecossistema, você precisa também ajustar um pouco seu código na aplicação. Caso queira ver na prática um excelente exemplo de como usufruir e implementar funcionalidades cloud-native, recomendo que você baixe e verifique o código da aplicação de exemplo gerada pelo site da especificação [Microprofile](#).

INFO: Por trazer um runtime mais leve, a especificação Microprofile abriu um leque de possibilidades para o Java no mundo da cloud. Existem diversas implementações, como Payara Micro, Open Liberty, Quarkus, Helidon e outros.

O Microprofile tem evoluído de forma rápida, e para garantir um conteúdo atualizado, optamos por não incluir todo o código de boas práticas, mas sim instruir você sobre como obter o conteúdo mais recente e de acordo com sua necessidade.

Para criar um projeto em que você pode estudar exemplos de implementação de práticas e funcionalidades cloud-native utilizando-se da especificação MicroProfile, execute os passos a seguir:

1. Acesse o site <https://start.microprofile.io/>
2. Insira um groupId, artifactId, selecione uma versão do MicroProfile, versão do Java SE, e o runtime. O runtime será a implementação da especificação MicroProfile.

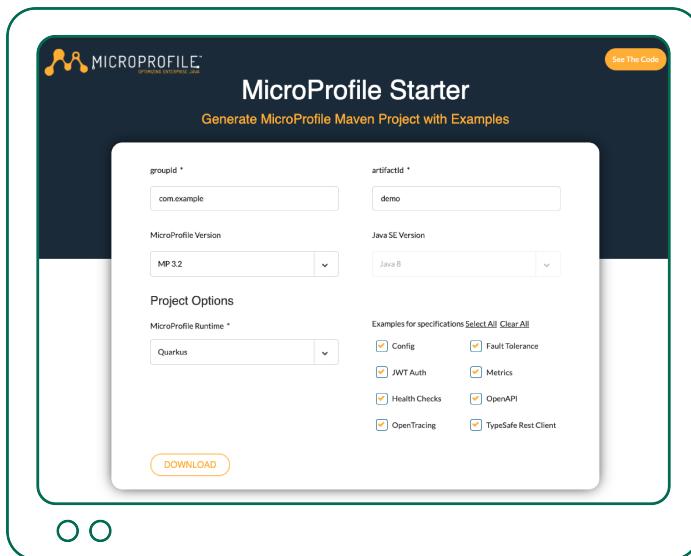


Imagen 08_01: Página de criação de aplicações MicroProfile - MicroProfile Starter

1. Clique em Download.

Será realizado o download de dois projetos em sua máquina. Você pode iniciar ambos, testá-los e analisar a simplicidade de se implementarem padrões cloud-native.

INFO: MicroProfile é uma especificação recomendada para a criação de microsserviços Java. Esteja atento ao fato de que entregar microsserviços **não** é sinônimo de entregar aplicações cloud-native.

Veja no código como são realizadas a implementação das apis de Health Checks com liveness e readiness (apis que serão consumidas pelo fato de o orquestrador de containers aumentar

a resiliência e suportar processos de deploy ao validar a saúde do pod), de métricas, tracing distribuído, resiliência a timeouts, segurança com JWT, injeção de propriedades de configuração através de anotações, e a utilização de RestClient (permite consumir um serviço rest apenas implementando-se uma interface no serviço cliente). A aplicação de exemplo acima é uma aplicação que inclui *várias*, mas não *todas* as features que iremos discutir.

Além da utilização da especificação MicroProfile para entrega de microserviços cloud-native, outra ferramenta amplamente utilizada é o [Spring Boot](#). É possível também construir serviços que usufruem de capabilities de aplicações cloud-native com a utilização de Spring e sua stack. [Spring Cloud](#) é um dos frameworks disponíveis que permitem entregar aplicações cloud-native e Java. Assim como demonstrado para o MicroProfile, também é possível criar aplicações de uma forma simples: <https://start.spring.io/>. Note que, ao clicar em Add Dependencies, você pode filtrar por cloud e escolher os componentes que deseja habilitar em sua aplicação:

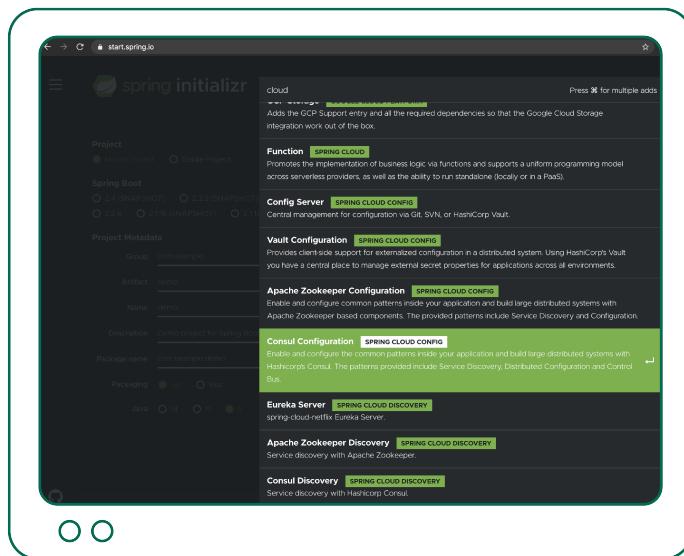


Imagen 08_02: Página de criação de aplicações Spring - Spring initializr.

Agora, uma vez que falamos sobre alguns detalhes de implementação da aplicação propriamente dita, vamos seguir em frente e entender melhores práticas de conteinerização dessas aplicações.

Princípios de design de conteinerização de aplicações

Em se tratando de detalhes de implementação de uma aplicação cloud-native, é consenso entre as fontes citadas que essas apps serão **conteinerizadas**. Com isso em mente, o(a) arquiteto(a) deve se preocupar não apenas com adoção do **SOLID** em seu código, mas também com os **princípios de design de conteinerização de aplicações**.

Princípios a serem considerados durante o tempo de **constru-**

ção de uma *imagem*:

- **Single Concern Principle:** Similar ao **S** do padrão **SOLID**, porém, neste contexto, cada container deve resolver *um* problema, e resolvê-lo *bem*; caso seja necessário acoplar mais features a um microservice, por exemplo, adicionar side-cars ao pod é uma boa alternativa.
- **Self-Containment Principle:** Devem estar contidas na imagem de build todas as bibliotecas, runtime da linguagem e ferramentas de construção necessárias para se realizar o build a aplicação. As exceções são dados que variam entre ambientes, dados estes que estarão, por exemplo, em variáveis de ambiente.
- **Image Immutability Principle:** Imagens imutáveis são essenciais para permitir escalabilidade e adoção de estratégias de deploy. Diferenças entre ambientes são providas ao container através de configurações (por exemplo, utilizando-se variáveis de ambiente ou ConfigMaps);

Princípios a serem considerados durante o tempo de **runtime** (execução, *container*):

- **High Observability Principle:** A aplicação conteinerizada deve prover as APIs padronizadas para que o orquestrador possa fazer health-checks de liveness e readiness. Coletas de logs e métricas também são parte das APIs que o orquestrador poderá consumir e disponibilizar através de ferramentas como, por exemplo, Fluentd, Logstash para logs centralizados, ou Prometheus e Grafana para métricas.
- **Lifecycle conformance principle:** É recomendado realizar graceful shutdown dos serviços sempre que possível, certo? Com containers, essa prática também é válida. O container precisa, através de APIs, permitir que o orquestrador envie comandos para graceful ou forceful shutdown.

E, caso necessário, pode até mesmo fazer uso de APIs como “pre-stop” e “post-stop” para implementação de necessidades específicas do componente pertencente ao container.

- **Process disposability principle:** Deve-se ter em mente que o container é volátil, e pode ser destruído e recriado várias vezes. Desta forma, considere o tempo de start-up e shutdown de seus containers. Além disso, caso seja necessário manter estado, deve-se recorrer a bancos de dados ou volumes providos pela plataforma.
- **Runtime confinement principle:** Ao seguir o princípio **Self-Containment Principle** em tempo de build, a imagem empregada para se gerar o container utilizado de fato para executar a aplicação terá uma menor necessidade de consumo de recursos (memória, espaço, cpu). O princípio de **Runtime Confinement** espera que o container tenha definidos os limites de recursos que serão utilizados. Com base nisso, o orquestrador poderá fazer uma melhor utilização da infraestrutura disponível.

Com base nesses princípios, é possível notar que não basta apenas criar um Dockerfile, conteinerizar uma aplicação e categorizá-la como “cloud-native”. No processo de conteinerização, é esperado que a aplicação implemente e disponibilize recursos que facilitarão seu gerenciamento, monitoramento e orquestração.

O ciclo de vida de uma aplicação cloud-native

A componentização dos serviços permite agora que os times de desenvolvimento entreguem mudanças com maior velocidade. Com isso, o time de operações precisa responder de forma equivalente, entregando serviços que permitam entregas velozes, porém estáveis. Para atingir o cenário ideal, é de comum

entendimento que a automação de processos de TI é essencial não só para o aumento da produtividade, mas também para o aceleramento da evolução da organização como um todo.

Nos cenários atuais, clientes esperam uma evolução constante e serviços altamente disponíveis e performáticos. Vejamos um cenário que considera essa necessidade e demonstra o ciclo de vida da aplicação, utilizando-se de melhores práticas de implantação.

Considerando que sua aplicação está pronta para deploy:

- O código fonte deve ser disponibilizado em um repositório de código fonte - *a single source of truth*.

TIP: Conforme [pesquisas](#) sobre os dados de 2020 (baseadas em códigos open-source), repositórios git são atualmente a escolha mais popular.

- Esse repositório, por sua vez, pode ser configurado de forma que ações-chave - como criação de uma nova tag, ou commit no branch master - disparem automaticamente uma ferramenta de integração contínua.
- A ferramenta iniciará a execução de testes unitários e o empacotamento dessa aplicação (podendo incluir mais passos, como, por exemplo, checagem da qualidade do código).

TIP: Tendo em mente uma melhor utilização de recursos, deve-se destacar que: a imagem-base utilizada para construir a aplicação *não* deve ser a mesma imagem-base utilizada para se executar a aplicação. Ferramentas para construção e empacotamento (como maven, npm etc.) que *não* são utilizadas na execução da aplicação *não*

devem estar incluídas na imagem-base de execução.

- Após empacotar a aplicação, a ferramenta de integração é responsável pela criação do container com base na imagem-base que inclua esse novo pacote (i.e. .jar).

TIP: É recomendado que a imagem-base gerada seja armazenada em um registro de imagens.

- Deve haver a preocupação de criar os arquivos de definição que serão usados para disponibilizar sua aplicação na cloud. Em se falando de Kubernetes por exemplo, deve-se criar os arquivos YAML, manter seu versionamento e processo de release;
- A partir dessa imagem, a plataforma utilizada (i.e. Kubernetes) irá criar a quantidade de containers especificada.

TIP: É muito comum a existência de casos de uso que podem usufruir da utilização de Operadores para gerenciamento e manutenção de aplicações.

Automação dos processos de entrega de software através da utilização de Entrega Contínua e Deploy Contínuo (CI/CD) é uma das práticas da cultura DevOps. Essa prática se mostrou eficaz em ambientes tradicionais e, agora, se mostra indispensável quando se trabalha com times menores e centenas de microserviços rodando em clouds privadas, públicas ou híbridas.

O processo acima descrito chega a ser simplista diante da qualidade e eficiência que podemos agregar ao nosso processo de integração. Vamos dar um passo atrás e entender melhor o que seria integração contínua.

Integração e Entrega Contínua

Começo este tópico com uma pergunta para você:

Você e sua organização estão prontos para colocar em produção uma nova versão do seu software a cada mudança?

Este é o estado da arte da integração e entrega contínua. Mas calma, mesmo que sua resposta seja não, a integração e entrega contínua ainda devem fazer parte da sua jornada cloud-native. **Integração** Contínua e **Entrega** Contínua são temas tão vastos que cada um possui seu próprio livro. Mas vamos discorrer sobre os principais tópicos a seguir.

