1º Entidade Usuário(Model)

Primeiramente, criaremos as entidades, mais especificamente, a entidade Usuario. Para isso, crie um novo pacote de nome “model.entity” dentro de “minhasfinancas/model”. A entidade usuário apresenta os seguintes atributos: nome, e-mail e senha.

(https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#reference)

Uma visão geral de uma classe em java que representa Usuário seria:

```
package com.bcipriano.minhasfinancas.model.entity;

import java.util.Objects;

public class Usuario {
    public Usuario(String nome, String email, String senha) {
        this.nome = nome;
        this.email = email;
        this.senha = senha;
    }

    public Usuario() {
    }

    private String nome;

    private String email;

    private String senha;

    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 getSenha() {
        return senha;
    }

    public void setSenha(String senha) {
        this.senha = senha;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Usuario usuario = (Usuario) o;
        return Objects.equals(nome, usuario.nome) && Objects.equals(email, usuario.email) && Objects.equals(senha,
usuario.senha);
    }
```

```
    @Override
    public int hashCode() {
        return Objects.hash(nome, email, senha);
    }

    @Override
    public String toString() {
        return "Usuario{" +
                "nome='" + nome + '\'' +
                ", email='" + email + '\'' +
                ", senha='" + senha + '\'' +
                '}';
    }
}
```

No entanto, precisamos considerar que essa aplicação deve transformar objetos instanciados a partir dessa classe em tabelas no banco de dados e transformar as tabelas em instâncias novamente. Para que isso seja possível, o SpringBoot utiliza o JPA(Java Persistence API), que nada mais é que uma especificação do Java EE para operações na base de dados. Suas principais características são:

Documentacao: (https://docs.spring.io/spring-data/jpa/docs/current/reference/html/)

1) Mapeamento Objeto Relacional (ORM) //Orientação a objetos e modelo relacional//
2) Evita a criação de SQL natio misturado com código Java.
3) Abstração da camada do banco de dados
4) Provê formas simples de realizar as operações na base de dados por meio da aplicação Java
5) Migra para outros SGBD's necessitando apenas de alterar sua referência no pom.xml

Observação:
A partir das versões do spring boot mais atuais, as anotations relacionadas as entidades foram transferidas do javax.persistence para o jakarta.persistence. Em suma, não houve grandes alterações, exceto pelo fato de que as anotations "@Entity", "@Column", "@Table"… são realizados do jakarta.persistence.

Com base nessas considerações, o mapeamento da entidade Usuario fica da seguinte forma:

```
package com.bcipriano.minhasfinancas.model.entity;

import jakarta.persistence.*;

import java.util.Objects;

@Entity
@Table(name = "usuario", schema="financas")
public class Usuario {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id; //Id da base de dados

    @Column(name = "nome")
    private String nome;

    @Column(name = "email")
```

```
    private String email;

    @Column(name = "senha")
    private String senha;

    public Usuario(Long id, String nome, String email, String senha) {
        this.id = id;
        this.nome = nome;
        this.email = email;
        this.senha = senha;
    }

    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 getSenha() {
        return senha;
    }

    public void setSenha(String senha) {
        this.senha = senha;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Usuario usuario = (Usuario) o;
        return Objects.equals(id, usuario.id) && Objects.equals(nome, usuario.nome) && Objects.equals(email,
usuario.email) && Objects.equals(senha, usuario.senha);
    }

    @Override
    public int hashCode() {
        return Objects.hash(id, nome, email, senha);
    }

    @Override
    public String toString() {
```

```
    return "Usuario{" +
            "id=" + id +
            ", nome='" + nome + '\'' +
            ", email='" + email + '\'' +
            ", senha='" + senha + '\'' +
            '}';
  }
}
```

Veja o significado de cada uma das anotations usadas:

@Entity:é usada na classe de modelo de dados para identificá-la como uma entidade no contexto do JPA (Java Persistence API). Esta anotação indica ao JPA que esta classe será mapeada para uma tabela no banco de dados relacional. Cada instância dessa classe será persistida como uma linha na tabela correspondente.
@Table: é usada para mapear uma entidade a uma tabela específica no banco de dados. Com ela, você pode especificar o nome da tabela a ser usada, bem como o esquema (se for o caso) e outras configurações relacionadas à tabela, como índices e restrições de chave estrangeira.
@Id: é usada para identificar a chave primária de uma entidade no banco de dados. Ela é usada para indicar qual atributo representa a chave primária na tabela correspondente no banco de dados. A anotação @Id geralmente é usada em conjunto com a anotação @GeneratedValue para indicar como a chave primária deve ser gerada (por exemplo, automaticamente pelo banco de dados ou pelo próprio aplicativo). Essa combinação de anotações é usada para garantir que cada registro na tabela tenha uma chave única e que o banco de dados possa indexar esses registros de maneira eficiente.
@GenerationValue: é usada para especificar a geração automática do valor da chave primária para uma entidade no JPA (Java Persistence API). Isso significa que, ao invés de um programador especificar manualmente o valor da chave primária, ele será gerado automaticamente pelo banco de dados ou pelo provedor de persistência JPA. A anotação @GeneratedValue tem alguns atributos que podem ser usados para especificar a estratégia de geração de valor da chave primária. Algumas das estratégias comuns incluem: • GenerationType.AUTO: Deixa o provedor de persistência escolher a estratégia mais adequada para a geração de valor da chave primária. • GenerationType.IDENTITY: Usa uma coluna de auto-incremento no banco de dados para gerar o valor da chave primária. • GenerationType.SEQUENCE: Usa uma sequência do banco de dados para gerar o valor da chave primária. • GenerationType.TABLE: Usa uma tabela separada para gerar o valor da chave primária.

@Column: permite que o programador especifique o nome da coluna na tabela da base de dados relacionada a um determinado atributo da entidade. Além disso, também é possível definir outros aspectos da coluna, como seu tipo, tamanho, se pode ser nulo ou não, entre outras configurações. Isso permite que haja uma maior flexibilidade e controle sobre a estrutura das tabelas na base de dados.

Uma ferramenta que usaremos para simplificar o código é o Lombok. Como sabemos toda classe java possui estruturas básicas relacionadas a cada atributo como getters, setters, construtores, toString, equals, hashCode.

O Lombok é uma biblioteca Java que facilita a escrita de código Java repetitivo, como getters, setters, construtores, entre outros. Ele permite que você reduza a quantidade de código escrito e aumente a produtividade. Alguns dos conceitos-chave do Lombok incluem:

1. Anotações: O Lombok utiliza anotações para identificar as classes e métodos que precisam ser modificados. Algumas das anotações mais comuns são @Data, @Getter, @Setter e @NoArgsConstructor.

2. Gerador de código: O Lombok (https://projectlombok.org) tem um gerador de código incorporado que gera automaticamente o código necessário, baseado nas anotações utilizadas.

