Universidade Fernando Pessoa

Material de apoio à componente prática da cadeira de Engenharia de Software

Alessandro Moreira
afmoreira@ufp.edu.pt

Setembro 2020

Conteúdo

Lista de Figuras

Lista de Códigos

Introdução

O objetivo deste documento é servir de apoio para as aulas práticas da cadeira de Engenharia de Software. É imprescindível que seja consultado antes de cada sessão para que as aulas possam servir como espaço de discussão e esclarecimento de dúvidas. O documento possui muita implementação de código e explicação de conceitos associados, bem como referências para aprofundamento.
Caso haja alguma inconsistência/erro em qualquer dos conteúdos aqui apresentados, pede-se que seja reportado para que as correções sejam realizadas.
As sessões práticas consistirão em implementar uma pequena aplicação. Todos os passos estão presentes no repositório GIT[1]

Caso de estudo

Pretende-se que desenvolvam um Web Service que suporte explicações a conteúdos programáticos de cadeiras do ensino superior. Deverá haver pelo menos dois tipos de utilizadores, estudantes que buscam por explicações e os explicadores. Os explicadores deverão dar explicações a uma ou várias cadeiras de um determinado curso e deverão informar as disponibilidades que possuem em termos de períodos de tempo (início e fim) para cada dia da semana. O aluno deverá poder pesquisar explicadores por cadeiras, dia da semana, períodos de tempo separadamente ou por conjunção destes critérios. O modelo deverá contemplar a modelização de uma universidade do ponto de vista de suas faculdades e dos cursos destas.

[1] `https://github.com/afmoreira-ufp/sweng-lectures`

Tópico 1: VCS

Docente: Alessandro Moreira

1.1. Primeiros passos

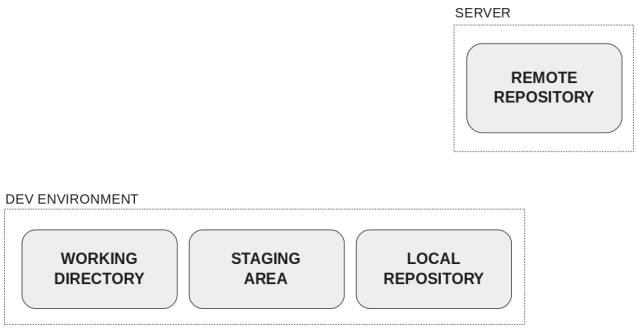
Deverão instalar a aplicação git no link abaixo e garantir que os executáveis/binários pertencem ao PATH:`https://git-scm.com/downloads`.
Para verificar que a instalação foi realizada corretamente podem utilizar o comando ***git --version***.
Em seguida, deverão criar uma conta num servidor git remoto, por exemplo `https:\www.github.com` ou `https:\www.bitbucket.com`.
Para maior comodidade, podem configurar o git para possuir informação sobre as credenciais em um destes servidores com os comandos abaixo:

git config --global user.name "FIRST_NAME LAST_NAME"
git config --global user.email "MY_NAME@example.com"
git config --global user.password "MY_PASSWORD"

1.2. Conceitos importantes

A figura abaixo ilustra as principais entidades do git.

Figura 1.1: Estados principais do git



O *Working Directory* equivale ao próprio sistema de ficheiros do SO, que é onde os ficheiros são criados, modificados e removidos. Sempre que há alguma alteração no *Working Directory* o git deve ser informado a respeito para que possa rastrear (*track*) estas alterações. Quando um ou mais ficheiros são rastreados pelo git eles pertencem ao *Staging Area*. Por estar rastreados pelo git, estes ficheiros são candidatos a serem confirmados (*commited*), que transforma o estado atual dos ficheiros em uma versão. Versões confirmadas de ficheiros pertencem ao *Local Repository*. O *Remote Repository* corresponde ao repositório existente em algum dos servidores remotos de git existentes. São utéis para salvaguarda externa dos documentos, bem como da sua distribuição. Só podem ser distribuídos para o *Remote Repository* os ficheiros confirmados (ou seja, existentes no *Local Repository*).
As figuras abaixo ilustram os relacionamentos entre estas entidades e os comandos que realizam estas transições, que serão mencionados a seguir.

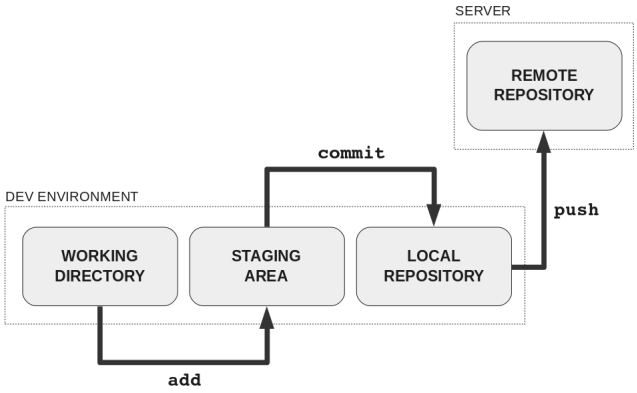


Figura 1.2: Fluxo básico de operações (commit-push)
Fonte: https://rachelcarmena.github.io/2018/12/12/how-to-teach-git.html

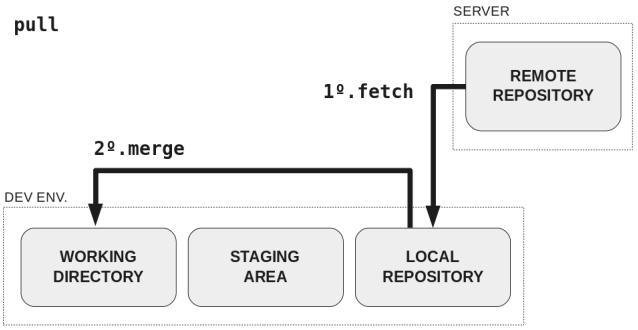


Figura 1.3: Fluxo básico de operações (pull)
Fonte: https://rachelcarmena.github.io/2018/12/12/how-to-teach-git.html

Os ficheiros possuem os estados que estão representados na figura 1.4.

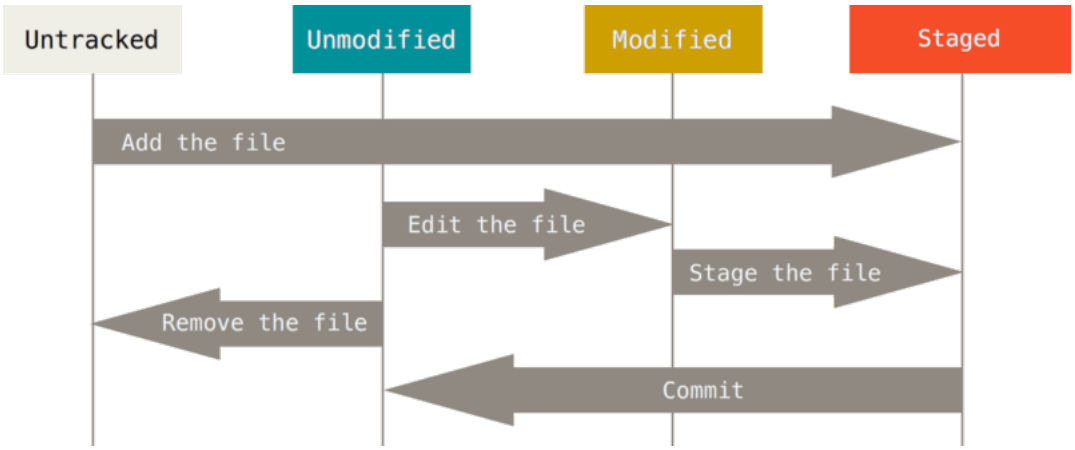


Figura 1.4: Estados de um ficheiro

1.3. Comandos importantes

A criação um repositório git pode ser feito de duas formas. Ou de raiz, com o comando **git init** ou reaproveitando um repositório remoto pré-existente, com o comando **git clone** e passando o url do repositório.
Dois comandos bastante importantes para a gestão de um repositório são o **git status** e **git log**. O primeiro comando informa o estado atual do repositório. Assim, pode-se verificar, por exemplo, se o repositório está atualizado, se há ficheiros não rastreados ou rastreados e não confirmados.

O segundo comando exibe todos os *commits* do repositório. Se for seguido do parametro --graph, o resultado apresenta, também, o grafo de commits (importante para mostrar relacionamento entre as *branches*) e o parametro --decorate exibe cores distintas para as *branches*.
Para fazer com que um ou mais ficheiros sejam rastreados pelo git, usa-se o comando ***git add***, usando como paramteros um ou mais ficheiros ou o caracter . para incluir todos os ficheiros no diretório atual e em subdiretórios.
A confirmação (*commit*) é feita com o comando ***git commit***. Cada *commit* é identificado por um condensado gerado para cada ficheiro confirmado, de modo a garantir a integridade da informação. Existem diversos parâmetros para este comando, em especial -m, que exige uma mensagem para identificar o commit e o -a, que permite que os ficheiros já rastreados e modificados possam ser incluídos no *commit*, dispensando mais uma utilização do comando ***git add***[2]. Como boa prática, aconselha-se que a mensagem de commit seja explicativa das principais modificações acrescentadas pelo *commit*.
Para realizar interações com repositórios remotos é preciso que ele esteja configurado. Se o repositório foi criado de raiz, tal é feito com o comando ***git remote add <remote alias> <remote url>***. O *remote alias* pode ser qualquer texto que identifique o repositório remoto, entretanto, o padrão é que esta alias seja a palavra *origin*, o que acontece quando o repositório foi criado com o comando ***git clone***.
O *remote alias* é utilizado nos comandos para interação com o repositório remoto. Quando é preciso atualizar o repositório remoto com as modificações feitas localmente, usa-se o comando ***git push <remote alias> <branch name>***[3]. Por defeito, todo o repositório tem uma branch chamada *master*, logo o mais comum será o comando completo ser ***git push origin master***. Se o que se pretende é verificar se há novidades no repositório remoto e sincronizar o local com base nestas modificações, o comando será ***git pull <remote alias> <branch name>***. Para interagir com um repositório remoto, é necessário que este possua o nome do utilizador que irá executar os *pushes* e *pulls* como colaborador.

1.3.1. Exercícios

- Devem criar um diretório qualquer
- Em seguida devem inicializar um repositório git neste diretório
- Executem o comando ***git status***
- Criem um ficheiro de texto qualquer (p. ex. file1.txt) com um conteúdo qualquer
- Executem novamente o comando ***git status*** e interpretem o resultado
- Executem o comando ***git add <nome ficheiro>*** ou ***git add .***
- Executem novamente o comando ***git status*** e interpretem o resultado
- Façam o *commit* desta versão do ficheiro com o comando ***git commit -m "first commit"***
- Executem novamente o comando ***git status*** e interpretem o resultado
- Executem o comando ***git log*** e verifiquem que o *commit* aparece listado.
- Devem criar um repositório no *gitHub* e adicioná-lo como repositório remoto com o comando ***git remote add origin <url remoto>***
- Executem o comando ***git push origin master*** e depois verifiquem que o repositório remoto já possui o ficheiro.

[2] O parametro -a do comando *git commit* não significa *add*, mas *all*, no sentido em que irá incluir os ficheiros modificados ou removidos

[3] A notação diamante (<>) denota que a palavra ou expressão em seu interior representa uma variável. No caso <remote alias> deverá ser substituído por uma expressão que será o alias do repositório remoto, de igual forma com <branch name> para o nome da *branch*, de modo que ***git push origin master*** é um comando válido

1.4. *Branches* e *merges*

As *branches* são divergências de ficheiros já existentes e confirmados. A idéia principal é permitir que os ficheiros já finalizados e confirmados sejam mantidos em salvaguarda e possam ser modificados de modo seguro em uma outra "linha temporal". Em havendo qualquer problema no desenvolvimento de modificações nos ficheiros, a versão estável mantem-se na *branch* original. Se as modificações feitas são satisfatórias, a *branch* original pode ser acrescentada com estas modificações, por meio do mecanismo de *merge*.
Toda nova *branch* é uma divergência, uma cópia da *branch* atual. O comando ***git branch*** lista todas as *branches* existentes no repositório. Com o argumento -a, lista-se as *branches* locais e as remotas. Se for usado o comando ***git branch <branch_name>***, cria-se uma nova *branch* com o nome passado como argumento. Se o comando for feito na *branch master*, esta nova *branch* será "filha"da *branch master*, no sentido em que terá todos os ficheiros desta no mesmo estado.
Cada *branch* possui *Working directory* e *Staging area* próprios, pelo que um mesmo ficheiro, presente em mais de uma *branch* pode apresentar no sistema de ficheiros do sistema operativo, conteúdos diferentes.
Para mudar de *branch* usa-se o comando ***git checkout <branch_name>***[4]. Este comando com o argumento -b cria e muda para uma nova *branch*.
Um *merge* é feito com o ***git merge <branch name>*** e causa o acréscimo do *Working directory* da *branch* passada para a *branch* atual. Ou seja, se estivermos na *branch master* e executarmos o comando *git merge dev*, os ficheiros **confirmados** existentes no *Working directory* da *branch* dev serão acrescidos à *branch master*. A *branch master* não deverá ter ficheiros por confirmar, o que é causa de erro de *merge*.

1.4.1. Exercícios

- Devem executar o comando ***git branch*** e verificar que apenas existe a *branch master*
- Devem criar uma nova *branch* chamada dev, com o comando ***git branch dev***.
- Executem novamente o comando ***git branch*** e verifiquem que existem as textitbranches master e dev (A *branch* ativa está marcada por * e deverá ser a *master*).
- Alternem para a *branch* dev com o comando ***git checkout dev***.
- Criem um novo ficheiro (p. ex. file2.txt) com um conteúdo qualquer, utilizem, em seguida, os comandos ***git add*** e ***git commit***.
- Verifiquem o estado do repositório com os comandos ***git status*** e ***git log***
- Retornem à *branch master* com o comando ***git checkout master*** e verifiquem que o novo ficheiro não existe.
- Façam o *merge* da *branch dev* na *branch master* com o comando ***git merge dev***
- Verifiquem que o novo ficheiro já existe.

1.5. Conflitos

Um conflito no git ocorre quando há duas versões modificadas e confirmadas de um ficheiro em zonas muito semelhantes. Um conflito é uma forma do git avisar o utilizador que não sabe como lidar com divergências

[4]O git possui um apontador especial chamado *HEAD* que define o estado atual dos ficheiros. Ao usar o *git checkout* para mudar para uma outra *branch*, o git muda o *HEAD* para esta *branch* de modo que o sistema de ficheiros reflete a versão atual dos ficheiros nesta *branch*.

muito díspares de versões de um mesmo ficheiro.
A forma mais comum de conflito é no *merge* de duas *branches* em que um mesmo ficheiro foi modificado e confirmado em ambas as *branches*. Quando ocorre esta situação, o git alerta o utlizador e não realiza o *merge* até que o conflito seja resolvido.
A forma mais simples de resolver o conflito é aceitar uma das duas versões do ficheiro, ou da *branch* atual ou da *merging branch*. Na ocorrência de um conflito, o git modifica o(s) ficheiro(s) em conflito, incluindo todas os conteúdos conflitantes do(s) ficheiro(s). Estes conteúdos conflitantes são separados com estes textos.

< < < < < < < HEAD
=======
> > > > > > > <merging branch name>

Tudo o que estiver entre **< < < < < < <** HEAD e ======= são conteúdos presentes na versão da branch atual e tudo o que estiver entre "======="e **> > > > > > > <merging branch name>"**, pertencem à *merging branch*. Assim, para solucionar o conflito, devem apagar estes separadores e o texto que não desejam que esteja no *merge* e depois confirmar a versão com um *commit*.

1.5.1. Exercícios

- No primeiro ficheiro criado, devem alternar para a *branch master*, acrescentar mais algum texto e confirmar as modificações
- Devem alternar para a *branch dev*, acrescentar texto distinto do acrescentado na *branch master* e confirmar (o conteúdo acrescentado na *branch master* não deve aparecer)
- Devem alternar para a *branch master* e executar o comando ***git merge dev***. Deve aparecer um aviso de conflito para ser solucionado
- Devem solucionar o conflito e confirmar a versão.

1.6. Referências

1. `https://git-scm.com/book/en/v2`
2. `https://guides.github.com/introduction/git-handbook/`
3. `https://www.atlassian.com/git/tutorials/learn-git-with-bitbucket-cloud`
4. `https://www.atlassian.com/git/tutorials/using-branches/merge-conflicts`
5. `https://rachelcarmena.github.io/2018/12/12/how-to-teach-git.html`
6. `https://dev.to/unseenwizzard/learn-git-concepts-not-commands-4gjc`
7. `https://medium.com/@patrickporto/4-branching-workflows-for-git-30d0aaee7bf`

Tópico 2: UML - Diagrama de Casos de Uso

Docente: Alessandro Moreira

2.1. Desenho

O desenho de um software é uma das fases mais importantes no desenvolvimento. Um bom desenho pode implicar em economizar tempo e esforços, prevendo problemas antes da implementação. Para além disto os desenhos de um software servem para comunicar com outros desenvolvedores, e até mesmo com outros interessados, tornando explícito o modo como o software é desenvolvido, seus modelos de dados, lógica, dependências, entre outros detalhes de implementação.
Há várias técnicas e ferramentas para esta atividade, sendo o UML a mais utilizada em desenvolvimento Orientado a Objeto. O UML é composto por 14 diagramas, sendo divididos em Estruturais e Comportamentais. Aqui serão explorados apenas dois, o diagrama de casos de uso e o diagrama de classes. Para mais pormenores, há algumas referências para serem seguidas no fim deste capítulo.

