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.

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:

```
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
```

```java
@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.

```java
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”.

```java
private UsuarioRepository usuarioRepository;

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

Temos portanto:

```java
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 usuario, 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 persistencia no banco de dados seja atomica, 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;
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.");
        }
    }
}
```