3. Validation: O Lombok também oferece anotações de validação, como @NotNull e @NonNull, que permitem que você adicione validações em seu código sem escrever muito código.

4. Métodos conveniência: Além dos getters, setters e construtores, o Lombok também fornece outros métodos úteis, como equals(), hashCode() e toString().

5. Plugin de IDE: O Lombok é compatível com várias IDEs, incluindo IntelliJ IDEA, Eclipse e NetBeans. É necessário instalar um plugin na IDE para que as anotações sejam reconhecidas e o código gerado automaticamente.

Em geral, o Lombok é uma ferramenta muito útil para programadores Java que buscam automatizar tarefas repetitivas e aumentar a produtividade.

Observações:
Para usar o Lombok é necessário incluí-lo no pom.xml e ter a extensão Lombok.plugin em sua ide.

Veja como fica o código aplicando essa ferramenta:

```java
package com.bcipriano.minhasfinancas.model.entity;

import jakarta.persistence.*;
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@Entity
@Table(name = "usuario", schema="financas")
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class Usuario {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id; //Id da base de dados

    @Column(name = "nome")
    private String nome;

    @Column(name = "email")
```

```
    private String email;

    @Column(name = "senha")
    private String senha;

}
```

Usamos as seguintes anotações do Lombok:

@Data: A anotação @Data do Lombok é uma anotação abreviada que gera automaticamente métodos getters e setters para todos os atributos, além de toString, equals, hashCode e outros métodos comuns de uma classe de Java. Com ela, é possível escrever classes mais curtas e legíveis, sem precisar escrever todos os métodos manualmente. Além disso, ela também adiciona a anotação @Getter e @Setter para cada atributo, tornando possível customizar a geração dos métodos ao adicionar outras anotações aos atributos, como @NonNull ou @Builder.
@Builder: é usada para gerar uma classe builder interna para a classe que está sendo anotada. Uma classe builder é uma classe que permite a criação de objetos de maneira fácil e legível, especialmente quando os objetos têm muitos atributos ou são complexos. A anotação `@Builder` gera um construtor de classe com todos os atributos e um método "build" que retorna uma instância da classe. É possível também definir valores padrão para alguns ou todos os atributos da classe através do builder. Além disso, a anotação `@Builder` permite que os atributos sejam definidos de forma fluente, o que torna o código mais legível e fácil de entender.
@NoArgsContructor: é utilizada para gerar automaticamente um construtor sem argumentos para a classe. Esse construtor é gerado com a finalidade de facilitar a criação de objetos sem precisar passar nenhum argumento para ele, tornando mais fácil trabalhar com essa classe. O construtor gerado pela anotação @NoArgsConstructor é útil, por exemplo, na criação de objetos que serão preenchidos posteriormente com os valores necessários. A anotação @NoArgsConstructor pode ser usada em combinação com outras anotações do Lombok para gerar construtores com diferentes configurações, como por exemplo, construtores com argumentos ou construtores com configurações avançadas de construção.

@AllArgsContructor: é usada para criar automaticamente um construtor que recebe todos os atributos da classe como argumentos. Isso pode ser útil em casos onde você deseja fornecer uma maneira simples de criar instâncias da classe, especialmente quando há muitos atributos. Ao adicionar esta anotação a uma classe, o Lombok irá gerar automaticamente um construtor que tem um argumento para cada atributo na classe, na ordem em que eles aparecem.

2º Repositório Usuário (Model)

Ainda dentro da camada model, construiremos o UsuárioRepository no pacote “/model/repository”. Suas linhas de código são:

```
package com.bcipriano.minhasfinancas.model.repository;

import com.bcipriano.minhasfinancas.model.entity.Usuario;
import org.springframework.data.jpa.repository.JpaRepository;

public interface UsuarioRepository extends JpaRepository<Usuario, Long> {
}
```

Vejamos os principais conceitos sobre JpaRepository:

JPA Repository é uma interface do Spring que permite acesso aos dados do banco de dados de maneira simples e eficiente. É uma interface genérica que estende a interface Spring Data JPA e fornece métodos padrão para realizar operações CRUD como salvar, atualizar, buscar e excluir entidades.

Aqui estão alguns dos principais métodos fornecidos pela JPA Repository:

1. save - Salva uma nova entidade no banco de dados ou atualiza uma entidade existente.
2. findOne - Busca uma entidade específica pelo seu ID.
3. findAll - Busca todas as entidades de uma determinada classe.
4. delete - Exclui uma entidade específica do banco de dados.
5. count - Retorna o número total de entidades no banco de dados.

Você pode definir o seu próprio repositório JPA extendendo a interface JPA Repository e definindo qual é a entidade e o tipo de ID que você deseja trabalhar.

Além disso, convém lembrar que precisaremos de mais dois métodos que não são padrão da interface JpaRepository. São eles:

```
package com.bcipriano.minhasfinancas.model.repository;

import com.bcipriano.minhasfinancas.model.entity.Usuario;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.Optional;

public interface UsuarioRepository extends JpaRepository<Usuario, Long> {
   boolean existsByEmail(String email);
   Optional<Usuario> findByEmail(String email);
}
```

Descrição sobre as linhas de código dessa classe:
O método existsByEmail verifica se existe um usuário com o email especificado e retorna true se existir e false se não existir.
O método findByEmail busca um usuário pelo email especificado e retorna um objeto Optional que pode ou não conter um usuário. O objeto Optional é uma classe que encapsula o resultado de uma operação que pode ou não retornar um valor. Isso é útil para evitar null pointer exceptions ao buscar por entidades que podem ou não existir na base de dados.
Com isso, a classe UsuarioRepository é uma interface que permite que você interaja com a base de dados de usuários de maneira fácil e segura.

3º Repositório Usuário (Serviço)

Agora, iniciaremos a camada de serviços para a Entidade usuário. Para isso, primeiramente devemos criar uma interface de nome “UsuarioService” para declarar a assinatura dos métodos que serão implementados posteriormente pela classe “UsuarioServiceImpl”.

```
package com.bcipriano.minhasfinancas.service;

import com.bcipriano.minhasfinancas.model.entity.Usuario;

public interface UsuarioService {

   Usuario autenticar(String email, String senha);

   Usuario salvarusuario(Usuario usuario);

   void validarEmail(String email);
}
```

Feito isso, criaremos agora dentro do pacote “service” um novo pacote de nome “impl”(de implementação). Esse pacote conterá o UsuarioServiceImpl, classe que implementará efetivamente os métodos assinados pela interface Usuario service.

```
package com.bcipriano.minhasfinancas.service.impl;

import com.bcipriano.minhasfinancas.model.entity.Usuario;
import com.bcipriano.minhasfinancas.service.UsuarioService;
import org.springframework.stereotype.Service;

@Service
public class UsuarioServiceImpl implements UsuarioService {

   @Override
   public Usuario autenticar(String email, String senha) {
      return null;
   }

   @Override
   public Usuario salvarusuario(Usuario usuario) {
      return null;
   }

   @Override
   public void validarEmail(String email) {

   }
}
```

Como foi dito no material “2-Arquitetura”, é nessa camada que implementaremos as regras de negócio da entidade usuário.

