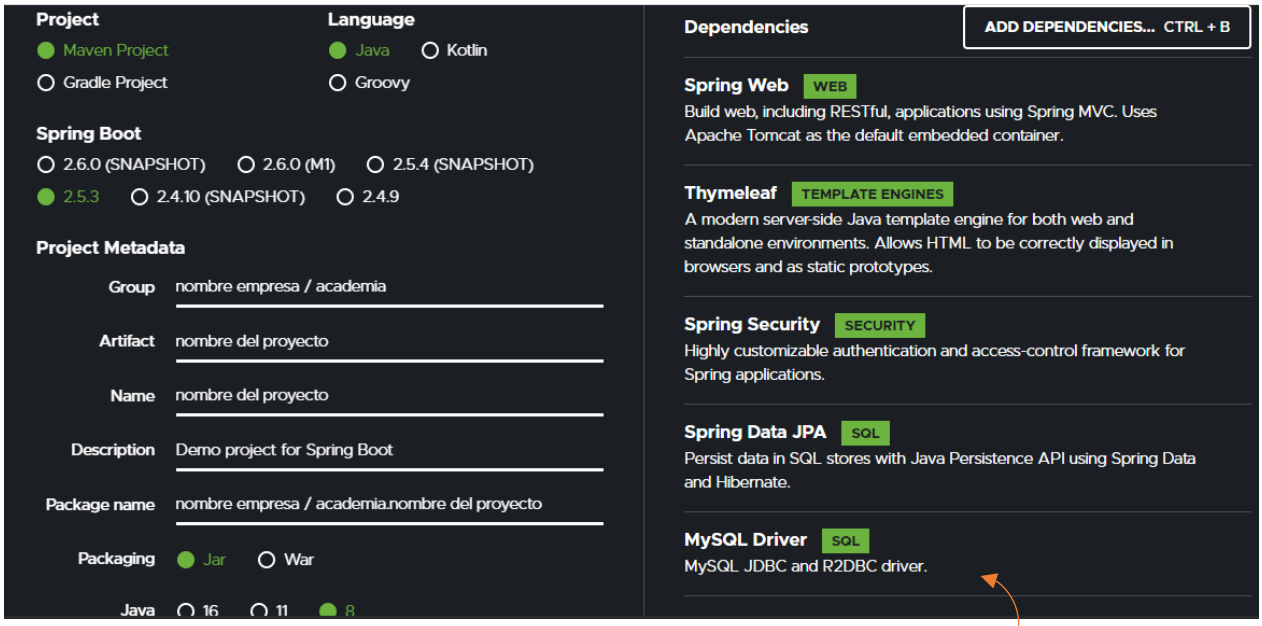
PROYECTOS CON SPRING

VERIFICAR INSTALACIÓN DE MAVEN

1-CONFIGURACIÓN DE SPRING:

NAVEGADOR: star.spring.io



Puedo agregar Java Mail Sender

Descargo el paquete y abro con NetBeans. Con el cursor sobre el nombre del proyecto clickeo CLEAN AND BUILD (corre en Maven)

2-CONSTRUCCIÓN DE CAPA DE DATOS

Diseño UML.

¿Qué entidades necesito? ¿Cuáles serán sus atributos y cómo se relacionan? ¿Cuál será su llave primaria? ¿Algún atributo será del tipo enumeración? ¿Necesito ROLES?

¿Tendré LogIn? - ¿Cuál será el usuario?

¿Pongo fotos?

Una vez que pensamos todo esto, armamos el esquema UML.

CREACIÓN DE ENTIDADES

- ✓ Agregar la anotación @Entity (arriba del nombre de la clase)
- ✓ Según el atributo, agrego las anotaciones correspondientes (@Id, @EnumeratedType)
- ✓ Getter & Setter → Clic derecho. REFACTOR. ENCAPSULATE FIELDS.
- ✓ Entidad FOTO

  - Relación OneToOne con la entidad de la foto.
  - Atributos:
    - String name
    - String mime (es el tipo)
      @Lob @Basic(Fetch=LAZY)
    - byte[] contenido (array de bytes)

CREACIÓN DE REPOSITORIOS

Este es un concepto nuevo. Anteriormente con JPA manejamos un EntityManager que se encargaba de persistir cambios en las BdD. Ahora tenemos "REPOSITORIOS", ¿qué hacen? Conectan la lógica del proyecto con las BdD.

