

Introdução à Engenharia de Software

José Mendes 107188

2023/2024



1 Maven

1.1 O que é o Maven?

É uma **ferramenta de gestão de projetos**, que inclui:

- Um **project object model** (POM) que descreve o projeto;
- Um conjunto de **standards**;
- Um **lifecycle** do projeto;
- Um sistema de gestão de **dependências**;
- Lógica para **executar plugins** em **fases** específicas do ciclo de vida.

Convenção sobre configuração (layout do projeto é padronizado).

1.2 Layout de Diretórios Padronizado

POM - Contém uma descrição completa do projeto de como construir o projeto.

src - Diretório que contém todo o código fonte para construir o projeto, o seu site, ...

target - Diretório que contém os resultados da construção, tipicamente um JAR ou WAR, juntamente com os ficheiros intermedios.

1.3 POM

Maven é baseado no conceito de um **Project Object Model** (POM). Este é um ficheiro XML, que está sempre localizado no diretório base do projeto como **pom.xml** (os users definiram POMs que estendem o Super POM).

O POM contém informação sobre o projeto e vários detalhes de configuração usados pelo Maven para construir o projeto.

O POM é declarativo, não necessita de detalhes de procedimento.

1.3.1 Estrutura do POM

O POM contém 4 categorias de descrição e configuração:

- Informação geral do projeto, isto é, informação human-readable;
- Configuração do build, que pode incluir, adicionar plugins, afixar plugins objetivo ao ciclo de vida;
- Ambiente de construção, que descreve o ambiente "familiar" em que o Maven está;
- Relações POM, isto é, coordenadas, herança, agregação, dependências.

1.4 Coordenadas Maven

As coordenadas definem o lugar único do projeto no universo Maven. São compostas por 3 partes: <groupId>, <artifactId> e <version> (The Maven trinity!).

As versões de um projeto são usadas para agrupar e ordenar lançamentos:

<major_version> . <minor_version> . <incremental_version> - <qualifier>

Exemplo: 1.0.0-SNAPSHOT ou 1.2.3-alpha-2

Se o qualifier contiver a palavra chave SNAPSHOT, então o Maven vai expandir este token para uma data e hora convertida para o formato UTC.

- **groupId** - Nome da empresa, organização, equipa, ..., normalmente usando a convenção de nomes de domínio invertidos (reverse URL naming, ex: org.apache.maven);
- **artifactId** - Nome único do projeto dentro do groupId;
- **version** - Versão do projeto;
- **packaging** - Tipo de empacotamento do projeto (jar (default), war, ...);
- **classifier** - Classificador opcional para distinguir artefactos

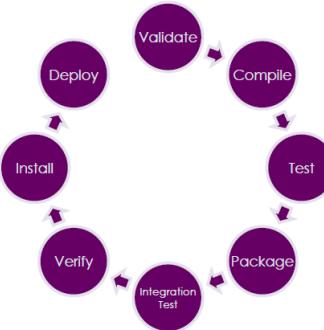
Nota: As coordenadas Maven identificam unicamente um projeto.

1.5 Ciclo de Vida Maven

Um ciclo de vida é uma sequência organizada de fases, que dão ordem a uma sequência de objetivos. Estes objetivos são empacotados em plugins que estão ligados as fases.

Clean Lifecycle	Default Lifecycle
pre-clean	test-compile
clean	process-test-classes
post-clean	test
	prepare-package
	package
	pre-integration-test
compile	integration-test
	post-integration-test
	verify
	install
	deploy
	process-test-resources

Chamar uma fase específica num ciclo de construção, vai executar todas as fases anteriores a essa fase.



1. **Validate** - Valida que a estrutura do projeto está correta. (ex: verifica se todas as dependências foram transferidas e estão disponíveis no repositório local);
2. **Compile** - Compila o código fonte, converte os ficheiros **.java** em **.class**, e armazenando-os no diretório **target/classes**;
3. **Test** - Corre testes unitários para o projeto;
4. **Package** - Empacota o código compilado num formato distribuível como **JAR** ou **WAR**;
5. **Integration Test** - Corre testes de integração para o projeto;
6. **Verify** - Corre verificações para verificar que o projeto é válido e que cumpre os critérios de qualidade;
7. **Install** - Instala o código empacotado no repositório Maven local, para uso como dependência noutros projetos locais;
8. **Deploy** - Copia o pacote final de código para o repositório remoto para partilha com outros developers e projetos.

1.6 Ciclo de Vida de Construção

O processo para contruir e distribuir um projeto. Consiste em vários passos designados por **fases**.

Algumas fases default são:

- validate
- compile
- test
- package
- deploy

1.7 Goals e Plugins

Os Goals são operações fornecidas pelas ferramentas Maven.

Cada fase é uma sequência de Goals, em que cada Goal é responsável por uma tarefa específica. Quando corremos uma fase, todos os Goals ligados a essa fase são executados, na ordem em que estão definidos.

Algumas Maven Plugins:

- resources
- compiler
- surefire
- jar, war

1.8 Arquétipos (Archetypes)

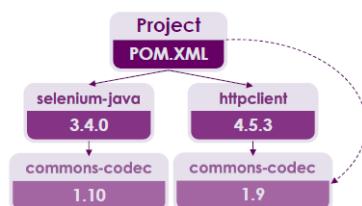
Um Archetype é um template para um projeto Maven, que pode ser usado para criar um novo projeto rapidamente.

Exemplo: *maven-archetype-quickstart* ou *maven-archetype-webapp*

Users podem criar os seus próprios Archetypes e publicá-los através de catálogos.

1.9 Gestor de Dependências

Uma **dependência** de um projeto é uma biblioteca da qual o projeto depende. Adicionar uma dependência ao projeto é simples, basta adicionar a dependência ao POM. O Maven vai automaticamente procurar a dependência no repositório local, e se não encontrar, vai procurar no repositório remoto e transferi-la.



2 Git e GitHub

2.1 Sistemas de Controlo de Versões

Um sistema de controlo de versões (também conhecido como sistema de controlo de código fonte) faz o seguinte:

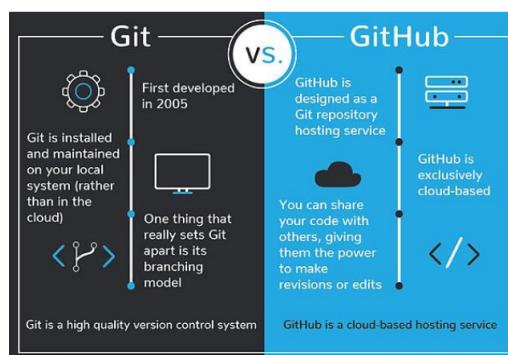
- Mantém várias (antigas e novas) versões de tudo (não só código fonte);
- Pede por comentários quando se fazem alterações;
- Permite "check-in" e "check-out" de ficheiros para saber em que ficheiros outras pessoas estão a trabalhar;
- Mostra as diferenças entre versões;

2.1.1 Vantagens

Ao trabalhar sozinho: Fornece uma "máquina do tempo" para voltar atrás para uma versão anterior, e fornece um bom suporte de diferentes versões do mesmo projeto.

Ao trabalhar em equipa: Simplifica muito trabalhar em concurrenceia, dando "merge" de alterações feitas por diferentes pessoas.

2.2 o que é Git e GitHub



Quando fazemos "git init" num diretório de um projeto, ou quando fazemos "git clone" de um projeto existente, o Git cria um repositório (.git).

Em qualquer momento, podemos fazer um "snapshot" de tudo no diretório do projeto e guardar este no repositório. Este "snapshot" é chamado de **commit object**.

Um **commit** ocorre quando fazemos alterações que estão prontas para serem guardadas no repositório.

Quando realizamos um commit, o Git guarda um **commit object**:

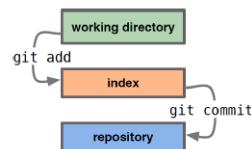
- Um estado completo do projeto, incluindo todos os ficheiros;
- O primeiro não possui pai;
- Normalmente, pegamos num commit object, fazemos alterações, e criamos um novo commit object, pelo que a maior parte dos commit objects têm apenas um pai;
- Quando fazemos **merge** de dois commit objects, forma um commit object com dois pais.

Pelo que, os commit objects formam uma **DAG** (Directed Acyclic Graph). O Git é tudo sobre usar e manipular este grafo.

2.2.1 Mensagem de Commit

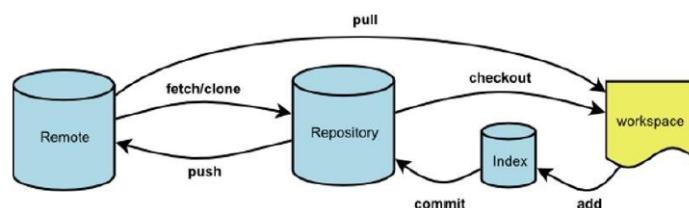
Os commits são "baratos" pelo que os devemos fazer com frequência, e com mensagens descriptivas sobre o que foi alterado. Devem ter apenas uma linha.

Como não devemos dizer muito numa linha, devemos fazer vários commits.



2.3 Manter simples

- ❖ If you:
 - Make sure you are current with the central repository
 - Make some improvements to your code
 - Update the central repository before anyone else does
- ❖ Then you don't have to worry about resolving conflicts or working with multiple branches
 - All the complexity in git comes from dealing with these
- ❖ Therefore:
 - Make sure you are up-to-date before starting to work
 - Commit and update the central repository frequently



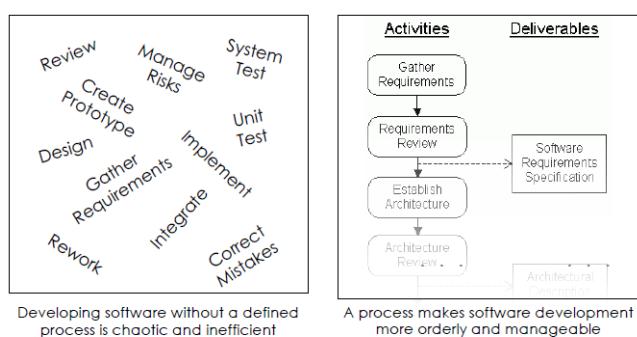
3 O Processo de Desenvolvimento de Software

3.1 Processo

A fundação para a Engenharia de Software é a camada de processo. Um processo de Software é uma framework para as atividades, ações, e tarefas necessárias para construir software de alta qualidade. Define as técnicas e a framework de gestão para aplicação de métodos, ferramentas e pessoas ao longo do processo de desenvolvimento.



3.2 Porquê o Processo de Software?



"It is better not to proceed at all, than to proceed without method." – Descartes

3.3 Processo de Software

Existem vários tipos de processos de software, no entanto, todos têm:

- **Especificação (comunicação e planeamento)** - definir o que o sistema deve fazer;
 - **Design e Implementação** - definir a organização do sistema e implementar o sistema;
 - **Validação** - verificar que faz aquilo que o cliente quer;
 - **Evolução** - alterar o sistema em resposta a novos requisitos impostos pelo cliente.

Quando discutimos sobre o processo de software, estamos a falar sobre:

- **Atividades** - como especificar um modelo de dados, design de uma interface de utilizador, ...;
 - **Ordem** a ordem destas atividades;

A descrição de processos pode também incluir, **produtos** (outcome da atividade do processo), **papéis** (roles, responsabilidades das pessoas envolvidas) e **pré-/pós-condições** (são condições que são verdadeiras antes e depois de atividade do processo ou de um produto ser produzido).

O processo de software específica:

- **O quê**
- **Quem**
- **Quando**
- **Como**

E incluí

- **Papéis** (Roles)
- **Fluxo de trabalho** (Workflow)
- **Procedimentos** (Procedures)
- **Normas** (Standards)
- **Modelos** (Templates)

3.4 Pontos Chave

O processo de Software é um guia

Não existe ”um melhor processo para escrever software”. Um processo que um individuo ou uma organização escolhe e segue depende de:

- das características específicas do projeto;
- da cultura da organização;
- das habilidades e preferências das pessoas envolvidas.

Um bom processo aumenta a produtividade de membros da equipa menos experientes sem impedir o trabalho/progresso de membros mais experientes.

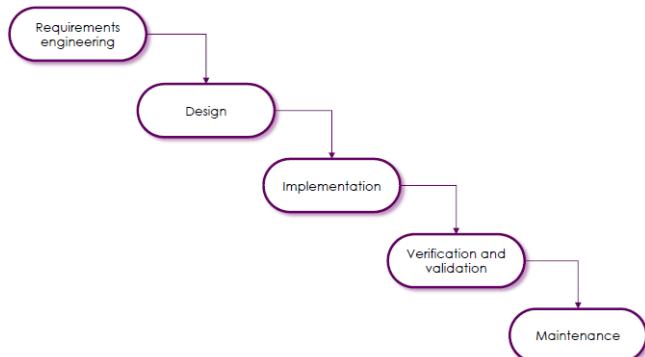
3.5 Resistência ao Processo de Software

Perceção: Algumas pessoas vêm seguir um processo como uma sobrecarga (overhead) desnecessária na produtividade.

- Interfere com a criatividade;
- Burocracia e regimento;
- Prejudica a agilidade em mercados que evoluem rapidamente.

A realidade: Grupos que não seguem um processo definido tendem a adicionar processo mais tarde no projeto, como reação a problemas que surgem. Quando o tamanho e a complexidade do projeto aumenta, a importância de seguir processos definidos aumenta proporcionalmente.

3.6 Fases de Software



3.7 Modelos de Processo de Software

Modelos abstratos que descrevem uma classe abordagens de desenvolvimento com características similares.

Alguns critérios utilizados para distinguir modelos de processos de software são:

- o tempo entre fases (timing);
- critérios de entrada e saída entre fases (entry/exit criteria);
- os artefactos criados durante cada fase;

Alguns **exemplos** incluem: Waterfall, Spiral, Rapid Prototyping, Incremental, Development,
...

3.7.1 Modelos (Tradicionais)

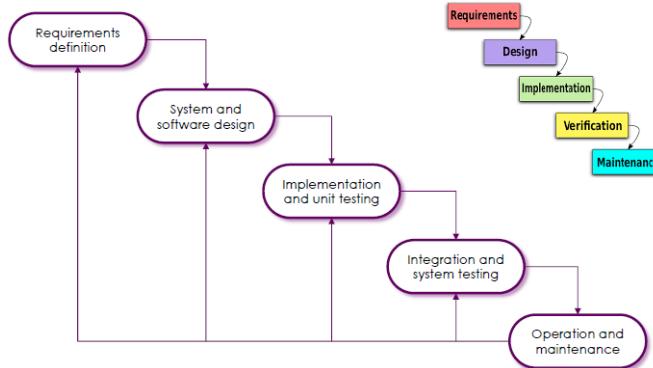
Modelo em Cascata (Waterfall): É um modelo Plan-Driven. Separa e distingue fases de especificação e desenvolvimento.

Desenvolvimento Incremental: A especificação, desenvolvimento e validação são intercalados. Pode ser Plan-Driven ou Agile.

Processos Evolucionários/Iterativos: O sistema é desenvolvido no ínicio usando uma especificação muito simples, sendo modificada e melhorada de acordo com as necessidades de software.

Muitos outros: A maior parte de sistemas grandes são desenvolvidos usando um processo que incorpora elementos de diferentes modelos.

3.7.2 O Modelo em Cascata (Waterfall)



Vantagens

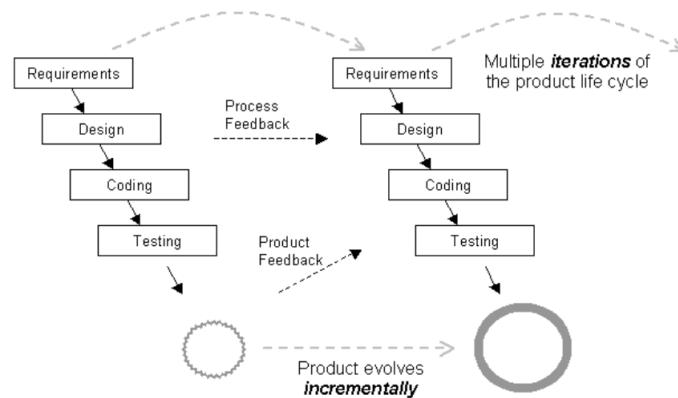
- É simples e fácil de perceber e usar;
- É fácil de planear, um schedule pode ser definido com deadlines para cada fase de desenvolvido e um produto pode ser processado através do processo de desenvolvimento como um carro numa lavagem automática, e ,teoricamente, ser entregue a tempo.
- Fácil de gerir, cada fase tem entregas específicas e um processo de revisão.
- Fases e processos são concluídos um de cada vez.
- Funciona bem onde os requisitos são bem compreendidos.

Desvantagens

- Dificuldade em acomodar mudanças após o processo começar. Em princípio, uma fase deve ser concluída antes de começar a próxima. Particionamento inflexível do projeto em fases distintas torna difícil responder a mudanças nos requisitos do cliente.
- Modelo não muito bom para projetos de longa duração ou que já estão em andamento (não é produzido nenhum software funcional até mais tarde no ciclo de vida).
- Não é adequado a processos onde os requisitos são incertos ou onde há risco de serem alterados.

3.7.3 Modelo Incremental

Uma característica de modelos com ciclos de vida modernos. O produto evolui através de uma série de iterações.



Benefícios

- O custo de **acomodar mudanças de requisitos do cliente** é reduzido. A quantidade de análise e documentação que tem de ser refeita é muito menor do que no modelo em cascata.
- É mais fácil **obter feedback do cliente** sobre o desenvolvimento do trabalho que já está concluído. Clientes podem comentar sobre demonstrações do software e ver quanto foi implementado.
- **Entrega mais rápida e deployment** do software útil para o cliente é possível. Os clientes podem usar e ganhar valor do software mais cedo do que se o sistema fosse desenvolvido com o processo em cascata.

Problemas

- **Cada fase de iteração é rígida** e não se sobreponem umas com as outras.
- O processo não é visível. Os gestores precisam de entregas regulares para medir o progresso. No entanto, se o sistema não for desenvolvido rapidamente, não é cost-effective produzir documentos que reflitam cada versão do sistema.
- A estrutura do sistema tende a degradar-se à medida que novos incrementos são adicionados. A não ser que tempo e dinheiro seja gasto na refatoração para melhorar o software, **regular mudanças tende a corromper a sua estrutura**. A medida que vamos incorporando novas mudanças de software, torna-se mais difícil e mais caro.

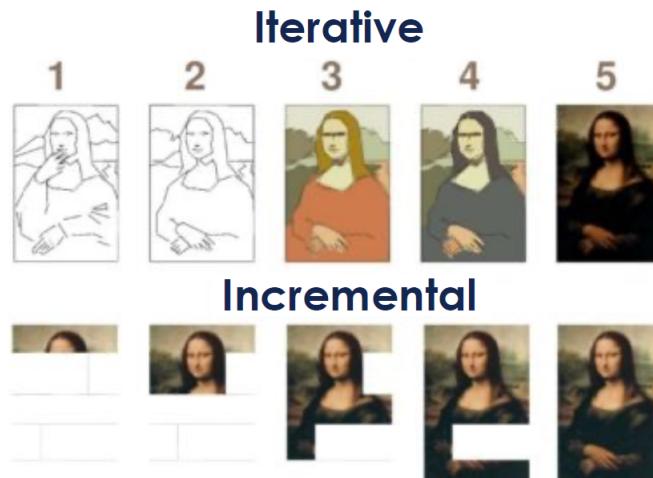
3.7.4 Modelos Evolucionários/Iterativos

Prototipagem: Geralmente, um cliente define um conjunto de objetivos gerais para o software, mas não identifica requisitos detalhados para funções e funcionalidades do sistema.