Primeiramente devemos criar um novo atributo para a classe, o “usuarioRepostitory” e fazer a injeção de dependencias com a anotation “@Autwired”.

```
private UsuarioRepository usuarioRepository;

@Autowired
public UsuarioServiceImpl(UsuarioRepository usuarioRepository){
    this.usuarioRepository = usuarioRepository;
}
```

Temos portanto:

```
package com.bcipriano.minhasfinancas.service.impl;

import com.bcipriano.minhasfinancas.model.entity.Usuario;
import com.bcipriano.minhasfinancas.model.repository.UsuarioRepository;
import com.bcipriano.minhasfinancas.service.UsuarioService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class UsuarioServiceImpl implements UsuarioService {

    private UsuarioRepository usuarioRepository;

    @Autowired
    public UsuarioServiceImpl(UsuarioRepository usuarioRepository){
        this.usuarioRepository = usuarioRepository;
    }

    @Override
    public Usuario autenticar(String email, String senha) {
        return null;
    }

    @Override
    public Usuario salvarusuario(Usuario usuario) {
        return null;
    }

    @Override
    public void validarEmail(String email) {

    }
}
```

Explicação do código:
A classe UsuarioServiceImpl é uma implementação da interface UsuarioService e é anotada com @Service, o que indica ao framework Spring que essa classe é um componente gerenciado pelo Spring e pode ser injetado como dependência em outras classes.
A propriedade repository é a dependência que essa classe precisa para funcionar corretamente. Em vez de instanciar diretamente essa dependência, o Spring é usado para injetá-la no construtor da classe usando a anotação @Autowired. Isso significa que o Spring irá procurar por uma implementação da interface UsuarioRepository e passá-la como argumento para o construtor quando uma instância de UsuarioServiceImpl for criada.
Dessa forma, a classe UsuarioServiceImpl não precisa se preocupar com a criação da dependência, ela apenas a usa. Isso torna a classe mais fácil de testar, já que você pode fornecer uma implementação falsa da dependência durante os testes, e também a torna mais fácil de manter, já que você pode mudar a implementação da dependência sem afetar a classe que a usa.

1)Implementação dos métodos de UsuarioServiceImpl:

```
@Override
public Usuario autenticar(String email, String senha) {
   Optional<Usuario> usuario = usuarioRepository.findByEmail(email);

   if(!usuario.isPresent()){
      throw new ErroAutenticacao("Usuario não encontrado!");
   }

   if(!usuario.get().getSenha().equals(senha)){
      throw new ErroAutenticacao("Senha inválida!");
   }

   return usuario.get();
}
```

Primeiramente chamamos o método findByEmail de usuarioRepository. Se esse e-mail não for encontrado uma exceção será lançada. Caso o e-mail seja válido, o usuário trazido da base de dados terá sua senha comparada com a senha passada como argumento. Caso contrário uma exceção de "Senha inválida" será exibida. Por fim, o método retorna o usuário carregado.
[Lembrando que o método get da classe Optional retorna um objeto, nesse caso, retorna usuário]
A classe de erro, é do tipo RunTimeException e foi criada no pacote exception.

2) Implementação do método "salvarUsuario":

```
@Override
@Transactional
public Usuario salvarusuario(Usuario usuario) {
   validarEmail(usuario.getEmail());
   return usuarioRepository.save(usuario);
}
```

Esse método recebe um objeto usuário, depois disso ele verifica se o e-mail passado se encontra na base de dados. Caso essa condição seja retorne "false" ele chama o método padrão do JpaRepository "save".

Observação:
Essa anotation "@Transactional" garante que a operação de persistência no banco de dados seja atômica, ou seja, ou todas as alterações ocorrem ou nenhuma delas ocorre.

3) Implementação da classe "validaEmail":

```
@Override
public void validarEmail(String email) {
   boolean existe = usuarioRepository.existsByEmail(email);
   if(existe){
      throw new RegraNegocioException("Já existe um usuário cadastrado com esse e-mail.");
   }
}
```

Esse método recebe uma string e-mail, a partir disso, ele verifica se essa e-mail existe na base de dados pelo método "usuarioRepository". Se ele existir uma exception(RegraNegocioException do tipo RunTimeException) é lançada. Esse método foi usado no método "salvarUsuario".

Classe UsuarioRepositoryImpl com todas as implementações:

```
package com.bcipriano.minhasfinancas.service.impl;

import com.bcipriano.minhasfinancas.exception.ErroAutenticacao;
```

```java
import com.bcipriano.minhasfinancas.exception.RegraNegocioException;
import com.bcipriano.minhasfinancas.model.entity.Usuario;
import com.bcipriano.minhasfinancas.model.repository.UsuarioRepository;
import com.bcipriano.minhasfinancas.service.UsuarioService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.Optional;

@Service
public class UsuarioServiceImpl implements UsuarioService {

    private UsuarioRepository usuarioRepository;

    @Autowired
    public UsuarioServiceImpl(UsuarioRepository usuarioRepository){
        this.usuarioRepository = usuarioRepository;
    }

    @Override
    public Usuario autenticar(String email, String senha) {
        Optional<Usuario> usuario = usuarioRepository.findByEmail(email);

        if(!usuario.isPresent()){
            throw new ErroAutenticacao("Usuario não encontrado!");
        }

        if(!usuario.get().getSenha().equals(senha)){
            throw new ErroAutenticacao("Senha inválida!");
        }

        return usuario.get();
    }

    @Override
    @Transactional
    public Usuario salvarUsuario(Usuario usuario) {
        validarEmail(usuario.getEmail());
        return usuarioRepository.save(usuario);
    }

    @Override
    public void validarEmail(String email) {
        boolean existe = usuarioRepository.existsByEmail(email);
        if(existe){
            throw new RegraNegocioException("Já existe um usuário cadastrado com esse e-mail.");
        }
    }
}
```

4º Testes de integração (Modelos)

Conceitos sobre teste de integração e testes unitários:

Testes de integração: são importantes para garantir que as diferentes partes de uma aplicação sejam capazes de trabalhar juntas de forma efetiva. No Spring Boot, você pode escrever testes de integração usando as ferramentas fornecidas pelo próprio framework, como o TestRestTemplate para testar APIs REST.

Além disso, é importante entender como configurar e utilizar bancos de dados para testes, bem como como criar mocks e stubs para simular o comportamento de dependências externas. O uso de ferramentas como o Mockito pode ajudar nessa tarefa.

