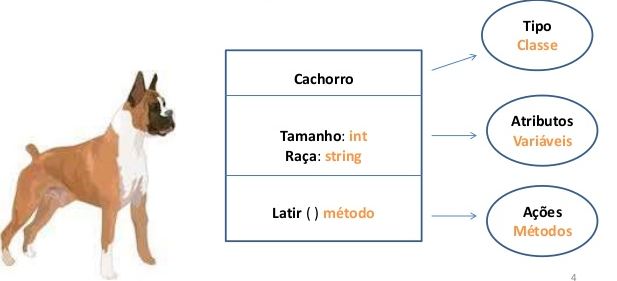
Treinamento Java

**Introdução**

Java é uma Linguagem multiplataforma mantida atualmente pela Oracle.

Está na versão 11, em nosso treinamento vamos utilizar a versão 8. Devido a instabilidade e grande utilização de nossos clientes.

A linguagem Java é orientada a objetos, com o intuito de tornar a linguagem mais próxima do mundo real. Dessa forma, temos as classes que são abstração de um conjunto de objetos com características similares, entidades do mundo real. Esses objetos são compostos de comportamentos que são os métodos e características que são os atributos.



Em relação a versões futuras do Java, existe o OracleJDK (LTS) que será pago a partir do Java 8 e o OpenJDK(releases) que continuará gratuito. Essa decisão é da empresa, e deve ser avaliada em cada caso concreto.

**Instalando o Java 8**

Para prosseguir com o nosso treinamento é necessário ter o o JDK versão 8 instalado.

Verifique a versão no terminal com o comando:

**java -version**

**javac -version**

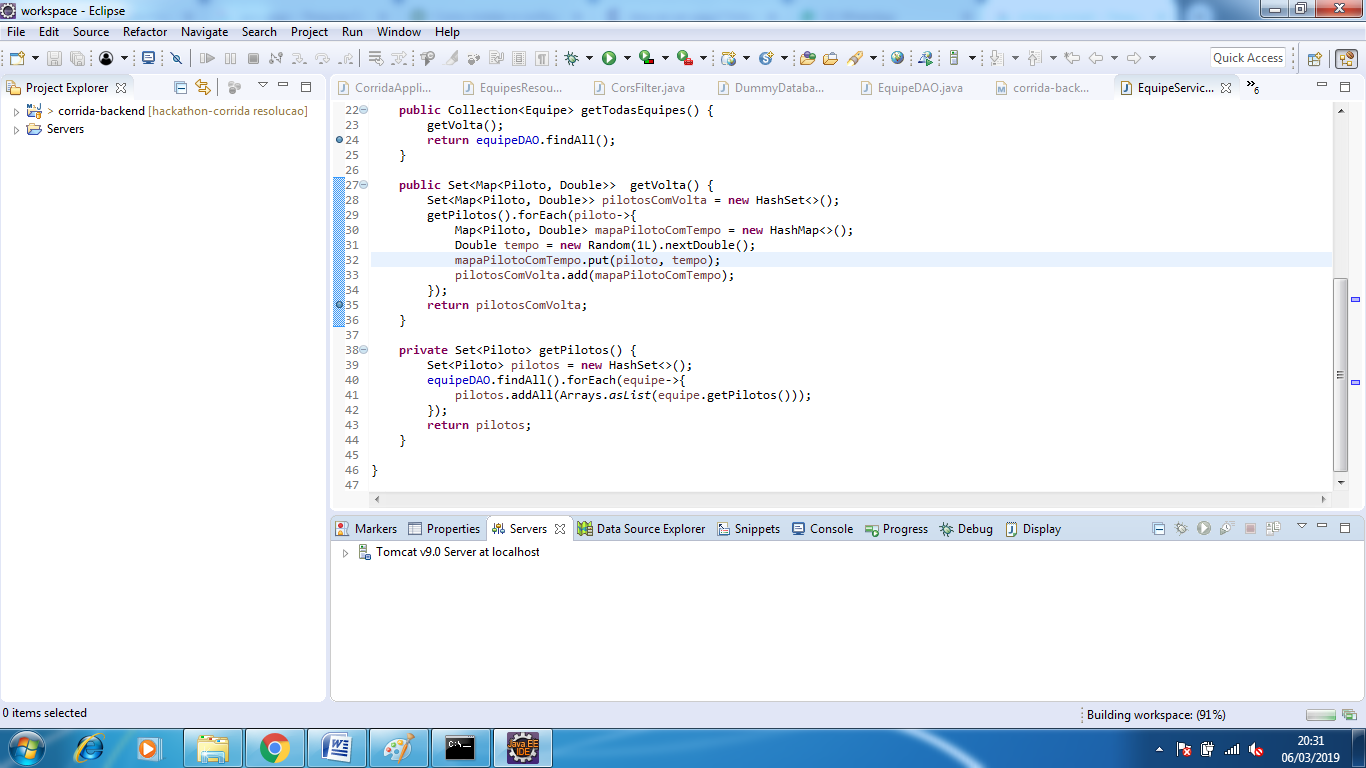


Caso não esteja configurado e instalado corretamente, podemos seguir com esse tutorial de instalação do Java no Windows ou Linux da DevMedia.