Primeiro de tudo, a base: **automação de tarefas**. “Mas até onde devo automatizar?”, você me pergunta. Vamos lá:

Integração Contínua (CI)

Utilize uma ferramenta que permita a você automatizar o processo de integração da sua aplicação. A integração é contínua, ou seja, a cada commit no branch master a sua ferramenta de automação deve:

- Compilar e executar o build da aplicação;
- Executar testes unitários e obter o percentual da cobertura de testes;
- Executar testes para validar a qualidade do código;
- Enviar notificação ao time, caso qualquer um dos itens acima falhe, de acordo com seus critérios (cobertura de testes aceitável? Qualidade do código aceitável?);

Os passos mencionados devem ser facilmente executáveis, quer dizer, ao clique de um botão. Devem ser reproduzidos de ponta a ponta sem necessidade de intervenção humana.

Com as regras citadas, as pessoas desenvolvedoras no seu time terão sempre a boa prática de versionar código que roda e de testar o build localmente *antes* de se comitar no branch master. Se um commit quebra qualquer uma das regras, deve-se imediatamente corrigir o problema.

Essas práticas tornarão o seu processo de desenvolvimento e release mais confiáveis e estáveis.

Entrega Contínua (CD)

Uma vez que, com a utilização de CI, você agora gera pacotes mais confiáveis, de acordo com os critérios da sua organização, o que te impede de entregar versões novas com mais frequência em ambiente produtivo? Vamos falar sobre automação do processo de deploy.

Entrega Contínua != Deploy Contínuo

Com Entrega Contínua, você estaria pronto para liberar - através de um deployment pipeline - a versão recente mais estável do seu código a qualquer momento em produção. Novamente, com um clique de um botão.

E não podemos deixar de falar sobre **Deploy Contínuo**, que é quando você libera - através de um deployment pipeline - uma nova versão em produção a cada commit no branch master. Empresas grandes que praticam deploy contínuo chegam a liberar dezenas de centenas de versões em produção diariamente.

Automação é o segredo

Pare por um instante e identifique o nível de automação que você possui no momento. Quanto maior a maturidade das práticas de CI mencionadas, mais confiante você se sentirá ao liberar

novas releases do seu software. E quanto mais frequentemente você liberar software, mais rápido você terá feedback do(a) usuário(a) final, e o mais importante: maior a garantia de entregar software que funciona e que entrega exatamente o que clientes precisam.

Tenha em mente:

- CI é o primeiro passo;
- Para realizar Entrega Contínua, você precisa praticar CI.
- Para praticar Deploy Contínuo, você precisa estar apto e praticando Entrega Contínua.

Estratégias de deployment

Escolha dentre estratégias de deploy maduras que sejam mais apropriadas à sua aplicação para permitir entregas mais confiáveis. Ao escolher a estratégia de deploy para sua aplicação, escolha com base no quanto importante é:

- Tempo de indisponibilidade - o quanto crítico é para o negócio que sua aplicação fique fora do ar durante o deploy?
- Sua aplicação suportaria que você executasse duas versões (antiga e a nova) ao mesmo tempo?
- Você gostaria que um determinado grupo de usuários(as) fizesse testes na sua nova release, antes que você libere para 100% dos(as) usuários(as)?
- Você não possui um grupo de usuários(as) de testes, mas, mesmo assim, gostaria de avaliar o funcionamento da nova versão, liberando a release apenas para um percentual de usuários(as) finais?

Veja algumas estratégias de deploy que você pode usar de maneira fácil com plataformas de orquestração como Kubernetes:

- **Recreate Strategy:** todos os pods existentes serão escalados a zero, e só então o Kubernetes criará pods com a nova versão do seu código. Uma estratégia ousada (tudo ou nada), mas que pode ser necessária em casos de mudanças radicais em estruturas de dados, ou caso você não possa rodar as duas versões simultaneamente em produção;
- **Canary release:** é um tipo de Rolling Strategy, em que se libera a nova versão e, apenas quando constatado que a nova versão é saudável (de acordo com o readiness check do Kubernetes), o Kubernetes começará a destruir os pods com a versão antiga; neste cenário, os pods novos e antigos precisam coexistir durante o período de deploy;
- **Blue-Green:** é uma boa estratégia para se mitigarem falhas, porém é mais custoso. Utilize caso você queira que um grupo de pessoas realize testes para garantir que a nova versão está de fato pronta e pode ir ao ar. Devem existir dois ambientes de produção idênticos, o azul e o verde, mas apenas um estará ativo por vez. Você terá um router que irá direcionar os usuários para o ambiente ativo.

TIP: Leitura recomendada sobre blue-green deployment: artigo [BlueGreenDeployment, por Martin Fowler](#).

Digamos que o ambiente azul está ativo, rodando seu código v1. O grupo de usuários(as) realizará os testes no ambiente verde, não ativo, na versão v2. Uma vez confirmado que a nova versão, v2, pode ir ao ar, você vira a chave, e todas as pessoas usuárias passam a utilizar, agora, o ambiente verde. Seguindo a mesma linha, como o ambiente verde agora está em produção, em um próximo deploy você usaria o ambiente azul para garantir o release antes de virar a chave, e assim por diante.

- **A/B testing:** neste cenário você executa duas versões da aplicação em produção ao mesmo tempo, como uma

forma de se testar uma hipótese. Você pode, por exemplo, comparar, durante o período de uma semana, qual das duas versões vai performar melhor. Ou, de uma outra perspectiva, se o fato de adicionar um novo botão na tela da aplicação na nova versão leva os(as) usuários(as) a comprarem mais. Uma vez realizados os testes, pode-se liberar a versão desejada em sua totalidade, usando por exemplo canary release.

A jornada cloud-native

Entendidos os conceitos que giram em torno de uma aplicação cloud-native e após visualizar o ciclo de vida de uma aplicação cloud-native, o próximo passo é entender as formas que essas aplicações podem ser entregues e as opções disponíveis no mercado.

Há de se concordar que a transformação digital rumo a cloud acarreta não só uma mudança na forma de se codificarem aplicações (time de desenvolvedores), mas se estende a outras áreas da organização de T.I., que precisarão se reformular e lidar com novos desafios. Cada organização se encontra em um momento diferente, possui budgets diferentes e times com características e perfis variados.

Vamos discorrer sobre como *infraestrutura como serviço* (*Infrastructure As A Service - IaaS*), *plataforma como serviço* (*Platform As A Service - PaaS*) e *software como serviço* (*Software As A Service - SaaS*) coexistem, seus prós e contras e como cada uma delas pode auxiliar no momento atual de sua organização na jornada cloud-native.

IaaS, PaaS e SaaS: uma perspectiva arquitetural

A melhor maneira de se pensar no cloud em termos de abstração para o negócio certamente é na analogia de um serviço de pizzaria como serviço. Podemos partir do cenário em que temos a opção de preparar toda a pizza em casa e ter que gerenciar todo o processo de criação e cozimento, ou sair para comer a pizza em um restaurante sem se preocupar com sua criação.

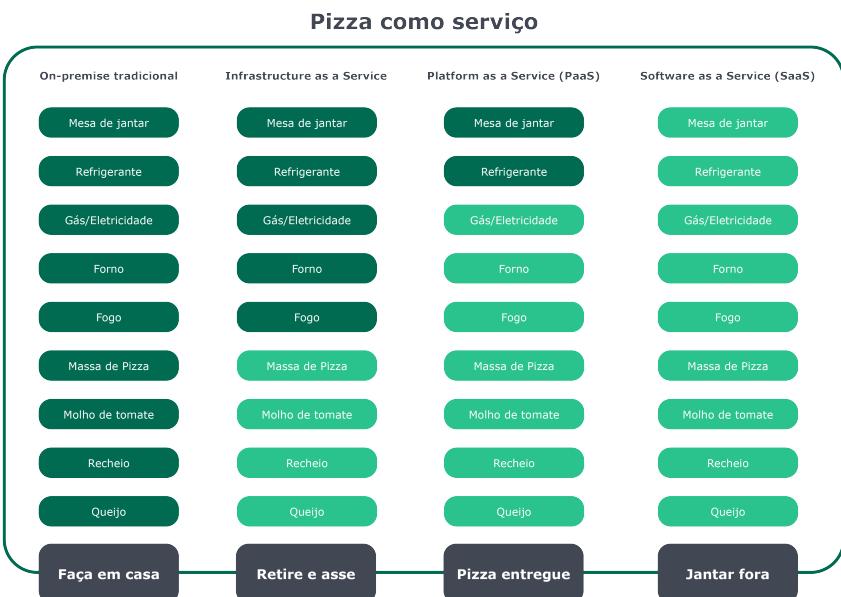


Imagen 08_03: Representação dos diferentes tipos de oferta de cloud através do exemplo “Pizza as a Service”.

Na imagem acima, temos dispostos quatro formatos: on-premise, IaaS, PaaS e SaaS. As caixinhas brancas são as tarefas de

nossa responsabilidade, e as azuis são de responsabilidade de terceiros. Essa mesma comparação pode ser feita se trocarmos as tarefas de criação de pizza por tarefas referentes a criação de software: instalação e gerenciamento de sistema operacional, networking, storage, orquestração de serviços, gerenciamento de middleware, runtime, pipelines de CI/CD, e até a criação da aplicação propriamente dita.

Quando falamos de cloud e seus serviços, note que quanto menor a abstração que utilizamos como serviço, por exemplo, com IaaS, maior a responsabilidade no processo de construção do software. Esse grande número de processos crescerá na mesma medida que o número de servidores aumenta, e essa complexidade é diretamente proporcional aos riscos. Em contrapartida, quanto menor a abstração, maior o controle da possibilidade de customização da arquitetura e componentes.

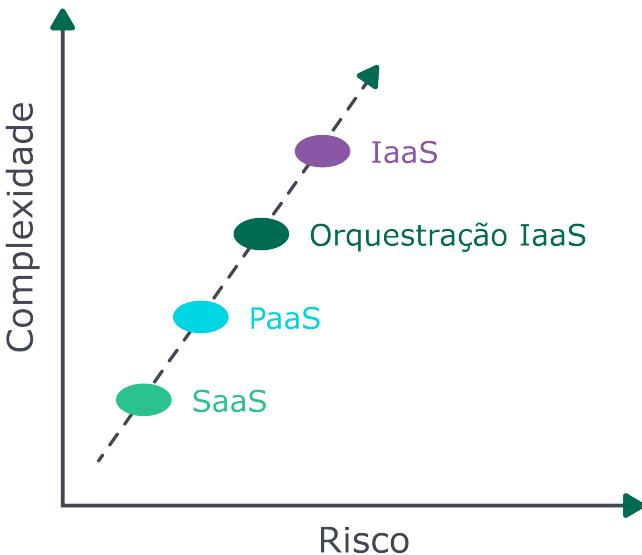


Imagen 08_03: Matriz Complexidade vs Risco dos diferentes tipos de oferta de cloud.

IaaS - Infra as a Service

Com esta abordagem, é possível obter todos os benefícios de um ambiente de cloud, porém toda a responsabilidade de manutenção do software é da sua equipe de T.I. A empresa que provê o serviço de IaaS tem o dever de garantir a comunicação entre os serviços, de lidar com quedas, problemas de hardware e eventuais consertos.

Neste cenário, a sua equipe assume tarefas referentes a banco de dados, backup, escalonamento tanto vertical quanto horizontal etc. Esse fator aumenta a possibilidade de customização e, por outra perspectiva, gera complexidade e mais risco. Nesta opção o hardware e a garantia de seu funcionamento pertencem

a terceiros, mas o serviço é executado pelo seu time; desta forma, pode ter um custo mais reduzido, se comparado a outras opções.

