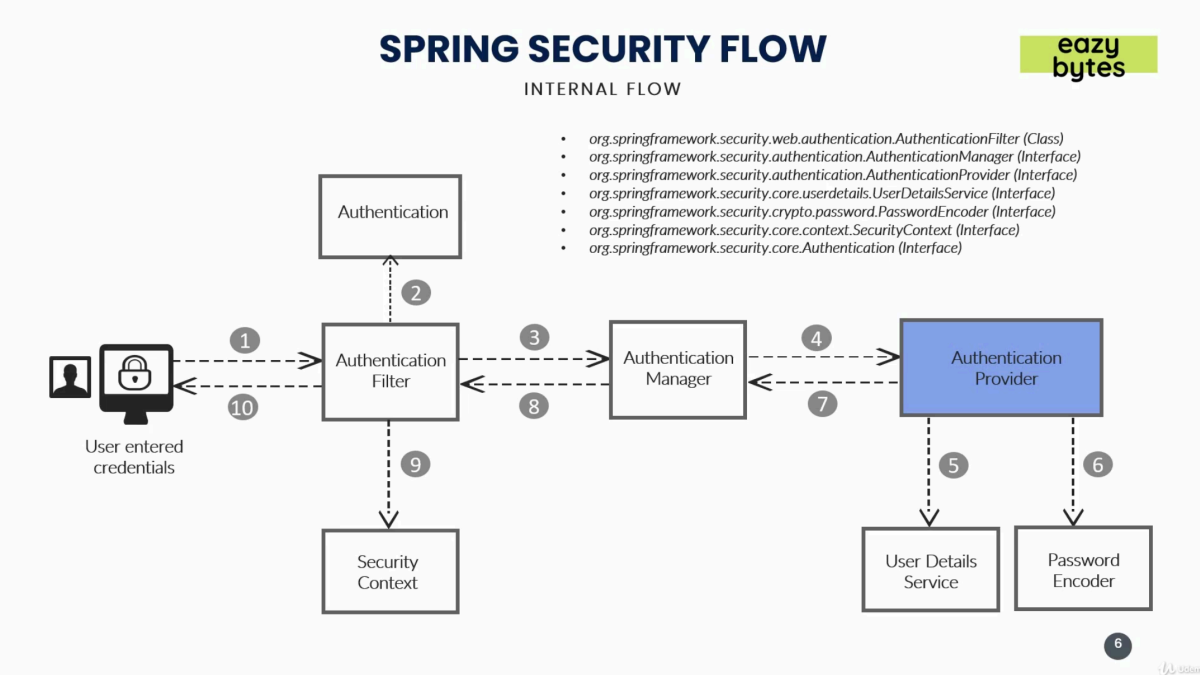
Spring Security



Flow

IDA

1. Request
2. Filter (filter chain)
3. AuthManager
4. AuthProvider (Usara el userDetailsService y passwordEncoder para autenticar)
5. UserDetails y PasswordEncoder

VUELTA

1. AuthProvider
2. AuthManager
3. Filter (set un Context, con la autenticacion realizada)
4. Response

Partes de Spring Security

== Config ==

- Security Config

== Authentication ==

- Authentication
- AuthenticationManager
- AuthenticationProvider

== User ==

- UserDetails Manager (CRUD y methods)
- Custom UserDetails (username, password, etc...)
- Custom UserDetailsService (loadUserByUsername())

Hay 2 grandes procedimientos

UserDetailsService o **AuthenticationProvider**. Ambos se encargaran de realizar una autenticacion completa con passwordEncoder y buscar en la DB.

El **primero** se usa en caso de usar unicamente Username y Password para la autenticacion.
El **segundo** se usa si necesitamos otro tipo de autenticacion como Huella Digital, Reconocimiento Facila, OTP, etc...

== UserDetailsService ==
Este procedimiento se usa en caso de que el metodo de autenticacion sea por Username y Password.

Se trata de crear CustomDetails y CustomService, sin tocar el otro bloque de AuthenticationProvider.
Usaremos este metodo para Override el USER(y sus atributos) y el SERVICE(loadUserByUsername) que vienen dados por defecto con Spring Security.

Crear CustomUserDetails
Crear CustomUserDetailsService

== AuthenticationProvider ==
Este segundo metodo se utiliza cuando necesitamos cambiar el tipo de autenticacion (Huella digital, Reconocimiento facial, OTP, etc...)
Crearemos una clase que implemente AuthenticationProvider y haremos Override a sus dos metodos

authenticate()
supports()

UserDetails

SpringSecurity trae por defecto un objeto USER con distintos atributos:

- *Username*
- *Password*
- *isAccountNonExpired*
- *isAccountNonLocked*
- *isCredentialsNonExpired*
- *isEnabled*

Segun necesitemos podemos Override cualquiera de dichos atributos. Creamos una nueva Class que implementara UserDetails y Override todos sus metodos.

