Documentando a implementação do código Spring

Por onde começar? Pois é, o difícil é definir por onde, por isso começaremos pelo básico, definindo quem serão as nossas tecnologias de trabalho, e avançando aos poucos. Mas antes de tudo, a primeira etapa a ser feita, é criar um repositório do projeto da Zup no GitHub, para salvar o projeto e disponibiliza-lo em um link, ao final deste artigo.

Começo informando que utilizaremos o JAVA 8 e o H2 para o Banco de Dados (banco de dados em memória do JAVA mais voltado para o desenvolvimento local), e apesar da escolha, reconheço que o PostGree também é muito utilizado por empresas para subir aplicações em um ambiente de produção. Além disso, para escrever o nosso código em Java, trabalharemos com uma IDE gratuita e muito utilizada, o Eclipse.

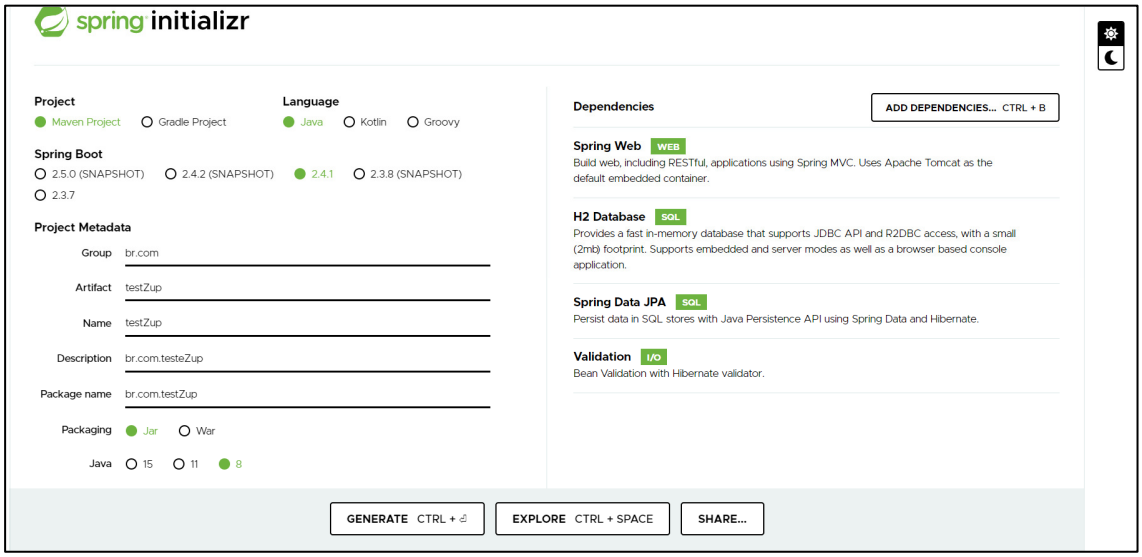
E mãos à obra, minha gente!

As imagens seguintes serão a indicação do passo a passo para a criação do projeto utilizando a tecnologia do Spring Inicializr, segue o endereço: https://start.spring.io/.

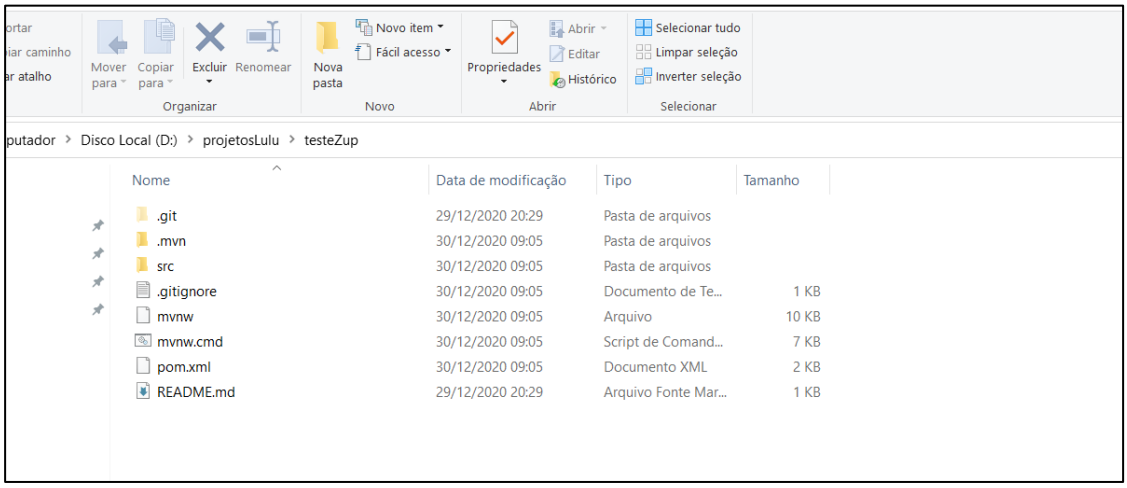
Tecnologias utilizadas:

- **Spring Web**: uma das funções do framework responsável por criar aplicações padrão Web;
- **H2 Database**: é a dependência responsável por utilizar um banco de dados em memória;
- **Spring Data JPA** é uma implementação do Spring para trabalhar com JPA.
- **Validation** validará o conteúdo a ser enviado através da API.

