# Exercício 1 - criando um projeto exemplo com spring boot (spring 1)

Obs: Eu deixei meu projeto publicado no meu github: https://github.com/fviel

1. Para criar um projeto spring do zero, utilizar o Spring Initializr, que é um site web que gera um zip do projeto conforme a configuração desejada:

http://start.spring.io

Lembrar que spring boot usa versão 2.0.0

2. baixar o zip. e importar ele na IDE;

3. Após carregado o projeto, rodar o comando maven de compilação do projeto .

3.1. entre no dir do projeto, e procure pelo arquivo mvnw, é neste dir onde deverá ser rodado o comando abaixo:

PS C:\Users\Fernando\Documents\NetBeansProjects\ spring5webapp> ls

Diretório: C:\Users\Fernando\Documents\NetBeansProjects\ spring5webapp

Mode LastWriteTime Length Name

---- ------------- ------ ----

d----- 22/02/2018 20:35 .mvn

d----- 22/02/2018 20:35 src

d----- 22/02/2018 20:58 target

-a---- 22/02/2018 23:30 249 .gitignore

-a---- 22/02/2018 23:30 6468 mvnw

-a---- 22/02/2018 23:30 4994 mvnw.cmd

-a---- 22/02/2018 23:30 2886 pom.xml

PS C:\Users\Fernando\Documents\NetBeansProjects\ spring5webapp> **./mvnw spring-boot:run**

4. Aguardar a compilação

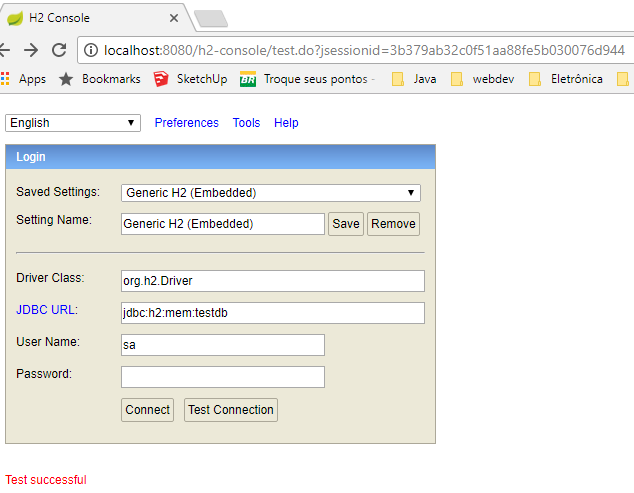
5. Para ativar o console web do bd H2, no arquivo application.properties insira o comando abaixo;

spring.h2.console.enabled = true

6. Recompile o projeto e acesse:

*http://localhost:8080/h2-console*

7. Altere a JDBC URL conforme a tela abaixo para que seja possível o acesso ao database de testes do H2:



8. Pronto, o primeiro projeto spring com bd já está funcional.

# Exercício 2

Depois de feito o primeiro projeto, anotei as entidades JPA, e foram criados seus DAOs, ou repositories.

O Livro do Eric alguma coisa sobre DDD define entre suas ideias de patterns que devemos ter um acesso padrão para os dados, não importando com o que estamos acessando, portando, seguindo a ideia da genericDAO, temos apenas que criar um interface de cada model que extends de uma interface padrão de genericdao já dentro do spring:

\*/

public interface AuthorRepository extends CrudRepository<Author, Long>{

}

E Por que não preciso fazer o AuthorRepositoryImpl??

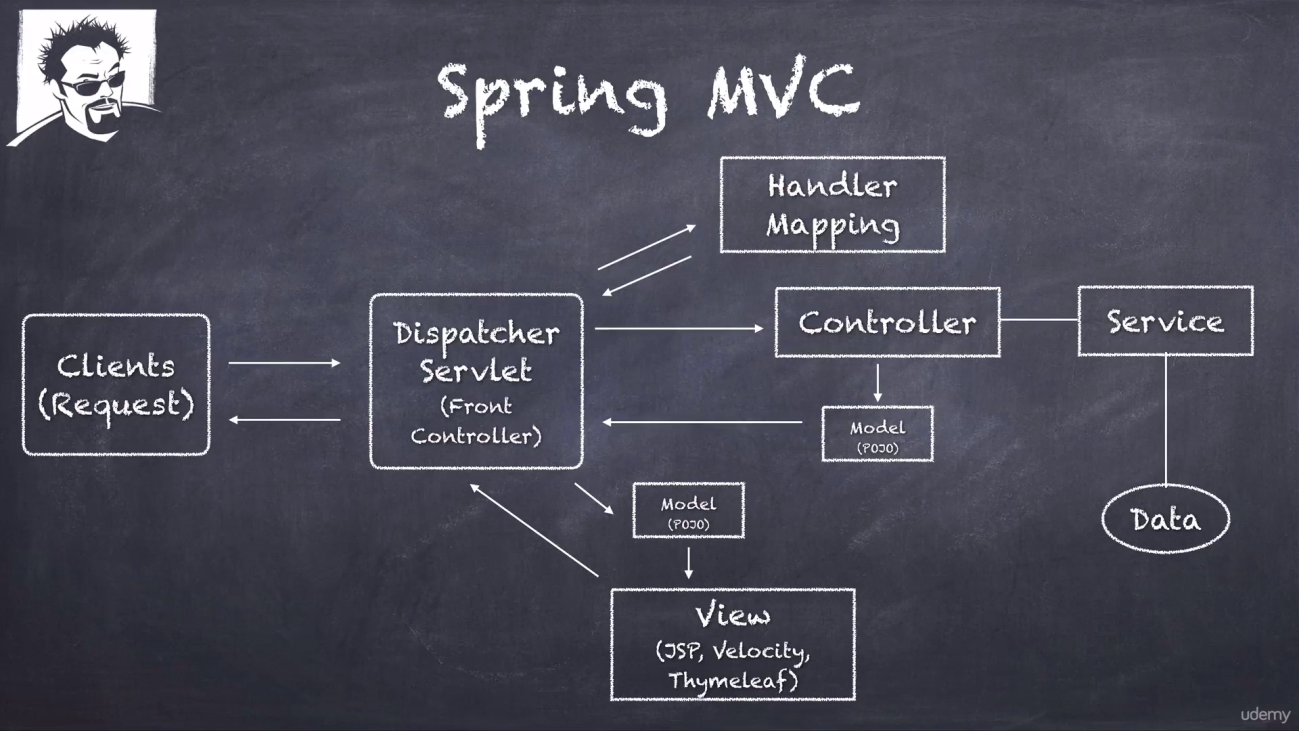
Porque o spring o cria automaticamente para mim! é o @Autowired funcionando...