2.2. Diagrama de Casos de Uso

O diagrama de casos de uso é utilizado para representar as funcionalidades que um sistema ou aplicação fornece aos seus utilizadores. É um diagrama de alto nível, ou seja, **não representa detalhes de implementação**. **Também não se presta à representação de lógica do sistema/aplicação**.
Um caso de uso pode ser interpretado como um cenário de utilização. Um diagrama de casos de uso é composto pelos elementos abaixo:

- Sistema: Quadrado
- Actores: Stick figures
- Casos de uso: Ovais
- Relacionamentos: Linhas/Setas

O sistema define a separação entre os actores e os casos de uso. Os casos de uso representam as funcionalidades do sistema ou aplicação a ser implementado. Os actores representam classes de intervenientes que interagem com as funcionalidades do sistema. Podem ser actores humanos ou não-humanos. Os utilizadores do sistema são necessariamente actores, mas podem haver outros exemplos de actores, como administradores do sistema, dispositivos, outros sistemas.
A ligação entre actores e casos de uso é feita por meio do relacionamento de associação, representado por linhas sólidas. Uma associação entre um actor e um caso de uso significa que aquela funcionalidade é utilizada ou fornecida para aquele actor. De outro modo, apenas os actores que estão associados a um determinado caso de uso **têm autorização para utilizá-lo**.

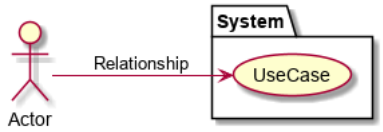


Figura 2.1: Elementos principais de uma diagrama de casos de uso

Notas adicionais

- Todos os actores devem estar associados a pelo menos um caso de uso
- Todos os casos de uso devem estar associados a pelo menos um ator OU relacionado a outros casos de uso (herança, inclusão ou extensão)
- Todo caso de uso é uma funcionalidade ou uma ação que um actor realiza ao interagir com o sistema, portanto deve conter pelo menos um verbo e um substantivo
- Os casos de uso são funcionalidades que o sistema deve possuir, extraídas da análise de requisitos feita, muitas das vezes, com a participação do cliente. Deve, portanto, ser tratado como uma lista de verificação para definir se o projeto está ou não concluído.

2.3. Exercício

Elaborar o diagrama de casos de uso para o enunciado do caso de estudo. Podem utilizar qualquer ferramenta para o efeito. Sugere-se as abaixo:

1. `https://www.lucidchart.com/`
2. `https://github.com/argouml-tigris-org/argouml/releases/tag/VERSION_0_35_1`

2.4. Referências

1. `https://link.springer.com/book/10.1007/978-3-319-12742-2` (Disponível na plataforma de e-learning)
2. `https://www.martinfowler.com/books/uml.html` (Disponível na plataforma de e-learning e na biblioteca)

Tópico 3: UML - Diagrama de Classes

Docente: Alessandro Moreira

3.1. Diagrama de Classes

O diagrama de classes representa a estrutura estática da aplicação para implementar as suas funcionalidades. Em programação orientada a objetos, o diagrama de classes é o diagrama mais utilizado.
O paradigma de programação orientado a objetos tem como objetivo tratar um programa como um conjunto de objetos que colaboram para implementar uma determinada solução. Objetos ou módulos são abstrações de um conjunto de instruções que possuem alguma coesão ou significado. Estas instruções podem ser dados ou ações que podem operar nestes dados. A colaboração entre objetos é feita por meio da passagem de mensagens. Dois objetos trocam mensagens quando um objeto "pede"para que o outro faça algo e retorne o resultado. Em outras palavras, um objeto invoca uma ação que o outro objeto pode executar e recebe o valor resultante desta ação.
Via de regra, objetos são caixas negras dos dados que possuem (encapsulamento). Isto significa que a operação sobre estes dados deve ser feita pelo próprio objeto. Um objeto pode divulgar estes dados ou permitir um modo de modificação, mas sempre de forma controlada. Há, para isso, o conceito de visibilidade, ou seja, quem tem permissão de operar com dados (atributos) ou ações (métodos). Estas permissões podem ser mais restritivas, apenas os objetos daquela classe podem operar com aqueles dados ou ações (visibilidade privada) ou mais permissivas, qualquer objeto pode operar com aqueles dados ou ações (visibilidade pública).
Um conjunto de métodos públicos define uma Interface. Uma Interface define o Tipo de um objeto.
Um diagrama de classes possui os seguintes elementos principais:

- Classe: Retangulo com 3 divisões
- Interface: Retangulo com 2 divisões
- Relacionamento: Linhas/Setas

Uma classe é uma fábrica de objetos. Um objeto é uma instância de uma classe e possui características próprias (valores vinculados a seus atributos). Quando um objeto é criado a partir de uma determinada classe, este objeto possui o conjunto de atributos e comportamentos determinados por esta classe.
Cada atributo deve possuir um tipo, que pode ser um tipo primitivo ou uma outra classe (ou interface). As classes podem possuir comportamentos, ou ações que desempenham. Em linguagem orientada a objeto são chamados de métodos. Um método deve possuir um nome. Podem retornar algo (um tipo) ou não retornar qualquer valor (void). Podem necessitar de um conjuto de parametros para realizar a sua tarefa ou nenhum parametro.
Um exemplo simples pode ser descrever uma Viatura, que possui uma matrícula, uma marca e um modelo. Uma viatura deverá ser capaz de apitar. A representação desta classe segue abaixo:

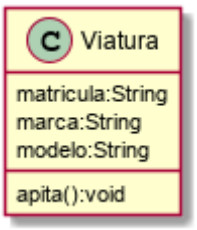


Figura 3.1: Exemplo de uma classe

Uma classe pode ter relacionamento com outras classes. As formas de relacionamento são as seguinte:

- Associação
- Agregação
- Composição
- Herança
- Implementação (Classe com Interface)

3.1.1. Associações, Agregações e Composições

A associação é o relacionamento entre duas classes que representa a possibilidade de comunicação entre as suas instâncias. Em termos de implementação, a associação entre duas classes implica na possibilidade de instâncias de pelo menos uma das classes possuir referência a instância da outra classe.
Associações têm as seguintes propriedades:

- Direcionalidade
- Multiplicadade

Uma associação pode ser unidirecional ou bidirecional. Uma associação unidirecional significa que uma instância conhece a outra instância, mas esta não conehce a primeira. Em termos de implementação, apenas a primeira instância possui uma referência para a segunda instância. Uma associação bidirecional significa que ambas as instâncias se conhecem.
Multiplicadade diz respeito à quantidade de instâncias referidas na associação. As associações devem possuir valores mínimos e máximos, sendo que ambos podem coincidir. Segue abaixo duas formas de representar a associação entre objetos das classes Viatura e Pessoa.

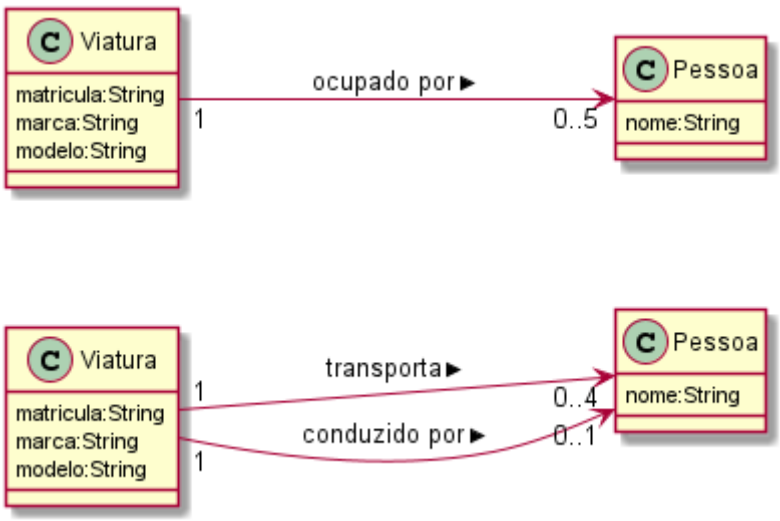


Figura 3.2: Dois modos de representar associação entre duas classes

Em ambas as situações, a capacidade máxima para cada objeto Viatura é 5 de Pessoas. Em um dos casos, porém, há dois tipos de ocupantes, passageiros e condutor. Em ambos os casos as relações são unidirecionais no sentido Viatura-Pessoa, ou seja, a Viatura conhece as Pessoas, mas as Pessoas não conhecem a Viatura.

A escolha por ser unidirecional depende, principalemente, do contexto da realidade que se deseja representar. Importante salientar que não há representação correta, sem que se considere o contexto. O contexto do problema a ser resolvido que irá determinar se uma representação é ou não adequada.
Os objetos possuem dados inerentes àquilo que representam e devem ser responsáveis por manipulá-los e permitir acesso a estes dados a terceiros. O traço mais distintivo de um objeto é **ser um conjunto de métodos que operam sobre os seus atributos**[5]. Estas características são chamadas de encapsulamento dos objetos/classes e é o motivo pelo qual, por defeito, os atributos de uma classe devem ser visíveis apenas por elas próprias (ou seja, os atributos devem ser private). Em termos de UML, atributos ou métodos privados são representados por um sinal de subtração (-), enquanto os públicos são por um sinal de soma (+). Há mais dois modificadores de visibilidade, o protegido (#) e o por defeito, em que não há qualquer símbolo. O modificador de visibilidade protegido implica que o atributo ou método é visível (acessível para leitura e escrita) para todas as classes do mesmo pacote (package), sub-pacotes e suas sub-classes.
Um pacote é uma coleção de classes, que convém terem um significado comum. Pode ser pensado como um diretório de ficheiros. A inclusão de classes num pacote é arbitrário, mas convém que tenha um sentido para fins de organização do projeto a ser desenvolvido, de modo a facilitar a vida dos programadores. O modificador de visibilidade por defeito, é visível apenas pelos objetos do mesmo pacote e sub-pacotes.
Como já referido, os objetos, via de regra, devem ser encapsulados, logo, terem seus atributos não acessíveis para objetos de outras classes. Todavia, podem possibilitar que outros objetos que dependam destes dados possam lê-los ou modificá-los de modo controlado. Existem métodos chamados de acessores, que são responsáveis por permitir a leitura (getters) e escrita (setters) destes dados.
Os setters permitem a modificação do valor dos atributos. Podem ser definidas condições para modificação destes dados de modo a controlar os valores válidos do atributo a ser modificado. Isto garante que há um único ponto em que esta condição será implementada, em vez de ter esta verificação espalhada pelo projeto (o que causaria problemas caso a verificação fosse esquecida, ou se a condição fosse alterada por mudança na regra do negócio).
Os getters permitem acesso aos dados dos atributos do objeto. Como os atributos são privados, o valor dos atributos só podem ser modificados via setters. Se os atributos forem tipos primitivos, não há problema, porque os valores só podem ser modificados via setters. De igual maneira ocoorre se o atributo for um objeto, pelo que não pode ser modificado, no caso, atribuir-lhe uma nova instância diretamente. O mesmo acontece com uma Coleção, por exemplo, um ArrayList. A diferença é que se o getter de uma Coleção apenas devolver a instância desta Coleção, o objeto que invocou o getter pode modificar esta Coleção, ou seja, incluir, remover uma ou todas as instâncias, o que quebra o encapsulamento da classe. Assim, o ideal é que **no caso de getters de Coleções sejam retornados Coleções imutáveis destas**. Em Java isto pode ser realizado com a classe Collections da biblioteca base.
As agregações e composições são espécies de associações, representando relacionamentos do tipo parte e todo. Agregações são relacionamentos do tipo parte e todo em que ambas as entidades podem existir de modo independente. No caso das composições, a parte não existe sem o todo.

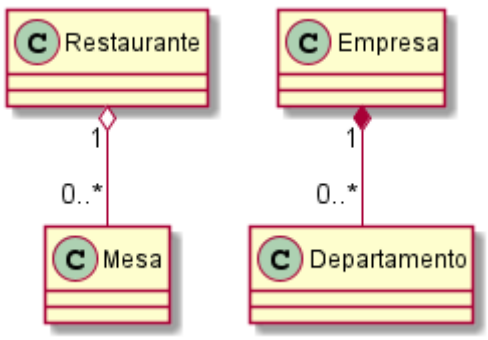


Figura 3.3: Exemplo de agregação e composição

A figura acima demonstra os dois tipos de associação. Um Restaurante é composto por Mesas, mas as Mesas

[5] https://blog.cleancoder.com/uncle-bob/2019/06/16/ObjectsAndDataStructures.html

podem existir sem o Restaurante (p. ex., o Restaurante fechou e vendeu as Mesas para outro). Neste caso, há uma associação do tipo agregação.
Uma Empresa é composta por vários Departamentos e estes não podem existir sem a Empresa. Neste caso, há uma associação do tipo Composição.
Em termos de implementação, não há muitas diferenças em termos de estrutura de dados, mas há uma transmissão de significado que pode ser importante. Assim, pode não fazer sentido a criação de um Departamento sem uma Empresa associada, mas pode fazer sentido uma Mesa sem um Restaurante.

3.1.2. Herança

A herança é uma relação polimórfica em que a subclasse (a Classe que herda) possui o mesmo tipo da superclasse (a classe herdada). O que se considera herdado são todos os atributos e métodos da superclasse.

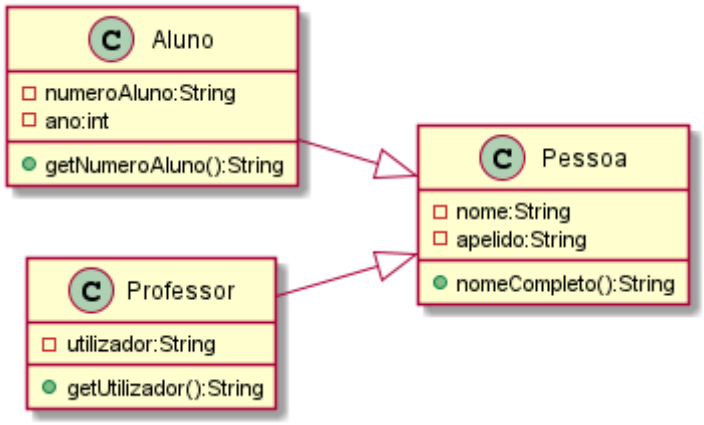


Figura 3.4: Exemplo de herança

A figura acima ilustra um exemplo de herança em que temos as classes Aluno e Professor como sendo subclasses de Pessoa, ou seja, Alunos e Professores são Pessoas, ou têm o mesmo Tipo. Todas as instruções abaixo são válidas e irão compilar sem qualquer problema.

```
Aluno aluno=new Aluno();
Professor professor=new Professor();

Pessoa aluno2=new Aluno();
Pessoa professor2=new Professor();
```

Código 3.1: Como instanciar subclasses

Há pelo menos duas vantagens em utilizar relacionamentos de herança. Reutilização de código, ou seja, não é necessário repetir os mesmo atributos e métodos nas subclasses Aluno e Professor. A segunda vantagem é desenhar as classes de modo que as superclasses sejam abstrações das subclasses, de modo que há um conjunto de comportamentos da superclasse que são concretizadas de formas distintas pelas subclasses.

