**Desafio 01 - GIT:**

**I –** Quando você está trabalhando em uma branch e outra pessoa também e vocês querem unificar essas modificações antes de mesclar com a branch main. E quando você termina de trabalhar em uma branch e necessita mesclar suas modificações com a main.

**II –** O commit é utilizado para criar um status do projeto (nele, inclusive, é possível inserir um comentário que facilita a identidicação das alterações feitas). O Commit é feito antes do comando push, que “joga” as alterações pra nuvem.

**III –**

**git add -all:** adiciona os arquivos não apenas onde a branch possui um arquivo correspondente, mas também onde o índice já possui uma entrada.

**git add -u:** adiciona arquivos onde já existe uma entrada anterior, atualizando-os.

**git add -A:** adiciona todos os arquivos de todas as branches, caso não tenha sido passado um caminho específico.

**git add .** : adiciona todos os arquivos novos ou modificados.

**IV –** git init / git status / git add . / git commit -m “mensagem” / git push -u origin main.

**V –** Sim.

**Desafio 02 – POO e Entity:**

**I –** Na camada entity ficam as características dos objetos. É nessa camada que esses objetos são criados e configurados seus getters / setters / toString, etc.

**II –** Pois a POO é um paradigma que possibilita criar camadas no código. Essas camadas são propícias para códigos bastantes seguros e com manutenção facilitadas. Além disso, existe essa possibilidade pela linguagem escolhida, que no nosso caso fora o Java.

**III –** O ideal é não modificar as propriedades dos objetos na camada da entidade, mas sim na função que for utiliza-las. O set modifica um valor, o get pega o valor modificado.

**IV –** Podemos sim criar uma classe com um construtor vazio. A função do construtor é criar o objeto. No cosntrutor se passa quais parâmetros serão utilizados para tal criação.

**V –** Sim.

**Desafio 03 – HTTP, Rest e Controller:**

**I –** A camada Controller determina o fluxo da apresentação servindo como uma camada intermediária entre a camada de apresentação e a lógica. Controla e mapeia as ações e é responsável por retornar os códigos HTTP.

**II –** Relativa facilidade de integração, uso de padrões difundidos como o HTTP, que é a implementação mais comum de REST, Escalabilidade.

**III –**

Create = PUT / POST

Read = GET

Update = PUT

Delete = DELETE

**IV –** @PathVariable é utilizado quando o valor da variável é passada diretamente na URL, mas não como um parametro que você passa após o sinal de interrogação (?) mas sim quando o valor faz parte da url. Ao utilizar o @RequestParam, por outro lado, podemos ter vários parametros passados por url mas que não são parte da url em sí. Por fim, o @RequestBody, ou corpo da requisição, é onde geralmente enviamos dados que queremos gravar no servidor. Não é muito utilizado em requisições do tipo GET, mas sim nas do tipo POST e PUT. É no corpo da requisição onde você envia dados de um formulário de cadastro em seu site, por exemplo

**V –** A Response Entity dá o retorno da solicitação. Apresenta o Status da requisição, como os códigos HTTP (200 – ok / 404 not found e outros).

**VI –** Sim.

**Desafio 04 – JPA e Repository:**

**I –** Repository é um padrão de projeto que visa mediar a pesquisa de objetos de dominio entre a camada de dominio e a camada de mapeamento de dados usando uma interface semelhante à de uma coleção.

**II –** A injeção do Repository funciona com a anotação @AutoWired que, como o nome sugere, “amarra” a classe a ser injetada.

**III –** POJOS Persistentes: os objetos possuem design simples que não dependem da herança de interfaces ou classes de frameworks externos. Qualquer objeto com um construtor default pode ser feito persistente sem nenhuma alteração numa linha de código. // Consultas em Objetos: As consultas usam um esquema abstraído que é baseado no modelo de entidade como oposto às colunas na qual a entidade é armazenada. // Configurações simples: Anotações são simples de usar, convenientes para escrever e fácil de ler. Além disso, JPA oferece diversos valores defaults, portanto para já sair usando JPA é simples, bastando algumas anotações // Integração e Teste: permite que a JPA possa ser utilizada sem a existência de um servidor de aplicação. Dessa forma, testes unitários podem ser executados mais facilmente.