Ejemplo: Nuestro UserEntity tendra Email en vez de Username.

```java
public class SecurityUser implements UserDetails {

    private final TestEntity testEntity;

    public SecurityUser(TestEntity testEntity) {
        this.testEntity = testEntity;
    }

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

    @Override
    public String getUsername() {
        return testEntity.getEmail();
    }

    @Override
    public String getPassword() {
        return testEntity.getPassword();
    }
```

UserDetailsService

SpringSecurity nos trae por defecto un Service con un method llamado *loadUserByUsername()*, quien se encargara de retornar un *UserDetails*.
Podemos Override para rellenar dicho UserDetails con la informacion que tengamos (ej: UserEntity en Database).

Asi mismo podemos agregar cuantos metodos querramos (SignupUser, ExistsUserInDatabase, etc...)

Ejemplo: findByEmail el Username enviado por Controller.

Repository

```java
@Repository
public interface TestRepository  extends JpaRepository<TestEntity, Integer> {

    List<TestEntity> findByEmail(String email);

}
```

UserDetailsService (Custom)

```java
@Service
public class CustomUserDetailsService implements UserDetailsService {

    @Autowired
    private TestRepository testRepository;

    // === ACA: Otro metodo si necesitamos ===
    // Ejemplo: signupNewUser(UserDTO newUser){}


    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        List<TestEntity> myList = testRepository.findByEmail(username);
        if(myList.isEmpty()){
            throw new UsernameNotFoundException("User Details NOT FOUND for username: " + username);
        }
        return new SecurityUser(myList.get(0));
    }
}
```

Spring Configuration

Habiendo modificado el UserDetails y UserDetailsService, debemos declarar en nuestras Confugartions, que queremos usar estas nuevas clases.

```java
@Configuration
public class ProjectSecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private CustomUserDetailsService customServ;

    // == AuthenticationManager by DEFAULT ===
    @Override
    @Bean
    public AuthenticationManager authenticationManagerBean() throws Exception {
        return super.authenticationManagerBean();
    }

    // === PasswordEncoder ===
    // Encriptaremos nuestras Passwords.

    // === Custom UserDetailsService ===
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception{
        auth.userDetailsService(customServ);
    }

    // === Main Config ===
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests() ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlReg
                .antMatchers("/test").authenticated()
                .antMatchers("/auth/login").permitAll()
                .and() HttpSecurity
            .formLogin().and()
            .httpBasic();
    }

}
```

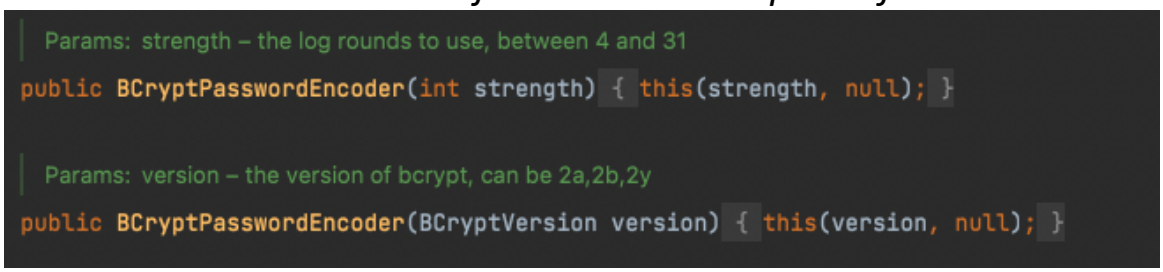
Recomendacion:

Hasta aca conviene comprobamos junto con la DB que todo funciona. Entramos a *localhost:8080* y nos abre el formulario, le damos username y password (mismos que en la DB) y nos redirige a la app.

PasswordEncoder

Veremos principalmente el uso BCrypt como encoder recomendado. Al ser tan conocido, la gente fue descifrandolo con mayor facilidad, por esto tenemos hoy en dia el SCrypt como "reemplazo"(contiene CPU cost, Memory cost, etc... haciendolo mas dificil de hackear).

Podemos pasar la version o el strength en el constructor, aunque es recomendado dejarlo como viene por defecto.



Como Funciona?