Também é possível utilizar o Spring Test para simplificar a configuração de testes de integração, permitindo a utilização de anotações como @SpringBootTest e @AutoConfigureMockMvc, que ajudam a configurar a aplicação e a simular requisições HTTP, por exemplo.

Testes Unitários: Testes unitários são testes automatizados que verificam o comportamento de pequenas partes (unidades) isoladas de código, como métodos de uma classe ou funções de um módulo. Esses testes são importantes porque permitem identificar e corrigir erros em partes específicas do código de forma rápida e confiável.

Os testes unitários são escritos em geral pelos desenvolvedores, utilizando frameworks de teste como o JUnit, e podem ser executados com frequência durante o processo de desenvolvimento para garantir que as mudanças não quebrem o código existente. Além disso, eles permitem que o desenvolvedor tenha mais confiança na qualidade do seu código e facilite a manutenção futura.

No Spring Boot, é possível escrever testes unitários para os componentes da aplicação, como controladores, serviços e repositórios, e utilizar mocks e stubs para simular o comportamento de dependências externas, tornando os testes mais rápidos e isolados. O Spring Test também fornece suporte para a escrita de testes unitários em conjunto com testes de integração.

Organização dos testes no SpringBoot

Existem 3 tipo de pacotes para testes no spring boot:
(https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.testing)

@SpringBootTest: Este pacote é utilizado para testes de integração em que a aplicação é iniciada por completo e executada em um ambiente simulado. Ele carrega toda a aplicação, bem como as configurações e dependências necessárias, para realizar testes que simulam o comportamento do sistema como um todo.

@WebMvcTest: Este pacote é utilizado para testes de integração específicos para o MVC, em que somente as camadas de controle e de apresentação da aplicação são iniciadas e testadas, enquanto as demais camadas são simuladas com mocks. Isso permite testar a lógica do controlador e das requisições HTTP sem executar toda a aplicação.

@DataJpaTest: Este pacote é utilizado para testes de integração específicos para acesso a banco de dados. Ele carrega somente as classes de configuração do JPA e o EntityManager, para que você possa testar consultas e operações no banco de dados de forma isolada, sem ter que iniciar toda a aplicação.

Primeiramente iniciaremos os testes usando o @SpringBootTest para visualizar como funcionam os testes realizados com essa anotação e depois realizaremos os mesmos testes usando o @DataJpaTest.

Antes de começar, é importante lembrar que cada pacote de teste pode conter testes unitários ou de integração correspondentes aos componentes do pacote que faz referência. Embora não seja uma regra, é importante que os pacotes de testes sejam criados de acordo a organização do pacote principal para ficar fácil de compreender posteriormente.

Construção dos teste para UsuarioRepository:

Antes de realizarmos os testes, criaremos usaremos uma ferramenta muito útil para testar as classes que integram funcionalidades do JPARepository sem ter que adicionar dados de teste em nossa base de dados real. Essa ferramenta é o banco de dados H2, um sistema de gerenciamento de banco de dados sql que salva os dados somente em tempo de execução. Veja como adiciona-lo ao projeto:

1° Criar um novo arquivo dentro de “resources”com nome de “application-test.properties”. Ele deverá contér a seguinte string de conexão:

```
spring.datasource.url=jdbc:h2:mem:db;DB_CLOSE_DELAY=-1;INIT=CREATE SCHEMA IF NOT EXISTS financas
spring.datasource.username=sa
spring.datasource.password=sa
spring.datasource.driver-class-name=org.h2.Driver
```

2º Depois disso, basta adicionar ao “pom.xml” a seguinte dependência:

```
<dependency>
	<groupId>com.h2database</groupId>
	<artifactId>h2</artifactId>
</dependency>
```

Dentro do pacote de “/testes/bcipriano/model/repository/” crie uma nova classe de nome “UsuarioRepositoryTest”. Essa classe terá as seguintes linhas de código:

```
package com.bcipriano.minhasfinancas.model.repository;

import com.bcipriano.minhasfinancas.model.entity.Usuario;
import org.assertj.core.api.Assertions;
import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.junit4.SpringRunner;

@SpringBootTest
@ActiveProfiles("test")
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
@RunWith(SpringRunner.class)
public class UsuarioRepositoryTest {

   @Autowired
   UsuarioRepository usuarioRepository;

   @Test
   public void deveVerificarAExistenciaDeUmEmail(){
```

```
        //Cria um novo usuario
        Usuario usuario = Usuario.builder().nome("usuario").email("usuario@email.com").build();

        //Salva na base de dados
        usuarioRepository.save(usuario);

        //Procura pelo email salvo
        boolean result = usuarioRepository.existsByEmail("usuario@email.com");

        //Verifica se o resultado é verdadeiro
        Assertions.assertThat(result).isTrue();

    }

}
```

Significado de cada uma das anotations que ainda não foram vistas:

@ActiveProfiles(“test”): é usada para ativar um perfil específico ao executar testes no Spring Framework. Os perfis do Spring são usados para configurar diferentes ambientes de execução da aplicação, como desenvolvimento, teste e produção. Cada perfil pode ter suas próprias configurações, como configurações de banco de dados, configurações de segurança e outras configurações específicas do ambiente. Ao usar a anotação @ActiveProfiles("test"), você está ativando o perfil "test" ao executar os testes de unidade ou de integração da sua aplicação. Isso pode ser útil para garantir que as configurações corretas sejam usadas durante os testes, como o uso de um banco de dados de teste em vez do banco de dados de produção.
Além do perfil "test", você pode ativar outros perfis usando a mesma anotação, como @ActiveProfiles("dev") ou @ActiveProfiles("prod"), por exemplo. Você também pode usar a anotação @Profile para definir um perfil específico em uma classe ou método de configuração do Spring.
[É valido lembrar que quando usamos essa anotação, o framework irá procurar pela string de conexão contida em “application-test.properties”]

@RunWith(SpringRunner.class): é usada para especificar o runner do JUnit que será usado para executar os testes de unidade ou integração no Spring Framework.
Na verdade, a classe SpringRunner é uma implementação da interface Runner do JUnit que é usada para executar testes com o Spring Framework. O SpringRunner inicializa o contexto de aplicação do Spring antes de executar os testes e, em seguida, limpa o contexto após a execução dos testes.
Isso significa que, ao usar a anotação @RunWith(SpringRunner.class), você está ativando o suporte do Spring Framework para testes de unidade ou integração e permitindo que o Spring gerencie as dependências e os objetos de teste. Além disso, a anotação @RunWith permite que você use outras anotações de teste do Spring, como @Autowired, @MockBean e @WebMvcTest.
Observe que a partir do JUnit 5, a anotação @RunWith foi substituída pela anotação @ExtendWith. Portanto, se você estiver usando o JUnit 5 ou posterior, deve usar a anotação @ExtendWith(SpringExtension.class) em vez de @RunWith(SpringRunner.class).

