

CRIANDO API REST COM SPRING DO INÍCIO AO FIM

INTRODUÇÃO

A proposta para a criação desse conteúdo surgiu através de um desafio, como parte de avaliação para um programa de treinamento. Meu desafio é produzir esse conteúdo descrevendo detalhadamente quais os passos para implementar uma **API** usando **SPRING** dado um contexto apresentado ao longo desse texto. Confesso que é um desafio interessante e no desenvolver deste texto irei detalhar todos os passos que segui, desde a concepção da ideia, a modelagem do problema a ideia de solução e finalmente a implementação.

Para melhor entendimento segue o contexto ao qual foi inserido para o desafio:

“Você está fazendo uma API REST que precisa suportar o processo de abertura de nova conta no banco. O primeiro passo desse fluxo é cadastrar os dados pessoais de uma pessoa. Precisamos de apenas algumas informações obrigatórias:

- Nome
- E-mail
- CPF
- Data de nascimento

Caso os dados estejam corretos, é necessário gravar essas informações no banco de dados relacional e retornar o status adequado para a aplicação cliente, que pode ser uma página web ou um aplicativo mobile.

Você deve fazer com que sua API devolva a resposta adequada para o caso de falha de validação.”

E o desafio proposto é utilizando esse contexto explicar ao longo desse texto, como eu farei para que possa desenvolver está aplicação atendendo a todos os requisitos incluindo a respostas para as perguntas:

- Quais tecnologias do mundo Spring você usaria?
- Quais classes seriam criadas nesse processo?
- Qual foi o seu processo de decisão para realizar a implementação?
- Qual o papel de cada tecnologia envolvida no projeto?

Com todas as informações “em mãos” nos próximos capítulos iremos abordar cada passo seguido até a conclusão do desafio proposto. A seguir iremos abordar sobre a análise da proposta e juntamente o levantamento de requisitos e seguir para a próxima etapa.

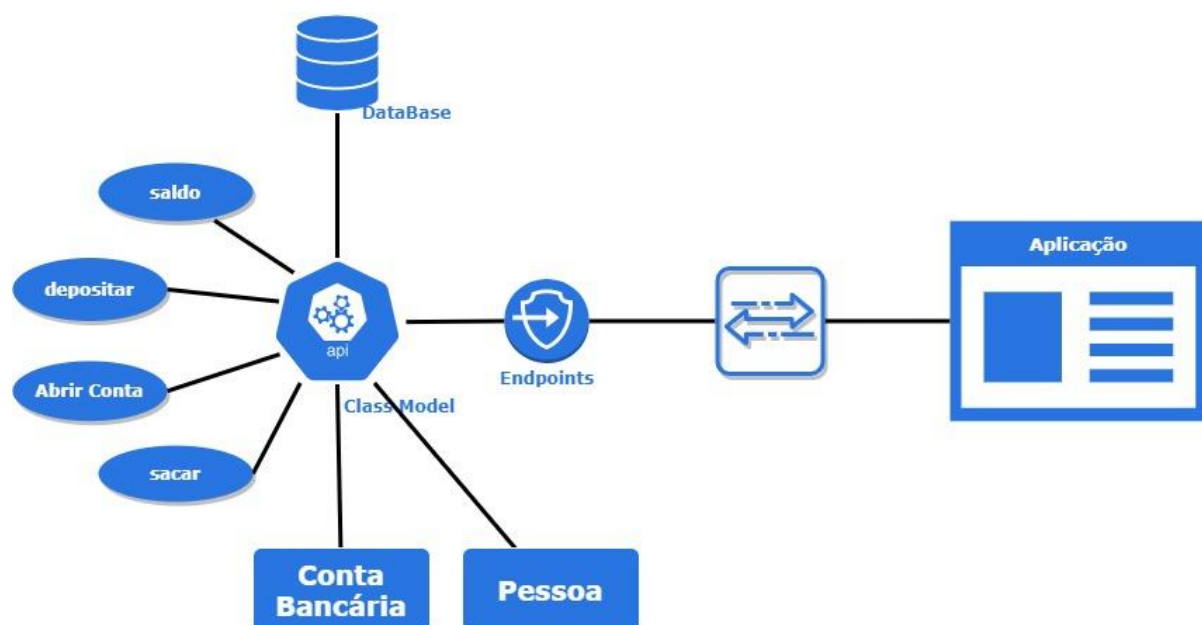
ANÁLISE LEVANTAMENTO DE REQUISITOS E MODELAGEM

Para a análise e levantamento de requisitos é necessário compreender o problema, e para isso existem diversas técnicas, nesse caso a leitura e compreensão do desafio proposto foi a maneira mais adequada de fazer esses dois procedimentos. Mas em outros casos pode se usar outras técnicas, principalmente quando a complexidade da aplicação é maior, nesse momento o texto apresentado é suficiente para extrair todas as informações necessárias.