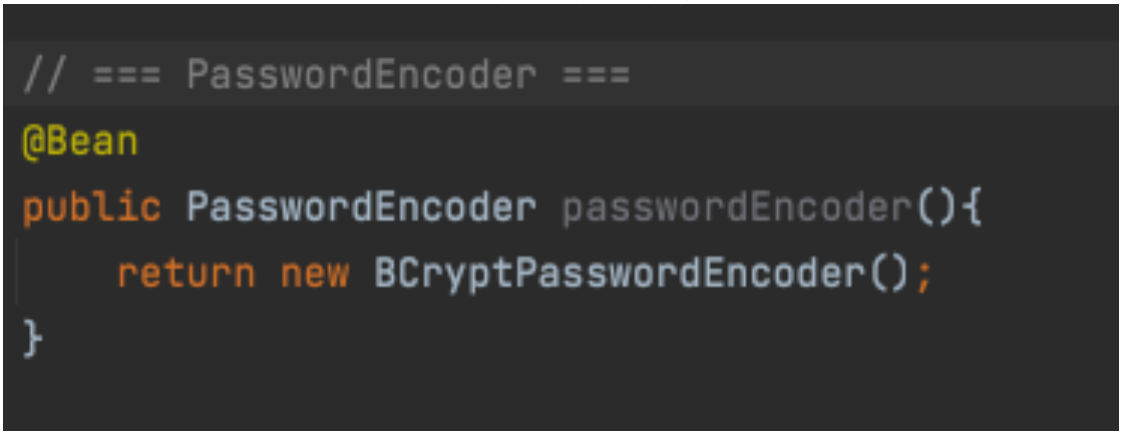
BCrypt llamara a sus metodos *encode()* y *matches()*.

encode() -> Recibe un String y lo encodea con la version y strength seleccionada, devuelve String.
matches() -> Recibe rawPassword y encodedPassword. Las va a comparar y devolvera un Boolean. Hashea la rawPassword obtenida en el Controller.

Porque SCrypt?

Por defecto: SCryptPasswordEncoder(16384, 8, 1, 32, 64).
Parametros:
cpuCost, memoryCost, maxParallel, keyLength, saltLength.
No solo cuesta Tiempo en resolver, sino que tambien memoria de procesado.

En Nuestra SpringConfiguration:



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

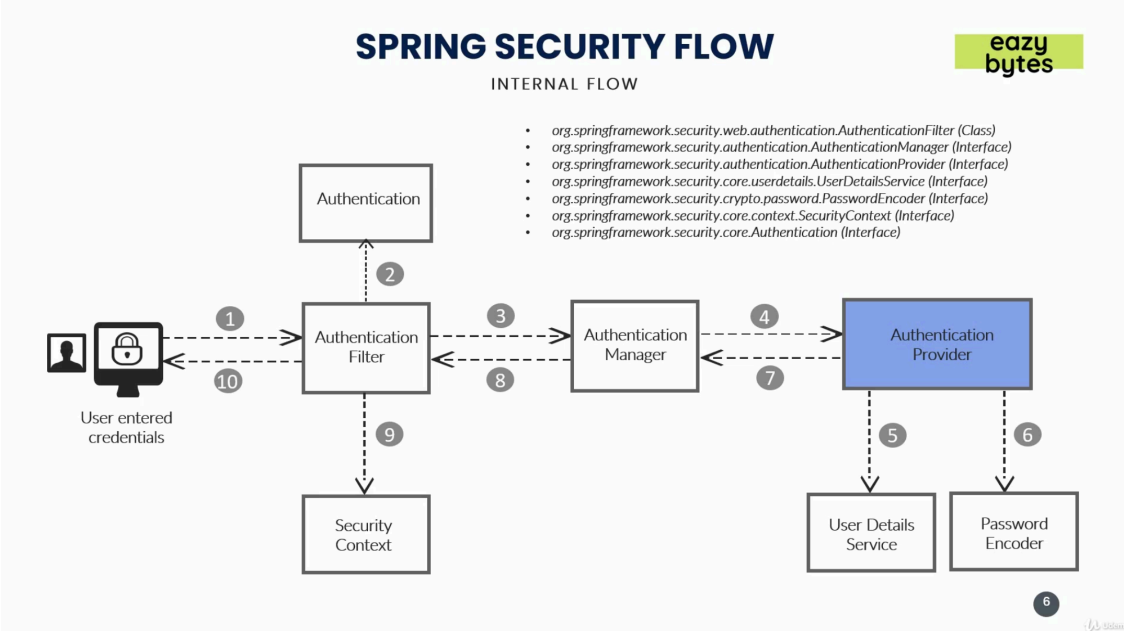
Al buscar en la DB usara el matches(). Nuestra password en DB deberia ser encriptada al estilo BCrypt. Para imitar la Bcrypt usamos:

https://bcrypt-generator.com/

Authentication in SpringSecurity

Hasta ahora pudimos modificar UserDetailsService, PasswordEncoder y la Security Configuration.
Nos Falta ver todo sobre Authentication, quien obviara estas configurations de ser necesario.