https://www.devmedia.com.br/instalacao-e-configuracao-do-pacote-java-jdk/23749

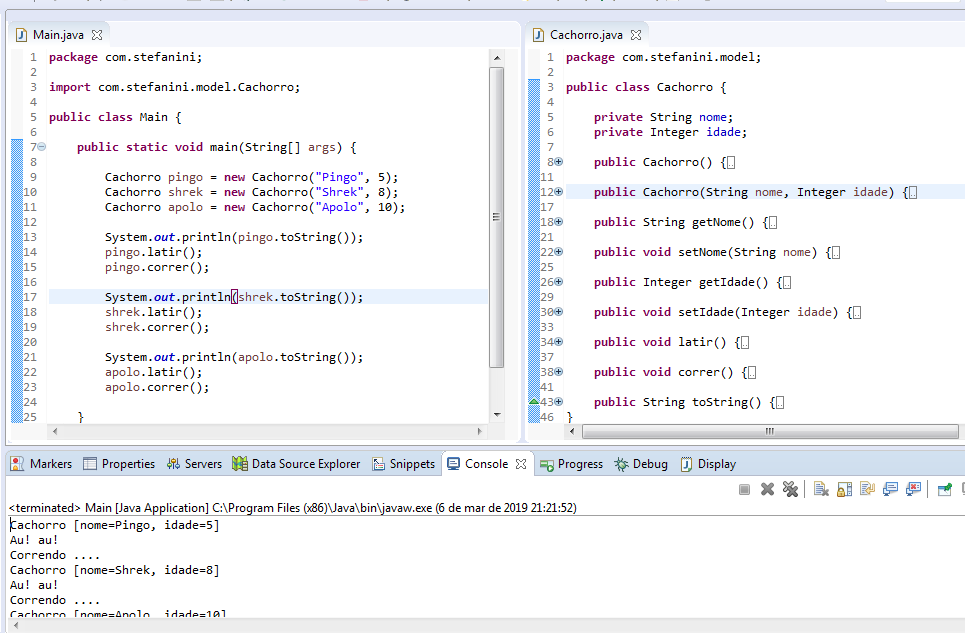
**IDE Desenvolvimento - Eclipse**

Para facilitar o nosso dia a dia utilizamos, geralmente, uma IDE Desenvolvimento. Vamos escolher o Eclipse, Open Source, é amplamente utilizada por diversos desenvolvedores.



**Projeto 1**

Nesse primeiro projeto, vamos criar uma classe Cachorro com os atributos nome e idade, e com os métodos latir() e correr(). Precisamos criar três cachorros: Pingo, Shrek e Apolo.



**Projeto 2**

Nesse segundo projeto, vamos criar uma Servelt que retorne uma lista de cachorros.

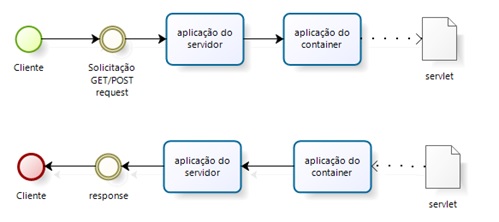
**Servlets**

Servlets são classes Java que de acordo com uma estrutura bem definida, são utilizadas para estender as funcionalidades de um Servlet Container.

**Servlet Container = Web Container**

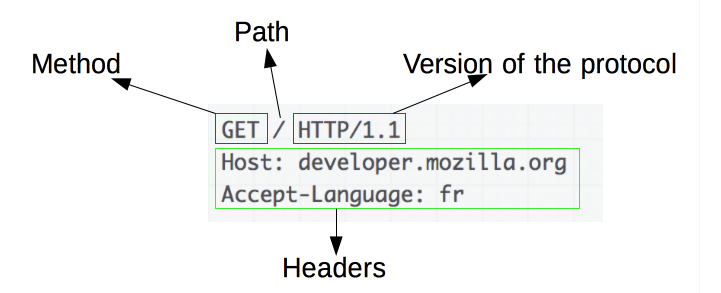
Melhor hora de entendermos o que é um Servelt Container é agora, onde estamos aprendendo o que é uma Servelt. Quando uma aplicação web faz uma requisição para uma Servelt o servidor não entrega a solicitação diretamente para a Servlet, mas sim para o Container que é uma 'super' Servelt.

A principal função do Container é gerenciar o ciclo de vida da Servelt, provendo uma camada que gerencia o multiprocessamento, segurança e controle do ciclo de vida das Servelts.



As Servelts são possuem o método main(), ou seja, não há método de entrada. Então como as Servelt são acionadas? Utilizam-se dos métodos no HTTP, na versão 1.1 temos:

* GET: Solicitação de um recurso, os dados são enviados na URI da solicitação;
* POST: Utilizado para criar um recurso, os dados são enviados no corpo da requisição;
* PUT: Utilizado para armazenar um recurso na URI fornecida;
* DELETE: Utilizado para excluir um recurso;
* TRACE: Utilizada para analisar a requisição;
* OPTIONS: Retorna os métodos HTTP suportados pelo servidor para URL especifica;
* PATCH: Serve para atualizar partes de um recurso, não o recurso todo;
* CONNECT: Converte a requisição para um túnel TCP/IP transparente. Utilizado para SSL;
* HEAD: Retorna somente o cabeçalho de um resposta.

ttpS

No HTTP 1.0 existia apenas os métodos GET, POST, HEAD, com as mesmas funções supra citadas.

**Estrutura da Servelt**

A Servelt é uma classe Java que estende HttpServelt, interessante que a classe HttpServelt estende de GenericServlet, caso nossa aplicação não utilize o protocolo Http.

Com o extend de HttpServelt herdamos os métodos service(), doGet(), doPost(), doPut(), doDelete(), doHead(), doDelete, doOptions(), doTrace, init(), destroy(), getServeltConfig().

Com exceção dos métodos Http, não citamos a função do init() e detroy() e getServletConfig(). Para tal, precisamos entender o ciclo de vida das Servlet.

**Ciclo de Vida da Servlet**

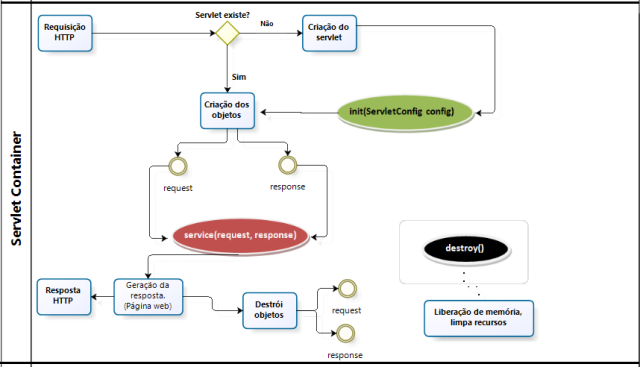
Geralmente, a Servelt é registrada no Container na inicialização, existe uma forma de fazê-la programaticamente, mas não nos interessa, pelo menos por agora, o registro pode ser efetuado diretamente no web.xml ou através da anotação @WebServelt.