Modelo Espiral: Utilizando um modelo espiral, o software é desenvolvido numa série de lançamentos (releases) evolutivos. Durante as primeiras iterações, o lançamento pode ser um protótipo ou um modelo.

Modelo Concurrente: Permite a uma equipa de software representar elementos iterativos e concurrentes de quaisquer modelos de processo.

3.7.5 Incremental vs Evolucionário/Iterativo



3.7.6 Exemplos

❖ Scenario

- You are developing a web-based e-commerce platform for a client. The client has requested **several features**, including user authentication, product catalogue, shopping cart functionality, and payment processing. They want to launch the platform as soon as possible to start generating revenue but also want to **add new features and improvements over time**.

❖ Which approach do you propose?

- Incremental

1. Minimal viable product (MVP) – user authentication and catalogue
2. Shopping cart
3. Payment process

❖ Scenario

- You need to develop a machine learning-based recommendation system for an online streaming service. The goal is to provide personalized content recommendations to users based on their viewing history and preferences. The **requirements are complex**, and it's **essential to continually refine the recommendation** algorithms for better accuracy and user satisfaction.

❖ Which approach do you propose?

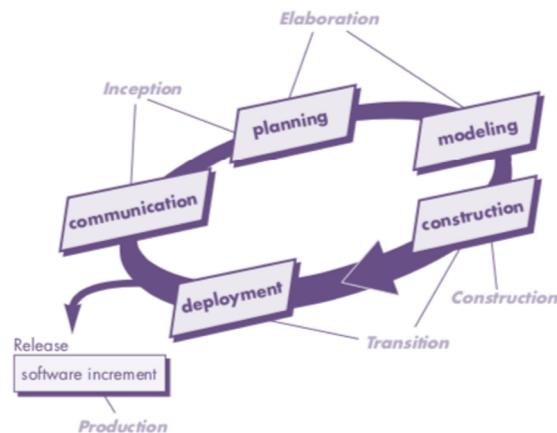
- Iterative

1. Initial version of the system
2. Simple recommendation algorithm
3. Refine algorithms and models (in subsequent iterations)

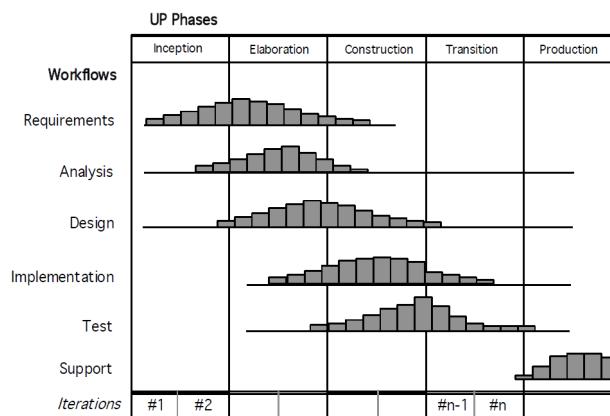
3.8 Outros Modelos de Processo

- Desenvolvimento **Component-Based** (COTS): O processo para aplicar quando reutilizar é um objetivo do desenvolvimento.
- **Métodos Formais**: Enfatiza a especificação matemática dos requisitos.
- **Processo Unificado (UP)**: Um processo de software "use-case driven, arquitetura-centric, iterativo e incremental", alinhado com o Unified Modeling Language (UML).

3.9 Processo Unificado (UP)



3.9.1 Fases



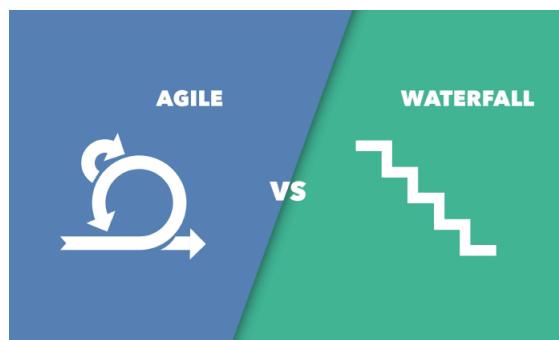
3.10 Processos Plan-Driven e Agile

Processos Plan-Driven são processos onde todas as atividades são planeadas antecidamente e o progresso é medido contra este plano.

Em **Processos Agile**, planear é incremental e é mais fácil mudar o processo para refletir a mudança nos requisitos do cliente.

Na prática, a maior parte dos processos incluem elementos de ambos, processos plan-driven e agile. Não existem processos de software "certos" ou "errados".

3.10.1 Processos Plan-Driven vs Agile



3.10.2 Processos Agile

Desenvolvimento rápido e entregas rápidas são, geralmente, os requisitos mais importantes para sistemas de software.

- Negócios operam num ambiente **fast-changing requirements** e é praticamente impossível produzir um conjunto de requisitos de software estáveis.
- O Software deve evoluir rapidamente para refletir mudanças de negócios.

Desenvolvimento Plan-Driven é essencialmente para alguns tipos de sistemas mas não cobre as necessidades do negócio.

Métodos Agile

- Métodos Agile foram desenvolvidos num esforço para ultrapassar fraquezas percebidas e reais em engenharia de software convencional.
- **Foca no código em vez de no design.**
- Baseados em **abordagens iterativas** ao desenvolvimento de software.
- Tem a intenção de **entregar software funcional rapidamente** e evoluir rapidamente para refletir mudanças de requisitos do cliente.

3.10.3 Origem: Manifesto Agile

"We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

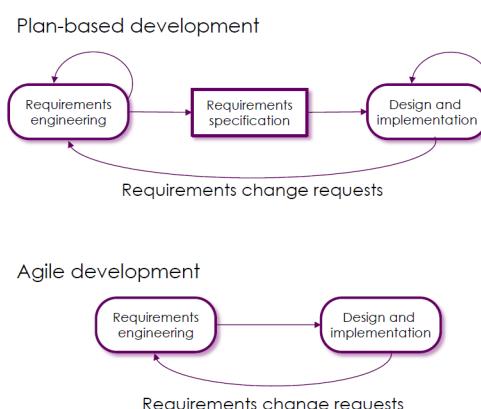
- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

That is, while there is value in the items on the right, we value the items on the left more."

3.10.4 Princípios dos Métodos Agile

Involvimento do cliente: Os clientes devem estar envolvidos durante todo o processo de desenvolvimento. O seu papel é fornecer e priorizar novos requisitos e avaliar as interações do sistema.

3.10.5 Desenvolvimento Plan-Driven e Agile



Desenvolvimento Plan-Driven:

- Baseado ao redor de um desenvolvimento separado de fases, com outputs a serem produzidos em cada fase planeada antecidiamente.
- Não é necessariamente o modelo em cascata - desenvolvimento incremental, plan-driven, é possível.

Desenvolvimento Agile:

- Especificação, design, implementação e testes são intercalados.
- Os outputs do processo de desenvolvimento são decididos através de um processo de negociação durante o processo de desenvolvimento de software.

3.10.6 Métodos Agile - Benefícios

- Requisitos num modelo Agile podem ser alterados conforme os requisitos do cliente mudam. Por vezes os requisitos não são muito claros. Mudanças nos requisitos são aceites mesmo em fases avançadas do processo de desenvolvimento.
- A entrega de software é continua. Clientes podem seguir cada feature funcional do Sprint do software.
- Refatorar o código não é muito caro.

3.10.7 Métodos Agile - Desvantagens

- A documentação é escassa.
- Com requisitos pouco claros, é difícil estimar o resultado pretendido. Mais difícil de estimar o esforço necessário.
- Alguns riscos desconhecidos/imprevisíveis que podem afetar o desenvolvimento do projeto.

3.10.8 Métodos Agile - Aplicabilidade

- Desenvolvimento de produtos, onde uma empresa de software está a desenvolver um produto pequeno/médio em tamanho para venda.
- Desenvolvimento de sistemas customizados dentro de uma organização, onde existe o compromisso do cliente ficar envolvido no processo de desenvolvimento e onde existem algumas regras/regulamentos externos que afetam o software.
- Virtualmente, todos os produtos de software e aplicações são desenvolvidas usando abordagens Agile.

4 Desenvolvimento de Software Agile

4.1 Princípios Agile

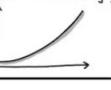
4.1.1 Porquê Agile?

O desenvolvimento rápido e entrega são, geralmente, os requisitos mais importantes para sistemas de software.

- Negócios operam num ambiente **fast-changing requirements** e é praticamente impossível produzir um conjunto de requisitos de software estáveis.
- O Software deve evoluir rapidamente para refletir mudanças de negócios.

Princípios Agile:

- **Focada no código em vez de no design;**
- Baseados em **abordagens iterativas** ao desenvolvimento de software;
- Tem a intenção de **entregar software funcional rapidamente** e evoluir rapidamente para refletir mudanças de requisitos do cliente.

Satisfy the customer through early and continuous delivery of valuable software. 	12 Agile Principles @OlgaHeismann Welcome changing requirements, even late in development. 	Business people and developers must work together. 
Build projects around motivated individuals. Give them the support they need. Trust them. 	The most efficient and effective method of conveying information is face-to-face conversation. 	Working software is the primary measure of progress. 
Continuous attention to technical excellence and good design. 	Simplicity—the art of maximizing the amount of work not done—is essential. 	The team reflects on how to become more effective and adjusts its behavior accordingly. 

Ver analogia nos slides 6 a 15.

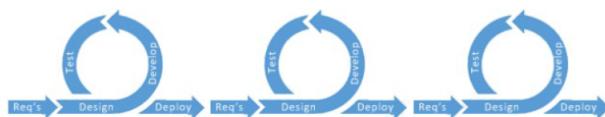
4.2 Técnicas de Desenvolvimento Agile

As fases especificação, design, implementação e avaliação são intercaladas:

- O sistema é desenvolvido como uma série de versões, envolvendo **stakeholders** na especificação e avaliação;
- Entregas de novas versões frequentes para avaliação;

Vasto suporte de ferramentas (ex: ferramentas de testes automáticos) usado para suportar o desenvolvimento.

Minima documentação, uma vez que o foco é o código.



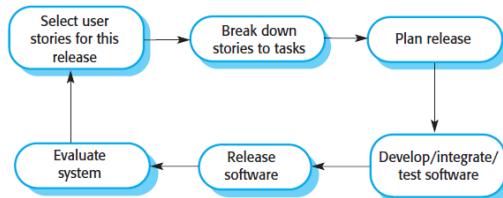
4.2.1 Extreme Programming (XP)

Extreme Programming (XP) é a abordagem mais utilizada para desenvolvimento ágil de software. **Leva uma abordagem "extrema" para o desenvolvimento iterativo:**

- Novas versões podem ser construídas várias vezes por dia;
- Os incrementos são entregues aos clientes a cada 2 semanas;
- Todos os testes devem correr para cada build e cada build apenas é aceite se todos os testes correrem com sucesso;

Utiliza uma abordagem orientada a objetos como o paradigma de desenvolvimento preferido. Abrange um conjunto de regras e práticas que ocorrem no contexto de quatro atividades fundamentais (framework activities): **planning, design, coding, testing**.

4.2.2 Release Cycle XP



4.2.3 Práticas XP

Principle or practice	Description	Principle or practice	Description
Incremental planning	Requirements are recorded on story cards and the stories to be included in a release are determined by the time available and their relative priority. The developers break these stories into development 'Tasks'.	Pair programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Small releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.	Collective ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop, and all the developers take responsibility for all the code. Anyone can change anything.
Simple design	Enough design is carried out to meet the current requirements and no more.	Continuous integration	As soon as the work on a task is complete, it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Test-first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.	Sustainable pace	Large amounts of overtime are not considered acceptable as the net effect is often to reduce code quality and medium-term productivity.
Refactoring	All developers are expected to refactor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable.	On-site customer	A representative of the end-user of the system (the customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.

4.2.4 Práticas influentes de XP

Extreme Programming tem um foco técnico e não é fácil de integrar com prática de gestão na maior parte das organizações. Consequentemente, enquanto desenvolvimento Agile utiliza práticas de XP, o método referido não é muito utilizado.

Práticas Chave:

- User stories para especificação;
- Refatoração;
- Desenvolvimento Test-First;
- Programação em pares (pair programming);

4.2.5 User Stories para Requisitos

Em XP, o cliente ou user são parte da equipa XP e são **responsáveis por tomar decisões sobre requisitos**.

Requisitos do user são expressos como **user stories** ou **scenarios**. Estes são escritos em cartões e a equipa de desenvolvimento divide-os em **tarefas de implementação**. Estas tarefas são a base das estimativas de custo e de schedule.

O cliente escolhe as **stories para inclusão** na próxima release baseado em prioridade e estimativa de schedule.

Exemplo slide 24 e 25.

4.2.6 Templates de User Stories

As a **(user)**, I **(want to)**, **(so that)**.

- **User:** Refere o user final do software;
- **Want to:** Refere à intenção do user, e não à feature que este utiliza. Temos de especificar aquilo que o user quer alcançar com a tarefa, sem mencionar a UI da aplicação;
- **So that:** Descreve a "bigger picture". Qual é o objetivo do user? O que é que ele quer alcançar com esta feature?

Exemplo: As a manager, I want to be able to understand my colleagues progress so that I can report our success and failures.

4.2.7 Organização de User Stories

- **Role-Goal-Benefit:** Força o cliente a realmente pensar quem é que vai beneficiar de uma feature, o que é que eles querem alcançar e porque é que eles querem alcançar isso;
- **Limites/Needs:** Subconjunto de situações que são de interesse para esta feature;
- **Definição de Done:** Como validar uma feature e saber que esta está concluída?
- **Tarefas de Engenharia:** Como esta feature interage com outras features dentro do sistema ou outros subsistemas;
- **Estimativa de esforço:** Fornece uma medida concreta sobre o valor de uma feature;

4.2.8 Refatoração

Conhecimento convencional em engenharia de software é **design for change**. Compensa gastar tempo e esforçarmo-nos a anticipar mudanças, uma vez que, reduz o custo mais à frente no ciclo de vida.

No entanto, XP mantém que este princípio não compensa, uma vez que, mudanças não podem ser previstas. Em vez disso, propõe **melhorias contantes do código** (refactoring) para tornar mudanças mais fáceis quando têm de ser implementadas.

Equipes de programação procuram por possíveis **melhorias de software** e fazem-nas mesmo que não sejam necessárias naquele momento. Isto melhora a **compreensão do software** e desta maneira reduz a necessidade de documentação.

Mudanças são **fáceis de fazer** uma vez que o código é **bem estruturado** e **claro**. No entanto, algumas mudanças requerem architecture refactoring, o que é muito mais caro.

Exemplo

- **Re-organização** de uma classe hierárquica para remover duplicação de código;
- **Renomear** atributos e métodos de modo a serem mais fáceis de entender;
- **Trocar inline code** com chamadas de métodos que estão incluídas em bibliotecas;

4.2.9 Desenvolvimento Test-First

Testar é uma parte central em XP. O **desenvolvimento test-first**:

- Teste incremental é desenvolvido a partir de cenários;
- Envolvimento de users no desenvolvimento de testes e validação;
- Correr todos os testes de componente de cada vez que uma nova versão é built.

Test 4: Dose checking	
Input:	1. A number in mg representing a single dose of the drug. 2. A number representing the number of single doses per day.
Tests:	1. Test for inputs where the single dose is correct but the frequency is too high. 2. Test for inputs where the single dose is too high and too low. 3. Test for inputs where the single dose * frequency is too high and too low. 4. Test for inputs where single dose * frequency is in the permitted range.
Output:	OK or error message indicating that the dose is outside the safe range.

4.2.10 Automação de Testes

Testes são escritos como **componentes executáveis** antes da tarefa ser implementada. Estes devem:

- Ser autonomos (stand-alone);
- Simular a submissão de input para ser testado;
- Verificar que o resultado é o esperado;

Uma **automated test framework** (ex: Junit) torna mais fácil de escrever e executar testes.

Uma vez que os testes são automáticos, existe um conjunto de testes que podem ser rapidamente e facilmente executados. Quando uma nova funcionalidade é adicionada ao sistema, todos os testes podem ser executados e identificar problemas no novo código imediatamente.

4.2.11 Problemas com o desenvolvimento Test-First

- Os programadores preferem programar em vez de escrever testes, pelo que, por vezes estes fazem **short cuts ao escrever testes**. Como pro exemplo, testes incompletos que não testam todas as possibilidades.
- Alguns testes são **muito difíceis de escrever incrementalmente**. Como testes para uma UI complexa, é difícil escrever testes com display logic.
- É difícil de julgar a **cobertura** de um conjunto de testes. Podemos ter muitos testes de sistema e não cobrir tudo.

4.2.12 Programação em Pares (Pair Programming)

Um parte programadores a trabalharem juntos para desenvolver código. Os programadores sentam-se juntos num mesmo computador para desenvolver software. Serve como uma forma de **revisão de código informal** uma vez que cada linha de código é olhada por mais do que uma pessoa.

Isto ajuda a desenvolver common ownership do código e compreensão coletiva pela equipa. Reduz o risco quando um programador deixa a equipa.

Em pair programming, os pares são criados dinamicamente para que todos os membros da equipa trabalhem uns com os outros durante o processo de desenvolvimento. Encoraja a refatorização, uma vez que, toda a equipa pode benificar com a melhoria do código do sistema.

Pair programming não é necessariamente ineficiente. Estudos mostram que um par ao trabalhar junto é mais eficiente do que dois programadores a trabalhar separadamente.

4.3 Gestão de Projeto Agile

A principiar responsabilidade da gestão de projeto de software é **gerir o projeto**, para que o software seja entregue a tempo e dentro do budget planeada para o projeto.

A abordagem tradicional para a gestão de projeto é **plan-driven**. Define o que deve ser entregue, quando deve ser entregue e quem vai trabalhar em cada entrega.

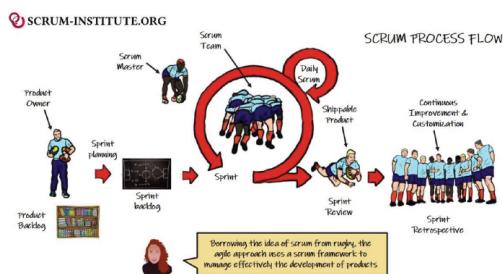
A gestão de projeto Agile requer uma abordagem diferente. É adaptada de um **desenvolvimento incremental** e as **práticas usadas em métodos agile**.

4.3.1 Scrum

É um **método agile** que foca na gestão incremental do desenvolvimento.

Existem três fases no Scrum:

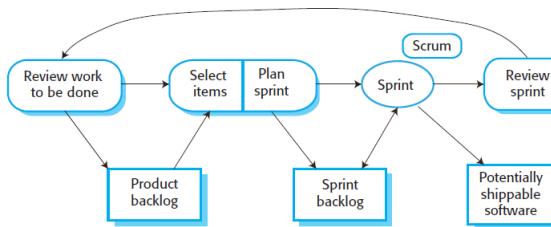
- A fase inicial é o **outline planning** onde tem de se estabelecer os objetivos gerais para o projeto e fazer o design da arquitetura de software.
- Isto é seguido por uma serie de **ciclos sprint**, onde cada ciclo desenvolve um incremento do sistema.
- A fase de **encerramento do projeto**, termina o projeto, completa a documentação necessária como ajudas nos frames e manual do utilizador e acessa aquilo que foi aprendido com o projeto.