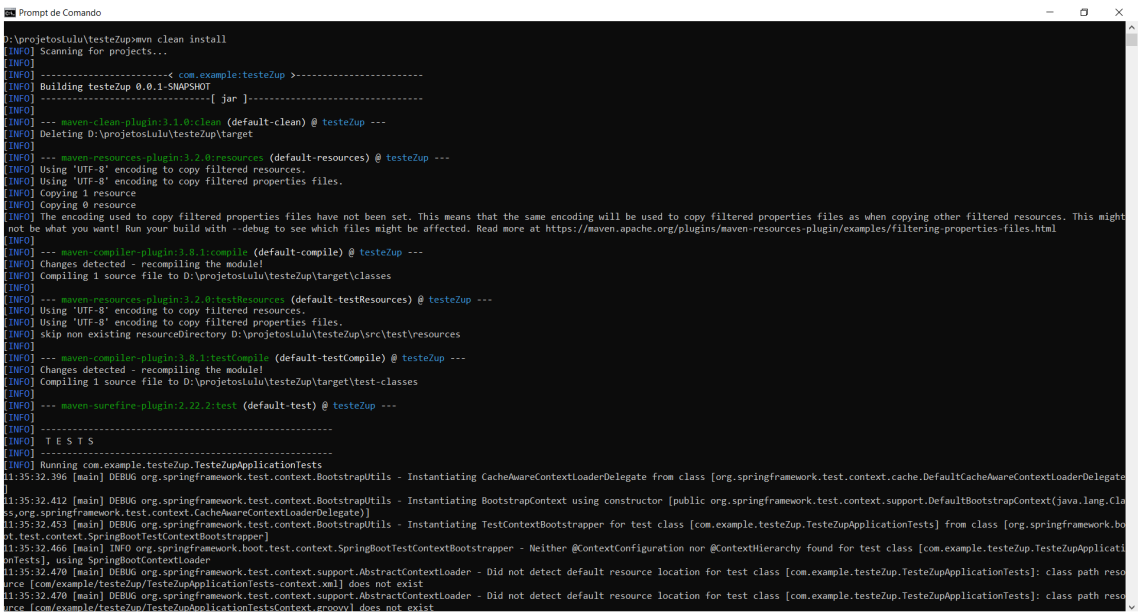
Começaremos pela configuração dos metadados básicos do projeto, e quais as suas dependências (**Web**, o **H2, Spring Data, e o Validation**), depois disso, é só baixar o Arquivo Zip com o projeto Spring Boot configurado e descompacta-lo onde você deseja, no meu caso, irei faze-lo utilizando o endereço D: na minha máquina local.



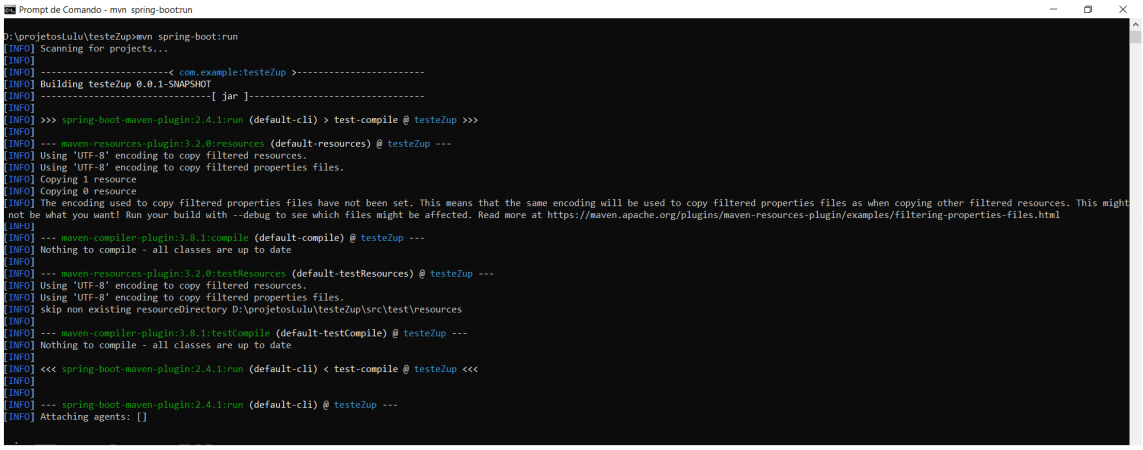
Note que o site Spring Inicializr criou automaticamente os arquivos src e o arquivo pom.xml, além de outros, mas estes não serão o foco deste projeto.