Quando o registro é efetuado, a Servelt é apenas instanciada, ou seja, está apenas carregada, na primeira requisição é executado método init(), isso mesmo, somente na primeira requisição que o método init() é chamado. Após, o método init() é executado o método service(), se sobrescrito e posteriormente o método Http correspondente da requisição.

Na próxima chamada, apenas o método service() e o método Http correspondente serão executados. O método init() é executado apenas quando a classe é inicializada.

Método destroy() é executado quando a Servelt é desregistrada do Container, mas não é aconselhável utilizá-lo, pois sua execução não é garantida.

Já o método getServletConfig() retorna um objeto de configuração para cada Servelt durante a inicialização.



Projeto disponível no github:

**Projeto 3**

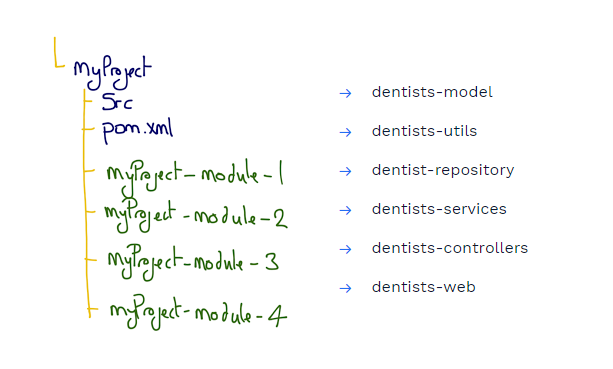
Nesse projeto agora, vamos utilizar o Hibernate para gerenciar nossa base de dados. Com isso, vamos utilizar o MAVEN para o controle das dependências, o Git para controlar a versão, vamos aprender a mapear as tabelas e unas instruções SQL.

Primeiro vamos entender o que é o MAVEN: Ferramenta de integração de projetos, responsável por gerenciar, controlar e garantir a execução de testes, dentre outras. Ok! Só queria saber o que era kkk.

Dessa forma, temos um repositório compartilhado em toda a equipe. Garantindo que todos os projetos utilizem as mesmas dependências, centralizadas em um arquivo central de configuração o famosão pom.xml.

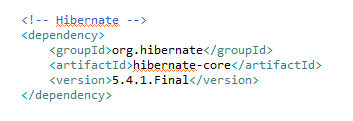
Interessante ter em mente alguns conceitos:

O Maven possibilita que os projetos sejam agrupados em módulos. Onde, temos um projeto principal ou pai, e os demais sub-módulos.

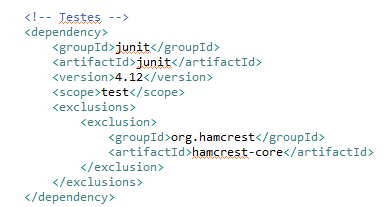


Para isso, utilizamos a tag <parent> para mantermos uma relação de childs. Mas, uma tag importante que foi introduzida recentemente é a dependencyManagement que é responsável por controlar as versões das dependências definidas no projeto.

Temos ainda as tags <dependency> que são cercadas da tag <dependencies>. São responsável pelo import da biblioteca propriamente dita para nossa aplicação.



Em certas ocasiões a biblioteca já traz consigo outras bibliotecas, e não precisamos de todas ou pior ainda, existe um conflito de versões de bibliotecas. É possível efetuar a exclusão de determinadas bibliotecas que são importadas 'por tabela'.



Ops! E essa tag <scope>, ela é muito relevante. Vamos entende-la?

* + **compile**: escopo padrão do maven para o momento em que o código é compilado e vai junto com o artefato;
  + **provided**: adicionado no momento da compilação, mas não vai junto com o artefato. Dessa forma você está dizendo que a dependência virá de maneira transitiva de alguma outra dependência;
  + **runtime**: não inclui no artefato, pois estará disponível em tempo de execução;
  + **test**: inclui apenas no escopo de testes;
  + **system**: não inclui no artefato, pois estará disponível no ambiente;
  + **import**: incluirá TODAS as dependências do ‘depencyManagement‘ que está definido no pom ‘parent’.

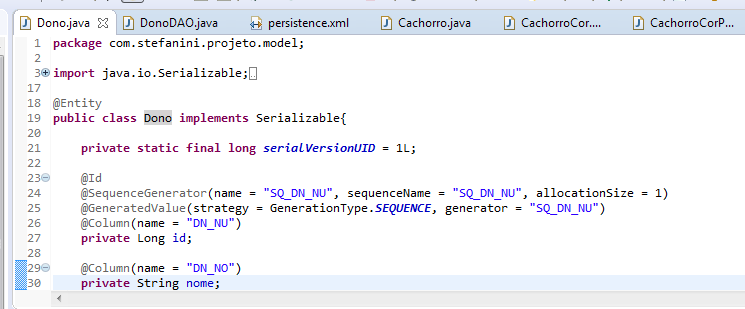
Gostou do Maven? Existem diversos artigos na Internet para solucioná-los diversos desafios, a minha sugestão é a página oficial <https://maven.apache.org/what-is-maven.html>, com esse conhecimento podemos prosseguir com nosso treinamento.

Para 'interagir' com a base de dados utilizamos o mapeamento relacional, onde, nossas entidades são 'convertidas' para objetos. Para isso, vamos utilizar um Framework excepcional o Hibernate<https://docs.jboss.org/hibernate/orm/5.4/quickstart/html\_single/>.

Vamos aprender na pratica a utilizar o Hibernate, primeiro vamos 'mapear' as tabelas que serão alvo do nosso projeto, as Entidades Dono e Cachorro. Onde um Dono tem um nome e pode ter um ou vários cachorros, cada cachorro tem nome, idade e Cores.