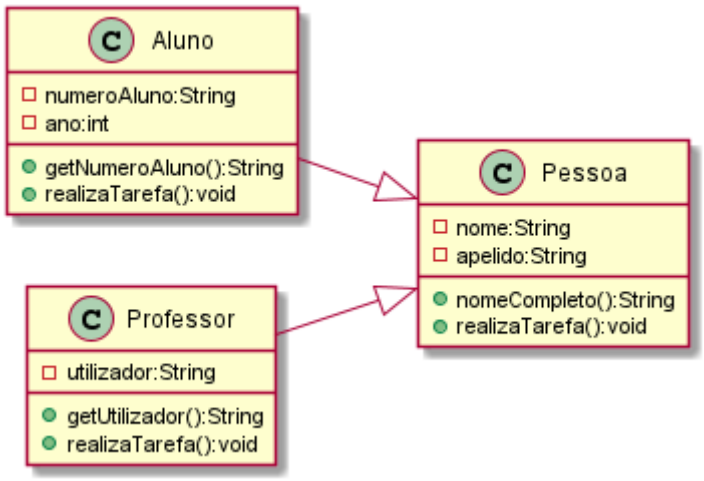


Figura 3.5: Exemplo de herança com comportamento abstrato

No diagrama acima, a superclasse Pessoa agora possui um método chamado realizaTarefa que será herdado pelas subclasses Aluno e Professor. Considerando que cada subclasse realiza as suas tarefas de uma forma diferente (Alunos assistem aulas, Professores ministram as aulas), cada subclasse **pode concretizar** o método realizaTarefa de formas distintas.

```
abstract class Pessoa{
    //atributos
    void abstract realizaTarefa();
}

class Aluno extends Pessoa{
    //atributos
    void realizaTarefa(){
        System.out.println("Aluno "+this.getNomeCompleto()+"
        assiste a aula");
    }
}

class Professor extends Pessoa{
    //atributos
    void realizaTarefa(){
        System.out.println("Professor "+this.getNomeCompleto()+"
        ministra a aula");
    }
}
```

Código 3.2: Como estender superclasses

Como todas as subclasses possuem o mesmo tipo da superclasse elas podem ser tratadas de forma identica, como o código abaixo demonstra. Em tempo, a keyword abstract é utilizada quando uma classe apenas declara um método sem o implementar, o que permite que a concretização do método seja delegada para as subclasses.

```
//Cria-se uma coleção de Pessoas
List<Pessoa> pessoas=new ArrayList<>();
//inclui-se na coleção um Aluno e um Professor
pessoas.add(new Aluno());
pessoas.add(new Professor());
//itera-se por todas as pessoas e para cada uma invoca-se o método realizaTarefa
//invoca-se a implementação para cada subclasse
//não é necessário realizar qualquer verificação
for(Pessoa pessoa:pessoas){
    pessoa.realizaTarefa();
}
```

Código 3.3: Como utilizar polimorfismo de subclasses

3.1.3. Implementação

A relação de implementação é a que existe entre Classes e Interfaces. Como já foi dito antes, uma interface é um conjunto de métodos públicos e impõe uma obrigação de concretização dos métodos pelas classes que a implementem. Este é o contrato imposto pelas interfaces às classes que as implementem. De outro modo, isto implica que as classes que implementam uma determinada interface **são do mesmo tipo que a interface**.

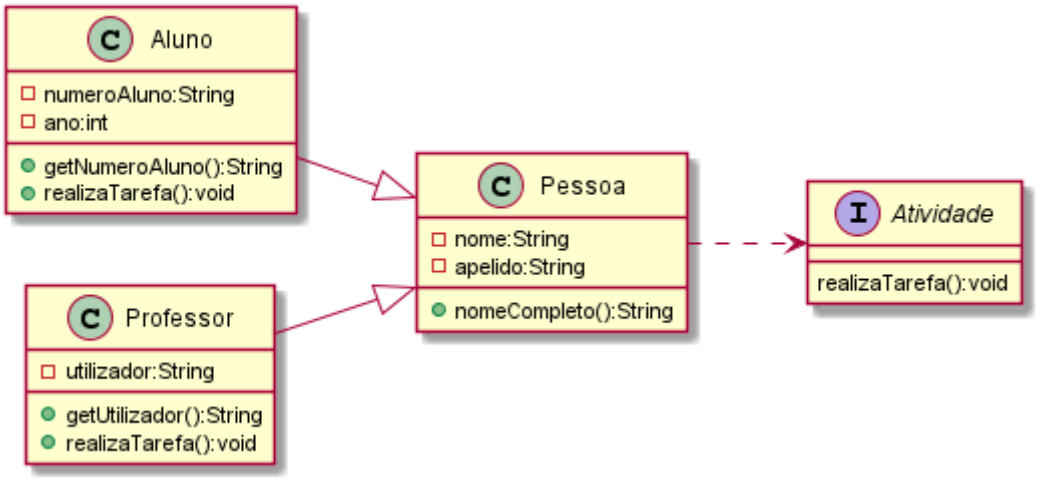


Figura 3.6: Exemplo de implementação de interface

De modo semelhante com a herança, as classes que implementam uma interface podem ser instanciadas como pertencentes ao Tipo da interface. No caso em análise, as classes Aluno e Professor herdam também a implementação da interface, logo pertencem a este Tipo.

```
Aluno aluno=new Aluno();
Professor professor=new Professor();

Pessoa aluno2=new Aluno();
Pessoa professor2=new Professor();

Atividade aluno3=new Aluno();
Atividade professor3=new Aluno();
```

Código 3.4: Como instanciar subclasses com interfaces

```
//Cria-se uma coleção de Atividade
List<Atividade> atividades=new ArrayList<>();
//inclui-se na coleção um Aluno e um Professor
atividades.add(new Aluno());
atividades.add(new Professor());
//itera-se por todas as atividades e para cada uma invoca-se o método realizaTarefa
//invoca-se a implementação para cada subclasse
//não é necessário realizar qualquer verificação
for(Atividade atividade:atividades){
    atividade.realizaTarefa();
}
```

Código 3.5: Como utilizar polimorfismo de interfaces

3.2. Exercício

Elaborar o diagrama de classes para o enunciado do caso de estudo.

Tópico 4: Padrão de Arquitetura e Frameworks

Docente: Alessandro Moreira

4.1. MVC

A arquitetura de um sistema ou software define os módulos ou componentes deste sistema ou software de modo que as funcionalidades necessárias possam ser implementadas. A decomposição de uma aplicação informática faz-se de modo que este seja organizado e desacoplado (decoupled), ou seja, as dependências dos componentes não sejam rígidas, possibilitando a a sua modificação, logo a manutenção global da aplicação.
Há diversos padrões de arquitetura de sistemas, modos de organizar os seus módulos, de modo a facilitar a compreensão dos membros das equipas de desenvolvimento acerca das funções destes módulos, as suas dependências, o comportamento esperado. O padrão de arquitetura MVC (Model-View-Controller) define 3 grandes blocos de componentes que uma aplicação pode ter. O Modelo (Model) deve conter toda a lógica do negócio, bem como lidar com os dados necessários para que as funcionalidades sejam implementadas. A Vista (View) refere-se à interface que a aplicação deve fornecer ao cliente de modo que este possa utilizar as funcionalidades apropriadas. O Controlador (Controller) é o mediador entre a Vista e o Modelo, devendo receber os pedidos vindos do cliente e encaminhar ao Modelo para que este realize o processamento necessário e devolver o resultado de volta ao cliente.
O padrão MVC é bastante utilizado em aplicaçãoes web e um exemplo de diagrama de blocos pode ser visto na figura 4.1.

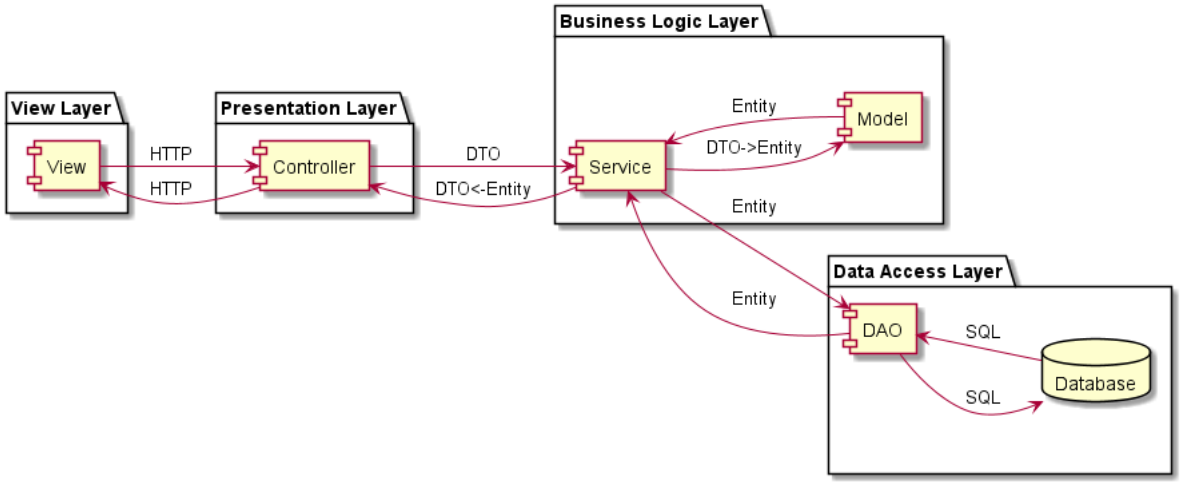


Figura 4.1: MVC

4.2. Spring (Boot)

A framework Spring tem como objetivo simplificar a criação de aplicações profissionais reduzindo a quantidade de configurações que o desenvolvedor precisa fazer. Para além de simplificar, impõe um conjunto de restrições que incentivam boas práticas de programação, especialmente a utilização extensiva de injeção de dependências. A framework Spring possui um conjunto de módulos que lidam com questões de infraestrutura, como acceso a bases de dados, transações, comunicação via HTTP e segurança.
O projeto Spring Boot faz parte da framework, mas utiliza o mote *Convention over configuration*, que traz um conjunto de parametros de configurações por defeito, de modo a permitir que uma aplicação possa ser rapidamente implementada. Tanto o projeto Spring Framework quanto Spring Boot e outros, são open source

e contam com grande participação da comunidade e têm os códigos-fonte disponível no Github (`https://github.com/spring-projects`).
O IDE Intellij IDEA, na versão ultimate, tem um bom suporte para projetos Spring e Spring Boot, contando com muitos plugins para o efeito. Especificamente para o Spring Boot, o IDE reflete a estrutura de criação de projeto usada pelo site Spring Initializr (`https://start.spring.io/`), como se pode ver em 4.2. Se a opção em destaque não for exibida, deverá ser necessário instalar ou habilitar o plugin Spring Boot no menu Settings, como mostra a figura 4.3.

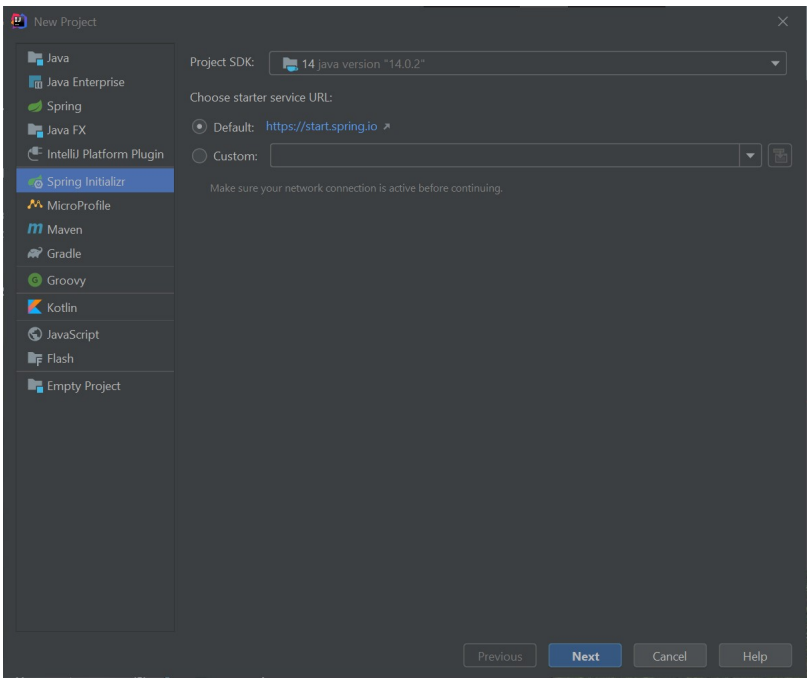


Figura 4.2: Novo projecto

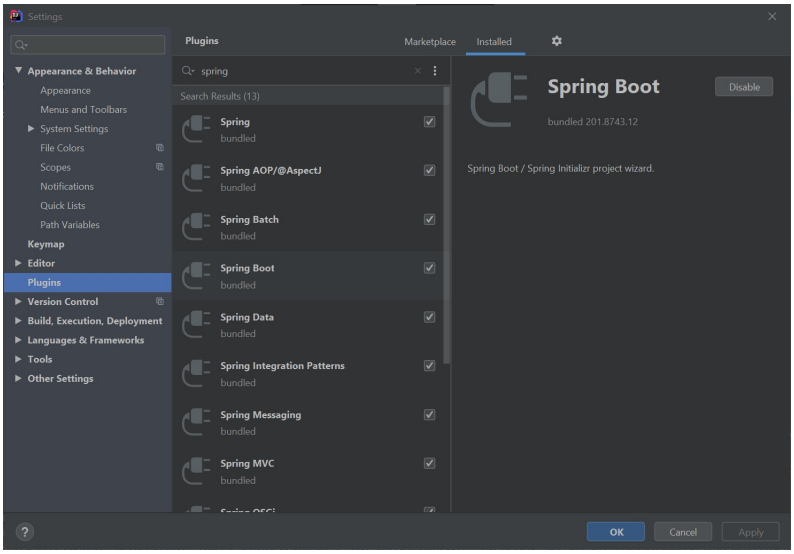


Figura 4.3: Habilitar o plugin spring boot

A figura 4.4 mostra a janela seguinte, em que se deve definir o Group (namespace base das packages do projeto) e Artifact (que é o nome da aplicação). É importante não deixar nenhum dos campos em branco. Pode-se, ainda, escolher a versão do jdk a ser utilizado, o que se recomenda que seja 11 ou superior.

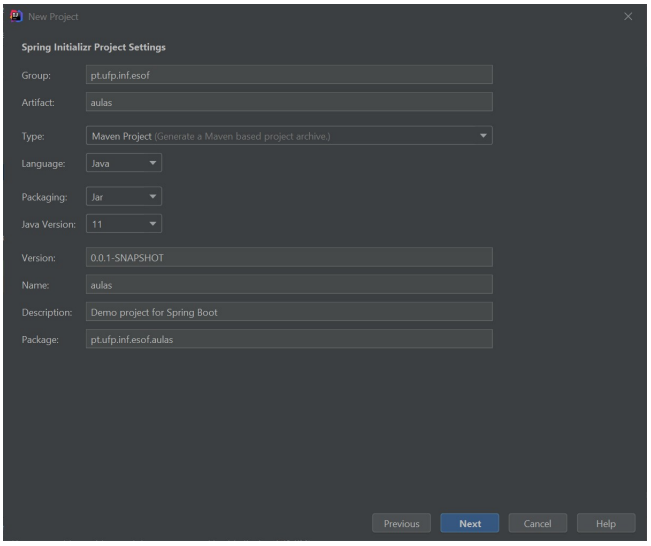


Figura 4.4: Identificação do projeto

Adiante na criação de um novo projeto segue a imagem 4.5 em que se deve escolher as dependências ou bibliotecas nativas do projeto Spring Boot que suportam as necessidades da aplicação. A mesma figura exibe as dependências sugeridas para as aulas e o projeto do semestre.

- Spring Boot DevTools: possibilita o hot reload da aplicação. Basta fazer build da aplicação após alteração no código para que esta se reflita, sem a necessidade de reiniciar toda a aplicação
- Lombok: Diminui a quantidade de código chamado boilerplate, os códigos muito repetidos, como os getters e setter
- Spring Web: Inclui e autoconfigura a infraestrutura necessária para um projeto web, inclusivamente com um servidor embebido no código, o que facilita a implantação do projeto finalizado
- Spring Data JPA: Biblioteca que suporta o acesso a base de dados com um ORM (por defeito o Hibernate), o que diminui a necessidade de código SQL
- H2 Database: Gestor de base de dados relacional em memória, útil para ambiente de desenvolvimento. Se incluída na dependência, será a base de dados escolhida por defeito pela aplicação
- PostgresSQL Driver: Possibilita a comunicação com o SGBD Postgres, que é uma opção para implantação da aplicação em modo de produção

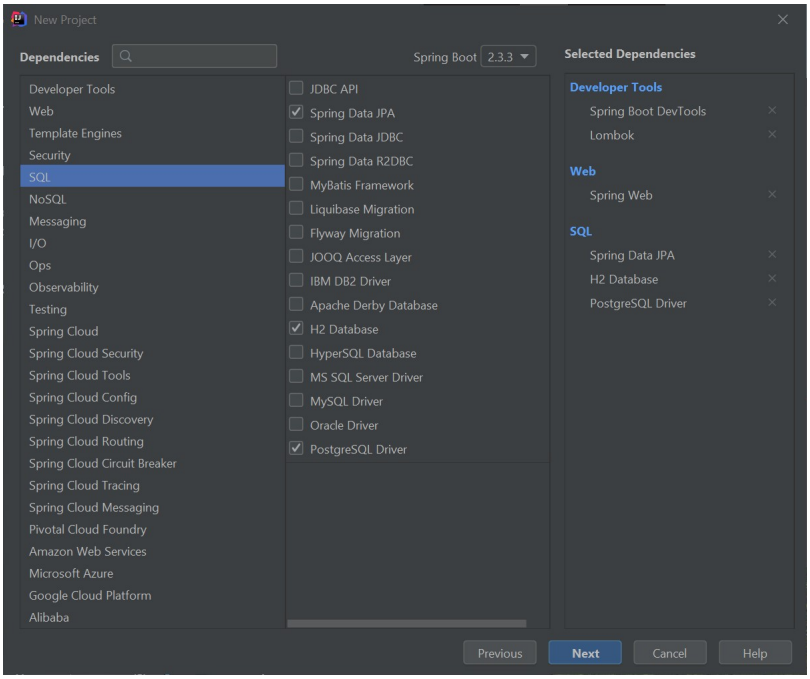
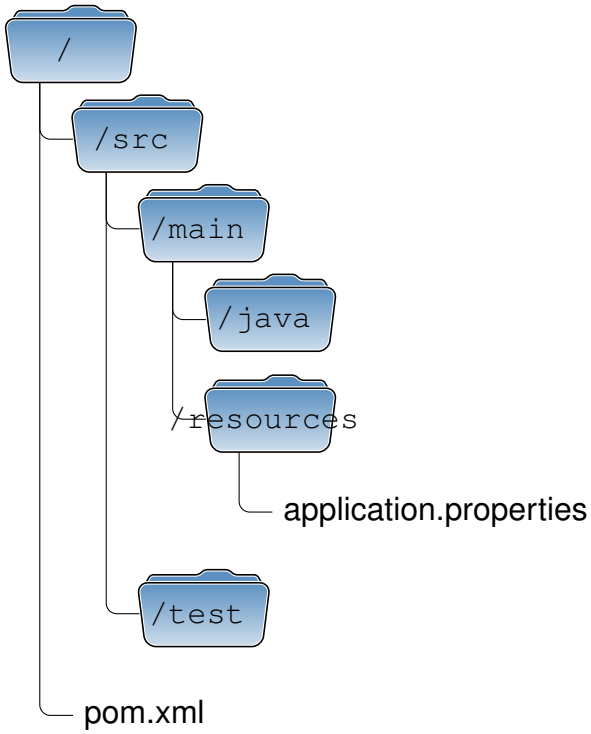


