

Lab 1 Ambiente de desenvolvimento em equipa (maven, git, containers)

Introdução e enquadramento

Objetivos de aprendizagem

- Utilizar um *framework* para a construção de projetos com gestão explícita de dependências.
- Aplicar práticas de gestão cooperativa do código fonte com o git.
- Utilizar tecnologia de containers para facilitar a preparação de ambientes de execução (e.g.: Docker).

Recursos e referências

- Livros: “[Maven by Example](#)”; “[Pro Git](#)”

Entrega

Cada aluno deve entregar, no Moodle, evidência do trabalho da aula. Para isso, deve ser preparado um ficheiro zip (único) com uma pasta por cada secção (íes1.1, íes1.2,...). Na raiz, deve existir um ficheiro “[readme.md](#)” com a identificação do autor e as explicações pedidas ao longo do exercício. Os itens para constar da entrega estão assinalados no guião com 📌.

1.1 Gestão da *build* com *maven*

A construção um projeto de código pode incluir [várias etapas](#) (também chamadas objetivos) tais como a obtenção de dependências, compilação do código fonte, produção de binários (*artifacts*), atualização da documentação, instalação no servidor, etc. Estas tarefas são coordenadas por uma ferramenta de construção (build tool), e.g.: Maven, Gradle. O [Maven](#) é muito popular entre projetos profissionais em Java.

Preparar o ambiente para usar projetos Maven

Os projetos configurados para trabalhar com o Maven podem ser usados em todos os IDE principais que dispensam configurações adicionais. No entanto, vamos instalar e configurar o Maven para estar disponível também na linha de comandos (CLI).

Todo o ciclo da *build* pode ser gerido a partir da linha de comandos, o que torna o Maven uma ferramenta flexível para integrar com ambientes de desenvolvimento (IDE) e em ferramentas de integração contínua (sem consola interativa).

- a) [Instalar o Maven](#) no ambiente de desenvolvimento. (Requer o JDK.)

Nota: a variável de ambiente “**JAVA_HOME**” deve existir e indicar a pasta **raiz do JDK** (não o JRE).

Pode testar a instalação do Maven com:

```
$ mvn --version
```

- b) Familiarize-se com os comandos principais executando o [guia introdutório: “Maven in 5 minutes”](#).

Adapte o “groupId” e “artifactId” para um exemplo mais adequado/personalizado, tendo em consideração as [regras convencionadas](#).

Criar um novo projeto (cliente meteorologia)

Vamos agora criar uma pequena aplicação Java para invocar a [API previsão meteorológica](#) disponível no [IMPA](#).

- c) Utilize o *browser* (ou o comando *curl*) para perceber a estrutura do pedido e das respostas da API. Por exemplo, para pedir a previsão dos próximos 5 dias para a cidade de Aveiro (que tem o código interno 1010500):

```
$ curl http://api.ipma.pt/open-  
data/forecast/meteorology/cities/daily/1010500.json | json_pp
```

- d) Crie um projeto (WeatherRadar) para uma aplicação Java, baseado em Maven.

Pode fazê-lo na linha de comandos ou pode utilizar um IDE com suporte para o Maven (e.g.: [VS Code](#), [IntelliJ IDEA](#), [NetBeans](#), [Eclipse](#)). Certifique-se que o seu IDE tem as extensões necessárias. Em todos os projetos Maven, deve indicar o “groupId”, “artifactId”; a “versão” inicial de um projeto deve ser “1.0-SNAPSHOT”. Verifique o conteúdo do ficheiro POM.xml.

- e) Configure alguns metadados do projeto (no POM), tais como: [a composição da equipa de desenvolvimento](#) e as propriedades que definem [versão do compilador](#) a usar (*target=1.8*).

A aplicação pretendida precisa de abrir uma ligação HTTP, criar um pedido GET, obter o JSON com a resposta, processar o conteúdo da resposta para usar o resultado. Há vários passos envolvidos, que podem ser grandemente simplificados recorrendo a bibliotecas externas. O uso dessas bibliotecas (ou *artifacts*) dá origem a [dependências](#) e o maven facilita a sua gestão.