4.3.2 Scrum - Terminologia

Scrum term	Definition	Scrum term	Definition
Development team	A self-organizing group of software developers, which should be no more than 7 people. They are responsible for developing the software and other essential project documents.	Scrum meetings	A daily meeting of the Scrum team that reviews progress and prioritizes work to be done that day. Ideally, this should be a short face-to-face meeting that includes the whole team.
Potentially shippable product increment	The software increment that is delivered from a sprint. The idea is that this should be 'potentially shippable' which means that it is in a finished state and no further work, such as testing, is needed to incorporate it into the final product. In practice, this is not always achievable.	ScrumMaster	The ScrumMaster is responsible for ensuring that the Scrum process is followed and guides the team in the effective use of Scrum. He or she is responsible for interfacing with the rest of the company and for ensuring that the Scrum team is not diverted by outside interference. The Scrum developers are adamant that the ScrumMaster should not be thought of as a project manager. Others, however, may not always find it easy to see the difference.
Product backlog	This is a list of 'to do' items which the Scrum team must tackle. They may be feature definitions for the software, software requirements, user stories or descriptions of supplementary tasks that are needed, such as architecture definition or user documentation.	Sprint	A development iteration. Sprints are usually 2-4 weeks long.
Product owner	An individual (or possibly a small group) whose job is to identify product features or requirements, prioritize these for development and continuously review the product backlog to ensure that the project continues to meet critical business needs. The Product Owner can be a customer but might also be a product manager in a software company or other stakeholder representative.	Velocity	An estimate of how much product backlog effort that a team can cover in a single sprint. Understanding a team's velocity helps them estimate what can be covered in a sprint and provides a basis for measuring improving performance.

4.3.3 Scrum sprint cycle



O ponto inicial para o planeamento é o **product backlog**. É uma lista do trabalho a ser feito no projeto.

A **fase selecionada** envolve toda a equipa. Quem trabalha com o cliente para selecionar as features e as funcionalidades do **product backlog** que vão ser desenvolvidas durante o sprint.

Durante a **fase de desenvolvimento**, a equipa está isolada do cliente e da organização. Todos os canais de comunicação passam pelo **"Scrum master"**.

No **final do sprint**, o trabalho realizado é revisto e apresentado aos stakeholders.

4.3.4 Teamwork in Scrum

O papel do **Scrum master** é proteger o a equipa de desenvolvimento de distrações externas.

- Realiza reuniões diárias;
- Vai verificando o backlog para ver o trabalho a ser feito;
- Toma decisões;
- Mede o progresso usando o backlog;
- Comunica com os clientes e a gestão fora da equipa;

A equipa vai a reuniões diárias curtas (**Scrums**)

- Membros partilham informação, descrevem o progresso/problemas desde a última reunião, e aquilo que estava planeado para o dia;
- Isto significa que todos na equipa sabem aquilo que se passa, os problemas que surgiram, podem re-planear num curto tempo o trabalho;

4.3.5 Scrum - Benefícios

- O produto é repartido num conjunto de pedaços geriveis. e faceis de compreender.
- Requisitos instáveis não prendem o progresso;
- Toda a equipa tem visibilidade de tudo, consequentemente, a comunicação da equipa melhora;
- Os clientes vêm as entregas on-time dos incrementos, ganhando feedback em como o produto funciona;
- Confiança entre a equipa e o cliente, todos esperam que o projeto tenha sucesso;

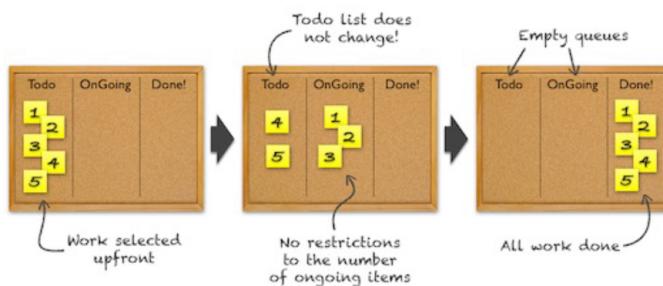
4.3.6 Kanban

É um sistema ágil que pode ser usado para melhorar qualquer desenvolvimento de software incluindo Scrum, XP ou Waterfall.

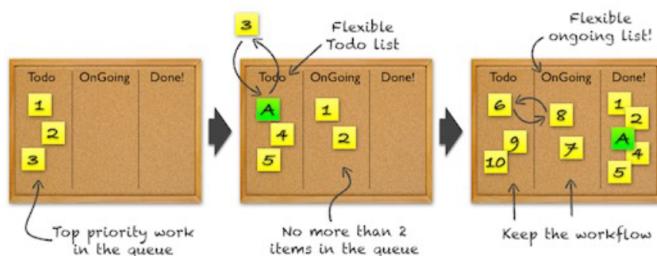
O nome "Kanban" vem do Japonês e significa "signboard" ou "billboard". Foi usado pela primeira vez pela Toyota para Just-in-time manufacturing plants, com o objetivo de limitar o trabalho em progresso (WIP - work in progress).

4.3.7 Scrum vs Kanban

Em **Scrum**, tu escolhes o trabalho que vais fazer durante o próximo sprint antes deste começar. Depois, damos lock ao sprint, fazemos o trabalho todo, e passado algumas semanas (duração normal para um sprint), a queue fica vazia.



Em **Kanban**, tudo o que é limitado é o tamanho das queues, chamado de **work in progress (WIP) limits**. Podemos alterar os items na queue a qualquer momento, não existindo um "fim de sprint". O trabalho continua a fluir.



	Scrum	Kanban
Cadence	Regular fixed length sprints (i.e., 2 weeks)	Continuous flow
Release methodology	At the end of each sprint	Continuous delivery
Roles	Product owner, scrum master, development team	No required roles
Key metrics	Velocity	Lead time, cycle time, WIP
Change philosophy	Teams should not make changes during the sprint.	Change can happen at any time

4.4 Scaling Agile Methods

Métodos Agile provaram ser muito eficazes para projetos pequenos/médios, que podem ser desenvolvidos por uma equipa pequena co-localizada. Escalar métodos Agile envolve mudar estes para poderem ser utilizados em projetos cada vez maiores, onde existem múltiplas equipas de desenvolvimento, podendo trabalhar até mesmo em diferentes locais.

Quando escalamos métodos Agile, é importante manter fundamentos Agile: Planeamento flexível, releases de sistema frequentes, integração contínua, desenvolvimento test-driven e boa comunicação entre a equipa.

4.4.1 Problemas de Sistema

- Quão grande é o sistema a ser desenvolvido? Métodos Agile são mais eficientes em equipas pequenas e co-localizadas, em que ocorre uma comunicação informal.
- Que tipo de sistema está a ser desenvolvido? Sistemas que necessitem de **muita análise antes da implementação** precisam de um design (mais) detalhado.
- Qual é o tempo de vida esperado do sistema? **Sistemas de longa duração necessitam de documentação** para comunicar as intenções do sistema com os developers da equipa de suporte.
- O sistema está sujeito a regulamentos externos? Se o sistema é regulado, muito provavelmente, é necessário **produzir documentação detalhada** como parte do caso de segurança do sistema.

4.4.2 Desenvolvimento de Sistemas Grandes

Sistemas grandes são normalmente coleções de sistemas de comunicação separados, onde **equipas separadas** desenvolvem cada sistema. Frequentemente, estas equipas trabalham em locais diferentes, por vezes até em zonas com diferentes fusos horários.

Sistemas grandes são ”**brownfield systems**” isto é, incluem e interagem com vários sistemas existentes. Muitos dos requisitos do sistema estão relacionados com a interação, logo não se rendem muito à flexibilidade e desenvolvimento incremental.

Sistemas grandes e os seus processos de desenvolvimento são muitas vezes restritos por **regras e regulamentos externos**, limitando a forma como podem ser desenvolvidos.

Sistemas grandes tem uma **aquisição longa** e um tempo de desenvolvimento longo também. É difícil de manter equipas coerentes que sabem sobre o sistema durante esse período, uma vez que, eventualmente, os membros vão para outros projetos ou trabalhos.

Sistemas grandes normalmente tem um conjunto diverso de **stakeholders**. É praticamente impossível envolver todos estes stakeholders no processo desenvolvido.

4.4.3 Agile: Scaling up em sistemas grandes

- Uma abordagem completamente incremental aos requisitos de engenharia é **impossível**;
- **Não pode existir** um único product owner;
- **Não é possível** focar apenas no código;
- Mecanismos de comunicação cross-team têm de ser **designed e utilizados**;
- Integração continua é praticamente **impossível**, no entanto, é essencial manter builds frequentes do sistema e regular releases do sistema;

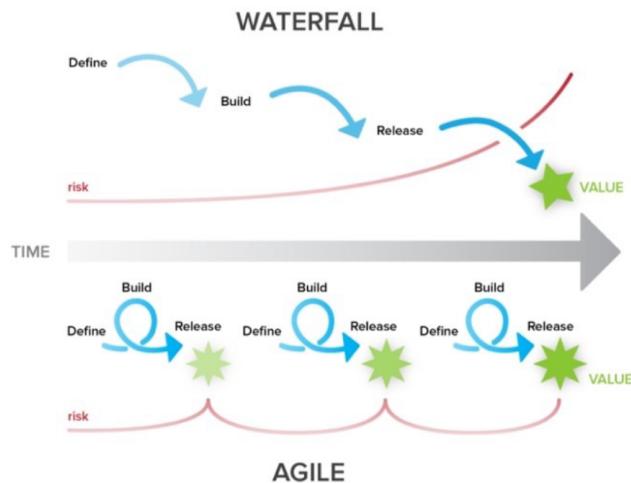
4.4.4 Multi-team Scrum

- **Role replication:** Cada equipa tem um Product Owner para o seu trabalho e Scrum Master.
- **Product architects:** Cada equipa escolhe um arquiteto do produto que colaboram para desenhar e evoluir toda a arquitetura do sistema.
- **Release alignment:** As datas de releases de produtos de cada equipa são alinhadas de modo a demonstrar que um sistema completo está produzido.
- **Scrum of scrums:** Existe um Scrum of scrums diário, onde os representantes de cada equipa se encontram para discutir o progresso e o plano de trabalho a ser concluído.

5 DevOps

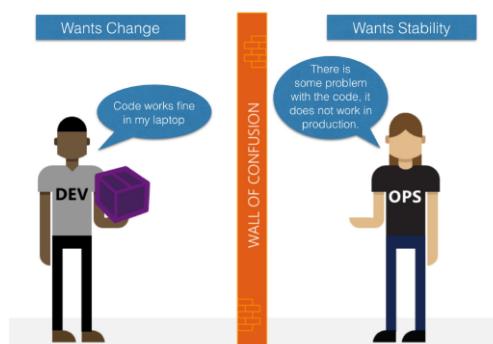
5.1 Metodologia Agile

- Cada projeto é dividido em várias iterações;
- Todas as iterações devem ter a mesma duração (2-8 semanas);
- No final de cada iteração, um produto funcional deve ser entregue;



5.2 Quais são as limitações de Agile?

Enquanto que a parte de Desenvolvimento é continua, a parte de Operações não é.



5.3 Dev vs Ops

A equipa de **desenvolvimento** começa a trabalhar no projeto, "atirando" uma release de software "por cima da parede" para as Operações.

As **Operações** pegam nos artefactos da release e começam a preparar o seu desenvolvimento. Estes também editam os ficheiros de configuração para refletir o ambiente de produção (que é **bastante diferente dos ambientes de Desenvolvimento**).

Em caso de falha...

- Os Developers são chamados para resolver o problema;
- **Operações** afirma que O Desenvolvimento deu código defeituoso;
- **Desenvolvimento** responde apontando o facto de que funcionava perfeitamente nos seus ambientes (logo o problema é das Operações, que fizeram algo de errado).
- Os Developers estão a ter dificuldades em sequer perceber o problema. Devido à configuração, localização dos ficheiros, ao procedimento utilizado para chegar a esse estado é diferente do esperado.

Qual é a Solução?

5.4 DevOps

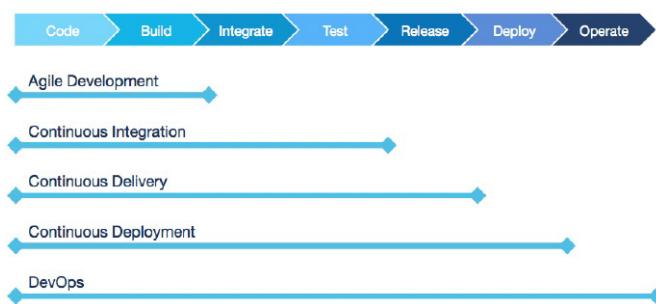
DevOps é sobre a remoção das barreiras entre duas equipas tradicionalmente isoladas: **Desenvolvimento e Operações**.

Em algumas organizações, pode nem haver uma equipa separada, **os engenheiros fazem ambos**.

As **duas equipas trabalham juntas** de modo a otimizar tanto a produtividade dos desenvolvedores e a fiabilidade das operações. Estas **comunicam frequentemente** aumentando a eficiência e a qualidade dos serviços que fornecem aos clientes.

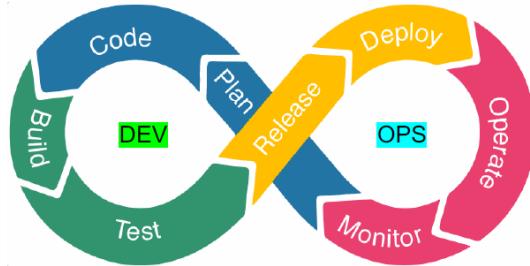
Têm um **ownership completo para os seus serviços**, pensando nas necessidades dos clientes finais e como podem contribuir para resolver essas necessidades.

As equipas vêm o ciclo completo de desenvolvimento e infraestruturas como parte das suas responsabilidades.

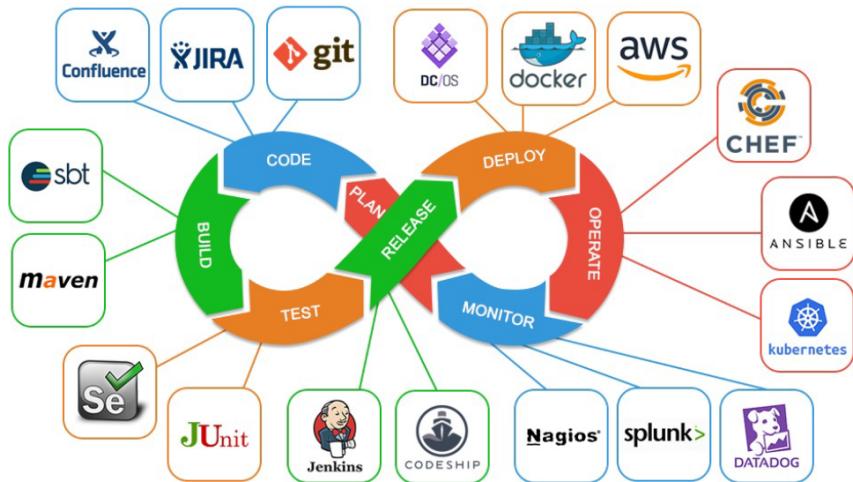


-
- **Desenvolvimento** Agile. Gestão do código fonte (Plan, Code, Build). Manutenção de diferentes versões do código;
 - **Integração Contínua** (Integrate, Test). Compilar, revisão de código, testes unitários, testes de integração;
 - **Entrega Contínua** (Release). Deploy da aplicação (build), realizando testes de aceitação nos users;
 - **Desenvolvimento Contínuo** (Deploy, Operate). Deploy na aplicação testada/aceite;
 - **Monitorização Contínua**;

5.4.1 DevOps - Ciclo de Vida



5.4.2 DevOps - Como implementar?



5.5 Um asset chave: Infraestrutura como Código (IaC)

a.k.a. Programmable Infrastructure ou software-defined infrastructure

- A infraestrutura é descrita por código e pode ser trackeada, validada, e reconfigurada de forma automática;
- Engenheiros DevOps podem interagir com a infraestrutura utilizando ferramentas code-based e tratando a infraestrutura de uma forma semelhante a como tratam o código da aplicação;

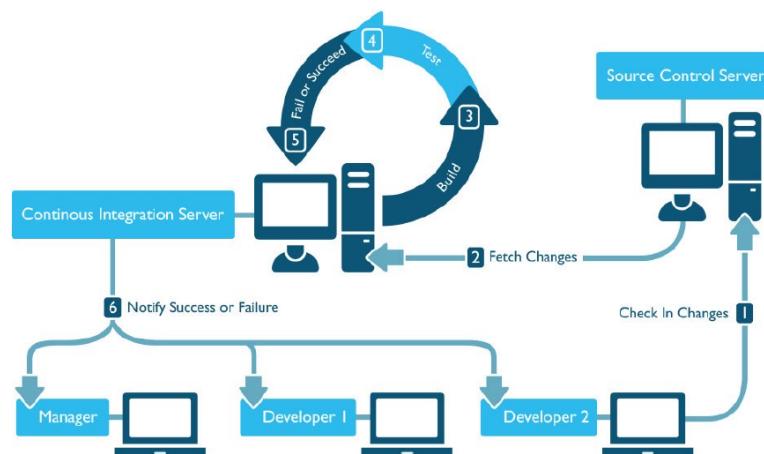
Gestão de configuração

- Developers e administradores de sistemas usam código para automar sistemas operativos e configurar hosts, tarefas operacionais, e mais;
- A utilização de código faz com que as mudanças na configuração sejam repetíveis e padronizadas;

5.6 Integração Contínua (CI)

5.6.1 Ideia base de CI

A continua integração é um processo onde o código é verificado para um repositório frequentemente.



5.6.2 Porque é que CI é crucial para DevOps?

Erros são encontrados mais cedo:

- Se existe um erro na cópia local ou no código, uma falha na build vai ocorrer no "estado apropriado";
- Força o desenvolvedor a corrigir o bug antes de proceder. Equipas QA (Quality Assurance) vão também beneficiar disto, uma vez que estas vão trabalhar mais frequentemente em builds estáveis;

Definir o estágio para CI/CD:

- CI reduz intervenção manual, uma vez que a build, sanidade e outros testes são supostamente executados automaticamente;
- Isto permite uma entrega continua de sucesso

Confidencialidade do Projeto

5.6.3 Getting Started with CI

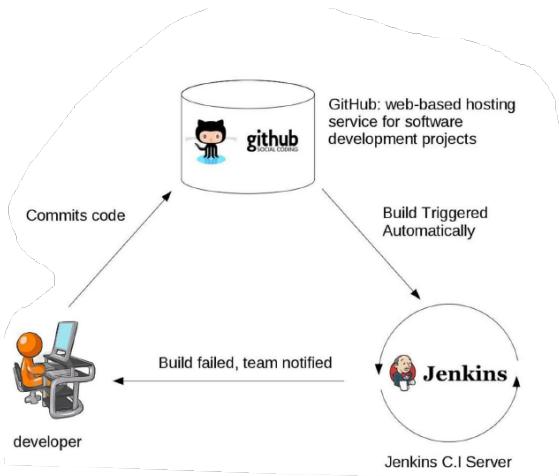
- **Build Script** (ex: Maven, Gradle, Ant, Make, ...), é um script, ou um conjunto de scripts, usados para compilar, testar, inspecionar e dar deploy a um software;
- **Version Control System** (ex: Git, GitHub, GitLab, ...), permite guardar todas as mudanças feitas a um ficheiro ou conjunto de ficheiros ao longo do tempo, de modo a que seja possível aceder a versões específicas mais tarde;
- **CI Server** (ex: Jenkins, Travis CI, Hudson, ...), é um servidor que corre uma build de integração quando uma mudança é committed num repositório de controlo de versões. Apesar de ser recomendado correr uma build a cada mudança, buikds podem ser agendadas (ex: nightly builds);
- **Automation testing framework** (ex: Selenium, Appium, TestComplete, ...);

5.6.4 Build Script com Maven

- ❖ Lifecycle phases (or goals)
 - **validate** - check if all information necessary for the build is available
 - **compile** - compile the source code of the project
 - **test-compile** - compile the test source code
 - **test**: run unit tests
 - **package**: package compiled source code into the distributable format (jar, war, ...)
 - **integration-test**: process and deploy the package if needed to run integration tests
 - **install** - install the package into the local repository, for use as a dependency in other projects locally
 - **deploy** - copies the final package to the remote repository for sharing with other developers and projects.
- ❖ Convention over Configuration

5.6.5 CI Server com Jenkins

O código é built e testado quando um desenvolvedor faz um commit. Se o build for bem sucedida, Jenkins faz deploy da fonte para um servidor de testes e notifica a equipa de deployment. Se a build falhar, então o Jenkins vai notificar os erros à equipa de desenvolvimento.



5.6.6 CI - Best Practices

Versioning

- Repositório partilhado para manter o código. Todos os ficheiros dente são committed para o repositório;
- Merge diário. As mudanças são committed para a mainline, diariamente;
- Short-lived branches. É assim que devem ser, idealmente menos de uns dias e nunca mais do que uma iteração;

Build Configuration

- Build independente. Escrever build scripts desacopelados de IDEs. Estes são executados pelo sistema CI para que o software possa ser built a cada mudança;
- Build em cada mudança. Building e testar software a cada mudança committed;
- Limite de build. Falhar uma build quando uma regra do projeto é violada (ex: testes lentos);

Team Policy

- Stop the line (build quebrada). Corrigir erros de entrega de software mal estes aconteçam;
- Fazer commits regularmente. Cada membro da equipa deve verificar o "trunk", pelo menos uma vez por dia;
- Feedback continuo. Enviar feedback automaticamente do sistema CI para os membros da Cross-functional team;
- Falhas rápidas. Falhar a build o mais rápido possível;
- Builds rápidas. Uma build fornece feedback em problemas comuns o mais rápido possível (normalmente menos de 10 minutos);

5.7 Entrega Contínua (CD)

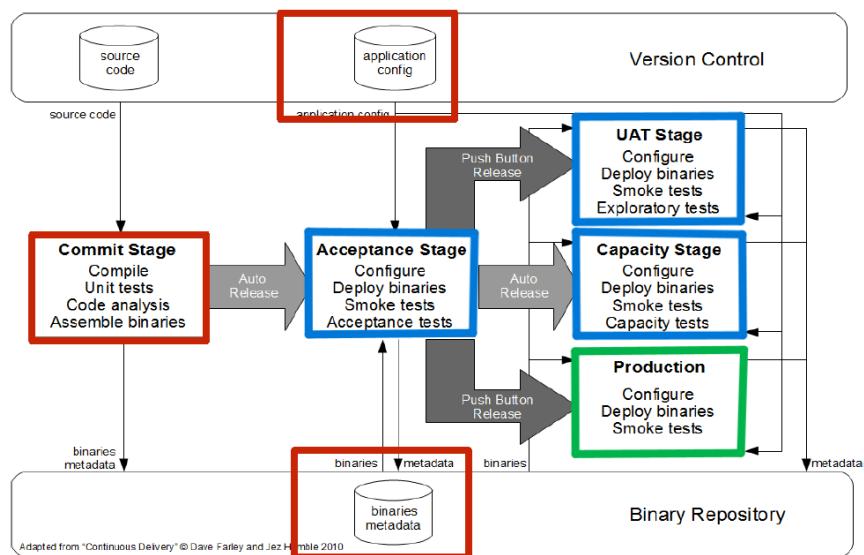
A Engenharia de Software é baseada em **CI**. CI foca nas equipas de desenvolvimento, em que o output é o input do processo de testes manuais e o resto do processo de release.

Muito tempo pode ser "perdido" desta forma através de testes e operações (ex: testers esperam por "boas" builds de software equipas de operações esperam por documentação e fixes).

CD previne isto, as relações entre stakeholders envolvidos na entrega é mais proxima e colaborativa (cultura DevOps).

Automação extensiva a todas as partes do processo de entrega, CD pipeline.

5.7.1 CD Pipeline



Vermelhos: Fase de CI; **Azuis:** Fase pós CI; **Verdes:** Fase de CD;

Fase pós CI

Consiste de múltiplos ambientes/estágios com o objetivo de testar. Inclui testes **manuais e automáticos**.

Alguns estágios podem ser executados em paralelo (ex: user acceptance testing (UAT) e performance testing em sistemas de backend).

Estágios podem ser automaticamente executados quando o anterior termina ou é manualmente aprovado (ex: clicar um botão).

Fase de CD

Novas versões estão disponíveis para os clientes. Existem várias estratégias e práticas para fazer isto:

- eat your own dog food;
- canary releases;
- dark launches;
- gradual rollouts;
- A/B testing;
- blue/green deployments;

5.7.2 Eat your own dog food

Ideia: Os empregados utilizam a versão mais recente internamente. Se tudo estiver como esperado, então a versão é lançada para os clientes.

”A companhia usa o seu próprio produto para testar e promover o produto”.

5.7.3 Canary Releases

Ideia: Lançar uma nova versão para um subconjunto de utilizadores primeiro, enquanto o resto interage com a versão antiga estável. Em caso de problemas com esta nova versão, apenas um pequeno número de utilizadores é afetado, logo o impacto do problema é reduzido.

Canário é comparado à versão existente em termos do conjunto dos critérios como estabilidade, performance, ou correção. Os utilizadores são escolhidos com base na sua localização, role (ex: admin, early, stable, ...) e de forma aleatória (ex: 1% dos utilizadores).

5.7.4 Dark Launches

Ideia: Mitigar problemas de performance e de reliability de novas funcionalidades ao enfrentar carga semelhante à de produção, através de um lançamento em produção, sem ser visível para os utilizadores.

Dark launches são normalmente released para um grupo de utilizadores que não sabem que estão a ser testados. A nova funcionalidade não lhes é indicada de qualquer maneira. Dev teams podem identificar e corrigir os erros restantes. Preocupações de escalabilidade antes de habilitar a funcionalidade para os utilizadores.

5.7.5 Gradual Rollouts

Ideia: Aumentar o número de users escolhidos para a nova versão de uma forma gradual, até substituir completamente a versão antiga.

Mostra se a nova versão pode lidar com aumentando a carga e se escala corretamente.

Muitas das vezes é combinada com **canary releases** e **dark launches**.



5.7.6 A/B Testing

Ideia: Comparar duas versões de software uma à outra, normalmente apenas diferencia num aspeto testado, para determinar o efeito de uma dada mudança

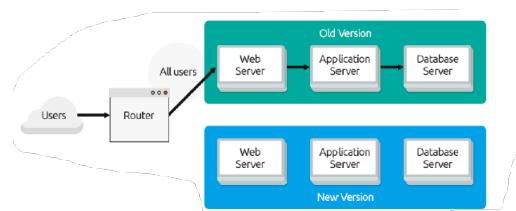
- forma estatística de testar hipóteses, pelo que requerem amostras grandes o suficiente para ter "poder estatístico";
- maioritariamente usado no UI para testar varios layouts ou designs;



5.7.7 Blue/Green Deployments

Ideia: ter dois ambientes de produção idênticos, um hospedando a versão atual do software (green), versão estável, a outra (blue) representa a nova versão.

- Release: simplesmente mudar o roteiro para que todo o tráfego seja direcionado para o ambiente blue (ou green) e utilizar o ambiente green (ou blue) para testar a nova versão;
- Vantagens: em caso de problemas depois da release, um rápido rollback (switch) para a versão anterior é possível;
- Obstaculo: As bases de dados fazem parte de ambos os ambientes, pelo que um switch requer uma migração de dados / sincronização da base de dados;



5.8 Benefícios de DevOps

Velocidade

- **Mover a elevada velocidade** de forma a inovar mais rápido para os clientes, adaptar as mudanças nos mercados de forma melhor, e crescer mais eficientemente em obter resultados de negócio;
- Por exemplo, CD deixam as equipas ter ownership dos serviços e depois dar release a updates para ser mais rápido;

Fiabilidade

- Garante a **qualidade dos updates da aplicação** e as mudanças de infraestrutura para poder entregar fiavelmente a um ritmo mais rápido, enquanto se mantém uma experiência positiva para o cliente;
- Usar práticas como CI e CD para testar cada alteração é funcional e seguro;

Escala

- Automação e consistência **ajudam a gerir sistemas complexos ou com mudanças frequentes** eficientemente e com risco reduzido;
- Por exemplo, infraestruturas como código ajuda a gerir o desenvolvimento, testar, e ambientes de produção de forma repetível e mais eficiente;

Melhora a colaboração

- **As equipas de Dev e Ops** colaboram, partilham muitas responsabilidades e combinam os seus workflows;
- Isto reduz ineficiências e salva tempo, escrevendo código, e a reduz a passagem de informação entre equipas;

Segurança

Mover rapidamente enquanto mantém o controlo e preserva a conformidade;

Um pode adotar o modelo DevOps sem sacrificar a segurança através de **políticas automaticas de compliance**, controlos fine-grained e configuração de técnicas de gestão;

Por exemplo, usar infraestruturas como código (e política como código), podemos definir e depois rastrear a conformidade em escala;

5.9 DevOps - CALMS

DevOps não é puramente técnico, também inclui aspectos sumarizados pelo acrônimo **CALMS**:



6 Clean Project

6.1 Engenharia de Software - Rever

1. Processo de desenvolvimento de software
 - Modelo sequencial (waterfall)
 - Modelo incremental
 - Modelo evolutivo/iterativo
2. Metodologias de desenvolvimento Agile
 - Princípios Agile e gestão de projeto
3. DevOps, benefícios técnicos
 - Entrega de software contínua
 - Entregas de features mais rápidas (time to market)

6.2 Papéis



6.2.1 Team Manager

- Modera as discussões (reuniões) da equipa. Promove a colaboração dentro da equipa e toma iniciativa para resolver problemas;
- Gere e atribui as tarefas;
- Pode ser visto como um Scrum Master;
- **Responsável por entregar os outcomes do projeto a tempo;**

6.2.2 Product Owner

- Representa os interesses dos stakeholders;
- Sabe aquilo que a aplicação deve fazer (features, requisitos, user stories);
- Responsável por aceitar os incrementos da solução. Deve rever novas releases;

6.2.3 Architect

- Responsável pela arquitetura de software. Modelação da aplicação e interações entre os componentes;
- Sabe as tecnologias usadas: Frontend, Backend, Caching, Meassage Queues, ...

6.2.4 DevOps Master

- Responsável pela infraestrutura;
- Garante a portabilidade do sistema;
- Sabe tudo sobre: Deployment Machine, repositório Git, Cloud Infrastructure, Operações de bases de dados, ...

6.3 Planeamento de Software

Especificação: Definir o que o sistema deve fazer;

Design e Implementação: Definir a organização do sistema e implementar o sistema;

Validação: Verificar que faz o que o cliente quer;

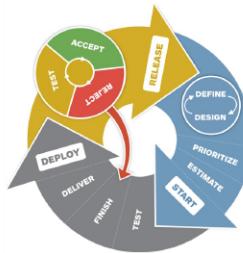
Evolução: Alterar o sistema como resposta à mudança nas necessidades do cliente;

6.3.1 Especificação

- Definição dos requisitos e stories;
- Ferramentas para gerir o desenvolvimento: Priorizar, atribuir, e dar track ao trabalho;
- Mas... Como fazemos isto? A resposta é utilizando ferramentas de planeamento de projetos como o **Jira**, **Pivotal Tracker**, **GitHub** e **GitLab** (estes 2 últimos possuem repositório de código)

Pivotal Tracker

- ❖ Agile project manager tool
- ❖ Allows the easy management of stories
 - Features, bugs, chores and releases
- ❖ Estimation of effort
 - Divide into 4 levels
- ❖ Backlog divide into iterations
- ❖ Provides good documentation



Workflow Overview

1. Escrever stories;
2. Priorizar stories;
3. Estimar stories;
4. Começar stories;
5. Acabar e entregar stories;
6. Testar stories;
7. Aceitar ou rejeitar stories;
8. As stories vão para o painel de "Done";

GitHub

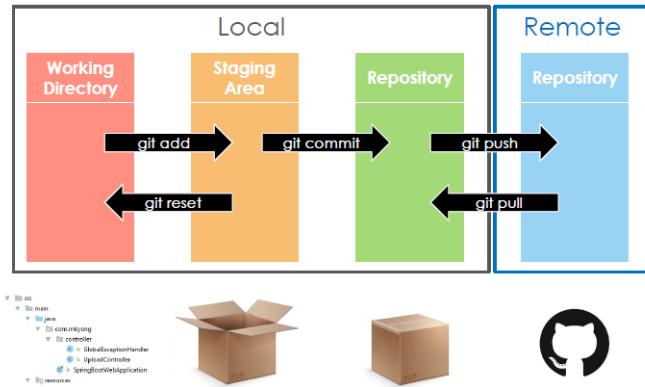
- É mais do que um repositório de código;
- Tem funcionalidades de gestão de projetos
 - Gestão de equipas;
 - Track de issues (podem seguir princípios semelhantes a stories);
- A comunidade cria novas apps de forma continua (para gestão personalizada);

GitHub Marketplace

Apps para integrar nos GitHub projects. Possui diferentes categorias: Code review, continuous integration, security, testing, monitoring, ...

6.3.2 Design e Implementação

Feature-branching workflow, com repositórios de código, utilizando Git como sistema de controlo de versões.



O que é um commit?

É uma operação fundamental para gravar mudanças no repositório. Um hash SHA-1 único é utilizado para identificar o commit.

Inclui: O conteúdo do ficheiro committed, a mensagem de commit, o nome e email do autor, o nome e email do committer, timestamp e talvez mais...

Commits diários

Cenário

- Trabalhar num projeto por duas semanas sem fazer um único commit;
- O disco do computador decide morrer;
- **E agora?**

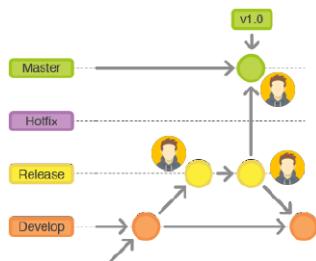
Nunca se deve esperar que uma funcionalidade esteja completa para fazer um commit. **Todos os dias**, deve-se fazer commit do trabalho e push ao código para o repositório.

Git Workflow



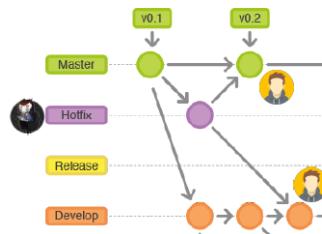
Nova release

- Preparar o produto para ser mostrado ao cliente, fechando assim um ciclo de desenvolvimento;
- Checkout de um dev;
- Quando a release está pronta: merge à release para a master e merge da branch para o dev;



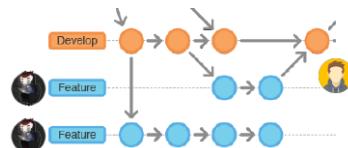
Hotfix

- Um bug catastrófico foi encontrado;
- O que fazer?
 - Checkout da master;
 - Corrigir o bug;
 - Merge para a master e a dev;
- Normalmente não é preciso uma nova branch para uma release;



Nova feature

- **Nova branch** para cada feature;
- Checkout da dev;
- Quando a feature está completa: merge da dev para a feature, merge da branch para a dev;

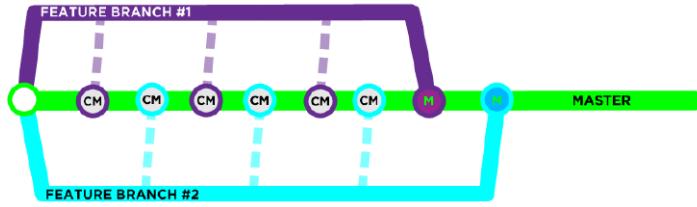


Pull/Merge Requests

- Dar merge a branches precisa de um pedido, normalmente para proteger as branches (master e dev);
- Um pull request precisa de aprovação, normalmente do manager do git (DevOps Master);
- Por vezes a implementação precisa de melhorias, como quando uma feature não está completa ou conflitos complexos durante o merge;

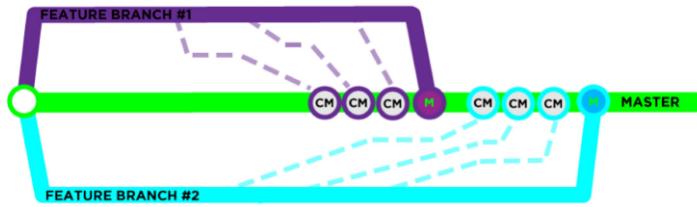
6.3.3 Merge Workflow

- Commit interlock (lock de commits);
- Difícil de seguir o histórico de commits;



6.3.4 Rebase Workflow

- Não há interlock de commits;
- É mais fácil de comunicar o histórico de commits;



6.3.5 Merge ou Rebase?

Merge

- **Vantagens:** Non-destructive, branches existentes não são alterados de qualquer maneira, apenas temos um novo commit (fácil de reverter);
- **Desvantagens:** Polui o histórico do repositório, torna difícil de perceber a evolução;

Rebase

- **Vantagens:** Projeto muito mais limpo, com histórico do projeto linear;
- **Desvantagens:** Fácil de fazer mal, reescrevendo o histórico, mais difícil de resolver conflitos;

6.3.6 Nomes das Branches

Cada programador gosta da sua própria convenção, no entanto estas convenções não são standards. O nome das branches é importante, como nomes bons para variáveis no código.

CATEGORY	DESCRIPTION
bug	Bug fixing
imp	Improvement on already existing features
new	New features being added
wip	Works in progress - Big features that take long to implement and will probably hang there
junk	Throwaway branch created to experimentation
release	New release before merging with master

Por exemplo um URL dá redirect para a página errada, #123:

- bug/fixURLRedirect (bom)
- bug/fix_url_redirect (também é bom)
- bug/fix_url_redirect_123 (melhor)

Se for para a página errada do perfil: bug/accounts/fix_url_redirect_123

6.4 Troubleshooting

Dar merge a conflitos que são muito complexos:

- Pedir ao developer para dar update à branch;
- Dar merge à dev atual para a branch;

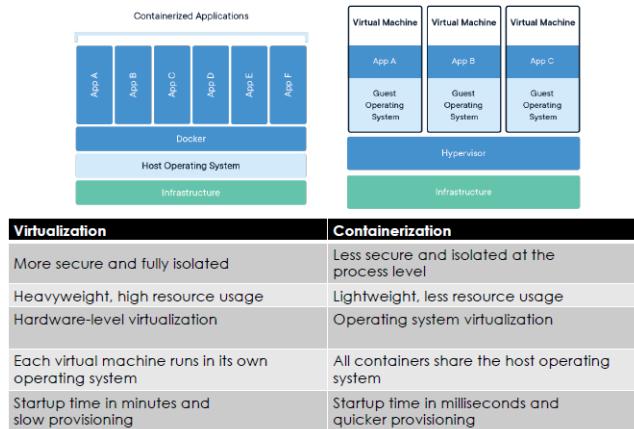
Se dermos commit a dados sensíveis, é possível reverter mas pode ser perigoso.

Dependências: "It works on my machine".

6.5 Desenvolvimento baseado em contentores

- Uma boa solução para problemas de dependências;
- Todos utilizam o mesmo ambiente;
- Ambientes de produção e desenvolvimento são muito semelhantes;
- Simplifica a integridade de diferentes serviços;
- Fácil de dar deploy;

6.5.1 Virtualização e Containerização



6.5.2 Docker

As imagens de produção e desenvolvimento são as mesmas, apenas possuem diferentes configurações. Em produção:

- Utiliza-se **sempre** volumes para dados sensíveis;
- Os containers morrem, mas os volumes não (normalmente);

6.5.3 Imagens

- As imagens são baseadas em contentores;
- Uma imagem pode servir múltiplos contentores, mas um contentor só pode ser uma imagem;
- Permite herança:
 - FROM ubuntu:20.04
 - FROM myImage:base

6.5.4 Docker Compose

- Simplifica a integração entre contentores;
- Permite a orquestração de contentores, baseada numa certa ordem;
- Não devemos alterar o ficheiro docker-compose.yml
 - Em vez disso, devemos definir variáveis (.env);
 - Criar um ficheiro .env-example com as variáveis base;
- Depois de toda a configuração, o startup é trivial: **docker-compose up -d**

Exemplo nos slides 45-54

7 Padrões de Arquitetura Enterprise

A **Arquitetura** é o **conceito de mais alto nível** para os desenvolvedores experientes. Define-se como um **Conhecimento partilhado** que descreve como o **sistema se divide em componentes** e como é que estes interagem entre si através de **interfaces**.

Nota: Diz-se **conhecimento partilhado** porque todos os desenvolvedores do projeto devem ter noção da arquitetura.

Cada componente mais abstrato é composto por vários componentes mais pequenos, mas se estes não forem do conhecimento/compreensão de todos os desenvolvedores, então não fazem parte da arquitetura.

7.1 Decisões de Arquitetura

As **decisões arquiteturais** têm um grande impacto na estrutura do projeto, pelo que implicam decisões bem pensadas e que nem sempre são fáceis de tomar.

Por exemplo, a decisão de desenvolver uma user interface **web-based** para a aplicação, ou a decisão de usar as java server faces (JSF) como a **web framework**. A decisão em que os componentes devem estar **distribuídos** remotamente para uma maior escalabilidade, ou a decisão de usar REST para **comunicar** entre componentes distribuídos.

Por isso, embora quando tomadas possam parecer demasiado óbvias e irrelevantes, **devem ser bem documentadas**, identificando as condições e restrições e analizando cada opção com base nas anteriores.

Caso contrário, pode acontecer o anti-pattern **Groundhog Day**, que diz que decisões arquiteturais importantes que foram feitas são perdidas, esquecidas ou não são comunicadas de forma eficiente. Ninguém percebe o porquê da decisão ter sido feita, continuando a ser discutida outra e outra vez...

7.1.1 Justificar Decisões

1. Identificar as condições e restrições;
2. Analisar cada opção com base nas condições;
3. Tomar uma decisão;
4. Justificar a decisão;

7.2 Documentação e Comunicação

Desde cedo estabelecer onde as decisões arquiteturais serão documentadas e **ter a certeza que todos os membros da equipa sabem** onde ir para encontrar a informação. Num documento ou wiki, em vez de ter múltiplos ficheiros espalhados por uma drive partilhada que já tem muitos ficheiros. A documentação deve ser **centralizada**.

Soluções email-driven (horrível): as pessoas esquecem-se, perdem, ou não sabem que uma decisão de arquitetura foi tomada, logo, não implementam a arquitetura corretamente.

7.3 Evitar a Arquitetura Witches Brew

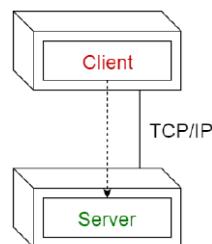
As arquiteturas desenhadas por grupos resultam em ideias complexas todas misturadas e sem visão clara. O problema começa quando os membros da equipa começam a discutir ideias completamente diferentes, e a baralhar tudo.

É fundamental que a equipa recolha o melhor do **conhecimento coletivo** de forma a atingir uma solução comum e consensual, chegando a uma visão unificada para a arquitetura. A arquitetura não deve por isso ser feita em pequenos grupos dentro da equipa, podendo nestes casos levar a uma mistura complexa de ideias sem uma visão clara do que deve efetivamente ser implementado.

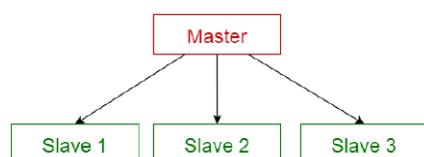
7.4 Padrões de Arquitetura de Software

7.4.1 Os mais simples

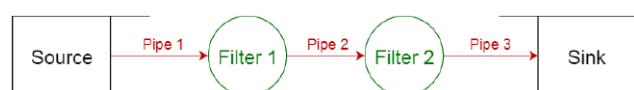
Cliente-Servidor - Esta arquitetura consiste na relação de um para um entre o cliente e o servidor (ex: aplicações online, email).



Master-Slave - Semelhante à anterior no que toca ao número de intervenientes na relação, mas distinta porque no servidor há um relacionamento interno de um para muitos, master/slave. Por exemplo: aplicações multi-threaded ou replicações de bases de dados.

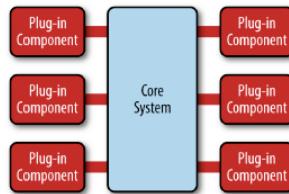


Pipe-filter - É utilizada em processos que exigem buffering (começamos numa fonte de dados e acabamos num consumidor de dados, podendo utilizar filtros pelo meio) ou sincronismo. Por exemplo: compiladores, processos em cascata (streamed), buffering ou sync.



7.4.2 Arquitetura Microkernel

Tem por base dois componentes: a **core** application/system (estável) e os **plugins** (dinâmicos). Novas funcionalidades da aplicação são adicionadas como plugins, que são independentes do core, fornecendo extensibilidade, flexibilidade e isolamento e separação de funcionalidades da lógica da aplicação.

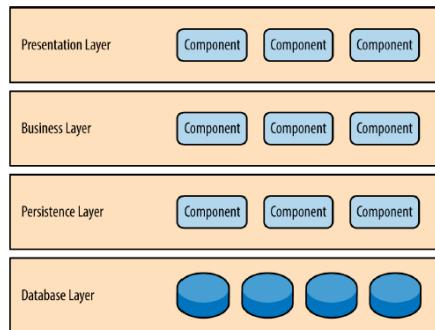


O **core system** geralmente providencia apenas as funcionalidades básicas e estritamente necessárias ao funcionamento do sistema. Deve ainda ter conhecimento dos plugins disponíveis e de como os obter (nome, contratos e protocolos de acesso). Pode ser integrado ou como parte de outro padrão de arquitetura. É uma boa primeira escolha para aplicações baseadas em produtos.

Exemplo: Os IDE fornecem um core system que depois é complementado pelos plugins que lhe adicionam novas funcionalidades.

7.4.3 Arquitetura em camadas (Layered Architecture) (n-tier)

Este é o padrão arquitetural mais comum hoje em dia (standard para a maior parte das aplicações Java EE). Os seus componentes são baseados em camadas horizontais, cada uma com um papel específico na aplicação (separation of concerns).



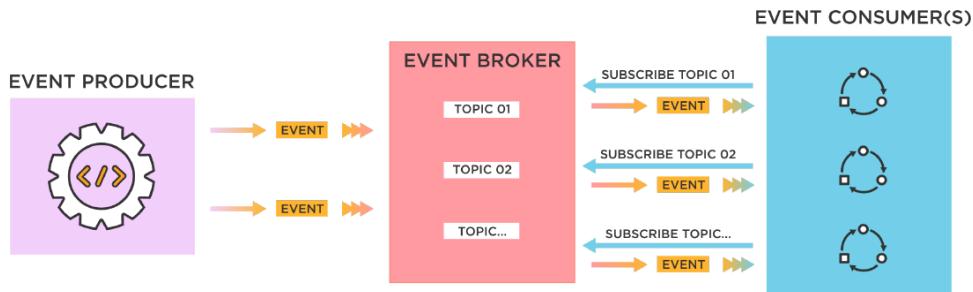
A sua característica principal é a **separação de papéis** que cada camada desempenha, sendo este específico para cada uma, que lida apenas com a lógica adjacente a si mesma. Cada camada forma assim uma camada de **abstração**.

É muito semelhante à organização estrutural encontrada em muitas empresas. É um padrão general-purpose sólido, fazendo com que seja um bom ponto de partida para a maioria das aplicações.

Exemplo: A camada de apresentação não precisa de saber como obter os dados do user, tendo como única preocupação a forma como os vai apresentar, a única lógica que implementa.

7.4.4 Arquitetura Event-Driven

Consiste num padrão de arquitetura **distribuída e assíncrona** utilizado para criar aplicações escaláveis, também conhecido por **message-driven** ou **stream processing**. É composto por componentes individuais desacoplados que recebem e processam eventos assíncronamente.



Este padrão é relativamente **complexo** de implementar, principalmente devido à sua natureza assíncrona distribuída. **Falta de transações atomicas** para apenas um processo de negócios:

- Os componentes do processamento de componentes são altamente desacoplados e distribuídos;
- É muito difícil de manter uma unidade transacional de trabalho através de todos os componentes;

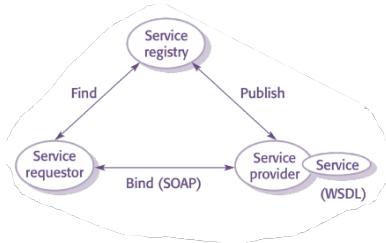
Um aspecto chave é a **criação, manutenção e governação** dos contratos dos componentes processadores de eventos. Útil em eventos com várias etapas e que necessitam de orquestração para serem processados.

Exemplo: Suponhamos que uma pessoa muda de casa e informa a sua seguradora, através de um evento relocation. Os passos envolvidos no seu processamento estão contidos num event broker, como mostrado na figura acima, que por cada passo vai criar um processing event, que é depois enviado para o canal correspondente, onde será processado pelo seu event consumer. O processo continua até que todos os passos estejam concluídos.

7.4.5 Arquitetura Orientada a Serviços (SOA)



Esta arquitetura tem como objetivo desenvolver serviços distribuídos cujos componentes são serviços standalone e podem ser utilizados por outros como serviços prontos a consumir (não como bibliotecas).



As características chave deste tipo de arquitetura são: **contratos standardizados de serviços, independente da linguagem, baixo acoplamento, reutilização e autonomia**.

Exemplo: Na UA o sistema de autenticação é utilizado por todos os serviços associados ao ensino. Foi desenvolvido como um componente standalone e é utilizado por outros serviços como um serviço (neste caso de autenticação) como o PACO, o Elearning, o site do NEI...

Pode ainda ser adicionada uma camada de **abstração** quando há vários componentes para o mesmo serviço com um service registry. Este é contactado pelo cliente para saber qual componente está disponível.

Exemplo: Na UA podiam existir vários sistemas de impressão que trabalhavam de forma complementar para aumentar a redundância. Quando queríamos imprimir contactávamos o registo dos serviços que nos dizia qual o sistema que estava disponível, para o qual era enviado o pedido de impressão.

SOAP (Simple Object Access Protocol)

Este é um protocolo baseado em XML que foi pioneiro na definição de comunicação entre aplicações, independentemente da plataforma linguagem de programação em que são desenvolvidos.

Serviços Web baseados em SOAP/WSDL: Por ser uma solução muito abrangente e demasiado estandardizada (tem estrutura pré-definida) foi considerada demasiado pesada e pouco eficiente.

❖ The structure of SOAP messages:

- An Envelope identifying the XML document as a SOAP msg
- A Header element that contains header attributes
- A Body containing call and response information
- A Fault element containing errors and status information

```
<?xml version="1.0"?>
<soap:Envelope
    xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
    soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
    <soap:Header>
        ...
    </soap:Header>
    <soap:Body>
        ...
        <soap:Fault>
            ...
            </soap:Fault>
        </soap:Body>
    </soap:Envelope>
```

REST (Representational State Transfer)

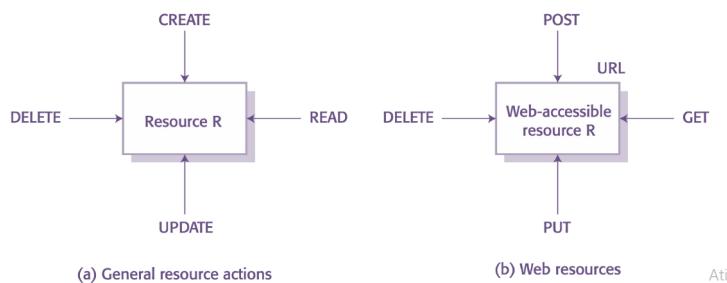
Para simplificar as comunicações foi criado o REST, que com menos estandardização torna a comunicação entre processos mais simples. É um estilo arquitetural baseado na **transferência de representações de recursos** de um servidor para um cliente. É bem mais simples do que SOAP/WSDL para implementar serviços web.

Os serviços RESTful envolvem uma carga (overhead) mais pequena do que as chamadas "big web services".

Tem por base a utilização de **endpoints HTTP**, através de pedidos GET, POST, DELETE, PUT, através dos quais são trocadas mensagens em formato JSON. Fornece uma forma de comunicação sem estado, mais simples e mais leve.

Um elemento fundamental nesta arquitetura são os **recursos** (catálogo, registo médico ou documentos, ...) (múltiplas representações, podendo existir em formatos diferentes (JSON, TXT, ...)) sobre os quais são feitas as operações CRUD (Create, Read, Update, Delete).

- ❖ Create – bring the resource into existence
- ❖ Read – return a representation of the resource
- ❖ Update – change the value of the resource
- ❖ Delete – make the resource inaccessible

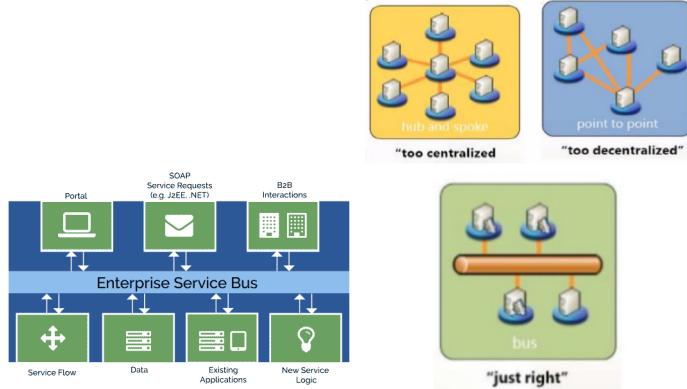


Nota: O create é feito pedido pedido POST, o read pelo GET, o update pelo PUT e o delete pelo DELETE.

Aprendemos no entanto algumas desvantagens, nomeadamente a **dificuldade em representar recursos/interfaces complexas**, a **necessidade de documentar** as interfaces para as entender, uma vez que não existe descrição standard, e ainda a **necessidade de implementar sistemas adicionais** para monitorizar e gerir a qualidade de um serviço e a fiabilidade do serviço (ex: REST não coloca qualquer tipo de segurança como o SOAP).

Topologia SOA

Esta arquitetura era inicialmente para desenvolver aplicações monolíticas, mas evoluiu a partir de então. Muitas organizações utilizam o SOA para resolver complexidades de integração, através de **Enterprise Service Bus (ESB)** (middleware de mensagens).



Algumas características chave do **ESB** são: desenvolvimento streamlined, suporta várias estratégias de binding, realiza várias transformações dos dados, routing inteligente, monitorização em tempo real, tratamento de exceções e serviço de segurança.

7.4.6 Arquitetura de Microserviços

Este padrão evoluiu de **2 fontes principais**:

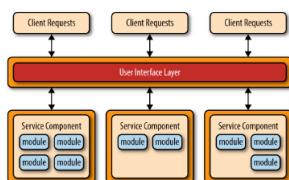
- aplicações desenvolvidas utilizando o padrão por camadas;
- aplicações distribuídas desenvolvidas utilizando uma arquitetura orientada a serviços (SOA);

Procura simplificar o seu desenvolvimento através da sua divisão em pequenos serviços.

O principal conceito deste padrão é a noção de **deploy de unidades separadas**. Estas unidades são componentes que prestam um serviço (deployed como uma unidade separada) e podem consistir num pequeno módulo ou numa grande porção da aplicação. Permitem uma deployment mais fácil e desacoplamento.

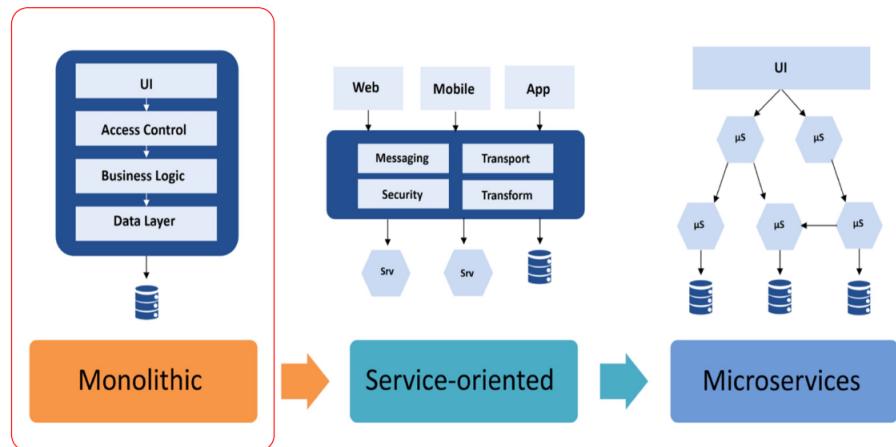
Em qualquer um dos cenários deve ser assegurada uma **arquitetura distribuída**, para a qual é fundamental o desacoplamento total entre os componentes e a definição do seu protocolo de acesso (REST, SOAP, JMS, ...). O grande desafio é atingir o grau certo de granularidade.

Nota: Se for necessário orquestrar ou passar mensagens entre os componentes na camada da interface ou API, então o serviço está demais granulado. Caso esta não consiga ser reduzida (ou seja, por mais que tentemos a orquestração não deixa de ser necessária), então este não será o padrão adequado para desenvolver a nossa aplicação.



As aplicações são geralmente mais robustas, fornecem melhor escalabilidade, e pode suportar mais facilmente a entrega contínua. A capacidade de realizar deployments de produção em real-time. Apenas os componentes do serviço que mudam precisam de ser deployed.

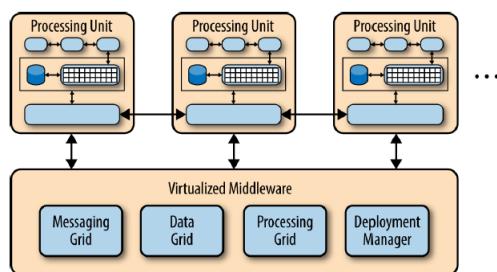
SOA vs Microservices



7.4.7 Arquitetura Space-Based

A maioria das aplicações web gera os pedidos num Fluxo servidor → aplicação → base de dados, fornecendo uma resposta em tempo útil para um fluxo normal de acesso. No entanto, quando o fluxo começa a crescer começamos a assistir a um aumento do contrangimento do tráfego. Inicialmente no servidor, que pode ser expandido facilmente e sem grandes custos, mas que faz com que se comece a sentir limitações na aplicação, esta mais difícil e cara para expandir e por fim na base de dados, que também requer um grande investimento para a sua expansão.

A **space-based architecture**, também conhecido por **cloud architecture** procura então resolver os problemas de **escalabilidade** e **concorrência**, consistindo numa melhor alternativa à de tentar escalar os elementos do referido fluxo. É uma boa escolha para aplicações com load variável (ex: redes sociais, sites de leilão, ...).



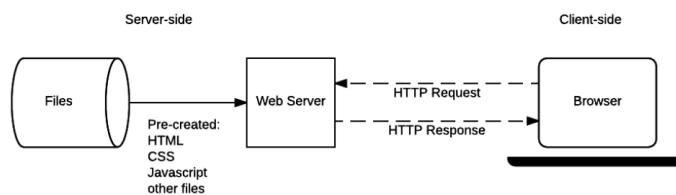
O nome deste padrão vem do conceito de **tuple space**, um paradigma de memória partilhada distribuída que implementa através da remoção da base de dados central, criando uma rede de dados replicada em memória.