Recordemos el Flow...



Flow

1. Http Request -> From UI
2. Filter -> intercepta Request (1) y la transforma en un Authentication Object (2) y lo pasa (3).
3. AuthenticationManager -> Pasa el AuthObj al AuthProvider (4).
4. AuthenticationProvider -> Por defecto utilizara UserDetailsService y PasswordEncoder para Validate el user (5 y 6).

Cuando **Override** AuthenticationProvider?

Podemos modificar el comportamiento de AP cuando nuestro sistema de login no sea con Username y Password.
Ejemplos: Fingerprint Auth, OTP Code, etc...
Para esto crearemos un AP por cada metodo existente de autenticacion.Podemos tener tantos AuthProvider como querramos.

AuthenticationManager se encargara de llamar al Provider necesario para realizar el login. (Recibira un formato de login y lo enviara segun corresponda a su AP).

Authentication Provider

Se encargara de la Logica de la autentication.
Como se llega hasta aca?
Filter transforma la Request en AuthenticationObject y el AuthenticationManager lo pasa al AuthenticationProvider correspondiende segun el tipo de autenticacion, quien efectuara el metodo *authenticate(authObject)*.

Si queremos modificar como se realiza un login, debemos Override dicho *authenticate()*.

Contamos con 2 methods:

authenticate() --> devuelve el mismo Object.
supports(Class<?> authentication) --> Verificara que tengamos el "tipo" de autenticacion que corresponde.

Ej: para un FaceRecognition --> support() verificara si tenemos lo necesario para hacerlo y luego llamara al authenticate().

AuthenticationManager

Es una Interface que contiene el metodo *authenticate()*.
Su trabajo sera llamar a cada AuthenticationProvider y pasarles el AuthenticationObject. Hasta que la authentication sea successfull o cuando ya no haya mas AP devolvera una Exception.

authenticate() --> *Dentro del Manager* Usara el support() para encontrar que el tipo de autenticacion correcto, en caso contrario nos devolvera la Exception. Una vez encontrado, pasara el AuthenticationObject al AuthenticationProvider y ejecutara el authenticate()*Dentro del Provider*.

authenticate() --> *Dentro del Provider* autenticara y devolvera el AuthenticationObject de nuevo al AuthenticationManager, una vez autenticado usara el "eraseCredentialsAfterAuthentication", que borrara las credentials recien usadas (no hay necesidad de guardar esta informacion, por cuestiones de seguridad).

Authentication Object

Implementa la interface de Java Security API el Principal quien tiene un solo metodo: *getName()*.

A su vez tiene sus propios metodos que vamos a utilizar:

- *getAuthorities()* -> authorities asociadas.
- *getCredentials()* -> es la password.
- *getDetails()* -> detalles como IP y adicionales.
- *getPrincipal()* -> extiende de Principal el *getName()*.
- *isAuthenticated()* -> es False, pero una vez auth True.
- *setAuthenticated()* -> Setteamos el Boolean de isAuth().

Spring Security usa UserDetails como objeto dentro de nuestra App...

Donde sucede esta transformacion de AuthenticationObject a UserDetails?

Spring Security tiene una Class *DaoAuthenticationProvider* que extiende de *AbstractUserDetailsAuthenticationProvider***.** Esta ultima tiene un method ***authenticate()*** que usara el *getUserDetailsService().loadUserByUsername(username)* para obtener el UserDetails.
Una vez obtenido lo transformara en un AuthenticationObject a partir del metodo: *createSuccessAuthentication(principalToReturn, authentication, user)*
Este metodo usara UsernamePasswordAuthenticationToken y como extiende de Authentication, no habra problema en retornar este tipo de dato. Este metodo mappeara el UserDetails y setteara un AuthenticationObject con sus detalles.

Ejemplo:
Crearemos un AuthenticationProvider que autentique con Username y Password.
Habra que Override el Authenticate() y el Supports() para declarar que usaremos Username y Password.
En caso de necesitar otro tipo de autenticacion se creara un nuevo AuthenticationProvider.