- f) Neste problema, vamos usar dependências para as bibliotecas [Retrofit](#) e [Gson](#). Comece por declarar estas dependência no POM do projeto (ver passo [#2, aqui](#)).

- g) Numa versão inicial da solução, o corpo principal pode ter uma estrutura semelhante a esta:

```
public static void main(String[] args) {  
    Retrofit retrofit = new Retrofit.Builder()  
        .baseUrl("http://api.ipma.pt/open-data/")  
        .addConverterFactory(GsonConverterFactory.create())  
        .build();  
  
    IpmaService service = retrofit.create(IpmaService.class);  
    Call<IpmaCityForecast> callSync = service.getForecastForACity(CITY_ID_AVEIRO);  
  
    try {  
        Response<IpmaCityForecast> apiResponse = callSync.execute();  
        IpmaCityForecast forecast = apiResponse.body();  
  
        System.out.println("max temp for today: " + forecast.getData().  
            listIterator().next().getTMax());  
    } catch (Exception ex) {  
        ex.printStackTrace();  
    }  
}
```

Nota: utilize a implementação indicativa das classes IpmaService e IpmaCityForecast disponíveis no Moodle, necessárias para a execução do projeto.

Assegure-se que o projeto está sem erros e experimente executá-lo.

Note que, tratando-se de um projeto configurado com o Maven, pode ser aberto e executado, sem alterações, em diferentes IDE e também na linha de comandos:

```
$ mvn clean package  
$ mvn exec:java -Dexec.mainClass="ies.lab1wradar.GetWeatherForCity"
```

- h) Altere a implementação para receber o código da cidade como parâmetro pela linha de comandos e imprima a informação da previsão de forma mais amigável e completa.

Note que na execução (mvn exec:java) pode também indicar argumentos (exec.args).

- 📄 Entregar o projeto de código (completo), depois de fazer “clean”.
- 📄 No ficheiro “readme” da aula, responder às questões: o que é um “maven goal”? quais os principais “maven goals” e a respetiva sequência de invocação?

- i) **[Exercício opcional]** confirme que as classes têm documentação adequada (Javadoc) na descrição da classe e métodos (adicione a documentação em falta).

[Gere um site com a documentação](#) do projeto e explore a documentação produzida num browser (./target/apidocs/index.html)

```
$ mvn site
```

Para a maior parte das instalações, o objetivo anterior vai falhar. Trata-se de um conflito entre as versões dos *plugins* do Maven assumidas por omissão e requer... *troubleshooting*. Resolva o problema [especificando a versão correta do plug-in](#).

Certifique-se que [inclui ainda a geração do javadoc na fase de reporting](#). Verifique o relatório (site) gerado.

1.2 Gestão do código fonte

Certifique-se que tem a instalação do Git disponível na linha de comandos no seu ambiente de desenvolvimento.

```
$ git config --list
```

Se não obtiver um output com as definições do *git*, incluindo o seu nome e email, então [complete/configure a instalação do git](#).

Os exercícios baseiam-se na linha de comandos. A utilização de clientes gráficos é opcional (e.g.: [SmartGit](#), [SourceTree](#), [GitKraken](#)). Para além disso, os IDE têm suporte integrado para executar comandos *git*.

- a) Para uma a iniciação ao *git*, faça o exercício proposto na [documentação do Bitbucket](#). Caso já esteja familiarizado com o *git*, pode avançar para os próximos passos. Nota: apesar do tutorial usar o Bitbucket, não há nenhuma preferência por este ambiente.

Os passos seguintes supõem uma equipa (2+ alunos).

- b) Crie um repositório (remoto) que servirá para a equipa partilhar o código. Existem, para isso, vários recursos gratuitos disponíveis: Code@UA, BitBucket, GitHub, GitLab,...

Tome nota do URL para acesso ao repositório.

- c) Tomando como ponto de partida o projeto já existente das alíneas anteriores, vamos colocá-lo no repositório partilhado.

Comece por adicionar a informação sobre as exclusões, isto é, indicar os ficheiros/pastas que só têm interesse local e não devem ser passados para o repositório comum (e.g.: toda a pasta `/target`, que tem as versões compiladas, e não pertence ao *source*).

Para isso, [crie um ficheiro `.gitignore` na raiz do projeto](#) adequando aos projetos maven.

- d) Adicione suporte ao projeto para trabalhar com git e, de seguida, publique as alterações para o repositório remoto. **Apenas um** dos membros da equipa deve partilhar o projeto.

Exemplo indicativo (tem de adaptar para o seu caso):

```
$ cd project_folder
$ git init                # initialize a local git repo in this folder
$ git remote add origin git@gitlab.com:ico_gl/ies-labs19.git      #adapt the url
$ git add .               # mark all existing changes in this root to be committed
$ git commit -m "Initial commit"                                #create the commit snapshot locally
$ git push -u origin master                                     #uploads the local commit to the shared repo
```

- e) Depois, o(s) outro(s) membro(s) deve obter uma cópia de trabalho (i.e., *clone*).
- f) Considere aqui que a equipa vai criar duas novas funcionalidades (ou *features*) no projeto. Certifique-se que adota um modelo de trabalho (*workflow*) cooperativo e distribuído, tal como o [feature-branch workflow](#).

Neste caso, cada *feature* pode ficar entregue a um membro da equipa:

- (1) Adicione suporte para *logging*, i.e., a aplicação deve **escrever um log** com o registo das operações que vão sendo feitas, bem como de eventuais problemas que tenham ocorrido. O log pode ser direcionado para a consola ou para ficheiro (ou para ambos). Recorra a uma biblioteca adequada de *logging* para Java [como o Log4j 2](#).

Note que no exemplo do *link* existe um [ficheiro de configuração](#) (`log4j2.xml`) que deve ser colocado na pasta de recursos do projeto maven, seguindo a [estrutura convencional](#).

- (2) Adicione o suporte necessário para receber (como argumento da linha de comandos) o **nome da cidade** que se pretende consultar, em vez do código. Note que pode [consultar programaticamente o código associado ao nome da cidade](#) usando a API do IPMA.

Pode, naturalmente, partir da base existente e acrescentar o novo serviço para obter a lista das cidades. A resposta pode ser mapeada em classes, que precisam de ser adicionadas ao projeto (considere [gerá-las a partir do JSON da resposta](#)).

- 📄 Entregar o log das operações de git realizadas, que pode ser obtido com:

```
$ git log --graph --oneline --decorate > lab1_3-2.log
```

- 📄 Entregar o projeto, depois das alterações solicitadas.

1.3 Introdução ao Docker

A utilização de *containers* (juntamente com ficheiros que contêm as instruções para preparar um ambiente de execução de uma solução) [facilita a vida](#) do *developer* (é mais fácil preparar e partilhar um ambiente) e da produção (“operations”).

Iremos recorrer várias vezes a *containers* na disciplina.

- a) Começamos por instalar o [Docker](#) Engine – Community.

Os [utilizadores de Windows devem ponderar](#) se querem mesmo fazê-lo (algumas aplicações de virtualização, como o VirtualBox, deixarão de funcionar): Em Windows:



— há sempre a alternativa de instalar dentro de uma VM.

- Se não há necessidade de partilhar o HyperV, então a instalação do [Docker Desktop](#) é mais adequada.
- b) Faça o tutorial do Docker “[Part 1: Orientation and Setup](#)”
- c) Faça o tutorial do Docker “[Part 2: Containers](#)”
- d) Embora toda a gestão dos *containers* possa ser feita na linha de comandos, por vezes é mais fácil usar um ambiente gráfico. O *portainer* é uma aplicação web que permite aceder, no browser, ao ambiente de gestão do Docker.

Naturalmente, para [instalar o portainer](#), prepare um container *Docker* com base na imagem partilhada no Docker Hub.

Depois de instalado, aceda a <http://localhost:9000> (na primeira vez, defina as credenciais de administração).

- e) Utilize o Docker para criar uma imagem do servidor PostgreSQL.
Certifique-se que faz as configurações necessárias (Dockerfile) para inicializar o servidor e deixar o serviço a reiniciar automaticamente (serviço). A área de dados (da base de dados) deve persistir entre *reboots* (a base de dados não deve ser volátil).

-  Resultado do comando: `$ docker container ls -all`
-  Conteúdo do Dockerfile da 1.3.e).