**IV –** Criar uma classe de configuração passando login, senha e porta de conexão; e injetar a dependência do banco de dados escolhido

**V –** Sim.

**Desafio 05 – Service:**

**I –** A camada Service é a camada que possui a lógica da aplicação. Ela é a responsável pelas regras de negócios, persistência com o banco de dados e as classes de entidades. A service recebe as requisições vindas do controller e gera respostas a partir destas requisições. É na service também que as operações de CRUD devem ser realizadas. Esta camada na verdade é responsável pelo motivo que a aplicação foi construída, ou seja, ela é o núcleo da aplicação.

**II –** Pois faz parte do princípio SOLID - Single Responsibility Principle (Princípio da Responsabilidade Única) “Nunca deve haver mais de um motivo para uma classe mudar.”

**III** – Métodos void não tem retorno e o método tipado é aquele que retorna o valor com o qual o método trabalha.

**IV –** Services podem e devem interagir com outros services. Um serviço deve ser escrito de tal forma que não se importe se o usuário o está chamando, um serviço na 'mesma camada' ou esteja sendo combinado em um serviço composto.

**V –** Sim.

**Desafio 06 – DTO:**

**I –** A DTO é responsável por mostrar para o usuário somente as informações que o programador escolhe, assim sendo, amplia a segurança do código, além de deixa-lo mais funcional.

**II –** Os construtores sempre tem que ter o mesmo nome da Classe, inclusive com as maiusculas, e nunca um tipo de retorno. Eles são inicializados quando a classe é instanciada. Já os getter and setters servem para o encapsulamento, garantindo a integridade das informações.

**III –** Com ele é possível mapear modelos complexos, com nenhuma ou poucas configurações – sempre seguindo convenções utilizando o Framework ModelMapper. Sua implementação é um tanto intuitiva e pouca pesquisa na internet já é suficiente para entendermos a aplicarmos seu uso.

**IV –** Sim.

**Desafio 07 – Manipulação de Dados:**

**I –** Existem várias formas de se proceder validação sem utilizar o @Valid. Uma delas é justamente criar uma classes de ENUM, onde você passa uma lista de valores que o programa irá aceitar receber e que se receber algum valor fora desse padrão, o programa não aceita.

**II –** O ENUM serve para validar dados que o programa espera receber. Como explicado acima, é criada uma classe que passa valores que o programa aceita ser recebido. Por exemplo:

public enum TipoLancamento {

RECEITA, DESPESA, receita, despesa, Receita, Despesa

}

**III –**

1 –Adicionar a dependencia do spring boot starter no Maven;

2 – Criar uma anotação customizada;

3 – Criar a lógica de validação (com if / else, por exemplo);

4 – Adicionar a anotação criada na classe a ser validada;

5 – Por fim, adicionar o exceptionHandler para tratar a exceção e lançar a mensagem.

**IV –** O método stream é uma forma de iterar sobre coleções, que é mais simples, concisa e ainda mais legível. Além disso, ela não termina com o laço. O método forEach é uma operação final, ou terminal, pois depois dela nada mais pode ser feito.

**V –** Sim.

**Desafio 08 – Tratamento de Exceptions:**

**I –** Durante o processamento de um programa, existem exceções que são esperadas e outras que podem surpreender. Trata-las faz com que o programa contorne-as e não pare de funcionar. Ainda, se configurado de tal modo, os tratamentos podem avisar ao usuário e ao próprio desenvolvedor qual foi a exceção que ocorreu.

**II –** Quando ocorre um erro ou uma exceção, o console manda uma mensagem avisando o que ocorreu.

**III –** Adicionando o “throws exception” com a exceção esperada/escolhida.

**IV –** É interessante criarmos um pacote separado para as Exceções e, nele, criar as classes para cada exceção esperada, tratando da forma devida.

**V –** Sim.

**Desafio 09 – Patterns, SOLID e Arquitetura:**

**I –** O Singleton é um padrão de projeto criacional, que garante que apenas um objeto desse tipo exista e forneça um único ponto de acesso a ele para qualquer outro código

**II –** Dependency Inversion Principle ou Princípio da Inversão de Dependência-DIP - “Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações; Abstrações não devem depender de detalhes.” E é dessa abstração que a camada Repository é responsável.

**III –** Garantir consistência e segurança na aplicação, pois com as camadas é possível manipular os dados de forma individualizada.

**IV –** Acredito que cada arquitetura tem vantagens e desvantagens, pois todas são utilizadas em aplicações mundo afora. Porém, ao se tratar de segurança, a arquitetura que está mais em voga é a Hexagonal.