Breve explicação sobre Junit:
O JUnit é um framework de testes de unidade para a linguagem de programação Java. Ele fornece um conjunto de anotações e classes que facilitam a criação, execução e análise de testes de unidade automatizados em projetos Java.
O objetivo principal do JUnit é garantir que cada método ou componente individual da aplicação esteja funcionando corretamente e produzindo os resultados esperados. Ao criar testes de unidade com o JUnit, é possível verificar rapidamente se as mudanças no código fonte estão produzindo o comportamento esperado.

Essa ferramenta é amplamente utilizada em projetos de desenvolvimento de software, especialmente em projetos de desenvolvimento ágil, onde a prática de testes automatizados é fundamental para garantir a qualidade do software. O JUnit também é usado em conjunto com outros frameworks de teste do ecossistema Java, como o TestNG e o Mockito.

Além disso é distribuído como uma biblioteca de classes Java e é executado dentro do ambiente de execução do Java, como o JVM. O framework pode ser usado com qualquer ambiente de desenvolvimento integrado (IDE) que suporte a linguagem Java, como o Eclipse, NetBeans e IntelliJ IDEA, e é compatível com uma ampla variedade de ferramentas de build e integração contínua, como o Maven e o Jenkins.

@Test: é uma das anotações mais usadas no JUnit e é usada para marcar um método como um método de teste. Essa anotação indica que o método deve ser executado como um teste de unidade ou integração pelo JUnit.

O método marcado com a anotação @Test deve ter a assinatura pública, sem argumentos e sem retorno (void). Ele pode realizar qualquer operação de teste necessária para verificar o comportamento de um método ou componente individual da aplicação.

Ao executar uma classe de teste com métodos marcados com a anotação @Test, o JUnit cria uma instância da classe de teste e chama cada método de teste separadamente, capturando os resultados e relatando quaisquer erros ou falhas.

A anotação @Test é uma das muitas anotações fornecidas pelo JUnit para ajudar a escrever testes de unidade e integração eficazes em projetos Java. Outras anotações importantes do JUnit incluem @Before, @After, @BeforeClass, @AfterClass e @Ignore.

Detalhes sobre a classe assertions:

O AssertJ é um framework que ajuda a escrever asserções (ou afirmações) mais expressivas e fáceis de ler. As asserções são declarações que verificam o comportamento esperado do código em teste. Por exemplo, se você está testando uma função que retorna o dobro de um número, você pode escrever uma asserção que verifica se o resultado da função é o dobro do número original.

Essa ferramenta fornece uma sintaxe de cadeia fluente para escrever asserções de forma clara e legível. Além disso, a classe Assertions fornece vários métodos úteis para asserções comuns, como **assertThat, isTrue, isFalse, isEqualTo, isNotEqualTo, isNotNull, isNull, hasSize** e muitos outros.

Criação do teste para verificar se um e-mail já está cadastrado no sistema:

```
@Test
public void deveRetornarFalseQuandoNaoHouverUsuarioCadastradoComOEmailInformado(){

    //Verifica a existencia de um email na base
    boolean result = usuarioRepository.existsByEmail("usuario@gmail.com");

    //Verifica se o resultado esperado (result = false) ocorreu
    Assertions.assertThat(result).isFalse();
}
```

Criação do teste para verificar o método “save” usuário:

```
@Test
public void devePersistirUmUsuarioNaBaseDeDados(){

    //Nova instancia de usuário
    Usuario usuario = Usuario.builder().nome("usuario").email("usuario@email.com").build();
```

```
    //Salva o usuario na base de dados e guarda sua resposta em usuarioResponse
    Usuario usuarioResposta = usuarioRepository.save(usuario);

    //Verifica se usuarioResponse é diferente de null
    Assertions.assertThat(usuarioResposta.getId()).isNotNull();

}
```

Criação do teste para verificar se o retorno é vazio quando um usuario inexistente é requisitado.

```
@Test
public void deveRetornarVazioAoBuscarUsuarioPorEmailQuandoNaoExisteNaBase(){

    //Procura um usuario pelo email
    Optional<Usuario> result = usuarioRepository.findByEmail("usuario@email.com");

    //Verifica o retorno contido na variavel 'result'
    Assertions.assertThat(result.isPresent()).isFalse();
}
```

Criação do teste para verificar o retorno da pesquisa por um usuário pode e-mail:

```
@Test
public void deveBuscarUmUsuarioPorEmail(){

    //Instancia e salva um usuário
    Usuario usuario = Usuario.builder().nome("usuario").email("usuario@email.com").build();
    usuarioRepository.save(usuario);

    //Busca pelo usuário salvo
    Optional<Usuario> result = usuarioRepository.findByEmail("usuario@email.com");

    //Verifica se há um objeto Usuario em result
    Assertions.assertThat(result.isPresent()).isTrue();
}
```

Classe UsuarioRepositoryTest após a implementação de todos os métodos:

```
package com.bcipriano.minhasfinancas.model.repository;

import com.bcipriano.minhasfinancas.model.entity.Usuario;
import org.assertj.core.api.Assertions;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.junit4.SpringRunner;

import java.util.Optional;

@SpringBootTest
@ActiveProfiles("test")
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
@RunWith(SpringRunner.class)
```

```java
public class UsuarioRepositoryTest {

    @Autowired
    UsuarioRepository usuarioRepository;

    @Test
    public void deveVerificarAExistenciaDeUmEmail(){

        //Cria um novo usuario
        Usuario usuario = Usuario.builder().nome("usuario").email("usuario@email.com").build();

        //Salva na base de dados
        usuarioRepository.save(usuario);

        //Procura pelo email salvo
        boolean result = usuarioRepository.existsByEmail("usuario@email.com");

        //Verifica se o resultado é verdadeiro
        Assertions.assertThat(result).isTrue();

    }

    @Test
    public void deveRetornarFalseQuandoNaoHouverUsuarioCadastradoComOEmailInformado(){

        //Verifica a existencia de um email na base
        boolean result = usuarioRepository.existsByEmail("usuario@gmail.com");

        //Verifica se o resultado esperado (result = false) ocorreu
        Assertions.assertThat(result).isFalse();

    }

    @Test
    public void devePersistirUmUsuarioNaBaseDeDados(){

        //Nova instancia de usuário
        Usuario usuario = Usuario.builder().nome("usuario").email("usuario@email.com").build();

        //Salva o usuario na base de dados e guarda sua resposta em usuarioResponse
        Usuario usuarioResposta = usuarioRepository.save(usuario);

        //Verifica se usuarioResponse é diferente de null
        Assertions.assertThat(usuarioResposta.getId()).isNotNull();

    }

    @Test
    public void deveRetornarVazioAoBuscarUsuarioPorEmailQuandoNaoExisteNaBase(){

        //Deleta todos os usuários
        usuarioRepository.deleteAll();

        //Procura um usuario pelo email
        Optional<Usuario> result = usuarioRepository.findByEmail("usuario@email.com");

        //Verifica o retorno contido na variavel 'result'
        Assertions.assertThat(result.isPresent()).isFalse();
```

```
    }

    @Test
    public void deveBuscarUmUsuarioPorEmail(){

        //Deleta todos os usuários
        usuarioRepository.deleteAll();

        //Instancia e salva um usuário
        Usuario usuario = Usuario.builder().nome("usuario").email("usuario@email.com").senha("senha").build();
        usuarioRepository.save(usuario);

        //Busca pelo usuário salvo
        Optional<Usuario> result = usuarioRepository.findByEmail("usuario@email.com");

        //Verifica se há um objeto Usuario em result
        Assertions.assertThat(result.isPresent()).isTrue();
    }

}
```

