

# Minicurso Spring e Android

Criado por Gabriel Schaidhauer - Aula 3



Segurança

# Segurança

Em APIs privadas, um fator importante é manter essa privacidade. Para isso utilizam-se técnicas de segurança tais como autenticação, criptografia, etc.



# Autenticação

Autenticação é a habilidade de identificar o usuário que está acessando o sistema, para, desta forma, permitir ou bloquear certas funcionalidades de acordo com o usuário.



# Criptografia

Criptografia é a habilidade do sistema de trocar informações de maneira que só as pessoas autorizadas tenham a capacidade de lê-las.



# Spring Security

# Spring Security

Spring Security é um módulo do Spring o qual é usado para aplicação de técnicas com a finalidade de proteger uma aplicação, permitindo somente acessos autorizados aos seus Endpoints.

# Spring Security Configuration

A camada de segurança do Spring é configurada através de uma classe de configuração contendo as informações necessárias para tal

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(securedEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter{

    @Autowired
    private AuthenticationFilter filter;

    @Override
    protected void configure(HttpSecurity httpSecurity) throws Exception {
        httpSecurity
            .authorizeRequests()
            .anyRequest()
            .authenticated()
            .and()
            .addFilterBefore(filter, UsernamePasswordAuthenticationFilter.class);
    }
}
```



Meu deus! É muita coisa!



Calma! Vamos por partes...



# @Configure

Não custa explicar outra vez!

Esta anotação indica que a classe anotada se refere a uma classe de configurações do Spring.

# @EnableWebSecurity

Esta anotação habilita configurações específicas para web, permitindo a sua configuração através de `HttpSecurity`.

# @EnableGlobalMethodSecurity

Esta anotação permite que sejam realizadas configurações específicas por método, quando necessário.

# configure(HttpSecurity)

Este método permite a realização das configurações de segurança para chamadas através do protocolo HTTP. Estas configurações são realizadas através do encadeamento de chamadas de método na classe `HttpSecurity`.

# HttpSecurity

Neste curso veremos somente algumas das configurações possíveis através de HttpSecurity, sendo elas expressas da seguinte forma:

```
httpSecurity
.authorizeRequests()
.anyRequest()
.authenticated()
.and()
.addFilterBefore(filter, UsernamePasswordAuthenticationFilter.class);
```

# authorizeRequests()

Este método indica a abertura de um bloco para configuração de endpoints que exigirão ou não autenticação, permitindo configuração por endpoint, por papel do usuário na aplicação ou ainda bloqueando todos os recursos.



# anyRequest()

Este método indica que a configuração que for aplicada após este método será aplicada a qualquer endpoint que ainda não tenha sido configurado.

# authenticated()

Este método informa que as requisições realizadas para o recurso específico, ou aplicada a `anyRequest()` deverá ter uma autenticação.

# and()

Este método permite finalizar um tipo de configuração e iniciar outro.

# addFilterBefore()

Este método é bem importante para nós. Da forma como estão sendo aplicadas as configurações, é aqui que registramos a nossa autenticação personalizada. Este método permite adicionar um filtro personalizado à cadeia de filtros personalizados do spring.

# Filtro de Autenticação

# Filtro de Autenticação

Um filtro de autenticação é onde podem ser colocadas as lógicas de autenticação, e onde deve ser instanciado o objeto de autenticação que é adicionado ao contexto de segurança.

```
@Component
public class AuthenticationFilter extends GenericFilterBean {

    @Autowired
    private UsuarioService service;

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain)
        throws IOException, ServletException {
        HttpServletRequest request = (HttpServletRequest) servletRequest;
        final String authorization = request.getHeader("Authorization");

        if (authorization != null && authorization.startsWith("Basic")) {

            // Authorization: Basic base64credentials
            String base64Credentials = authorization.substring("Basic".length()).trim();

            String credentials = new String(Base64.getDecoder().decode(base64Credentials),
                Charset.forName("UTF-8"));

            // credentials = username:password
            final String[] values = credentials.split(":", 2);

