

Engenharia de Softwares III
Atividade prática

API Spring Boot

Beatriz Nataly Silva (SP3070263)

Carlos H. Biagolini Junior (SP3076113)

Gabriel S. Parreira (SP3072819)

São Paulo / 2022

Documentação versão 01

Índice

Descrição do projeto	3
Proposta.....	3
Referências.....	3
Git	3
SO de desenvolvimento	3
Linguagem	3
IDE.....	3
Software para consulta de APIs	3
Base do projeto	4
Abrir projeto na sua IDE.....	5
Eclipse.....	5
IntelliJ	7
VSC.....	8
Controle de versionamento	9
Definir entidade e configuração de BD H2.....	10
Criar endpoint	13
Tokens.....	15

Descrição do projeto

Proposta

Desenvolvimento de uma API para consultar dados de pedidos em uma loja hipotética.

Referências

Hack Reactor (2014) Bcrypt & Password Security - An Introduction. Disponível em <https://www.youtube.com/watch?v=O6cmuiTBZVs>. Data de acesso 26 de março de 2022.

Rodrigo Ferreira (2022) Curso de Spring Boot API Rest: Segurança da API, Cache e Monitoramento. Disponível em <https://cursos.alura.com.br/course/spring-boot-seguranca-cache-monitoramento>. Data de acesso 26 de março de 2022.

Git

Repositório: <https://github.com/biagolini/ifspES3Projeto>

Commit relacionado ao desenvolvimento de endpoint sem segurança: <https://github.com/biagolini/ifspES3Projeto/commit/2f3939d543a276b5af44fb5b19748158c267af45>.

Commit relacionado a versão com requisição de bearer token para acesso ao endpoint: <https://github.com/biagolini/ifspES3Projeto/commit/2eb10905e4344218a60089fda9aec4c3296136d4>.

SO de desenvolvimento

Edition Windows 11 Pro

Version 21H2

OS build 22000.556

Experience Windows Feature Experience Pack 1000.22000.556.0

Linguagem

java version "1.8.0_321"

Java(TM) SE Runtime Environment (build 1.8.0_321-b07)

Java HotSpot(TM) 64-Bit Server VM (build 25.321-b07, mixed mode)

IDE

IntelliJ IDEA 2021.3.1 (Community Edition).

Build #IC-213.6461.79, built on December 28, 2021

Runtime version: 11.0.13+7-b1751.21 amd64

VM: OpenJDK 64-Bit Server VM by JetBrains s.r.o.

Software para consulta de APIs

Postman for Windows Version 9.15.6 UI Version 9.15.6-ui-220325-0715

Base do projeto

Passo 1: Acesse o site <https://start.spring.io/>

Passo 2: Defina as configurações gerais do seu projeto.

Project: Maven Project

Language: Java

Spring Boot: 2.6.4

Group: com.es3

Artifact: vendas

Name: vendas

Description: Tutorial de servidor backend para recebimento informar dados de clientes

Package name: com.loja.vendas


Packaging: Jar

Java: 17

Passo 3: Defina as dependências.

- **Lombok:** Java annotation library which helps to reduce boilerplate code.
- **Spring Boot DevTools:** Provides fast application restarts, LiveReload, and configurations for enhanced development experience.
- **Spring Web:** Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
- **Spring Data JPA:** Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.
- **H2 Database:** Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.

Passo 4: Clique em GENERATE para gerar o projeto. Escolha uma pasta para salvar o arquivo .zip que será apresentado.

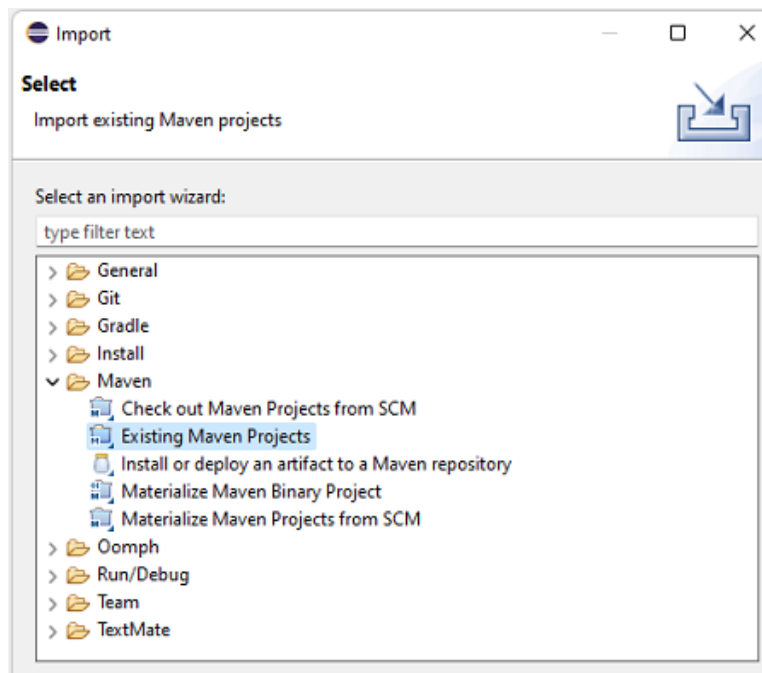


GENERATE CTRL + ↵

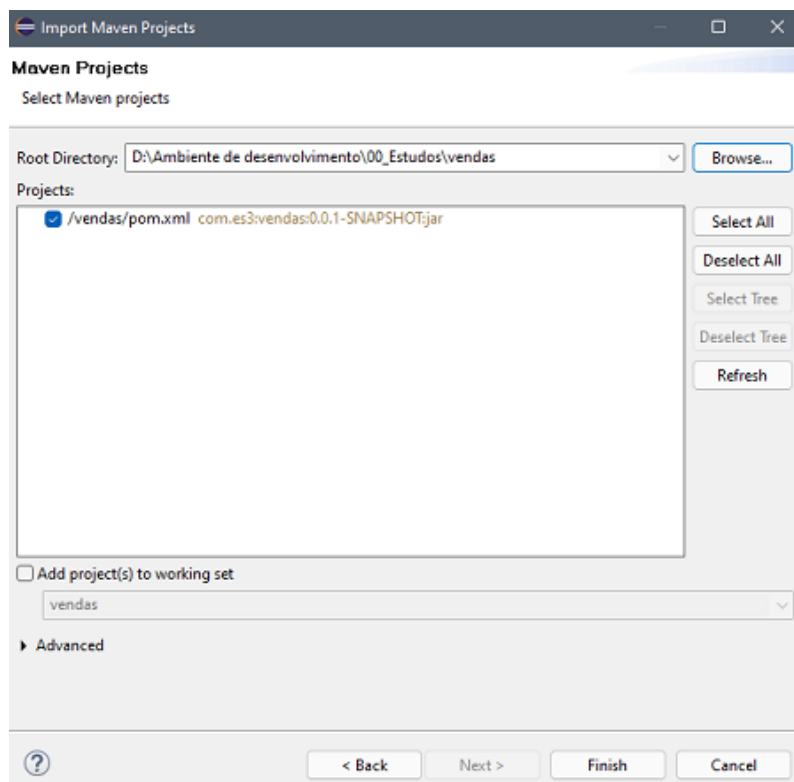
Abrir projeto na sua IDE

Eclipse

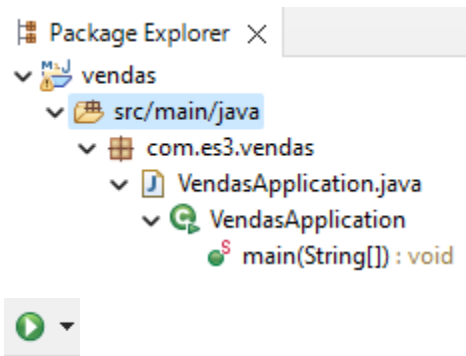
Passo 1: Feito download do arquivo do projeto (do site <https://start.spring.io/>). Extraia o arquivo, e no eclipse faça: File → Import.. → Macen → Existing Maven Projects → Environment Variable...