Cada **unidade de processamento** vai então ter uma cópia da base de dados, pelo que estas podem ser inicializadas e desligadas dinamicamente de acordo com a variação do tráfego de acesso.

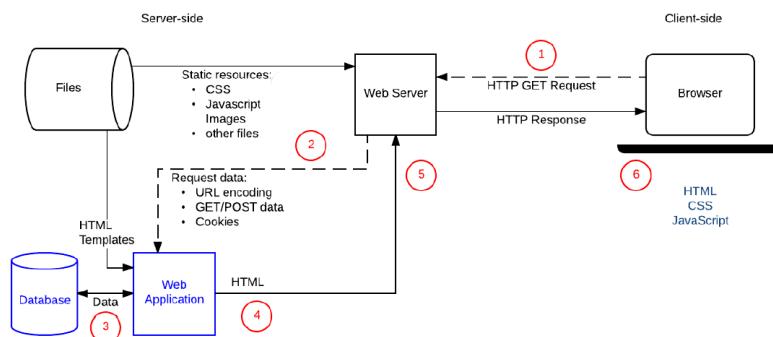
8 Web Frameworks - Spring

Até há uns anos atrás, os programas corriam apenas localmente nos computadores. Hoje em dia é muito diferente, as aplicações web têm um papel central nas aplicações que utilizamos diariamente.

As páginas web evoluíram de versões **estáticas**, que retornam conteúdos hardcoded armazenado no servidor quando um recurso particular é requisitado.



Nas **dinâmicas**, o conteúdo das páginas é gerado dinamicamente, apenas quando necessário, normalmente introduzindo informação de bases de dados em templates HTML.



Um site dinâmico pode retornar dados diferentes para um URL, baseado na informação fornecida pelo user ou preferências armazenadas e pode realizar outras operações como parte de retornar uma resposta (ex: mandar notificações).

A maior parte do código que suporta o website dinâmico corre num servidor web. Criar este código é conhecido como **server-side programming** ou **backend programming**.

Para que serve o **backend**? Como sabemos, os programas executam código binário, e também texto (linguagens interpretadas), mas o browser não consegue entender binário, apenas HTML e alguns outros scripts (ex: JavaScript). O backend terá como função gerar (ou obter de uma base de dados) dados e enviar para o client-side (browser) em HTML.

8.1 Frameworks

Nos anos 2000 começaram a surgir finalmente as frameworks, que quer sejam focadas no lado do servidor ou do cliente, são um **design reutilizável** de todas as partes do sistema que é representado por um conjunto de classes abstratas e na forma que as suas intâncias interagem (estrutura). Fornecem um **esqueleto** de uma aplicação, que pode ser **customizado** pelo programador.

Exemplo: A framework **Spring** permite gerar o esqueleto para um projeto Maven e com alguns cliques no Spring Initializr ter uma aplicação a dar Hello World sem ter de escrever qualquer linha de código.

Desta forma evita a "reinvenção da roda", evitando a escrita do código comum a vários projetos cada vez que se cria um novo.

Uma framework fornece uma fundação de software genérica, sobre a qual **código personalizado específico para a aplicação** pode ser escrito. Normalmente incluem múltiplas bibliotecas, juntamente com ferramentas e regras de como estruturar e usar os componentes.

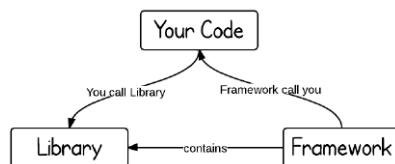
As frameworks diferem de classes de outras bibliotecas através da reutilização do design de alto nível.

- As bibliotecas são usadas pelo código específico da aplicação para suportar funcionalidades específicas;
- As frameworks controlam o flow da aplicação e chamam o código específico da aplicação;

Frameworks e Bibliotecas

Bibliotecas: Chamam as funções na API para fazerem coisas por nós;

Frameworks: "Don't call us, we'll call you", containers de inversão de controlo;



8.1.1 Vantagens

- Poderosas e complexas;
- Velocidade de implementação;
- Soluções testadas e comprovadas;
- Acesso a soluções off-the-shelf;
- Manutenção (updates,...);

8.1.2 Desvantagens

- Difícil de aprender;
- Performance mais baixa;
- Independência reduzida;
- Dependência de entidades externas;
- Lock-in tecnológico;

8.2 Client-Side Frameworks (Frontend)

As mais utilizadas são o **Angular**, **React** e **Vue.js** e permitem várias funcionalidades.

- **Manipulação de documentos** carregados pelo browser - O DOM (Document Object Model) permite a manipulação de um documento junto com HTML e CSS;
- **Atualizar dados (Fetch)** - Através de chamadas assíncronas ao servidor (ex: AJAX);
- **Desenhando e manipulando gráficos** - complexo com desenho no navegador;
- **Armazenamento Client-Side** - através de "bases de dados" locais (ex: IndexedDB)

8.3 Server-Side Frameworks (Backend)

Componentes Core:

- **Request Routing** - Faz a ligação entre os HTTP requests e o código;
- **Data Access** - Acesso a dados uniforme, mapeamento e configuração;
- **Template Engine** - Estruturar e separar a apresentação da lógica;

Componentes Comuns:

- **Security** - Proteção contra ataques web comuns;
- **Sessions** - Gestão de sessões e configurações;
- **Error Handling** - Capturar e gerir erros ao nível da aplicação;
- **Scaffolding** - Gerar interfaces de CRUD rapidamente, baseadas em modelos de dados;

Por exemplo, em **SpringBoot**, O Request Routing é feito utilizando **RequestMapping**

8.3.1 Acesso aos Dados

Independência Lógica: A habilidade de alterar o esquema lógico sem alterar o esquema externo ou programmas de aplicação

- Pode adicionar novos campos, novas tabelas, sem alterar as views;
- Pode alterar estruturas de tabelas sem alterar as views;

Independência Física: A habilidade de alterar o esquema físico sem alterar o esquema lógico

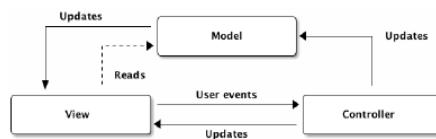
- O tamanho do armazenamento pode mudar;
- O tipo de alguns dados pode mudar para razões de otimização;

Solução: Manter a **VIEW** (o que o user vê) independente do **MODEL** (domínio do conhecimento).

Model-View-Controller (MVC)

Muitas frameworks de backend seguem, ou são uma evolução de um padrão de design chamado **Model-View-Controller (MVC)**.

- Separa a funcionalidade core da apresentação e controlo da lógica que utiliza esta funcionalidade;
- Permite que múltiplas views partilhem o mesmo modelo de dados;
- Torna suportar múltiplos clientes mais fácil de implementar, testar e manter;



Model: Representação por parte do programa do objeto/base de dados (Users, Products, Cars,...);

View: A representação visual dos dados. É aquilo que o user da web app vê (HTML, CSS, JavaScript);

Controller: É o go-between entre o Model e a View.

- One way: pega no input da View e dá a ao Model para este poder agir;
- Other way: Vai buscar os dados dos Model(s) e dá os à View para incorporar na representação visual;
- **Exemplo:** Novo signup de um user, listas de produtos que são menos do que 10€, redireciona o user.

8.3.2 Template Engine

As Template Engines pegam nas strings tokenizadas e produzem rendered strings com valores nos locais do token como output. Estes templates são tipicamente usados como um formato imediato por desenvolvedores para, programaticamente, produzir um ou mais output formats desejados. Normalmente, HTML, XML ou PDF (ex: Java Server Pages, Thymeleaf).

8.4 Spring Framework

- Arquitetura
- Anotações
- Inversion of Control (IoC)
- Dependency Injection (DI)
- Beans
- Aspect-Oriented Programming (AOP)

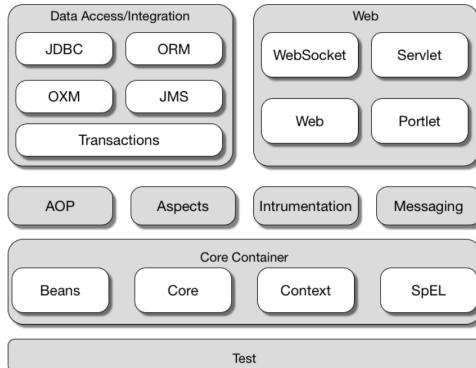
Spring não foi a primeira framework a tentar resolver os problemas relacionados com a construção de aplicações empresariais em Java. **J2EE ou Java EE (Enterprise Edition)** que deu origem ao Spring tentou fazer a mesma coisa, fornece uma boa separação entre várias partes da aplicação.

No inicio dos anos 2000, surgiu a **Spring Framework**, desde então, cresceu de ser um (já sofisticado) container de injeção de dependências para um **ecossistema de frameworks que cobre vários casos de uso** no desenvolvimento de uma aplicação.

Coexiste com a **JEE** (Java Enterprise Edition), mais tarde **Jakarta EE**. Apesar de definirem especificações distintas, a Spring implementa algumas do EE.

Nota: Para cada interface o JEE define **JSR - Java Specification Request**, que estabelecem a interface para cada funcionalidade.

8.4.1 Arquitetura



A arquitetura Spring divide-se em vários componentes como podemos verificar.

O **core container** tem quatro módulos, que são a base de qualquer aplicação:

- **Spring Core** - Fornece implementação para funcionalidades como IoC (Inversion of Control) e DI (Dependency Injection);
- **Spring Beans** - Este módulo fornece implementações do padrão de fábrica do BeanFactory;
- **Spring Context** - Contrói em cima do Core e Beans e fornece uma forma de aceder a objetos definidos e fornece suporte a interações third-party como caching, mailing, and template engines. A interface ApplicationContext é a a core part deste módulo;
- **Spring Expression Language (SpEL)** - Permite aos utilizadores usarem a String Expression Language para aceder e manipular o gráfico de objetos durante a runtime;

8.4.2 Anotações

No processo de compilação de um programa é feita uma análise sintática para verificar que o código cumpre com uma determinada estrutura. No entanto, há situações em que queremos adicionar verificações adicionais, ou mesmo alterar o código de forma sistemática em determinados trechos durante o processo de compilação.

Para responder a estes cenários e de forma a evitar a complexidade da alteração ao código, metadados que dão informação sobre o código ao compilador e que podem ser considerados por outro desenvolvido por nós para esse fim.

Facilita a deteção de erros ou supressão de avisos, por exemplo. Podem ter informação para o compilador (detetar erros e suprimir warnings), processamento compile-time deployment-time, e processamento runtime. **As Anotações são muito utilizadas em na Spring Frameworks.**

Exemplos: De anotações interpretadas pelo Java Compiler são o @Override, @Deprecated. Caso num programa utilizemos um método de uma classe anotado como @Deprecated, quando compilarmos este programa o compilador irá mostrar-nos o aviso de que estamos a utilizar um método obsoleto.

8.4.3 Inversão de Controlo (IoC)

Quando trabalhamos com uma framework, como esta fornece a maioria das funcionalidades necessárias ao funcionamento de uma aplicação web, a framework passa a ser a responsável pela **criação dos objetos**.

IoC é um processo em que definimos uma classe, mas não criamos os objetos (apenas definimos como devem ser criados pelo IoC container).



O atributo name da propriedade no XML deve ser mapeado `getName` e `setName` na classe do objeto a criar com Name o valor de outro atributo.

No exemplo acima para os atributos `id` e `name` devem haver os métodos `getId()` e `setId()` e `getName()` e `setName()`.

8.4.4 Injeção de Dependências (DI)

Quando um objeto tem como atributo outro objeto, vai depender dele, sendo por isso necessário criá-lo antes do primeiro. A **injeção de dependências** é um padrão que remove as dependências do código. Podemos especificar as dependências das classes através de **anotações** ou ficheiros externos como um XML.

A injeção pode ser através de um construtor ou de um seller.

POJO (Plain Old Java Object)

The diagram shows a code snippet for `MyService` and `Main` classes. `MyService` has a private final attribute `repository` and a constructor that takes a `MyRepository` parameter. It also has a `setRepository` method. `Main` class creates instances of `JpaRepository` and `MyService` and sets the repository for the service.

`MyService` expresses a dependency to `MyRepository` instead of creating it itself:

- by constructor
- by setter method

The manual setup code creates an instance of `MyRepository` to be passed to the service instance to be created.

```

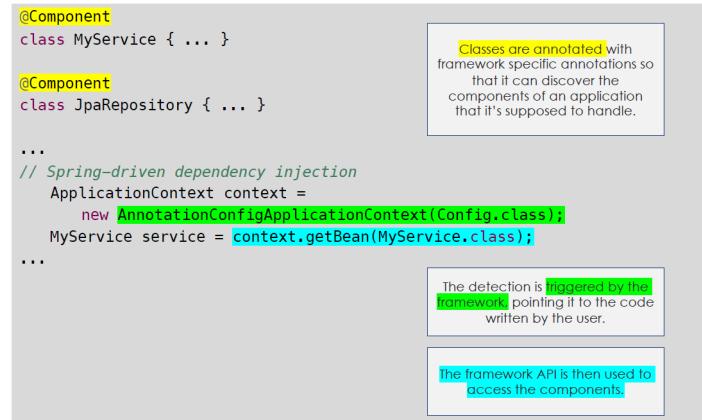
interface MyRepository {
    MyEntity save(MyEntity entity);
}

class JpaRepository implements MyRepository {
    MyEntity save(MyEntity entity) { ... }
}

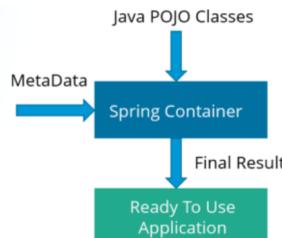
class MyService {
    private final MyRepository repository;
    MyService(MyRepository repository){
        this.repository = repository;
    }
    void setRepository(MyRepository newRepository) {
        this.repository = newRepository;
    }
}
// Manual dependency injection
class Main(String ...) {
    MyRepository repository = new JpaRepository(...);
    MyService service = new MyService(repository);
}

```

Com Spring DI



Em **Spring** as classes identificadas com a anotação `@Component` são geridas de acordo com estes princípios de IoC e DI pelo **Spring Container**. O Spring IoC é o coração da Spring Framework, cada objeto delega o **trabalho de construir** um IoC container. Este recebe metadados a partir de ficheiros XML, anotações Java ou código Java e produz um sistema ou uma aplicação completamente configurado e pronto a correr.



As principais tarefas realizadas pelo IoC container são:

- Instanciar o bean/objeto;
- Escrever os beans juntos (injeção de dependências entre eles);
- Configurar os beans;
- Gerir o ciclo de vida dos beans;

Existem dois tipos de Spring IoC containers (interfaces):

- **BeanFactory** - É o container mais simples, fornece a configuração da framework e funcionalidades básicas;
- **ApplicationContext** - construído em cima da interface BeanFactory, adiciona mais funcionalidades enterprise-specific (várias interfaces para implementações distintas);

8.4.5 Beans

São definidas pela documentação Spring como os objetos base da aplicação (backbone), que são geridos pelo Spring IoC Container (desde a instanciação e construção até à gestão).

Algumas características chave dos beans são:

- Todos os campos devem ser **privados**;
- Todos os campos devem ter **getters e setters**;
- Devem ter um **construtor vazio**;
- Os campos são acessados exclusivamente através dos getters/setters e construtor;

O Spring é responsável por criar os objetos bean, mas, primeiro, temos de dizer à framework quais são os objetos que queremos que sejam criados. As **Bean definitions** dizem ao Spring quais classes a framework deve usar como beans. As Bean definitions são como receitas (também descrevem as propriedades do bean).

Existem **3 formas diferentes** de definir um Spring Bean: declarar a definição num **ficheiro XML configurável**, anotando uma classe com a anotação **@Component** (ou derivadas como **@Service**, **@Repository**, **@Controller**) ou escrevendo um bean factory method anotado com a anotação **@Bean** num class Java personalizada de configuração.

Hoje em dia apenas se usam os dois últimos, mas o outro ainda é permitido para compatibilidade com versões anteriores.

A **escolha entre os métodos de bean definition** é principalmente ditada pelo acesso ao código fonte da classe que queremos usar como um Spring Bean.

A anotação **@Component** é utilizada quando **temos acesso ao código fonte**. Quando esta só tem um construtor, o Spring considera os seus parâmetros como a lista de dependências obrigatórias. Caso haja mais do que um, devemos anotar o que define as dependências com **@Autowired**.

No runtime, o Spring vai procurar por classes anotadas com **@Component** e vai usá-las como definições de bean. O processo de procurar por classes anotadas chama-se **component scanning**.

Quando **não temos acesso ao código fonte** da classe, definimos uma classe com um método de fábrica anotado com **@Bean**. Esta classe por sua vez é anotada com **@Configuration**, que marca a classe como sendo um container de definições **@Bean**. Podem ser definidos vários métodos de fábrica dentro da mesma classe de configuração.

```
@Configuration  
class MyConfigurationClass {  
  
    @Component  
    class MySpringBeanClass {  
        //...  
    }  
  
    @Bean  
    public static NotMyClass not MyClass() {  
        return new NotMyClass();  
    }  
}
```

8.4.6 Spring Expression Language (SpEL)

Pode ser usada para manipular e aceder a um gráfo de objetos durante a runtime. SpEL está disponível via XML ou anotações e é avaliado durante o tempo de criação do bean.

Possui vários operadores disponíveis como: aritméticos (+, -, *, /, %), relacionais (==, !=, i, i, i=, i!=), lógicos (and, or, not), condicionais (?:), regex (matches).

```

❖ 1: Create employee.properties file in the resources' directory.
employee.names=Petey Cruiser,Anna Sthesia,Paul Moline,Buck Kinar
employee.type=contract,fulltime,external
employee.age={one: '26', two : '34', three : '32', four: '25'}
❖ 2: Create a class EmployeeConfig as follows:
@Configuration
@PropertySource (name = "employeeProperties",
    value = "employee.properties")
@ConfigurationProperties
public class EmployeeConfig {
}
❖ SpEL: The @Value annotation can be used for injecting values into fields in Spring-managed beans.
- it can be applied at the field, constructor, or method parameter level.
@Value ("#{${employee.names}.split(',')[0]}")
private List<String> employeeNames;
    [Petey Cruiser, Anna Sthesia, Paul Moline, Buck Kinar]
    @Value ("#${employee.names}.split(',')[0]")
private String firstEmployeeName;
    @Value ("#{systemProperties['java.home']}")
private String javaHome;
    @Value ("#{systemProperties['user.dir']}")
private String userDir;
    @Value("#{someBean.someProperty != null ? someBean.someProperty : 'default'}")
private String ternary;

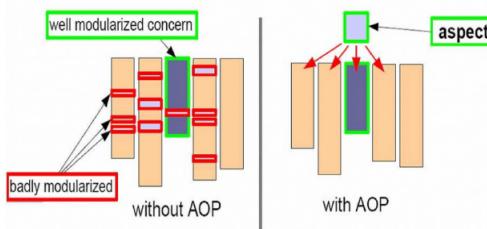
```

A anotação **@Value** permite a injeção de valores nos atributos nos beans

8.4.7 Aspect-Oriented Programming (AOP)

É uma abordagem de programação que permite a propriedades globais de um programa determinarem como é que o código é compilado para um programa executável.

É um paradigma complementar à Programação Orientada a Objetos e que tem como objetivo a separação das responsabilidades de forma a melhorar a modularidade.



AOP divide a lógica do programa em partes distintas chamadas **concerns**. Isto aumenta a modularidade através de **cross-cutting concerns**. Um **cross-cutting concern** é um concern que afeta toda a aplicação e é centralizado num único local.