O conceito de *orquestração* no contexto de IaaS é a configuração, gerenciamento e coordenação automatizada dos serviços, aplicações e sistemas de computador. Essa orquestração permite a manutenção da infraestrutura através de programação (infra as code) e tem como objetivo facilitar e abstrair a utilização da IaaS.

Exemplos de IaaS são Microsoft Azure, Google Compute Engine e [Amazon EC2](#). De uma maneira geral, pode-se pensar nela como sendo um grande aluguel de máquinas por cujo uso se pode pagar, semelhante à nossa conta de luz. Para ilustrar isso, vamos analisar a realização do deploy de uma aplicação conteinerizada no ambiente da Amazon. A [primeira tarefa](#) é a criação de uma instância, atividade que requer cerca de sete passos. Em seguida, a [instalação do Docker](#), para que assim seja possível criar [uma imagem para executar essa instância](#). Neste caso, nota-se que um simples deploy de uma aplicação requer não apenas know-how extra como a execução de tarefas de sysadmin por parte de seu time.

PaaS - Platform as a Service

Uma redução drástica de complexidade para focar a criação do software, certamente, é a maior vantagem dentro do PaaS. Com ele, no geral, não é necessário se preocupar com manutenção das máquinas, criação de rotina de backup, compra de licenças etc. Todos os cuidados do [desenvolvimento serão na criação do software](#). Porém o PaaS costuma apresentar custo mais elevado, se comparado com serviços sem maior abstração.

TIP: Veja abaixo algumas opções de PaaS atualmente disponíveis no mercado:

- **Platform.sh:** é um PaaS que utiliza todos os conceitos de infraestrutura como serviço e também é orientado ao Git, além de possibilitar realizar o deploy da aplicação, deixando todo trabalho para a plataforma. Basicamente, a partir de três arquivos **aplicação, serviços e rotas**, podemos fazer o push para um repositório Git. Um simples push para o sistema remoto do Platform.sh criará automaticamente os containers da aplicação, dos serviços como banco de dados e as rotas da aplicação. Nesse caso, a abstração é gigantesca e faz com que o time foque muito mais a aplicação central da empresa.
- **Red Hat OpenShift Online:** uma opção para se utilizar o OpenShift (a.k.a. **OKD**) como serviço. Neste PaaS, o OpenShift é disponibilizado na AWS e permite a desenvolvedores de aplicações Ruby, PHP, Node.js e Java utilizarem seus runtimes e banco de dados de preferência para rodar suas aplicações. Possui uma opção self-service e free para pessoas desenvolvedoras.
- **Heroku:** nascido em 2007 exclusivamente para Ruby, hoje o Heroku suporta as mais diversas linguagens, como Go, Java, PHP, Node.JS e outras, além de várias ferramentas do ecossistema de uma aplicação;
- **Google Kubernetes Engine (GKE):** o PaaS oferecido pelo Google é totalmente baseado em Kubernetes Vanilla. Simples de se iniciar, ao se cadastrar você ganha uma quantidade de créditos para poder rodar seus workloads, e recebe também acesso a outros produtos da Google.

Com base em leituras e conceito aplicação de PaaS por diversas empresas, podemos analisar por duas perspectivas:

1. Você delega o gerenciamento de toda a plataforma (leia-se Kubernetes e similares) para terceiros. Desta forma, a sua equipe não precisa instalar, lidar com manutenções, backups e monitoring. Essa perspectiva é apenas viável quando o caso de uso pode usufruir de uma cloud pública. Heroku e os vários flavors de Kubernetes, como Platform.sh, GKE e OpenShift Online, são exemplos de solução de PaaS que se adequam a essa perspectiva.
2. No segundo cenário, devido a restrições na sua empresa, você necessariamente deve utilizar uma cloud privada. Nesse caso, você pode prover um PaaS ao disponibilizar uma ferramenta que aumente a autonomia da pessoa desenvolvedora ao prover uma forma eficiente e automatizada de entregar aplicações conteinerizadas, seja através do uso de catálogo de serviços, seja por abstração de apis de entrega de containers ou automação de deploy. Dessa perspectiva, é válido considerar que ferramentas como, por exemplo, OpenShift, instalado em uma cloud-privada e gerenciada pelo seu time de infraestrutura.

Em ambos os cenários, é comum assumir que há maior autonomia da pessoa desenvolvedora ao se utilizar PaaS.

SaaS - Software as a Service

O software como serviço é a oferta que provê uma solução mais rápida para determinado problema. Nessa oferta, clientes optam por consumir um programa pronto para uso e não precisam se preocupar com hospedagem, escalabilidade etc., e nem mesmo com desenvolvimento. Toda a complexidade e o risco já foram resolvidos. No entanto, a customização é bem reduzida, e a possibilidade de configuração depende diretamente do provedor.

Conclusão sobre IaaS, Paas e SaaS

De uma maneira geral, temos que pensar nos seguintes três princípios cíclicos, que compararam:

- Quanto maior a complexidade, menor a abstração;
- Quanto maior a abstração, menor o risco;
- Quanto menor o risco, menor a complexidade.

É verdade que existem vários benefícios na navegação nos mares da computação em nuvem, porém é muito importante conhecer os tipos de serviços que a cloud disponibiliza e fazer uma análise profunda da complexidade de um serviço e seu respectivo risco, além do tempo que o time está disposto a gastar para criar e manter toda a infraestrutura. O PaaS fornece uma grande vantagem de abstração de toda a infraestrutura e manutenção para que a pessoa desenvolvedora foque seu negócio. O IaaS te garantirá grande flexibilidade e poder para instalar e configurar o que seu time deseja sem nenhum problema, mesmo que tudo tenha que ser configurado manualmente. É muito importante que tanto o time quanto a empresa tenham noção de que, independentemente da escolha, haverá benefícios e desvantagens. É muito importante para que os(as) arquitetos(as) avaliem o que melhor se encaixa na instituição, afinal, cloud e computação não estão relacionados ao quando, mas ao como.

Kubernetes - quando usar e quando não usar

Como decidir quando é a hora de migrar seu workload para uma arquitetura conteinerizada e orquestrada por plataformas como Kubernetes? Entenda mudanças culturais e operacionais para

tomar uma decisão mais embasada. Os pontos abaixo descritos são válidos para plataformas como Kubernetes e seus vários flavors.

- Instalação e manutenção da plataforma de orquestração: em caso de adoção de IaaS, o time de sysadmins deve estar preparado para realizar e gerenciar a instalação de um ou mais clusters de Kubernetes. Deve estar antenado em melhores práticas de gerenciamento e configuração de clusters Kubernetes.
- Utilização da plataforma Kubernetes: em caso de IaaS ou PaaS:
 - Da perspectiva de pessoas desenvolvedoras, a containerização da aplicação garante que o código que roda em sua máquina funcionará também nos ambientes produtivos. Basta garantir que a imagem seja corretamente descrita. Além disso, uma vez absorvidos os conceitos de criação de uma aplicação cloud-native e as vantagens de um ambiente de orquestração, o(a) desenvolvedor(a) tem a vantagem da perspectiva da carreira, de estar apto(a) a trabalhar em diferentes vendors, uma vez que há certa padronização na forma de trabalho. Por outro lado, a gestão deve estar ciente da curva de aprendizado inicial que a equipe enfrentará.

TIP: É recomendada a leitura do livro [Kubernetes patterns for designing cloud-native apps](#). Traz boas práticas na implementação de serviços cloud-native.

- Da perspectiva do time de middleware ou sysadmins, o dia a dia passa a ser diferente. Não é mais necessário despender tempo aprendendo como entregar diferentes tipos de aplicação e desvendando suas respectivas peculiaridades e dependências. Basta aprender

a lidar com containers, e a forma de trabalho passa a ser padrão, independentemente da tecnologia utilizada nas aplicações. Por outro lado, deve-se entender conceitos de operators, segregação e segurança no ambiente Kubernetes, estratégias de deployment (rolling, blue-green, canary), como lidar com storages etc.

Kubernetes Vanilla e seus sabores

Esse projeto se popularizou por demonstrar que é capaz de lidar com orquestração de containers em larga escala. É possível identificar diversas ofertas baseadas em Kubernetes: Heptio, OpenShift, Platform.sh, Rancher, entre outras. A intenção é que cada empresa por trás dessas plataformas possa agregar valor com features únicas, porém respeitando todos os conceitos e arquiteturas existentes em sua plataforma de origem, Kubernetes.

As diferentes plataformas trazem consigo, de maneira pré-selecionada, porém não definitiva, um conjunto de tecnologias que irão atender aos requisitos de uma plataforma de orquestração. Para entender melhor essa afirmação, analise a [CNCF landscape](#) e note a quantidade de tecnologias que endereçam, por exemplo, a categoria de Service Mesh. Uma das opções disponíveis é o [Istio](#), solução disponibilizada de forma suportada em plataformas como o OpenShift, o que não significa que a mesma tecnologia é exclusiva dessa plataforma. O que se torna diferenciado entre plataformas são as distribuições utilizadas. Ainda tendo o Service Mesh como exemplo, o projeto de Istio entregue pelo OpenShift é o [Maistra](#), que é um projeto que adiciona mais features on top of Istio.

Nesse contexto, ao selecionar sua próxima plataforma de orquestração, considere:

- As features exclusivas de cada plataforma e como elas podem tornar sua equipe mais eficiente;
- O quanto é importante para sua empresa o nível de estabilidade do serviço (em casos de ofertas de PaaS em clouds públicas ou híbridas);
- Como você e sua equipe irão lidar com problemas: por si sós, ou precisam de suporte enterprise?
- Se vai rodar em cloud pública, valide se a plataforma oferece suporte ou tem cases de sucesso na devida opção (Amazon, Google, Azure etc.).
- Avalie as ferramentas de CI/CD suportadas pela plataforma;
- Avalie as ferramentas que seu(sua) desenvolvedor(a) terá que utilizar no dia a dia para interagir com a ferramenta de orquestração: existe apenas CLI? Há um console que permite manutenção e monitoramento do ambiente?
- Caso seja parte do seu cenário uma migração parcial e de médio/longo prazo de um ambiente on-premise para uma cloud pública, avalie as possibilidades de um ambiente híbrido onde você possa usufruir de ambas as clouds, privada e pública, sem maiores problemas com a plataforma em questão.

Como escolher seu set de tecnologias

Existem diversas opções de implementação de cada uma destas funcionalidades, e você pode ver as diversas opções recomendadas para um cenário de cloud na [CNCF Cloud Native Interactive Landscape](#).

INFO: Pensando em organizações que estão na jornada para a cloud nasceu a Cloud Native Computing Foundation (CNCF). A CNCF nasceu com o objetivo de definir o termo Cloud Native e fornecer espaço

para projetos open source, como Kubernetes, Prometheus e [muito mais](#). A CNCF adota projetos e os encaixa na arquitetura de cloud computing, segundo a visão dos membros do comitê.

A CNCF assume projetos e os classifica como **graduados**, **incubando** ou em **sandbox**. Essa classificação deriva da maturidade de cada software, de acordo com sua utilização no mercado:

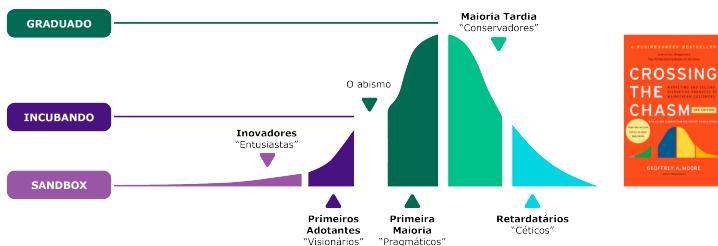


Imagen 08_04: O que é o “Chasm” e os níveis de adoção de tecnologia.