5º Testes de integração UsuarioService (Serviço)

Agora, testaremos os métodos fornecidos pela camada de serviços. Para isso, crie uma nova pasta no diretorio de testes com o nome de “service” e dentro dela adicione uma nova classe de nome “UsuarioServiceTest”. O código base dessa classe é:

```
package com.bcipriano.minhasfinancas.service;

import org.junit.runner.RunWith;
import org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.junit4.SpringRunner;

@SpringBootTest
@RunWith(SpringRunner.class)
@ActiveProfiles("test")
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
public class UsuarioServiceTest {

}
```

Primeiramente testaremos o método que validaEmail:

```
@Test(expected = Test.None.class) //Test.None.Class esperado que não retorne nenhum erro
public void deveValidadeEmail(){

    //Executa o método válidar e-mail
    usuarioService.validarEmail("email@email.com");

}
```

Feito isso, testaremos o código que valida e-mail quando o e-mail requisitado já se encontra na base de dados:

```
@Test(expected = RegraNegocioException.class)
public void deveLancarExceptionQuandoHouverEmailCadastrado(){

    //Salva um novo usuário
    Usuario usuarioTest = Usuario.builder().nome("name").email("email@email.com").senha("senha").build();
    usuarioRepository.save(usuarioTest);
```

```
    //Passa o mesmo email cadastrado para verificar o resultado da validação
    usuarioService.validarEmail("email@email.com");

}
```

A forma com que os testes estão sendo construídos não está errada, porém o SpringBoot está carregando todo o contexto da aplicação para executar cada um dos métodos. A partir de agora usaremos algumas ferramentas que permitem uma otimização desses testes. Veja como ficou a classe UsuarioServiceTest:

```
package com.bcipriano.minhasfinancas.service;

import com.bcipriano.minhasfinancas.exception.RegraNegocioException;
import com.bcipriano.minhasfinancas.model.repository.UsuarioRepository;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.Mockito;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.junit4.SpringRunner;



@ActiveProfiles("test")
@RunWith(SpringRunner.class)
public class UsuarioServiceTest {

    @SpyBean
    UsuarioServiceImpl usuarioService;

    @MockBean
    UsuarioRepository usuarioRepository;

    @Test(expected = Test.None.class) //Test.None.Class esperado que não retorne nenhum erro
    public void deveValidadeEmail(){

        Mockito.when(usuarioRepository.existsByEmail(Mockito.anyString())).thenReturn(false);

        //Executa o método válidar e-mail
        usuarioService.validarEmail("email@email.com");

    }

    @Test(expected = RegraNegocioException.class)
    public void deveLancarExceptionQuandoHouverEmailCadastrado(){

        //Salva um novo usuário
        Mockito.when(usuarioRepository.existsByEmail(Mockito.anyString())).thenReturn(true);

        //Passa o mesmo email cadastrado para verificar o resultado da validação
        usuarioService.validarEmail("email@email.com");

    }

}
```

Como os métodos da camada de Modelos já foram testados(camada de interação com banco de dados) não precisamos incluir seus métodos na classe de Service. Agora, faremos apenas os testes unitários da camada service, ou seja, somente os testes de regras de negócio. Para isso usaremos a ferramenta Mock do Spring. Mock é uma técnica comum em testes de software que permite simular o comportamento de um objeto real em um ambiente de teste. No Spring Framework, o Mockito é uma das bibliotecas de mocking mais usadas em testes unitários. O Mockito permite que você crie objetos simulados (ou mock objects) que se comportam como os objetos reais, mas que podem ser configurados para retornar valores predefinidos ou gerar exceções em determinadas situações.

Com o Mockito, você pode criar mocks de classes e interfaces, e usar esses mocks em seus testes para simular o comportamento de dependências externas da sua classe em teste. Isso permite que você isole sua classe em teste de outras classes ou serviços, tornando seus testes mais rápidos e mais confiáveis.

Por exemplo, imagine que você esteja testando uma classe de serviço que usa um repositório para salvar e buscar dados de um banco de dados. Em vez de usar um repositório real em seus testes, você pode criar um mock do repositório usando o Mockito e configurá-lo para retornar valores predefinidos quando for chamado. Isso permite que você controle o comportamento do repositório em seus testes, e se concentre na lógica da sua classe de serviço. Documentação: (https://site.mockito.org/)

Para facilitar a compreensão do Mock, considere a linha:
Mockito.when(usuarioRepository.findByEmail(email)).thenReturn(Optional.of(usuario))
Podemos ler da seguinte forma: Mockito, “finja” que o método findByEmail da classe “usuarioRepository” foi executado com o parametro “email” e que para essa situação ele retornou um objeto Optional de usuário.

Explicação das linhas refatoradas de cada um dos métodos:

```
@Test(expected = Test.None.class) //Test.None.Class esperado que não retorne nenhum erro
public void deveValidadeEmail(){

   Mockito.when(usuarioRepository.existsByEmail(Mockito.anyString())).thenReturn(false);

   //Executa o método válidar e-mail
   usuarioService.validarEmail("email@email.com");

}
```

Primeiro, é definido que esse teste não deve lançar nenhuma exceção com a anotação @Test(expected = Test.None.class). Ou seja, espera-se que o método usuarioService.validarEmail não lance exceção ao ser executado.

Em seguida, a linha Mockito.when(usuarioRepository.existsByEmail(Mockito.anyString())).thenReturn(false); indica que, quando o método existsByEmail do repositório de usuários for chamado com qualquer parâmetro de entrada (indicado por Mockito.anyString()), ele deve retornar false.