Deben generarse en un paquete APARTE (repositorios) y cada entidad creada, tendrá su interfaz repositorio.

- ✓ Anotación @Repository (antes del nombre)
- ✓ interface entidadRepository extends JpaRepository<Entidad, Tipo>

Por c/ entidad en el Paquete de Entidades, repito el nombre y agrego repository

Agregar la extensión de JPA para realizar la conexión con la BdD (se hace automáticamente y ya no armamos la unidad de "persistencia".

Nombre de la Entidad del Repo. Respetar mayúsculas usadas en la Clase Entity

Tipo de dato del Primary Key de esa entidad. PRIMITIVO!

CREACIÓN DE REPOSITORIOS

Es en esta interface donde ahora escribimos nuestras Querys. Tienen el siguiente formato:

```
@Query("SELECT a FROM Tabla a WHERE a.atributo LIKE :parámetro")
    public Objeto buscarPor…..(@Param ("parámetro")TipoDeDato parámetro);
```

El nombre de la tabla se escribe **IGUAL** a la entidad.
No sean como yo que rompí todo mi programa por poner la primera en minúscula.

CREACIÓN DE SERVICIOS

Ésta es la parte lógica y operativa del proyecto. Acá nuestros datos interactúan con métodos que operan entre la BdD y el Controller. Por el momento, hay que armar las funciones (muy probablemente armen el método de cierta manera, y luego deban cambiar alguna que otra cosa)

- ✓ Agregar la anotación @Service antes del nombre
- ✓ Para cada entidad, su clase Service
- ✓ **SIEMPRE** deberé colocar la anotación:
  @Autowired
  Public EntidadRepository entRepo;
  ¿Qué hace esta anotación? Vincula esta clase de servicio con el repositorio, por ende, con la BdD. Cuando agregamos la extensión del Repo, nos habilita a usar varios métodos que ya están armados (ex. Repo.save(objeto) que nos hace persistir objetos en la BdD. Explorar antes de escribir Querys, ya que tiene armados varios métodos de búsqueda)

```
Optional<Objeto> rta = repositorio.findById(parámetro);
If(rta.ispresent){
        Objeto obj = rta.get();
}
```
*Busco en el repo y si la búsqueda es exitosa, alojo el resultado en un objeto.

IMPORTANTE: Cada Service debería tener un método de validación que chequee que la info llegue como corresponde – no nula. Si uso claves, en ese método valido que la clave1=clave2.

IMPORTANTÍSIMO: cada método que realice CAMBIOS en la BdD debe tener la anotación @Transactional 😊

MANEJO DE ERROR

Pues claro, nuestros códigos van a generar errores, y para refrescarles un poco la guía de excepciones, manejarlos hace que el programa no explote y nos salga un bonito mensaje diciendo donde saltó el error. Principalmente, en donde el usuario no cargue bien los FORMS.

Creamos un nuevo paquete “Errores”, dentro una clase ErrorService que extienda de Exceptions y adentro armamos un constructor de errores:

```
Public ErrorService (String msn){

Super(msn);

}
```

Cada método de cada clase de servicio puede tirar errores por lo que agregamos “throws ErrorService” y englobamos la lógica del método en un try/catch. Algo asi:

```
public void EliminarCliente(Long DNI) throws ErrorService {


    Optional<Cliente> respuesta = clienteRepositorio.findById(DNI);

    if (respuesta.isPresent()) {

        Cliente c2 = respuesta.get();

        clienteRepositorio.delete(c2);

    } else {

        throw new ErrorService("El DNI ingresado no corresponde a un usuario del sistema");

    }
```

SECURIZACIÓN

Supongamos que nuestro proyecto necesita que los usuarios realicen LogIns, Spring nos da todas las herramientas que necesitamos para que valide los datos. Arme un paso a paso de cómo y dónde hay que configurarlo.

Primero necesitamos identificar cuál es la entidad que realizará los ingresos (Lo mas habitual sería una Entidad Usuario/Cliente. Y posiblemente tengamos distintos ROLES. ¿Qué sería esto? Que hay usuarios que pueden hacer o ver algunas cosas que otros no.