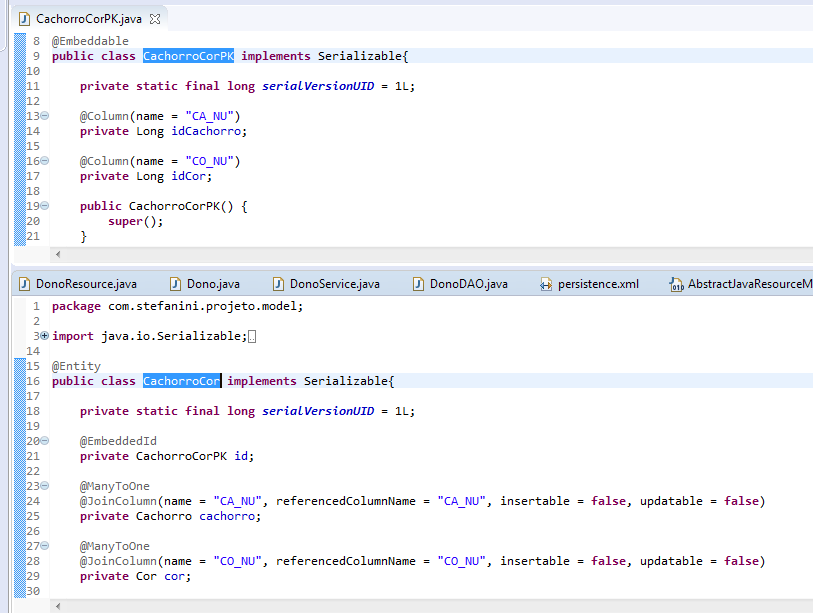
A anotação @Entity indica para o Hibernate que essa classe é uma Entidade. Toda tabela precisa de uma chave primária. Existem duas formas de mapear as chaves primárias. Utilizando uma sequence, onde teremos uma chave auto-incremento ou uma chave composta, onde a chave é composta por 2(dois) ou mais atributos.

Desse modo, vamos ver o mapeamento da tabela Dono para uma tabela com apenas um atributo na chave primária.



Para nosso estudo, escolhemos também mapear uma tabela com dois atributos na chave primaria, caso da tabela CACHORRO\_COR.

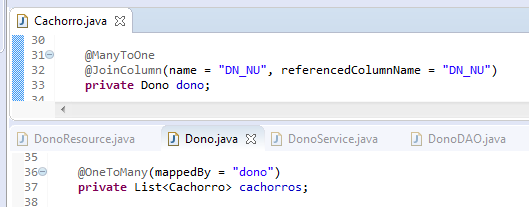
Para esse caso, criamos duas tabelas: CachorroCor e a CachorroCorPK.



As colunas são facilmente mapeadas com a anotação "@Column" e se precisarmos de definir um nome diferente do padrão CamelCase, utilizamos a propriedade "name".

Para finalizarmos essa parte, temo os mapeamentos @OneToMany e @ManyToOne.

@OneToMany é quando precisamos definir um mapeamento um para muitos. Por exemplo, um dono pode ter nenhum, um ou muitos cachorros. E o @ManyToOne é quando precisamos definir muitos para um, ou seja, muitos cachorros podem ter apenas um dono.



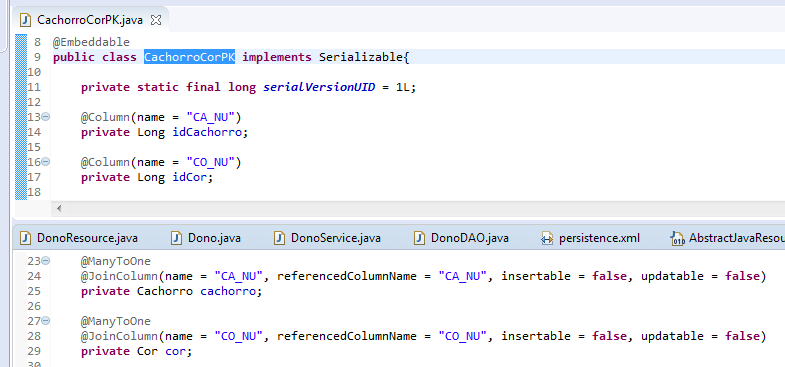
Ponto importante, nos mapeamentos supra citados são as propriedades LAZY e EAGER.

@OneToMany e @ManyToMany são Lazy, logo @ManyToOne e @OneToOne são Eager.

Lazy são preguiçosos, ou seja, só são carregados quando você solicita (get()) ou força o seu carregamento no momento da consulta (fetch), Eager ao contrário, sempre são carregados. Fácil essa. kkk.

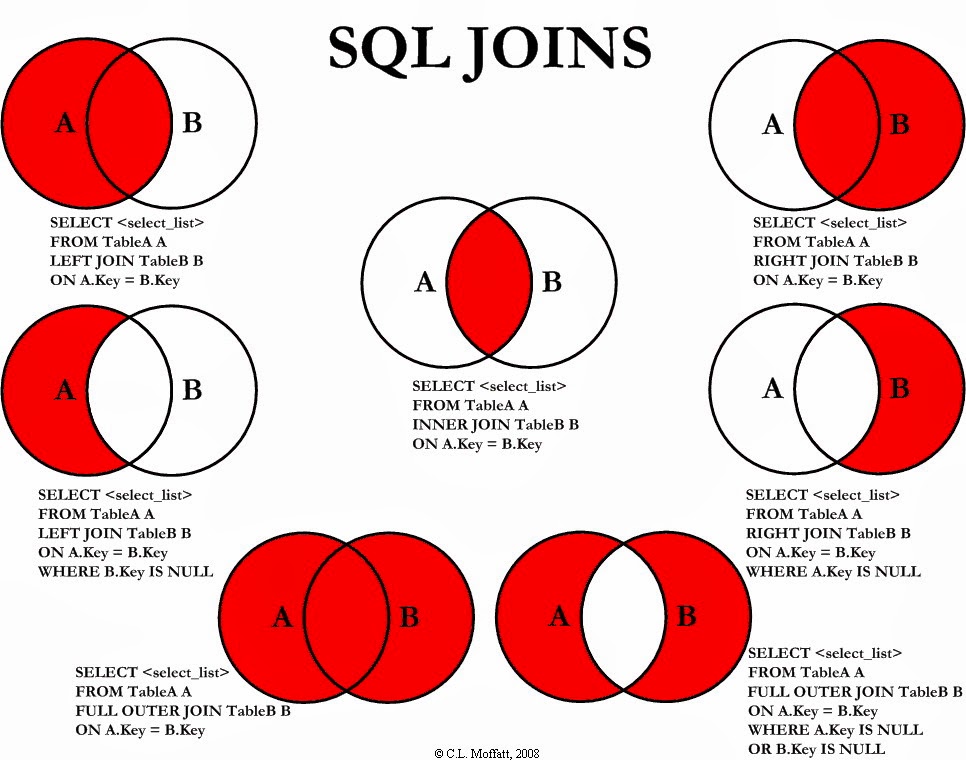
Agora essas anotação são novas @OneToOne e @ManyToMany, devido ao nosso tempo, não será possível abordá-las. Ressalto que a @ManytoMany é menos utilizada que as duas citadas, ocasionalmente encontramo-la. Tente efetuar esse exemplo: <https://www.devmedia.com.br/manytomany-hibernate-variacoes-unidirecional-e-bidirecional/29535>.

Em alguns casos pode ser interessante ter a coluna repetida no mapeamento, o identificador da tabela por exemplo e no outro mapeamento o objeto da entidade. Podemos facilmente utilizar desse artifício, adicionando a propriedade insertable e updatable com o valor false.



O Framework instrui que as classes mapeadas tenham sempre um construtor padrão, sem argumentos, e os métodos equals() e hashcode() implementados. Essa dica vale uma coca gelada. kkk

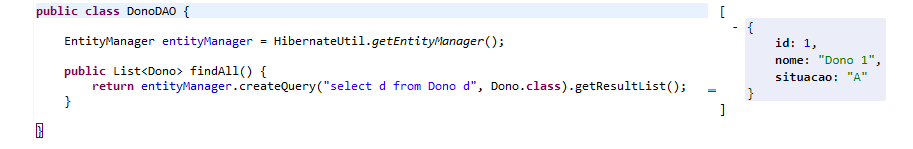
Continuando, como andam os seus conhecimentos em SQL? Consegue criar uma tabela? Efetuar um update/insert/delete? Efetuar um Join? Right ou Left Join?



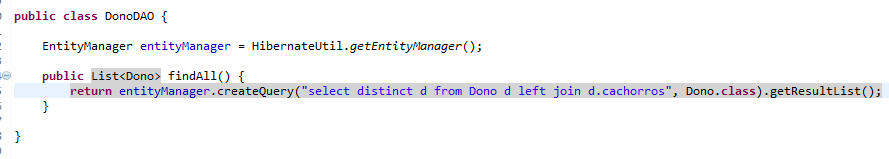
Ok! Você é um fera no SQL.kkk

Utilizando o Hibernate, vamos utilizar uma linguagem conhecida como Hql, muito similar ao SQL para efetuar nossas consultas principalmente. Pois, os comando INSERT/UPDATE/DELETE são normalmente executados com os objetos. Novamente nossa base será a documentação oficial do Framework: <https://docs.jboss.org/hibernate/core/3.5/reference/pt-BR/html/queryhql.html>

Para fazermos uma consulta em todos os donos de cachorros por exemplo:



Cadê os cachorros?

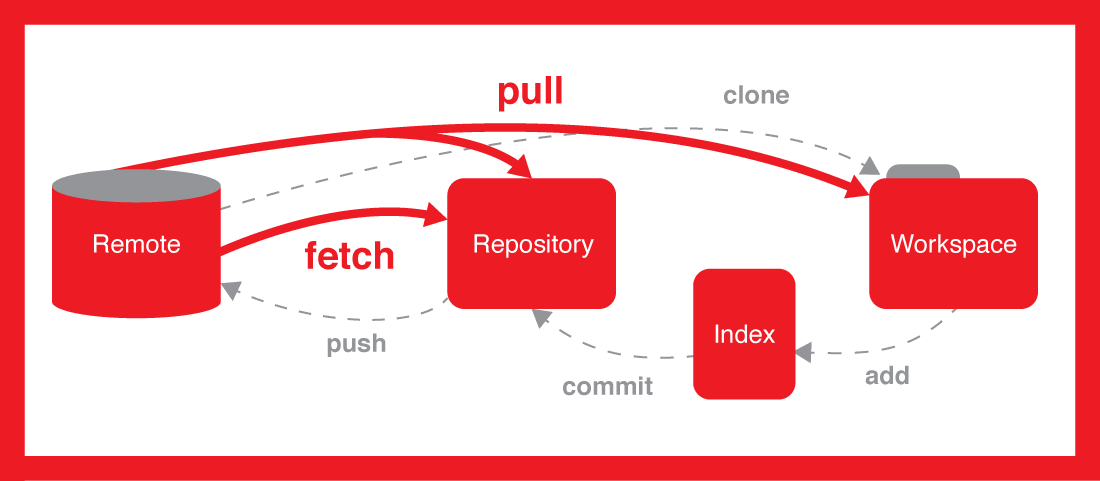


Ops! Que classe é essa DAO? Objeto de Acesso a Dados. Nessa classe, devemos colocar nossas iterações com a base de dados. E nossa lógica de negócio? Devemos colocá-lo na service ou bussiness. E geralmente, existe uma classe de Controller/Rest/Resource onde prover a entrada da requisição HTTP.

Importante é você saber desse padrão e complementar com a leitura do documento de arquitetura, para entender as particulares de cada projeto.



Acredito que agora já podemos 'versionar' o nosso código. Para isso, vamos utilizar o Git. Diferente do SVN o Git é um controle de versão descentralizado, ou seja, onde todos tem o seu repositório local e efetuam iterações com o repositório central.