Inicialmente identifiquei os verbos, posteriormente os substantivos e adjetivos e filtrando aqueles relevantes chegando ao seguinte cenário:

“Criar uma API REST, usando Spring + Hibernate, que suporta a abertura de nova conta no banco, e para isso são necessárias as seguintes informações, nome; e-mail; CPF e data de nascimento de forma obrigatória. Também é necessário validar todos os dados antes de salvar em um banco de dados relacional e devolver uma resposta adequada para aplicação cliente.”

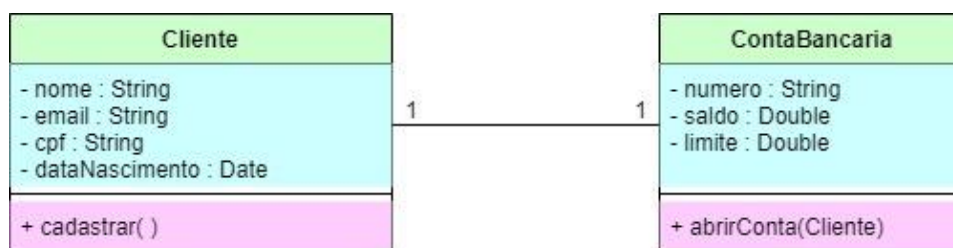
Nitidamente este não é um caso complexo, porém ao utilizar esses procedimentos ajuda no desenvolvimento e evita que possíveis erros possam acontecer por desatenção, ou por uma interpretação equivocada. Para facilitar ainda mais a compreensão do sistema proposto, foi elaborado um diagrama conceitual (Figura 1) tentando representar toda a lógica envolvida.



Agora temos uma representação visual de todo o problema, como apresentado no diagrama (Figura 1), podemos observar que existe uma aplicação que se comunica através dos endpoints com a API por meio de tráfego via HTTP e esses endpoints representam as funcionalidades que essa API executa, as quais são elas, saldo, depositar, sacar, e abrir conta que é a principal funcionalidade exigida para o desafio, as outras foram inseridas para melhor compreensão das funcionalidades que podem existir em uma API destinada a contas bancárias. Também é possível observar classes de modelo e banco de dados, as classes de modelo visam representar os objetos do mundo real dentro das diretrizes computacionais, e o banco de dados é onde será armazenada todas as informações transacionais da API.

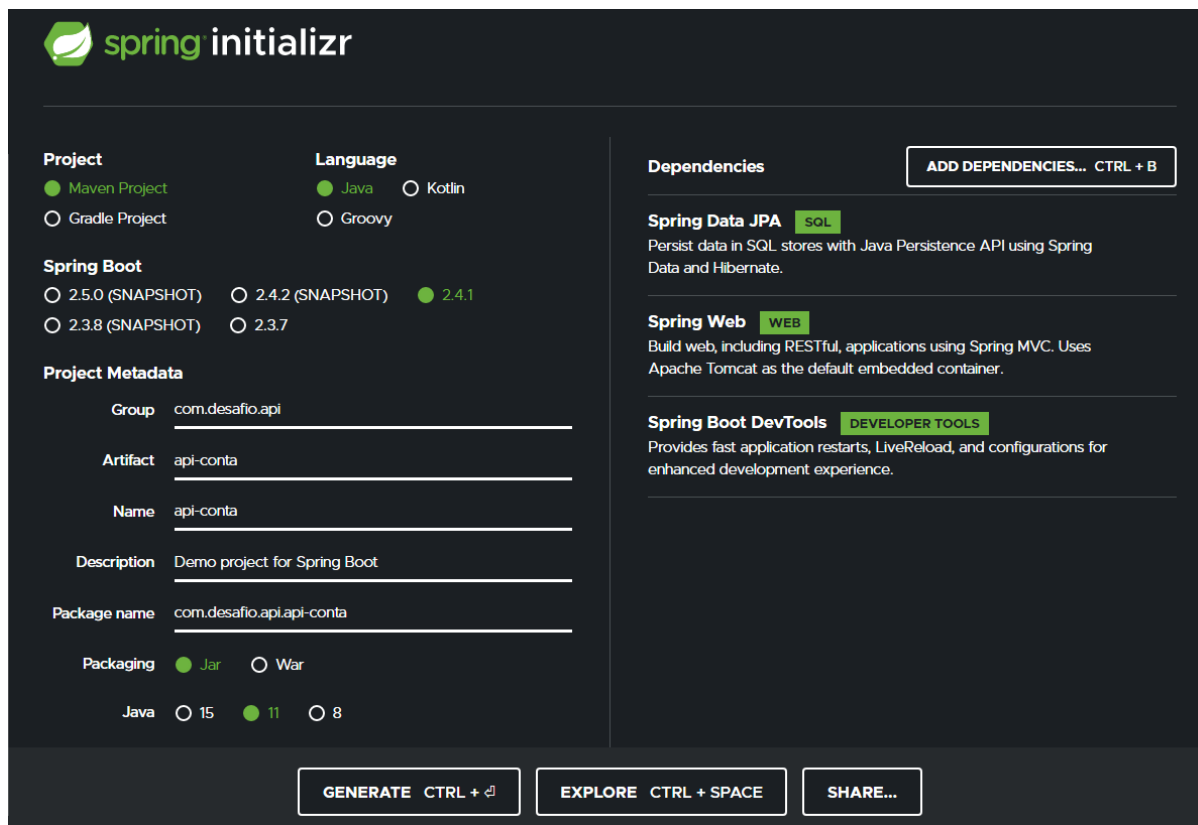
Estreitando mais a parte de levantamento de requisitos, podemos concluir que para a classe de modelo Pessoa, as informações necessárias para cada objeto são nome, e-mail, CPF e Data de nascimento e para a Conta Bancária, pegando um modelo popular de representação de contas bancárias, seria necessário que para cada conta tenha uma pessoa (cliente), um gerente, um número de conta e uma agência, além de saldo e um limite. Considerando que a aplicação inicialmente deve suportar abertura de nova conta no banco, o desenvolvimento da API ficará restringido a criação de endpoint para a funcionalidade de abrir conta, e limitando seus atributos, cliente; número da conta; saldo e limite, ficando as demais funcionalidades e atributos a serem desenvolvidos em uma publicação futura.