1- Dentro de Source Packages tengo el paquete base, donde esta el main. Agrego una clase dentro de ese paquete "Seguridad configuración". Ahí dentro va toooodo este código:

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SeguridadConfiguracion extends WebSecurityConfigurerAdapter {

    @Autowired
    public ClienteService clienteService;

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws
Exception {
        auth
            .userDetailsService(clienteService)
            .passwordEncoder(new BCryptPasswordEncoder());
    }

    @Override
    protected void configure (HttpSecurity http) throws Exception{
        http.headers().frameOptions().sameOrigin().and()
                .authorizeRequests()
                    .antMatchers("/css/*", "/js/*", "/img/*")
                    .permitAll()
        .and().formLogin()  ->avisa que hay un FORM p/ LogIn
                    .loginPage("/login")  ->URL donde está ese FORM
                        .loginProcessingUrl("/logincheck") ->Action del Form (vincula Controller)
                        .usernameParameter("email") ->Atributo name del input HTML
                        .passwordParameter("clave") ->Atributo name del input HTML
                        .defaultSuccessUrl("/inicio") ->URL donde va si esta todo Ok. (vincula Controller)
                        .failureUrl("/login?error=error") ->URL donde va si esta todo MAL.
                         .permitAll()
        .and().logout()
                        .logoutUrl("/logout") ->Donde se procesa el LogOut (vincula Controller)
                        .logoutSuccessUrl("/") ->URL donde va si esta todo Ok.
                        .permitAll();
    }
}
```

Anotaciones para avisar que usaremos Spring Security

EntidadServicio donde están los datos para LogIn

Conecta el Service con el sistema potencial de LogIn

"Aviso" que el userDetailsService va a estar en esta EntidadService

Autorizaciones para TODOS.

LOGIN

LOGOUT

SECURIZACIÓN 2

2- En la clase EntidadService designada (usada en el paso 1) agregamos que implementa UserDetailsService y nos obliga a importar un método Override. Y debería quedar asi:

O lo que use para logearme

```
@Override
public UserDetails loadUserByUsername(String email) throws
UsernameNotFoundException {

Usuario user = usuarioRepository.buscarPorEmail(email);

if (user != null) {
        List<GrantedAuthority> permissions = new ArrayList<>();
        GrantedAuthority p = new SimpleGrantedAuthority("ROLE_" +
user.getRol().toString());
        permissions.add(p);
        ServletRequestAttributes attr = (ServletRequestAttributes)
RequestContextHolder.currentRequestAttributes();
        HttpSession session = attr.getRequest().getSession(true);
        session.setAttribute("user", user);
return new org.springframework.security.core.userdetails.User(user.getEmail(),
user.getClave(), permissions);
                }
                return null;
        }
```

Array de permisos

Con esta línea creo el permiso en función del atributo ROL. Partiendo de la base de que el mismo objeto puede funcionar para cosas distintas (User!=Admin). Entonces concateno el permiso “ROLE_” y traigo el atributo y parseo a String.

Permite varios roles pero SOLO UNO por persona.

Tomo los datos de la sesión activa y los guardo en “user”. Mas adelante se usa para mandar datos del objeto loggeado al front.

INTERFAZ / VISTAS

Tomemos un descancito de la securización (todavía no termina, mildis, pero es momento de hablar del FRONT)

IMPORTANTE A LA HORA DE ARMAR EL FRONT

- ✓ En la etiqueta xmlns:th="http://www.thymeleaf.org" xmlns:sec="http://www.thymeleaf.org/extras/spring-security" > Estoy avisando que voy a usar etiquetas de thymeleaf y spring security

- ✓ Armar el index para usar de plantilla para el resto de los html.
- ✓ Tener en mente a la hora de diseñar los FORMS que el atributo name de cada etiqueta input es lo que va de parámetro al Controller
- ✓ Identificar las vistas sujetas a ROLES, estas tendrán etiquetas th:if. Dejo ejemplo:

<a th:if="${#authorization.expression('hasAnyRole('USUARIO')')}" th:href="@{/url}">

- ✓ Los html van en la carpera TEMPLATES
- ✓ Los css, js, json, imágenes en la carpera STATIC

USAR THYMELEAF P/ MOSTRAR VARIOS DATOS DE UNA TABLA

Por ejemplo: Tengo una tabla con datos y quiero mostrarlo mediante select

HTML:

<select name="idZona">

<option th:each="zona : ${zonas}" th:value="${zona.id}" th:text="${zona.nombre}"

SECURIZACION

POM: Verificar que esten TODAS las dependencias de Security (3)

ENCRIPTACIÓN DE CLAVES

En el método EntidadService donde se realiza la securización, al setear la clave como atributo del objeto hacerlo de esta manera:

String claveEncriptada = new BCryptPasswordEncoder().encode(clave);

Entidad.setClave(claveEncriptada);

SEGURDIAD EN LOS CONTROLLERS

Si ya se, todavía no dije NADA de los Controllers, ya llegará. PERO es importante tener en mente que agrearemos la etiqueta

@PreAuthorize("hasRole('ROLE_ADMIN')")

Para acceder a ese mapping SI SOLO SI tengo ese rol (después lo explico mejor.)

CONTROLLERS

Creo un nuevo paquete Controller y dentro una clase PortalController.

El controlador es el encargado de dos cosas SUMAMENTE importantes. Por un lado va a "mapear" las url del front, es decir, cada armar enlaces que irán a los html (evaluando permisos), arma una telaraña de nuestros html. Además se encarga de recibir la información de los Forms y mandarla a los Service. Es importante que tengamos mucha noción de cómo se conectan nuestros html.

La clase PortalController hace un primer mapeo de nuestra landing page (index). Para mantener cierto orden, iremos creando mas controllers para gestionar la info y el mapping.

Debemos agregar las etiquetas