Imagen obtida em <https://www.cncf.io/projects/>

Se tiver a possibilidade, acesse a [CNCF Cloud Native Interactive Landscape](#) e repare as inúmeras tecnologias disponíveis para solucionar os problemas de um ambiente de nuvem. Note que, apesar de ser hoje a mais popular, Kubernetes é apenas *uma* dentre as opções de orquestração de contêineres. Não podemos deixar de ressaltar algumas tecnologias populares:

- Diversas distribuições de Kubernetes, como [Red Hat OpenShift](#) e [Rancher](#);
- Ofertas de PaaS, como [Heroku](#) e [platform.sh](#);
- [Jaeger](#) para tracing de aplicações, e [Prometheus](#) e [Grafana](#) para monitoramento;
- [Strimzi](#) e [Apache Spark](#) para streaming e mensageria;
- [Istio](#) para service mesh, [Envoy](#) para Service proxy;

- [Jenkins](#), [JenkinsX](#) e [Tekton](#) para CI/CD;
- [Helm](#), [Operator Framework](#) e [Podman](#) para construção de imagens e definição de aplicações.

Ainda no contexto de ferramentas e funcionalidades de que aplicações cloud-native podem usufruir, vamos falar a seguir sobre service mesh e as tecnologias existentes.

Bonus Topic: Service Mesh

Vamos à definição de service mesh (malha de serviços) por [William Morgan, 2017](#):

Uma malha de serviço é uma camada de infraestrutura dedicada a lidar com a comunicação serviço a serviço. É responsável pela entrega confiável de solicitações por meio da topologia complexa de serviços que inclui um aplicativo nativo da nuvem moderno. Na prática, a malha de serviço é normalmente implementada como uma matriz de proxies de rede leves que são implantados juntamente com o código do aplicativo, sem que o aplicativo precise estar ciente.

Estamos falando de um desacoplamento entre o “Dev” e o “Ops”, ou melhor, a pessoa desenvolvedora não precisa se preocupar se o código do seu microserviço pode prover as capacidades de que a operação precisa. E toda vez que a operação precisar alterar algo, não é necessária uma nova compilação da aplicação, gerando um desacoplamento que prove a resiliência, a segurança, toda a parte de observability e reteamento para a malha e fora da aplicação.

Se avaliarmos que o conceito começou a ganhar popularidade em 2017, é um conceito um tanto quanto novo, mas a ideia é

criar uma camada de abstração por cima da aplicação. E, através da malha, prover a segurança entre os serviços, e também delegar para a malha toda a observação e resiliência de que ela possa precisar.

Uma camada de abstração pode existir em três níveis:

- **Biblioteca** - Cada serviço implementa uma biblioteca que inclui as capacidades da malha; bibliotecas como [Hystrix](#) ou [Ribbon](#) são exemplos. Essas implementações possuem trade-offs, pois, neste caso, você precisa fornecer as bibliotecas e, com isso, limitar as tecnologias que se pode trabalhar. Se seu projeto tem múltiplas linguagens, você também precisará de múltiplas implementações, e precisará gerenciar esse cenário.
- **Node Agent** - Existe um agente rodando em cada nó da malha e com a responsabilidade de cuidar das capacidades da malha. A implementação do Linkerd no Kubernetes usa esse modelo. O trade-off aqui é que o time de Ops precisa trabalhar em conjunto com o time de Dev para que as capacidades estejam bem estabelecidas.
- **Sidecar** - Atualmente, a implementação mais sugerida é o modelo usado pelo Istio com Envoy. Nesse modelo, para cada container de aplicação existirá um container ao lado - no mesmo pod. Esse sidecar lida com tudo que a malha precisa para chegar ou sair do serviço.

Mas o que são essas capacidades da Malha que são mencionadas? A malha de serviços deve seguir as regras de ORASTAR sem ter códigos implementados no seu microsserviço:

- **Observability** - O Painel de Controle provê todos os serviços de observação que estão rodando no plano de dados. Ou seja, métricas para ver a latência, consumo de banda, logging e tracing, tudo que precisa para monitorar a saúde

dos seus serviços ficam na malha, provendo visualizações gráficas de todas as requisições.

- **Routing** - As regras de roteamento para o controle do tráfego pode ser feito tanto visualmente quanto por arquivos de configuração e, então, enviadas do controle para todas as aplicações. Estamos falando de mudança e divisão de tráfego, controle sobre o que entra e sai e capacidade de injetar falhas e latências para fazer testes.
- **Automatic Scaling** - O painel de controle tem que ser capaz de escalar automaticamente para lidar com os aumentos ou diminuições do workload.
- **Separation of duties** - Usar o painel de controle para separar a operação da malha e dar mais independência para as pessoas desenvolvedoras.
- **Trust** - Delegue para a malha e o plano de dados os protocolos de comunicação de segurança, bem como a renovação e manutenção de certificados. E a regra é sempre a mesma: considere que se está sempre inseguro, mesmo abaixo de um firewall.
- **Automatic service registration and discovery** - O painel de controle interage com o gerenciamento do cluster para provisionar a descoberta dos serviços automaticamente, assim como registrar a aplicação durante o procedimento de deploy.
- **Resilient** - As regras de resiliência para toda a sua malha: é necessário assumir que a rede pode falhar. Então é preciso ter a capacidade de blindar o tráfego e, automaticamente, balancear para outros pontos que estejam funcionando e que possam prover a mesma capacidade. Aqui entra o conceito de Circuit Breaker.

Arquitetura

E, para implementar as regras de ORASTAR, temos a arquitetura de um painel de controle e um plano de dados. Vamos entender esses conceitos.

Painel de Controle é onde ficam as configurações, políticas e a administração dos serviços que estão no plano de dados para o controle de rotas, tráfego, monitoração, descoberta e registro dos serviços. É aqui que está a responsabilidade pela comunicação entre os microserviços, através de autenticação, autorização e segurança no tráfego da rede.

Plano de dados é onde estão todos os microserviços e seus “sidecars”, não importando como eles estão implementados. É com essa camada que o painel de controle interage para conseguir controlar os serviços.

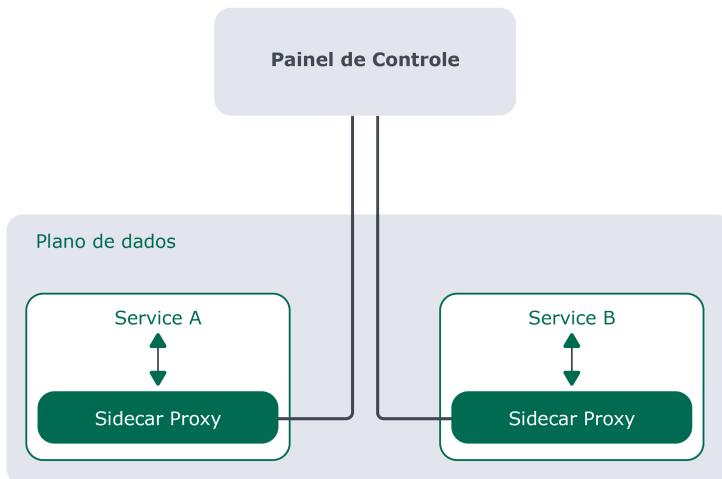


Imagen 08_05: Represetação de um Data Plane

Ferramentas

Service Mesh é um conceito que surgiu por volta de 2018 e está em constante evolução. Em termos de implementação, basicamente vamos falar de três ferramentas: Istio, Linkerd e Consul.

Istio - Ferramenta criada pela Google, IBM e Lyft. Hoje o [Istio](#) é

a implementação mais recomendada do mercado, e construída em cima da plataforma do Kubernetes.

Ele é um painel de controle centralizado que permite que se administre e coordene a aplicação no plano de dados - inclusive pode-se rodar o core dele fora do kubernetes -, tem suporte à integração com VMs e service discovery com diferentes produtos.

Linkerd - iniciado pelo Twitter para lidar com seu volume massivo. Hoje está na versão [Linkerd 2.0](#). Possui a limitação de rodar apenas ambientes com Kubernetes. Também possui um painel de controle centralizado como o Istio, e usa seu próprio Sidecar, em vez do Envoy como proxy, como os concorrentes.

Consul - criado pela Hashicorp. O [Consul](#) é mais velho que o próprio service mesh, sendo que sua primeira versão foi de 2014. Ele é composto de um binário em GO para o servidor e aplicações para prover as capacidades da malha. Mas, diferentemente das ferramentas acima, ele tem um painel de controle distribuído, pois é feito para funcionar próximo às máquinas.

TIP: Para optar por um dos serviços, veja a comparação de funcionalidades entre os serviços encontrada no artigo [Kubernetes Service Mesh: A Comparison of Istio, Linkerd and Consul](#).

Do ponto de vista do autor, uma das desvantagens do Linkerd é não possuir Circuit Breaker nativo, fazendo com que você tenha que implementar o Circuit Breaker no código de seu serviço. Em contrapartida, ele é o mais fácil de operar, dentre as outras opções.

Não podemos deixar de fora os provedores de serviços na nuvem pública que também têm oferecido service mesh, como a AWS e a Microsoft:

AWS App Mesh - Se o seu ambiente está na AWS, vale a pena dar

uma olhada neste serviço que usa o Envoy como sidecar proxy: [AWS App Mesh](#);

Azure Service Fabric Mesh - Apesar de usar o nome Service Mesh, este é o que mais difere dos demais. Está mais comparado a um Red Hat OpenShift e se faz necessário que você tenha um plano de dados de malha de serviço já em uso. É um serviço gerenciado, e as pessoas desenvolvedoras não têm acesso ao painel de controle: [Azure Service Fabric](#).

Quando não usar Kubernetes

Com base na explicação de aplicações cloud-native, note que, para entregar arquiteturas cloud-native, você não precisa necessariamente utilizar Kubernetes. Principalmente se você está no início da jornada para a cloud, Kubernetes (e seus vários flavors) podem não ser a solução de todas as questões na jornada para a cloud, e pior, talvez até te traga mais pontos de atenção.

Se seu cenário ainda possui um baixo número de imagens e containers, talvez ainda não seja a hora de adotar uma arquitetura mais robusta como Kubernetes. Além do aumento do consumo de recursos, seus times de Dev e Ops precisarão estar aptos a imergir nesse novo paradigma de entrega de aplicações. Talvez seja apropriado iniciar com uma ferramenta como [Docker Swarm](#). O Docker Swarm está disponível junto ao Docker engine que você instala, e sua utilização se dá através do próprio comando Docker:

```
1 $ docker swarm
2
3 Usage:      docker swarm COMMAND
4
5 Manage Swarm
6
7 Commands:
8   ca          Display and rotate the root CA
9   init        Initialize a swarm
10  join        Join a swarm as a node and/or manager
11  join-token Manage join tokens
12  leave       Leave the swarm
13  unlock     Unlock swarm
14  unlock-key Manage the unlock key
15  update     Update the swarm
```

Você pode criar as máquinas que farão parte do cluster, instalar o Docker e configurar o Swarm e a camada rede. Uma vez configurado seu Swarm, você pode escalar as réplicas de pod e gerenciar em uma escala menor, se comparado ao Kubernetes, os seus containers.

Com a introdução do conceito de containers e orquestração de containers ao seu time, naturalmente haverá uma evolução para a automação dos processos de construção e entrega dessas imagens e containers. Com isso, o time terá a chance de praticar e absorver o conhecimento necessário para imergir mais naturalmente em um cenário mais complexo de orquestradores robustos.

Deve-se também ter em mente o cenário em que você já utilizou o Kubernetes e já tem um conhecimento mais aprofundado da ferramenta. Com isso, você já tem ideias de como melhorar ou facilitar os fluxos internos de orquestração, práticas de desenvolvimento, de entrega contínua. Tem ideias de UI que podem auxiliar o(a) usuário(a) ou práticas que podem acelerar a entrega

de aplicações. Com base nisso, você pode, ao invés de usar o Kubernetes, partir para um dos flavors disponíveis, e até mesmo criar o seu próprio flavor!