Para que o sistema criasse automagicamente seus próprios dados para fins de estudos, o autor criou uma classe chamada "DevBootstrap", que é executada assim que a aplicação sobe no servidor, automagicamente.

public class DevBootstrap implements ApplicationListener<ContextRefreshedEvent> {

Observe que ela extends ApplicationListener, pelo que entendi o spring possui eventos, e ao fazer esta extends estou aplicando um evento na entidade. acho que é isso....

# Descrição de como o spring funciona



Melhor explicação que achei sobre @Autowired:

First, and most important - all Spring beans are managed - they "live" inside a container, called "application context".

Second, each application has an entry point to that context. Web applications have a Servlet, [JSF](http://en.wikipedia.org/wiki/JavaServer_Faces)uses a el-resolver, etc. Also, there is a place where the application context is bootstrapped and all beans - autowired. In web applications this can be a startup listener.

Autowiring happens by placing an instance of one bean into the desired field in an instance of another bean. Both classes should be beans, i.e. they should be defined to live in the application context.

What is "living" in the application context? This means that the *context* instantiates the objects, not you. I.e. - you never make new UserServiceImpl() - the container finds each injection point and sets an instance there.

In your controllers, you just have the following:

@Controller // Defines that this class is a spring bean

@RequestMapping("/users")

public class SomeController {

// Tells the application context to inject an instance of UserService here

@Autowired

private UserService userService;

@RequestMapping("/login")

public void login(@RequestParam("username") String username,

@RequestParam("password") String password) {

// The UserServiceImpl is already injected and you can use it

userService.login(username, password);

}

}

A few notes:

* In your applicationContext.xml you should enable the <context:component-scan> so that classes are scanned for the @Controller, @Service, etc. annotations.
* The entry point for a Spring-MVC application is the DispatcherServlet, but it is hidden from you, and hence the direct interaction and bootstrapping of the application context happens behind the scene.
* UserServiceImpl should also be defined as bean - either using <bean id=".." class=".."> or using the @Service annotation. Since it will be the only implementor of UserService, it will be injected.
* Apart from the @Autowired annotation, Spring can use XML-configurable autowiring. In that case all fields that have a name or type that matches with an existing bean automatically get a bean injected. In fact, that was the initial idea of autowiring - to have fields injected with dependencies without any configuration. Other annotations like @Inject, @Resource can also be used.

# Thymeleaf

É uma Java Template Engine;

É uma alternativa ao uso das .jsp's;

É uma template engine natural, isto significa que você pode ver templates no seu browser;

Assim como jsp, basta editar, salvar e atualizar o browser, sem ter que redeploiar....

# SOLID principles for OOP

**Single responsabiblity principle** - just because you can, doesn't mean you should.

Cada classe deve ter uma única responsabilidade;

Nunca deve haver mais de uma razão para uma classe mudar;

Classes geralmente devem ser pequenas em termos de quantidade de código;

Evitar 'god' classes;

Divida classes grandes em menores.

**Open/Close principle** - A class should be open for extension without the need for modification.

Use private variables only when necessary.

Use abstract base classes;

**Liskov substitution principle** - A Square is a Rectangle, but a Rectangle is not a square;

Objects would be replaceable with instances of their subtypes WITHOUT altering the correctness of the program.

**Interface Segregation principle** - Make fine grained interfaces that are specific for client, not a 'god' interface.

Keep componentes focused and minimized dependencies between them;

**Dependency Inversion principle** - Por que você usa um plugue no fio da lâmpada ao invés de soldar o fio direto na tomada?

Abstraction should not depend upon details;

Details should not depend upon abstractions;

# Spring Application Context

É onde o spring faz todo o seu trabalho de rastrear dependências.

# Dependency Injection

Dependency Injection is where a needed dependency is injected by another object;

The class being injected has no responsibility in instantiating the object being injected;

In other words, it's avoid the developer to instantiate objects using 'new', but this definition is not 100% correct, because if you need something small, yes, you can still use the new operator to instantiate this small object;

For example, to get a database connection is a good practice to use an dependency injection, because you could do a lot of wrong things like not closing the connection, but if you injected the connection, the container will help managing it for you.

Generally avoid DI with concrete classes, and prefer Interfaces;

Example: the object that i've injected;

Some ways to use:

1. Property based DI;
2. setter based DI;
3. Constructor based DI. Generally, use DI in the constructors, it's the most correct in the perspective of SOLID and OO;

# Inversion of Control - IoC

Is a technique to allow dependencies to be injected at runtime;

Dependencies aren't predetermined;

Example: Spring Framework Container