Figura 4.5: Dependências recomendadas

Após à escolha do nome do projeto, o IDE irá processar o projeto até que no final terá um aspeto semelhante ao da figura 4.6:



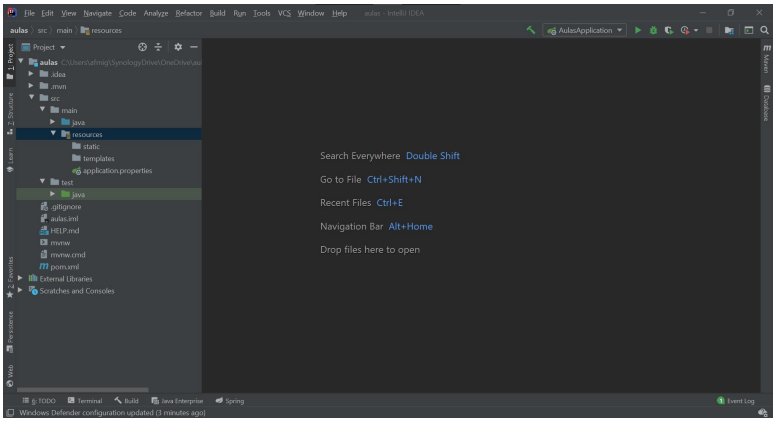


Figura 4.6: Estrutura de um projecto Spring Boot

O ficheiro pom.xml é o mais importante da ferramenta Maven, que visa, entre outros objetivos, facilitar a gestão das bibliotecas e a criação de ficheiros binários Java para implantação. As figuras 4.7 e 4.8 mostram como deve aparentar o ficheiro pom.xml após a criação do projeto. Dentro da etiqueta <dependencies> devem estar a lista de bibliotecas do projeto Java. Cada biblioteca é identificada pela etiqueta <dependency> que é composta 3 campos, <groupId>, <artifactId> e <version>, também conhecidas como coordenadas Maven para uma determinada biblioteca. Todas as bibliotecas públicas são acessíveis pelo Maven Central Repository, cujo motor de pesquisa mais utilizado é o `https://mvnrepository.com/`. Vê-se nas figuras as dependências escolhidas no processo de criação do projeto.

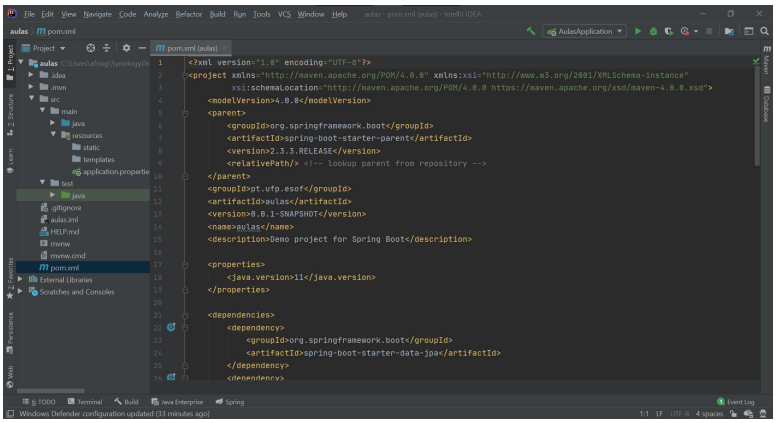


Figura 4.7: Maven pom.xml [1]

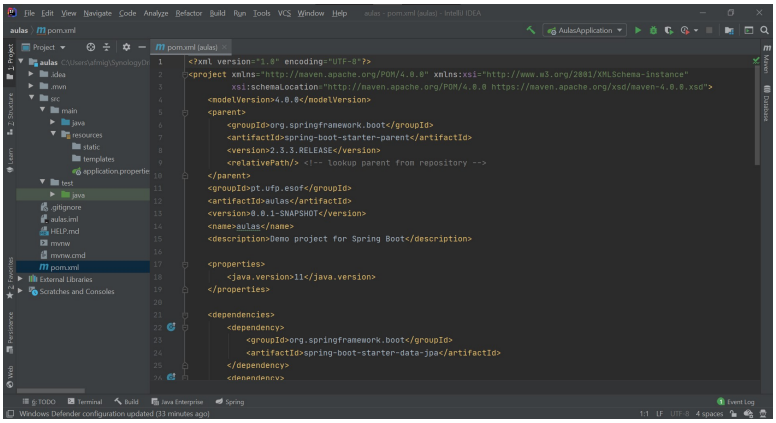


Figura 4.8: Maven pom.xml [2]

Dentro da raiz do projeto está a pasta /src que tem duas subpastas, /main e /test. A pasta /test conterá os testes automatizados da aplicação. A pasta /main possui duas subpastas, a /java, que conterá o código-fonte da

aplicação e a /resources que conterá ficheiros de configuração e ficheiros de suporte à vista, como .html, .css e .js.
O ficheiro application.properties é o principal ficheiro de configurações de um projeto Spring Boot, onde se pode alterar parametros como a porta utilizada pelo servidor web, o url da base de dados, níveis de logging. A imagem 4.9 mostra um exemplo de coniguração que deve ser utilizado para que se possa aceder facilmente a consola da base de dados H2, o que será visto em aulas seguintes.

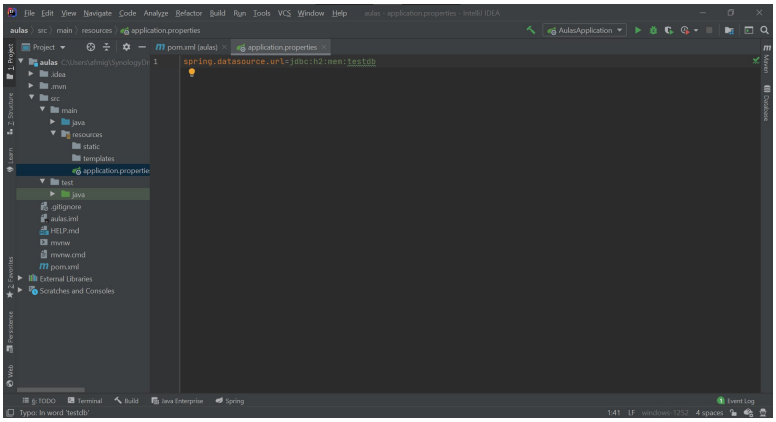


Figura 4.9: Ficheiro application.properties

No nível mais profundo da pasta /main/java estará o ficheiro principal da aplicação, que é utilizado para iniciá-la. Este ficheiro será visto também em aulas seguintes.

4.3. Exercicio

1. Deverão criar uma nova package chamada models. Deverá estar um nível abaixo da aplicação principal
2. Nesta nova package deverão implementar as classes definidas pelo diagrama de classes do caso de estudo

Tópico 5: Injeção de Dependência

Docente: Alessandro Moreira

5.1. Injeção de Dependencia

Dá-se o nome de dependência ao relacionamento entre classes A e B, na qual uma necessita da outra para realizar sua atividade. Assim, se A precisa de B para fazer algo, B é uma dependência de A. No exemplo da classe Viatura, uma hipótese de implementação poderia ser a que segue:

```
class Pessoa{

}
class Viatura{
    private ArrayList<Pessoa> passageiros=new ArrayList<>();
}
```

Código 5.1: Classes fortemente dependentes

Uma implementação deste tipo traz uma dependencia entre a classe Viatura e a classe ArrayList. Todo objeto viatura é instanciado com um objeto do tipo ArraysList. Há uma forte dependência da Viatura com o ArrayList, o que impede modificações desta dependência.
A classe ArrayList é a implementação mais utilizada da interface List que estende a interface Collection. Mais informações sobre as Coleções em Java podem ser encontrada na ligação `https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/doc-files/coll-reference.html`.
As coleções que implementam a interface List possuem um conjunto de propriedades, como garantia da ordem de inserção e possibilidade de objetos duplicados.
Outro tipo de coleção é o conjunto (Set), no qual não há uma garantia de ordem de inserção e não é possível haver objetos duplicados.
As implementações de cada interface deverão respeitar estas características e possibilitam que o desenvolvedor possa utilizá-las sem se preocupar com efeitos adversos.
De volta à questão das dependências, uma dependência forte ou estrita dificulta a modificação de um programa, caso as regras de negócio mudem, bem como a reutilização do código, visto que aquela classe só funciona com aquela dependência. Assim, o código inicialmente apresentado pode ser modificado por outro que diminui a dependência da classe Viatura à classe ArrayList.

```
class Viatura{
    //uma viatura por defeito é criada com um ArrayList
    private List<Pessoa> passageiros=new ArrayList<>();

    public Viatura(){}

    //injeção de dependência via constructor
    public Viatura(List<Pessoa> passageiros){
        this.passageiros=passageiros;
    }
    //injeção de dependência via setter
    public void setPassageiros(List<Pessoa> passageiros){
        this.passageiros=passageiros;
    }
}
```

Código 5.2: Formas de injeção de dependência

O código apresentado possui algumas características. Por defeito, a coleção utilizada será *ArrayList*, mas nada impede que o desenvolvedor utilize uma outra implementação, aumentando a flexibilidade do programa.

Apresenta ainda uma forma de reduzir a dependência de uma classe, chamada de injeção de dependência (*Dependency Injection*). A injeção de dependência consiste na possibilidade de incluir a dependência na classe, em vez da classe ser a única responsável por incluir a sua própria dependência.
Assim, o cliente (qualquer zona de código que necessite utilizar a classe dependente), pode definir qual a implementação da interface *List* deseja utilizar.

```
class Main{
    public static void main(String args[]){
        List<Pessoa> pilhaPassageiros=new Stack<>();
        pilhaPassageiros.add(new Pessoa());
        Viatura viaturaPilha=new Viatura(pilhaPassageiros);

        List<Pessoa> listaLigadaPassageiros=new LinkedList<>();
        listaLigadaPassageiros.add(new Pessoa());
        Viatura viaturaListaLigada=new Viatura(listaligadaPassageiros);

    }
}
```

Código 5.3: Como utilizar a injeção de dependência

5.2. DI em Spring

5.2.1. Beans e Components

A *framework* Spring tem como objetivo simplificar a criação de aplicações profissionais, trazendo um conjunto de funcionalidades. A principal filosofia por trás da *framework* é permitir que o desenvolvedor implemente a infraestrutura da aplicação (p. ex., acesso a base de dados, comunicação por HTTP) com recurso a classes simples, chamadas POJO (Plain Old Java Object). **Um POJO é uma classe que deve possuir um construtor por defeito, getters e setters**.
A *framework* então pode ficar responsável pela criação destas classes e entregar instâncias destas classes nas zonas de código que o desenvolvedor definir. O modo como a *framework* realiza esta operação é por meio de injeção de dependências.
O código abaixo demonstra como a *framework* permite estas operações:

```
@Component
public class TestClass{

    private String attribute;

    public void doSomething(){
        System.out.println(this);
    }

    public void setAttribute(String attribute){
        this.attribute=attribute;
    }

    public String getAttribute(){
        return this.attribute;
    }
}
```

Código 5.4: Criação de uma classe com a anotação @Component

A anotação *@Component* é um dos estereótipos definidos pelo Spring para classes a serem geridas. Outros estereótipos, que serão vistos mais adiante são *@Repository*, *@Service*, *@Controller*. Uma classe anotada

por um destes estereótipos é incluída no contentor da *framework* que pode ser acedido pelas classes que implementam a interface *ApplicationContext*.

```
@Component
public class Bootstrap implements ApplicationListener<ContextRefreshedEvent>{

    //injeção por atributo
    @Autowired
    private TestClass test;

    //injeção por setter
    /*
    @Autowired
    public void setTest(TestClass test){
        this.test=test;
    }
    */

    //injeção por construtor
    /*
    @Autowired
    public Bootstrap(TestClass test){
        this.test=test;
    }
    */

    @Override
    public void onApplicationEvent(ContextRefreshedEvent contextRefreshedEvent) {
        test.doSomething();
    }
}
```

Código 5.5: Criação de uma classe que arranca junto com a aplicação

O código acima demonstra como possibilitar execução de código quando do arranque (ou reload) da aplicação. As classes que implementam a interface *ApplicationListener* podem utilizar como genéricos as classes *ContextRefreshedEvent*, *ContextStartedEvent*, *ContextClosedEvent*, *ContextStoppedEvent*. A anotação @Component é utilizada para que o Spring a inclua no seu contentor e possa executar o método onApplicationEvent, que irá invocar o método *doSomething* do atributo *test*.
Uma instância da classe *TestClass* será injetada no atributo *test* graças à anotação *@Autowired*. Se esta anotação for comentada, o resultado será *NullPointerExcpetion*, demonstrando que o atributo é instanciado pela *framework* Spring. Estão presentes no código 3 formas de injetar uma dependência com Spring, injeção por atributos, por *setters* (pode ser qualquer método) e por construtor. Em regra, as injeções por atributo devem ser evitadas por exigirem da *framework* que utilize *reflection* para identificar os atributos da classe que possui a dependência. O método mais recomendado para injetar dependências é por construtor, que possui a vantagem adicional de permitir que a dependência seja imutável.
O mesmo efeito pode ser conseguido com a anotação @Bean. Esta anotação deve ser utilizada em classes de configuração, aquelas que possuem a anotação @Configuration. A classe principal criada pelo Initializr do Spring Boot tem a anotação @SpringBootApplication que é uma meta-anotação composta por @Configuration, @ComponentScan e @EnableAutoConfiguration.

```
@SpringBootApplication
public class AulasApplication {

    public static void main(String[] args) {
        SpringApplication.run(AulasApplication.class, args);
    }

    @Bean
    public TestClass testClass(){
        return new TestClass();
    }
```

```
}
```

Código 5.6: Utilização da anotação @Bean

Para o efeito da demonstração é necessário que a anotação @Component seja removida da classe TestClass. O que a anotação @Bean faz é indicar à *framework* que inclua no contentor uma instância da classe TestClass. A principal diferença entre a utilização das anotações @Bean e @Component é que a primeira permite a injeção de dependência de classes de bibliotecas de terceiros.

5.3. Exercícios

1. Refatorizar a classe TestClass para que implemente uma interface chamada Test que deve declarar o método doSomething.
2. Criar uma nova classe que implementa a nova interface
3. Modificar a classe Bootstrap para que dependa da interface e não da implementação
4. Utilizar a anotação @Bean para injetar na classe Bootstrap um grafo pesado dirigido com 10 nós, utilizando a biblioteca de princeton. Devem incluir a biblioteca no pom.xml com a etiqueta abaixo e carregar as modificações no maven:

```
<dependency>
    <groupId>com.googlecode.princeton-java-algorithms</groupId>
    <artifactId>algorithms</artifactId>
    <version>4.0.1</version>
</dependency>
```

5. Povoar o grafo com 10 arestas e imprimir o resultado

Tópico 6: Implementação do modelo de Classes e Mapeamento JPA

Docente: Alessandro Moreira

6.1. Anotações Lombok

Uma das bibliotecas incluídas no projeto é a Lombok e o principal objetivo é diminuir a quantidade de código repetitivo (boilerplate) nas aplicações Java. Esta biblioteca disponibiliza uma quantidade de anotações que podem ser utilizadas a nível da classe ou a nível de atributos para gerar métodos sem "poluir"o código. As principais anotações são:

- @Getter: Gera getters para todos os atributos ou para um atributo específico
- @Setter: Gera setters para todos os atributos ou para um atributo específico
- @ToString: Gera o método toString que inclui todos os atributos, para um atributo específico ou para excluir um atributo deste método
- @EqualsAndHashcode: Gera os métodos equals e hashCode que incluem todos os atributos, para um atributo específico ou para excluir um atributo destes métodos
- @NonNull: Define o atributo como obrigatório
- @NoArgsConstructor: A classe anotada terá um constructor por defeito
- @RequiredArgsConstructor: A classe anotada terá um constructor que inclui todos os atributos obrigatórios (os anotados com @NonNull)
- @AllArgsConstructor: A classe anotada terá um constructor que inclui todos os atributos
- @Data: Meta-anotação que inclui as anotações @Getter, @Setter, @ToString, @EqualsAndHashcode e @RequiredArgsConstructor

```
@Getter
@Setter
public class Person{
    @ToString.Include
    private String firstName;
    @ToString.Include
    private String lastName;
    @EqualsAndHashcode.Include
    private String username;
}
```

Código 6.1: Utilização de anotações Lombok

O resultado do código acima é que todos os atributos terão getters, setters, o método toString() incluirá os atributos firstName e lastName e os métodos equals() e hashCode() icluirão apenas o atributo username. De ressaltar que os métodos são gerados durante a compilação do código, pelo que o IDE pode acusar o erro na utilização do código porque não encontra os métodos definidos no código. Para resolver isto, no Intellij IDEA, há um plugin, chamado Lombok, que permite a integração do IDE com a biblioteca.