Conclusão

Não existem dúvidas de que o futuro da tecnologia reside na cloud. Com o surgimento dos microsserviços, cada vez mais precisamos do conhecimento do ecossistema que gira ao seu redor. Com o aumento do uso de containers e práticas de DevOps, cada vez mais as pessoas desenvolvedoras precisam conhecer sobre plataforma, e o system admin precisa conhecer sobre desenvolvimento.

Se ainda não fazem parte da sua realidade, aplicações cloud-ready e cloud-native com certeza o farão em um futuro muito próximo. Junto a elas, todo o vasto ecossistema de containers, orquestração de containers e demais funcionalidades que giram ao redor desses serviços serão necessários. Esteja pronto para aderir ao movimento.

Precisamos falar sobre atualizações

Não vou mentir pra você: quando ouvi falar que o Java seria atualizado a cada seis meses, pensei comigo: “xiiii... esse negócio não vai dar certo”.

Oras, e por quê? Por alguns motivos (seis, para ser mais exato):

1. Se olharmos a plataforma Java apenas do ponto de vista de código fonte, já veremos que se trata de um projeto absolutamente gigantesco, que vem sendo evoluído há 25 anos e que, durante esse tempo, passou por mudanças da própria computação em si (vide os casos de cloud e containers, só para citar dois exemplos rápidos);
2. Milhões de linhas de código, milhares e milhares de classes, um “buzilhão” de coisas que dependem de outras tantas;
3. Cada feature deriva de uma JSR, que não raro está debaixo de um projeto “guarda-chuva”. Em cada uma dessas partes, muitas vezes, há pessoas diferentes trabalhando, e ritmos diferentes de evolução dos projetos;
4. Tudo o que acontece dentro da plataforma é regulado pelo JCP, que não é famoso pela sua velocidade;
5. O ecossistema de ferramentas e frameworks em torno do Java é um dos maiores do mundo (se não for o maior). Alguns open source, outros proprietários. E todos totalmente dependentes do ritmo de evolução da plataforma;
6. E, finalmente, os(as) usuários(as), que normalmente são desenvolvedores(as) que trabalham em empresas. Sem dúvida, são as pessoas mais impactadas por qualquer

questão referente a ciclo de atualizações, já que são a ponta dessa cadeia de interesse.

Eu poderia acrescentar mais itens a essa lista, mas creio que ela já é suficiente para justificar o meu ceticismo inicial.

Mas hoje cá estamos nós e... não é que eles conseguiram? Sim, hoje temos novidades no Java a cada seis meses! E não apenas funcionou, como está indo bem. Bem demais!

Nas próximas linhas vamos conversar sobre implicações e desafios específicos que esse novo ciclo ajuda a resolver e que também serviram de inspiração para que ele fosse adotado.

Com que frequência você entrega alguma coisa para seu (sua) usuário (a)?

Se você já leu/ouviu/estudou alguma coisa sobre DevOps, talvez já tenha passado por algo parecido com essa pergunta.

Só para entendermos rapidamente o princípio aqui envolvido: suponha que você entrega uma versão/atualização de algum software para o(a) usuário(a) final uma vez ao ano.

Ou seja, você e sua equipe trabalharam um ano inteiro para criar uma versão nova. Codificaram muito, mexeram no banco, alteraram configurações, atualizaram telas e tantas outras coisas que podem ser feitas de acordo com cada tipo de aplicação.

Agora vamos olhar para as implicações desse cenário:

1. Você descarregou um ano de trabalho no colo de seu(ua) usuário(a). Claro que você pode tornar essa experiência o menos dolorosa possível. Mas ainda assim vai doer;

2. Você publicou uma massa imensa de alterações. Se ocorrer um bug, a sua superfície de busca/debug é gigantesca;
3. Você tinha alterações/melhorias simples, pequenas, que já estavam prontas desde o primeiro mês, mas que tiveram que esperar “a grande release” para chegarem até seu(usa) usuário(a).

E, vamos lembrar, estamos falando de um ano. Imagine se você fizer isso a cada dois anos? Ou a cada três anos?

Bom, era mais ou menos isso que estava acontecendo com o Java. Normalmente levava-se de dois a três anos para uma nova versão. Aliás, chegamos a incríveis (quase) cinco anos entre a versão 1.6 e a 1.7.

Agora vamos pensar na situação oposta, em que você entrega com uma frequência cada vez maior. O que acontece?

1. Você entrega menos coisas para seu(usa) usuário(a) de uma vez só. É mais fácil ele(a) absorver as mudanças. É mais fácil fazer seu treinamento. Ele(a) vai ter uma maior percepção daquela pequena melhoria fantástica que se perderia no meio de uma release maior;
2. Se um bug ocorrer, você tem muito menos lugares para procurar. E, como tem menos código entregue, a chance de ter erros também é menor. Tem gente que até prefere nunca entregar nada, assim nunca dá erro...;
3. As melhorias chegarão às mãos de seus(suas) usuários(as) em um tempo muito menor. Você vai deixá-los(as) mais feliz com uma frequência maior. Ponto pra você!

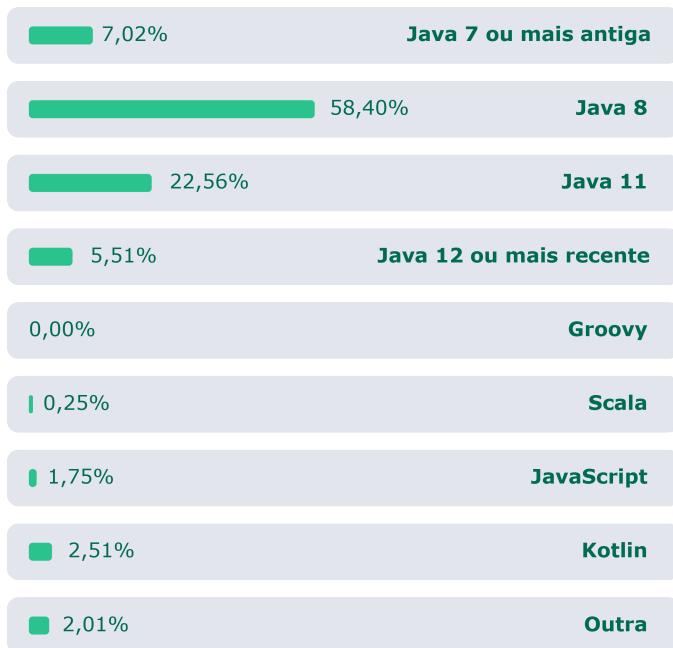
Foi exatamente o que ocorreu com a plataforma Java. Eles passaram de um período aproximado de três anos entre as releases para outro de seis meses. É uma frequência seis vezes maior.

Só para te dar uma ideia, o JDK 9 teve noventa e um itens em sua release, enquanto o JDK 14 teve apenas dezesseis. Olhe para os três itens que mencionei acima e veja-os materializados na linguagem mais utilizada no mundo.

Os problemas de não atualizar a versão de JVM

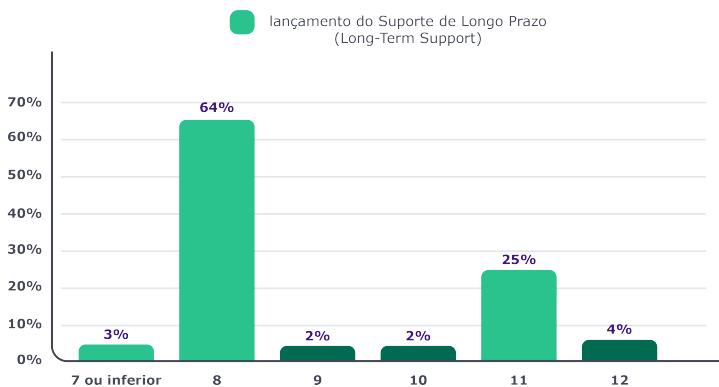
A empresa JRebel publicou no ano de 2020 os resultados de uma pesquisa realizada com centenas de profissionais ao redor do mundo. Veja abaixo um dos resultados específicos em relação às versões de Java:

Qual linguagem de programação Java você está usando na sua aplicação principal?



Fonte: <https://www.jrebel.com/blog/2020-java-technology-report>

Outra pesquisa interessante também, publicada em 2020 pela Snyk, trouxe os seguintes resultados:



Fonte: <https://snyk.io/blog/developers-dont-want-to-leave-java-8-as-64-hold-firm-on-their-preferred-release/>

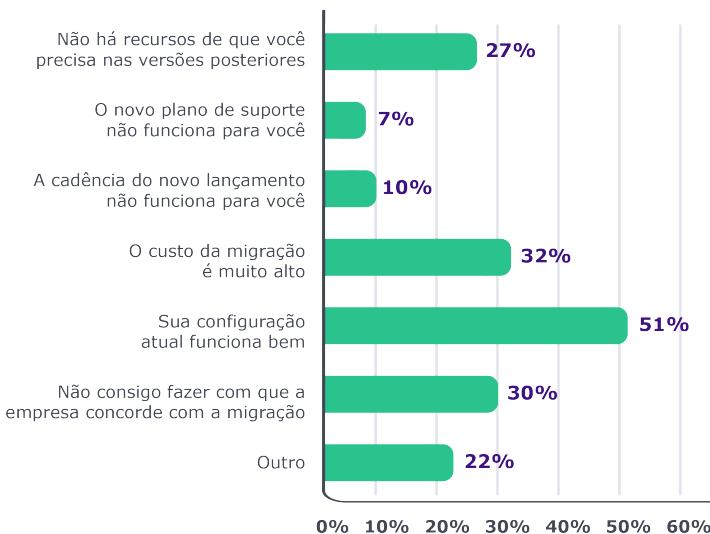
Tirando apenas a média dessas duas pesquisas, já temos cerca de 60% dos desenvolvedores Java ainda na versão 8. E há pesquisas por aí que dizem que esse número pode ser ainda maior, chegando a algo em torno de 80%.

Considerando que o Java 8 foi lançado em 2014 e que já temos uma outra LTS (Java 11) desde 2018, deveríamos fazer no mínimo duas perguntas, para as quais já elenco as devidas respostas.

Por que o mercado ainda está no JDK 8?

Vamos encarar a realidade: o mercado, principalmente quando se fala em aplicações enterprise, não anda na mesma velocidade que todos nós gostaríamos.

E por que isso? Eu diria que os principais motivos estão claramente listados na mesma pesquisa da Snyk já mencionada acima:



Fonte: <https://snyk.io/blog/developers-dont-want-to-leave-java-8-as-64-hold-firm-on-their-preferred-release/>

Esse resultado, aliás, não deveria surpreender ninguém. Os principais motivos que fazem profissionais e empresas ao redor do mundo não terem essa ânsia pelas últimas atualizações são:

1. A versão atual funciona (o famoso “em time que está ganhando não se mexe”);
2. O custo de migração é muito alto;
3. A área de negócios não concorda com a migração.

Eu diria que o item 3 é devido ao item 1. Ou seja, a área de negócios não concorda com a migração exatamente porque a versão atual funciona.

Precisamos ter em mente uma realidade de forma clara: projetos em ambiente corporativo são resultado dos acordos feitos a partir de diferentes interesses.

Atualização de versão (por qualquer motivo que seja) significa risco de erros e indisponibilidade. A área de negócios não quer isso, assim como a área de operações.

No caso do item 2 (custo de migração elevado), ele também é perfeitamente justificável. Afinal, sempre que pensar em mudar a versão de uma JVM, você terá que:

1. Fazer o build do código atual na versão nova (muitos já desistem aqui);
2. Fazer todos os ajustes de código necessários para funcionar na versão nova;
3. Fazer todos os testes necessários para garantir que nenhum erro foi adicionado na aplicação pela pura troca de versão de JVM (e, nessas horas, testes unitários serão seu melhor amigo);
4. Atualizar as ferramentas do seu ecossistema que dependiam da versão anterior;
5. Garantir que a nova versão do Java estará instalada e configurada em todos os ambientes em que sua aplicação rodar (container ou VMs);
6. Se chegar até aqui, provavelmente já terá boa segurança pra migrar.