Por fim, é invocado o método usuarioService.validarEmail passando um endereço de e-mail como parâmetro. Como foi definido que o método existsByEmail do repositório deve retornar false independentemente do parâmetro, então o serviço de usuário deve validar o e-mail corretamente e não lançar exceção.

```
@Test(expected = RegraNegocioException.class)
public void deveLancarExceptionQuandoHouverEmailCadastrado(){

   //Salva um novo usuário
   Mockito.when(usuarioRepository.existsByEmail(Mockito.anyString())).thenReturn(true);

   //Passa o mesmo email cadastrado para verificar o resultado da validação
   usuarioService.validarEmail("email@email.com");

}
```

A anotação @Test(expected = RegraNegocioException.class) indica que o teste deve lançar uma exceção do tipo RegraNegocioException, caso a validação de e-mail feita pelo serviço de usuário encontre um e-mail já cadastrado. Ou seja, espera-se que o método usuarioService.validarEmail lance uma exceção do tipo RegraNegocioException.

A linha Mockito.when(usuarioRepository.existsByEmail(Mockito.anyString())).thenReturn(true); indica que, quando o método existsByEmail do repositório de usuários for chamado com qualquer parâmetro de entrada, ele deve retornar true. Isso simula o comportamento do repositório quando o serviço de usuário tenta cadastrar um novo usuário com um e-mail que já está em uso.

Por fim, é invocado o método usuarioService.validarEmail passando o mesmo endereço de e-mail que foi simulado como já cadastrado no sistema. Como o repositório de usuários foi simulado para retornar true quando o serviço de usuário tenta verificar se o e-mail já está em uso, o método usuarioService.validarEmail deve lançar uma exceção do tipo RegraNegocioException.

Após a refatoração dos 2 métodos adicionados podemos adicionar também os demais métodos para o teste de toda a camada de service da aplicação. Veja:

```
@Test(expected = Test.None.class)
public void deveAutenticarUsuarioComSucesso(){

   //Strings que serão usadas para a instancimento e parametro para método autenticar
   String email = "email@email.com";
   String senha = "senha";

   //Cria uma nova instancia de usuário
   Usuario usuario = Usuario.builder().email(email).senha(senha).id(1l).build();

   //Simula uma pesquisa por email e obtem como resposta o Objeto Optional de usuairo
   Mockito.when(usuarioRepository.findByEmail(email)).thenReturn(Optional.of(usuario));

   //Verifica se o resultado da autenticação retornou um objeto usuario para o método autenticação
   Usuario result = usuarioService.autenticar(email, senha);

   //Verificar se result é diferente de null
   Assertions.assertThat(result).isNotNull();

}
```

Esse teste verifica se o método autenticar do serviço UsuarioService está retornando corretamente um objeto Usuario quando é passado um email e senha que existem na base de dados. Primeiramente, é criada uma instância de Usuario com um email e senha válidos, e um id qualquer. Em seguida, é simulada uma pesquisa por email na base de dados, utilizando o método Mockito.when(usuarioRepository.findByEmail(email)).thenReturn(Optional.of(usuario)). Esse código indica que, quando o método findByEmail for chamado com o email definido anteriormente, ele deve retornar um objeto Optional contendo o usuário criado anteriormente. Em seguida, é chamado o método autenticar com o mesmo email e senha utilizados na criação do

usuário. A resposta desse método é armazenada na variável result.
Por fim, é verificado se a variável result é diferente de null, o que indica que o método autenticar conseguiu autenticar o usuário corretamente. Se o valor de result for null, o teste falhará.

```
@Test void deveLancarErroQuandoNaoEncontrarUsuarioCadastradoPeloEmail(){

    //Simule que a execução do método findByEmail(qualquer string) retornou um objeto Optional vazio.
    Mockito.when(usuarioRepository.findByEmail(Mockito.anyString())).thenReturn(Optional.empty());

    //Assertion deve obter a mensagem de erro gerada apos a execução do método autenticar de usuarioService
    Throwable exception = Assertions.catchThrowable(() ->usuarioService.autenticar("email@email.com", "senha"));

    //Assertion deve comparar se mensagem capturada é igual a "Usuário não encontrado"
    Assertions.assertThat(exception).isInstanceOf(ErroAutenticacao.class).hasMessage("Usuário não encontrado!");
}
```

```
@Test
public void deveLancarErroQuandoSenhaNaoBater(){

    //Cria uma nova instancia de usuário
    Usuario usuario = Usuario.builder().email("email@email.com").senha("senha").build();

    //Simula a execução do método findByEmail('qualquer string') para um retorno de um Option de Usuario
    Mockito.when(usuarioRepository.findByEmail(Mockito.anyString())).thenReturn(Optional.of(usuario));

    //Pega a mensagem de excessão para um email e senha apos a execução do metodo autenticação com uma senha
diferente da que foi 'salva'
    Throwable exception = Assertions.catchThrowable(() ->usuarioService.autenticar("email@email.com",
"senhaDiferente"));

    //Verifica se a mensagem de erro retornada foi 'Senha inválida'
    Assertions.assertThat(exception).isInstanceOf(ErroAutenticacao.class).hasMessage("Senha inválida!");
}
```

```
@Test(expected = RegraNegocioException.class)
public void naoDeveSalvarUsuarioComEmailJaCadastrado(){

    //Inicializa uma variavel com a string e-mail que será usada mais de uma vez
    String email = "email@email.com";

    //Cria uma nova instancia de usuário
    Usuario usuario = Usuario.builder().email(email).build();

    //Moquito usado para configurar o comportamento do metodo validarEmail e lançar RegradNegocioException
    Mockito.doThrow(RegraNegocioException.class).when(usuarioService).validarEmail(email);

    //Salva o usuario
    usuarioService.salvarUsuario(usuario);

    //Mockito verifica que o método save do objeto usuarioRepository nunca foi chamado
    Mockito.verify(usuarioRepository, Mockito.never()).save(usuario);
}
```

```java
@Test(expected = Test.None.class)
public void deveSalvarUmUsuario(){

    //Indica que o metodo validarEmail de usuarioService não deve fazer nada durante esse teste
    Mockito.doNothing().when(usuarioService).validarEmail(Mockito.anyString());

    //Cria uma nova instancia de usuário
    Usuario usuario = Usuario.builder().id(1l).nome("nome").senha("senha").email("email@email.com").build();

    //Mockito simula a execução do usuario .save() para um objeto qualquer to tipo Usuario e retorna o objeto
    //Usuario instanciado anteriormente
    Mockito.when(usuarioRepository.save(Mockito.any(Usuario.class))).thenReturn(usuario);

    //Apos isso, uma instancia usuario armazena o retorno do método usuarioService(novo Usuario())
    Usuario usuarioSalvo = usuarioService.salvarUsuario(new Usuario());

    //Verifica se retorno é diferente de null
    Assertions.assertThat(usuarioSalvo).isNotNull();

    //Verifica se retorno tem id igual a 1
    Assertions.assertThat(usuarioSalvo.getId()).isEqualTo(1);

    //Verifica se retorno tem nome igual a "nome"
    Assertions.assertThat(usuarioSalvo.getNome()).isEqualTo("nome");

    //Verifica se retorno tem email igual a "email@email.com"
    Assertions.assertThat(usuarioSalvo.getEmail()).isEqualTo("email@email.com");

    //Verifica se retorno tem senha igual a "senha"
    Assertions.assertThat(usuarioSalvo.getSenha()).isEqualTo("senha");
}
```