**V –** Sim.

**Desafio 10 – Spring:**

**I –** Java é a linguagem de programação, já o SPRING é um framework que utiliza a linguagem Java para agilizar e/ou automatizar algumas funcionalidades de desenvolvimento.

**II –** Não, pois você terá que escolher com qual framework de persistência ele vai se relacionar. No projeto, por exemplo, utilizamos o spring data jpa.

**III –** É possível sim, trazendo as dependências no POM.xml e realizando as devidas anotações onde esses frameworks serão utilizados dentro do código.

**IV –** Sim.

**Desafio 11 – Libs:**

**I –** Junit – integração com o front end // modelMapper – mapeamento de entidades // hibernate – manipulação de BD // lombok – criação de atributos automatizado // swagger – BD de testes.

**II –** As libs, digamos, têm as "regras" e os "métodos" padronizados. Drivers retornam dados consultados em um formato relacional e fornecem conectividade entre banco de dados por meio das interfaces de programa aplicativo JDBC disponíveis na plataforma Java.

**III –** Não.

**IV –** Porque o gerenciador concentra as dependências em um só lugar, onde podemos verificar sua versão, por exemplo, e se torna fácil a inserção a extinção de uma dependência específica.

**V –** Sim.

**Desafio 12 – Testes Unitários:**

**I –** Porque a metodologia DevOps compartilha a responsabilidade da segurança da aplicação. Sendo assim, a parte de desenvolvimento e testes é imprescindível para a garantia de que o software está em bom estado de funcionamento antes de subir para o uso para o qual fora projetado.

**II –** O Mockito é um framework de Test and Spy e o seu principal objetivo é simular a instancia de classes e comportamento de métodos. Ao mockar uma dependência com o mockito, eu faço com que a classe que vai ser testada simule o método testado e suas dependências. Ex:

@Test

public void verificaValoresFiltradosStatusOK() throws Exception{

Lancamentos lancamentos = new Lancamentos();

lancamentos.setDataLancamento(lancamentos.getDataLancamento());

lancamentos.setAmount(lancamentos.getAmount());

lancamentos.setPaid(lancamentos.getPaid());

when(lancamentosService.getLancamentosDependentes(lancamentos.getDataLancamento(), lancamentos.getAmount(), lancamentos.getPaid())).thenReturn(Collections.singletonList(lancamentos));

mockMvc.perform(MockMvcRequestBuilders.get("/api/lancamentos/filter")

.contentType(MediaType.APPLICATION\_JSON))

.andExpect(status().isOk());

}

}

**III –** O Sonar é uma plataforma de código aberto desenvolvida pela SonarSource para inspeção contínua da qualidade do código para realizar revisões automáticas com análise estática de código para detectar bugs, códigos cheirosos e vulnerabilidades de segurança. Ele oferece relatórios sobre código duplicado, padrões de codificação, testes de unidade, cobertura de código, complexidade de código, comentários, bugs e vulnerabilidades de segurança.

**IV –** Sim.

**Desafio 13 – Integração:**

**I –** O processo de integração foi bem tranquilo da minha parte, pois o front já estava bem adequado ao meu projeto de back. Tive unicamente que baixar o NodeJs abrindo um chamado no click e modificar o caminho HTTP do server para alinhar com a URI.

**II –** A principal dificuldade foi entender como a mágica acontece quando se dá os comandos para subir a aplicação (npm e ng). Também tive um pouco de dificuldade em relação à versão do Angular que não deixava eu instalar corretamente e, consequentemente, não permitia subir a aplicação de front. Precisei ver video tutorial na internet para aprender a instalar de maneira forçada a versão correta. Depois disso, funcionou corretamente.

**III –** Não. Eu tinha mesmo era um erro na função CREATE, pois chamava create() e não funcionava. Quando troquei para .save() funcionou tanto to Postman, quanto no Front.

**IV –** O CORS é uma especificação de segurança de navegadores, assim é o seu navegador em que você esta desenvolvendo sua aplicação que esta barrando suas requisições. Isso ocorre pois os navegadores usam a técnica de preflight, uma requisição de verificação que seu browser faz ao recurso que você deseja consumir e verifica se sua requisição para consumi-lo está de acordo com as políticas de CORS definidas no back-end.

**V –** Sim.