El @Component y la Implementacion haran que SpringSecurity utilice nuestro CustomAuthProvider en vez del que viene por defecto.

```java
package com.flowpractice.security.auth.config;

import ...

@Component
public class CustomAuthenticationProvider implements AuthenticationProvider {
    @Override
    public Authentication authenticate(Authentication authentication) throws AuthenticationException {
        return null;
    }

    @Override
    public boolean supports(Class<?> authenticationType) {
        return authenticationType.equals(UsernamePasswordAuthenticationToken.class);
    }
}
```

authenticate() --> Usaremos Username y Password para autenticar, buscando en nuestra DB

```java
@Override
public Authentication authenticate(Authentication authentication) throws AuthenticationException {

    String username = authentication.getName();
    String pwd = authentication.getCredentials().toString();

    List<TestEntity> myList = testRepo.findByEmail(username);
    if(!myList.isEmpty()){
        if(passwordEncoder.matches(pwd,myList.get(0).getPassword())){
            List<GrantedAuthority> authorities = new ArrayList<>();
            authorities.add(new SimpleGrantedAuthority(myList.get(0).getRole()));
            return new UsernamePasswordAuthenticationToken(username, pwd, authorities);
        } else {
            throw new BadCredentialsException("Invalid Password");
        }
    } else {
        throw new BadCredentialsException("Now User with these Details");
    }
}
```

supports() --> Usara este Provider cuando reciba un input de UsernamePasswordAuthenticationToken

```java
@Override
public boolean supports(Class<?> authenticationType) {
    return authenticationType.equals(UsernamePasswordAuthenticationToken.class);
}
```

Automaticamente SpringSecurity NO UTLIZARA el UserDetailsService, ya que el AuthProvider esta ANTES en el FLOW.
SpringSecurity SALTEARA el CustomUserDetailsService que creamos previamente.

Request - Filter - Manager(lista de Providers) - Provider(athenticate()) - Manager(AuthObject) - Filter(AuthObject) - Context(AuthObject) - Response.

CORS y CSRF

En este punto tenemos un UI realizado. Al intentar conectar a los endpoints obtendremos el error de:
Access to XMLHttpRequest at 'http://localhost:8080/notices' from origin 'http://localhost:4200' has been blocked by CORS policy: Response to preflight request doesn't pass access control check: No 'Access-Control-Allow-Origin' header is present on the requested resource.

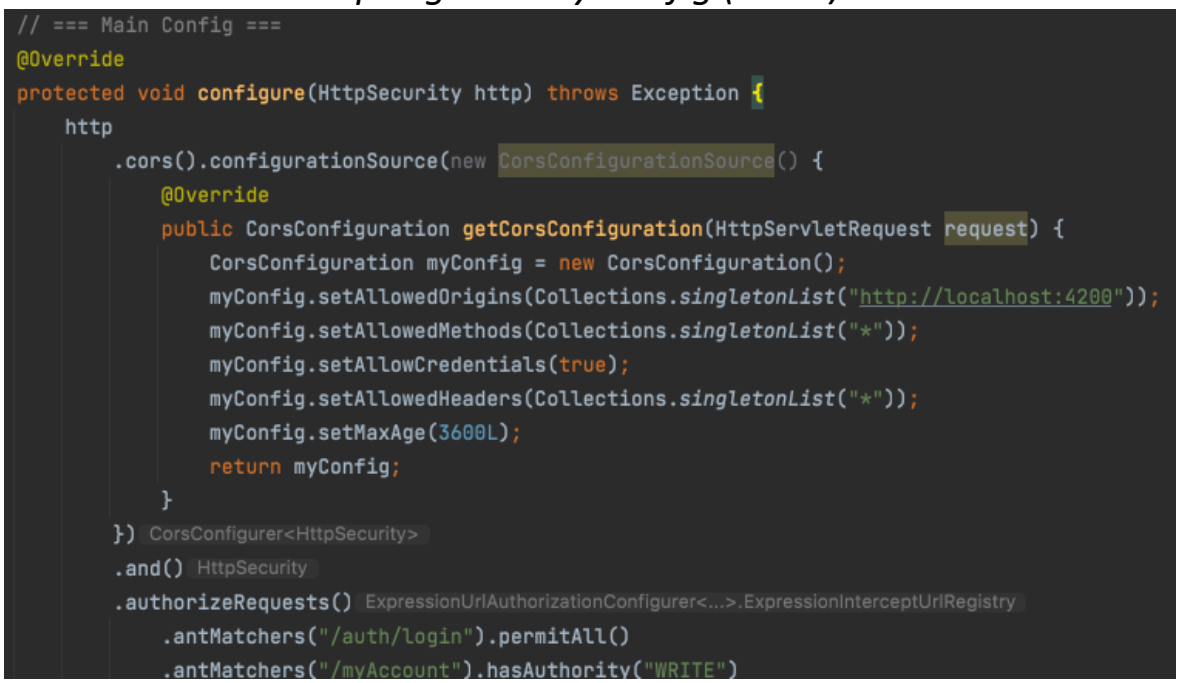
Cross-Origin Resource Sharing(CORS)

Sucede cuando hay Apps con distintos origines, tratando de comunicarse entre ellas.
Ejemplo: Una UI App con "X" Domain quiere comunicarse con el backend con "Y" Domain. Sera bloqueada por defecto.
CORS es una proteccion por defecto, dado por los browsers para prevenir el compartir data entre diferentes origenes (HTTP, diferente domain, diferente puerto, etc...)
Cuando querramos, legitimamente, conectar nuestro UI - Backend que pertenecen a distintos puertos; agregaremos Headers especiales que luego seran evaluados para **permitir** o **negar** el acceso.