Passo 2: Na tela seguinte, clique em “Browse...” e selecione a pasta descompactada relacionada ao projeto. Selecionado o projeto clique em Finish.



Passo 3: Para rodar um teste do projeto, bastar rodar a classe main “src\main\java\com\vendas\src\main\java\com\es3\vendas\VendasApplication.java”. Clicando no ícone de Play da IDE.



Passo 4: Abra o endereço <http://localhost:8080/>, e confira o que está saindo. Saindo uma mensagem como a seguinte, quer dizer que está tudo certo. A mensagem de erro é porque ainda não configuramos nada para aparecer na pagina, mas o fato de aparecer essa mensagem de erro indica que está tudo certo, porque se o sistema não fosse carregado, essa mensagem não apareceria, e você teria um endereço invalidado.

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

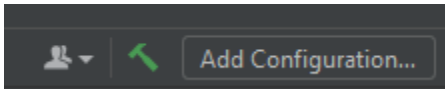
Mon Jan 17 19:19:13 BRT 2022

There was an unexpected error (type=Not Found, status=404).

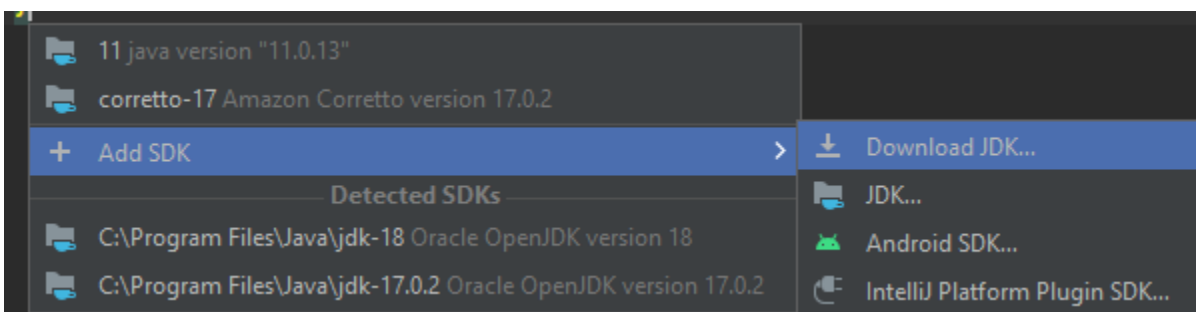
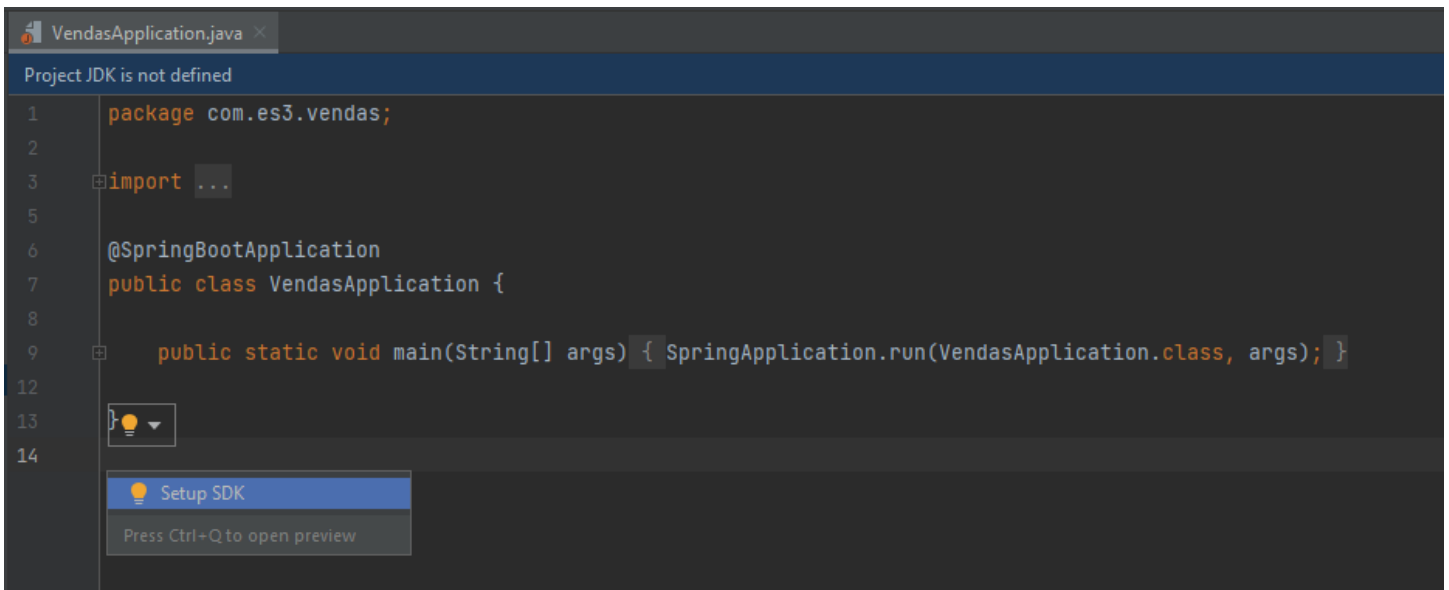
IntelliJ

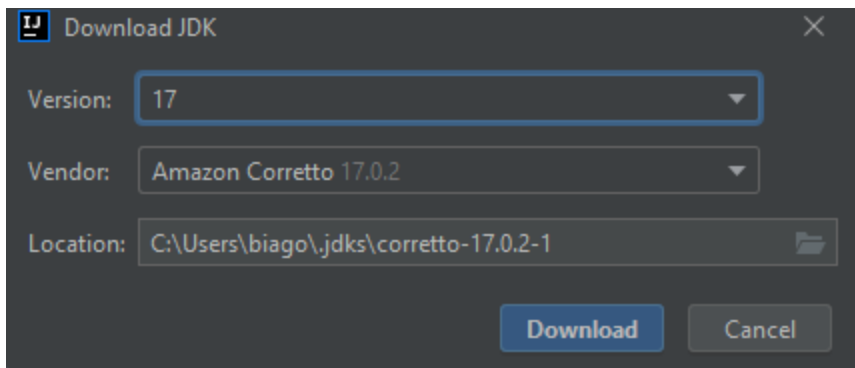
Passo 1: Feito download do arquivo do projeto (do site <https://start.spring.io/>). Extraia o arquivo para alguma pasta do seu computador. No IntelliJ faça: File → Open.. → Open File or Project → Selecione a pasta do seu projeto.