```
@Controller
@RequestMapping("/")

@GetMapping("/")
Public String index(){
        Return "index.html";
}
```

Aviso que el mapeo de rutas comienza a "leerse" desde la / en adelante. Como es el index, este va con / sola

Cada href del html que tenga "/" ejecutará este método.

Retorna la vista que quiero que se renderice.

CONTROLLERS

ENVIAR INFO DE LOS FORMS -> AL CONTROLLER -> AL SERVICE ->A LA BdD

Sólo si el FORM tiene contenido multimedia

```
HTML
<form th:action="@{/url}"
method="POST"
enctype="multipart/form">
<input name="parametro">
```

```
CONTROLLER
@PostMapping("/url")
Public String método(ModelMap model
@RequestParam tipo parametro)
```

Acá podemos ver cómo se vincula el HTML con el CONTROLLER y lo IMPORTANTISIMO que es el atributo name de las etiquetas input.

Dentro del Controller vamos a llamar al método de la Entidad de Servicio y pasaremos esos parámetros. No se olviden de usar el @Autowired para instanciar esa EntidadServicio.
No olvidar usar SIEMPRE el try/catch, ya que esos mensajes personalizados que armamos en la clase service, ahora debemos mostrarlos por pantalla para avisarle al usuario qué esta cargando mal. Para eso viene el ModelMap

MODELMAP
Es una gran herramienta, permite llevar cosas al FRONT. Cómo? Dejo en ejemplo:
CONTROLLER

```
Try{
 entidadService.metodo(parametros);
} catch (ErrorService e) {
 model.put("error", e.getMessage());
 return "error.html";
}
Return "succes.html";
```

HTML

```
<p th:if="${error  != null}" th:text = "${error}"></p>
```

Le estoy diciendo al HTML que si el error no es nulo, me muestre el error.

Con esta misma lógica, en el método puedo agregar
Model.put("nombre atributo name", parametro) y carga en el form los datos que se enviaron la primera vez, para que el usuario no deba cargarlos de nuevo.

MANEJO DE ERRORES
Puede ocurrir por ejemplo que falle la securización porque el usuario equivoco los datos, entonces hay un error en el getMapping, por lo que debo perfeccionar un poco mis métodos:

No es obligatorio

```
@GetMapping("/login")
Public String login (@RequestParam(required = false) String error, ModelMap model) {
        If(error  != null){
                Model.put("error", "Nombre de usuario o clave incorrectos")
}
Return "index.html";
}
```

CONTROLLERS

Vamos a necesar un Controller específico para manejar errores y un html al que nos dirija el controller, para mostrar esos errores.
El html facilito, con la misma plantilla de todos y agregamos lo siguiente:

```
<h1 th:text="${codigo}">
<h2 th:text="${mensaje == null  ? 'Intenta nuevamente'  : mensaje}"

@Controller
public class ErrorController implements
org.springframework.boot.web.servlet.error.ErrorController {

	@RequestMapping(value = "/error", method = { RequestMethod.GET,
RequestMethod.POST })
	public ModelAndView renderErrorPage(HttpServletRequest httpRequest) {