E, no caso da migração do Java 8 para versões mais atuais, temos ainda um “pequeno” detalhe pelo caminho: o Java 9.

Sim, Java tem retrocompatibilidade. Sim, um código feito em Java 1.2 deveria funcionar no Java 14. Mas também é verdade que o Java 9 introduziu mudanças brutais na plataforma que, sim, quebraram muito projeto por aí.

Em linhas bem gerais, a modularização lançada no JDK 9 (projeto Jigsaw) trouxe não apenas a possibilidade de você modularizar a sua aplicação internamente, mas a própria plataforma em si foi modularizada. Então a forma como o seu projeto lida

com a plataforma Java mudou, e o resultado disso para muitos projetos acabou inviabilizando a saída destes da versão 8.

Aliás, existe um artigo fantástico da Trisha Gee falando sobre como migrar seu projeto para o JDK 9 (caso ainda esteja em versões anteriores). Além de útil para a migração em si (claro!), ele ainda vai te dar uma ideia de alguns problemas que podem ocorrer durante o processo. Está aqui: <https://www.infoq.com/br/articles/Java-Jigsaw-Migration-Guide/>

Por todos esses motivos, é de certa forma compreensível que o mercado ainda esteja em sua maior parte “preso” ao Java 8.

Aliás, de certa forma é até irônico ver que muita gente reclamava que o Java demorava para receber novas atualizações, e agora boa parte dessas pessoas não consegue acompanhar o ritmo de inovações da plataforma.

Porém, se por um lado é compreensível que se pense duzentas vezes antes de migrar para uma nova JVM, é também verdade que ficar parado no tempo tem suas implicações técnicas.

Quais os impactos de não atualizar para as últimas versões?

Há motivos para que se lancem novas versões de qualquer software: corrigir erros, melhorar funcionalidades, criar outras novas, ou mesmo introduzir tendências que surgem no mercado de tecnologia.

Com a plataforma Java não é diferente. Ela não é atualizada, evoluída e modernizada “por esporte”. Logo, se há motivos técnicos que justificam sua atualização, os(as) bons(as) arquitetos(as), desenvolvedores(as) e engenheiros(as) do mercado devem prestar atenção a isso.

Vamos imaginar que você é um(a) profissional que trabalha em uma aplicação Java que está usando o JDK 8. Vamos supor que,

nesse nosso ambiente imaginário, não há interesse em atualizar o Java para as versões mais recentes (mesmo que seja a versão 11, que é a versão LTS - Long Term Support - mais recente em 2020). Vamos olhar alguns poucos e importantes itens que você está perdendo:

- Com a modularização do JDK 9, surgiu também o jlink, que permite que você, de certa forma, gere o seu próprio JRE. Ou seja, você consegue gerar a sua aplicação utilizando estritamente as dependências necessárias. Em um mundo cada vez mais ligado aos containers, é possível gerar imagens até 70% menores utilizando jlink;
- Falando em containers, até o JDK 8, a plataforma Java não tinha sido concebida para lidar com restrições de memória do processo onde a JVM está rodando (que, no caso dos containers, derivou do cgroups - algo que já existe em sistemas UNIX há décadas). Ou seja, um container rodando até a versão do JDK 8 poderia alocar memória indefinidamente, até que ocupasse todos os recursos disponibilizados para o daemon, o que exigiu que muitos fizessem workarounds (= gambiarras) para evitar maiores problemas. Desde o JDK 9 (vide este artigo <https://www.infoq.com/br/news/2017/03/java-memory-limit-container/>), a cada nova release temos alguma melhoria com relação à gestão de consumo de recursos da plataforma. Hoje podemos dizer que Java é extremamente performático, eficiente e viável para uso com containers;
- A partir do JDK 10, o compilador JIT do Graal VM está disponível em qualquer distribuição de Java. Ele é em média 13% mais rápido que o JIT padrão, e tem evoluído rapidamente;
- No JDK 9, o G1 passou a ser o Garbage Collector default da plataforma, o que traz ganhos substanciais de performance se você simplesmente mudar a JVM (sem tocar no

seu código);

- Falando em Garbage Collector, você poderia (quem sabe?) estar avaliando o uso do Shenandoah, um Garbage Collector chamado de “low-pause-time”, ou seja, que faz a sua coleta praticamente sem gerar overhead na sua JVM.

Poderíamos escrever outro livro só listando coisas que as aplicações rodando em uma JVM na versão 8 estão “perdendo”. E veja que nem foram mencionadas aqui questões de segurança, que é algo crítico em qualquer ambiente corporativo.

É claro, tudo é uma questão de balancear interesses e necessidades. Mas, sem dúvida, planejar se manter atualizado ao menos com as versões LTS é algo que deveria estar nos seus planos e nos planos da sua organização.

Utilidade versus Hype

Há um tempo atrás eu estive em uma grande empresa brasileira para falarmos sobre seu desejo de migrar para uma arquitetura de microservices. Após conversarmos algumas amenidades, fiz a pergunta de um milhão de deploys: por que vocês querem usar microservices?

E a resposta, claro, não poderia ser diferente: *porque o nosso chefe viu num evento e mandou a gente fazer.*

Isso acontece milhares e milhares de vezes ao redor do mundo, diariamente. Empresas e profissionais adotam tecnologias, padrões, abordagens e linguagens sem ao menos saber do que se trata, para que serve, que problemas aquilo resolve e se ao menos elas têm esses problemas. Apenas porque alguém disse que é legal, é novo, é batuta.

Não seja esse(a) profissional! Cada vez que você faz isso, um pod morre em algum cluster de Kubernetes por aí. Ajude a salvar os pods...

O que um(a) bom(a) arquiteto(a) deveria fazer, então?

Ter equilíbrio. Como em tudo na vida.

Cada dia tem uma nova bala de prata por aí, algo que vai resolver

todos os problemas, que você pode utilizar em todos os projetos. É a fórmula do fracasso.

E daí vem o outro extremo: há quem não preste atenção em novidade alguma, não experimenta nada, não faz uma POC sequer para ver a aplicação de alguma nova tendência. Tudo em nome de evitar a hype. Mais gente matando pods por aí.

De novo, equilíbrio.

A hype em si não é ruim. Tudo o que está consolidado hoje foi hype um dia. O primeiro JavaOne teve mais de seis mil participantes e foi realizado apenas um ano após o lançamento da primeira versão. Era hype!

O segredo é usar a hype a seu favor. Tirar proveito dela.

Como? Veja, tenho algumas sugestões:

1. Quando algo está na moda, tem muita gente falando. Tem muito material sendo criado. Muita palestra sendo dada. Todo esse material de referência é um tesouro de valor inestimável para quem quer saber de fato o que está acontecendo;
2. Quando tem muita gente tentando algo novo, tem muita gente enfrentando erros, dificuldades, vendo coisas que não funcionam. É aqui que profissionais inteligentes mais tiram proveito: aprendendo com os erros dos outros;
3. Se houver muitos relatos de sucesso sobre uma determinada hype, preste atenção aos cenários onde esses sucessos ocorreram. São parecidos com o seu? Resolvem algum problema que você tem? Se não se aplicam hoje, podem se aplicar a algo que você estava planejando para o futuro;
4. Junte tudo isso (material coletado, casos de fracasso e casos de sucesso) e faça sua própria análise. Analise também junto com colegas de trabalho. Façam simulações e provas de conceito. Discutam, criem conteúdo sobre o assunto.

Se você seguir esses passos, dificilmente será iludido pela próxima hype que surgir. E se ela se provar mais do que uma nova “modinha”, quem sabe você não sairá na frente como um case de sucesso?

Destrinchando performance de aplicações

Introdução e conceitos

Muitas empresas utilizam a abordagem “deixe o desempenho para depois” para tratar com possíveis problemas de performance que possam ocorrer no ambiente de produção para a fase de testes. Na fase de testes, é provável que não se leve em consideração a carga que a aplicação possa sofrer e o tempo de resposta esperado, pois isso não foi especificado nos requisitos não funcionais - requisitos como o tempo de resposta de uma determinada operação ou a quantidade de acessos simultâneos. Além disso, temos o desafio de conciliar uma arquitetura evolutiva juntamente com a engenharia de desempenho ou performance. Na arquitetura evolutiva, temos que ser ágeis e permitir constantes melhorias na arquitetura, deixando as decisões para última hora, enquanto a engenharia de desempenho se preocupa em identificar possíveis gargalos e definir as tecnologias utilizadas já no início do projeto.

Antes de irmos mais a fundo, vamos rever alguns conceitos importantes sobre desempenho de aplicações.

Throughput: É a vazão ou taxa de transferência que sua aplicação consegue dar às requisições. Número de requisições por segundo/minuto ou hora. Quanto maior esse número, melhor. Geralmente, nas ferramentas de teste de carga, o Throughput

é medido da seguinte forma: Throughput = (número de requisições) / (tempo total).

Latência: É o tempo que leva para um pacote de dados ir de um ponto a outro. Ao contrário do Throughput, quanto maior a latência, pior a performance. Em sistemas distribuídos, este pode ser um grande problema.

Sistemas de alta performance têm como requisitos alta confiabilidade, baixa latência e alta taxa de vazão. Isso parece comum para quem vai desenvolver um serviço, porém não é uma tarefa das mais fáceis manter e melhorar a performance à medida que o sistema vai sendo utilizado, porque novas funcionalidades são adicionadas, a base de dados cresce.

Agilidade versus Performance

Se a performance é a prioridade no projeto, considere verificar as dependências que está utilizando ou que irá utilizar. Ao utilizar uma biblioteca para substituir alguma parte de código, podemos ter a sensação de que estamos produzindo menos código, código limpo. Quanto menos código, menos bugs, certo? Nem sempre essa afirmação é correta. Você pode estar utilizando uma biblioteca que possui bugs também. Muitas pessoas desenvolvedoras acabam não acompanhando a evolução da linguagem e ficam “amarradas” a certas bibliotecas que utilizavam em versões passadas e que, na nova versão da linguagem, não se fazem mais necessárias. Bibliotecas podem trazer agilidade no desenvolvimento, mas será que elas degradam a performance ou deixam sua aplicação mais pesada? Avalie os prós e os contras de cada dependência que irá utilizar no projeto. Entenda o seu funcionamento e busque pelo histórico de bugs e vulnerabilidades.

Verifique o que está carregando na bagagem

Em um projeto Java utilizando o Maven, por exemplo, podemos verificar a árvore de dependências utilizando o comando `mvn dependency:tree`, e ainda podemos filtrar somente por dependências de compilação, ex: `mvn dependency:tree -Dscope=compile`.

```
1 [INFO] --- maven-dependency-plugin:2.8:tree (default-cli\
2 ) @ testcontainers ---
3 [INFO] com.testcontainers:jar:1.0-SNAPSHOT
4 [INFO] \- mysql:mysql-connector-java:jar:5.1.47:compile`
```

É muito comum desenvolvemos serviços que são empacotados em um arquivo jar conhecido como *fatjar*, em que todas as dependências são colocadas dentro do jar. Em muitos casos você não precisará daquela dependência para rodar o seu serviço. Ela pode ser uma dependência de teste, ou estar provida por um servidor de aplicações. Então é sempre uma boa ideia verificar as dependências do seu projeto.

Cuidado com a performance também quer dizer, em muitos casos, implementar mais código em vez de utilizar alguma biblioteca que facilita o trabalho e acelera o desenvolvimento, o que pode lhe custar a performance lá na frente.

Como medir a performance?