Passo 2: Feito Defina as configurações de execução, clicando em “Add Configuration”.



Passo 3: Clique no ícone +, em seguida vá em Maven.

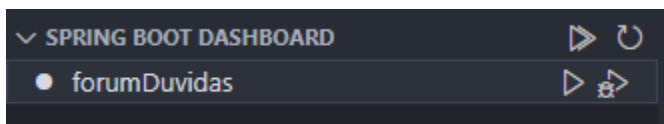




VSC

No Visual Studio code é mais simples. Basta abrir a pasta do projeto que você extraiu do arquivo .zip, e pronto.

Na hora de executar o projeto, basta selecionar o arquivo “src\main\java\com\ vendas\src\main\java\com\es3\ vendas\VendasApplication.java”, depois rodar o projeto no botão de play em:



Controle de versionamento

Afim de criar um backup do nosso projeto, e facilitar o desenvolvimento de atualizações futuras, recomendamos o controle de versionamento por meio do Git.

Passo 1: Acesse o link <https://github.com/new> e crie o projeto: ifspES3Projeto

Passo 2: Envie para o Github uma cópia do seu projeto.

```
git init
git config --local user.name "SEU NOME"
git config --local user.email "SEU_EMAIL@email.com"
git remote add origin https://github.com/USUARIO_GITHUB/ifspES3Projeto.git
git add .
git commit -m "Base"
git branch -M main
git push -u origin master
```

Definir entidade e configuração de BD H2

Passo 1: Definir sua classe de pedidos.

src/main/java/com/es3/vendas/entities/Pedidos.java

```
package com.es3.vendas.entities;

import lombok.Getter;
import lombok.Setter;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
@Getter
@Setter
public class Pedidos {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String email;
    private String descricao;
    private Double valor;
}
```

Passo 2: Configurar o acesso ao seu banco de dados. Em projetos com Spring Boot, quase todas as configurações ficam em um arquivo chamado `application.properties`. Se você olhar no diretório `"src/main/resources"`, você vai ver que quando aberto o projeto, o Spring initializer já estabeleceu o arquivo `application.properties`. Neste momento, se você abrir este documento, vai observar que o documento está vazio. Neste passo vamos configurar algumas linhas para configurar o H2 e o JPA. Você não precisa decorar nenhuma delas, basta seguir exemplos de implementação para seus projetos. Entenda o que cada seção desta configuração está definindo:

- **Datasource:** Configurações relacionadas ao acesso do seu BD. Aqui você passa o driver do seu banco de dados, URL, usuário e senha.
- **JPA:** Configurações específicas da JPA. Aqui você passa o dialeto do seu BD e informe se o Hibernate deve fazer ou não a atualização automática do BD.
- **H2:** Configurações específicas do BD H2. O `"console.enable"` indica a criação de uma interface para ser acessada no navegador. Já o `"path"` é o caminho para acessar a interface visual indicada anteriormente.

```
# data source
spring.datasource.url=jdbc:h2:mem:tutorial
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=

# jpa
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.show_sql=true
spring.jpa.properties.hibernate.format_sql=true

# Nova propriedade a partir da versao 2.5 do Spring Boot:
spring.jpa.defer-datasource-initialization=true
```

Passo 3: O banco de dados H2 será utilizado para testarmos nossa API. Como optamos por trabalhar com um banco de dados salvo em memória, isso quer dizer que toda vez que você reiniciar seu projeto, todas as alterações de dados serão perdidas. Então, para ter alguns registros básicos no BD, precisamos estabelecer um arquivo com extensão .sql, onde iremos escrever queries para inserção de dados no BD. Dessa forma, toda vez que seu projeto foi inicializado, o banco de dados será populado com aqueles dados. O nome da tabela a qual você estará adicionando dado é o mesmo nome da sua classe criada no passo 1. Atenção, em projetos que estão em produção, você nunca deve salvar senhas no banco de dados da mesma forma como o usuário a digita no login (ie. a string direta da senha, eg. “123456”). O procedimento indicado é calcular um hash para senha (se possível combinando com outra senha única), e salvar os dados na forma de hash. Contudo, como esse tutorial tem fins didáticos para explicar conceitos de API e não de segurança, optamos por manter a senha descrita literalmente, para facilitar a compreensão dos estudantes.

```
INSERT INTO PEDIDOS(email, descricao, valor) VALUES
('cliente1@eail.com', 'Pedido 1', 11.11),
('cliente2@eail.com', 'Pedido 2', 22.22),
('cliente1@eail.com', 'Pedido 3', 33.33),
('cliente1@eail.com', 'Pedido 4', 44.44),
('cliente3@eail.com', 'Pedido 5', 55.55);
```

Passo 4: Veja seu banco de dados de pé. Acessar o link <http://localhost:8080/h2-console/>, e indique a URL “jdbc:h2:mem:tutorial”. Clique em Connect. Em seguida observe seu BD.

The screenshot shows the H2 Console Login dialog box. At the top, there is a language dropdown set to 'English' and links for 'Preferences', 'Tools', and 'Help'. The 'Login' section has a 'Saved Settings' dropdown set to 'Generic H2 (Embedded)'. Below this, the 'Setting Name' is also 'Generic H2 (Embedded)', with 'Save' and 'Remove' buttons. The 'Driver Class' is 'org.h2.Driver'. The 'JDBC URL' is 'jdbc:h2:mem:tutorial'. The 'User Name' is 'sa', and the 'Password' field is empty. At the bottom are 'Connect' and 'Test Connection' buttons.

Passo 5: Confira que os dados que você inseriu no arquivo data.sql, estão contidos no BD rodando o comando SELECT.

The screenshot shows the H2 Console interface. The top toolbar includes icons for undo, redo, and other actions, along with 'Auto commit' (checked), 'Max rows' (1000), and 'Auto complete' (Off). The left sidebar shows the database structure: 'jdbc:h2:mem:tutorial' with tables 'PEDIDOS' and 'INFORMATION_SCHEMA', 'Sequences', and 'Users'. The main area shows the SQL statement 'SELECT * FROM PEDIDOS' being executed. The results are displayed in a table with 5 rows and 4 columns: ID, DESCRICAO, EMAIL, and VALOR. Below the table, it says '(5 rows, 2 ms)' and there is an 'Edit' button.

ID	DESCRICAO	EMAIL	VALOR
1	Pedido 1	cliente1@eail.com	11.11
2	Pedido 2	cliente2@eail.com	22.22
3	Pedido 3	cliente1@eail.com	33.33
4	Pedido 4	cliente1@eail.com	44.44
5	Pedido 5	cliente3@eail.com	55.55

Criar endpoint

Passo 1: Criar repositório para consultar dados no BD.

src/main/java/com/es3/vendas/repository/PedidosRepository.java

```
package com.es3.vendas.repository;

import com.es3.vendas.entities.Pedidos;
import org.springframework.data.jpa.repository.JpaRepository;

public interface PedidosRepository extends JpaRepository<Pedidos, Long> {

}
```

Passo 2: Criar Classe DTO (Data Transfer Object) para receber dados para entrega ao usuário.

src/main/java/com/es3/vendas/dto/PedidosDto.java

```
package com.es3.vendas.dto;

import com.es3.vendas.entities.Pedidos;
import lombok.Getter;

@Getter
public class PedidosDto {

    private Long id;
    private String email;
    private String descricao;
    private Double valor;

    public PedidosDto(Pedidos pedidos) {
        this.id = pedidos.getId();
        this.email = pedidos.getEmail();
        this.descricao = pedidos.getDescricao();
        this.valor = pedidos.getValor();
    }

}
```

Passo 3: Criar serviço que vai integrar seu endpoint com a consulta no banco de dados.

src/main/java/com/es3/vendas/services/PedidosService.java

```
package com.es3.vendas.services;

import com.es3.vendas.dto.PedidosDto;
import com.es3.vendas.entities.Pedido;
import com.es3.vendas.entities.repository.PedidosRepository;
import lombok.AllArgsConstructor;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Service;
import org.springframework.web.server.ResponseStatusException;

@AllArgsConstructor
@Service
public class PedidosService {

    private PedidosRepository pedidosRepository;
```

```

    public PedidosDto findPedidosById(Long id) {
        Pedido pedido = pedidosRepository.findById(id).orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND));
        PedidosDto pedidosDto = new PedidosDto(pedido);
        return pedidosDto;
    }
}

```

Passo 4: Criar controller, que será o responsável por receber as requisições HTTP disparar o processo de consulta e retorno da resposta ao usuário.

src/main/java/com/es3/vendas/controller/PedidosController.java

```

package com.es3.vendas.controller;

import com.es3.vendas.services.PedidosService;
import lombok.AllArgsConstructor;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@AllArgsConstructor
@RequestMapping("pedidos")
public class PedidosController {

    private final PedidosService pedidosService;

    @GetMapping("/{id}")
    public ResponseEntity<?> findPedidosById(@PathVariable Long id) {
        return ResponseEntity.ok(this.pedidosService.findPedidosById(id));
    }
}

```

Passo 5: Testar seu endpoint

Postman → GET → URL: **http://localhost:8080/pedidos/1** → Send

Passo 6: Postar parcial no git

```

git add .
git commit -m "Endpoint"
git push origin master

```

Tokens

Passo 1: Inclua novas dependências no projeto. Como não adicionamos ele na criação do nosso projeto no inicializr, vamos adiciona-lo manualmente no nosso arquivo pom.xml. OBS: por algum motivo obscuro, durante o desenvolvimento desse projeto, ao adicionar as duas dependências juntas, nossa IDE não conseguia baixar todas as dependências. Para resolver isso, adicionamos primeiro a dependência “spring-boot-starter-security”, em seguida reiniciamos a IDE, e então adicionamos a dependência “jjwt”.

pom.xml

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>

<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt</artifactId>
</dependency>
```

Passo 2: Primeiro vamos criar uma entidade para salvar dados de usuários autorizados a utilizar o sistema. Como vamos usar essa entidade (chamada Usuario) para fazer login no sistema, precisamos transformar essa classe em “algo” que possamos usar no login, devemos informar isso ao Spring fazendo uma extensão de interface. Por definição, se você implementou uma interface, você é obrigado a sobrescrever, implementar os métodos que estão definidos nela.

Sendo assim, chame a implementação do UserDetails. Uma das exigências dessa implementação é criar um perfil dos usuários (i.e. um modelo para você reunir informações como id, nome e tipo de usuário – ex. usuário comum, admin, etc..). Por isso vamos ter que criar essa classe (próximo passo) que vai ser uma entidade de JPA.

src/main/java/com/es3/vendas/entities/Usuario.java

```
package com.es3.vendas.entities;

import lombok.Getter;
import lombok.Setter;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

import javax.persistence.*;
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

@Entity
@Getter
@Setter
public class Usuario implements UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nome;
    private String email;
    private String senha;
```

```
@ManyToMany(fetch = FetchType.EAGER)
private List<Perfil> perfis = new ArrayList<>();

@Override
public Collection<? extends GrantedAuthority> getAuthorities() {
    return this.perfis;
}

@Override
public String getPassword() {
    return this.senha;
}

@Override
public String getUsername() {
    return this.email;
}

@Override
public boolean isAccountNonExpired() {
    return true;
}

@Override
public boolean isAccountNonLocked() {
    return true;
}

@Override
public boolean isCredentialsNonExpired() {
    return true;
}

@Override
public boolean isEnabled() {
    return true;
}
}
```


Passo 3: Criar a classe perfil, que vai conter dados de nome do usuário e seu id no banco de dados

src/main/java/com/es3/vendas/entities/Perfil.java

```
package com.es3.vendas.entities;

import lombok.Getter;
import lombok.Setter;
import org.springframework.security.core.GrantedAuthority;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
@Getter
@Setter
public class Perfil implements GrantedAuthority {

    private static final long serialVersionUID = 1L;

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

    private String nome;

    @Override
    public String getAuthority() {
        return nome;
    }
}
```

Passo 4: Criar e desenvolver repositório associado a classe Usuario.

src/main/java/com/es3/vendas/repository/UsuarioRepository.java

```
package com.es3.vendas.repository;

import com.es3.vendas.entities.Usuario;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.Optional;

public interface UsuarioRepository extends JpaRepository<Usuario, Long> {

    Optional<Usuario> findByEmail(String email);
}
```

Passo 5: Criar um serviço para autenticação.

src/main/java/com/es3/vendas/security/AuthService.java

```
package com.es3.vendas.security;

import com.es3.vendas.entities.Usuario;
import com.es3.vendas.repository.UsuarioRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

import java.util.Optional;

@Service
public class AuthService implements UserDetailsService {

    @Autowired
    private UsuarioRepository usuarioRepository;

    @Override
    public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {
        Optional<Usuario> usuario = usuarioRepository.findByEmail(email);
        if (usuario.isPresent()) {
            return usuario.get();
        }
        throw new UsernameNotFoundException("Usuário ou Senha inválidos");
    }
}
```

Passo 6: Adicionar dados de Usuários no nosso BD. Neste projeto, vamos armazenar as senhas no nosso BD usando o hash BCrypt. Esse algoritmo tem como vantagem ser desenhado para ser lento. Isso é uma vantagem porque num eventual ataque hacker, o atacante levaria muito tempo para conseguir criar um dicionário de senhas para aproveitar as senhas observadas (Veja Hack Reactor 2014 <https://www.youtube.com/watch?v=O6cmuiTBZVs>). Além deste fator, quando toda vez que geramos um hash desse método, obteremos um valor diferente de hash. Essa é uma característica que difere este algoritmo de outros como SHA256. No nosso sistema, os Clientes 1 e 2, terão a senha hipotética de “123456”, enquanto o Cliente 3 terá a senha de 654321. Para calcular valores de hash BCrypt você pode implementar o método “encode” da classe “BCryptPasswordEncoder”. Veja um exemplo de implementação desta no projeto depositado no seguinte repositório: <https://github.com/biagolini/JavaSpringSecurityCrypto>.

data.sql

```
INSERT INTO PEDIDO (email, descricao, valor) VALUES
('cliente1@eail.com', 'Pedido 1', 11.11),
('cliente2@eail.com', 'Pedido 2', 22.22),
('cliente1@eail.com', 'Pedido 3', 33.33),
('cliente1@eail.com', 'Pedido 4', 44.44),
('cliente3@eail.com', 'Pedido 5', 55.55);
```

```
INSERT INTO USUARIO (nome, email, senha) VALUES
('Nome do cliente 1', 'cliente1@email.com', '$2a$10$2.P3Sj9JMG5pHGJYZAKA.eMX97aUcQw-MWRNX9Puo1k2YBQuY0BvKS'),
('Nome do cliente 2', 'cliente2@email.com', '$2a$10$WXfQsblcdtTpL/dndZUC9.NUXDk6bvO-TpCr33N7E2/cB/U83RTfvu'),
('Nome do cliente 3', 'cliente3@email.com', '$2a$10$.xuFBDRtX-nUYZT1wHb0vGO8VPB13x/xQF89xNQ1kAhktqzOqQ/h.G');
```

Passo 7: Para gerar os Tokens, precisamos criar 2 novas variáveis para o projeto, uma senha para gerar Tokens (para o procedimento de Salt) + um valor de número de milissegundos que o Token permanecerá válido após sua criação. Essas variáveis ficaram disponíveis nas configurações do properties.

src/main/resources/application.properties

```
# jwt
tutorial.jwt.secret=A+X;fTJpPd,TD9dwVq(hsHX,ya^<wsD_UK7L+@=S;{'CydP}{v@}G'b>et;yz$*\yL5S8EJN:%P:X%H9>#nYlrX}
@\s?CQcpspH,2emzBc!Q[V'AYa~uzF8WR~AUrMzxp/V$9([S9X#zj/CH('#]B_Hc+%fGhe27YB;^j4\Xk=Ju"Ap~_&<L;=!Z;!;2UP;!hF3P]
j85#*`&T]/kB/W^6$v~u6qpejL>kY^f)sy4:qTq_Ec!-z!@aAp~sLKGU>$
tutorial.jwt.expiration=86400000
```

Observações sobre o script

tutorial.jwt.secret: Senha para gerar tokens.

tutorial.jwt.expiration: Duração do token.

Passo 8: Desenvolva sua classe LoginForm, com getters e setters. Essa classe deve retornar apenas dados primitivos.

src/main/java/com/es3/vendas/dto/LoginForm.java

```
package com.es3.vendas.dto;

import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;

import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class LoginForm {

    private String email;
    private String senha;

    public UsernamePasswordAuthenticationToken converter() {
        return new UsernamePasswordAuthenticationToken(email, senha);
    }

}
```

Observações sobre o script

UsernamePasswordAuthenticationToken: Método que será usado para validar o token.

Passo 9: Desenvolva uma classe de lidar com de erros. Essa classe será requerida no próximo passo (ErroDeValidacaoHandler). A Classe ErroDeFormularioDto classe irá apresentar apenas duas informações, o nome do campo que deu erro (atributo campo), e a descrição do erro (atributo erro). Aproveite e já crie um construtor que recebe os dados de campo e erro, bem como os getters.

src/main/java/com/es3/vendas/security/ErroDeFormularioDto.java

```
package com.es3.vendas.security;

import lombok.AllArgsConstructor;
import lombok.Getter;

@Getter
@AllArgsConstructor
public class ErroDeFormularioDto {

    private String campo;
    private String erro;
}
```

Passo 10: Desenvolva seu ErroDeValidacaoHandler.

src/main/java/com/es3/vendas/security/ErroDeValidacaoHandler.java

```
package com.es3.vendas.security;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.MessageSource;
import org.springframework.context.i18n.LocaleContextHolder;
import org.springframework.http.HttpStatus;
import org.springframework.validation.FieldError;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestControllerAdvice;

import java.util.ArrayList;
import java.util.List;

@RestControllerAdvice
public class ErroDeValidacaoHandler {

    @Autowired
    private MessageSource messageSource;

    @ResponseStatus(code = HttpStatus.BAD_REQUEST)
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public List<ErroDeFormularioDto> handle(MethodArgumentNotValidException exception) {
        List<ErroDeFormularioDto> dto = new ArrayList<>();

        List<FieldError> fieldErrors = exception.getBindingResult().getFieldErrors();
        fieldErrors.forEach(e -> {
            String mensagem = messageSource.getMessage(e, LocaleContextHolder.getLocale());

            ErroDeFormularioDto erro = new ErroDeFormularioDto(e.getField(), mensagem);
            dto.add(erro);
        });

        return dto;
    }
}
```

```
}  
}
```

Passo 11: Pensando na saída do token para o Cliente, vamos desenvolver o TokenDto que será uma classe usada no serviço de criação do token (próximo passo).

src/main/java/com/es3/vendas/dto/TokenDto.java

```
package com.es3.vendas.dto;  
  
import lombok.AllArgsConstructor;  
import lombok.Getter;  
  
@Getter  
@AllArgsConstructor  
public class TokenDto {  
  
    private String token;  
    private String tipo;  
  
}
```

Observações sobre o script

Obs: não faz sentido um token que não tem dados de valor do token, por isso eliminamos o construtor default, e com um novo construtor fechamos que só se pode criar um token se tivemos dados de token e tipo.

Passo 12: Desenvolver o serviço de criação e validação de tokens.

src/main/java/com/es3/vendas/security/TokenService.java

```
package com.es3.vendas.security;  
  
import com.es3.vendas.entities.Usuario;  
import io.jsonwebtoken.Claims;  
import io.jsonwebtoken.Jwts;  
import io.jsonwebtoken.SignatureAlgorithm;  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.security.core.Authentication;  
import org.springframework.stereotype.Service;  
  
import java.util.Date;  
  
@Service  
public class TokenService {  
  
    // Chamamos a senha usada para criar os tokens  
    @Value("${tutorial.jwt.secret}")  
    private String secret;  
  
    // Chamamos o dado de duração dos tokens  
    @Value("${tutorial.jwt.expiration}")  
    private String expiration;  
  
    public String gerarToken(Authentication authentication) {  
        Usuario logado = (Usuario) authentication.getPrincipal();  
        Date hoje = new Date();  
        Date dataExpiracao = new Date(hoje.getTime() + Long.parseLong(expiration));  
  
        String token = Jwts.  
            builder().setSubject(logado.getId().toString()).setExpiration(dataExpiracao).signWith(  
                SignatureAlgorithm.HS256, secret).compact();  
    }  
  
}
```

```

        return Jwts.builder()
            .setIssuer("userType: Cliente")
            .setSubject(logado.getId().toString())
            .setIssuedAt(hoje)
            .setExpiration(dataExpiracao)
            .signWith(SignatureAlgorithm.HS256, secret)
            .compact();
    }

    public boolean isTokenValido(String token) {
        try {
            Jwts.parser().setSigningKey(this.secret).parseClaimsJws(token);
            return true;
        } catch (Exception e) {
            return false;
        }
    }

    public Long getIdUsuario(String token) {
        Claims claims = Jwts.parser().setSigningKey(this.secret).parseClaimsJws(token).getBody();
        return Long.parseLong(claims.getSubject());
    }
}

```

Observações sobre o script

Value("\${...}"): usado para importar as variáveis (strings) declaradas no properties.

authentication.getPrincipal: Usado para obter quem é o usuário que está logado. Como o getPrincipal devolve um object, então tenho que fazer um cast para usuário, i.e. declara o usuários entre parênteses.

dataExpiracao: data que o token vai expirar

Long.parseLong(expiration): dado que vamos pegar lá no application properties que vai representar um numero certo de dias, para o qual o token será valido

setIssuer: Pode ser usado para inserir uma string dentro do texto. Isso é muito útil quando você tem mais de um serviço de geração de tokens, e em alguns casos libera tokens para um tipo de usuário (ex. usuário padrão) e em outros casos libera tokens para usuários com acesso privilegiado (ex. usuário administrador).

setSubject: Quem é o dono token (qual foi o usuário que estava logado quando fez a requisição do token).

logado.getId().toString(): Conversão de tipo de variável, necessária porque a id de usuário é um numero do tipo long, e precisamos dessa informação na forma de uma string.

setIssuedAt(...): data de criação do Token.

setExpiration(...): data de validade do Token.

signWith(SignatureAlgorithm.HS256, secret): algoritmo + senha usada para fazer a assinatura do token.

compact(): Compacta tudo que foi feito e transforma numa String.

isTokenValido: A lógica da validação é simples. Você passa o token (que vai conter as informações de como foi feito para gerar o token + mensagem), e internamente o Spring vai gerar outro token e comparar se as assinaturas são iguais.

getIdUsuario: Neste método extraímos os dados do corpo do token. Na demonstração de teste da geração de Token (final desse capítulo) você vai ver isso dentro de uma mensagem que será: {"iss":"userType: Cliente","sub":"1","iat":1648333122,"exp":1648419522}}

parser: Método que faz o parser de um token, i.e. quebra as partes do token.

setSigningKey: identificação da senha que usamos para gerar o token.

parseClaimsJws: método que retorna um token e as mensagens que você setou dentro do token. Nesse método, se estiver tudo certo, ele vai devolver um objeto do tipo Claim. Se o token for inválido, vai devolver uma exception. Por isso usamos o try-catch. Se der certo retornara a próxima linha (true), se der errado executa o catch (que vai retornar false).

Passo 13: Desenvolvimento do filtro. O fluxo da autenticação por filtro será a seguinte. Chamamos uma cópia do método de criar novos tokens, e do método de busca de id de usuário por e-mail (necessário para se gerar um novo token e comparar com o apresentado). Em seguida aplicamos o filtro (doFilterInternal), onde primeiro vamos extrair o token do cabeçalho da requisição (recuperarToken). Em seguida, aplicamos a autenticação (autenticarCliente) onde internamente o Spring vai gerar outro Token e comparar com o token apresentado para ver se as assinaturas são iguais.

src/main/java/com/es3/vendas/security/AutenticacaoViaTokenFilter.java

```
package com.es3.vendas.security;

import java.io.IOException;

import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.es3.vendas.entities.Usuario;
import com.es3.vendas.entities.repository.UsuarioRepository;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.web.filter.OncePerRequestFilter;

public class AutenticacaoViaTokenFilter extends OncePerRequestFilter {

    private TokenService tokenService;
    private UsuarioRepository repository;

    public AutenticacaoViaTokenFilter(TokenService tokenService, UsuarioRepository repository) {
        this.tokenService = tokenService;
        this.repository = repository;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
        throws ServletException, IOException {
```

```

        String token = recuperarToken(request);
        boolean valido = tokenService.isTokenValido(token);
        if (valido) {
            autenticarCliente(token);
        }

        filterChain.doFilter(request, response);
    }

    // Metodo que vai pegar as informações do que estão no Payload do token (ou seja é a
    // mensagem que enviamos dentro do token) onde falamos os dados do usuario e horario da re-
    // quisição. Geramos um novo token com esses dados, e comparamos se os tokens são iguais.
    private void autenticarCliente(String token) {
        Long idUsuario = tokenService.getIdUsuario(token);
        Usuario usuario = repository.findById(idUsuario).get();
        UsernamePasswordAuthenticationToken authentication = new UsernamePasswordAuthen-
        ticationToken(usuario, null, usuario.getAuthorities());
        SecurityContextHolder.getContext().setAuthentication(authentication);
    }

    // Metodo para pegar o token dentro do Header da requisição
    private String recuperarToken(HttpServletRequest request) {
        String token = request.getHeader("Authorization");
        if (token == null || token.isEmpty() || !token.startsWith("Bearer ")) {
            return null;
        }

        return token.substring(7, token.length());
    }
}

```

Observações sobre o script

Construtor: como toda vez que essa classe for chamada vamos precisar de acesso (ou ter uma cópia) ao serviço de criação de token e busca de id por email, precisamos criar uma estrutura que faça a injeção de dependência. Como não é possível fazer a injeção via `@Autowired`, porque a classe `AutenticacaoViaTokenFilter` não é um bean gerenciado pelo Spring (O filtro foi instanciado manualmente por nós, na classe `SecurityConfigurations` e portanto o Spring não consegue realizar injeção de dependências via `@Autowired`), estabelecemos a injeção de dependência colocando os elementos que temos dependência dentro do construtor. Dessa forma, quando formos chamar o `AutenticacaoViaTokenFilter` pelo `SecurityConfigurations`, passamos uma cópia desses métodos. Como no `SecurityConfigurations`, onde é possível fazer a injeção de dependência por `@Autowired` (passo anterior desse tutorial).

OncePerRequestFilter: classe que vamos herdar que “impõe” que o filtro seja chamado uma única vez a cada requisição.

doFilterInternal: ao usarmos o `OncePerRequestFilter` somos obrigados a sobrescrever esse método (método abstrato). Será aqui que vamos implementar a lógica de verificação, i.e. pegar o token do cabeçalho, verificar se está ok, e se tudo estiver certo fazer a autenticação.

filterChain.doFilter: Linha de código que diz mais ou menos os seguinte “fiz tudo que tinha que ser feito, pode seguir com a requisição”.

recuperarToken: método privado (método que criamos dentro dessa classe e não está disponível para ninguém de fora) onde pegamos o token no Header da requisição

getHeader("Authorization"): método para extrair a string do cabeçalho da requisição.

if (token == null... Bearer "): Verificamos se o token veio certinho, se o token não passar por essa validação, isso quer dizer que tem algo errado, então você já retorna um null e mata o processo aqui mesmo.

token.substring(7,..): Se passar por essa validação, então você pega a string que veio no Header e joga fora os 7 primeiros dígitos (que representam uma identificação do token – veja introdução desse capítulo). Dessa forma o método recuperarToken vai retornar apenas o token propriamente dito.

autenticarCliente: método privado (método que criamos dentro dessa classe e não está disponível para ninguém de fora) onde fazemos a validação do token propriamente dita.

SecurityContextHolder: É um método do spring que força a mudança de status do cliente que faz a requisição para autorizado. Ou seja, é como se você estivesse fingindo que foi feito uma autenticação com login e senha e deu tudo ok.

setAuthentication(authentication): usamos essa estrutura para passar a informação de quem é o usuário que está autenticado

UsernamePasswordAuthenticationToken(usuario, senha, perfil_de_acesso): Classe nativa do Spring que passa a estrutura do usuário que está fazendo a autenticação

Passo 14: Até aqui, criamos arquivos relacionados a coleta de dados e processamento dos tokens. Nesse passo e no seguinte vamos de fato modificar a estrutura de autenticação. Aqui, vamos desenvolver o AutenticacaoController para testar se um Token recebido é de fato autêntico.

src\main\java\com\es3\vendas\controller\AutenticacaoController.java

```
package com.es3.vendas.controller;

import com.es3.vendas.dto.LoginForm;
import com.es3.vendas.dto.TokenDto;
import com.es3.vendas.security.TokenService;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.AuthenticationException;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/auth")
public class AutenticacaoController {

    @Autowired
    private AuthenticationManager authManager;

    @Autowired
    private TokenService tokenService;

    @PostMapping
    public ResponseEntity<TokenDto> autenticar(@RequestBody LoginForm form) {
        UsernamePasswordAuthenticationToken dadosLogin = form.converter();

        try {
            Authentication authentication = authManager.authenticate(dadosLogin);
            String token = tokenService.gerarToken(authentication);
            return ResponseEntity.ok(new TokenDto(token, "Bearer"));
        } catch (AuthenticationException e) {
            return ResponseEntity.badRequest().build();
        }
    }
}
```

Observações sobre o script

@RequestMapping("/auth"): Mapeamento de como cair nessa classe. Dessa forma para fazer a autenticação, o frontend vai enviar os dados de login para o endereço: <http://localhost:8080/auth>.

autenticar(...): metodo da autenticação

@PostMapping: Como estamos enviado dados de login (usuário e senha), precisamos usar o método post, para que o dado entre no sistema e seja processado (porcamente falando: no post o dado entra e é processado e pode sair ou não, no get o dado é processado e sai).

@RequestBody @Valid LoginForm form: @RequestBody = usado para indicar que os dados do form vão vir dentro do corpo da requisição. @Valid LoginForm form = indicação de qual é o dado do login.

AuthenticationManager: Essa classe é do Spring, mas ele não consegue fazer a injeção de dependências dela automaticamente. Por isso no SecurityConfiguration vamos fazer um sobrescrita no método, para permitir essa injeção (ou seja, essa parte só vai funcionar quando você terminar o próximo passo).

UsernamePasswordAuthenticationToken: Objeto que é usado para receber os dados de usuário e senha

form.converter(): Método que criamos para converter os dados que vão entrar na forma de texto do front, para um objeto do tipo UsernamePasswordAuthenticationToken (o método foi implementado no LoginForm.java).

Try: tente fazer o login

authenticate(...): Método que faz a autenticação, obrigatoriamente deve receber os dados de login de um objeto do tipo UsernamePasswordAuthenticationToken. Se der certo, roda as próximas linhas dentro do { }, se der errado sai do { } e vai para o catch

gerarToken(...): roda a função de gerar o token, e salva o resultado dentro de uma string.

gerarToken(authentication): authentication é usado para enviar junto com a requisição do token a informação de quem é o usuário que fez o pedido de login

return ResponseEntity.ok(...): Retorne o status 200 + mensagem de dentro os parênteses.

return ResponseEntity.ok(new TokenDto(token, "Bearer")): Cria uma instancia, que vai virar uma JSON, com o dado do Token que acabou de ser criado + outra string ("Bearer") usada para descrever o tipo de token. Bearer é um dos mecanismos de autenticação utilizados no protocolo HTTP, tal como o Basic e o Digest.

catch (AuthenticationException e): Se não deu certo, pega a resposta de exceção

ResponseEntity.badRequest().build(): Apenas para definir que em caso de erro em vez de um erro 500 (erro de servidor) você quer retornar um erro de dados inválidos (400s).

Passo 15: A Classe SecurityConfigurations é o seu quartel general da segurança da sua aplicação. É aqui que definimos quais são os endpoints e métodos (Get, Post, Put,...) que são de livre acesso, e quais são restritos a usuários com tokens.

src/main/java/com/es3/vendas/security/SecurityConfigurations.java

```
package com.es3.vendas.security;

import com.es3.vendas.entities.repository.UsuarioRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpMethod;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.builders.WebSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

@EnableWebSecurity
@Configuration
public class SecurityConfigurations extends WebSecurityConfigurerAdapter {

    @Autowired
    private AuthService authService;

    @Autowired
    private TokenService tokenService;

    @Autowired
    private UsuarioRepository usuarioRepository;

    @Override
    @Bean
    protected AuthenticationManager authenticationManager() throws Exception {
        return super.authenticationManager();
    }

    //Configurações relacionadas a autenticação
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(authService).passwordEncoder(new BCryptPasswordEncoder());
    }

    //Configurações relacionadas a autorização
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
```

```

        .antMatchers(HttpMethod.POST, "/auth").permitAll()
        // .antMatchers(HttpMethod.GET, "/pedidos/*").permitAll() // PARA LIBERAR
ACESSO AO ENDPOINT, REMOVA A TAG DE COMENTÁRIO DESSA LINHA
        .anyRequest().authenticated()
        .and().csrf().disable()
        .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATE-
LESS)
        .and().addFilterBefore(new AutenticacaoViaTokenFilter(tokenService, usu-
arioRepository), UsernamePasswordAuthenticationFilter.class);
    }

    //Configurações de recursos estaticos(js, css, imagens, etc.)
    @Override
    public void configure(WebSecurity web) throws Exception {
    }
}

```

Observações sobre o script:

@EnableWebSecurity: Habilita o Spring Security

@Configuration: Permite que façamos alterações de configurações em Bins do Java.

WebSecurityConfigurerAdapter: Classe que herdamos para ter alguns métodos de segurança que iremos sobrescrever

antMatchers: Funciona como uma espécie de filtro para liberar acesso a algum endpoint a ser especificado a seguir

.antMatchers(HttpMethod.POST, "/auth").permitAll(): Você precisa liberar a URL de auth para que os usuários possam ter acesso a autenticação.

HttpMethod.GET: Estamos especificando que somente o GET que é liberado. Se você tentar fazer um DELETE vai receber um 403 (Forbidden).

anyRequest().authenticated(): indica que qualquer coisa que não estiver listado acima, deve ser requerido a autenticação. Portanto, evita que uma URL que não foi configurada seja pública.

passwordEncoder(new BCryptPasswordEncoder()): Metodo usado para fazer a criptografia da senha do usuário. Por conta desse código o usuário vai digitar “123456” e será feita a conversão para o HASH do algoritmo BCrypt (de acordo com o professor é um algoritmo famoso hoje em dia).

csrf().disable(): Csrf é uma abreviação para cross-site request forgery, que é um tipo de ataque hacker que acontece em aplicações web. Como vamos fazer autenticação via token, automaticamente nossa API está livre desse tipo de ataque. Nós vamos desabilitar isso para o Spring security não fazer a validação do token do csrf.

sessionManagement(): método para passar parâmetros de como serão gerenciadas as sessões

sessionCreationPolicy(...): método que vamos chamar para dizer que não queremos usar sessão.

SessionCreationPolicy.STATELESS: define que nosso projeto, quando for feita autenticação, não é para criar sessão (porque vamos usar token).

AuthenticationManager: Essa classe é do Spring, mas ele não consegue fazer a injeção de dependências dela automaticamente, a não ser que nós configuremos isso. Podemos fazer isso na nossa classe SecurityConfiguration.

Passo 16: Testar o processo de geração de tokens (“login”), e o acesso ao endpoint fechado.

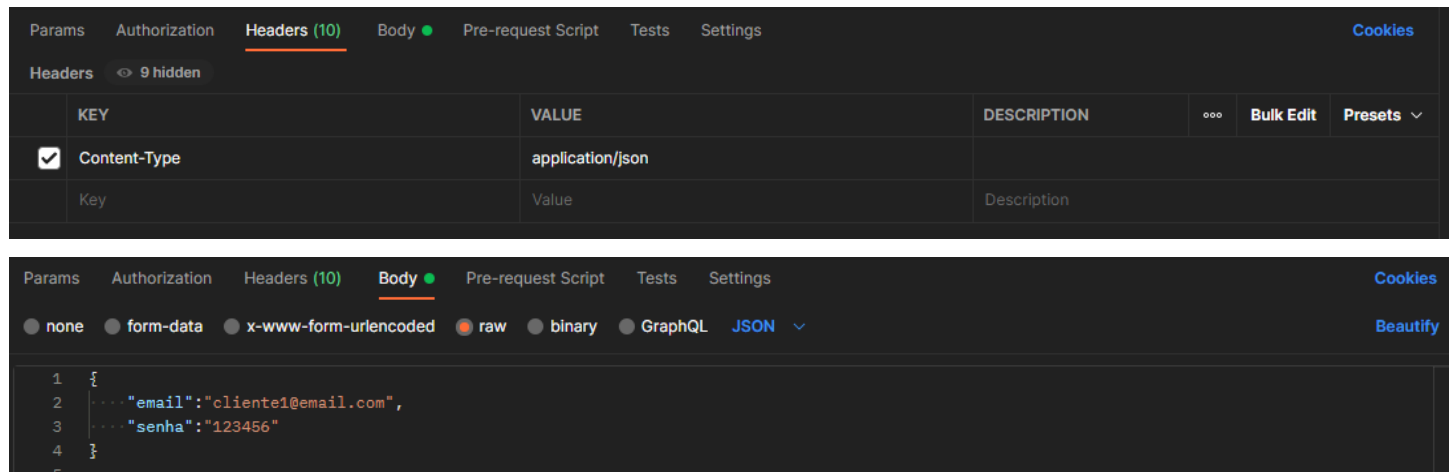
TENTATIVA DE GERAR TOKEN COM USUARIO E SENHA ERRADO

Postman → URL: `http://localhost:8080/auth` → Selecione opção POST.

Postman → Body → raw → insira o seguinte JSON:

```
{
  "email": "cliente1@email.com",
  "senha": "11111"
}
```

RESPOSTA ESPERADA: 404 Bad request



TENTATIVA DE GERAR TOKEN COM USUARIO E SENHA CORRETOS

Postman → URL: `http://localhost:8080/auth` → Selecione opção POST.

Postman → Body → raw → insira o seguinte JSON:

```
{
  "email": "cliente1@email.com",
  "senha": "123456"
}
```

RESPOSTA ESPERADA: 200 Ok

Exemplo de corpo de resposta

```
{
  "token":
"eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJ1c2VyVHlwZTogQ2xpZW50ZSIsInN1YiI6IjEiLCJpYXQiOiJlMjNDgZMzMjIsImV4cCI6MTY0ODQxOTUyMn0.twnFEu6-H38ofxjk2UV0vNGHZnhilJBfEKWTPhKTTik",
  "tipo": "Bearer"
}
```

Extra: Os dois primeiros valores até o “.” são códigos base64 do token. Neste, vamos teremos as informações relacionadas a autenticação desse usuário. Se quiser ler o que apareceu no seu token use o site <https://www.base64decode.org/> para decodificar essas informações.

```
{
  "token":
  "eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJlc2VybWlwa2VzTogQ2xpZW50ZSI6InN1YiI6IjEiLCJpYXQiOiJlbnQzZmZmMjIjImV4cCI6MTY0ODQxOTUyMn0.twnFEu6-H38ofxjk2UV0vNGHZnhiLJBfEKWTPPhKTTik",
  "tipo": "Bearer"
}
```

eyJhbGci... = {"alg":"HS256"}

eyJpc3Mi... = {"iss":"userType: Cliente","sub":"1","iat":1648333122,"exp":1648419522}}

IZyLQ3E... = assinatura, não faz sentido ver a tradução disso.

TENTATIVA DE CONSULTA SEM TOKEN

Postman → URL: `http://localhost:8080/pedidos/1` → Selecione opção GET.

RESPOSTA ESPERADA: 403 Forbidden

TENTATIVA DE CONSULTA COM TOKEN

Postman → Authorization → Bearer Token → insira o token recebido no processo acima:

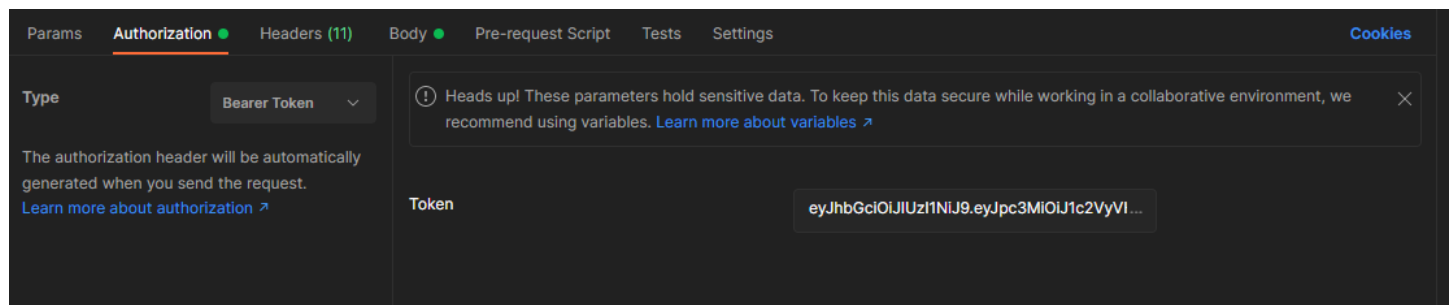
```
eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJlc2VybWlwa2VzTogQ2xpZW50ZSI6InN1YiI6IjEiLCJpYXQiOiJlbnQzZmZmMjIjImV4cCI6MTY0ODQxOTUyMn0.twnFEu6-H38ofxjk2UV0vNGHZnhiLJBfEKWTPPhKTTik
```

outra opção para passar o token na requisição

Postman → Header → Authorization = Bearer + espaço + token que foi recebido na autenticação

OBS: o Content-Type não precisa estar selecionado (mas se tiver não vai ter problema)

RESPOSTA ESPERADA: 200 ok



Params	Authorization	Headers (10)	Body	Pre-request Script	Tests	Settings	Cookies
Headers 9 hidden							
	KEY	VALUE	DESCRIPTION	...	Bulk Edit	Presets	▼
<input checked="" type="checkbox"/>	Authorization	Bearer eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJ1c2VyVHlwZTo...					
	Key	Value	Description				

Body	Cookies (1)	Headers (11)	Test Results	Status: 200 OK Time: 12 ms Size: 419 B Save Response ▼
Pretty Raw Preview Visualize JSON ▼				
<pre>1 { 2 "id": 1, 3 "email": "cliente1@eail.com", 4 "descricao": "Pedido 1", 5 "valor": 11.11 6 }</pre>				

Passo 17: Postar parcial no git

```
git add .
git commit -m "Token"
git push origin master
```