Para esto debemos configurar nuestro backend:

- Allow-Origin: Quien tendra acceso.
- Allow-Methods: Http Methods que seran permitidos.
- Allow-Headers: Request Headers permitidos.
- Allow-Credentials: Cuando es TRUE, definira si mostrar o no Response.
- Max-Age: Tiempo de Expiracion del preflight cacheado.

Spring Security Config (CORS)



Corremos la APP (Frontend y Backend) y si bien estamos Logueados, NO nos permite ver nada de nuestra cuenta (myBalance, myCards, etc...) por cuestiones de **CSRF**.

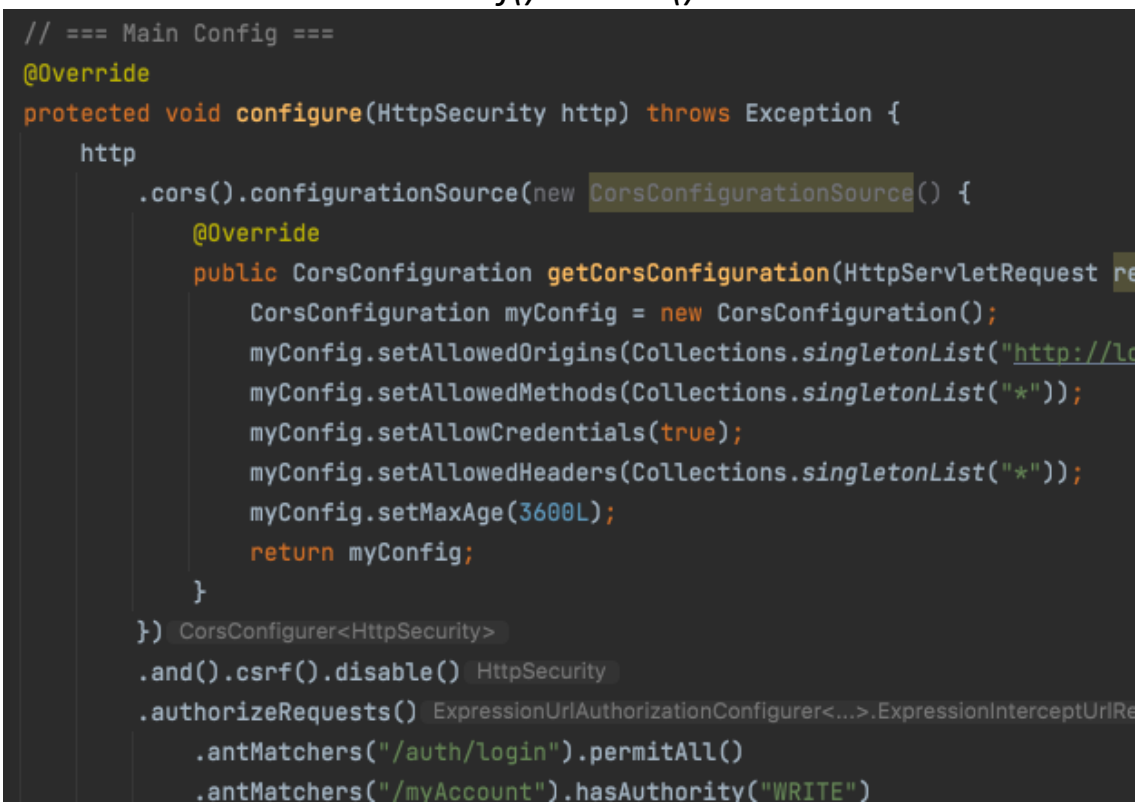
Cros-Site Request Forgery (CSRF)

Por defecto bloqueara toda operacion en nuestra Web.
Un ataque de CSRF o XSRF buscara realizar operaciones en nombre de un User sin su consentimiento. Generalmente no se trata de un "Robo" de Cuenta, sino de explotar ducho user

para llevar a cabo una operacion sin su conocimiento, pero con sus credenciales.

Solucion MALA:

.csrf().disable()



Solucion BUENA:

Es usar un CSRF Token. Una String atada a la session del user, pero que no es enviada automaticamente.

La solicitud se hace unicamente cuando hay una Token y las Cookies correspondientes. No habiendo manera de saber la Token del user, el atacante no podra realizar operaciones en su nombre.