6.2. JPA

O JPA, Jakarta Persistence (anteriormente conhecido como Java Persistence API) é o padrão Java para bibliotecas que lidem com a gestão de persistência de dados e ORM (Mapeamento Objeto Relacionamento). As bibliotecas de ORM visam reduzir ou eliminar a utilização de linguagens de consulta (p. ex. SQL) para leitura, escrita ou mesmo construção das bases de dados. A adesão das bibliotecas ao JPA simplifica a tarefa dos desenvolvedores, de modo que basta o entendimento do padrão para que possam utilizar qualquer uma das implementações do padrão. A implementação escolhida pela framework Spring é a biblioteca Hibernate, já configurada para utilização devido à inclusão da biblioteca Spring Data JPA.
O acesso aos dados via Spring e Spring Boot é dividido em duas etapas:

1. Declarar as entidades que irão representar as tabelas da base de dados e mapear os relacionamentos
2. Declarar os repositórios para interagir com a base de dados

6.2.1. Entity

O JPA define algumas anotações para tornar um POJO em uma entidade. Uma entidade em termos de JPA implica em interagir com a base de dados por meio de objetos Java. O ORM irá utilizar estas classes para ler da base de dados para o objeto Java e ler do objeto Java para escrever para a base de dados.
As principais anotações para este efeito são:

- @Entity: Define a classe anotada como uma entidade JPA. Obrigatoriamente deverá possuir um id, que representará a chave primária na base de dados.
- @Id: Define o atributo da classe que será o id.
- @GeneratedValue: Deve ser usada em conjunto com @Id. Define o modo como o valor do id será gerado. Equivale ao auto-increment na base de dados.
- @Table: Pode ser utilizada para definir o nome da tabela na base de dados. Se não for utilizada, o nome será o da classe (pode gerar problemas se o nome da classe for uma palavra restrita na base de dados).
- @Column: Pode ser utilizada no atributo para nomear o campo na base de dados. Se não for utilizada, o nome será o do atributo.

```
@Data
@Entity
@Table(name="pessoa")
public class Person{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name="nome")
    private String firstName;
    @Column(name="apelido")
    private String lastName;
    @Column(name="utilizador")
    private String username;
}
```

Código 6.2: Utilização de anotações JPA

Se o ficheiro application.properties possuir a linha ***spring.datasource.url=jdbc:h2:mem:testdb***, é possível verificar se o código acima se refletiu na base de dados H2. Para tal, deve arrancar o projeto e, não havendo erros de execução, deve-se ir ao navegador e utilizar o endereço ***localhost:8080/h2-console***. Na janela mostrada pela figura 6.1, deve-se garantir que o valor no campo JDBC URL é o mencionado no ficheiro application.properties. A inclusão da linha no ficheiro application.properties é necessária visto que na ausência deste parâmetro, a aplicação gera um url aleatório, que pode ser visto na consola do IDE. O utilizador por defeito é sa e não há uma palavra-passe.
Após conectar à base de dados, deverá aparecer a janela exibida na figura 6.2, onde deverá aparecer a tabela Pessoa (definido pela anotação @Table), com os campos definidos pela anotação @Column.

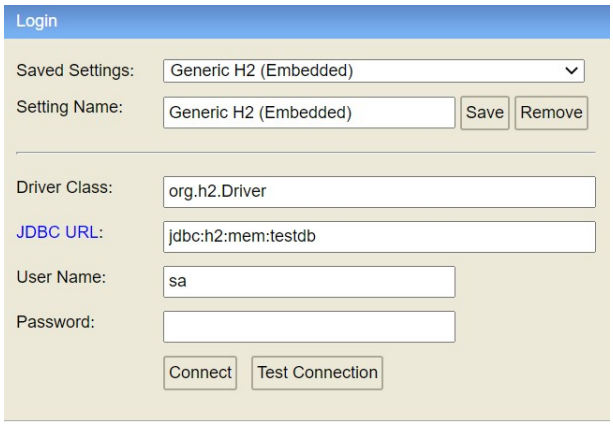


Figura 6.1: Consola H2

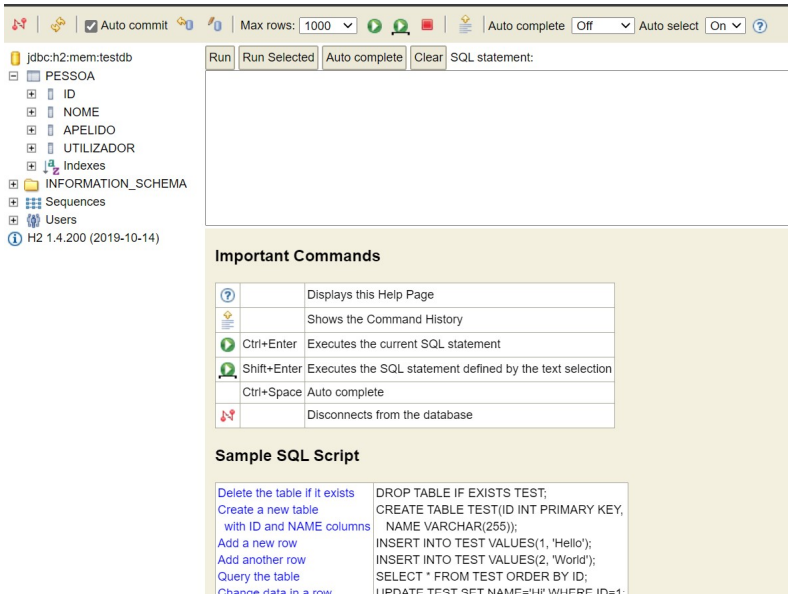


Figura 6.2: Consola H2

6.2.2. Mapeamentos

Após definir classes como entidades JPA é necessário mapear os relacionamentos que estas classes possuem. São 4 os tipos de relacionamento que as classes podem possuir e são representadas pelas anotações abaixo, que devem ser utilizadas nos atributos que representam a cardinalidade da relação:

- @OneToOne
- @OneToMany

- @ManyToOne
- @ManyToMany

@OneToOne

A figura abaixo apresenta o diagrama de classes para o relacionamento entre uma pessoa que possui uma morada. Neste exemplo, não faz muito sentido o relacionamento ser bidirecional, pelo que a classe Address não possui referência para a classe Person. A classe Person deve possuir referência para a classe Address e este atributo deve ser anotado com @OneToOne, como mostra o código 6.3.

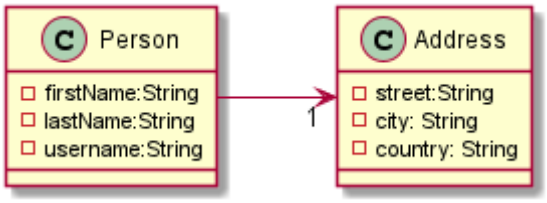


Figura 6.3: Implementação do mapeamento @OneToOne

```
@Data
@Entity
@Table(name="pessoa")
public class Person{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name="nome")
    private String firstName;
    @Column(name="apelido")
    private String lastName;
    @Column(name="utilizador")
    private String username;
    @OneToOne(cascade=cascade = CascadeType.ALL)
    private Address address;
}

@Data
@Entity
public class Address {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String street;
    private String city;
    private String country;

}
```

Código 6.3: Mapeamento @OneToOne

Todas as anotações de mapeamento possuem alguns argumentos, dentre os quais, cascade. Este argumento propaga operações da classe que possui o atributo, neste caso Peson, para a classe referenciada, Address. Estas operações podem ser:

- Persist: quando a classe referenciadora é gravada na base de dados (tornada persistente), a classe referenciada também será

- Merge: quando houver modificação dos dados nas duas classes, a atualização dos dados na classe referenciadora implica na atualização da classe referenciada
- Remove: se a classe referenciadora for removida da base de dados, a classe referenciada também será (em regra, nunca deve ser utilizada com mapeamentos @ManyToMany e nas demais, irá depender da regra do negócio)
- Detach: O objeto com esta operação deixa de estar gerido pelo ORM e precisa ser explicitamente marcado como tal. As informações permanecem na base de dados (difere do remove), mas modificações no objeto não serão persistidas até que uma operação de merge seja executada
- Refresh: A operação de refresh implica em descartar o objeto da entidade em memória e substituir pelos dados vindos da base de dados.
- All: todas as operações acima

As entidades JPA possuem alguns estados relevantes para a interação com a base de dados, cujo diagrama de estados encontra-se na figura 6.4[6].

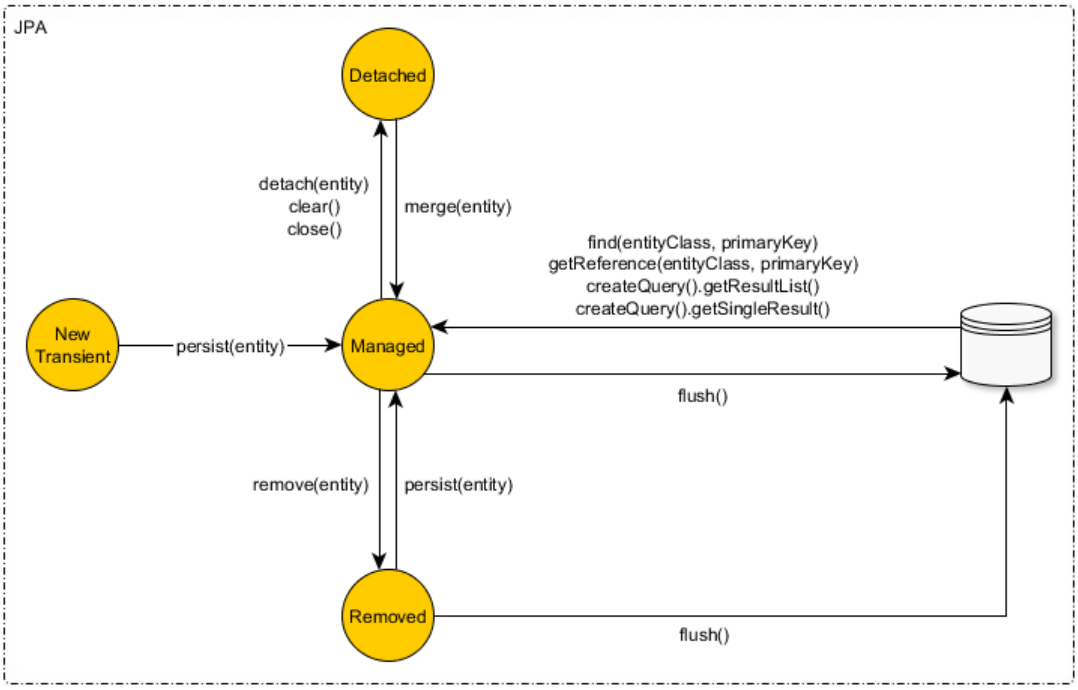


Figura 6.4: Estados JPA

Um erro muito comum é encontrado quando se tenta gravar na base de dados (persist) um objeto que possui referência a outro objeto que ainda não está na base de dados (transient). A mensagem de exceção diz *object references an* ***unsaved transient instance*** *- save the transient instance before flushing*, ou seja antes de guardar o objeto, é preciso guardar a referência que é um objeto novo (transient). Ao parametrizar a anotação de mapeamento com o CascadeType.All, a operação de persist será propagada para a referência que será automaticamente guardada na base de dados.

Com ou sem cascade, o resultado do mapeamento @OneToOne na consola do H2 é exibida na figura 6.5, em que a tabela Pessoa possui uma chave estrangeira para a tabela Address (ADDRESS_ID).

[6] `https://vladmihalcea.com/a-beginners-guide-to-jpa-hibernate-entity-state-transitions/`

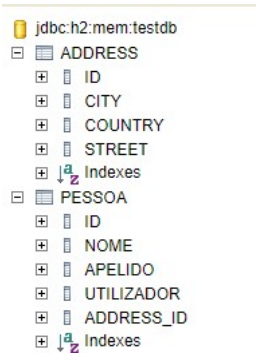


Figura 6.5: Resultado da anotação @OneToOne na base de dados

@OneToMany-@ManyToOne

Um exemplo de relacionamento @OneToMany é o de uma pessoa que pode possuir muitas encomendas. Considerando que uma encomenda tenha apenas uma pessoa, é um relacionamento @ManyToOne. A figura 6.6 mostra o diagrama de classes deste relacionamento bidirecional e o código 6.4, a implementação em JPA.

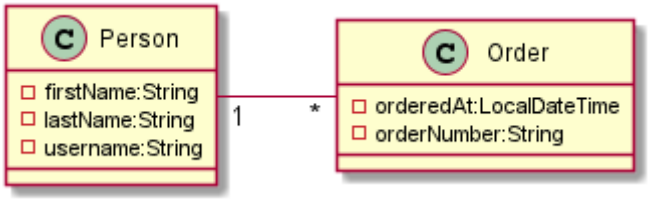


Figura 6.6: Implementação do mapeamento @OneToMany e @ManyToOne

```
@Getter
@Setter
@Entity
@Table(name="pessoa")
public class Person{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name="nome")
    private String firstName;
    @Column(name="apelido")
    private String lastName;
    @Column(name="utilizador")
    private String username;
    @OneToOne(cascade = CascadeType.ALL)
    private Address address;
    @OneToMany(cascade = CascadeType.ALL,orphanRemoval=true)
    private Set<Order> orders=new HashSet<>();

    public void addOrder(Order order) {
        this.orders.add(order);
        order.setPerson(this);
    }

    public void removeOrder(Order order) {
        this.orders.remove(order);
        order.setPerson(null);
    }
}

@Getter
```

```java
@Setter
@Entity
@Table(name = "order_table")
public class Order {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private LocalDateTime orderTime;

    @ManyToOne
    private Person person;

}
```

Código 6.4: Implementação dos mapeamentos @OneToMany e @ManyToOne

Vê-se na anotação @OneToMany da classe Person a utilização do argumento orphanRemoval=true, o que indica que se a classe referenciada deixar de ter referência para esta classe, o ORM deverá remover a classe referenciada. Pode-se ver nesta classe a adição de métodos de gestão de coleções, addOrder e removeOrder.
O método addOrder, para além de incluir o objeto na coleção, faz setPerson atribuindo o objeto person que invocou o método, de modo a garantir a bidirecionalidade do relacionamento. Se o setPerson não fosse chamado, ou o objeto Person de Order seria null ou o desenvolvedor teria de fazer a atribuição após incluir o objeto Order, o que traria código repetido e possibilidade de esquecer de fazer esta operação.
O método removeOrder também busca garantir a bidirecionalidade do relacionamento, mas, para além de remover o objeto Order da coleção de Person, remove a referência de Person do objeto Order, de modo a garantir que este Order agora é orfão, pelo que o ORM poderá proceder à sua remoção.
A classe Order possui uma referência para a classe Person e a anotação @ManyToOne. A classe também é anotada com @Table para modificar o nome da tabela para order_table. Isto é necessário devido ao nome da classe que é uma das palavras restritas em SQL, que, se usada como nome de tabela causa erros de sintaxe. O resultado das anotações podem ser vistas na figura 6.7.

Figura 6.7: Resultado da anotação @OneToMany na base de dados

Vê-se na figura que existem agora 4 tabelas, Address, Order_Table (classe Order), Pessoa (classe Person) e Pessoa_Orders. E última tabela é desnecessária para uma relação de um para muitos, mas foi criada pelo ORM porque não lhe foi indicado qual o lado da relação que será o pai, ou seja quem terá a chave estrangeira. O código 6.5 mostra como utilizar o mappedBy para indicar que Person será mapeado na classe Order como sua chave estrangeira. A figura 6.8 mostra o resultado encontrado na consola do H2 em que a tabela Pessoa_Orders já não é encontrada.

```java
//anotações do exemplo anterior
public class Person{
    //mesmo código do exemplo anterior
```

```
    @OneToMany(mappedBy = "person",cascade = CascadeType.ALL,orphanRemoval=true)
    private Set<Order> orders=new HashSet<>();

    //mesmo código do exemplo anterior
}
```

Código 6.5: Implementação dos mapeamentos @OneToMany e @ManyToOne com mappedBy

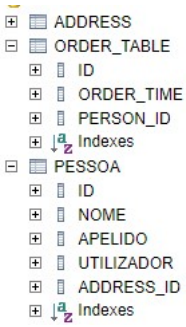


Figura 6.8: Resultado da anotação @OneToMany na base de dados após o uso do argumento mappedby

@ManyToMany

Um exemplo de relacionamento @ManyToMany é o de uma encomenda que possui muitos produtos e produtos podem estar em muitas encomendas. A figura 6.9 mostra o diagrama de classes e o código 6.6, a implementação em JPA.

