

Olá meu querido(a) dev!

Você está procurando por um tutorial de como fazer uma API REST? Ok!

Mais especificamente uma API REST que faz cadastra os dados pessoas de uma pessoa para abrir uma conta no banco? Maravilha! Você está no lugar certo!

Neste post nós iremos utilizar Java 11, Spring Boot (um dos muitos projetos do [ecossistema Spring](#)) e Maven junto com outras tecnologias como Hibernate, Lombok, JUnit, JPA e Hibernate Validator. A Minha IDE de escolha será o IntelliJ, e por tanto, utilizaremos o [Spring Initializr](#) para criar o projeto.

Spring Boot

Antigamente as configurações tomavam muito tempo de quem queria desenvolver qualquer aplicação utilizando Spring, pois era necessário vários arquivos de configurações para que a aplicação pudesse rodar um “Hello World!”. Utilizando do conceito de configuração, foi criado o projeto/framework Spring Boot dentro do ecossistema Spring, que facilita e reduz o tempo gasto com arquivos de configurações já pré-configurados na sua aplicação. Não é algo novo uma vez que outros frameworks como Ruby on Rails já aplicavam este conceito.

Spring Data JPA

O Spring Data JPA faz parte de um projeto maior que é o [Spring Data](#), e nos dá o poder de trabalhar especificamente com JPA, além de trazer integração a outras tecnologias como Hibernate de uma forma mais fácil, prática e menos repetitiva do que apenas utilizar a sua especificação criada em 2006.

Lombok

Deixa nosso código [bem enxuto](#), [especialmente quando formos criar classes](#)

JUnit

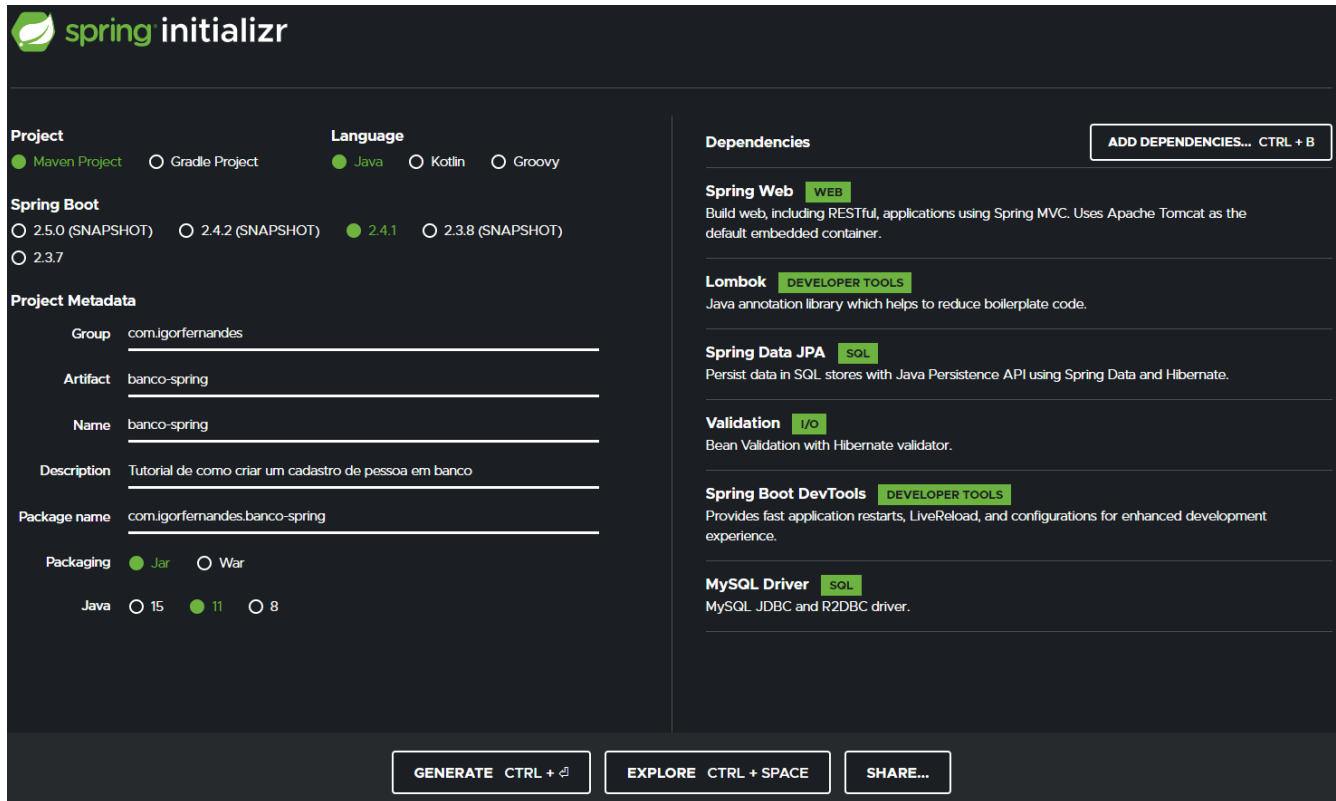
Principal framework para testes de todos os tipos em nossas aplicações, independente do tipo, e já vem como nosso framework padrão para testes na dependência spring-boot-starter-test localizado no nosso pom.xml