Existem muitas formas de medir a performance da aplicação, seja com um monitoramento em tempo real, seja com um teste de ‘stress’ antes de liberar uma funcionalidade em produção. Tudo vai depender do requisito não funcional solicitado. É claro que todo(a) usuário(a) quer ter sempre a resposta o mais rápido

possível. Mas qual o limite aceitável? Essa é a medida que se deve ter em mente até mesmo antes de iniciar a codificação. Por exemplo: o tempo de login não pode ser superior a um segundo. Como posso medir isso? Lembrando que não é simplesmente colocar que o ‘login precisa ser feito em menos de um segundo’. Deve-se avaliar em quais circunstâncias esse login pode ou não demorar mais. Até onde o sistema de login pode escalar. Uma boa diretriz de média seria dizer: o tempo de resposta do login é de 1 segundo para 500 solicitações simultâneas, com uma carga de CPU de 60% e uma utilização de memória de 80%.

Capturando o tempo da requisição

Neste capítulo, vamos utilizar a ferramenta [jMeter](#), muito utilizada para criar diversos tipos de teste de carga e medir o desempenho. O objetivo aqui não é ser um tutorial do jMeter, mas mostrar como é possível gerar e visualizar dados através dele. Abaixo, um exemplo simples de medição de tempo de login considerando 10 usuários(as):

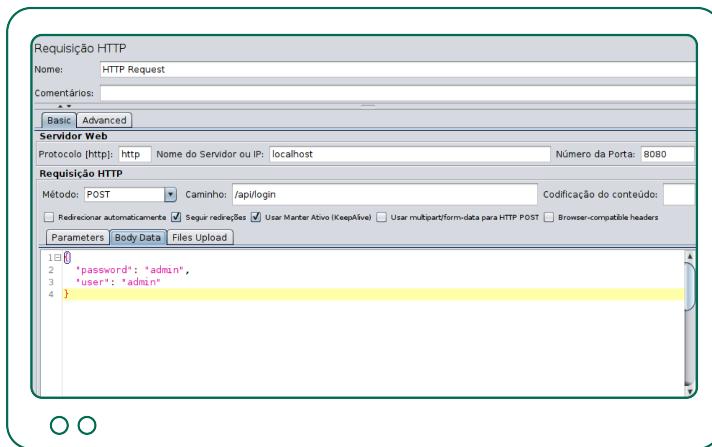


Imagen 10_01: Teste com JMeter que realiza a invocação de API de login, enviando usuário e senha via POST.

Ao rodar o teste acima, verificamos no Relatório de Sumário os resultados:

The screenshot shows the 'Relatório de Sumário' (Summary Report) dialog box. It has fields for 'Nome:' (Name) set to 'Summary Report' and 'Comentários:' (Comments). Below these are buttons for 'Escrever resultados para arquivo / Ler a partir do arquivo' (Write results to file / Read from file), 'Nome do arquivo' (File name), 'Procurar...' (Search...), and checkboxes for 'Apenas Logar/Exibir' (Only Log/Display), 'Erros' (Errors), and 'Sucessos' (Successes). A 'Configurar...' (Configure...) button is also present. The main area displays a table of results:

Rótulo	# Amostras	Média	Mín.	Máx.	Desvio Pa...	% de Erro	Vazão	KB/s	Sent KB/sec	Média de...
HTTP Req..	10	591	575	618	17.59	0.00%	6.6/sec	3.33	1.54	516.0
TOTAL	10	591	575	618	17.59	0.00%	6.6/sec	3.33	1.54	516.0

Imagen 10_02: Resultado do teste de invocação de API de login.

Alguns dados importantes neste momento:

Amostras: Número de requisições realizadas.

Média: Número médio de tempo de todas as requisições.

Min: Menor tempo dentre todas as requisições.

Max: Maior tempo dentre todas as requisições.

Ainda podemos ter gráficos mais ricos, utilizando o plugin [PerfMon](#), por exemplo:



Imagen 10_03: Utilização de plugin para análise de resultados com gráficos.

No gráfico acima, podemos ver que a maioria das requisições ficou entre 600 e 700 milissegundos em um cenário de testes com 1000 requisições. Podemos ter gráficos ainda mais bonitos e em tempo real, podendo utilizar o [grafana](#) como visualizador de gráficos.

Veja que capturamos o tempo total de um processo de login, porém se o login não está em um tempo adequado ou queremos melhorar ainda mais o tempo, precisamos visualizar cada componente em separado. Compreender como um valor é calculado e o que isso significa é essencial para tirar as conclusões corretas. Para esse fim, devemos examinar os métodos estatísticos usados para calcular e agregar dados de desempenho. Nunca utilize somente o valor da **média** para tirar conclusões sobre per-

formance, pois durante um período de 24 horas com milhares de requisições, os valores de pico serão ocultados pela média.

Entendendo e separando os componentes

Medimos o tempo total de um login e precisamos melhorar o tempo de resposta. Para isso, precisamos testar separadamente cada componente da arquitetura para descobrir onde podemos diminuir o tempo. É possível que, com apenas uma ferramenta, não se possa medir a performance da sua aplicação. É provável que você utilize uma ferramenta de carga para estressar a aplicação e várias outras para coletar os dados. Como exemplo, podemos ter uma aplicação que tem uma api para o login com acesso ao banco de dados. No entanto, podemos ter cenários bem mais complexos. A imagem abaixo é uma representação da arquitetura para servir milhões de usuários(as):

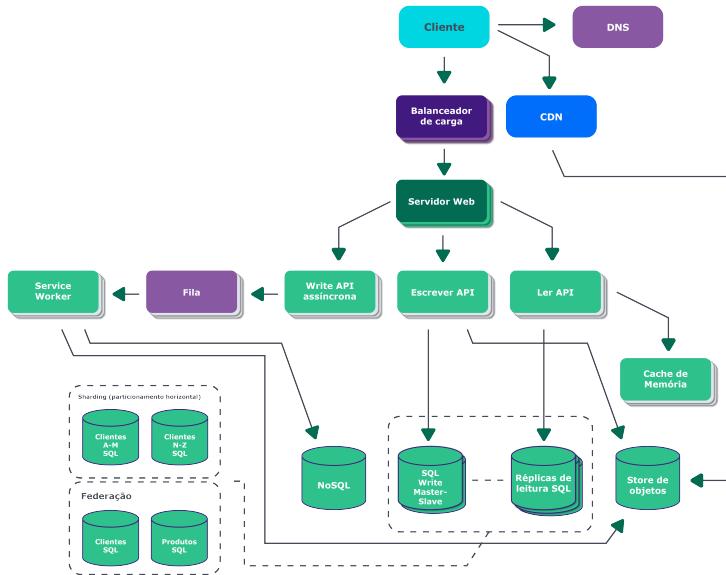


Imagen 10_04: Arquitetura de uma aplicação complexa com capacidade para atender um alto número de requisições.

Créditos: https://github.com/donnemartin/system-design-primer/blob/master/solutions/system_design/scaling_aws/

No entanto, não foi de primeira que esta arquitetura foi definida. Foram muitos experimentos, testes e medições para chegar a uma arquitetura escalável. Deve ser possível medir a performance da réplica de leitura do banco de dados MySQL separadamente, por exemplo.

Monitorando a performance por componente

Conforme demonstrado acima, não é de primeira que se define uma arquitetura para milhões de usuários(as). É necessário estressar e medir para verificar onde estão os pontos que podem sofrer carga. Existem várias opções que mostram onde estão os gargalos da aplicação. Uma das várias opções é o [JavaMelody](#), que pode ser utilizado em modo standalone junto com a sua aplicação Java, é free e muito simples de colocar na aplicação.

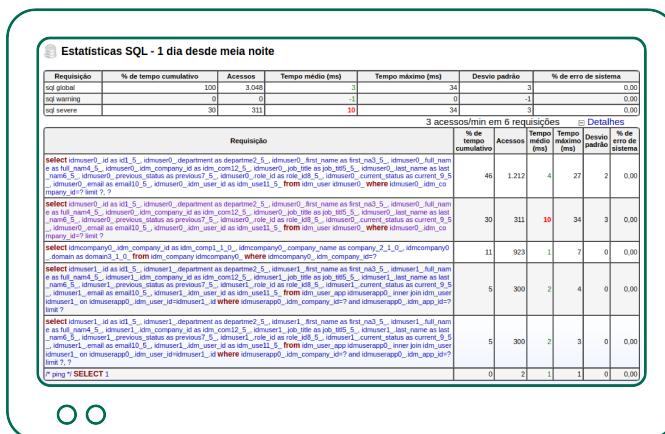
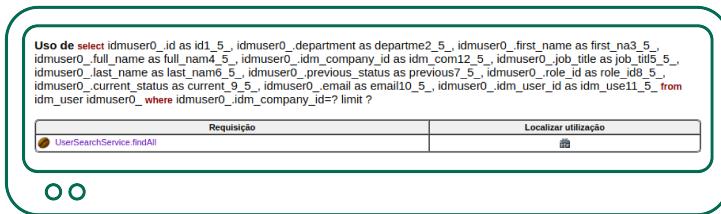


Imagen 10_05: Exibição de métricas para queries SQL executadas, através do JavaMelody

Na imagem acima, podemos notar que uma das consultas SQL demorou, em média, mais que o normal em relação a outras. Podemos descobrir de onde veio esse comando SQL, como também executar o comando SQL em modo ‘Explain’, a fim de revelar que a query está fazendo um ‘full scan’ e que será preciso ajustar a query ou criar índices específicos na tabela.



No outro exemplo abaixo, podemos ver um desvio bem grande no método ‘findById’, que, por sua vez, não utiliza um banco de dados MySQL, mas sim uma outra fonte de dados externo. Com essas informações em mãos, já é possível analisar de modo isolado cada comportamento.

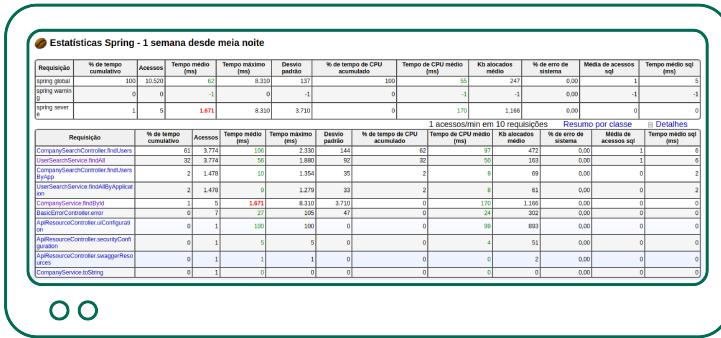


Imagen 10_07: Repare o longo tempo de execução do método CompanyService.findById que impacta diretamente no tempo de resposta da requisição ao servidor Spring.

Existem muitas ferramentas de monitoramento, e o que fica aqui como exemplo é que, em alguns casos, você vai precisar ir no detalhe e fazer algum ajuste fino na infraestrutura, ou até mesmo no código.

Monitorando a performance em sistemas distribuídos

Sistemas distribuídos, atualmente mais populares com a utilização de microsserviços, são complexos e difíceis de monitorar a performance. Nesse caso, vamos precisar de mecanismos mais sofisticados, como a utilização de um ‘tracing’ distribuído por exemplo. Aqui também existem várias soluções, como os famosos APMs, tais como o New Relic, AppDynamics, DataDog e Dynatrace. Vale lembrar que muitos provedores de cloud fornecem ferramentas de análise de performance, tal como o AWS Performance Insights. No mundo OpenSource, vale destacar a ferramenta [Jaeger Tracing](#), cuja especialidade é fazer o monitoramento de serviços distribuídos rodando em uma infraestrutura do Kubernetes, por exemplo.

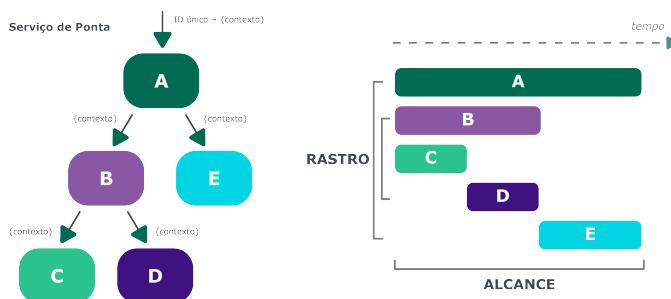


Imagen 10_08: Arquitetura de serviços distribuídos e a relação entre rastro e alcance nestes serviços.