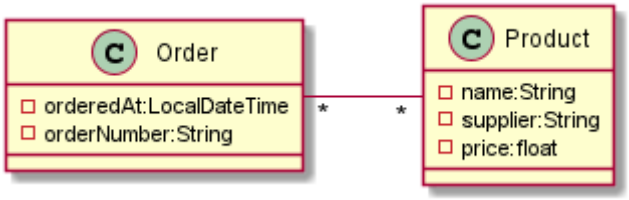


Figura 6.9: Mapeamento @ManyToMany

```
@Getter
@Setter
@Entity
@Table(name = "order_table")
public class Order {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private LocalDateTime orderTime;

    @ManyToOne
    private Person person;

    @ManyToMany(cascade ={CascadeType.DETACH,CascadeType.MERGE,
    CascadeType.PERSIST,CascadeType.REFRESH})
    private Set<Product> products=new HashSet<>();

    public void addProduct(Product product){
        products.add(product);
```

```
        product.addOrder(this);
    }

    public void removeProduct(Product product){
        products.remove(product);
        product.removeOrder(this);
    }

    public float orderPrice(){
        float price=0.0f;
        for(Product product:products){
            price+=product.getPrice();
        }
        return price;
    }

}

@Entity
@Getter
@Setter
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    private float price;

    @ManyToMany
    private Set<Order> orders=new HashSet<>();

    public void addOrder(Order order) {
        orders.add(order);
    }
    public void removeOrder(Order order){
        orders.remove(order);
    }
}
```

Código 6.6: Implementação do mapeamento @ManyToMany

Na classe Order, incluiu-se a coleção de produtos, com a anotação @ManyToMany e os métodos de gestão de coleções. A anotação de mapeamento possui o argumento cascade, que aceita um ou muitos valores, com todas as operações, exceto Remove. Tal se dá para evitar que a remoção de uma encomenda remova também os produtos. Nestas circunstâncias, pode ocorrer de se remover produtos que pertençam a outras encomendas. São poucos os casos em que faz sentido a propagação de Remove em relações @ManyToMany, pelo que se recomenda que seja evitada.
Na classe Product, há também a coleção de encomendas e os métodos de gestão de coleções.
O resultado dos mapeamentos pode ser visto na figura 6.10 que reflete a consola do H2.

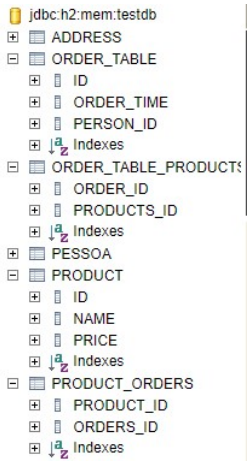


Figura 6.10: Resultado da anotação @ManyToMany na base de dados

Vê-se, portanto, 6 tabelas, 4 referentes às entidades do modelo e as tabelas ORDER_TABLE_PRODUCTS e PRODUCT_ORDERS. Assim como no mapeamento @OneToMany, como não foi usado o argumento mappedBy, o ORM não sabe quem será o pai da relação, então cria uma tabela adicional para garantir a consistência do relacionamento ao custo de mais operações a serem realizadas.
O código 6.7 demonstra a implementação do mapeamento @ManyToMany com o argumento mappedBy e o resultado na consola H2 pode ser encontrado na figura 6.11, que já não apresenta a tabela PRODUCT_ORDERS.

```
//anotações do exemplo anterior
public class Product {
    //mesmo código do exemplo anterior
    @ManyToMany(mappedby="products")
    private Set<Order> orders=new HashSet<>();
    //mesmo código do exemplo anterior
}
```

Código 6.7: Implementação do mapeamento @ManyToMany com o uso do mappedby

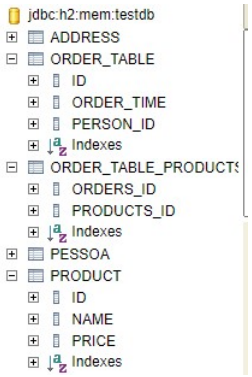


Figura 6.11: Resultado da anotação @ManyToMany na base de dados após o uso do argumento mappedby

6.3. Repository

As anotações vistas na seção anterior resolvem a questão de como as tabelas na base de dados podem ser transformadas em objetos em memória. O modo como se realizam as consultas na base de dados é resolvido com os repositórios. Os repositórios são marcados pelo estereótipo @Repository, o que faz com que a framework faça a gestão das instâncias.
A framework Spring possui uma interface CrudRepository que facilita a interação com a base de dados. A documentação pode ser encontrada em `https://docs.spring.io/spring-data/commons/docs/`

current/api/org/springframework/data/repository/CrudRepository.html e mostra a lista de operações permitidas.
O código em 6.8 mostra um exemplo de como utilizar a interface CrudRepository.

```
@Repository
public interface PersonJPA extends CrudRepository<Person,Long> {
}
```

Código 6.8: Criação de um repositório que estende a interface CrudRepository

É preciso criar uma interface que estende a interface CrudRepository e passar como genéricos a entidade gerida pelo ORM e o tipo que representa o id desta entidade. O exemplo mostra um repositório que lida com as operações relacionadas com a entidade Person.
Um exemplo de utilização deste repositório pode ser encontrado no código 6.9

```
@Component
@Transactional
public class Bootstrap implements ApplicationListener<ContextRefreshedEvent> {

    private final PersonJPA personJPA;

    @Autowired
    public Bootstrap(PersonJPA personJPA) {
        this.personJPA = personJPA;
    }

    @Override
    public void onApplicationEvent(ContextRefreshedEvent contextRefreshedEvent) {
        Person person=new Person();
        person.setFirstName("fn1");
        person.setLastName("ln1");
        person.setUsername("username");

        Address address=new Address();
        address.setStreet("Rua");
        address.setCity("Cidade");
        address.setCountry("Pais");

        person.setAddress(address);

        personJPA.save(person);
    }
}
```

Código 6.9: Utilização do repositório

O código injeta uma instância da interface do repositório PersonJPA na classe Bootstrap por meio do construtor. No método onApplicationEvent é criado um objeto Person, e a ele associa-se um objeto Address, para em seguida, guardar o objeto Person na base de dados via repositório. O resultado desta operação é que, após arrancar o programa, a base de dados será povoada com uma entrada na tabela Pessoa e uma entrada na tabela Address.
A classe Bootstrap possui mais uma anotação, a @Transactional, que permite que o ORM faça a gestão das transações necessárias nas consultas, especialmente aquelas relativas a leituras de tabelas que tenham relacionamento de um para muitos. Aconselha-se a por esta anotação em todas as classes que dependam de repositórios.
Haveria um erro no arranque do programa se a anotação @OneToOne na classe Person não possuísse o argumento cascade a propagar todas operações, em especial o Persist, visto que o ORM iria tentar guardar o objeto Person na base de dados, mas como o objeto Address ainda não foi guardado (transient) uma exceção seria lançada. Uma alternativa à propagação de operações seria guardar primeiro o objeto Address, com o seu próprio repositório, e em seguida guardar o objeto Person.

A figura 6.12 mostra o resultado do código 6.9 na consola do H2 em que vê-se as entradas referentes às tabelas Pessoa e Address.

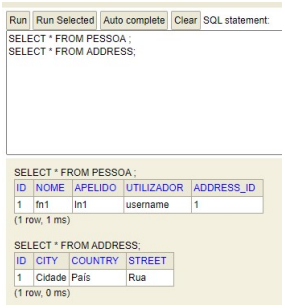


Figura 6.12: Resultado da operação de save na base de dados

O que deve ser notado é que não foi preciso implementar qualquer código SQL para a operação de inserção na base de dados, sendo de responsabilidade do ORM fazê-lo. Nem foi necessário implementar a interface PersonJPA, sendo responsabilidade da framework fazê-lo. Como a interface PersonJPA estende a CrudRepository, herda todos os métodos desta, com destaque para o save, mas há métodos para listar todas as entradas (findAll), pesquisar pelo id (findById), remoção de objeto (delete) e outros.
Como a implementação da interface pela framework utiliza reflexão, ou seja, consegue aceder aos métodos e atributos da classe, ela permite que as subclasses de CrudRepository declarem outros métodos de consulta à base de dados. O código 6.10 mostra um exemplo de método que pode ser declarado.

```
@Repository
public interface PersonJPA extends CrudRepository<Person,Long> {
    Person findByUsername(String username);
}
```

Código 6.10: Método personalizado para a classe Person

O método declarado permite a consulta de uma pessoa por seu username. Importante notar que o findBy (e outras operações permitidas) deve ser seguida do nome de um dos atributos da classe Person e o argumento do método deve ser do mesmo tipo do atributo. A figura 6.13 mostra exemplos de métodos/consultas que a interface CrudRepository permite. Tal é conseguido, em ambiente Windows com as teclas CTRL+Espaço.

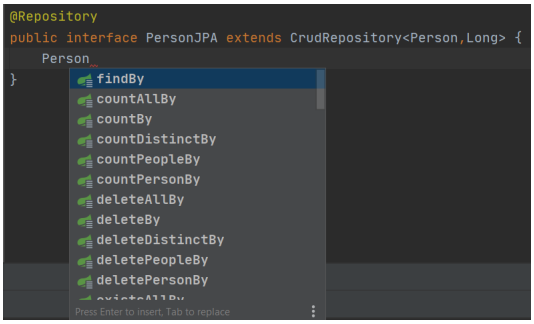


Figura 6.13: Sugestões fornecidas pelo autocompletar do IDE Intellij IDEA

6.4. Exercícios

1. Mapear as classes do modelo definidos pelo caso de estudo
2. Criar repositórios para os modelos
3. Utilizar a classe Bootstrap para povoar a base de dados

Tópico 7: Testes Automatizados

Docente: Alessandro Moreira

7.1. Tipos de testes automatizados

Testes são a forma de verificar a correção do código desenvolvido. Geralmente é feito de maneira intuitiva, fornecendo dados de entrada e verificando se o resultado é o esperado. Fazer testes manuais exige muito tempo e é passível de erros, pelo que é **exigível** que seja feito por via da automação. A framework Spring suporta uma biblioteca de automação de testes chamada JUnit que implementa muitas técnicas de testes automatizados.
Há diversos tipos de testes automatizados, cuja distinção pode ser dada em termos do âmbito do que será testado. A figura 7.1 reflete uma das classificações de testes existentes:

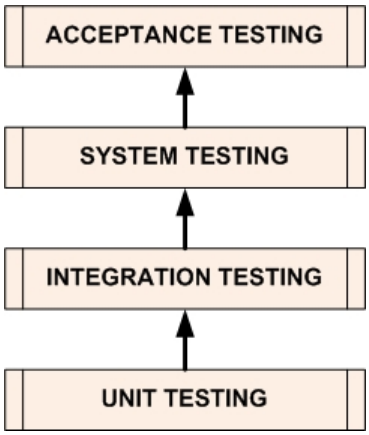


Figura 7.1: Classificação de testes automatizados

7.2. JUnit e testes unitários

Os testes unitários são os de menor âmbito possível, em que se pretende testar uma classe apenas, sem interação com outros objetos. Os testes de integração são aqueles em que se testa a interação entre duas unidades/classes. O teste de sistema pretende verificar se o sistema funciona corretamente (eficácia). Por fim, o teste de aceitação pretende verificar se o sistema faz aquilo que o cliente espera (efetividade).
Os testes devem ser classes existentes no diretório test, já mencionado. Estas classes devem ter pelo menos um método *public void* anotado com @Test. Um exemplo trivial de teste é o seguinte:

```
@Test
public void trivialTest1(){
    assertEquals(1,1);
    int someValue=10;
    assertNotEquals(1,someValue);
}
```

A biblioteca JUnit fornece um conjunto de métodos estáticos da família assert*. No exemplo mostrado, o objetivo é assegurar (assert) que 1 é igual a 1, e que 1 não é igual a 10. Há diversos outros tipos de assert, como o Null, NotNull, True, False. Como será visto, uma aplicação WebMVC possui diversas camadas (p. ex., domínio de negócio, repositórios, controladores, serviços) e convém que todas elas sejam testadas, de modo a garantir que o código funciona corretamente e sem falhas. Funcionar corretamente significa que dentro do domínio de dados de entradas válidos, o resultado é o esperado. Sem falhas, significa que se dados de entrada inválidos forem passados, o sistema não quebra (p. ex. ignora ou lança exceções).

As classes de teste devem estar na pasta test/java, de preferência na mesma package da classe a ser testada. O IDE Intellij IDEA auxilia a criação das classes de teste, ao selecionar o nome da classe a ser testada e utilizar as teclas Alt+Enter e escolher Create Test. O código 7.1 mostra um exemplo de como implementar o teste para o método orderPrice da classe Order. O código cria 3 produtos, cada qual com um preço, adiciona-se todos a um objeto Order para confirmar que o valor do orderPrice é 45, a soma de todos os preços.

```
class OrderTest {

    @Test
    void orderPrice() {
        Product p1=new Product();
        p1.setPrice(10);

        Product p2=new Product();
        p2.setPrice(20);

        Product p3=new Product();
        p3.setPrice(15);

        Order order=new Order();
        order.addProduct(p1);
        order.addProduct(p2);
        order.addProduct(p3);

        assertEquals(45,order.orderPrice());
    }
}
```

Código 7.1: Teste Unitário da classe Order

Também é possível fazer testes automatizados dos repositórios. Em 7.2 vê-se um modo de implementar teste ao repositório PersonJPA já criado.

```
@DataJpaTest
class PersonJPATest {

    @Autowired
    private PersonJPA personJPA;

    @Test
    void findByUsername() {
        assertEquals(0,personJPA.count());

        Person person=new Person();
        person.setFirstName("fn1");
        person.setLastName("ln1");
        person.setUsername("user1");

        Person person2=new Person();
        person2.setFirstName("fn2");
        person2.setLastName("ln2");
        person2.setUsername("user2");

        personJPA.save(person);

        assertEquals(1,personJPA.count());

        personJPA.save(person2);

        assertEquals(2,personJPA.count());

        assertNotNull(personJPA.findByUsername("user1"));
```

```
        assertNotNull(personJPA.findByUsername("user2"));
        assertNull(personJPA.findByUsername("user"));

        personJPA.delete(person);
        assertEquals(1,personJPA.count());
        personJPA.delete(person);
        assertEquals(1,personJPA.count());
    }
}
```

Código 7.2: Teste Unitário do repositório PersonJPA

A classe de teste é anotada com @DataJpaTest o que permite utilizar criar um contexto Spring apenas com as classes anotadas com @Repository. Tal é importante para injetar PersonJPA, o que é feito com @Autowired no atributo. E seguida, é verificado que a tabela Pessoa está vazia, cria-se duas pessoas, vê-se que a quantidade de entradas na tabela aumenta a cada inserção, testa-se, ainda, a pesquisa por username e em seguida remove-se as pessoas e verifica-se que a quantidade diminui.
Os teste unitários com utilização de mocks e os testes de integração e de sistema serão vistos em tópicos a seguir.

7.3. Exercícios

1. Criar testes unitários para os modelos do caso de estudo, especialmente os métodos da classe Explicador/Tutor
2. Criar testes dos repositórios já implementados

Tópico 8: HTTP e Controllers

Docente: Alessandro Moreira

8.1. HTTP

Uma aplicação web utiliza o protocolo HTTP para interagir com o seu utilizador. É um protocolo cliente-servidor e suporta a existÊncia da World Wide Web e de outros serviços da internet. Quando se acede a uma página web, por exemplo, facebook.com, o navegador realiza um pedido HTTP contra o servidor que hospeda a página do facebook, que é encontrado pelo domínio facebook.com, e recebe como resposta os recursos para que o navegador possa exibir a página, dentre estes recursos, a estrutura em HTML, o layout em CSS e o comportamento em Javascript.

8.1.1. URL

O URL é um caminho para localização de um recurso por meios informáticos (Uniform Resource Locator) e tem de respeitar o formato abaixo:

<protocol>://<host>:<port>/<path>?<query>

- protocol: protocolo utilizado para aceder ao recurso (exemplos, file, tcp, http, ftp, ssh)
- host: um recurso encontra-se num determinado local, determinado pelo valor host que pode ser um nome (domínio) ou um endereço IP
- port: é um valor opcional que determina a porta do socket a ser utilizado no host para aceder ao recurso
- path: um recurso dentro de um determinado host pode estar hospedado em um determinado caminho, que deve ser separado por "/", tal qual o sistema de ficheiros em ambientes Unix
- query: representa uma consulta que pode ser utilizada para implementar filtros de pesquisa em recursos. Em geral, utiliza-se pares chave-valor, (key=value) separados por "&"

Um exemplo completo de um URL com todos estes parâmetros é:

http://localhost:8080/path/to/resource?key1=value1&key2=value2

No caso do acesso ao facebook.com, e em pedidos de páginas Web em geral, a porta é omitida, visto que os navegadores, por defeito, completam este parâmetro com o valor 80 que é a porta TCP utilizada por defeito por servidores web[7].

[7]Lista de portas TCP e UDP utilizadas por defeitohttps://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers#Well-known_ports

8.1.2. Verbos HTTP e REST

O protocolo HTTP define alguns métodos ou verbos que dão significado aos pedidos realizados. Os métodos estão associados com operações que o cliente pretende que o servidor realize, mas é importante ter em vista que a sintaxe do protocolo não impõe ao servidor a observância a estes significados. O desenvolvedor é quem deve garantir que as operações são realizadas com o verbo adequado.
Os métodos mais importantes são:

- GET: serve para pedir por um determinado recurso existente no servidor (leitura apenas)
- POST: serve para realizar algum tipo de processamento relativo a um dado recurso no lado do servidor (escrita)
- PUT: serve para substituir toda a representação de um recurso no servidor pela informação passada pelo cliente
- PATCH: serve para realizar modificação parcial no recurso
- DELETE: server para remover o recurso do servidor

Uma lista mais completa com uma breve descrição e apontadores para os RFC's que definem o protocolo pode ser encontrada em `https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods`.
REST (REpresentational State Transfer) é um estilo arquitetural que impõe restrições na utilização do protocolo HTTP. REST é resultado de uma tese doutoral feita por Roy Fielding[8], que trabalhou na padronização do protocolo HTTP. Um resumo da implicação do REST para o desenvolvedor pode ser encontrado em `https://martinfowler.com/articles/richardsonMaturityModel.html`. Aqui será usada uma simplificação do REST que consiste no mapeamento dos verbos do HTTP com as operações CRUD. Para os que desejarem avançar para uma implementação mais completa podem visitar a ligação `https://spring.io/guides/gs/rest-hateoas/`.
É possível verificar as semelhanças existentes entre as operações no CRUD e os verbos HTTP, de modo manter a semântica do HTTP e evitar que os clientes de um servidor sofram efeitos adversos ao utilizar um serviço disponível (p. ex. modificar algum recurso com um pedido GET, que deve ser apenas leitura). O mapeamento segue abaixo:

CRUD	**HTTP**
Create	POST
Read	GET
Update	PUT e PATCH
Delete	DELETE

Assim, um pedido HTTP com o verbo POST deverá implicar numa operação de criação de um determinado recurso na base de dados, GET uma leitura deste recurso, PUT e PATCH uma atualização e DELETE, a remoção deste recurso na base de dados. A distinção entre um PUT e PATCH reflete o significado destes verbos no HTTP. Quando se utiliza o PUT, o efeito deverá ser de substituição do recurso já existente na base de dados pelo recurso enviado pelo pedido HTTP. Quando se utiliza o PATCH, deve ser uma modificação parcial. No PUT, passa-se a informação completa do recurso e no PATCH passa-se apenas uma parte da informação do recurso.
Um pedido POST, PUT e PATCH, deverá possuir dados no corpo do pedido HTTP. É possível passar informações no corpo de um pedido GET, porque a sintaxe do HTTP não impede, mas viola a semântica do protocolo e em alguns servidores podem gerar erros, visto que não é uma prática a ser seguida.
Os pedidos devem conter o método HTTP e o URL do recurso. Para simplificar pode-se omitir o URL até o

[8] `https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`

caminho, p. ex., para se obter todas as pessoas do servidor local, deve-se fazer GET /pessoa (o URL completo deveria conter http://localhost:8080/pessoa). Para criar uma nova pessoa no servidor, POST /pessoa e passar os dados da pessoa no corpo do pedido.
A informação no corpo dos pedidos pode ser enviada em vários formatos, dentre os quais JSON. A sintaxe do JSON representa basicamente objetos e arrays. Objetos são representados dentro de chavetas ("") e objetos, dentro de parênteses retos ("[]"). Os atributos dos objetos são representados pelo nome e valor separado por dois pontos (":"). Um exemplo de um objeto JSON está no código 8.1.

```
{
    "firstName":"João",
    "lastName":"Silva",
    "username":"joaosilva"
}
```

Código 8.1: Exemplo objeto JSON

8.1.3. Cabeçalho e Corpo do HTTP

O pedido e a resposta possuem 2 componentes principais, o cabeçalho e o corpo. O corpo no HTTP é o campo que transfere informação entre cliente e servidor. O cliente ao fazer um pedido GET espera encontrar a informação requisitada no corpo da resposta. O servidor ao receber um pedido POST espera encontrar alguma informação no corpo do pedido e pode enviar alguma informação no corpo da resposta.
O cabeçalho no pedido e na resposta é o campo do protocolo HTTP em que cliente e servidor partilham informação sobre os pedidos e respostas. Eles trocam informações úteis acerca da interação que estão a realizar neste instante, como o tamanho do corpo, se está ou não codificado ou comprimido, se é possível fazer cache e qual o tipo do conteúdo.
Há várias formas de verificar estes campos nos pedidos HTTP, inclusive com a ferramenta Wireshark, mas os navegadores já trazem algumas funcionalidades para este efeito. As figuras 8.1 e 8.2 mostram a inspeção feita pelo navegador Chrome do pedido e resposta quando se acede à página da consola H2 (CTRL+SHIFT+I -> aba Rede/Network).

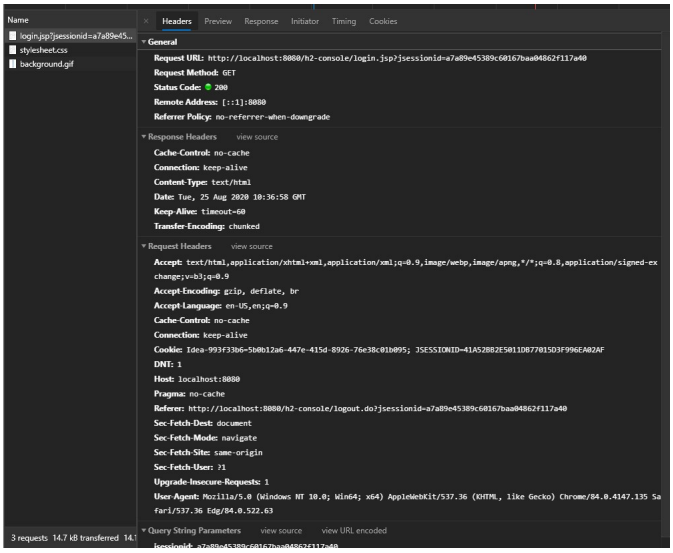


Figura 8.1: Resultado do acesso à consola H2 na ferramenta de desenvolvedor do navegador

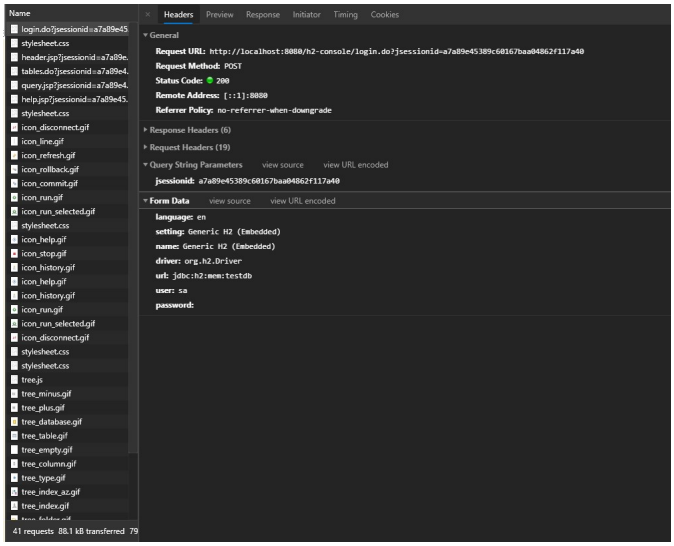


Figura 8.2: Resultado do login na consola H2 na ferramenta de desenvolvedor do navegador

Em 8.1 vê-se o resultado de apontar o navegador para o URL localhost:8080/h2-console, o que se verifica em Request URL e o método utilizado para o pedido é GET, visto que o navegador deseja receber a informação para exibir a página da consola. No campo Request Headers, ou seja, o cabeçalho do pedido feito pelo navegador está a informação Accept, em que o navegador envia para o servidor quais os tipos de conteúdo que ele aceita, entre eles, text/html. No campo Response Header, há a informação Content-Type, em que o servidor informa ao navegador que o tipo do conteúdo que será enviado é text/html. Na aba Response/Resposta é possível verificar o html enviado, que é o mesmo utilizado para compor a estrutura da página inicial da consola H2.
Em 8.1 vê-se o resultado de fazer login na consola H2. Como um login implica em criar uma sessão para um utilizador após a autenticação do mesmo, usa-se o método POST e envia-se a informação para que o servidor verifique se o utilizador é quem diz ser, por meio do nome de utilizador e palavra-passe que são enviada no corpo do pedido. Como o pedido foi bem sucedido, o servidor envia informação para criar a apresentação da consola da base de dados H2.
Em ambos os casos vê-se a informação acerca do código de estado da resposta com o valor 200, que é uma resposta genérica de pedido bem sucedido. A tabela 8.1 mostra as famílias dos códigos de estado e o respectivo significado.

Família Código de Estado	**Significado**
1xx	Informação
2xx	Bem sucedido
3xx	Redirecionamento
4xx	Erro do cliente
5xx	Erro do servidor

Tabela 8.1: Famílias de códigos de estado

Mais detalhes sobre os valores e significados dos códigos de estado podem ser vistos em `https://en.wikipedia.org/wiki/List_of_HTTP_status_codes`.

8.2. Controllers

A framework Spring possui a anotação @Controller que é um estereótipo próprio para os controladores de uma aplicação WebMVC. Esta anotação deve ser usada na classe que implementa o controlador. Cada controlador

deve lidar com um determinado **recurso** disponibilizado pela aplicação. O código 8.2 mostra um exemplo muito simples de implementar um controlador dentro da classe principal da aplicação.

```
@SpringBootApplication
public class AulasApplication {
    public static void main(String[] args) {
        SpringApplication.run(AulasApplication.class, args);
    }
}

@Controller
class HelloController{
    @RequestMapping(method = RequestMethod.GET,produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<String> hello(){
        return ResponseEntity.ok("hello");
    }
}
```

Código 8.2: Exemplo de utilização da anotação @Controller

O código apresenta outra anotação, @RequestMapping, que serve para definir os parametros do pedido que será tratado pelo método anotado. No exemplo, a anotação foi parametrizada com os argumentos method e produces. O argumento method determina qual o verbo HTTP deve ser utilizado para invocar o método hello(), no caso GET. O argumento produces determina qual o formato Mime que será utilizado na resposta para o cliente, no caso, Application/JSON. Para verificar se o controlador funciona, pode ser utilizado o navegador e fazê-lo apontar para o endereço localhost:8080. Localhost é um domínio universal que significa o próprio dispositivo, ou o endereço IP 127.0.0.1 e 8080 é a porta TCP em que a aplicação fica à escuta, que por defeito tem este valor, mas que pode ser modificado no ficheiro application.properties com a chave *server.port*. O resultado na página do navegador deverá ser a String hello, que foi devolvida pelo controlador com o método estático ok() da classe ResponseEntity.
ResponseEntity é uma classe que representa a resposta devolvida pelo controlador, e o método ok() representa o código de estado 200 do protocolo HTTP, a resposta genérica que o pedido é válido. A figura 8.3 exibe os parametros do pedido e resposta relativos ao controlador e método implementado pelo código 8.2, em especial, o url do pedido (Request URL), o método do pedido (Request method), o código de estado da resposta (Status Code) e o formato da resposta (Response Headers -> Content-Type). Podem experimentar outros valores do enum MediaType para constatar que o valor do Content-Type é modificado.

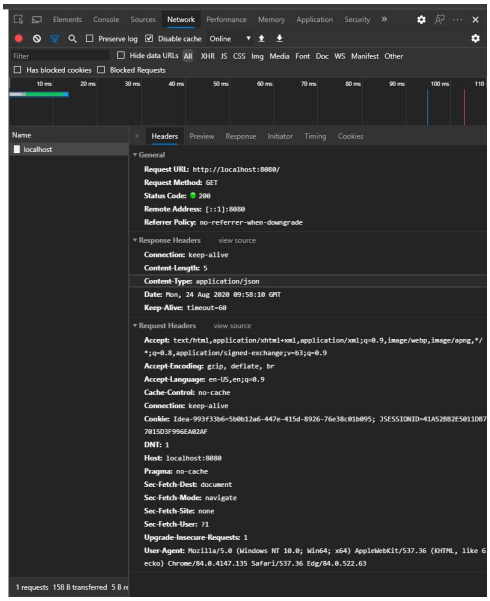


Figura 8.3: Resultado do pedido GET / na ferramenta de desenvolvedor do navegador

Para melhor organização do código, é conveniente que os controladores estejam juntos numa mesma package,

pelo que se deve criá-la na raiz do projeto e deve conter todas as classes que implementam os controladores. Para já, os controladores devem depender dos repositórios necessários para interagir com a base de dados. O código 8.3 apresenta a implementação do controlador para o recurso pessoas.

```
@RequestMapping("/person")
@Controller
public class PersonController{
    private final PersonJPA personJPA;

    @Autowired
    public PersonController(PersonJPA personJPA) {
        this.personJPA = personJPA;
    }
    @RequestMapping
    public ResponseEntity<Iterable<PersonDTO>> getPerson(){
        return ResponseEntity.ok(personJPA.findAll());
    }

}
```

Código 8.3: Implementação PersonController

A anotação @RequestMapping usada na classe determina o caminho base para os seus métodos. A classe irá injetar PersonJPA pelo construtor e define um endpoint para devolver todas as pessoas existentes na base de dados para o cliente que realize o pedido GET /person.
O código 8.4 mostra como usar a classe Bootstrap para escrever uma pessoa com todos os dados na base de dados. Para já, a anotação @Component estará comentada para não interferir na demonstração que se pretende.

```
//@Component
@Transactional
public class Bootstrap implements ApplicationListener<ContextRefreshedEvent> {

    private final PersonJPA personJPA;

    public Bootstrap(PersonJPA personJPA) {
        this.personJPA = personJPA;
    }

    @Override
    public void onApplicationEvent(ContextRefreshedEvent contextRefreshedEvent) {
        Person person=new Person();
        person.setFirstName("fn1");
        person.setLastName("ln1");
        person.setUsername("username");

        Address address=new Address();
        address.setStreet("Rua");
        address.setCity("Cidade");
        address.setCountry("País");

        person.setAddress(address);

        Order order1=new Order();
        order1.setOrderTime(LocalDateTime.now());

        Order order2=new Order();
        order2.setOrderTime(LocalDateTime.now().plusDays(1));

        Product product1=new Product();
        product1.setName("prod1");
        product1.setPrice(10);
```

```
        Product product2=new Product();
        product2.setName("prod2");
        product2.setPrice(20);

        Product product3=new Product();
        product3.setName("prod3");
        product3.setPrice(15);

        order1.addProduct(product1);

        order2.addProduct(product2);
        order2.addProduct(product3);

        person.addOrder(order1);
        person.addOrder(order2);

        personJPA.save(person);
    }
}
```

Código 8.4: Classe Bootstrap para povoar a base de dados com uma pessoa com todos os dados

O resultado do pedido GET /person feito no navegador deverá ser uma janela branca que exibe apenas [], o que indica um array vazio, o que indica que o pedido funcionou e que a base de dados está vazia.
Ao se descomentar a anotação @Component e se tentar realizar o mesmo pedido, a janela irá exibir uma quantidade massiva de texto e a consola do IDE uma mensagem de erro, que é resultado dos relacionamentos bidirecionais no modelo, especialmente entre a classe Person e a classe Order.

8.2.1. (De)Serialização

Serialização é o processo que transforma um objeto no servidor em informação em um determinado formato, no caso JSON, para o cliente/vista. De-serialização é o processo inverso, em que o controlador recebe informação em um determinado formato e converte para o objeto no servidor.
O erro mostrado no tópico anterior decorre da serialização de um objeto que possui um relacionamento bidirecional. Nestes casos, o controlador entra em um ciclo infinito, em que ao serializar um dos lados do relacionamento, serializa a referência ao outro objeto, que possui uma referência para o objeto inicial.
A biblioteca utilizada para realizar a serialização e de-serialização nos controladores Spring, por defeito é a Jackson (`https://github.com/FasterXML/jackson`) e ela traz algumas formas de resolver o problema da referência circular nestas tarefas.
O problema de referência circular também acontece se a anotação @ToString for utilizada em todos as entidades e se queira imprimir o objeto Person na consola. Uma forma de resolver este problema é retirar um dos lados do relacionamento do método toString, com a anotação @ToString.Exclude. Via de regra, é recomendado que esta anotação seja colocada no lado muitos para um. Em relacionamentos muitos para muitos, é preciso analisar qual a entidade mais importante e que deve continuar no toString.
Uma estratégia semelhante pode ser adotada com a anotação @JsonIgnore. Esta anotação faz com que a biblioteca Jackson não utilize os atributos anotados nem para a serialização nem para a de-serialização. Se esta anotação for utilizada nas classes de modo a quebrar a referência circular, esta deixará de existir. O código 8.5 reflete o resultado ao utilizar @JsonIgnore nos atributos person na classe Order e orders na classe Product.

```
[
  {
    "id": 1,
    "firstName": "fn1",
    "lastName": "ln1",
    "username": "username",
    "address": {
      "id": 1,
```

```
      "street": "Rua",
      "city": "Cidade",
      "country": "País"
    },
    "orders": [
      {
        "id": 1,
        "orderTime": "2020-08-26T14:12:08.023696",
        "products": [
          {
            "id": 1,
            "name": "prod1",
            "price":10.0
          }
        ]
      },
      {
        "id": 2,
        "orderTime": "2020-08-27T14:12:08.023696",
        "products": [
          {
            "id": 2,
            "name": "prod2",
            "price":20.0
          },
          {
            "id": 3,
            "name": "prod3",
            "price":15.0
          }
        ]
      }
    ]
  }
]
```


Código 8.5: Resultado de GET /person com @JsonIgnore

É preciso ter em consideração que a anotação @JsonIgnore afeta a serialiazação e a de-serialização, assim, o atributo com esta anotação não será utilizado em nenhum dos casos. No caso da de-serialização, o atributo marcado com @JsonIgnore terá o valor por defeito, no caso de objetos, null.
Uma alternativa ao @JsonIgnore e que não traz o efeito secundário da de-serialização é utilizar a anotação @JsonIdentityInfo. Esta anotação deve ser utilizada nas classes do modelo. O código 8.6 mostra como a anotação deverá ser utilizada em todas as classes do model.

```
@JsonIdentityInfo(
  generator = ObjectIdGenerators.PropertyGenerator.class,
  property = "id")
```

Código 8.6: Anotação @JsonIdentityInfo a ser utilizada em todas as classe

Em 8.7 vê-se o resultado do GET /person com a anotação @JsonIdentityInfo, que exibe todos os atributos, mas em alguns deles, p. ex. nos atributos person da classe Order e orders da classe Product, em vez de todo o objeto, inclui apenas o id do objeto. A de-serialização neste caso não será ignorada, o que pode ser útil em algumas circunstâncias.


```
[
  {
    "id": 1,
    "firstName": "fn1",
    "lastName": "ln1",
    "username": "username",
    "address": {
```

```
      "id": 1,
      "street": "Rua",
      "city": "Cidade",
      "country": "País"
    },
    "orders": [
      {
        "id": 2,
        "orderTime": "2020-08-26T14:58:40.660363",
        "person": 1,
        "products": [
          {
            "id": 1,
            "name": "prod1",
            "orders": [
              2
            ],
            "price":10.0
          }
        ]
      },
      {
        "id": 1,
        "orderTime": "2020-08-27T14:58:40.660363",
        "person": 1,
        "products": [
          {
            "id": 2,
            "name": "prod2",
            "orders": [
              1
            ],
            "price":20.0
          },
          {
            "id": 3,
            "name": "prod3",
            "orders": [
              1
            ],
            "price":15.0
          }
        ]
      }
    ]
  }
]
```

Código 8.7: Resultado do GET /person com @JsonIdentityInfo

8.2.2. @RequestBody

A anotação @RequestBody é utilizada para se extrair o corpo do pedido HTTP e de-serializá-lo em um objeto no lado do servidor. Será tipicamente utilizado nos métodos de escrita na base de dados, principalmente POST, PUT e PATCH. A anotação deve ser utilizada no argumento do método no controlador e deve antecedor o objeto a ser de-serializado. O código 8.8 mostra como a anotação pode ser utilizada para criar o endpoing POST /person, que deve criar uma nova pessoa na base de dados.

```
    @PostMapping
    public ResponseEntity<Person> createPerson(@RequestBody Person person){
        return ResponseEntity.ok(this.personJPA.save(person));
```

```
    }
```

Código 8.8: Implementação do enpoint POST /person

A anotação @PostMapping é equivalente à @RequestMapping(method= RequestMethod.POST). O argumento do método createPerson é da classe Person e está anotada com @RequestBody, o que faz com que o de-serializador Jackson receba os dados contidos no corpo do pedido POST /person e povoe os atributos de Person com os presentes no pedido. **É importante garantir que os atributos na classe e no JSON possuem os mesmos nomes**.
Para realizar pedidos HTTP com métodos distintos do GET, será necessário usar outras ferramentas para além do navegador. Há várias ferramentas que permitem fazer pedidos HHTP, a que se recomenda utilizar é a Postman (`https://www.postman.com/`).
A figura 8.4 mostra a janela da ferramenta que permite realizar os pedidos HTTP.

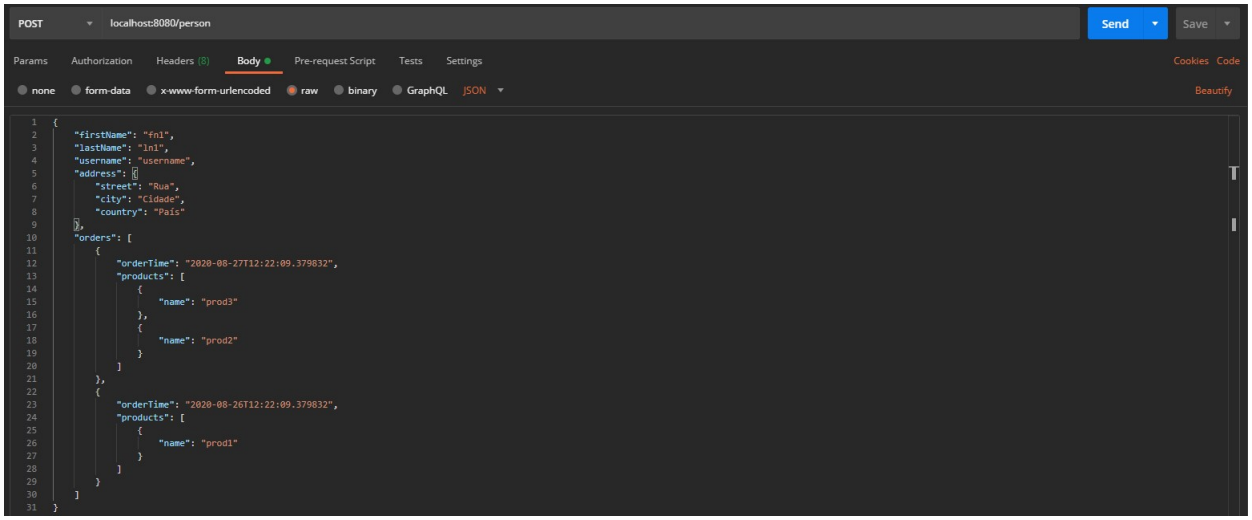


Figura 8.4: Janela do Postman

Ao lado do URL está a lista de métodos HTTP suportados, tendo sido escolhido POST. Há diversas abas que representam os campos do pedido HTTP que podem ser paramterizáveis, interessa para já Body que está em evidência. Para enviar um pedido em formato JSON, deve se escolher a opção raw e ao lado de GraphQL há uma lista de formatos textuais disponíveis, entre eles JSON. No campo de texto abaixo, deve ser incluído o JSON que se deseja enviar para o servidor, com a sintaxe adequada[9].

8.2.3. @PathVariable e @RequestParam

O URL de um pedido HTTP pode conter informação no path. Já foi visto que um pedido GET /person na máquina local é uma simplificação para o URL completo http://localhost:8080/person. O path pode ser utilizado para enviar mais informação, incluindo variáveis. Assim, o pedido GET /person/1 pode ser utilizado para devolver a pessoa que possua o id 1. O path pode ser mais profundo, por exemplo, GET /person/username/joaosilva, pode ser utilizado para devolver a pessoa que possua o username joaosilva. Em ambos os casos a última parte do URL pode conter outros valores, logo uma variável. Assim, os endpoints podem ser representados como GET /person/id e GET /person/username/username, em que os valores entre chavetas são variáveis atribuídas quando o pedido efetivo for realizado (http://localhost:8080/person/1 e http://localhost:8080/person/username/joaosilva).
O controlador deve implementar os métodos que tratem destes pedidos. Em 8.9 vemos como se pode utilizar a anotação @PathVariable para este efeito.

[9]Podem utilizar sites na web para validar o JSON antes do envio

```
    @GetMapping("/{id}")
    public ResponseEntity<Person> getPersonById(@PathVariable("id")Long id){
        Optional<Person> optionalPerson=this.personJPA.findById(id);
        if(optionalPerson.isPresent()){
            return ResponseEntity.ok(optionalPerson.get());
        }
        return ResponseEntity.notFound().build();
    }
    @GetMapping("/username/{username}")
    public ResponseEntity<Person> getPersonByUsername(@PathVariable String username){
        Person person=this.personJPA.findByUsername(username);
        if(person!=null){
            return ResponseEntity.ok(person);
        }
        return ResponseEntity.notFound().build();
    }
```

Código 8.9: Utilização da anotação @PathVariable

O método getPersonById está anotado com @GetMapping e o argumento "/id"que cria a variável de path id. No argumento do método está a anotação @PathVariable que recebe como argumento a mesma variável "id". Assim, caso o pedido seja GET /person/1, o valor do argumento id do método getPersonById será 1. Como o argumento tem o tipo Long, se o pedido for GET /person/abc, o servidor irá devolver um erro do tipo 400, porque o controlador não consegue converter a string "abc"em um Long.
Se o valor passado no path for um Long válido, o controlador usa o repositório para pesquisar uma pessoa que possua aquele valor. O método findById do repositório devolve não o objeto da classe Person, mas um objeto Optional<Person>. O objeto Optional é uma espécie de envelope para qualquer objeto e possui o método isPresent para verificar se há ou não algum objeto, ou seja, null. A função do Optional é evitar NullPointerExceptions no código, visto que impõe ao desenvolvedor que verifique a existência de algum objeto.
Se existir alguma pessoa na base de dados com o id passado pelo path, o repositório irá devolver um Optional com o objeto Person correspondente e o controlador pode devolver para a vista o objeto Person com o método get do Optional. Se não existir nenhuma pessoa com aquele id, o controlador devolve para a vista uma resposta notFound, código 404.
O método getPersonByUsername também usa a anotação @PathVariable, mas mostra que não é preciso passar qualquer argumento para a anotação, desde que o nome do argumento do método seja igual ao variável de path existente no RequestMapping, ou no caso, no GetMapping. É importante ter em conta também que não pode haver mais de um endpoint igual, por isto que este endpoint possui o path /person/username/username, para distinguir do /person/id.
Além das variáveis de path, há as query strings, em que se pode passar 1 ou mais variáveis para o servidor via URL. As query strings devem ser precedidas por "?", a chave e valor devem estar separados por "="e cada par chave-valor, deve ser separado por "&". A função é possibilitar uma forma flexível de se fazer uma consulta por quantidades variáveis de parâmetros. Assim, um endpoint que suporta um pedido como GET /person/query?id=1&username=joaosilva, suporta também GET /person/query?username=joaosilva&id=1, GET /person/query?username=joaosilva ou mesmo GET /person/query.
No controlador a anotação @RequestParam é utilizada para receber os valores passados por query strings. Tal qual @PathVariable ela deve ser utilizada no argumento do método que implementa o endpoint que irá receber o pedido HTTP. Em 8.10 vê-se a implementação do endpoint para os pedidos GET /person/query.

```
@GetMapping("/query")
public ResponseEntity<Person> getPersonQuery(@RequestParam Map<String,String> params){
    if(params.containsKey("username")){
        String username=params.get("username");
        Person person=this.personJPA.findByUsername(username);
        if(person!=null){
            return ResponseEntity.ok(person);
        }
    }
    return ResponseEntity.notFound().build();
```

```
}
```

Código 8.10: Implementação do endpoint GET /person/query

O método getPersonQuery recebe como argumento um objeto do tipo Map que possui chave e valor em String e é precedido da anotação @RequestParam, o que faz com que todos os pares chave-valor da query string do pedido HTTP seja incluído neste argumento. Com este endpoint é possível receber um pedido HTTP com o método GET e path /person/query?username=joaosilva. Se fosse passado outros pares chave-valor na query string, todos eles estariam presentes no objeto Map.
O método getPersonQuery verifica se o objeto Map possui a chave username e, caso possua, passa o valor desta chave para o método findByUsername do repositório e se houver uma pessoa na base de dados, devolve-a para a vista em uma resposta com código de estado 200. Se não houver a chave username ou se não houver a pessoa com aquele username na base de dados, é devolvida uma resposta com código de estado 404. Para tornar possível fazer uma pesquisa com outros atributos, era necessário verificar a existência destes valores no mapa e implementar a forma de pesquisar com estes valores.

8.3. Testes de controladores

Para implementar testes automatizados nos controladores, é necessário criar uma classe na pasta test. O IDE Intellij facilita esta tarefa, como já foi visto, ao selecionar a declaração da classe do controlador a ser testado, e utilizar as teclas Alt+Enter (em ambiente Windows), escolher a opção Create Test e pressionar Enter.
Como o controlador possui uma dependência para realizar a sua tarefa, o seu teste unitário precisará de alguns passos a mais. Será preciso criar um objeto que simule as operações da dependência, de modo que no teste seja possível definir o que esta dependência simulada deve fazer quando uma das suas operações for invocada pelo controlador. A esta dependência simulada dá-se o nome de mock e é preciso configurá-la na classe de teste. O código 8.11 mostra um exemplo de como criar um teste de controlador.

```
import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.print;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;

@ExtendWith(SpringExtension.class)
@WebMvcTest(PersonController.class)
public class PersonControllerTest{
    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private PersonJPA personJPA;

    @Test
    public void test() throws Exception {
        this.mockMvc.perform(get("/person"))
            .andDo(print())
            .andExpect(status().isOk());

        this.mockMvc.perform(get("/fail"))
            .andDo(print())
            .andExpect(status().isNotFound());
    }
}
```

Código 8.11: Teste unitário de Controlador

A anotação @ExtendWith(SpringExtension.class) permite que o JUnit (versão 5) utilize as classes geridas pela framework Spring. A anotação @WebMvcTest(PersonController.class) indica que a classe irá testar apenas o controlado PersonController e que este será injetado no objeto MockMvc, que está anotado com @Autowired.

A anotação @MockBean cria um mock de PersonJPA, a dependência do controlador, para que seja injetado neste. É preciso injetar todas as dependências do controlador a ser testado, caso contrário será gerado um erro. O método test, anotado com @Test implementa um teste muito simples. O objetivo é testar se o endpoint GET /person retorna um código de estado 200 e se o endpoint GET /fail retorna um código de estado 404, visto que não existe nenhuma implementação para este pedido.
O objeto mockMvc permite simplificar pedidos HTTP com o método perform. Dentro deste método é possível definir o método e o path para o pedido HTTP. Os métodos HTTP são definidos por métodos estáticos da classe MockMvcRequestBuilders, que corresponde ao import da linha 3 do código. O método perform pode ser encadeado com a chamada de outros métodos. No exemplo, a chamado ao método andDo apenas é feita para fazer um rastreio na consola das informações do pedido e da resposta. O método andExpect faz a verificação se o código de estado da resposta é igual ao valor que se espera, 200 para o endpoint GET /person e 404 para GET /fail. O mock neste exemplo não é necessário para nada além de garantir que o controlador é criado de forma correta. Em 8.12 há um exemplo de como utilizar o mock para testes unitários.

```
    @Test
    public void testGetPersonById() throws Exception {
        Person person=new Person();
        person.setFirstName("name");
        person.setLastName("lastname");

        when(personJPA.findById(10L))
            .thenReturn(Optional.of(person));

        this.mockMvc.perform(get("/person/10"))
            .andDo(print())
            .andExpect(status().isOk());

        this.mockMvc.perform(get("/person/1"))
            .andDo(print())
            .andExpect(status().isNotFound());
    }
```

Código 8.12: Teste unitário de Controlador com uso do mock

Os mocks funcionam na lógica do quando-então (when-then). É preciso dizer para o mock que quando um determinado método for invocado então ele deve fazer algo, que pode ser retornar algo ou disparar uma exceção. No código é criado um objeto Person com alguns valores associados a atributos e em seguida diz-se para o mock de PersonJPA que quando o método findById for chamado com o argumento 10L (L, devido ao id ser um Long), ele deve retornar um Optional do objeto Person criado. Em seguida é feito um pedido GET /person/10, imprime-se o resultado do pedido e da resposta e verifica-se se o pedido é um 200, o que deve ser verdade, visto que o controlador irá invocar o mock que lhe foi injetado, irá chamar o método findById do mock com o valor 10L e irá receber um Optional com o objeto Person. O segundo pedido HTTP deverá receber um código de resposta 404, porque o controlador irá receber o id 1, irá chamar o método findById do mock e como não foi dito ao mock o que fazer quando receber o valor 1 neste método, irá retornar o valor por defeito, que no caso é um Optional vazio. O controlador ao receber um Optional vazio, devolve uma reposta com código de estado 404 e o teste é bem sucedido.

8.4. Exercícios

- Criar controladores que sejam permitam ler e criar recursos para os modelos do caso de estudo.
- Criar teste para cada um destes controladores e respectivos métodos