Hibernate Validator

Irá ajudar a validar os dados recebidos pela nossa API vindo dos [DTOs](#)

Antes de qualquer coisa, vamos criar nosso projeto no [Spring Initializr](#):

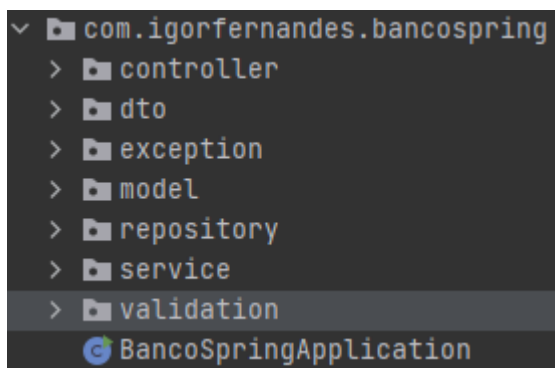
Opcionalmente você pode adicionar o [Spring DevTools](#) para facilitar o desenvolvimento sem ter que reiniciar a aplicação. Caso esteja utilizando o IntelliJ, recomendo também ler [este artigo](#) para configurar o reload.



The screenshot shows the Spring Initializr web interface. On the left, under 'Project', 'Maven Project' is selected. Under 'Language', 'Java' is selected. Under 'Spring Boot', '2.4.1' is selected. The 'Project Metadata' section includes fields for Group (com.igorfernandes), Artifact (banco-spring), Name (banco-spring), Description (Tutorial de como criar um cadastro de pessoa em banco), and Package name (com.igorfernandes.banco-spring). Under 'Packaging', 'Jar' is selected, and under 'Java', '11' is selected. On the right, the 'Dependencies' section lists several dependencies: Spring Web (WEB), Lombok (DEVELOPER TOOLS), Spring Data JPA (SQL), Validation (I/O), Spring Boot DevTools (DEVELOPER TOOLS), and MySQL Driver (SQL). At the bottom, there are buttons for 'GENERATE CTRL + G', 'EXPLORE CTRL + SPACE', and 'SHARE...'.

Após baixar e extrair o projeto, abra-o em sua IDE de preferência.

Criaremos nossas classes seguindo esta estrutura de arquivos:



Trabalharemos em torno de uma entidade chamada Pessoa, que conterá os campos Nome, Email, CPF e DataNascimento. Aplicaremos também algumas regras a ela:

- Todos os campos String (Nome, Email e CPF) não podem ser nulos ou em branco
- Nome deve conter entre 3 a 40 caracteres
- Email e CPF devem ser únicos
- A data de nascimento não pode ser nulo e deve ser no passado

Seguindo um pouco dos padrões estabelecidos pelo [12factor](#), nos deixamos nosso application.properties de forma que o usuário e a senha do nosso banco estejam em uma variável de ambiente (Não esqueça de criar o banco antes!):

```
# Config Banco
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/bancospring?
useSSL=false&useTimezone=true&serverTimezone=UTC
spring.datasource.username=${DB_USER}
spring.datasource.password=${DB_PASS}

# Config JPA
# Caso tenha problemas com compatibilidade de versão utilize o MySQL5Dialect
spring.jpa.database-platform=org.hibernate.dialect.MySQL5InnoDBDialect
spring.jpa.show-sql=true # Mostra a query sendo executada
spring.jpa.hibernate.ddl-auto=update
```