Resultado final da classe UsuarioServiceTest:

```java
package com.bcipriano.minhasfinancas.service;

import com.bcipriano.minhasfinancas.exception.ErroAutenticacao;
import com.bcipriano.minhasfinancas.exception.RegraNegocioException;
import com.bcipriano.minhasfinancas.model.entity.Usuario;
import com.bcipriano.minhasfinancas.model.repository.UsuarioRepository;
import com.bcipriano.minhasfinancas.service.impl.UsuarioServiceImpl;
import org.assertj.core.api.Assertions;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.Mockito;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.boot.test.mock.mockito.SpyBean;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.junit4.SpringRunner;

import java.util.Optional;



@ActiveProfiles("test")
@RunWith(SpringRunner.class)
public class UsuarioServiceTest {

    @SpyBean
    UsuarioServiceImpl usuarioService;
```

```java
@MockBean
UsuarioRepository usuarioRepository;


@Test(expected = Test.None.class)
public void deveSalvarUmUsuario(){

    //Indica que o metodo validarEmail de usuarioService não deve fazer nada durante esse teste
    Mockito.doNothing().when(usuarioService).validarEmail(Mockito.anyString());

    //Cria uma nova instancia de usuário
    Usuario usuario = Usuario.builder().id(1l).nome("nome").senha("senha").email("email@email.com").build();

    //Mockito simula a execução do usuario .save() para um objeto qualquer to tipo Usuario e retorna o objeto
    //Usuario instanciado anteriormente
    Mockito.when(usuarioRepository.save(Mockito.any(Usuario.class))).thenReturn(usuario);

    //Apos isso, uma instancia usuario armazena o retorno do método usuarioService(novo Usuario())
    Usuario usuarioSalvo = usuarioService.salvarUsuario(new Usuario());

    //Verifica se retorno é diferente de null
    Assertions.assertThat(usuarioSalvo).isNotNull();

    //Verifica se retorno tem id igual a 1
    Assertions.assertThat(usuarioSalvo.getId()).isEqualTo(1);

    //Verifica se retorno tem nome igual a "nome"
    Assertions.assertThat(usuarioSalvo.getNome()).isEqualTo("nome");

    //Verifica se retorno tem email igual a "email@email.com"
    Assertions.assertThat(usuarioSalvo.getEmail()).isEqualTo("email@email.com");

    //Verifica se retorno tem senha igual a "senha"
    Assertions.assertThat(usuarioSalvo.getSenha()).isEqualTo("senha");
}

@Test(expected = RegraNegocioException.class)
public void naoDeveSalvarUsuarioComEmailJaCadastrado(){

    //Inicializa uma variavel com a string e-mail que será usada mais de uma vez
    String email = "email@email.com";

    //Cria uma nova instancia de usuário
    Usuario usuario = Usuario.builder().email(email).build();

    //Moquito usado para configurar o comportamento do metodo validarEmail e lançar RegradNegocioException
    Mockito.doThrow(RegraNegocioException.class).when(usuarioService).validarEmail(email);

    //Salva o usuario
    usuarioService.salvarUsuario(usuario);

    //Mockito verifica que o método save do objeto usuarioRepository nunca foi chamado
    Mockito.verify(usuarioRepository, Mockito.never()).save(usuario);
}

@Test(expected = Test.None.class)
public void deveAutenticarUsuarioComSucesso(){
```

```
        //Strings que serão usadas para a instancimento e parametro para método autenticar
        String email = "email@email.com";
        String senha = "senha";

        //Cria uma nova instancia de usuário
        Usuario usuario = Usuario.builder().email(email).senha(senha).id(1l).build();

        //Simula uma pesquisa por email e obtem como resposta o Objeto Optional de usuairo
        Mockito.when(usuarioRepository.findByEmail(email)).thenReturn(Optional.of(usuario));

        //Verifica se o resultado da autenticação retornou um objeto usuario para o método autenticação
        Usuario result = usuarioService.autenticar(email, senha);

        //Verificar se result é diferente de null
        Assertions.assertThat(result).isNotNull();

    }

    @Test
    public void deveLancarErroQuandoSenhaNaoBater(){

        //Cria uma nova instancia de usuário
        Usuario usuario = Usuario.builder().email("email@email.com").senha("senha").build();

        //Simula a execução do método findByEmail('qualquer string') para um retorno de um Option de Usuario
        Mockito.when(usuarioRepository.findByEmail(Mockito.anyString())).thenReturn(Optional.of(usuario));

        //Pega a mensagem de excessão para um email e senha apos a execução do metodo autenticação com uma senha
diferente da que foi 'salva'
        Throwable exception = Assertions.catchThrowable(() ->usuarioService.autenticar("email@email.com",
"senhaDiferente"));

        //Verifica se a mensagem de erro retornada foi 'Senha inválida'
        Assertions.assertThat(exception).isInstanceOf(ErroAutenticacao.class).hasMessage("Senha inválida!");
    }

    @Test
    public void deveLancarErroQuandoNaoEncontrarUsuarioCadastradoPeloEmail(){

        //Simule que a execução do método findByEmail(qualquer string) retornou um objeto Optional vazio.
        Mockito.when(usuarioRepository.findByEmail(Mockito.anyString())).thenReturn(Optional.empty());

        //Assertion deve obter a mensagem de erro gerada apos a execução do método autenticar de usuarioService
        Throwable exception = Assertions.catchThrowable(() ->usuarioService.autenticar("email@email.com", "senha"));

        //Assertion deve comparar se mensagem capturada é igual a "Usuário não encontrado"
        Assertions.assertThat(exception).isInstanceOf(ErroAutenticacao.class).hasMessage("Usuario não encontrado!");
    }

    @Test(expected = Test.None.class) //Test.None.Class esperado que não retorne nenhum erro
    public void deveValidarEmail(){

        Mockito.when(usuarioRepository.existsByEmail(Mockito.anyString())).thenReturn(false);

        //Executa o método válidar e-mail
        usuarioService.validarEmail("email@email.com");

    }
```

```
    @Test(expected = RegraNegocioException.class)
    public void deveLancarExceptionQuandoHouverEmailCadastrado(){

        //Salva um novo usuário
        Mockito.when(usuarioRepository.existsByEmail(Mockito.anyString())).thenReturn(true);

        //Passa o mesmo email cadastrado para verificar o resultado da validação
        usuarioService.validarEmail("email@email.com");

    }

}
```