Podemos observar em qual dos serviços o tempo de resposta não está adequado e tomar as devidas ações.

Mapeamento Objeto Relacional

No mundo Java, utilizamos o JPA para trabalhar com a camada de persistência em bancos de dados relacionais. Neste tópico, abordaremos algumas boas práticas para obtermos uma boa performance na utilização desse padrão.

Estratégia de geração de IDs

O gerador de identificador TABLE é muito ineficiente. Ele é mais genérico e portável para a maioria dos bancos de dados, porém requer uma transação de banco de dados separada, como também uma conexão separada para garantir que o processo de geração do identificador não esteja vinculado à transação que iniciou. Emprega o uso de bloqueios no nível de linha que são pesados em comparação com as usadas pelas estratégias de geração de identificador como IDENTITY ou SEQUENCE. Portanto, se o banco de dados suportar sequences, é muito mais eficiente usar a estratégia SEQUENCE.

Enums

Geralmente mapeamos **Enums** como String para facilitar a leitura no banco de dados ou exibir diretamente o literal do enum.

```
1 @Enumerated(EnumType.STRING)
2 @Column(length = 9)
3 private TipoTelefoneEnum tipo;
4
5 public enum TipoTelefoneEnum {
6     CASA,
7     COMERCIAL,
8     CELULAR;
9 }
```

Por mais legível que isso possa ser para a pessoa desenvolvedora, essa coluna ocupa muito mais espaço do que o necessário. Nesse caso, a coluna de tipo ocupa 9 bytes. Se armazenarmos 10 milhões de registros, apenas a coluna tipo de telefone ocupará 90 MB.

Mapeando como inteiro

Observe que a coluna @Enumerated não precisa receber o valor ORDINAL EnumType, pois é usado por padrão. Também estamos usando o tipo de coluna número inteiro smallint, pois é improvável que precisemos de mais de 2 bytes para armazenar todos os valores do tipo Enum.

```
1 @Enumerated
2 @Column(columnDefinition = "smallint")
3 private TipoTelefoneEnum tipo;
```

O valor será armazenado como inteiro, iniciando com zero para o tipo CASA. Agora, isso é muito mais eficiente, mas menos expressivo. Então, como podemos ter desempenho e legibilidade?

Basta criarmos uma tabela no banco de dados representando o enum e, na consulta, fazer o join com a tabela de constantes. Parece trabalhoso? Mas pode valer a pena, se tivermos milhões de registros.

É claro que existem os contras dessa solução. Se o enum for alterado, mudado de ordem ou acrescido de novos valores, os registros da base de dados terão que ser ajustados também.

Portanto, é tudo uma questão de escolha! Então, escolha sabiamente.

Outras medidas de melhoria

Muitas outras medidas de performance podem ser adotadas ou verificadas no seu código. Abaixo, alguns materiais, em inglês, para auxiliar:

[Associação unidirecional ou bidirecional](#)

[Problema de consulta N + 1](#)

Referências: <https://vladmihalcea.com/tutorials/hibernate/>

Conclusão

Não tente resolver todos os problemas ao mesmo tempo. Comece construindo uma lista dos cinco principais contribuidores da hora e da queima da CPU, memória ou IO e explore soluções. Ataque um dos problemas e reavalie a arquitetura. Abaixo, algumas etapas que podem ajudar a encontrar e solucionar um problema de performance.

Descobrir: Por que este ponto está com baixa performance?

Entender: O que está causando a baixa performance?

Corrigir ou Melhorar: Oportunidade de corrigir ou melhorar com base nos dados obtidos nas etapas acima.

Apêndice A: Segurança

Práticas de Segurança

Código seguro deveria ser uma regra em qualquer design de aplicação, pois em alguns ramos são tão importantes quanto qualquer regra de qualidade que se possa ter. Para microserviços não é diferente, e aqui não existe nenhum segredo, é a mesma regra para qualquer desenho de aplicação: no mínimo se deve olhar para o [Top 10 do OWASP](#) e seguir essas recomendações no seu projeto. É essencial que profissionais que trabalham com desenvolvimento conheçam as vulnerabilidades descritas no OWASP.

Sério mesmo, não há nada novo aqui, e qualquer pessoa que esteja tentando convencer você de um “Cross Microservice Injection” ou algo do tipo está tentando te enrolar.

No caso de Microserviços, a abordagem que muda é em relação a Autorização e Autenticação, e vamos discutir isso mais à frente.

Uma dica importante, em se tratando de código seguro, utilize as práticas de CI/CD para validar se há alguma vulnerabilidade no seu código. É interessante confirmar se as bibliotecas de terceiros que você planeja utilizar não trarão problemas ao ambiente.

E ferramentas para isso há várias, entre elas temos o [Fortify](#), [Snyk](#), [JFrog Xray](#). Porque às vezes uma dependência desatualizada pode colocar seu serviço em posição de risco, então olhar a melhor prática no código e uma ferramenta para ajudar a apontar onde melhorar formam um time imbatível.

Outra prática que eu vejo em algumas empresas, principalmente quando os serviços estão expostos apenas internamente e não para fora, é não usar o HTTPS, ou melhor, usar um TLS (Transport Layer Security). E para que você precisa disso? Para ter privacidade, integridade e identificação.

E quando estamos falando de microserviços, um cenário que vai acabar sempre existindo diz respeito à necessidade de falar com servidores de autorização, e podemos estar falando de um API Key, ou de um “client secret”, ou até mesmo de credenciais para uma autenticação básica. Então a primeira regra básica é: não deixe essas chaves no seu repositório de fonte, esses itens precisam ser variáveis de ambientes ou chaves de configuração externa, e elas devem estar sempre encriptadas.

Como estamos falando de containers, as práticas valem também para lá, e nunca rode seu container como “root”. Você precisa assumir a premissa de que seu sistema nunca é 100% seguro, alguém vai conseguir explorar algo. Então você não pode só prevenir, você precisa detectar e reagir a isso.

A chave é seguir ao menos cinco pilares:

- Segurança por design (Secure by design);
- Vasculhe suas dependências;
- Use sempre HTTPS;
- Use Tokens de Acesso e Identidade;
- Proteja e criptografe seus segredos.

Soluções para Autenticação e Autorização

Para o mundo de microserviços, o principal ponto é verificar quem você é (Autenticação) e aquilo que você pode fazer (Autorização). E dentro da arquitetura de microserviços você vai estar espalhado em muito serviços pela rede e terá que lidar com alguns problemas em relação a como resolver isso.

Autenticação e Autorização precisam ser resolvidos em cada um dos microsserviços, e parte dessa lógica global vai ter que ser replicada em todos os serviços. Neste caso, um jeito para resolver isso é criar bibliotecas para padronizar essa implementação, só que isso vai fazer que você perca um pouco da flexibilidade de quais tecnologias usar, pois a linguagem ou framework precisa suportar essa biblioteca padrão.

O uso da biblioteca também ajuda a não quebrar o princípio da responsabilidade única, já que o serviço deveria se preocupar apenas com a lógica de negócio.

E outro ponto que é preciso analisar é que os microsserviços devem ser *stateless*, então é necessário usar soluções que consigam manter isso.

Podemos abordar a Autorização e Autenticação pelo modelo de sessão distribuída, usando ferramentas para você armazenar essa sessão. Você pode abordar/manter a sessão das seguintes maneiras:

Sticky Session - A ideia aqui é usar o load balancer e manter as conexões de origem sempre no mesmo servidor onde veio o primeiro request. Só que esta configuração só te permite escalar horizontalmente.

Replicação de Sessão - Toda instância salva a sessão e sincroniza através da rede. Só que aqui isso vai causar um “overhead” de rede. Quanto mais instâncias, mais será preciso replicar e lidar com a latência disso.

Sessão Centralizada - Isso significa que os dados podem ser recuperados em um repositório compartilhado. Em vários cenários, esse é um ótimo desenho, porque se pode dar alto desempenho para as aplicações, e você deixa o status do login escondido dentro dessa sessão. Mas claro que existe a desvantagem de que você precisa criar mecanismos para proteger essa sessão e replicar entre as aplicações, o que pode também adicionar

latências na sua rede.

Mas quando estamos neste cenário de microsserviços, a recomendação passa a ser o uso de Tokens. A maior diferença para o modelo de sessão descrito acima é que deixamos de ter algo centralizado em um servidor e passamos a responsabilidade para a própria pessoa que vai utilizá-lo.

O Token vai ter a informação de identificação da pessoa (user) que está logada, e toda vez que chega ao servidor, podemos validar no server a identidade e a autorização. O token é encriptado e pode seguir um padrão como o [JWT](#).

E usando token conseguimos delegar a responsabilidade do estado da pessoa (user) logada, para algum processo que possa a validade do mesmo. Habilitamos vários tipos de validações de segurança que podem ser colocadas na malha (Service Mesh) ou no seu gateway de entrada e retirar essas responsabilidades dos serviços e aplicações e mesmo assim ainda continuar garantindo a segurança.

Com o uso do JWT, você passa a ter um “client token”, e você vai passar a algum servidor para que ele possa fazer a validação/criação do mesmo.

E quando se fala em Tokens, a chave é não querer reinventar a roda, e sim usar aquilo que já está consolidado! É onde entra o OpenId e o OAuth/OAuth2. O Oauth 2 é praticamente o padrão mais utilizado para autenticação.

O padrão de OpenID é aquele usando quando você pode se conectar ou usar o token para se logar em vários sites ou serviços. Mas no seu padrão local, a recomendação é o OAuth, que estabelece um protocolo para que você tenha acesso aos recursos de que você precisa, e ele trabalha com quatro papéis:

Resource Owner - Esse é o papel que controla os acessos aos recursos.

Resource Server - É onde ficam os serviços a serem acessados, ou seja, aqui é onde estão as suas API's, aplicações e etc.

Client - É quem faz a solicitação ao Resource Owner do recurso que precisa consumir.

Authorization Server - Quem gera os tokens de acesso, permite que o Client chegue ao recursos que foram permitidos, com o nível de acesso definido.

E como funciona isso? Basicamente o “client” solicita ao Resource Owner acesso ao recurso, e este, quando autoriza, envia para o “Client” o “authorization grant”, que é a credencial que representa que o Resource Owner autorizou a passagem. Então o “Client” vai solicitar ao Authorization Server um Token de acesso; tudo sendo válido, o Client recebe o seu token de acesso, que vai ser repassado para o Resource Server para que ele possa consumir aquilo que foi solicitado.

E existem 4 tipos de fluxos para obtermos o token de acesso no padrão OAuth. Temos o Authorization Code, que é o tipo mais comum; o Implicit, que é muito utilizado por aplicações SPA; o Resource Owner, que estabelece a confiança entre as aplicações; e o Client Credentials, que é usado para falar de um serviço para outro.

Tirando a parte de Autenticação e Autorização, que precisam de um cuidado à parte, a segurança de microsserviços é como a segurança de qualquer aplicação e precisamos prestar atenção a isso.

Bibliografia

- Java Efetivo;
- Clean Code;
- Domain Driven Design (DDD):
 - Implementing DDD
 - DDD Distilled
 - The Business Value of Using DDD
 - Domain Driven Design Quickly
- Clean Architecture;
- Building Microservices;
- The Twelve Factors;
- Monolith to Microservices (Sam Newman)
- Fundamentals of Software Architecture: an Engineering Approach
- Cloud-Native Continuous Integration and Delivery: Build, test, and deploy cloud-native applications in the cloud-native way, Onur Yilmaz
- Kubernetes Patterns: Reusable Elements for Designing Cloud-Native Applications, por Bilgin Ibryam, Roland Huß
- Teaching an Elephant to Dance: Intentional evolution across teams, processes, and applications, por Burr Sutter e Deon Ballard
- Bigtable: a Distributed Storage System for Structured Data, by Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber