Spring Security com Basic

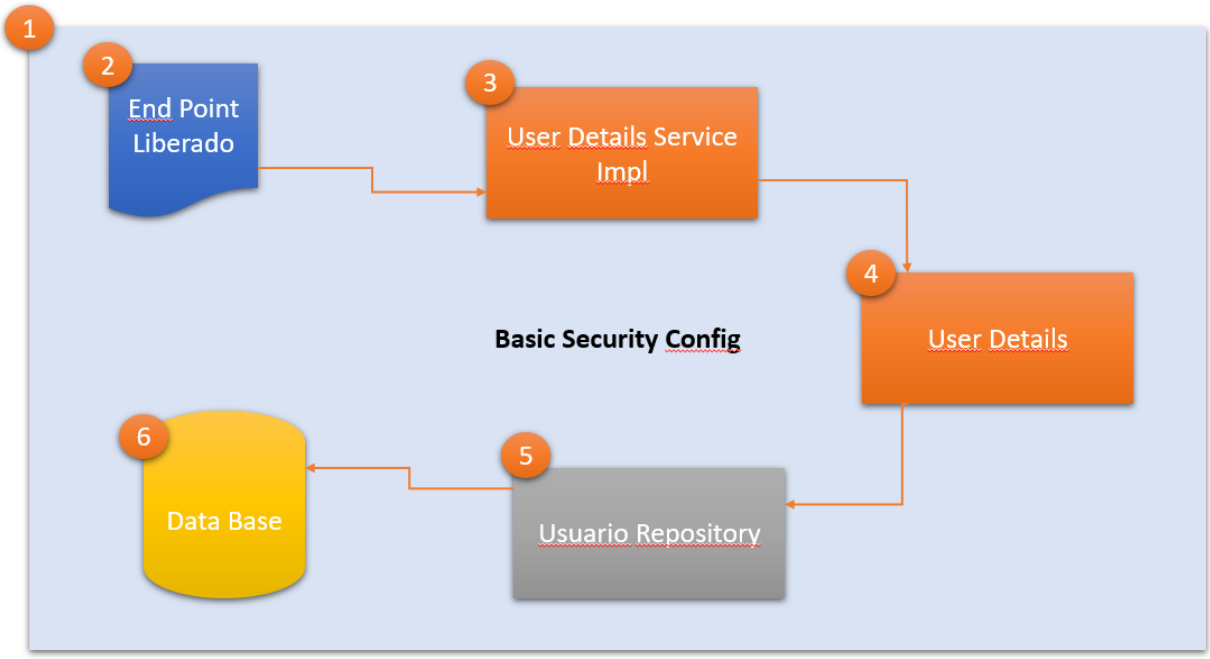
Analisando o nosso e-commerce podemos perceber que nossa Api não possui nenhuma segurança, ou seja, qualquer pessoa pode acessar nossos end point e ter acesso aos nossos recursos, precisamos entender que algumas aplicações contem informações como, dados pessoais, dados bancários, login e senha, precisamos garanti que nossa Api e estes dados estejam devidamente protegidos.
E para que isto seja pessoal podemos contar com Dependência do Spring chamada **Spring Security**.

Assim como o Spring Data (que vimos na aula anterior, Api Rest Spring com JPA) o Spring Segurity é um framework para Java que provê autenticação, autorização e diversas outras funcionalidades para aplicações corporativas. Iniciado em 2003 por Ben Alex o Spring Security é distribuído sob a licença Apache Licence. Ele oferece diversos recursos que permitem muitas práticas comuns de segurança serem declaradas ou configuradas de uma forma direta.

O modo de autenticação que utilizaremos e o Basic, onde entraremos com o login e a senha de usuário através de um end point liberado, o Spring Security irá encriptar a senha e fazer uma consulta no nosso banco para saber se o usuário existe com a senha no nosso banco de dados, isto deverá ser feito através de uma camada de service.
Se a consulta conseguir localizar o usuário e a senha, o Spring security devolverá como resposta um Authorization com o prefixo Basic + um token.

Este token ficara registrado na nossa nossa aplicação na camada de Security, e apenas por meio desta chave o usuário poderá consumir a nossa Api.

Vamos entender o um pouco sobre este fluxo



1. A camada Basic Security Config é uma uma classe de configuração, é ela que configurar o tipo de criptografia que será utilizada na senha, qual o tipo de segurança que utilizaremos (no nosso caso o basic), e também quais End Point's que serão liberados para que usuário possa acessar, como por exemplo End Point de Logar e Cadastrar.

2. O Controller que rebeberá o Usurio com seu login e sua senha, e devolverá o recurso com a Authorization Basic + token.
3. User Details Service Camada responsável diretamente para validação da senha e criação do Basic token.
4. User Details é uma camada que contem todos os métodos que podemos chamar no User Details Service e em outras camadas.
5. Usuario Repository é a camada que ja existia no projeto de ecommerce, porem dentro dela fizemos um findByusuario, atravé deste Method Query podemos implementar a consulta do login.

Agora amos ver como isto fica no código...

Implementando segurança no nosso projeto.

Antes de tudo vamos adicionar duas dependências no nosso pom para termos acesso a todas as funcionalidades do Spring Security, a primeira dependência será a do prpria Spring Security:

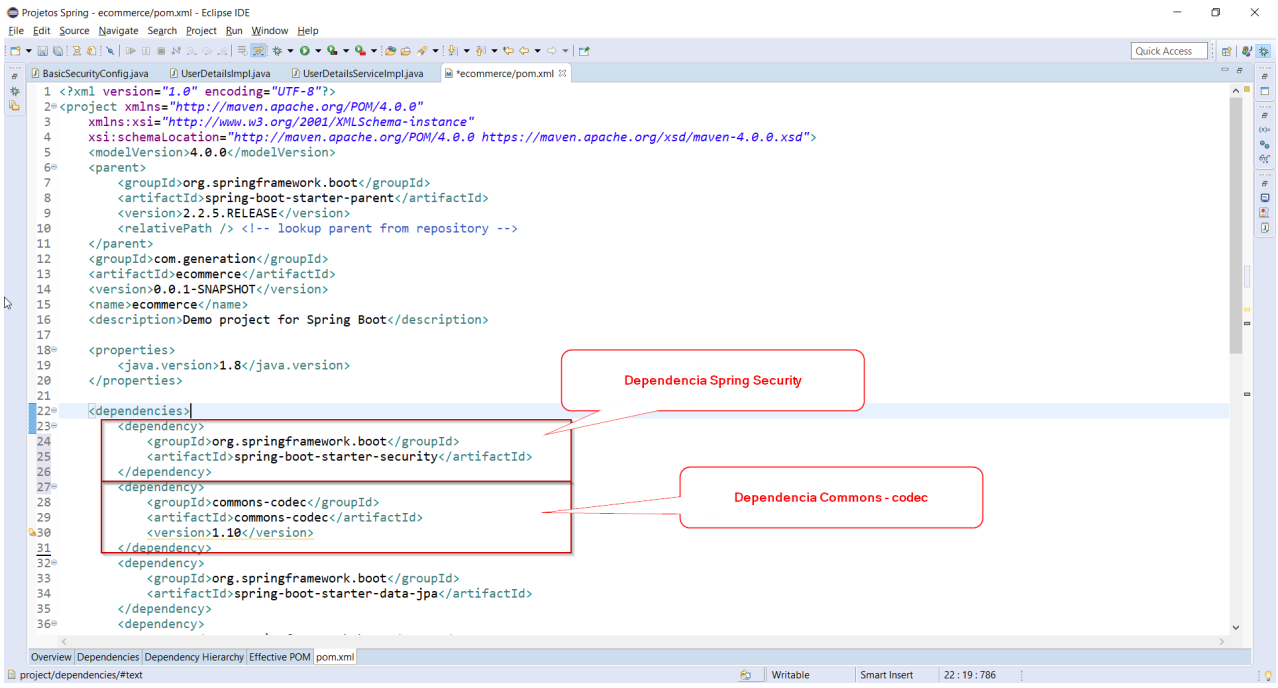
Basta pegar esta dependência e colar no pom.xml do nossa aplicação.