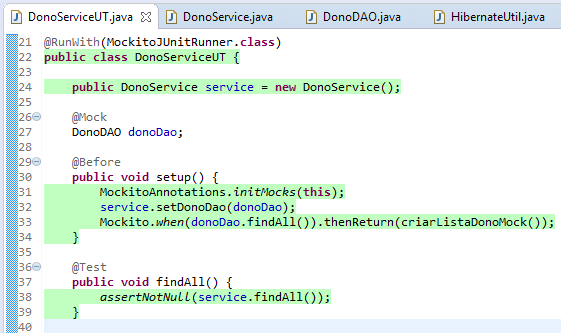
Iniciamos no Git com o Clone de um projeto, ao iniciarmos nossas atividades, modificando os arquivos estamos trabalhando no nosso "Workspace", ao finalizamos nossas iterações podemos adicionar ao nosso repositório local, com o comando "git add" para dar um 'index' para os arquivos modificamos, e depois o envio "git commit" para confirmar o envio para o repositório local. Com os arquivos disponíveis no repositório local podemos efetuar o envio para o repositório remoto, onde os demais podem obter o nosso código "git push". Todas as vezes que temos atualizações no repositório central, podemos obter através do comando "git pull".

Os comandos básicos são os citados acima. Nosso repositório pode ser organizado de diversas maneiras. Mas uma maneira bem interessante de organiza-lo é utilizando o git flow.



Temos a Branch Master que é responsável por manter a versão do código entregue para o cliente(geralmente a versão de produção), a branch develop que disponibiliza a versão mais estável da aplicação e as feature são as iterações de evolução da develop. Caso ocorra um erro na versão entregue para o cliente é criada uma Hotflix é a mesma é disponibilizada para a Develop e para a Master na sua finalização. As Releases são marcos no projeto, podem ser versões intermediárias ou pré-disponibilizadas para o cliente.

Ops! Já estávamos esquecendo dos testes unitários. Teste Unitário é um teste de Caixa Branca, exige conhecimento do código interno do programa. Teste Unitário testar uma única unidade. Para conseguirmos esses requisitos, vamos utilizar o Mockito para simular as saídas externas, e para executar o teste, vamos utilizar o JUnit.



@RunWith configura por onde iniciamos os testes unitários;

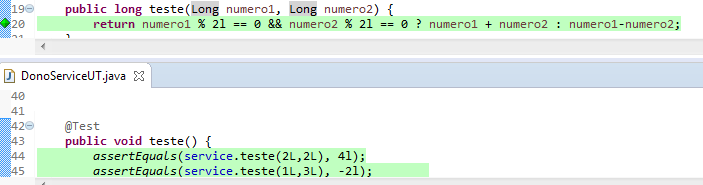
@Mock diz que essa classe será fake, um Mock;

@Before antes que os testes sejam de fatos executados, faça esses passos para mim;

@Test método que testa em si, com o seu(s) devidos Asserts.

Importante salienta, a função do Assert. É importante que no nosso código haja diversos Asserts, para de fato validar o código executado.

Vamos pensar em um método que recebe dois números, caso sejam ímpares, subtraem-se, caso seja pares somam-se.

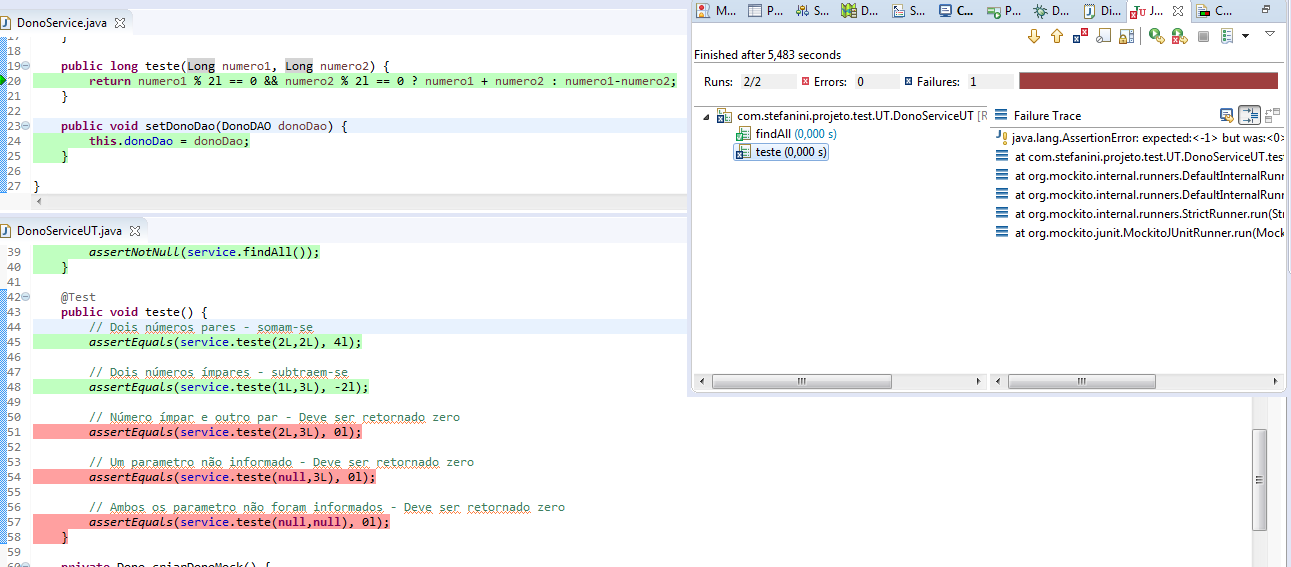


Tudo muito bonito, cobertura em 100%. Será que está tudo bem realmente?

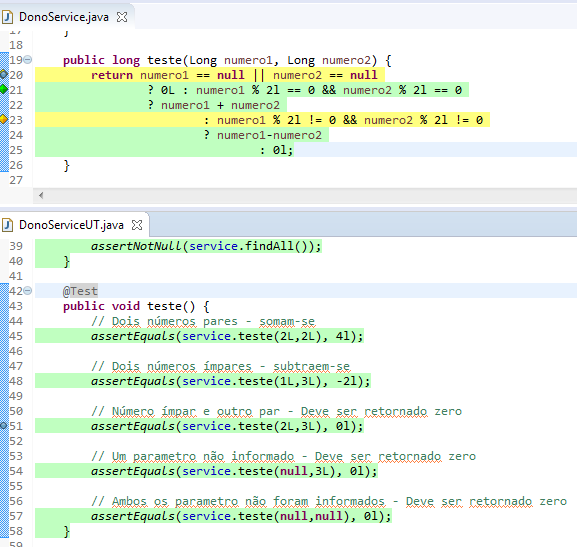
Se passarmos um valor ímpar e outro par? Está correto o resultado do código?