AOP pode ser considerado como um padrão de design **decorador dinâmico** (o padrão decorador permite que um novo comportamento seja adicionado a uma classe já existente, dando wrap à classe original e duplicando a sua interface e depois delegar o original).

Conceitos AOP Core

- **Aspect** - Uma modularização de um concern que corta múltiplas classes, pode ser uma classe normal, configurado através de configuração XML ou anotações com **@Aspect**;
- **JoinPoint** - Um ponto durante a execução do programa, onde o fluxo de execução foi alterado devido à exception catching ou a chamada de um método;
- **Pointcut** - São basicamente JoinPoints, mas com advice (ou call aspect). Expressões que são correspondidas em join points para determinar se o advice deve ser executado ou não;
- **Advice** - A ação tomada por um aspect num join point particular (@Before, @After, @Around, @AfterReturning, @AfterThrowing);

```

package foo
import org.aspectj.lang.*;
import org.springframework.util.StopWatch;
import org.springframework.core.annotation.Order;
@Aspect
public class ProfilingAspect {
    @Around("methodsToBeProfiled()") // action taken at the join point.
    public Object profile(ProceedingJoinPoint pjp) throws Throwable {
        StopWatch sw = new StopWatch(getClass().getSimpleName());
        try {
            sw.start(pjp.getSignature().getName());
            return pjp.proceed();
        } finally {
            sw.stop();
            System.out.println(sw.prettyPrint());
        }
    }
    @Pointcut("execution(public * foo..*,*(..))")
    public void methodsToBeProfiled(){}
}

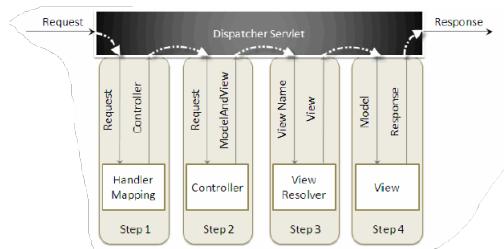
```

8.4.8 Spring Web

Este componente aplica a arquitetura Model-View-Controller (MVC).

Quando recebe um pedido HTTP, o DispatcherServlet consulta o **HandlerMapping** para invocar o **Controller** apropriado. Este pega mo pedido e encaminha-o para o método adequado de processamento (GET ou POST) (o método vai dar set ao modelo de dados baseado em lógica de negório definida e retorna o nome da view para o DispatcherServlet).

De seguida, o DispatcherServlet pede ajuda ao **ViewResolver** para ir buscar a view definida para o pedido. Quando a view wstiver finalizada, o DispatcherServlet envia o **modelo de dados para a view** que é finalmente renderizada no browser.



```

@Controller
@RequestMapping("/funcionarios") // a kind of path..
public class FuncionariosController {
    @Autowired
    private FuncionarioService funcionarioService;
    @Autowired
    private Funcionarios funcionarios;

    @ResponseBody
    @GetMapping("/todos") // curl -i http://localhost:8080/iesapp/funcionarios/todos
    public List<Funcionario> todos() {
        return funcionarios.findAll();
    }
    @GetMapping("/lista")
    public ModelAndView listar() {
        ModelAndView modelAndView = new ModelAndView("funcionario-lista.jsp");
        modelAndView.addObject("funcionarios", funcionarios.findAll());
        return modelAndView;
    }
    ...
}

```

8.4.9 Sprint Data Access Modules

Há vários módulos que permitem criar a abstração entre os dados e o seu armazenamento.

JDBC: Permite trabalhar diretamente com SQL. A camada abstrata elimina a necessidade de overhead de exception handling;

ORM (Object Relational Mapping): Ao utilizarmos o ORM temos uma maior abstração, uma vez que trabalhamos sobre objetos Java, sendo responsabilidade do módulo mapear as alterações na base de dados. Fornece consistência/portabilidade para o código;

OXM (Object XML Mapping): Paradigma de cima mas sobre XML. Converte os objetos para o formato XML e vice-versa. A Spring OXM fornece uma API uniforme para acessar qualquer framework OXM;

JMS (Java Messaging Service): Facilita a interação com serviços de mensagens. Tem funcionalidades para produzir e consumir mensagens entre vários clientes;

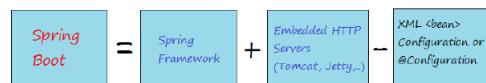
Transaction: Facilita a implementação dos conceitos de transações ao nível da organização. suporta gestão de transações declarativa e programática para classes que implementam interfaces especiais e para todos os POJOs. Todas as transações ao nível enterprise podem ser implementadas pelo Spring utilizando este módulo;

9 Spring Boot

Resumidamente, a Spring Framework fornece uma **infraestrutura de suporte para o desenvolvimento de aplicações Java**. Tem muitas funcionalidades, como a Injeção de Dependências e módulos como o Spring MVC, Spring JDBC, Spring Security, Spring ORM, Spring AOP, Spring Test.

Estes módulos **podem reduzir o tempo de desenvolvimento** de uma aplicação.

O **Spring Boot** é uma estensão da Spring Framework, que tem como objetivo eliminar a necessidade de definir as configurações padrão para iniciar uma aplicação Spring e fornecer uma estrutura de projeto padrão, para um desenvolvimento mais rápido e eficiente. Algumas funcionalidades são: "starter" dependencies para simplificar a build e as configurações da aplicação, servidor embutido para evitar complexidade e configurações automáticas para a funcionalidade Spring.



Os objetivos principais do Spring Boot são:

- **Redução do tempo de desenvolvimento**, juntamente com tempo de Unit Test e Integration Test;
- Evita completamente a necessidade de **configuração XML**;
- Fornece uma **anotação simples** (baseada nas do Spring);
- Evita escrever vários imports;
- Abordagem **opinionated development**, de forma a fornecer algumas configurações padrão para começar um novo projeto rapidamente;

Os principais componentes do Spring Boot são:

- **Spring Initializr** - Interface Web para dar um quick start no desenvolvimento de aplicações Spring Boot;
- **Starters** - Combina um grupo de dependências comuns e relacionadas numa só, permitindo adicionar jars no classpath;
- **AutoConfigurator** - Reduz a configuração Spring, tenta configurar automaticamente a aplicação Spring baseado nas dependências jar;
- **CLI** - Permite correr e testar aplicações Spring Boot na linha de comandos;
- **Actuator** - Fornece endpoints e métricas;

Os seus projetos são iniciados em <https://start.spring.io/>, onde é possível selecionar várias dependências e obter um projeto pré-configurado e com a estrutura padrão.

A aplicação principal consiste num esqueleto padrão, onde a classe principal é passada à SpringApplication para esta a executar. Verifica-se a **inversão de controlo**.



A anotação **@SpringBootApplication** é uma combinação de três anotações:

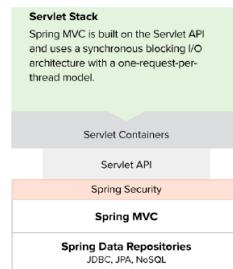
- **@Configuration** - Designa a classe como sendo de configuração, utilizando a configuração Spring;
- **@ComponentScan** - Indica que a aplicação deve fazer scan no pacote atual e sub-pacotes de modo a encontrar classes anotadas com @Component, @Service, @Repository, @Controller, configurando-as como beans no contexto da aplicação;
- **@EnableAutoConfiguration** - Ativa a auto-configuração, evitando a escrita de páginas XML para este fim

Por exemplo, se quisermos desenvolver uma aplicação Spring WebApplication com Tomcat, precisavamos de adicionar várias dependências, no entanto, com o Spring Boot, basta adicionar a dependência **spring-boot-starter-web**. Esta pode ser adicionada no ficheiro pom.xml ou automaticamente através do Spring Initializr.

9.1 Arquitetura Spring Web MVC (Model-View-Controller)

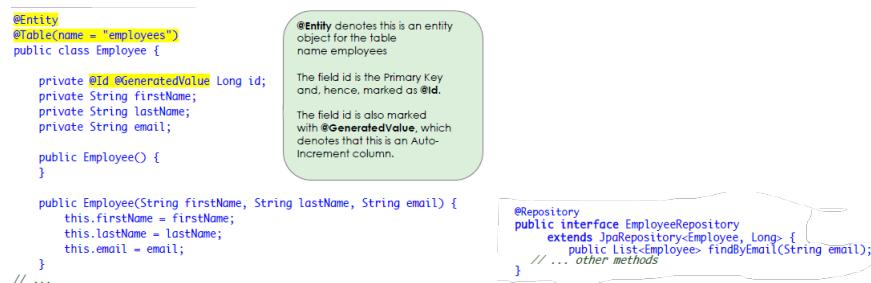
Este componente é responsável por identificar quais as classes que vão trabalhar com os dados e quais vão atender pedidos do navegador, funcionando como a ponte que os liga.

Permita a criação de beans especiais, **@Controller** e **@RestController**, responsável por capturar os pedidos HTTP e encaminhá-los para as funções responsáveis por cada um (**@RequestMapping**).



O **modelo de dados** é definido através de uma típica classe Java, com atributos privados com getters e setters e um construtor vazio por defeito. Podem-lhe ser associadas anotações como `@Entity`, `@Table(name=X)` e aos seus atributos `@Id` ou `@GeneratedValue`.

Os **repositórios** definem interfaces que fazem a ligação entre as entidades do modelo de dados e a base de dados. A sua definição é bastante simples e consiste na extensão de um JpaRepository, a partir da qual são herdados os métodos de manipulação de dados CRUD. Podem no entanto ser definidos métodos adicionais para queries mais complexos.



Os métodos adicionais têm de seguir as convenções impostas pelo Spring Boot, uma vez que continuam a ser configurados automaticamente.

No exemplo acima, apesar de não ser um método fornecido por defeito, o `findByEmail(String email)` tem de ser declarado desta forma, uma vez que o Spring Boot vai procurar na classe `Employee` pelo atributo com o nome que segue "findBy" e se esse nome não existir, dará erro.

Spring Data JPA suporta **find**, **read**, **query**, **count**, **get**, por exemplo, podemos fazer `queryByName` e o Spring Data teria se comportado da mesma forma.

Também podemos usar **Distinct**, **Top**, **First** para remover duplicados, ou limitar o result set: `findTop3ByFirstName(String firstName)`.

O **controlador**, para além de anotado como tal através da anotação @RestController, é também mapeado para um determinado caminho na aplicação com @RequestMapping("/api").

Cada um dos seus métodos é também mapeado para um caminho relativo ao anterior, com @GetMapping("/abc"). Os seus parâmetros são anotados com @RequestParam(required=X) no caso de serem enviados através do URL ou @RequestBody no caso de serem enviados no corpo do pedido HTTP.

```
@RestController
@RequestMapping("/api/v1")
public class EmployeeController {
    @Autowired
    private EmployeeRepository employeeRepository;

    @GetMapping("/employees")
    public List<Employee> getAllEmployees(
        @RequestParam(required = false) String email,
        @RequestParam(required = false) String lastname) {
        if (email != null)
            return employeeRepository.findByEmail(email);
        else if (lastname != null)
            return employeeRepository.findByLastName(lastname);
        else
            return employeeRepository.findAll();
    }
    // ...
}

@GetMapping("/employees/{id}")
public ResponseEntity<Employee>
getEmployeeById(@PathVariable(value = "id") Long employeeId)
throws ResourceNotFoundException {
    Employee employee = employeeRepository.findById(employeeId)
        .orElseThrow(() -> new ResourceNotFoundException("Employee not
found for this id :: " + employeeId));
    return ResponseEntity.ok().body(employee);
}

@PostMapping("/employees")
public Employee createEmployee(@Valid @RequestBody Employee employee) {
    return employeeRepository.save(employee);
}
// ...
```

A anotação dos métodos varia dependendo do pedido HTTP a que respondem, podendo ser por exemplo PostMapping, PutMapping...

Definidos o modelo dos dados, a interface dos seus repositórios e o controlador que os manipula, falta apenas definir a **ligação da aplicação à base de dados**.

Para terminar o cenário de configuração, podemos usar a base de dados H2 embutida, basta colocar as suas dependências no ficheiro **pom.xml** e configurar o ficheiro **application.properties** com as propriedades da base de dados:

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=user
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
```

A definição dos dados de conexão à base de dados é feita no ficheiro application.properties de forma a desacoplá-la do código. Assim, quando houver necessidade de alterar o end point da base de dados, basta editar este ficheiro, sem necessidade de fazer qualquer alteração e recompilar o código.

Por fim, a aplicação pode ser **executada** facilmente, dado o facto de incorporar um servidor HTTP embutido.

```
$ mvn spring-boot:run
```

9.2 Spring WebFlux Framework

Uma nova web framework **reativa** (uma versão paralela da Spring MVC, introduzido na versão 5.0), é um modelo **assíncrono**, em que o sistema **não bloqueia** à espera de resposta, e implementa a especificação Reactive Streams (Reactor lib). Esta especificação serve para sistemas onde muitos eventos são gerados e consumidos assincronamente. Existem duas versões: funcional e baseada em anotações. A última é bastante parecida com a Spring MVC.

Permite que uma página seja refrescada continuamente sem ter uma resposta da base de dados para um determinado pedido que foi feito.

```

@RestController
@RequestMapping("/users")
public class MyRestController {
    @GetMapping("/{user}")
    public Mono<User> getUser(@PathVariable Long user) {
        // ...
    }
    @GetMapping("/{user}/customers")
    public Flux<Customer> getUserCustomers(@PathVariable Long user) {
        // ...
    }
    @DeleteMapping("/{user}")
    public Mono<User> deleteUser(@PathVariable Long user) {
        // ...
    }
}

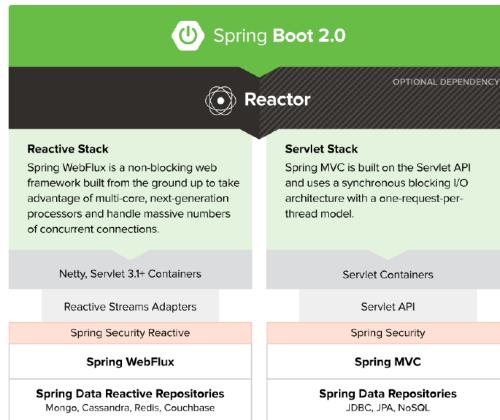
@Configuration(proxyBeanMethods = false)
public class RoutingConfiguration {
    @Bean
    public RouterFunction<ServerResponse> monoRouterFunction(UserHandler userHandler) {
        return route(GET("/{user}")).and(accept(APPLICATION_JSON), userHandler::getUser)
            .andRoute(GET("/{user}/customers"))
            .and(accept(APPLICATION_JSON), userHandler::getUserCustomers)
            .andRoute(DELETE("/{user}"))
            .and(accept(APPLICATION_JSON), userHandler::deleteUser);
    }
}

@Component
public class UserHandler {
    public Mono<ServerResponse> getUser(ServerRequest request) {
        // ...
    }
    public Mono<ServerResponse> getUserCustomers(ServerRequest request) {
        // ...
    }
    public Mono<ServerResponse> deleteUser(ServerRequest request) {
        // ...
    }
}

```

9.2.1 Spring MVC vs Spring WebFlux

O MVC é mais utilizado com bases de dados relacionais, enquanto que o WebFlux com NoSQL.



9.3 Spring Data

Este é o módulo que permite a gestão dos dados de uma forma abstrata, permitindo a utilização da grande maioria dos tipos de bases de dados, relacionais e não relacionais.

Para obter persistência dos dados, esta possui 4 camadas: Persistence Layer (Controllers), Business Layer (Services), Persistence Layer (Repositories), Database Layer (Database (SQL ou NoSQL)).

9.3.1 DAO (Data Access Object)

Padrão estrutural que isola as camadas de apresentação e negócio da camada de persistência utilizando **uma API abstrata**. Não é um módulo Spring, mas sim convenções que devem ditar escrever DAO.

```
public interface Dao<T> {  
    Optional<T> get(long id);  
    List<T> getAll();  
    void save(T t);  
    void update(T t, String[] params);  
    void delete(T t);  
}
```

9.3.2 ORM (Object Relational Mapping)

O pacote ORM está relacionado com o acesso à base de dados. Fornece **camadas de integração para API's de mapeamento objeto-relacional populares como JDO e Hibernate**.

Com base nestes dois conceitos foram criados vários projetos, para diferentes tipos de utilização.

Java/Jakarta Persistence API (JPA)

É a especificação padrão para mapear objetos Java a bases de dados relacionais. É uma abordagem possível para **ORM**. Sendo uma especificação, é utilizado em várias implementações como o Hibernate.

Hibernate

É uma ferramenta **ORM** baseada em Java que fornece um framework para mapear um objeto orientado a objetos para uma tabela na base de dados e vice-versa. É uma implementação do **JPA**.

Spring Data Persistence

É um projeto que fornece uma abordagem unificada para a persistência de dados, independentemente do tipo de armazenamento de dados, relacional ou não relacional.



Spring Data JPA

Uma **abstração** utilizada para reduzir significativamente a quantidade de código de implementação necessário para implementar camadas de acesso a dados. Adiciona as próprias funcionalidades como **implementação de repositórios sem código** e **criação de queries de base de dados a partir dos nomes**. Pode também gerar queries JPA através das convenções de nomes.

É novamente uma especificação que é implementada pelo **Hibernate**, associado ao projeto através da dependência spring-boot-starter-jpa.

Temos de criar um repositório que extende a interface JpaRepository, tendo assim acesso as anotações @Query, @Procedure (para chamar stored procedures), @Lock e @Modifying.

O Spring Boot pode fazer a configuração de bases de dados mais simples, como a H2, uma BD in-memory.

Para **mapear uma classe na base de dados**, basta anotá-la com @Entity e eventualmente @Table para definir qual a tabela onde os seus objetos serão armazenados. Cada um dos seus atributos pode ser mapeado com @Id e @GeneratedValue, para especificar uma chave primária com uma determinada estratégia de geração automática, e @Column para mapear atributos para colunas específicas da tabela.

A **ligação entre classes** é feita através da anotação @Embedded. Adicionalmente o atributo pode ser anotado com @AttributeOverride(s) para fazer o mapeamento entre os atributos da sua classe e as colunas para as quais estes vão ser mapeados.

As **relações entre classes** são anotadas com @OneToOne ou @OneToMany para referenciar entidades, @ManyToOne ou @ManyToMany para referenciar conjuntos de entidades e @MappedSuperClass ou @Inheritance para herança entre entidades.

Para fazer queries personalizados, pode ser utilizada a anotação @Query, que permite a escrita de código SQL ou JPQL (JPA Query Language), com funcionalidades de ordenação e até paginação.

– SQL native – over JDBC
@Query(value = "SELECT * FROM USERS u WHERE u.status = 1",
nativeQuery = true)
Collection<User> findAllActiveUsersNative();

– JPQL (JPA Query Language) – over Hibernate
@Query("SELECT u FROM User u WHERE u.status = 1")
Collection<User> findAllActiveUsers();

– JPQL (JPA Query Language) – over Hibernate
@Query(value = "SELECT u FROM User u")
List<User> findAllUsers(Sort sort); Sorting

– JPQL (JPA Query Language) – over Hibernate
@Query(value = "SELECT u FROM User u ORDER BY id")
Page<User> findAllUsersWithPagination(Pageable pageable); Pagination

JPA com MongoDB

Para além de suportar bases de dados relacionais, tem muitos drivers para outros tipos de BD, como o MongoDB. Estes por vezes apresentam ligeiras alterações no modelo, mas de um modo geral, este é bastante semelhante.