            Usuario usuario = service.findUserByNomeAndSenha(values[0], values[1]);

            if (Objects.nonNull(usuario)) {
                SecurityContextHolder.getContext().setAuthentication(new AuthenticatedUser(usuario));
            }
        }

        filterChain.doFilter(servletRequest, servletResponse);
    }
}
```

# doFilter()

É o método invocado no momento em que é realizada uma chamada HTTP que irá passar pelo fluxo de filtros.

# ServletRequest

É o objeto que contém os dados fornecidos na requisição, tais como headers, etc.



# ServletResponse

É o objeto que representa a resposta que será fornecida ao usuário.

# FilterChain

É o objeto que representa a cadeia de filtros da aplicação, e é usado para indicar quando as ações do filtro terminaram e o fluxo pode seguir.

# SecurityContext

O contexto de segurança é quem informa ao Spring que há uma autenticação. Para que isso ocorra deve ser registrado no contexto de segurança algum tipo de autenticação indicando que o usuário está autenticado e fornecendo certos dados sobre ele. Isto é feito da seguinte forma:

```
SecurityContextHolder.getContext().setAuthentication(new  
AuthenticatedUser());
```

# AuthenticatedUser

Na implementação que criamos a nossa autenticação é um objeto criado por nós implementando Authentication e que possui um AuthenticatedPrincipal igualmente criado por nós.

```
public class AuthenticatedUser implements Authentication {  
    private AuthenticatedPrincipal principal;  
  
    public AuthenticatedUser (Usuario usuario) {  
        this.principal =  
            new AuthenticatedPrincipal(usuario);  
    }  
  
    @Override  
    public Collection<? extends GrantedAuthority> getAuthorities() { return null; }  
  
    @Override  
    public Object getCredentials() { return null; }  
  
    @Override  
    public Object getDetails() { return null; }  
  
    @Override  
    public AuthenticatedPrincipal getPrincipal() { return principal; }  
  
    @Override  
    public boolean isAuthenticated() { return true; }  
  
    @Override  
    public void setAuthenticated(boolean b) throws IllegalArgumentException {}  
    @Override  
    public String getName() { return null; }  
}
```

# Authenticated Principal

Esta é a classe que contém os dados do nosso usuário, e que pode ser injetada na nossa controller por exemplo para poder utilizar dados do usuário logado nas nossas regras de negócio.

```
public class AuthenticatedPrincipal implements UserDetails {  
    private Usuario usuario;  
  
    public AuthenticatedPrincipal(Usuario usuario) { this.usuario = usuario; }  
  
    @Override  
    public Collection<? extends GrantedAuthority> getAuthorities() { return null; }  
  
    @Override  
    public String getPassword() { return null; }  
  
    @Override  
    public String getUsername() { return null; }  
  
    @Override  
    public boolean isAccountNonExpired() { return false; }  
  
    @Override  
    public boolean isAccountNonLocked() { return false; }  
  
    @Override  
    public boolean isCredentialsNonExpired() { return false; }  
  
    @Override  
    public boolean isEnabled() { return false; }  
  
    public Usuario getUsuario () { return usuario; }  
}
```

Injetar?



# @AuthenticationPrincipal

Lembram do @Autowired que utilizamos anteriormente para que as nossas services e repositories fossem instanciados pelo spring para a gente na nossa classe ou nos nossos construtores? Então, o @AuthenticationPrincipal funciona de forma parecida. Podemos inserí-lo como parâmetro nos métodos das nossas controllers, e desta forma o Spring irá instanciar ali o AuthenticatedPrincipal que passamos para o nosso SecurityContext.

# Consumo de serviços REST no Backend



# RestTemplate

Para consumo de serviços REST no nosso backend podemos usar a classe RestTemplate, fornecida pelo Spring, a qual realiza a chamada na api e retorna o dado que queremos na classe informada no segundo parâmetro.

```
@Component
public class RandomUserServiceConsumer {

    private RestTemplate restClient;

    public RandomUserServiceConsumer(){
        this.restClient = new RestTemplate();
    }

    public String getRandomUser() {
        String result = restClient.getForObject(
            url: "https://randomuser.me/api/",
            String.class);
        return result;
    }
}
```

# Tarefa para casa

Vocês irão aplicar as técnicas apresentadas na aula de hoje para proteger a API de vocês, de forma a garantir que o usuário somente poderá acessar os recursos de vocês caso tenha autorização para tal.

Dúvidas?