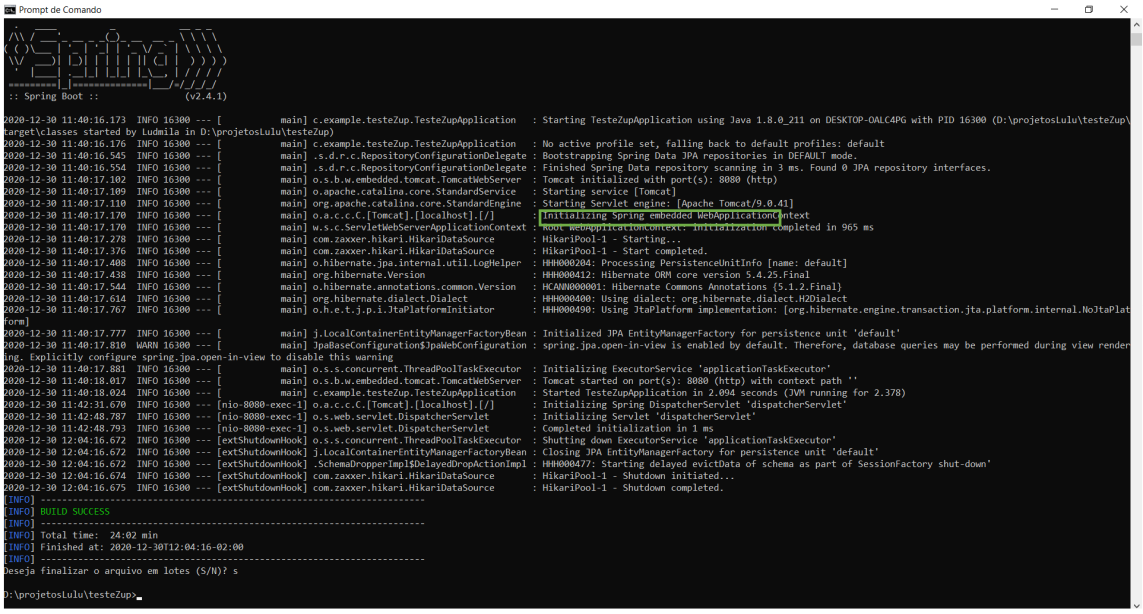
O próximo passo é *inicializar o projeto no terminal*, primeiramente localizamos o endereço da pasta, e na sequência digitamos o comando **mvn clean install**, e veja que o mvn irá instalar as dependências e verificar se há algo quebrando o código.



Note que o Build terminou com SUCCESS!!



Agora sim, o nosso projeto subiu e está rodando na porta 8080.



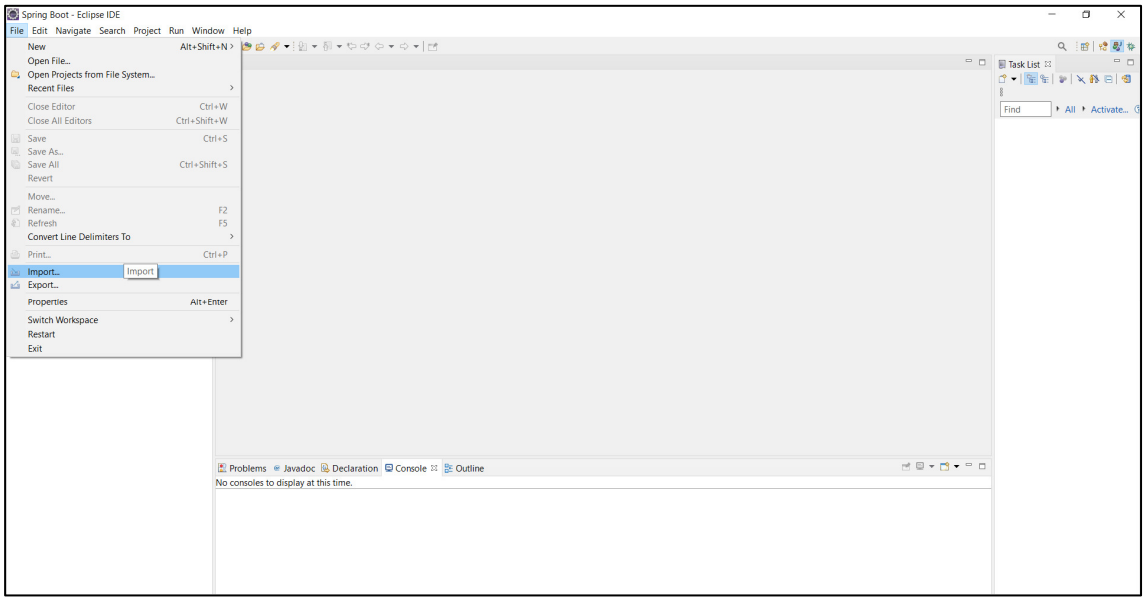
Para confirmar, utilizamos o navegador na URL http://localhost:8080/. A indicação do conteúdo *Whitelabel Error Page,* irá desaparecer assim que o projeto for concluído.

Conteúdo gerado pelo Spring Inicializr:

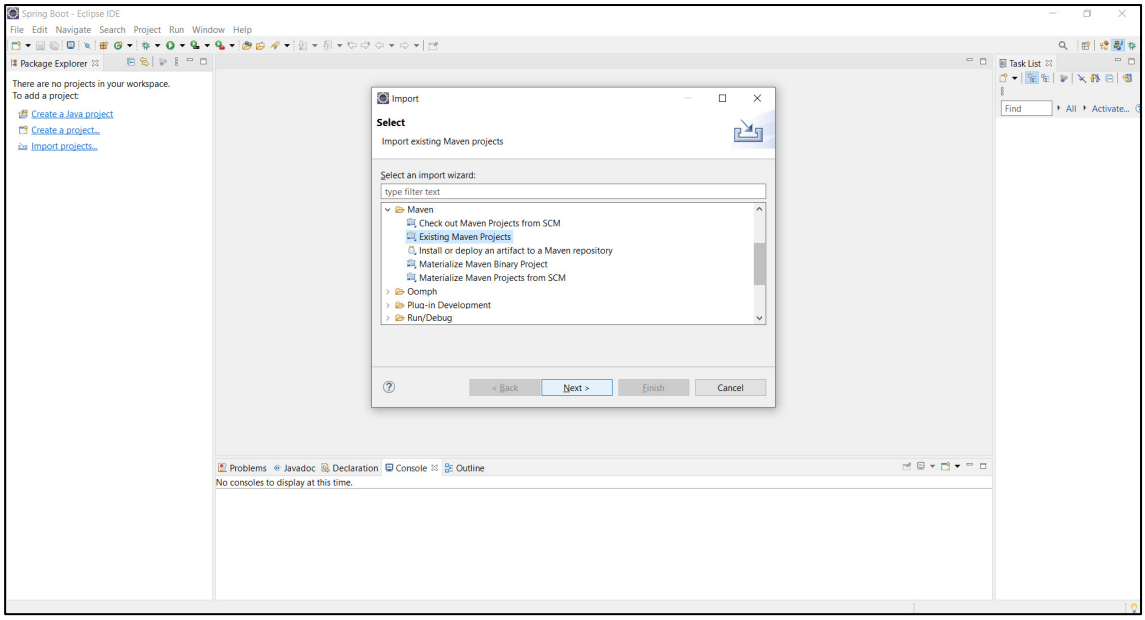
- **SRC**: na pasta src é onde serão gerados os arquivos do projeto;
- **POM.XML**: é onde estão os endereços de todas as dependências e plugins que serão utilizados no projeto, além de definirmos outras informações como versão do JAVA, descrição do projeto, nome do projeto, etc.

Etapas do processo

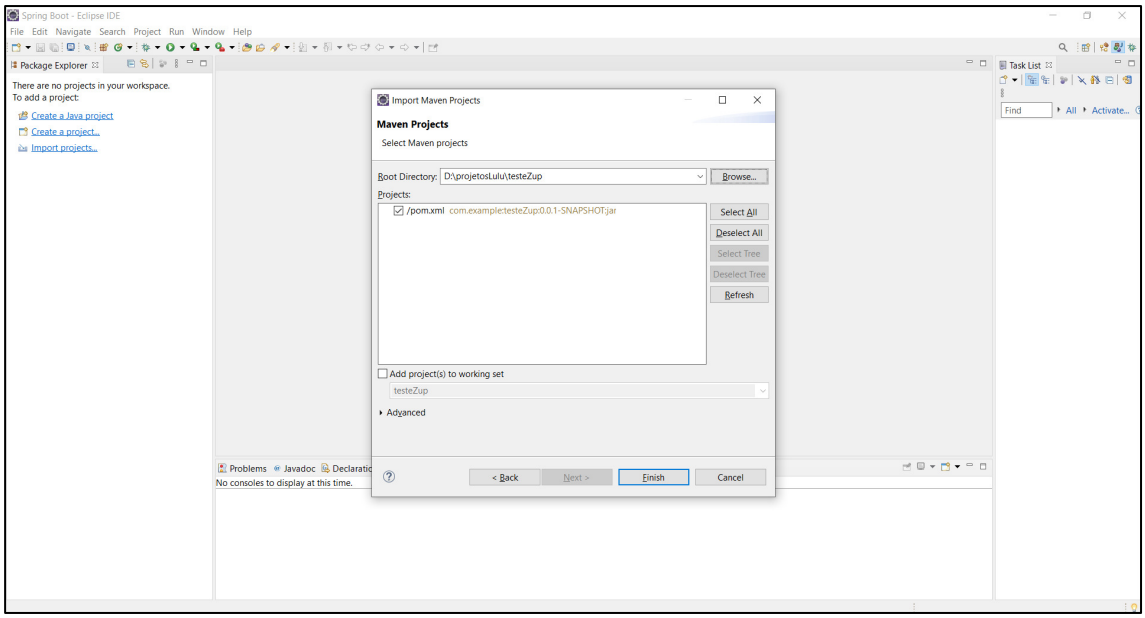
Depois que iniciar o Eclipse, vamos **importar** o projeto pré-definido pela ferramenta Spring Inicializr,



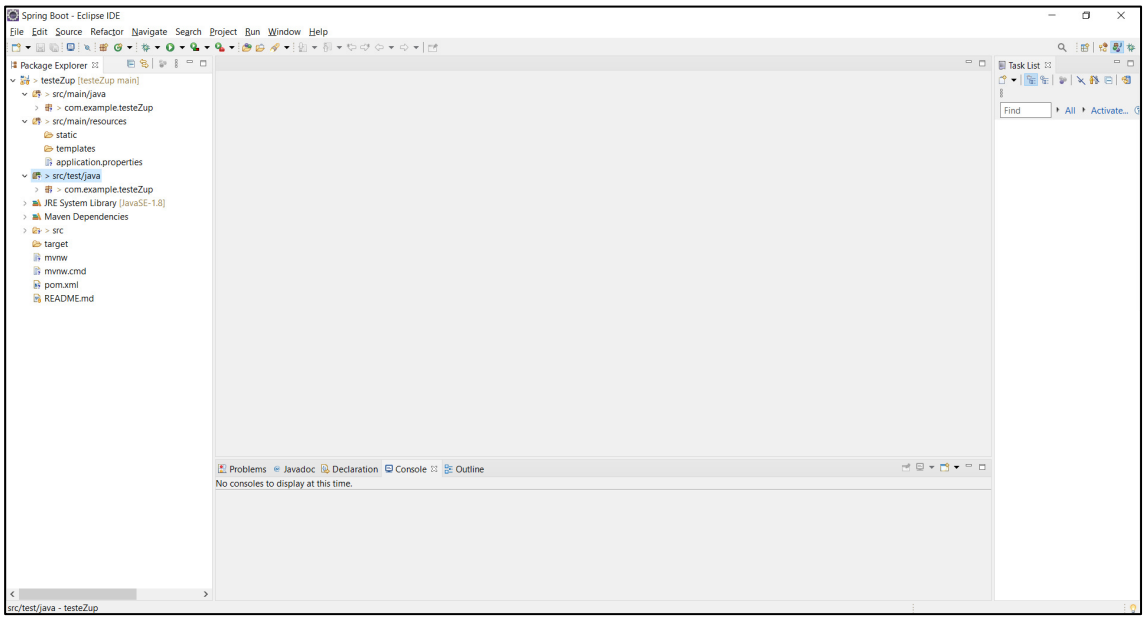
Depois selecione a pasta Maven/Existing Maven Projects, e clique em Next;



Agora clicamos em Browse e selecionamos o endereço onde o nosso projeto inicial foi salvo, e depois clicamos em Finish.



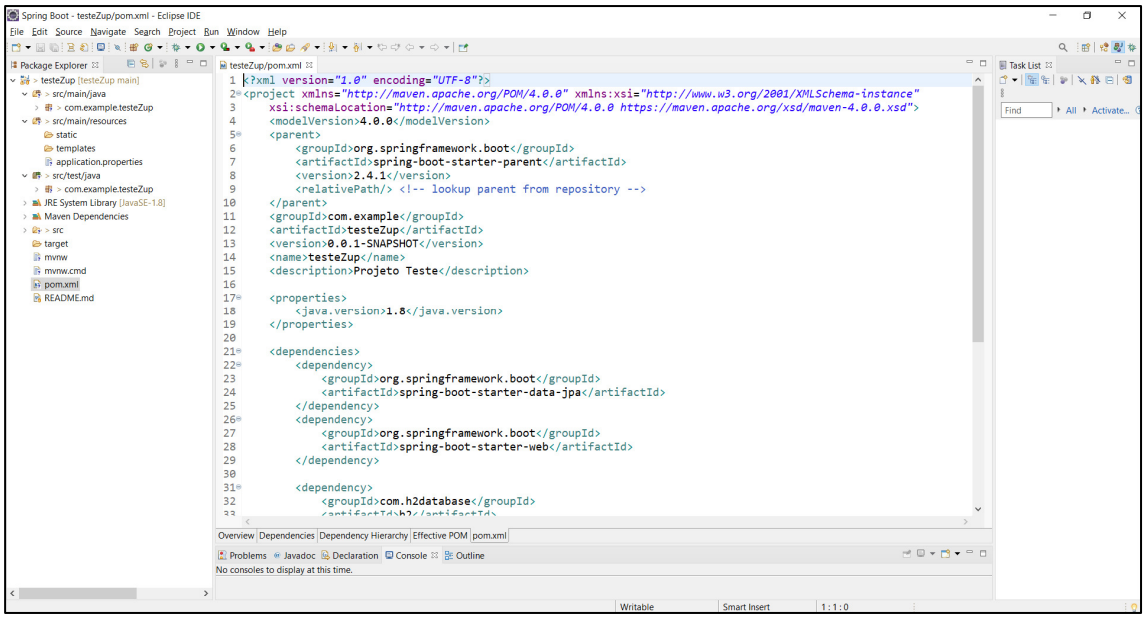
Observe que o projeto será aberto na pasta Package Explorer, na lateral esquerda da página;