```
@Document(collections = "school")
public class Person {
    @Id
    private ObjectId id;
    private Integer ssn;
    @Indexed
    private String name;
}

@Repository
public interface PersonRepository
    extends MongoRepository<Person, String> {
    Person findByName(String name);
}
```

No exemplo acima vemos uma ligeira adaptação, com a anotação @Document em vez de @Table, @Field em vez de @Column e a criação de um MongoRepository em vez de um JpaRepository. No entanto, de resto a sintaxe é bastante semelhante e no fim vamos ter a mesma interface no repositório com os queries CRUD disponíveis sem necessidade de os definir.

10 Microservices

Nos últimos anos, o paradigma de desenvolvimento de aplicações alterou-se de um modelo focado em armazenamento local (seja no desktop ou em servidores), para um baseado na nuvem, sendo hoje estes dois conceitos altamente interligados.

Vários fatores levaram ao conceito de aplicações "cloud native": a disponibilidade geral de **plataformas de cloud computing**, o avanço nas **técnoogias de virtualização** e o aparecimento de **práticas agile e DevOps**, uma vez que as organizações procuram formas de simplificar e encurtar os ciclos de release.

Micro-serviços definem-se então como aplicações nativas da nuvem, compostas por peças pequenas, independentes, self-contained, substituíveis e poliglotas.

Os ambientes de cloud computing são **dinamicos**, com alocação on-demand e libertação de recursos de uma pool de recursos virtualizados partilhados.

Aplicações ou serviços (micro-serviços) são **pouco acoplados**, com dependências explicitas descritas. Aplicações e processos são **corridos em containers de software** como unidades isoladas. Os porcessos são geridos utilizando processos de **orquestração central** de forma a melhorar a utilização de recursos e reduzir custos de manutenção.

10.1 Boas práticas no desenvolvimento

No desenvolvimento deste tipo de serviços, há um conjunto de boas práticas que deve ser seguido. São 12, mas destacam-se as principais. (3, 8 e 10 mais importantes). Designa-se por **12-factor application methodology** e não são específicas do cloud provider, plataforma ou linguagem, são apenas um conjunto de boas práticas para aplicações portáveis e resilientes (especialmente aplicações SaaS).

1. **Codebase:** Deve haver uma associação de um para um entre cada serviço e o seu repositório de código, usando um sistema de versionamento de código, como o Git. Mas uma mesma base de código pode conter vários serviços;
2. **Dependencies:** Os serviços devem declarar todas as suas dependências (não deve ser assumida a presença de ferramentas ou bibliotecas do sistema). Podemos usar Maven para Java, Pip para Python e NPM para Node.js;
3. **Configurations:** As configurações que variem entre ambientes de execução devem ser armazenadas no ambiente (por exemplo variáveis de ambiente). Por exemplo, em Spring Boot, as dependências são declaradas no ficheiro pom.xml e as configurações no ficheiro application.properties;
4. **Backing services:** Todos os seviços de armazenamento são tratados como attached resources e devem ser desacoplados do serviço, sendo possível que este funcione sem o primeiro, ou que seja facilmente substituído por outro. Estes serviços são geridos (attached, detached) pelo **ambiente de execução**;
5. **Build, release, run:** O processo de desenvolvimento deve distinguir e separar as fases de **construção, entrega e execução**;

-
- 6. **Processes:** As aplicações devem ser executadas como um ou mais **processos** independentes (stateless). Os dados persistentes devem ser armazenados num serviço de armazenamento apropriado; Por exemplo uma aplicação que seja composta por vários microserviços, deve ver cada serviços deployed num processo separado e independente, apesar de trabalharem em conjunto.
 - 7. **Port binding:** Serviços auto-contidos devem ser disponibilizados aos restantes através de uma porta;
 - 8. **Concurrency:** A concorrência é executada através da escalada horizontal dos serviços (multiplicação), tirando vantagem do SO;
 - 9. **Disposability:** Os processos devem ser descartáveis, ou seja, inicializar rapidamente e parar de forma responsável, levando a um sistema mais robusto e resiliente. Se tivermos um sistema de escrita de ficheiros, este deve ser inicializado rapidamente para ficar disponível assim que necessário, mas deve também garantir que quando é terminado guarda o buffer de escrita que estava foi introduzido pelo utilizador;
 - 10. **Dev/Prod parity:** Todos os ambientes, desde o desenvolvimento local até à produção, devem ser o mais semelhantes possível. Se acumularmos alterações no Dev, as operações de merge no Main vão ser muito mais complexas;
 - 11. **Logs:** Os processos devem produzir **logs** como fluxo de eventos, e confiar no ambiente de execução para agregar e armazenar os mesmos;
 - 12. **Admin processes:** Se tarefas de **admin** são necessárias, devem ser mantidas no código fonte e empacotadas ao lado da aplicação, para garantir que corre com os mesmo ambiente que a aplicação;

10.2 Microservices

Não existe uma definição padrão para os microserviços, mas a maioria referencia o papel seminal de Martin Fowler, que diz: "Microservices: A new architectural term". Este diz que os microserviços são utilizados para compôr aplicações complexas, estes são: processos **pequenos, independentes** (autonomos) e **substitutivos** que **comunicam** através de uma lightweight API que não depende da linguagem.

Cada microserviço é uma aplicação de 12 fatores (como visto anteriormente), com armazenamento substitutivo que fornecem um message broker, registo de serviços e independentes do armazenaento de dados.

O termo **"pequeno"** significa que este é focado num único propósito. Os microserviços **apenas devem fazer uma coisa e fazê-la bem**.

Independentes e Autonomos significa que deve ser possível parar um componente para manutenção por exemplo sem “partir” os restantes, que podem até dar timeouts nas consultas caso estejamos a trabalhar com uma BD por exemplo. No entanto, quando este voltar a ser inicializado, as restantes devem voltar ao seu funcionamento normal sem necessidade de intervenção.

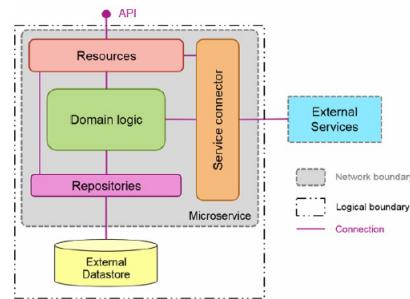
Substitutivos, porque deve ser possível alterar o funcionamento interno de cada componente, ou mesmo substituí-lo por um novo, desde que este mantenha as suas funcionalidades, ou seja, a mesma interface de acesso (API).

Poliglotas, porque dada a independência dos micro-serviços de uma aplicação, cada um destes componentes pode ser desenvolvido numa linguagem de programação diferente, sem prejuízo para as restantes, desde que todas comuniquem de acordo com um protocolo de comunicação agnóstico da linguagem. Isto é muito poderoso e eficiente, mas apenas é possível com protocolos language-agnostic.

Representational State Transfer (REST) é um padrão de arquitetura que define guidelines para criar interfaces uniformes que separam os representações dos dados "on-the-wire" da implementação do serviço. Estas arquiteturas requerem que o request state seja mantido pelo cliente, permitindo que o servidor seja stateless.

10.3 Estrutura Interna

Um micro-serviço é como uma pequena aplicação, uma vez que tal como estas vai trabalhar num domínio do problema, expôr recursos através de uma determinada interface de acesso (API), aceder a serviços externos e aceder a bases de dados externas. Geralmente cada micro-serviço tem a si associada uma base de dados de uso exclusivo. Não é comum haver a partilha destes recursos por vários serviços.



10.4 Antipadrões que adotam os microserviços

Microservices are a magic pixie dust - Acreditar que os microserviços resolvem todos os problemas de implementação;

Microservices as the goal - Fazer da adoção dos microserviços o objetivo principal e medir o sucesso em termos de número de serviços escritos;

Red Flag Law - Manter o mesmo processo de desenvolvimento e estrutura organizacional que eram utilizadas no desenvolvimento de aplicações monolíticas;

Scattershot adoption - Ter equipas para adotar a arquitetura de microserviços sem ter uma estratégia de adoção (sem qualquer coordenação);

Trying to fly before you can walk - Começar a praticar técnicas básicas de software como clean code, good design e testes automáticos;

Focar na tecnologia - Focar nos aspectos tecnológicos (mais comum é no desenvolvimento da infraestrutura) e esquecer de issues chave, como decomposição de serviços;

10.5 Criar Microserviços em Java

Como identificar e criar microserviços?

Java Platforms → Versioned Dependencies → Identifying Services → Creating REST APIs

10.5.1 Plataformas Java

Usar **anotações** pode simplificar o código, eliminando a necessidade de escrever código boilerplate, mas pode chegar a um ponto onde existe "demasiada magia".

Podemos criar microserviços perfeitamente funcionais usando nada a não ser bibliotecas Java de baixo nível. No entanto, é geralmente recomendado ter algo que forneça um nível razoável de suporte para concerns comuns, permitindo que o código de um serviço se foque em satisfazer os requisitos funcionais.

O **Spring Boot** dá ênfase à convenção sobre configuração, utiliza um conjunto de anotações e descoberta de classpath para configurar funções adicionais.

Spring Cloud é uma coleção de integrações entre tecnologias de cloud e o modelo de programação Spring.

Até agora foi abordado o **Spring Boot**, um conjunto de convenções que simplificam o desenvolvimento de aplicações Spring. Este é bastante utilizado para a criação de aplicações standalone, que são disponibilizadas num determinado porto e funcionam de forma isolada. Mas spondoo que queremos conectar várias aplicações e construir um sistema distribuído, como é que o fazemos?

Spring Cloud, é um projeto que reduz a complexidade associada aos SD, permite a descoberta de serviços, oferece mecanismos de redundância, load balancing, problemas de performance e complexidade de deploy.

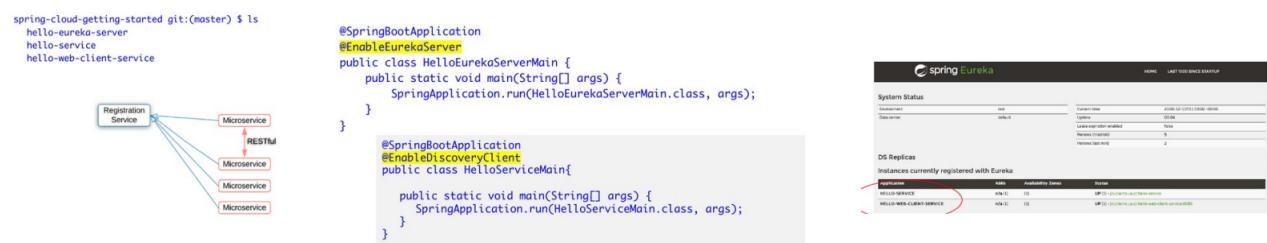
Para facilitar a gestão de configurações, mais vez recorre-se a ficheiros de configuração como o **application.properties**. Neste ficheiro podem ser definidas configurações como endereços de bases de dados e serviços externos com dados de acesso, mas também texto em várias línguas de forma a facilitar o suporte de diferentes idiomas pela nossa aplicação.

Para utilizar uma **configuração** numa classe da aplicação, basta anotá-la com `@EnableAutoConfiguration` e definir a variável que vai receber o seu valor anotada com `@Value("'X')`, sendo X o nome da propriedade definida no ficheiro de configurações. Property files podem usar Encryption/Decryption (ex: password: {cipher}cryptedsedpassword).

Para facilitar a **escalabilidade** destaca-se ainda a o servidor de **nomes / descoberta** Eureka, que permite o registo dos vários serviços da aplicação, que deixam assim de ter necessidade de ter um endereço fixo. Os clientes que lhes querem aceder consultam primeiramente o Eureka para resolver o seu endereço.

A aplicação de resolução de endereço é anotada com `@EnableEurekaServer`, o que vai disponibilizar este serviço no porto 8761. Os clientes que devem ser registados no Eureka, devem ver a sua classe principal anotada com `@EnableDiscoveryClient`.

Temos ainda, **Load Balancing** com Ribbon, **Declarative REST** Client com OpenFeign, que cria uma implementação dinâmica de uma interface decorada com anotações JAX-RS ou Spring MVC, **Circuit Breaker** Resilience4 ou Sentinel, para detetar falhas e encapsular a lógica para prevenir falhas futuras e **Data Flow** que fornece ferramentas para criar topologias complexas para streaming and batch data pipelines.



10.5.2 Controlo de Versões e Dependências

Ferramentas de build como o Maven e o Gradle fornecem mecanismos fáceis para definir e gerir versões de dependências. Explicitamente declara a versão, garantindo que o artefato corre igualmente em produção como em desenvolvimento.

10.5.3 Identificação de Serviços

Quando criamos endpoints de acesso aos serviços, é importante documentá-los. Para este efeito, existe uma norma definida pela **Open API Initiative** (OAI) para web services. Esta é baseada em **Swagger**, que define a estrutura e formata os metadados para criar uma representação de uma API RESTful. A definição é normalmente escrita num ficheiro portável, em JSON ou YAML.

A definição do swagger pode ser ainda mais extendida para gerar **stubs** (código de exemplo) para clientes e servidores, simulando o comportamento do código.

10.5.4 Criação de APIs REST

Não há qualquer restrição quanto à funcionalidade implementada em cada **método HTTP**. No entanto, é importante que a convenção seja seguida, de forma a facilitar a sua utilização. Por exemplo podemos criar entidades através de um pedido GET. No entanto, a convenção é fazê-lo com POST e é assim que deve ser feito!

- **POST** - Criação de recursos. Operação não idempotente (mas pode ser), isto é, por exemplo se um POST criar um recurso, se este for usado várias vezes, um recurso novo e único deve ser criado como resultado de cada invocação.
- **GET** - Consulta de informação. Operações idempotentes e nulipotentes (não alterar a base de dados, não devendo ter efeitos colaterais).
- **PUT** - Atualização de recursos. Operação idempotente, muitas das vezes é necessário copiar o recurso, daí ser idempotente.

-
- **PATCH** - Atualização parcial de recursos. Pode ou não ser idempotente, dependendo de como é implementado.
 - **DELETE** - Eliminação de recursos. Operação idempotente, uma vez que o recurso apenas pode ser eliminado uma vez.

O **valor de retorno** dos pedidos deve também ser relevante e útil. Para além do código HTTP, deve também ser retornada a informação manipulada, de forma a evitar uma nova chamada à API e a tornar a comunicação mais eficiente. Se um recurso for criado ou atualizado, deve estar incluído na resposta, eliminando a necessidade de uma nova chamada para o obter (GET).

Por exemplo em caso de sucesso deve ser retornado código 200 (OK), ou 400 em caso de um pedido mal formado (BAD REQUEST), 204 (NO CONTENT), 201 (CREATED), 409 (CONFLICT), 404 (NOT FOUND).

A forma como os **URIs dos recursos** são definidos tem também definido um padrão, que define que este deve ser composto por nomes plurais e nunca por verbos. Para manipular o mesmo recurso, deve ser utilizado o mesmo URI, com /ID quando se pretende consultar ou manipular um recurso concreto. As relações são modeladas a partir do URI do recurso.

Por exemplo, GET /accounts/16, PUT /accounts/16, PATCH /accounts/16, DELETE /accounts/16, são operações sobre o mesmo recurso, e se quisermos alterar as credenciais deste utilizador, usariammos o URI /accounts/16/credentials.

A **evolução da API** é também uma questão bastante importante, uma vez que as alterações à sua interface não devem implicar a alteração dos serviços que a utilizam, uma vez que iria violar o princípio da independência.

Um dos benefícios dos microserviços é permitir que os serviços **evoluam de forma independente**. Dado que Microserviços podem chamar serviços, esta independência não pode "quebrar" a API. Se precisarmos de fazer alterações importantes à API, existem algumas maneiras de lidar com a versão do recurso REST: **colocar a versão no URI**, definir a **versão no cabeçalho do pedido HTTP** ou até mesmo no cabeçalho do HTTP Accept header e permitir a negociação.

A versão deve ser aplicada à aplicação como um todo. Por exemplo /api/v1/user e não /api/user/v1.

10.6 Serviços de Localização

Como visto anteriormente, em micro-serviços a escalada é feita horizontalmente, ou seja, através da multiplicação dos recursos. No entanto, esta traz a si associado os problemas da localização dos serviços e de distribuição do trabalho entre as várias instâncias. Para lhes dar resposta foram criados métodos para a localização de serviços e load balancing sobre as diferentes instâncias de um serviço.

Existem 4 razões para o porquê de um microserviço querer comunicar com os serviços de localização:

- **Registration** - Depois de um serviço ser deployed com sucesso, o microserviço deve estar registado com o serviço de registo;
- **Heartbeats** - O microserviço deve enviar heartbeats regulares para o serviço, para mostrar que está pronto para receber pedidos;
- **Service Discovery** - Para comunicar com outro serviço, um microserviço deve chamar o serviço de localização para obter uma lista de instâncias disponíveis;
- **De-Registration** - Quando um serviço está em baixo, deve ser removido da lista de serviços disponíveis.

A utilização de terceiros implica a inspeção constante dos serviços para determinar o seu estado atual e atualizá-lo no serviço de localização. Tem como vantagem a separação das lógicas do negócio e do registo. No entanto, implica o deploy de um componente extra.

10.7 Tolerância a Falhas

Para garantir que os serviços disponibilizados nas aplicações são tolerantes a falhas, é importante que incorporem algumas funcionalidades.

- **Timeouts** - Tempo máximo para a resposta a cada pedido (sendo o pedido assíncrono ou não);
- **Circuit Breakers** - Contagem de timeouts que quando atinge um determinado patamar assume que vai dar sempre timeout e deixa de fazer pedidos para esse endpoint;
- **Bulkheads** - Gestão de falhas, de forma a garantir que esta não vai a baixo. Como que o bloco catch. Garantem que as exceções não comprometem o normal funcionamento do serviço. A forma mais fácil é um fallback quando um serviço não vital falha;
- **Consuming APIs** - Como consumidores de uma API, precisamos de **validar a resposta** para verificar que contém a informação. Quando se faz a validação ou JSON parsing (para obter a informação se recebermos um JSON), apenas **validar o pedido contra as variáveis ou atributos** que são necessários. Desta maneira, os microserviços são mais resistentes a mudanças;
- **Producing APIs** - Quando produzimos uma API, para clientes externos, é necessário fazer duas coisas na receção/retorno de respostas: **aceitar atributos desconhecidos como parte do pedido e apenas retornar atributos que são relevantes para a API ser invocada** (deixando espaço para a implementação de um serviço para mudar ao longo do tempo). Se cumprirmos estes, podemos alterar a API para mudar requisitos para os clientes - Princípio da Robustez.

Os deployment artifacts podem correr em sistemas diferentes, muitos serviços devem ser configurados para fazer os serviços funcionarem. A intervenção humana deve ser mínima, usando: **Source Code Management** (SCM) como Git, **Dependency Management** como Maven, **Repository Management Tools** e **CI-Tools** como Jenkins.

Depois de construir o microserviço, precisamos de decidir no **formato do package para dar deploy à aplicação**. Pode ser JAR/WAR, Executable JAR, Containers, ou cada microserviço pode ter o seu próprio servidor virtualizado.

JAR/WAR - É o formato padrão para aplicações Java EE. Tem alguns problemas como, algumas aplicações necessitam de ser reiniciados, load de uma aplicação pode afetar outras. Se este padrão for usado com microserviços, a **relação deve ser de 1 para 1 entre o serviço e o middleware do servidor**.

Executable JAR - Dar pack a tudo o que é necessário para correr a aplicação, dentro de um ficheiro JAR.

Containers - **Trazer mais do sistema operativo para ti**, evitando o problema que resultam em inconsistências na consistência do sistema operativo. Os containers são lightweight virtualization artifacts que são hosted num sistema operativo existente.

Virtual Machines - Para gerir este tipo de servidores, podemos usar **Vagrant**, **VMWare**, **VirtualBox**.