Ejemplo usando una Token:

```
        return myConfig;
    }
}) CorsConfigurer<HttpSecurity>
.and() HttpSecurity
    .csrf().csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
.and() HttpSecurity
.authorizeRequests() ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry
    .antMatchers("/auth/login").permitAll()
```

Vemos la CookieCsrfTokenRepository:

```
public final class CookieCsrfTokenRepository implements CsrfTokenRepository {

    static final String DEFAULT_CSRF_COOKIE_NAME = "XSRF-TOKEN";

    static final String DEFAULT_CSRF_PARAMETER_NAME = "_csrf";

    static final String DEFAULT_CSRF_HEADER_NAME = "X-XSRF-TOKEN";

    private String parameterName = DEFAULT_CSRF_PARAMETER_NAME;

    private String headerName = DEFAULT_CSRF_HEADER_NAME;
```

El frontend debera:

```
validateUser(loginForm: NgForm) {
  this.loginService.validateLoginDetails(this.model).subscribe(
    responseData => {
      this.model = <any> responseData.body;
      this.model.authStatus = 'AUTH';
      window.sessionStorage.setItem("userdetails",JSON.stringify(this.model));
      let xsrf = this.getCookie("XSRF-TOKEN");
      window.sessionStorage.setItem("XSRF-TOKEN",xsrf);
      this.router.navigate(['dashboard']);
    }, error => {
      console.log(error);
    });

}
```

- Agarrar la Cookie llamada "XSRF-TOKEN"

  let xsrf = Cookie("XSRF-TOKEN");

- Set en la Session un Item que contenga el Name y la Cookie elegida:

setItem("XSRF-TOKEN", xsrf);

```
intercept(req: HttpRequest<any>, next: HttpHandler) {
  let httpHeaders = new HttpHeaders();
  this.user = JSON.parse(sessionStorage.getItem('userdetails'));
  if(this.user && this.user.password && this.user.email){
    httpHeaders = httpHeaders.append('Authorization', 'Basic ' + btoa(this.user.
  }
  let xsrf = sessionStorage.getItem("XSRF-TOKEN");
  if(xsrf){
    httpHeaders = httpHeaders.append("X-XSRF-TOKEN", xsrf);
  }
  httpHeaders = httpHeaders.append('X-Requested-With', 'XMLHttpRequest');
  const xhr = req.clone({
    headers: httpHeaders
  });
  return next.handle(xhr).pipe(tap(() => { },
    (err: any) => {
      if (err instanceof HttpErrorResponse) {
```

- Asi mismo debera Intercept cada request y tomar el Item de la Session para agregarlo al HttpHeader. Default name para el Header es: "X-XSRF-TOKEN".

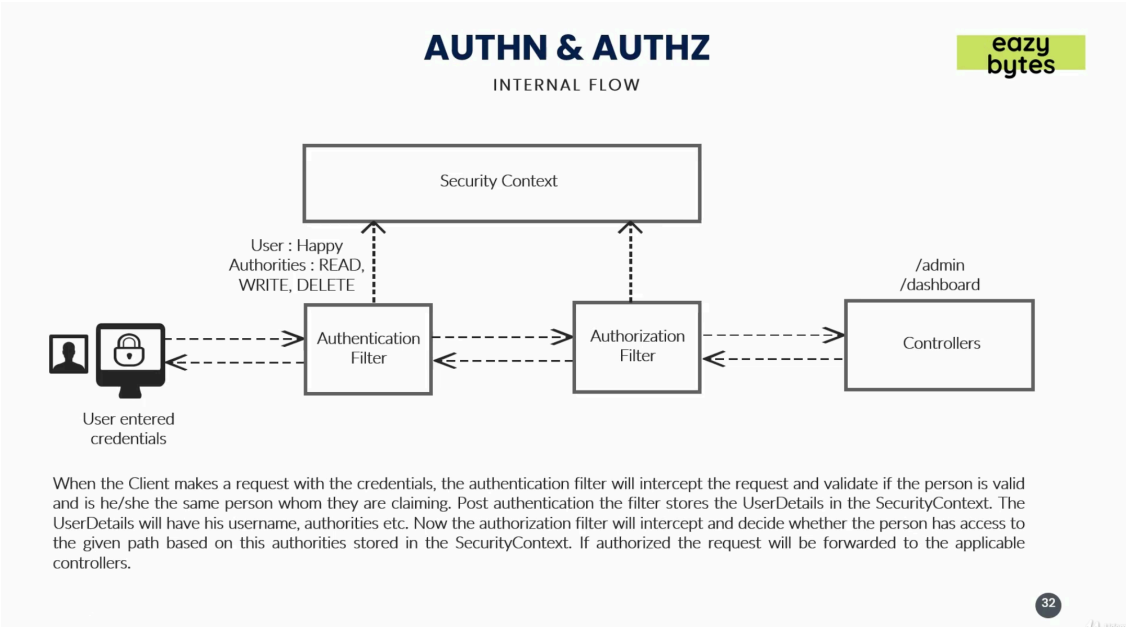
let xsrf = sessiontStorage.getItem("XSRF-TOKEN");
httpHeaders.append("X-XSRF-TOKEN",xsrf);