		ModelAndView errorPage = new ModelAndView("error");
		String errorMsg = "";
		int httpErrorCode = getErrorCode(httpRequest);

		switch (httpErrorCode) {
		case 400: {
			errorMsg = "El recurso solicitado no existe.";
			break;
		}
		case 403: {
			errorMsg = "No tiene permisos para acceder al recurso.";
			break;
		}
		case 401: {
			errorMsg = "No se encuentra autorizado.";
			break;
		}
		case 404: {
			errorMsg = "El recurso solicitado no fue encontrado.";
			break;
		}
		case 500: {
			errorMsg = "Ocurrió un error interno.";
			break;
		}
		}

		errorPage.addObject("codigo", httpErrorCode);
		errorPage.addObject("mensaje", errorMsg);
		return errorPage;
	}

	private int getErrorCode(HttpServletRequest httpRequest) {
		return (Integer) httpRequest.getAttribute("javax.servlet.error.status_code");
	}

	@Override
	public String getErrorPath() {
		return "/error";
	}

}
```

CONTROLLERS

Ahora si, veamos un poco de cómo armar otros controllers. Nos va a servir mucho para organizar las vistar o accesos de los usuarios según sus roles.

Va ejemplo:

```
@Controller
@RequestMapping("/usuario")


@GetMapping("/editar-perfil")
Public String editarPerfil(@RequestParam tipoDato id, ModelMap model){
    Try{
            Usuario usuario = usuarioService.buscarPorId(id);
            Model.addAttribute("perfil", usuario);
} catch (ErrorService e) {
 model.addAttribute("error", e.getMessage());
}
Return "perfil.html";
```

El action de este form sería /usuario/editar-perfil
La primera parte ubica el controller y la segunda el método.

Guardo el objeto encontrado en la variable perfil, y puedo usarla en el front

¿MOSTRAR EN EL HTML?
<input type="hydden" name="id" th:value="${perfil.id}"/> →id de usuario oculto
En el resto de las etiquetas input agergo th:value="${perfil.atributo}" y lo carga.

RE RE MIL ULTRA IMPORTANTE

A este método llegue desde otro lugar donde el usuario ya estaba logueado. La etiqueta o botón que me hizo llegar hasta ahí debe tener SI O SI lo siguiente:

<a th:href="@{/usuario/editar-perfil(id=__${session.usuario.id}__)}"

¿CÓMO HAGO QUE ESTAS MODIFICACIONES SE MUESTREN AUTOMÁTICAMENTE?

Dentro del try del PostMapping donde llevo los datos, antes del retorno agrego un:
Session.setAttribute("usuario", usuario);

SECURIZACION DE LOS CONTROLLERS

Como dijimos antes, la etiqueta @Preauthorize se utiliza antes de los métodos Get/Post, pero también puedo pre autorizar un controller completo.

Como medida extra de seguridad, debemos chequear que los ID que viajen como parámetros para realizar modificaciones, sean iguales a los ID de la sesion logeada.

Cómo?

Pasando como parámetro en los métodos el objeto session tipo HttpSession y agrego la validación:

```
Usuario login = session.getAttribute("usuario");

If(login==null ||  !login.getId().equals(id)){

Return "redirect:/inicio";

}
```

Recupero el usuario de la sesión

LLEVAR AL FRONT VISTAS QUE NO SEAN OBJETOS: FOTOS

1. Generar un FotoController (“/foto”)
2. Generamos un método

```
@Autowired usuarioService
@GetMapping(“/usuario”)
Public ResponseEntity<byte[]> fotoUsuario(@RequestParam String id){
Try{
        Usuario us=usuarioService.buscarPorId(id);
        If(us.getFoto==null){
        Throw new ErrorService(“El usuario no tiene foto”);
        }
        Byte[] foto = us.getFoto.getContenido();
        HttpHeaders headers = new HttpHeaders();
        Headers.setContentType(MediaType.IMAGE.JPEG);
        Return new ResponseEntity<>(foto, header, HttpStatus.OK)

} catch (ErrorService ex){
        Logger….
        Return new ResponseEntity<>(HttpStatus.NOT_FOUND)

}

}
```

Le avisa al navegador que retorna una img

3. MUESTRO EN EL FRONT

```
<img th:if=”${perfil.foto  != null}” th:src=”${‘/foto/usuario?id =’ + perfil.id}”
```