E se não informamos nenhum, ou nenhum dos dois números? Nosso código continua funcionado?

Resumindo, nosso código está precisando de atualizações. E o teste unitários pode nós auxiliar em relação a isso.



Algumas condições de nosso código não passaram, precisamos refatorar nosso código.



Muitas vezes, vamos nos deparar com códigos que estão com boa cobertura, mas negocialmente, não são tão interessantes. Essa responsabilidade e do desenvolvedor, é um teste de caixa branca da sua unidade. Fica a dica :)

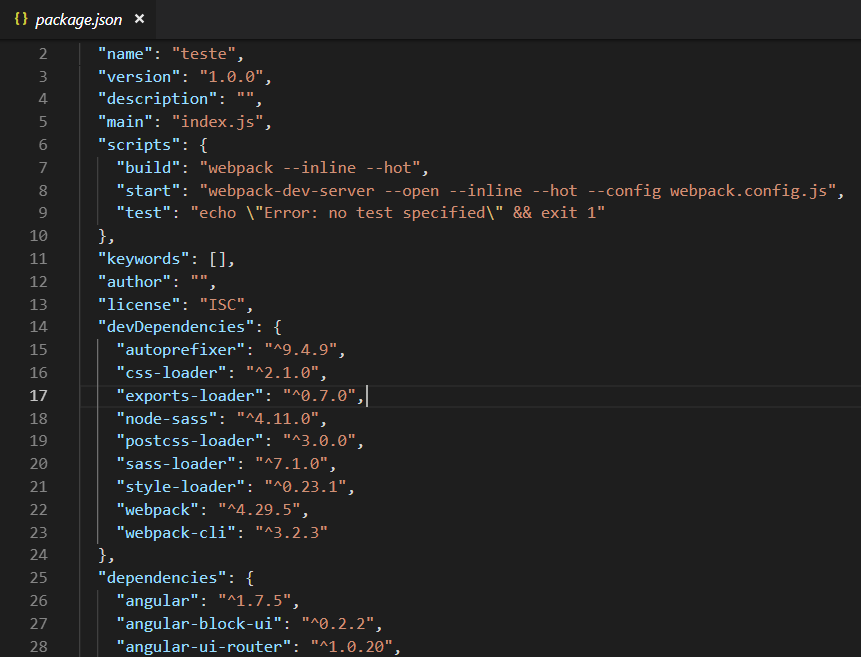
Como lição de casa, vamos criar esse Projeto 3 com os desafios repassados e disponibilizá-lo em um serviço que utilize o Git (GitHub ou GitLab, por exemplo).

**Projeto 4**

Nesse projeto vamos utilizar projeto anterior para prover a informação e vamos desenvolvedor uma consulta para aprendermos o Angular JS.

Um poderoso Framework Frontend.

Primeiro, arquivo que vamos entender sua função é o package.json:  Ponto de partida de qualquer projeto NodeJS. Ele é responsável por descrever o seu projeto, informar as engines (versão do node e do npm), url do repositório, versão do projeto, dependências de produção e de desenvolvimento dentre outras coisas.



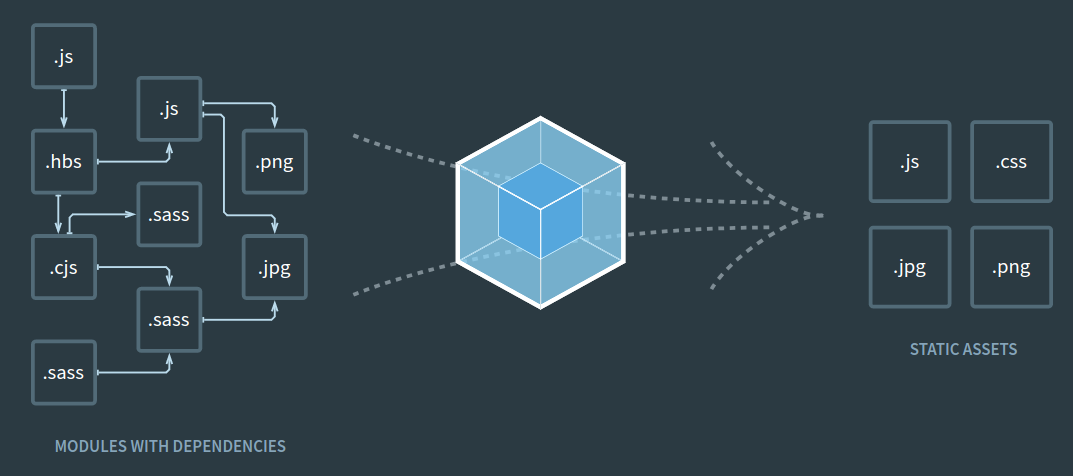
Vamos entender algumas opções:

Scripts: São as opções de execução disponíveis - npm + nomeScript 'start' = npm start;

DevDependencies: São as dependências de desenvolvimento, algumas bibliotecas não são necessárias no ambiente de produção. Então, temos apenas no ambiente de desenvolvimento.

Para efetuar o download e configurações das bibliotecas no package.json utiliza-se o comando 'npm install' ou simplemente 'npm i'.

No nosso projeto temos o WebPack que é um blunder JavaScript que permite que nosso código seja divido em módulos, carregados sobre demanda.



Depois do package.json é importante conhecemos o webpack.config.js <https://medium.com/criciumadev/configurando-webpack-es6-7-e5368e4e33c>.

No nosso app.js importamos os nossos módulos, tanto bibliotecas externas, como nossos módulos de negócio.