Tambien se puede **NO APLICAR CSRF** a ciertos endpoints.

```
.and() HttpSecurity
    .csrf().ignoringAntMatchers("/contact").csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
    .and() HttpSecurity
.authorizeRequests() ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry
```

AUTHORIZATION

Sucedera una vez que la autenticacion sea aprobada, seran los Privilegios o Roles para distintos usuarios.
Si falla devolvera un 403 (Forbidden).
Las Authorities decidiran quee clase de acciones puede realizar el usuario.



Flow

1. Request
2. Authentication Filter (Auth Filter, setContext())
3. Authorization Filter(vera las authorities para el user in Context, ej: Read, Write, Delete, etc...)
4. IF Authorized, mandara la Request al Controller.

UserDetails

Tendra las Authorities como Collection<GrantedAuthority>, podran ser obtenidas usando el .getAuthorities().

Granted Authority - Interface

Podemos usar el *getAuthority()* que nos devolvera el Role/Authority como String. Usando dicho valor el framework validara las Authorities del usuario en Context.

Configuring Authorities

Contamos con 3 methods para manejar las authorities y Roles.

- *hasAuthority()* --> Acepta una authority, el endpoint sera configurado y el usario validado, para esta unica Authority. Solo los users que tengan dicha Authority podran llamar al endpoint.
- *hasAnyAuthority()* --> Multiples Authorities, endpoint configurado y unicamente los users que tengan alguno de dichas authorities podran usar el endpoint.
- *access()* --> Usa SpringExpressionLanguage(SpEL) para brindar posibilidades ilimitadas a la hora de configurar Authorities, que no serian posibles con los methods anteriormente dichos. Podemos usar AND y OR operators dentro de access.

Vemos como se aplican las Authority

```
// === Main Config ===
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests() ExpressionUrlAuthorizationConfigurer<...>.Expre
            .antMatchers("/auth/login").permitAll()
            .antMatchers("/myBalance").hasAuthority("UPDATE")
            .antMatchers("/myAccount").hasAuthority("READ")
            .antMatchers("/myCards").hasAuthority("DELETE")
            .and() HttpSecurity
        .formLogin().and()
        .httpBasic();
}
```

Modificaremos nuestro AuthenticationProvider para obtener la lista de Authorities para cada User deseado.

```java
@Override
public Authentication authenticate(Authentication authentication) throws AuthenticationException {

    String username = authentication.getName();
    String pwd = authentication.getCredentials().toString();

    List<Customer> myList = testRepo.findByEmail(username);
    if(!myList.isEmpty()){
        if(passwordEncoder.matches(pwd,myList.get(0).getPwd())){
            return new UsernamePasswordAuthenticationToken(username, pwd, getGrantedAuthorities(myList.get(0).getAuthorities()) );
        } else {
            throw new BadCredentialsException("Invalid Password");
        }
    } else {
        throw new BadCredentialsException("Now User with these Details");
    }
}
private List<GrantedAuthority> getGrantedAuthorities(Set<Authority> authorities){
    List<GrantedAuthority> grantedAuthorities = new ArrayList<>();
    for (Authority authority : authorities){
        grantedAuthorities.add(new SimpleGrantedAuthority(authority.getName()));
    }
    return grantedAuthorities;
}
```

- Usamos *findByEmail()* y *getPwd()* - overriding el default user de SpringSecurity -
- Para el User encontrado(mylist[0]) usaremos un method inventado llamado *getGrantedAuthorities()*. Obtendremos una List de cada una de sus authorities.

Authorities vs Roles

Auth -> Privilegio Individual. Ej: Read, Update, Dlete
Role -> Grupo de Privilegios. Ej: Admin, User, etc...

Roles

Podemos validar o configurar con los siguientes metodos:

- *hasRole()* -> Acepta un solo Role
- *hasAnyRole()* -> Multiple Roles
- *access()* -> SpEL, tenemos operators como OR, AND

Vemos como se aplican los Roles

Ant matchers, MVC matchers, Regex

A ver

FILTERS

A ver

Autenticacion con JWT

A ver

Seguridad a nivel de Methods

A ver