Para finalizar este capítulo segue a imagem do diagrama de classes elaborado para construção da solução, nesse diagrama (Figura 2) está representado as duas classes identificadas inicialmente, juntamente com as informações definidas e o relacionamento, que para este caso ficou com cardinalidade 1/1 (1 para 1).



DESENVOLVIMENTO

Os primeiros passos já foram dados, agora é preciso firmar o pé no chão e seguir adiante com o desenvolvimento da aplicação. Para dar início a essa etapa, irei desconsiderar o uso de qualquer IDE específica, e irei considerar a estrutura de um projeto Spring Boot utilizando Maven como gerenciador de dependências e as seguintes tecnologias spring: spring web, spring devtools e spring data jpa que irá instanciar toda a estrutura de projeto necessária para dar início ao desenvolvimento da API. Spring Web cria toda a estrutura para criação de aplicações web nos padrões RESTfull, MVC além de provisionar automaticamente o servidor Apache Tomcat, dentro dos padrões exigidos pela linguagem JAVA. Com Spring Devtools é possível desenvolver de forma mais cômoda, fornece reinicialização rápida da aplicação trazendo mais agilidade ao desenvolvimento, além de LiveReload e configurações aprimoradas. E por último, mas não menos importante, spring data jpa fornece toda a estrutura de persistência do JAVA utilizando inclusive o hibernate, baixando todas as dependências necessárias para implementar o padrão JPA.



The image shows the Spring Initializr web interface, which is used to generate a Spring Boot project. The interface is dark-themed and contains several sections for configuring the project.

Project

- ☒ Maven Project
- ☐ Gradle Project

Language

- ☒ Java
- ☐ Kotlin
- ☐ Groovy

Spring Boot

- ☐ 2.5.0 (SNAPSHOT)
- ☐ 2.4.2 (SNAPSHOT)
- ☒ 2.4.1
- ☐ 2.3.8 (SNAPSHOT)
- ☐ 2.3.7

Project Metadata

Group:

Artifact:

Name:

Description:

Package name:

Packaging: ☒ Jar ☐ War

Java: ☐ 15 ☒ 11 ☐ 8

Dependencies ADD DEPENDENCIES... CTRL + B

Spring Data JPA SQL
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

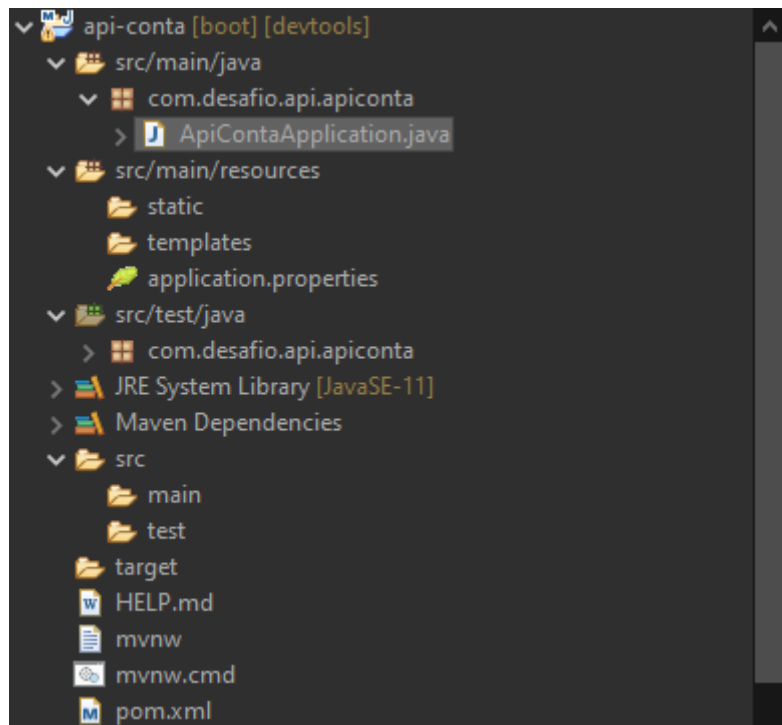
Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Boot DevTools DEVELOPER TOOLS
Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

Buttons:

- GENERATE CTRL + G
- EXPLORE CTRL + SPACE
- SHARE...