Para os nossos módulos internos, que implementam nossa lógica de negócio temos o nosso arquivo base que é o módulo.js que será importado pelo Webpack.

import angular from 'angular';

import uirouter from 'angular-ui-router';

import DonoController from './dono.controller';

export default angular.module('myApp.dono', [uirouter])

.controller('DonoController', DonoController)

.name;

A principal função desse módulo é disponibilizar um controlador para a view.html.

Controller

export default class DonoController {

constructor() {

this.name = 'Dono';

}

}

Html

<div class="jumbotron">

<h1>Hello, {{dono.name}}</h1>

</div>

Essa vinculação é efetuada pela biblioteca UI Router.

let donoState = {

name: 'dono',

url: '/dono',

templateUrl: './modulos/dono/dono.view.html',

controller: 'DonoController',

controllerAs: 'dono'

}

$stateProvider.state(donoState);

E tudo isso é encapsulado no módulo supra citado, para o WebPack.

import dono from './modulos/dono/dono.js';

import home from './modulos/home/home.js';

angular

.module('myApp', [

uirouter,

blockUI,

dono,

home

])

.config(routing);

Existem algumas diretivas importantes no Angular JS que são importantes conhecemos:

ng-app: inicializar a aplicação AngularJS;

ng-bind: coloca um observador sobre a variável, só é alterada quando tiver alteração de valor;

ng-model: vinculada uma propriedade ao escopo;

ng-class: permite efetuar alterações de estilo dinamicamente;

ng-click: efetua a chamada de uma função quando o elemento html é acionado;

ng-repeat: laço de iteração;

ng-show: adiciona uma condição para exibir um atributo;

ng-hide: adiciona uma condição para não exibir um atributo;

ng-if: adiciona uma condição em um elemento html;

Integração com o Backend?

Para as chamadas Rest, podemos utilizar o $http. Que nós retorna uma promise. Ou seja, é uma promessa que essa requisição será respondida, não sabemos quando, nem se essa requisição será respondida com sucesso. São as nossas services.

import angular from 'angular';

class DonoService {

constructor($http) {

this.$http = $http;

const apiBase = "http://localhost:8080/projeto3/api/";

this.path = apiBase + "dono";

}

getDonos() {

return this.$http.get(this.path);

}

}

export default angular.module('services.dono-service', [])

.service('donoService', DonoService)

.name;

Tratamento da promise.

vm.SolicitacaoService.getDonos().then(resp => {

this.donos = resp.dados();

}, error => {

this.mensagemErro = error;

});

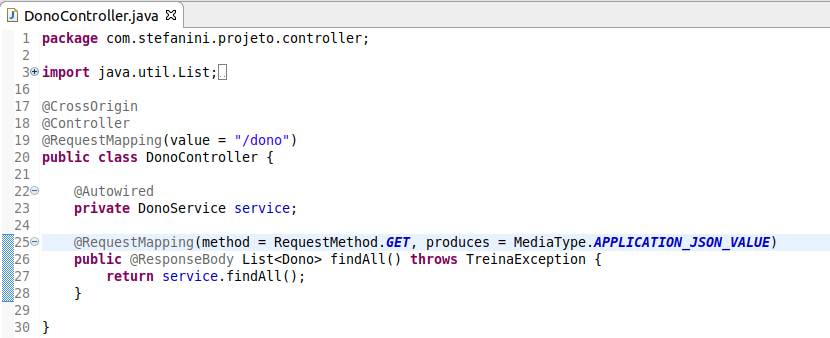
Finalizamos por aqui o nosso código.

Projeto 5

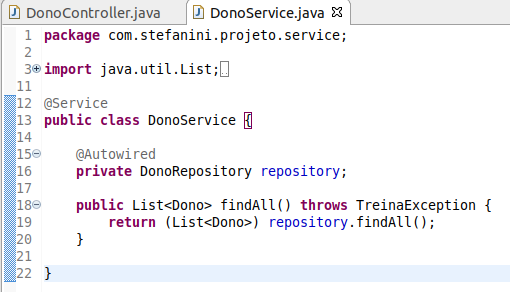
Vamos aprender Spring? Sim, Spring Boot.

Estrutura do nosso projeto com Spring será bem similar. Antes de tudo, vamos conhecer os Stereotypes do Spring:

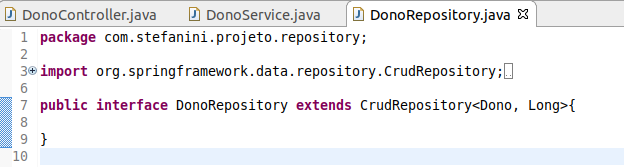
@Contoller - Utilizada para a camada de apresentação. Normalmente anotamos classes que representam a interação do usuário, como managed beans, actions e controllers;



@Service - Utilizada para a camada de aplicação e negócio. Comumente usada para anotar classes que denotam serviços de aplicação e de regras de negócio;

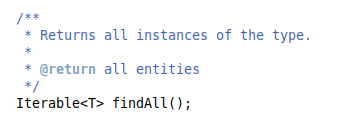


@Repository - Utilizada para a camada de persistência. Normalmente anotamos classes que representam um DAO, Repositório etc;

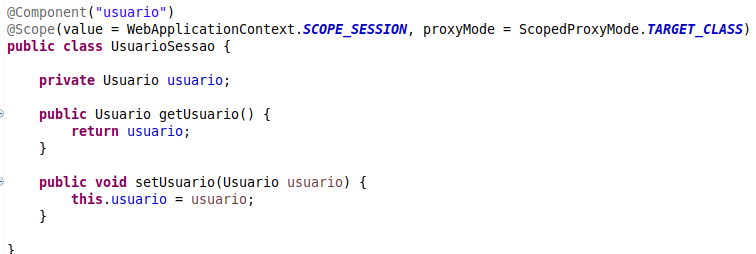


Ops! Cadê a consulta?

Estamos utilizando o Spring Data, e com isso, as consultas padrões são encapsuladas pelo Framework.



@Component - Utilizada para classes que não se encaixam em nenhuma das anotações acima. É comum anotar classes utils, parsers, helpers, processadores de arquivo etc;



Para utilizar, podemos, utilizar o Inject do Spring.



Ficou fácil agora né!?

Código também disponível no Github.