Navegando entre as pastas, notamos que vários arquivos padrão foram criados quando utilizamos o Spring Inicializr;

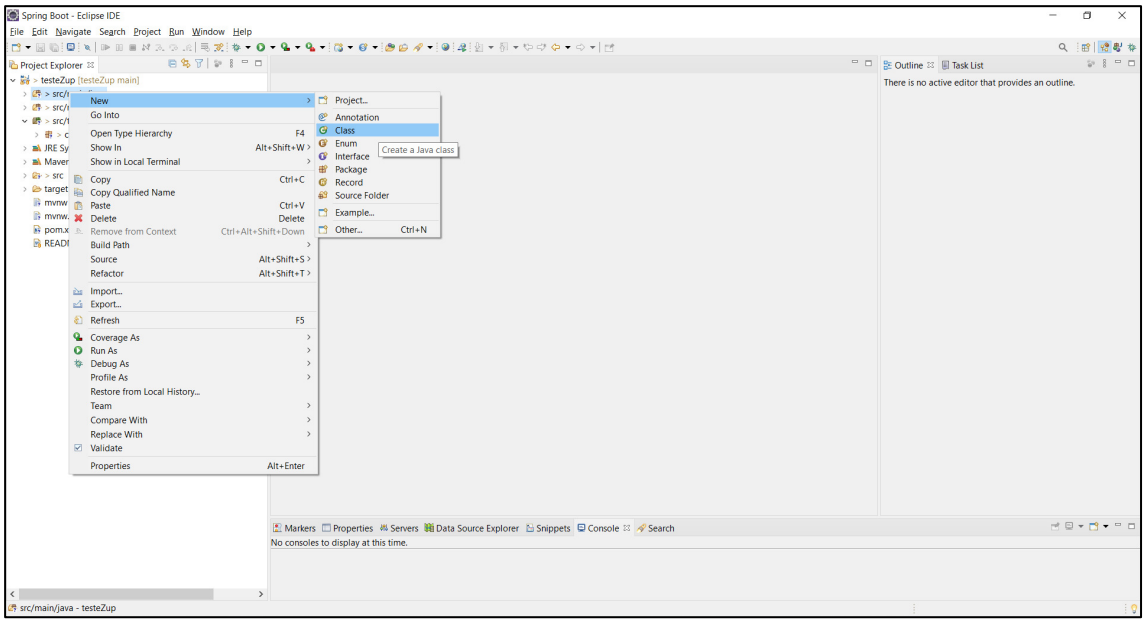
Abrindo o arquivo POM.XML a seguir, encontramos os dados da identidade do nosso projeto, além das dependências e plugins que serão utilizados.

É muito importante lembrar que havendo necessidade para a inclusão de mais dependências ou plugins, é aqui que acrescentaremos as mesmas. Ora dentro da guia: `<dependencies></dependencies>` ora dentro da guia `<plugins></plugins>`.