Daqui em diante iremos utilizar também o Lombok para reduzir nosso código, e assim criamos nossa classe de modelo Pessoa.

```
@Entity
@Getter
@NoArgsConstructor
public class Pessoa {

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

    @Column(length = 60)
    private String nome;

    @Column(unique = true)
    private String email;

    @Column(length = 11, unique = true)
    private String cpf;

    @Column(name = "data_nascimento")
    private LocalDate dataNascimento;

    public Pessoa(String nome, String email, String cpf, LocalDate dataNascimento) {
        this.nome = nome;
        this.email = email;
        this.cpf = cpf;
        this.dataNascimento = dataNascimento;
    }
}
```

Nosso Controller:

Service:

```
@Service
public class PessoaService {

    @Autowired
    private PessoaRepository pessoaRepository;

    public Pessoa save(Pessoa pessoa){
        return pessoaRepository.save(pessoa);
    }

    public Boolean existsByEmail(String email){
        return pessoaRepository.existsByEmail(email);
    }

    public Boolean existsByCpf(String cpf){
        return pessoaRepository.existsByCpf(cpf);
    }

    public List<Pessoa> findAll(){
        return pessoaRepository.findAll();
    }
}
```

Repository: (Repare que estamos utilizando apenas nomes para criarmos as queries)

```
// Query creation:
// https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation
// https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#appendix.query.method.subject
public interface PessoaRepository extends JpaRepository<Pessoa, Long> {

    Boolean existsByEmail(String email);
    Boolean existsByCpf(String cpf);
}
```

Nossos DTOs presentes no Controller.java são:

PessoaDTO (DTO de envio do cliente):

```
// Ordem de exibição dos campos
@JsonPropertyOrder({"nome", "email", "cpf", "dataNascimento"})
@Value // Objeto imutável
public class PessoaDTO {

    @NotBlank(message = "O nome não pode ser vazio!")
    @Size(min = 3, max = 40, message = "O nome precisa ter entre {min} e {max} caracteres!")
    private String nome;

    @UniqueEmail
    @NotBlank(message = "E-mail precisa ser fornecido!")
    @Email(message = "E-mail inválido!")
    private String email;

    @UniqueCPF
    @CPF(message = "CPF Inválido!", ignoreRepeated = true) // Caelum Stella
    @NotBlank(message = "CPF não pode ser vazio!")
    private String cpf;

    @NotNull(message = "A data de nascimento precisa ser fornecida!")
    @Past(message = "A data precisa ser no passado!")
    private LocalDate dataNascimento;

    public Pessoa transformaParaObjeto() {
        return new Pessoa(nome, email, cpf, dataNascimento);
    }
}
```

PessoaRespostaDTO (DTO de resposta da nossa API):

```
@JsonPropertyOrder({"id", "nome", "email"})
@Value
public class PessoaRespostaDTO {
    private Long id;
    private String nome;
    private String email;
    // CPF Ocultado

    public static PessoaRespostaDTO transformaEmDTO(Pessoa pessoa) {
        return new PessoaRespostaDTO(pessoa.getId(), pessoa.getNome(), pessoa.getEmail());
    }
}
```

Repare que nós inserimos duas annotations que não existem no pacote das Validations que são o `@UniqueEmail` e `@UniqueCPF`. Por padrão, podemos tratar e evitar customizadamente nossos erros de duplicidade através do utilizando um handler customizado e tratando o erro da classe `ConstraintViolationException` que contém os erros retornados pelo banco. Porém aqui nós iremos adotar um método diferente do convencional: Vamos criar duas annotations que fazem isso para nós!