Para facilitar, a imagem anterior é possível reproduzir pelo site <http://start.spring.io> assim podendo qualquer um inicializar o projeto com a mesma estrutura e podendo replicar os passos em conformidade, ao completar todas informações apenas clique em “generate” e um download do projeto será baixado e pode ser aberto em uma IDE preferencial. Abaixo uma imagem da estrutura geral do projeto criado.



Dando continuidade ao desenvolvimento, nesse momento é necessário adicionar as dependências do banco de dados a ser utilizado, neste projeto será usado o H2 para teste, e o postgresql para ambiente de dev e produção. Para isso é necessário adicionar arquivos do tipo properties, um para cada ambiente. São eles, **application-test.properties**, **application-dev.properties**, **application-prod.properties**, além do arquivo padrão já incluso. Para cada um deles são necessárias algumas configurações, são elas:

Application.properties:

```
1 # Ativa o ambiente a ser utilizado
2 spring.profiles.active=test
3
4 # Desabilita o OSIV, um anti padrão ativado por default no spring
5 spring.jpa.open-in-view=false
```

Application-test.properties:

```
1 # Adicionar a URL do banco de dados para teste, o usuário e a senha
2 spring.datasource.url=jdbc:h2:mem:testdb
3 spring.datasource.username=sa
4 spring.datasource.password=
5
6 # Ativa um console do banco de dados H2 com uma URL dentro da aplicação
7 spring.h2.console.enabled=true
8 spring.h2.console.path=/h2-console
```

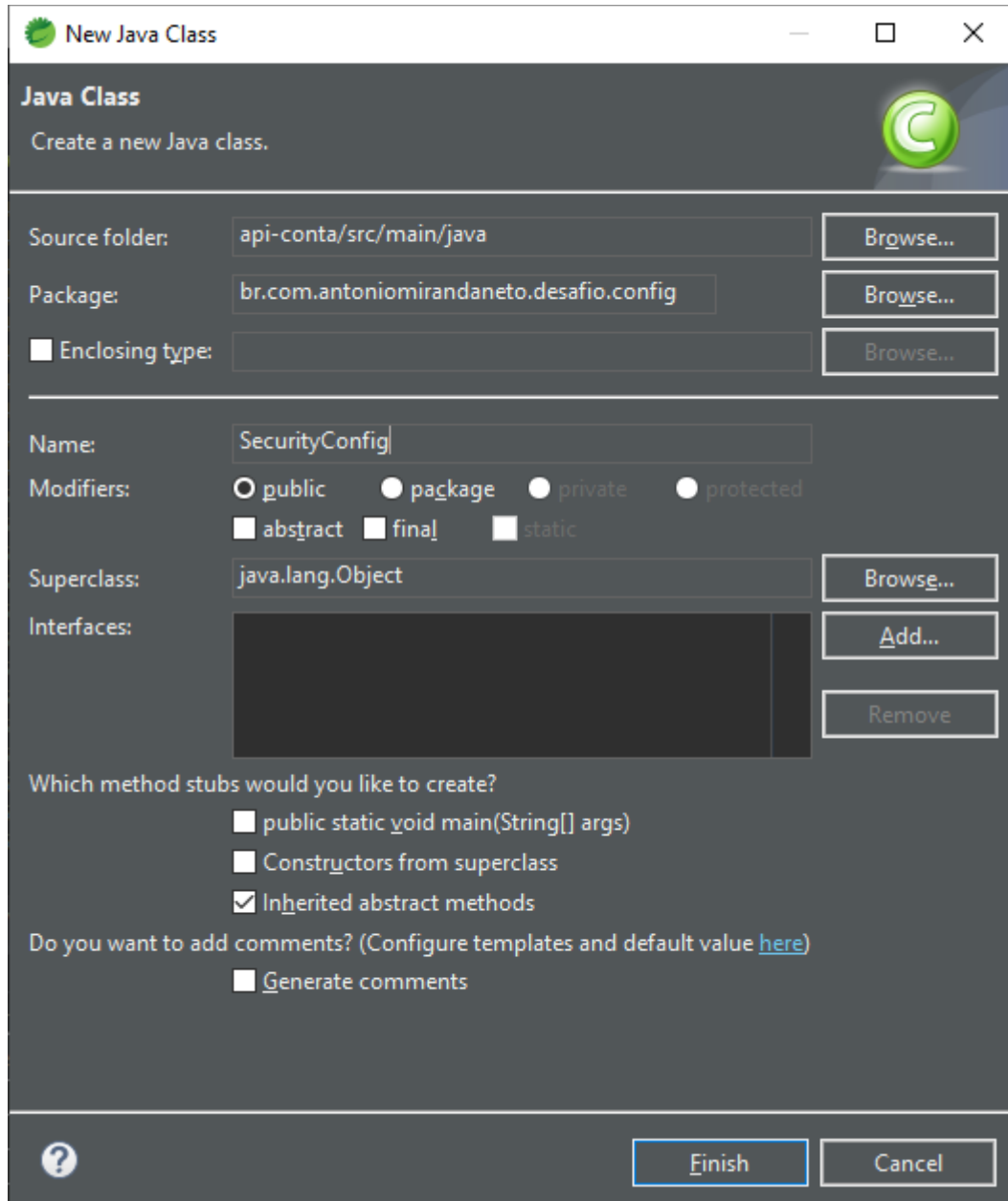
Application-dev.properties:

```
1 # As linhas comentadas devem ser descomentadas apenas para criar o modelo de relacional com base nas classes
2
3 #spring.jpa.properties.java.persistence.schema-generation.create-source=metadata
4 #spring.jpa.properties.java.persistence.schema-generation.scripts.action=create
5 #spring.jpa.properties.java.persistence.schema-generation.scripts.create-target=create.sql
6 #spring.jpa.properties.hibernate.hbm2ddl.delimiter=;
7
8 #configura a base de dados a ser utilizada em desenvolvimento em conformidade a base de dados de produção
9 spring.datasource.url=jdbc:postgresql://localhost:5432/desafio
10 spring.datasource.username=postgres
11 spring.datasource.password=1234567
12
13 spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation=true
14 spring.jpa.hibernate.ddl-auto=none
```