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

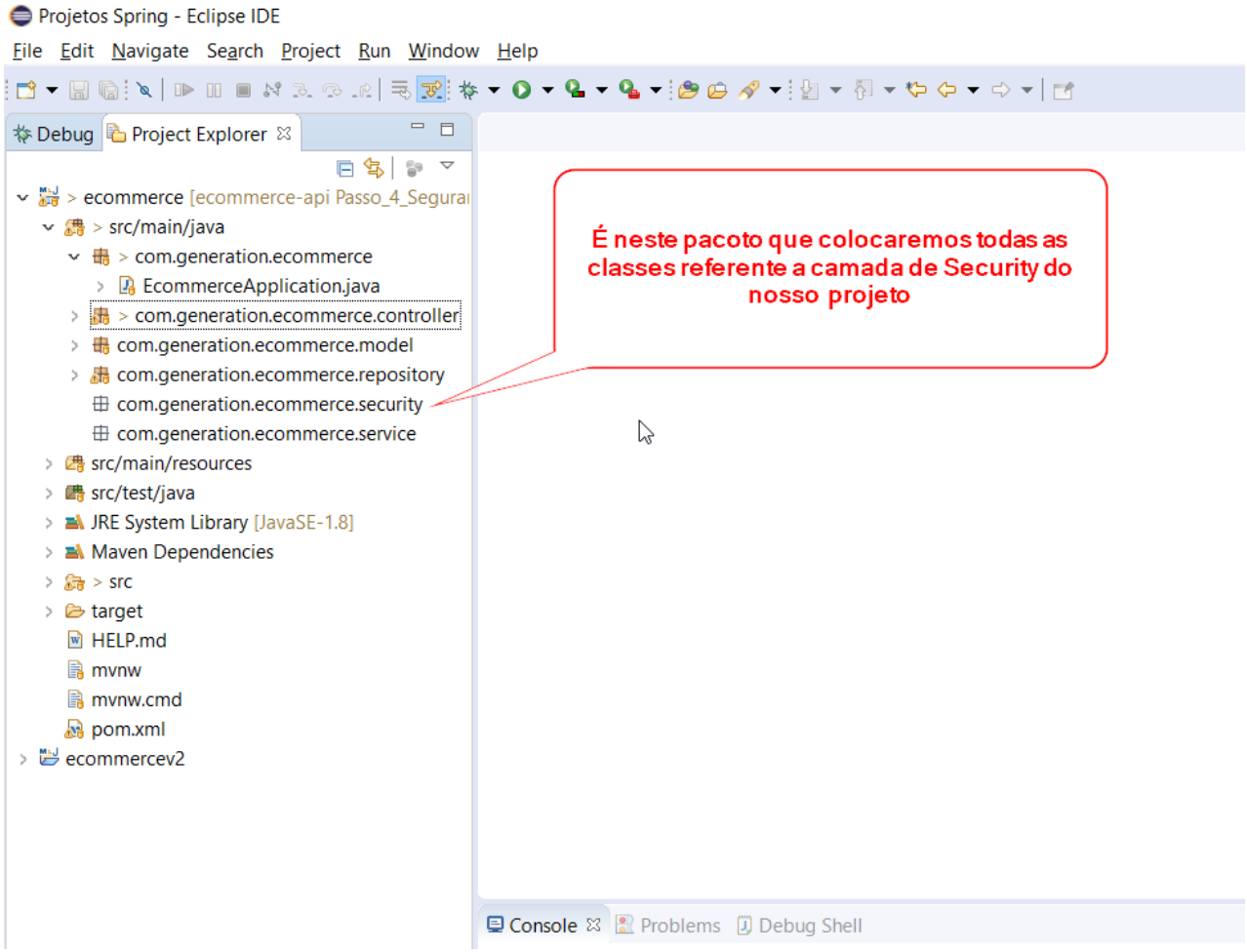
e uma outra dependência que nos ajudará com a criptografia das senhas Commons Codec.

```
<dependency>
        <groupId>commons-codec</groupId>
        <artifactId>commons-codec</artifactId>
        <version>1.10</version>
</dependency>
```

Nosso pom.xml ficará assim:



Bom agora que já entendemos todas as camadas necessárias para implementar a segurança do nosso projeto e já adicionamos todas as dependências necessárias, vamos por a mão na massa.
Para isso trabalharemos com um packed especifico para trabalhar com a segurança o packed **security**.



1. **Basic Security Config**
   Vamos criar a classe de configuração para definir algumas configurações de segurança do nosso projeto, esta classe levará o nome de BasicSecurityConfig.

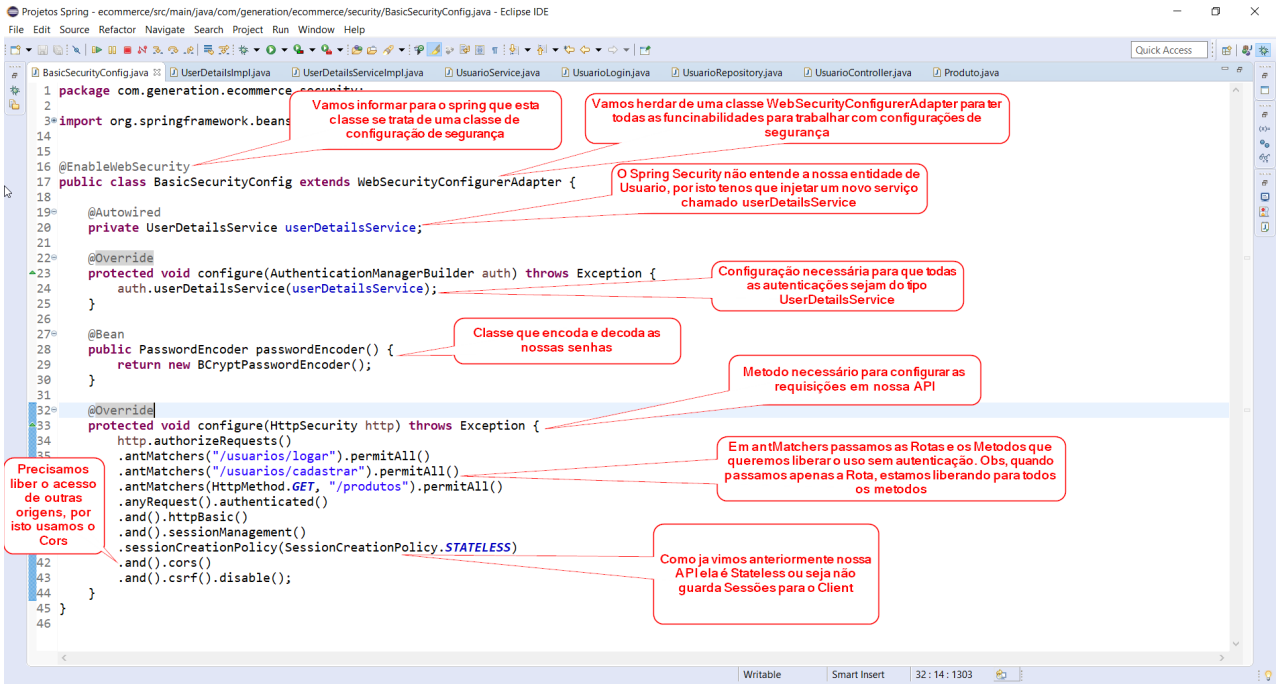
```
package com.generation.ecommerce.security;
```

```
org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerA

import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;


@EnableWebSecurity
public class BasicSecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
```

```
    private UserDetailsService userDetailsService;

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(userDetailsService);
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
        .antMatchers("/usuarios/logar").permitAll()
        .antMatchers("/usuarios/cadastrar").permitAll()
        .antMatchers(HttpMethod.GET, "/produtos").permitAll()
        .antMatchers("/produtos/porId/{id}").permitAll()
        .antMatchers(HttpMethod.GET, "/categorias").permitAll()
        .anyRequest().authenticated()
        .and().httpBasic()
        .and().sessionManagement()
        .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
        .and().cors()
        .and().csrf().disable();
    }
}
```

Vamos entender o código:



O código `.antMatchers(HttpMethod.GET, "/produtos").permitAll()` esta restringindo o end point produtos, apenas para os método Get, para os demais métodos neste end point só será possível, por meio de uma autenticação.

2. **User Details Impl**

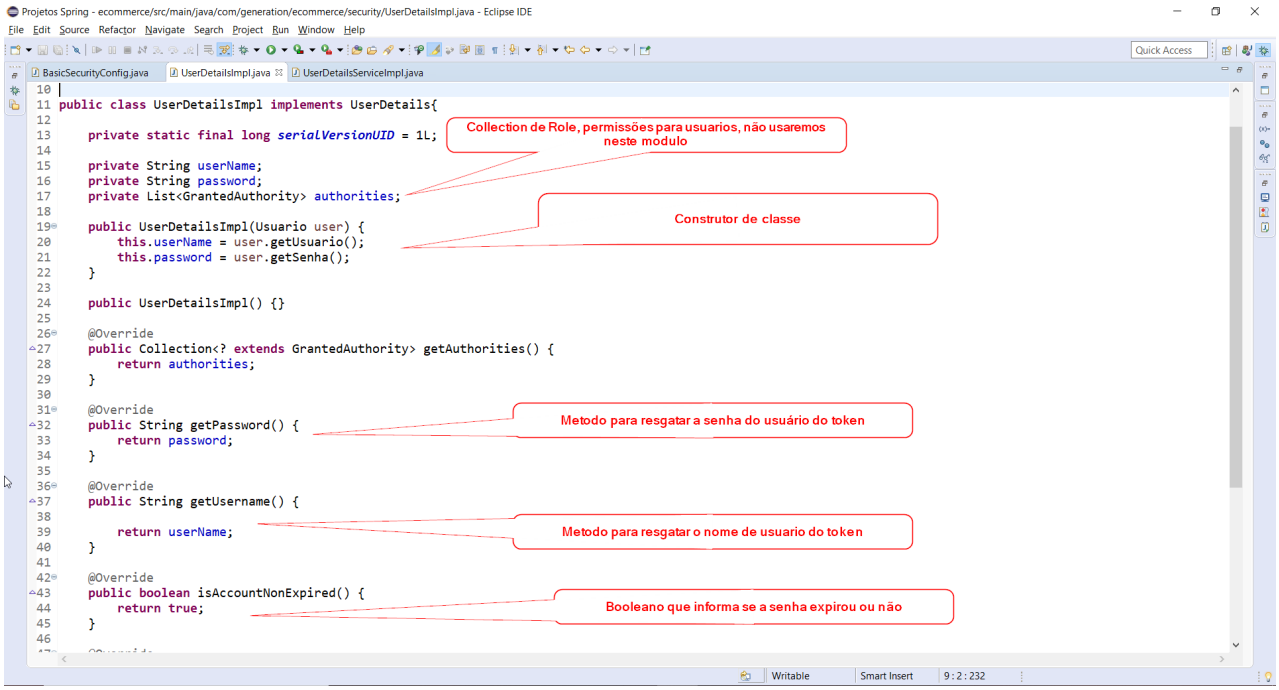
   Vamos expor todas as funcionalidades que a iremos atribuir para o UserDetailsImpl.

```
package com.generation.ecommerce.security;
```

```
import java.util.Collection;
import java.util.List;

import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

import com.generation.ecommerce.model.Usuario;

public class UserDetailsImpl implements UserDetails{
    private static final long serialVersionUID = 1L;

    private String userName;
    private String password;
    private List<GrantedAuthority> authorities;

    public UserDetailsImpl(Usuario user) {
        this.userName = user.getUsuario();
        this.password = user.getSenha();
    }

    public UserDetailsImpl() {}

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

    @Override
    public String getPassword() {
        return password;
    }

    @Override
    public String getUsername() {

        return userName;
    }

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

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

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

    @Override
    public boolean isEnabled() {
```

```
        return true;
    }
}
```

Vamos entender o código:



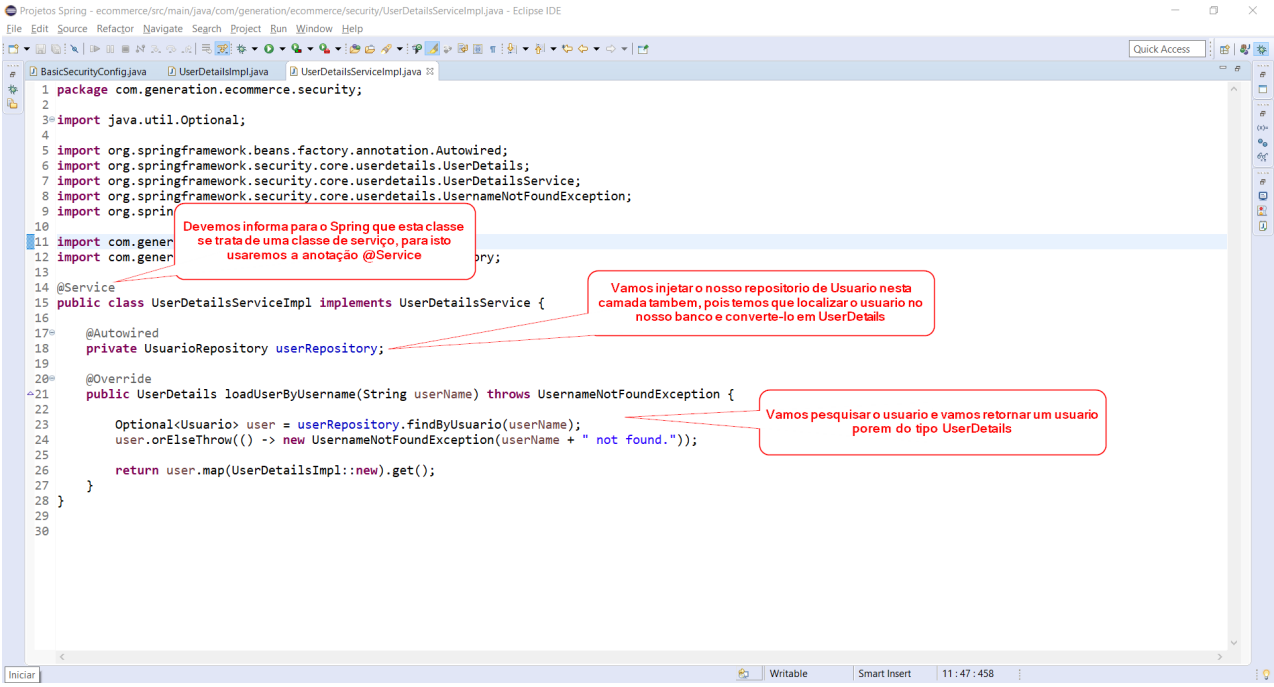
3. **User Details Service Impl**

   Pronto! agora vamos criar de fato a nossa classe de User Datails Service, que se encarregara de receber o nosso usuário e converter para User Details.

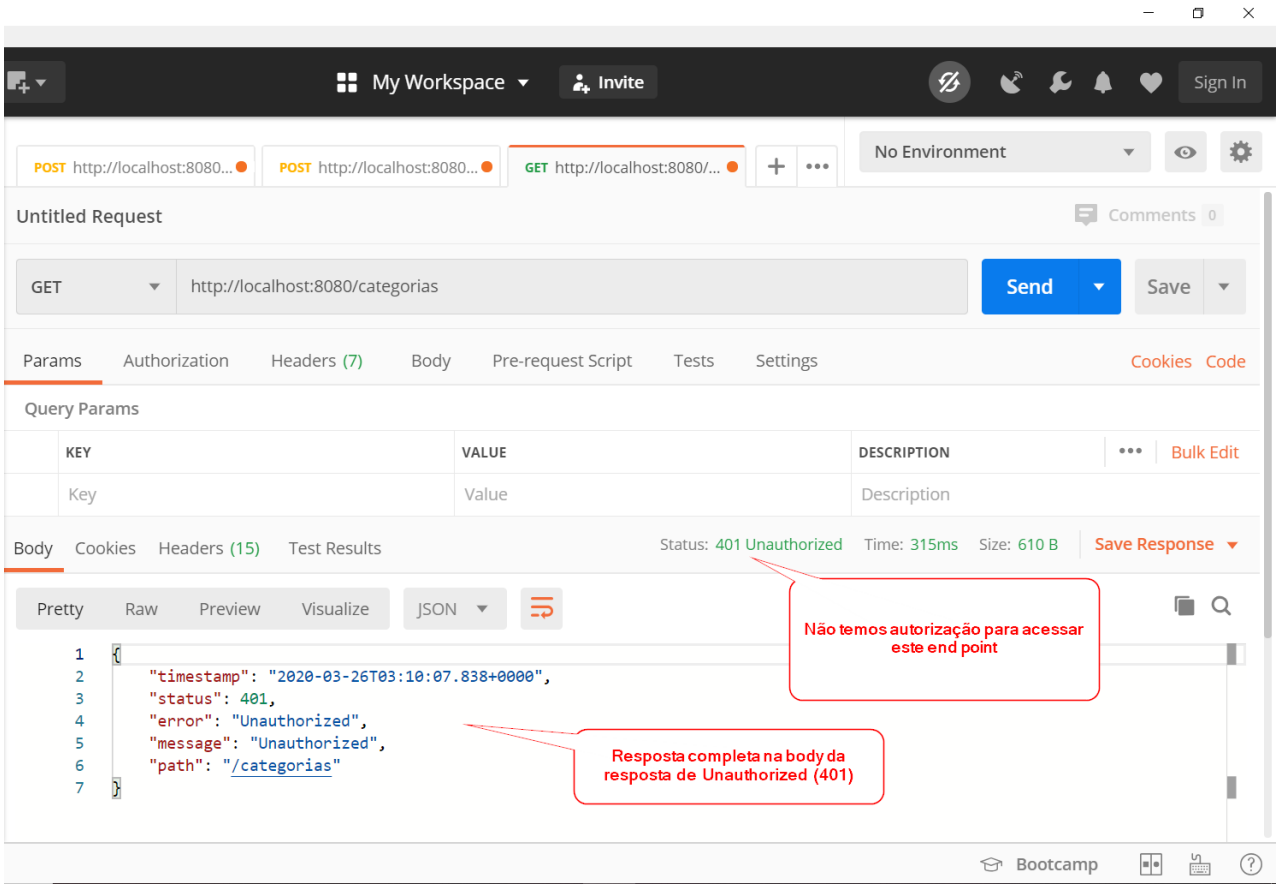
```
package com.generation.ecommerce.security;
```

```
import java.util.Optional;
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 com.generation.ecommerce.model.Usuario;
import com.generation.ecommerce.repository.UsuarioRepository;

@Service
public class UserDetailsServiceImpl implements UserDetailsService {

    @Autowired
    private UsuarioRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String userName) throws UsernameNotFoundException {

        Optional<Usuario> user = userRepository.findByUsuario(userName);
        user.orElseThrow(() -> new UsernameNotFoundException(userName + " not found."));

        return user.map(UserDetailsImpl::new).get();
    }
}
```

Vamos intender o código:



Vamos tentar acessar o end point de categorias via postman para ver o que receberemos como resposta.



Pronto! Apensa com estas configurações a nossa Api ja esta protegida, agora precisamos implementa um Servico para que o usuario possa se logar e se cadastrar. Então vamos la...

Login e Cadastro de Usuários

1. **Usuario Service**
   Como falamos anteriormente precisamos de implementar um serviço que logue e cadastro os nossos usuários de acordo com a ***Regra de negócio*** da camada de segurança da nossa Api, então dentro do pacote Service iremos criar uma classe UsuarioService.

A classe ficá assim:

```
package com.generation.ecommerce.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.generation.ecommerce.repository.UsuarioRepository;

@Service
public class UsuarioService {

    @Autowired
    private UsuarioRepository repository;

}
```

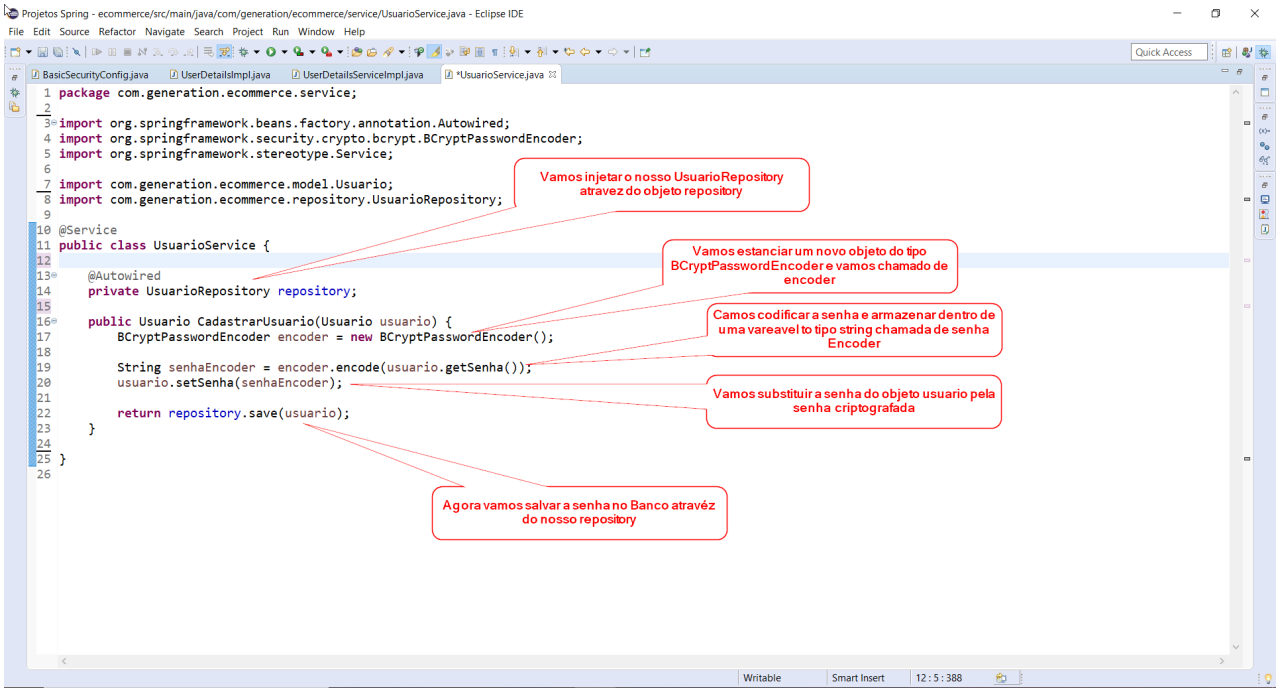
- ***A anotação `@Service` indica ao Spring que esta classe se trata de uma classe de serviço***
- ***Precisamos injetar um objeto do tipo `UsuarioRepository` vamos chama-lo de repository, ele servirá tanto para cadastrar nossos usuários quando para localiza-los na nossa base de todas e gerar o token.***

Pronto, agora que já temos a classe e o objeto que servirá para fazer as consultas, vamos implementar os métodos de ogar e de cadastrar.

1.1 **Cadastrar**
Insira o método abaixo dentro da classe `UsuarioService` depois do `private UsuarioRepository repository;`

```
public Usuario CadastrarUsuario(Usuario usuario) {
        BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();

        String senhaEncoder = encoder.encode(usuario.getSenha());
        usuario.setSenha(senhaEncoder);

    return repository.save(usuario);
}
```

Vamos entender o código:



1.2 **Logar**

Bom, para que o client possa enviar apenas as informações referente **autenticação** precisamos criar uma model nossa, vamos chama-la de `UsuarioLogin`, é uma model simples, sem as anotações do spring, ela não terá nenhuma interação com a base de dados, utilizaremos ela apenas para fazer o **Request**(requisição) e o **Response**(resposta) da nossa Api a Model ficará assim:

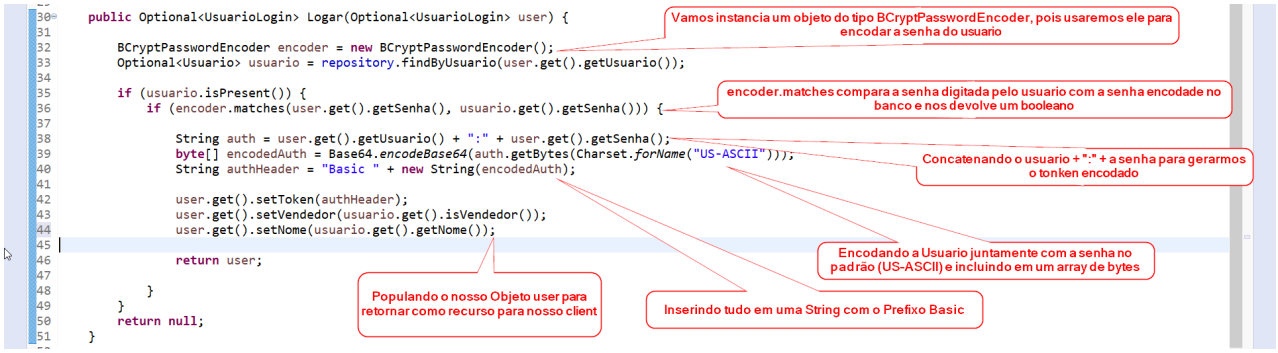
```
package com.generation.ecommerce.model;

public class UsuarioLogin {

    private String nome;

    private String usuario;

    private String senha;

    private String token;

    private boolean vendedor;

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getUsuario() {
        return usuario;
    }

    public String getToken() {
        return token;
```

```
    }

    public void setToken(String token) {
        this.token = token;
    }

    public boolean isVendedor() {
        return vendedor;
    }

    public void setVendedor(boolean vendedor) {
        this.vendedor = vendedor;
    }

    public void setUsuario(String usuario) {
        this.usuario = usuario;
    }

    public String getSenha() {
        return senha;
    }

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

Pronto! Uma vez que já criamos a nossa model, vamos voltar nosso `UsuarioService` para criar o nosso método de ogar, ele ficará assim:

```
public Optional<UsuarioLogin> Logar(Optional<UsuarioLogin> user) {

    BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();
    Optional<Usuario> usuario = repository.findByUsuario(user.get().getUsuario());

    if (usuario.isPresent()) {
        if (encoder.matches(user.get().getSenha(), usuario.get().getSenha())) {

            String auth = user.get().getUsuario() + ":" + user.get().getSenha();
            byte[] encodedAuth = Base64.encodeBase64(auth.getBytes(Charset.forName("US-ASCII")
            String authHeader = "Basic " + new String(encodedAuth);

            user.get().setToken(authHeader);
            user.get().setVendedor(usuario.get().isVendedor());
            user.get().setNome(usuario.get().getNome());

            return user;

        }
    }
    return null;
}
```

Vamos entender o código:



Feito todo isto, o código completo ficará assim:

```
package com.generation.ecommerce.service;

import java.nio.charset.Charset;
import java.util.Optional;

import org.apache.commons.codec.binary.Base64;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.stereotype.Service;

import com.generation.ecommerce.model.Usuario;
import com.generation.ecommerce.model.UsuarioLogin;
import com.generation.ecommerce.repository.UsuarioRepository;

@Service
public class UsuarioService {

    @Autowired
    private UsuarioRepository repository;

    public Usuario CadastrarUsuario(Usuario usuario) {
        BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();

        String senhaEncoder = encoder.encode(usuario.getSenha());
        usuario.setSenha(senhaEncoder);

        return repository.save(usuario);
    }

    public Optional<UsuarioLogin> Logar(Optional<UsuarioLogin> user) {

        BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();
        Optional<Usuario> usuario = repository.findByUsuario(user.get().getUsuario());

        if (usuario.isPresent()) {
            if (encoder.matches(user.get().getSenha(), usuario.get().getSenha())) {

                String auth = user.get().getUsuario() + ":" + user.get().getSenha();
                byte[] encodedAuth = Base64.encodeBase64(auth.getBytes(Charset.forName("US-ASC
                String authHeader = "Basic " + new String(encodedAuth);

                user.get().setToken(authHeader);
                user.get().setVendedor(usuario.get().isVendedor());
```

```
                user.get().setNome(usuario.get().getNome());

                return user;

            }
        }
        return null;
    }
}
```

Pronto!!!

Agora precisamos Modificar o nosso UsuarioController, Vamos criar um end point para Logar, e vamos alterar o Método de cadastrar (Post)

- **Importante - Para logar precisamos passar na body o login e a senha, o método http utilizado para fazer isto é o método post**

Em UsuárioController a primeira coisa a ser feita é injetar o nosso UsuarioService:

```
@Autowired
private UsuarioService usuarioService;
```

e também iremos criar um end point Logar, ele ficará assim:

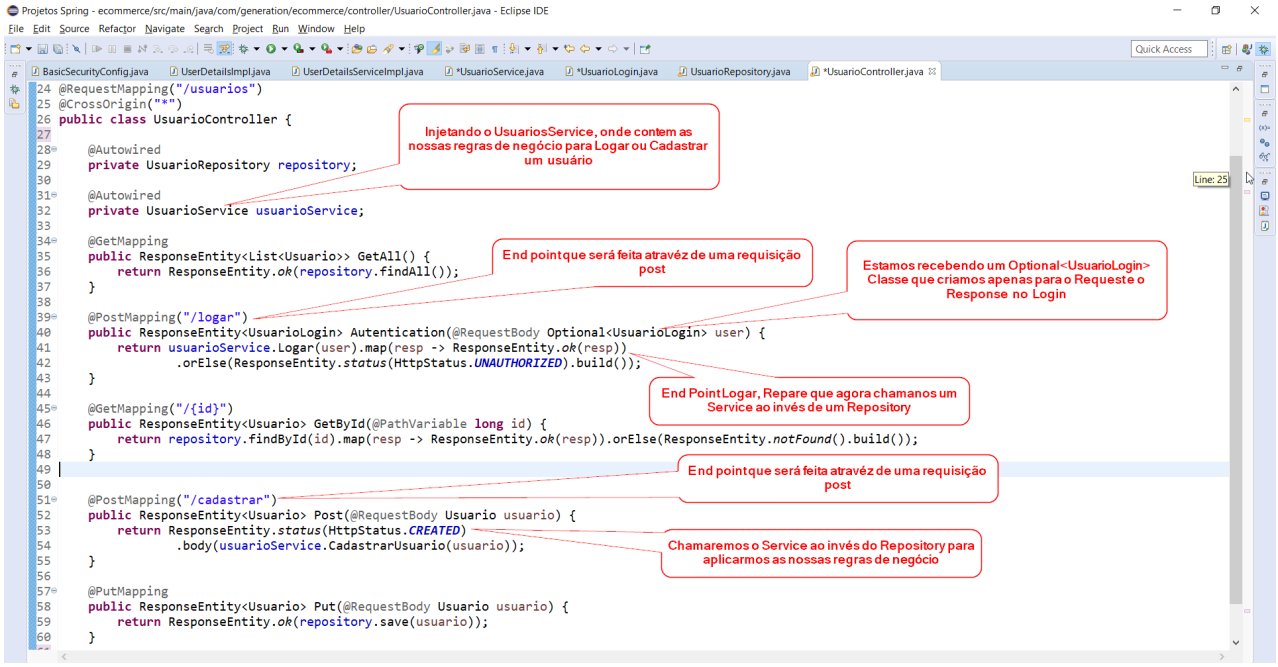
```
@PostMapping("/logar")
public ResponseEntity<UsuarioLogin> Autentication(@RequestBody Optional<UsuarioLogin> user) {
        return usuarioService.Logar(user).map(resp -> ResponseEntity.ok(resp))
                .orElse(ResponseEntity.status(HttpStatus.UNAUTHORIZED).build());
    }
```

Feito isto vamos implementar o nosso end point de cadastrar um usuário novo:

```
@PostMapping("/cadastrar")
public ResponseEntity<Usuario> Post(@RequestBody Usuario usuario) {
    return ResponseEntity.status(HttpStatus.CREATED)
            .body(usuarioService.CadastrarUsuario(usuario));
}
```

> *Repare que em ambos os end points utilizamos o service ao repository, temos que seguir esta arquitetura pois logar e cadastrar exige uma*
> ***regra de negócio*** *que seria garantir que a senha seja encriptada.*

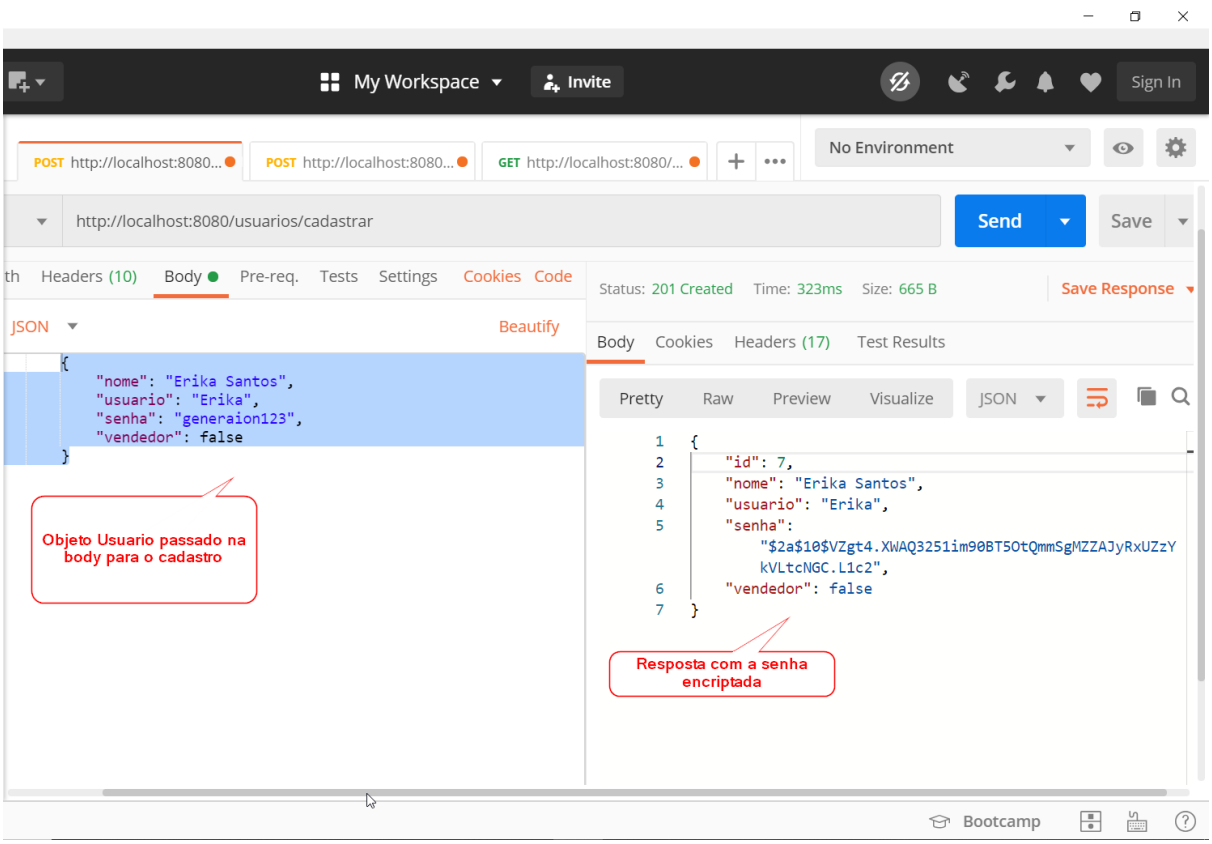
Vamos entender o código:



Pronto agora vamos testar via postman.

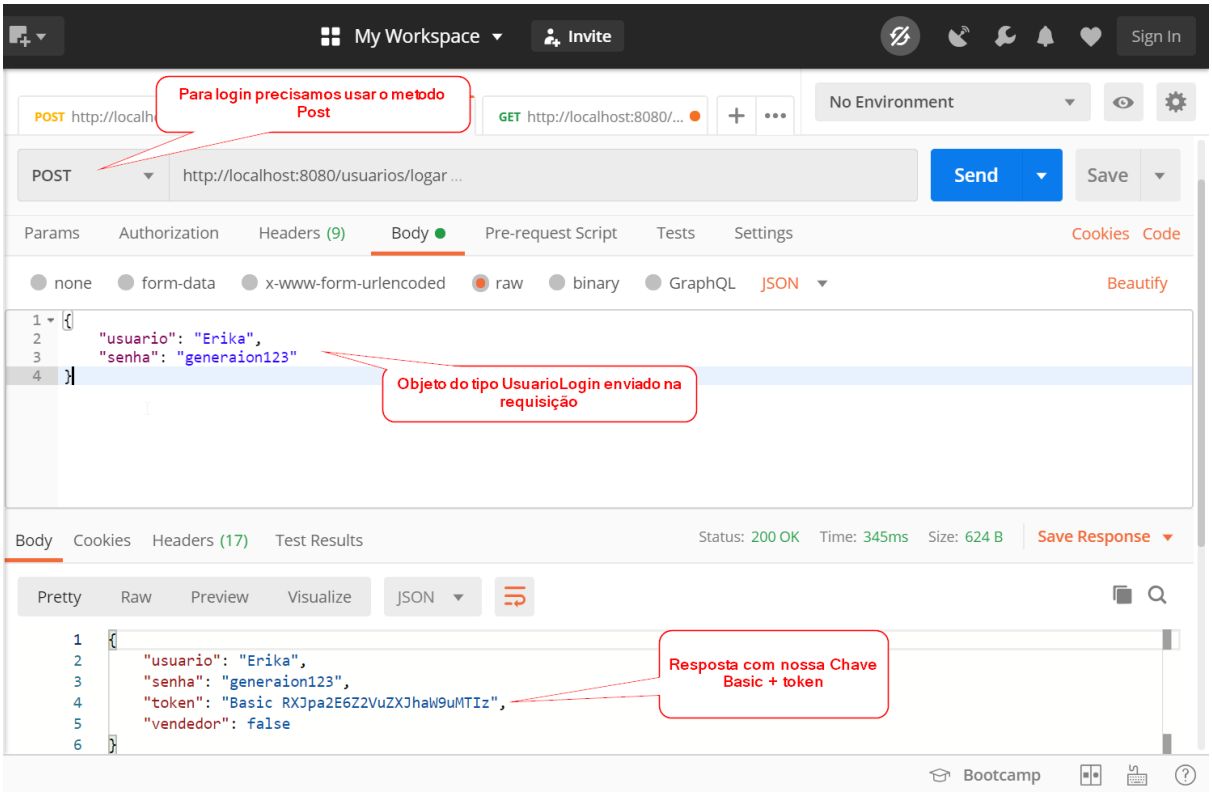
- Cadastro de usuário, agora vamos cadastrar um Usuário:
  Objeto Json passado na body da requisição

```
{
    "nome": "Erika Santos",
    "usuario": "Erika",
    "senha": "generaion123",
    "vendedor": false
}
```

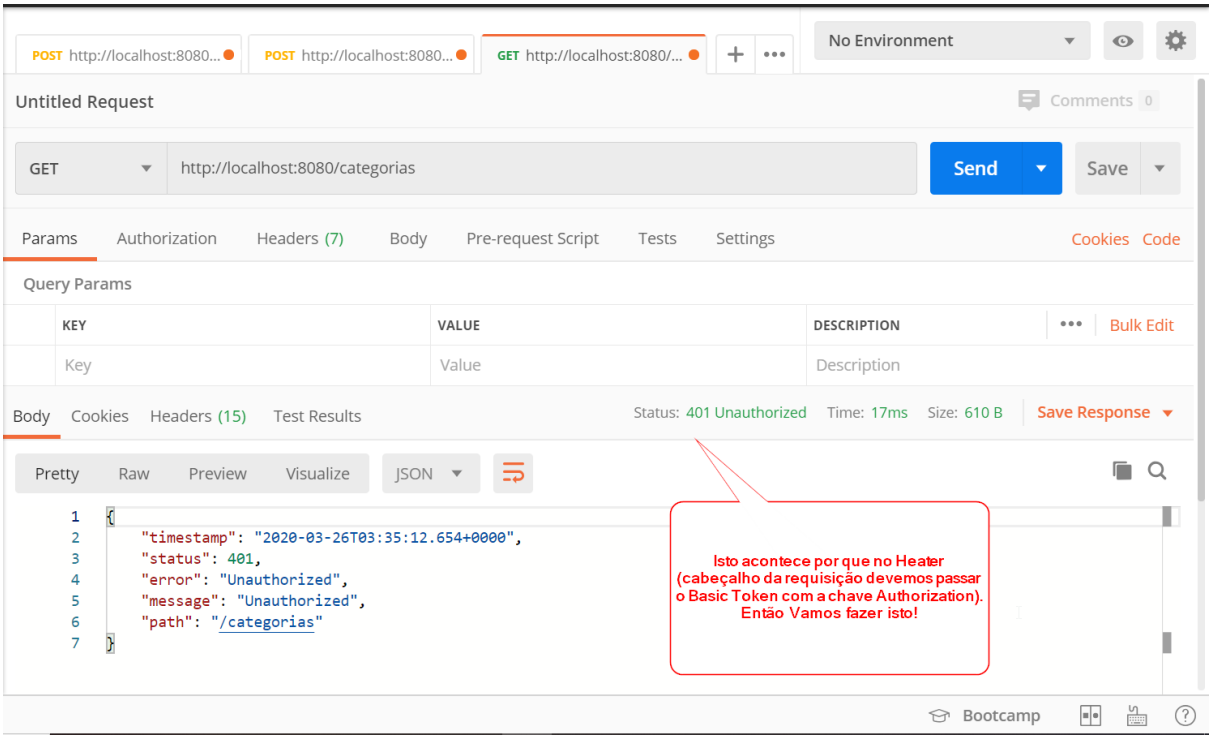


- Vamos logar este usuário para ver o que acontece
  Objeto Json passado na body da requisição

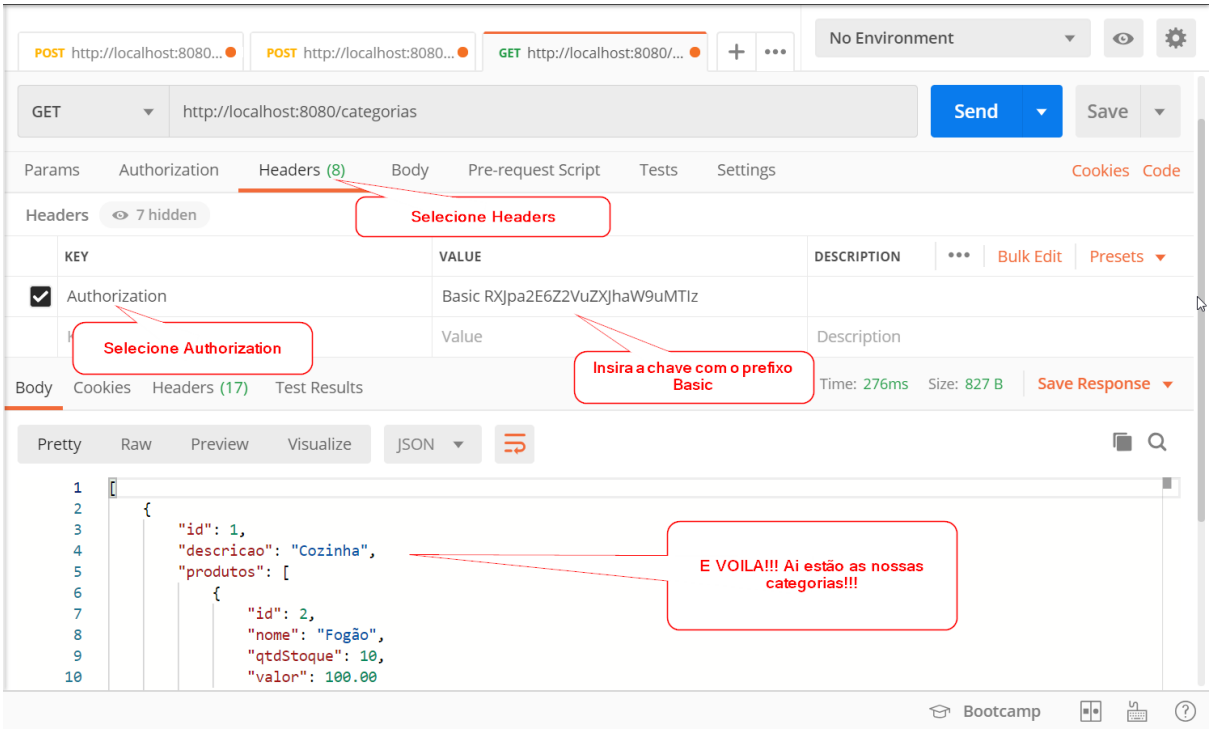
```
{
  "usuario": "Erika",
  "senha": "generaion123"
}
```



- Pronto! agora que já logamos, vamos tentar acessar o end point de cadastro novamente



- Vamos tentar fazer a mesma requisição só que desta vez iremos passar o Basic Token no Hearder

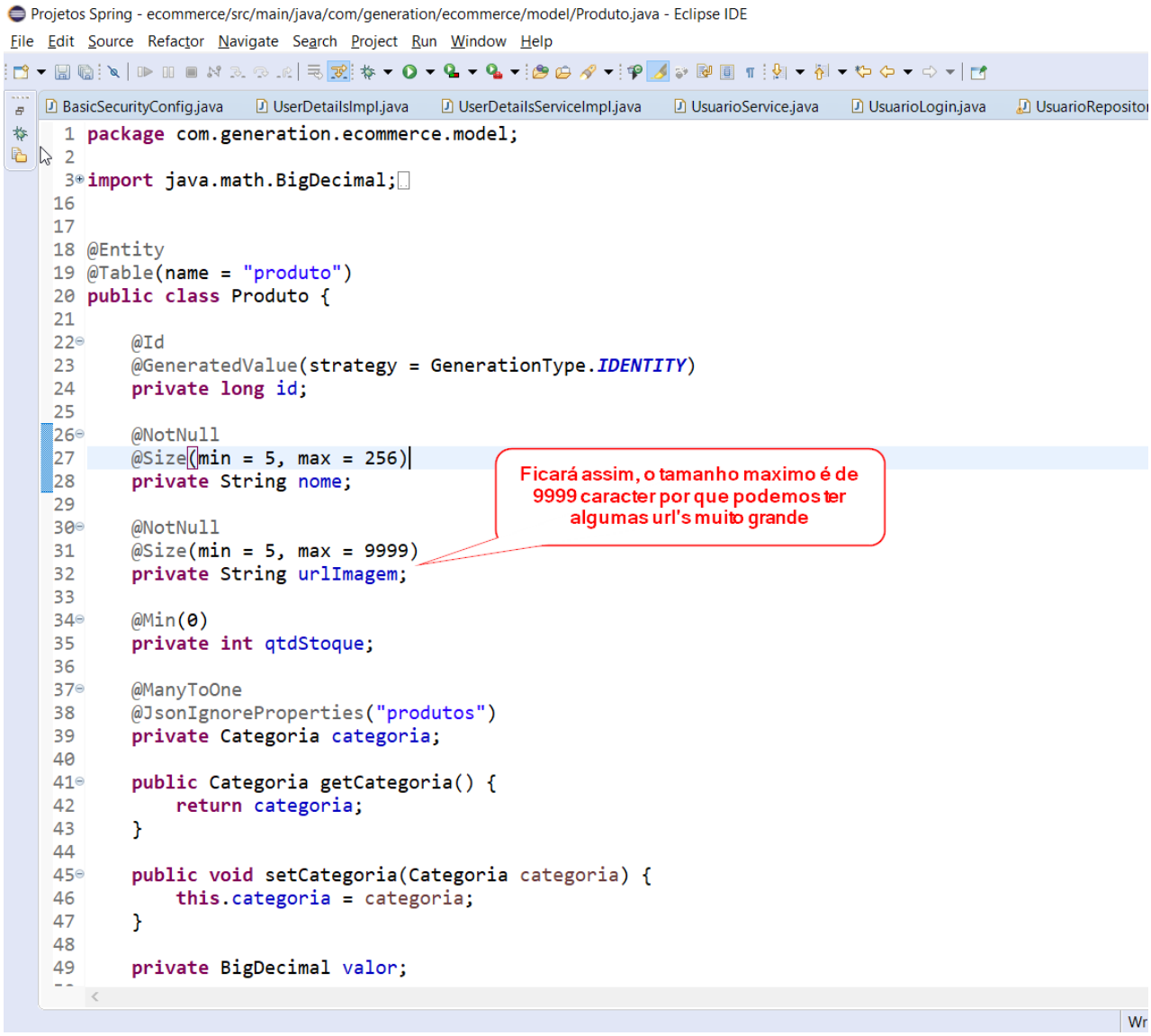


Antes de terminar vamos fazer um ultimo ajuste, na nossa Entidade Produtos não temos uma String para receber a URL da imagem do produto, então vamos fazer isto para que la no front end possamos carregar a imagem através de uma URL.
Vamos chamar este atributo de urlImagem do tipo String

```
@NotNull
@Size(min = 5, max = 9999)
private String urlImagem;
```

Importante- não se esqueça de gerar o get e o set

Ficará assim:



Até a próxima...