Criando as annotations:

UniqueCPF.java

```
@Documented
@Target({ElementType.FIELD, ElementType.METHOD}) // Campos e métodos
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = {UniqueCPFValidator.class}) // Classe validadora
public @interface UniqueCPF {
    String message() default "CPF já cadastrado!";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};
}
```

UniqueEmail.java

```
@Documented
@Target({ElementType.FIELD, ElementType.METHOD}) // Campos e métodos
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = {UniqueEmailValidator.class}) // Classe validadora
public @interface UniqueEmail {
    String message() default "Email já cadastrado!";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};
}
```

Validadores:

UniqueCPFValidator.java

```
public class UniqueCPFValidator implements ConstraintValidator<UniqueCPF, String>{

    @Autowired
    private PessoaService pessoaService;

    @Override
    public boolean isValid(String cpf, ConstraintValidatorContext constraintValidatorContext) {
        return !pessoaService.existsByCpf(cpf);
    }
}
```

UniqueEmailValidator.java

```
public class UniqueEmailValidator implements ConstraintValidator<UniqueEmail, String> {  
    @Autowired  
    private PessoaService pessoaService;  
  
    @Override  
    public boolean isValid(String email, ConstraintValidatorContext constraintValidatorContext) {  
        return !pessoaService.existsByEmail(email);  
    }  
}
```

Pronto, com isso nós podemos validar campos de email e cpf que possam vir duplicados dentro das funções, por exemplo:

```
recebeCPF(@UniqueCPF String cpf)
```

Ou marcando-os em cima de propriedades como fizemos na classe pessoaDTO:

```
@UniqueCPF  
private String cpf;
```

Além disso, optei por adicionar a lib Caelum Stella que provê uma validação de CPF como fizemos em PessoaDTO. Caso faça isso, adicione essas dependências ao seu pom.xml:

```
<!-- Validação de CPF -->  
<!--  
    Nota: Não use esta lib em situações reais.  
    Um CPF real deve ser buscado via Receita Federal ou WS integrado a mesma.  
-->  
<dependency>  
    <groupId>br.com.caelum.stella</groupId>  
    <artifactId>caelum-stella-core</artifactId>  
    <version>2.1.2</version>  
</dependency>  
  
<dependency>  
    <groupId>br.com.caelum.stella</groupId>  
    <artifactId>caelum-stella-bean-validation</artifactId>  
    <version>2.1.2</version>  
</dependency>
```

Testes

Por fim mas não menos importante, iremos criar uma classe robusta de testes para garantir que tudo esta funcionando como deveria. Para isso usaremos as libraries que vem por padrão na dependência spring-boot-starter-test, como o JUnit e Spring MockMvc.

```

@SpringBootTest // Auto configuração das classes e configurações da aplicação
@AutoConfigureMockMvc // Auto configura o Spring MockMvc
@TestInstance(TestInstance.Lifecycle.PER_CLASS) // Cria uma instância por teste
class PessoaControllerTest {
    @Autowired
    private MockMvc mockMvc;

    @Autowired
    private ObjectMapper objectMapper;

    @Autowired
    private PessoaService pessoaService;

    @Autowired
    private PessoaRepository pessoaRepository;

    // Data fictícia
    private LocalDate localDate = LocalDate.of(2000, 1, 1);

    @BeforeEach
    void setup() {
        pessoaRepository.deleteAll();
    }

    @Test
    void salvar() throws Exception {
        PessoaDTO pessoaDTO = new PessoaDTO("Teste", "Teste@gmail.com", "1111111111", localDate);

        mockMvc.perform(post("/pessoas").contentType(
            "application/json;charset=UTF-8").content(objectMapper.writeValueAsString(pessoaDTO)))
            .andExpect(status().isCreated())
            .andExpect(content().contentType(MediaType.APPLICATION_JSON));
    }

    // Nome
    @Test
    void salvarNomeInvalidoTest() throws Exception {
        testarNome(null, "O nome não pode ser vazio!");
        testarNome("aa", "O nome precisa ter entre 3 e 40 caracteres!");
    }

    void testarNome(String nome, String respostaEsperada) throws Exception {
        PessoaDTO pessoaDTO = new PessoaDTO(nome, "Teste@gmail.com", "1111111111", localDate);

        mockMvc.perform(
            post("/pessoas").contentType(
                "application/json;charset=UTF-8").content(objectMapper.writeValueAsString(pessoaDTO)))
            .andExpect(status().isBadRequest())
            .andExpect(jsonPath("$.nome").value(respostaEsperada))
            .andExpect(content().contentType(MediaType.APPLICATION_JSON));
    }

    // Email
    @Test
    void salvarEmailInvalido() throws Exception {
        // DRY - pessoa válida no banco
        salvar();

        testarEmail(null, "E-mail precisa ser fornecido!");
        testarEmail("Teste", "E-mail inválido!");
        testarEmail("Teste@gmail.com", "E-mail inválido!");
        testarEmail("Teste@gmail.com", "Email já cadastrado!");
    }

    void testarEmail(String email, String respostaEsperada) throws Exception {
        PessoaDTO pessoaDTO = new PessoaDTO("aaa", email, "1111111111", localDate);

        mockMvc.perform(
            post("/pessoas").contentType(
                "application/json;charset=UTF-8").content(objectMapper.writeValueAsString(pessoaDTO)))
            .andExpect(status().isBadRequest())
            .andExpect(jsonPath("$.email").value(respostaEsperada))
            .andExpect(content().contentType(MediaType.APPLICATION_JSON));
    }
}