Application-prod.properties:

```
1 #URL da base de dados de produção
2 spring.datasource.url=${DATABASE_URL}
```

Para melhor segurança da aplicação adicionar as seguintes dependências ao projeto: spring-boot-starter-validation; spring-boot-starter-security. Após isso, cria no pacote padrão da aplicação uma nova class adicionando um subpacote chamado config e a classe com nome SecurityConfig como na imagem abaixo:



New Java Class

Java Class
Create a new Java class.

Source folder:

Package:

☐ Enclosing type:

Name:

Modifiers: ☒ public ☐ package ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass:

Interfaces:

Which method stubs would you like to create?

☐ public static void main(String[] args)

☐ Constructors from superclass

☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))

☐ Generate comments

Dentro dessa classe insira o seguinte código de configuração de segurança:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

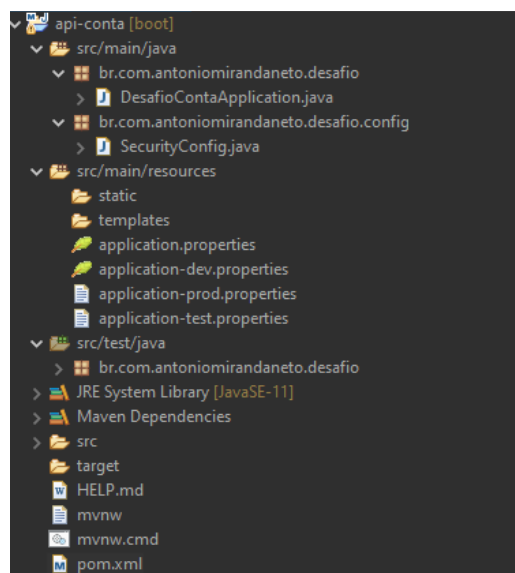
    @Autowired
    private Environment env;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        if (Arrays.asList(env.getActiveProfiles()).contains("test")) {
            http.headers().frameOptions().disable();
        }

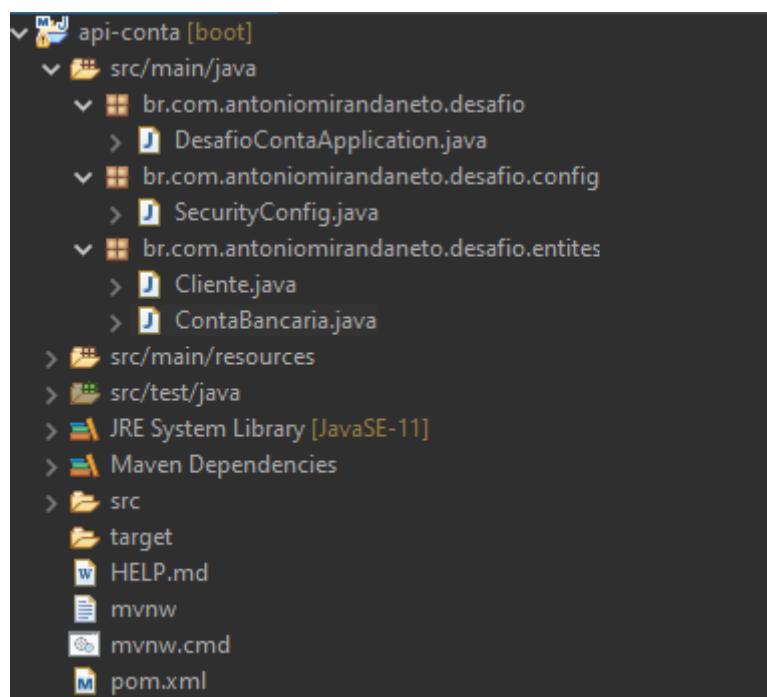
        http.cors().and().csrf().disable();
        http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
        http.authorizeRequests().anyRequest().permitAll();
    }