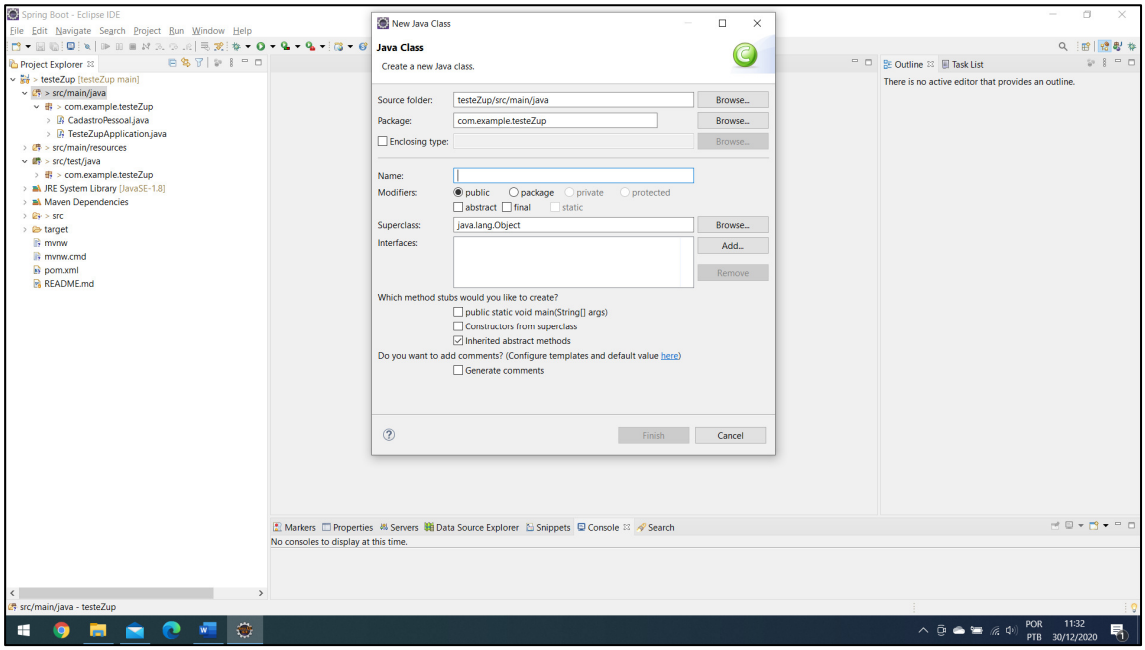


Antes de mais nada, informarei que utilizaremos a versão do *Eclipse IDE for JAVA for Enterprise JAVA Developers 2020-06*. Agora sim, vamos iniciar o nosso código.

Para criar uma classe, utilizaremos o caminho a seguir:



Selecionamos no **Browse** onde queremos salvar a nossa classe, e depois o nome que escolhemos.



A nova classe aparecerá no Package Explorer, ou Project Explorer (Depende de qual workspace estará sendo utilizada no momento);

Criando a Classe CadastroPessoa

```
package br.com.testeZup.modelo;

import java.sql.Date;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class CadastroPessoa {

	@Id @GeneratedValue(strategy = GenerationType.IDENTITY)
	private Long id;
	private String nome;
	private String email;
	private String cpf;
	private Date dataDeNascimento;

	public CadastroPessoa() {
	}

	public CadastroPessoa(Long id, String nome, String email, String cpf,
Date dataDeNascimento) {
		this.id = id;
		this.nome = nome;
		this.email = email;
```

```
		this.cpf = cpf;
		this.dataDeNascimento = dataDeNascimento;
	}

	public CadastroPessoa(String nome, String email, String cpf, Date
dataDeNascimento) {
		this.nome = nome;
		this.email = email;
		this.cpf = cpf;
		this.dataDeNascimento = dataDeNascimento;
	}

	public Long getId() {
		return id;
	}

	public void setId(Long id) {
		this.id = id;
	}

	public String getNome() {
		return nome;
	}

	public void setNome(String nome) {
		this.nome = nome;
	}

	public String getEmail() {
		return email;
	}

	public void setEmail(String email) {
		this.email = email;
	}

	public String getCpf() {
		return cpf;
	}

	public void setCpf(String cpf) {
		this.cpf = cpf;
	}

	public Date getDataDeNascimento() {
		return dataDeNascimento;
	}

	public void setDataDeNascimento(Date dataDeNascimento) {
		this.dataDeNascimento = dataDeNascimento;
	}
}
	}
}
```

Explicando um pouquinho sobre o código escrito...

Primeiramente comecei escrevendo todos os atributos da **classe CadastroPessoa** (id, nome, e-mail, cpf e dataDeNascimento), em seguida criei o **construtor da classe** (para a construção do objeto quando for chamado), e por último, criei os **getters e setters**, já que no caso do exemplo os atributos serão privados.

- @Entity – é a entidade responsável pelo mapeamento entre a tabela do banco de dados e a minha classe(CadastroPessoa);
- @Id - é o identificador da tabela;
- @GeneratedValue – geração automática de idenficadores.

Criando a Classe CadastroPessoaController

```
package br.com.testeZup.controller;

import java.util.List;
import java.util.Optional;

import javax.validation.Valid;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import br.com.testeZup.controller.form.CadastroPessoaForm;
import br.com.testeZup.modelo.CadastroPessoa;
import br.com.testeZup.repository.CadastroPessoaRepository;

@RestController
@RequestMapping("/cadastro")
public class CadastroPessoaController{

	@Autowired
	CadastroPessoaRepository cadastroRepository;

	@PostMapping
	public ResponseEntity abreConta(@RequestBody @Valid CadastroPessoaForm
form) {

		CadastroPessoa cadastroConversao = form.converter();
		try {
			Optional<CadastroPessoa> temCpf =
cadastroRepository.findByCpf(form.getCpf());
			Optional<CadastroPessoa> temEmail =
cadastroRepository.findByEmail(form.getEmail());
```

```
                if(temCpf.isPresent()) {
                      return ResponseEntity.ok("Cpf já existe");
                }

                if(temEmail.isPresent()) {
                      return ResponseEntity.ok("Email já existe");
                }

                cadastroRepository.save(cadastroConversao);

                return ResponseEntity.ok(cadastroConversao);

          } catch (Exception e) {

                return ResponseEntity
                      .status(HttpStatus.FORBIDDEN)
                      .body("Error Message " + e.getMessage());
          }

    }
```

Explicando um pouquinho sobre o código escrito...

Na classe Controller começo pela **criação da classe CadastroPessoaController**, contaremos com as seguintes etapas na nossa classe:
Começaremos injetando a classe **repositório CadastroPessoaRepository**;
Em seguida utilizaremos o método PostMapping, que falaremos mais em *anotações utilizadas* e por último criaremos um Try cath para tratar se o cpf e o e-mail já existem.