```

```

// CPF
// Nota: Como setamos o ignoreRepeated = true na classe PessoaDTO
// nós podemos registrar CPFs com números repetidos como "11111111111".

@Test
void salvarCPFInvalido() throws Exception {
    // DRY - pessoa válida no banco
    salvar();

    testarCPF(null, "CPF não pode ser vazio!");
    testarCPF("3", "CPF Inválido!"); // Caelum Stella
    testarCPF("34274324832487387", "CPF Inválido!"); // Caelum Stella
    testarCPF("11111111111", "CPF já cadastrado!");
}

void testarCPF(String cpf, String respostaEsperada) throws Exception {
    PessoaDTO pessoaDTO = new PessoaDTO("aaa", "Teste@gmail.com", cpf, LocalDate);

    mockMvc.perform(
        post("/pessoas").contentType(
            "application/json;charset=UTF-8").content(objectMapper.writeValueAsString(pessoaDTO))
            .andExpect(status().isBadRequest())
            .andExpect(jsonPath("$.cpf").value(respostaEsperada))
            .andExpect(content().contentType(MediaType.APPLICATION_JSON));
    }
}

```

Muito bem! Nossos testes estão funcionais e mostram que nossa API e mostram que ela está validando corretamente nossos campos!

✓ Test Results	592 ms
✓ PessoaControllerTest	592 ms
✓ salvarCPFInvalido()	520 ms
✓ salvar()	17 ms
✓ salvarNomeInvalidoTest()	25 ms
✓ salvarEmailInvalido()	30 ms

Com isso nós concluímos o objetivo básica de nossa API fazer uma integração contínua via TravisCI de novas funcionalidades.

Espero que tenha gostado do nosso post de hoje! Dúvidas e sugestões são bem-vindas nos comentários, até a próxima!

Fontes utilizadas para a construção desse projeto:

Spring DevTools:

<https://www.baeldung.com/spring-boot-devtools>

<https://mkkyong.com/spring-boot/intellij-idea-spring-boot-template-reload-is-not-working/>

Estrutura:

<https://medium.com/the-resonant-web/spring-boot-2-0-project-structure-and-best-practices-part-2-7137bdcba7d3>

Exceptions:

<https://www.baeldung.com/exception-handling-for-rest-with-spring>
<https://www.baeldung.com/global-error-handler-in-a-spring-rest-api>

Bean Validation:

<https://cursos.alura.com.br/forum/topico-bean-validation-em-versoes-mais-atuais-do-spring-boot-113451>
<https://www.baeldung.com/spring-boot-bean-validation>
<https://www.devmedia.com.br/jpa-como-usar-a-anotacao-generatedvalue/38592>
<https://github.com/caelum/caelum-stella/wiki> -- Validação de CPF com Bean Validation

Testes:

<https://medium.com/@gcbrandao/testando-uma-api-rest-spring-boot-2-com-junit5-e-mockmvc-db603c65a306>
<https://stackoverflow.com/questions/32952884/junit-beforeclass-non-static-work-around-for-spring-boot-application>
<https://www.baeldung.com/integration-testing-in-spring>
<https://howtodoinjava.com/spring-boot2/testing/springboottest-annotation/>