    @Bean
    CorsConfigurationSource corsConfigurationSource() {
        CorsConfiguration configuration = new CorsConfiguration().applyPermitDefaultValues();
        configuration.setAllowedMethods(Arrays.asList("POST", "GET", "PUT", "DELETE", "OPTIONS"));
        final UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration("/**", configuration);
        return source;
    }
}
```

De forma resumida, essa configuração habilita o acesso a outras aplicações em servidores externos, o que é útil para que aplicações front-end possam se comunicar com o back-end assim termina as configurações iniciais do projeto, dando seguimento adiante com a implementação dos modelos de domínio, Cliente e Conta Bancária. Abaixo uma imagem da estrutura de pastas e arquivos do projeto até o momento.



O próximo passo é criar uma classe para Cliente e outra para Conta Bancária, para isso crie um subpacote do pacote inicial chamado entites e dentro crie as classes Cliente e ContaBancaria, ficando a estrutura de projeto conforme imagem abaixo:



Para a criação de todas as classes de domínio, considerar a construção dos getters e setters que foram suprimidos na imagem para melhor visualização. Agora é necessário fazer o mapeamento objeto-relacional, criando os códigos dentro das classes. Para a classe cliente o código ficou assim:

```
@Entity // Anotação informa ao spring que esta classe será uma entidade a ser mapeada
@Table(name = "tb_cliente") // atribui um nome personalizado a tabela
public class Cliente implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id // informa que a variável abaixo da anotação é o ID
    @GeneratedValue(strategy = GenerationType.IDENTITY) // informa que o valor será gerado automaticamente
    private Long id;
    private String nome;
    private String email;

    @Column(unique=true) // Informa que só poderá existir um registro de cliente por CPF
    private String cpf;
    private Date dataNascimento;

    public Cliente() {

    }

    public void cadastrar(Long id, String nome, String email, String cpf, Date dataNascimento) {
        try {
            this.id = id;
            this.nome = nome;
            this.email = email;
            this.cpf = cpf;
            this.dataNascimento = dataNascimento;
        } catch (Exception e) {
            System.out.println("Erro: "+ e +" ao cadastrar cliente");
        }
    }
}
```

Para a classe Conta Bancária ficou assim:

```
@Entity // Anotação informa ao spring que esta classe será uma entidade a ser mapeada
@Table(name = "tb_conta_bancaria") // atribui um nome personalizado a tabela
public class ContaBancaria implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id // informa que a variável abaixo da anotação é o ID
    @GeneratedValue(strategy = GenerationType.IDENTITY) // informa que o valor será gerado automaticamente
    private Long id;

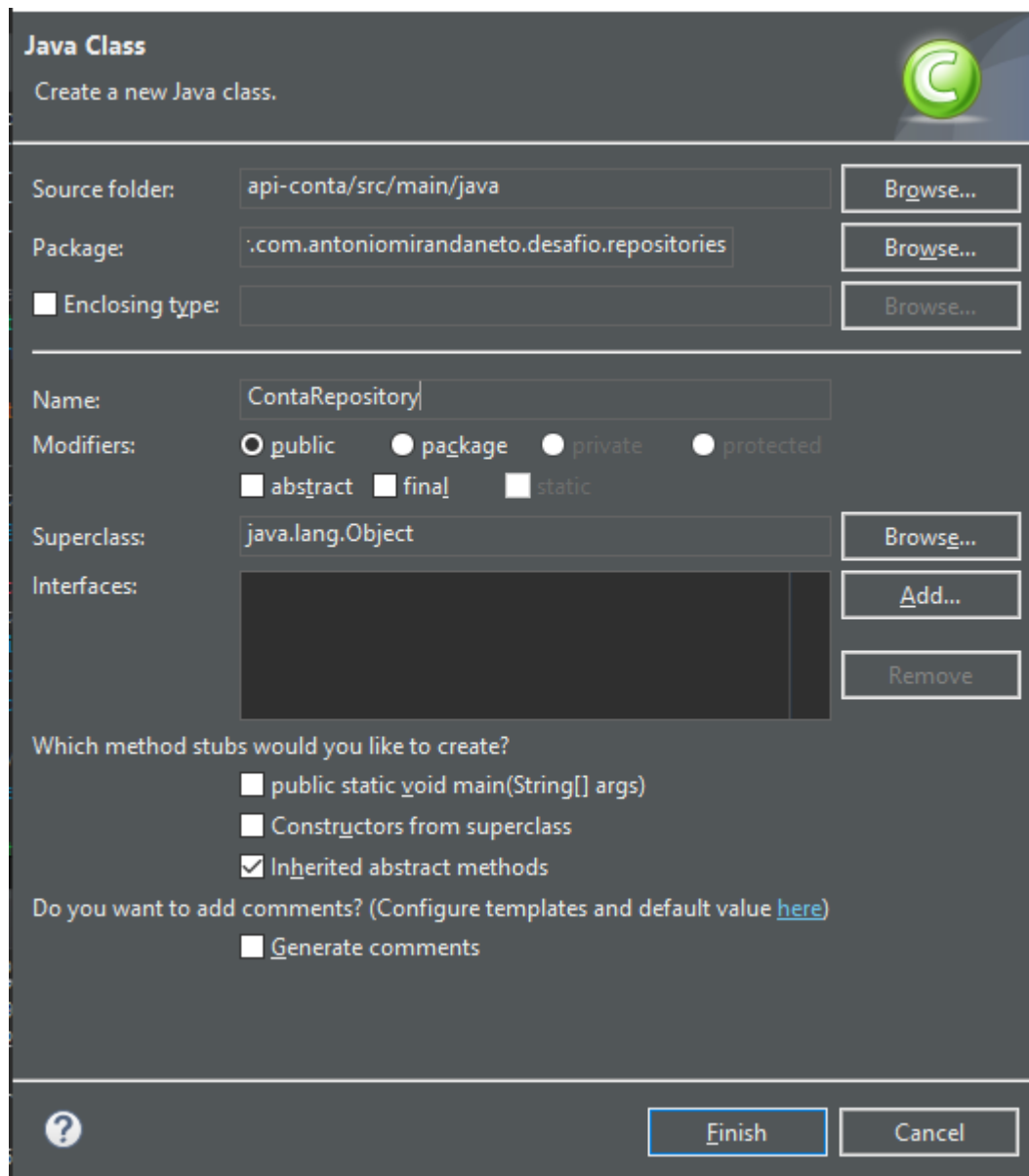
    @Column(unique=true) // Informa que só poderá existir um registro de conta com o numero definido
    private String numero;
    private Double saldo;
    private Double limite;

    @ManyToOne // Cria relacionamento 1 pra 1
    private Cliente cliente;

    private ContaBancaria() {
        this.saldo = 0.0;
        this.limite = 0.0;
    }

    public void abrirConta(Long id, String nome, String email, String cpf, Date dataNascimento) {
        Cliente cliente = new Cliente();
        cliente.cadastrar(id, nome, email, cpf, dataNascimento);
        this.cliente = cliente;
    }
}
```

Até este momento está finalizado o mapeamento entre classes de domínio e base de dados e chegando ao final do desenvolvimento da solução, mas antes é necessário criar os endpoints, nesse caso será criado o endpoint para abertura de nova conta e para isso é necessário criar um novo pacote chamado repositories, como na imagem abaixo:



Java Class
Create a new Java class.

Source folder:

Package:

☐ Enclosing type:

Name:

Modifiers: ☒ public ☐ package ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass:

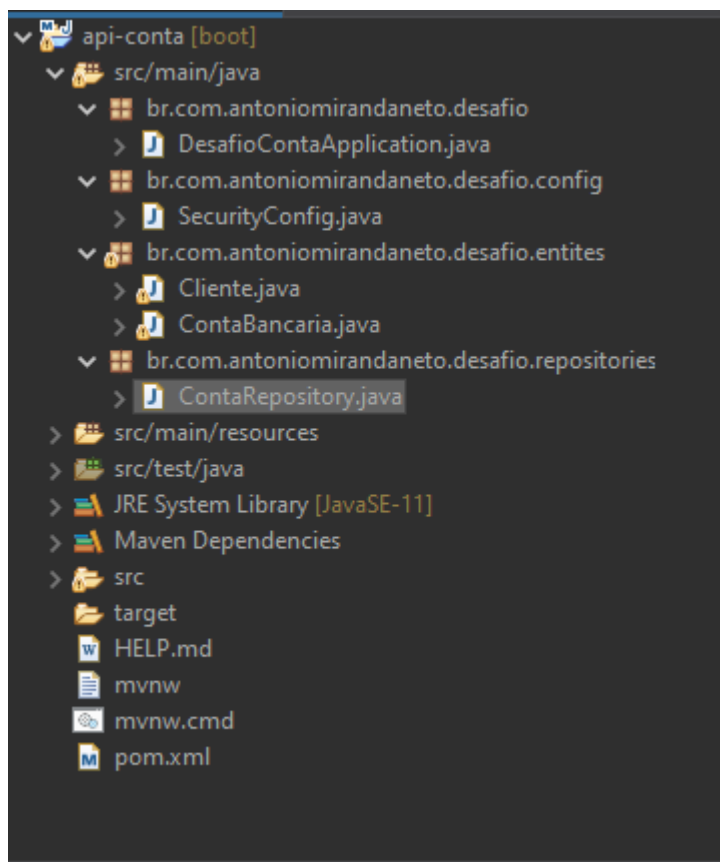
Interfaces:

Which method stubs would you like to create?

☐ public static void main(String[] args)
☐ Constructors from superclass
☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))
☐ Generate comments

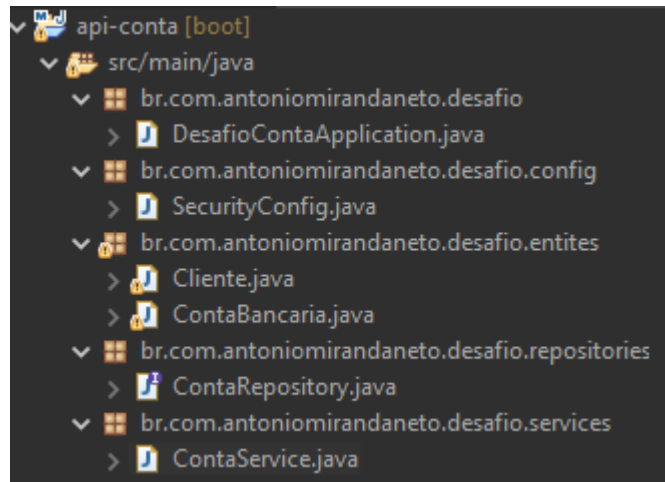
Após a criação a estrutura do projeto fica assim:



Feito isso é necessário fazer algumas alterações, como mudar de classe para interface e estender JpaRepository, pois esse implementará estruturas de manipulação de banco de dados, facilitando a persistência dos dados. Para isso, faça as seguintes alterações na classe ContaRepository conforme a imagem abaixo:

```
public interface ContaRepository extends JpaRepository<ContaBancaria, Long> {  
    }  
}
```

Agora crie um novo pacote chamado services conforme os outros já criados anteriormente e crie uma class chamada ContaService, e a estrutura do projeto deverá ficar assim:



Agora é necessário fazer uma anotação na classe ContaService criada para registra-la como componente do spring, para isso pode ser feita a anotação @Component ou @Service e também será necessário criar um pacote chamado dto e uma classe chamada ContaDTO, pois será usado o padrão Data Transfer Object, para que quando a resposta seja retornada ao cliente, já não exista mais conexão com a base de dados, também será necessário criar um pacote controllers e uma classe chamada ContaController com a seguinte anotação @RestController e o caminho para o mesmo, @RequestMapping(value = "/contas") e também uma injeção de dependências de ContaService. Para facilitar o entendimento segue imagem com os códigos para implementação de cada classe citada.

ContaService:

```
@Service
public class ContaService {

    @Autowired
    private ContaRepository contaRepository;

    @Autowired
    private ClienteRepository clienteRepository;

    @Transactional
    public ContaDTO insert(ContaDTO dto) {

        ContaBancaria conta = new ContaBancaria();
        conta.abrirConta(null, dto.getCliente().getNome(), dto.getCliente().getEmail(), dto.getCliente().getCpf(),
            dto.getCliente().getDataNascimento());
        clienteRepository.save(conta.getCliente());
        Random rand = new Random();
        Integer numero = rand.nextInt();
        conta.setNumero(numero.toString());
        conta = contaRepository.save(conta);
        return new ContaDTO(conta);
    }
}
```

ContaDTO:

```
public class ContaDTO implements Serializable {

    private static final long serialVersionUID = 1L;
    private Long id;
    private String numero;
    private Double saldo;
    private Double limite;
    private Cliente cliente;

    public ContaDTO() {

    }

    public ContaDTO(ContaBancaria entity) {
        id = entity.getId();
        numero = entity.getNumero();
        saldo = entity.getSaldo();
        limite = entity.getLimite();
        cliente = entity.getCliente();
    }
}
```

ContaController:

```
@RestController
@RequestMapping(value = "/contas")
public class ContaController {

    @Autowired
    ContaService service;

    @PostMapping
    public ResponseEntity<ContaDTO> insert(@RequestBody ContaDTO dto) {
        dto = service.insert(dto);
        URI uri = ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}").buildAndExpand(dto.getId()).toUri();
        return ResponseEntity.created(uri).body(dto);
    }
}
```

Também é necessário criar uma classe ClienteRepository e estender JpaRepository como na imagem:

```
public interface ClienteRepository extends JpaRepository<Cliente, Long>{

}
```

CONCLUSÃO

Diante de todo processo descrito no decorrer deste texto, pode ser abordada todas as etapas para a criação de uma API REST utilizando Spring e suas tecnologias. Alguns pontos não foram abordados, porém aproveito o espaço da conclusão para mencionar algumas regras que podem ajudar na validação dos dados. Como os dados do cliente como CPF e email, devem ser únicos, pois também caracteriza dados sensíveis, seria interessante validá-los pois não é interessante ter dois clientes com mesmo cpf ou email. Uma abordagem mais simples, seria introduzir uma constraint no banco de dados, informando cpf unique= true e para o email também, impossibilitando que exista dados duplicados e inconsistentes, outras abordagens também podem ser feitas para um melhor tratamento da mensagem de erro.