Anotações e referências utilizadas

- @RestController = notifica o SpringBoot que a classe Controller receberá requisições Rest (Post, Get, Put, Delete, Path, etc);
- @RequestMapping = é o mapeamento da rota onde a controller receberá as requisições (“/cadastro”);
- @Autowired = responsável por injetar a dependência do CadastroRepository;
- @PostMapping = é a anotação que indica para a Controller que receberemos requisições do tipo Post.
- ResponseEntity = envelopa a resposta em forma de objeto;
- @RequestBody = captura a informação do corpo do formulário de CadastroPessoaForm;
- Try cath = o nosso tratamento contará com a seguinte lógica, a ferramenta findByCpf e findByEmail começará verificando na tabela do banco de dados se a informação já existe, comparando-a com o formulário da classe CadastroPessoaForm, e caso já exista, o próximo passo será retornar uma mensagem cadastrada no if, e se houver qualquer erro além daqueles tratados na classe ErroDeValidação, exemplo; falha na conexão do banco de dados, ele pegará essa informação no Cath.

Criando a ClassePessoaForm

```
package br.com.testeZup.controller.form;

import java.sql.Date;

import javax.validation.constraints.Email;
import javax.validation.constraints.NotEmpty;
import javax.validation.constraints.NotNull;

import org.hibernate.validator.constraints.Length;

import br.com.testeZup.modelo.CadastroPessoa;

public class CadastroPessoaForm {

	@NotNull @NotEmpty
	private String nome;
	@NotNull @NotEmpty @Email
	private String email;
	@NotNull @NotEmpty @Length(min = 11)
	private String cpf;
	@NotNull
	private Date dataDeNascimento;

	public CadastroPessoa converter() {
		return new CadastroPessoa(this.nome, this.email, this.cpf,
this.dataDeNascimento);
	}
	public String getNome() {
		return nome;
	}
	public void setNome(String nome) {
		this.nome = nome;
	}
	public String getEmail() {
		return email;
	}
	public void setEmail(String email) {
		this.email = email;
	}
	public String getCpf() {
		return cpf;
	}
	public void setCpf(String cpf) {
		this.cpf = cpf;
	}
	public Date getDataDeNascimento() {
		return dataDeNascimento;
	}
	public void setDataDeNascimento(Date dataDeNascimento) {
		this.dataDeNascimento = dataDeNascimento;
	}

}
```

Explicando um pouquinho sobre o código escrito...

O que fizemos nesta classe foi escrever os argumentos do projeto, indicar o **tipo de validação** (optou-se pelo `@NotNull @NotEmpty @Length @Email )` e acrescentar os getters e setters, já que são argumentos privados, além do método converter, que transfere os dados de CadastroPessoa para que após ser preenchido persista no Banco de Dados;

Criando a Classe CadastroPessoaRepository

```
package br.com.testeZup.repository;

import java.util.Optional;

import org.springframework.data.jpa.repository.JpaRepository;

import br.com.testeZup.modelo.CadastroPessoa;

public interface CadastroPessoaRepository extends
JpaRepository<CadastroPessoa, Long>{

	Optional<CadastroPessoa> findByCpf(String cpf);

	Optional<CadastroPessoa> findByEmail(String email);


}
```

Explicando um pouquinho sobre o código escrito...

Criaremos uma interface chamada CadastroPessoaRepository que herdará da JpaRepository, todas as funções da Jpa que precisaremos para salvar, deletar, atualizar, listar, entre outras do banco de dados. E porque uma interface e não uma classe? Porque esta última será um contrato entre o nosso código e a JPA.

Implementações utilizadas da convenção do SpringData;

- findByCpf (String cpf) – procura na tabela do banco de dados, CadastroPessoa, o campo cpf, e retorna o registro igual;
- findByEmail (String email) – procura na tabela do banco de dados, CadastroPessoa, o campo Email, e retorna o registro igual;

O nosso TesteZupAplicattion terá a anotação `@SpringBootApplication`, que será a classe que utilizaremos para rodar a nossa aplicação dentro do main, onde há um Tomcat embutido e roda para subir a nossa aplicação, logo não precisamos adicionar o servidor separado.

Configurando o arquivo application.properties

É um arquivo onde constará as configurações do nosso projeto Spring, adicionaremos algumas linhas para configurar o H2, o JPA e o DataSource.

```
# data source
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.url=jdbc:h2:mem:testeZup
spring.datasource.username=sa
spring.datasource.password=

# jpa
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=update

# h2
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

Sobre a Package Validação

O pacote ***br.com.testeZup.config.validacao*** será responsável por capturar as nossas exceções e trata-las, confesso que ainda não dominei o que acontece na captura da exceção, mesmo assim, considerei utilizar a implementação, inclusive, retirei a implementação do código do curso Spring Boot da Alura realizado recentemente.

Observação

Todos os testes foram realizados no **Postman**.

Link do projeto

https://github.com/ludmilaaraujo/testeZup