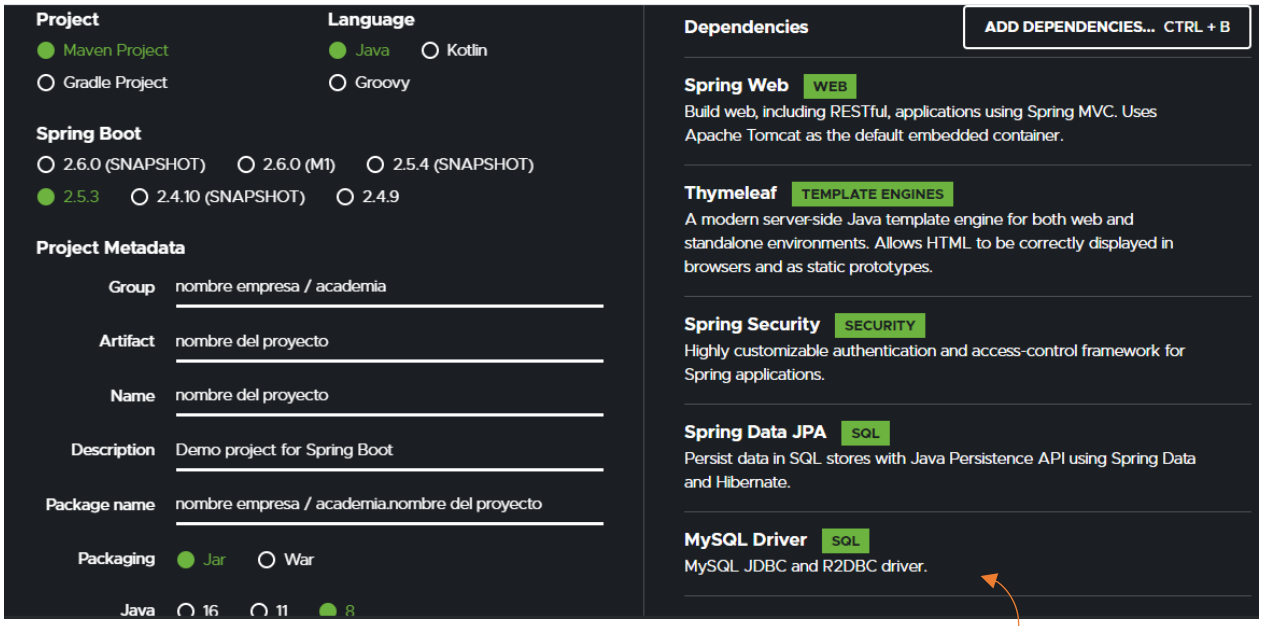
PROYECTOS CON SPRING

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}”
```

Proyecto Final

Nombre del Proyecto:

Integrantes:

Líder del Proyecto. (Elijan a un referente del equipo, la persona que va a organizar el proyecto y va a ser el nexo entre los tutores y el equipo)

Breve descripción del proyecto. Definir en no mas de 20 palabras de que se trata el sistema a realizar

Ciclo de vida de un proyecto de Software

El ciclo de vida de un proyecto hace referencia a las etapas por las cuales pasa el proyecto desde su concepción hasta su implementación y puesta en marcha final.

A continuación se muestran las etapas por las cuales va a ir pasando nuestro proyecto.

El ciclo de vida básico de un *software* consta de los siguientes procedimientos:

1- Definición de objetivos: define la finalidad del proyecto.

2- Análisis de los requisitos y su viabilidad: recopila, examina y formula los requisitos del cliente y examina cualquier restricción que se pueda aplicar.

3- Diseño general: requisitos generales de la arquitectura de la aplicación.

4- Diseño en detalle: definición precisa de cada subconjunto de la aplicación.

5- Programación (programación e implementación): implementación de un lenguaje de programación para crear las funciones definidas durante la etapa de diseño.

6- Prueba de unidad: prueba individual de cada subconjunto de la aplicación para garantizar que se implementaron de acuerdo con las especificaciones.

7- Integración: garantiza que los diferentes módulos se integren con la aplicación. Este es el propósito de la prueba de integración que está cuidadosamente documentada.

8- Prueba beta (o validación): garantiza que el *software* cumple con las especificaciones originales.

9- Documentación: sirve para documentar información necesaria para los usuarios del *software* y para desarrollos futuros.

10- Implementación

11- Mantenimiento: comprende todos los procedimientos correctivos (mantenimiento correctivo) y las actualizaciones secundarias del *software* (mantenimiento continuo).

El orden y la presencia de cada uno de estos procedimientos en el ciclo de vida de una aplicación dependen del tipo de modelo de ciclo de vida acordado entre el cliente y el equipo de desarrolladores.

Durante la ejecución del proyecto final vamos a atravesar por varias de estas etapas, para poder llevar a cabo la mejor solución para nuestro problema a resolver.

Etapa 1. Definición del objetivo general y los objetivos específicos del proyecto.

Identificar los objetivos del proyecto

El primer paso para definir los objetivos del proyecto es identificar qué aspectos del proyecto son realmente importantes; y aquí debemos separar dos ámbitos:

- Los objetivos del proyecto a nivel del producto que este debe entregar a la organización, cliente, u otros involucrados.
- Aquellos aspectos internos del proyecto que son más importantes, y por tanto van a requerir mayor atención, para la consecución de los objetivos.

Para el primer punto deberemos empezar identificando y priorizando los objetivos a través de la información que nos den los involucrados (cliente, sponsor, usuarios, etc.),

y de los documentos contractuales que puedan dar origen al proyecto (pedido, oferta, etc.). El tema de priorizar es muy importante, ya que aunque nuestro objetivo será cumplir con todo, la realidad es que nos podemos encontrar con situaciones que nos impidan poder hacerlo, por lo que debemos ser capaces de saber cuáles objetivos pueden ser más flexibles.

Por otro lado, una vez hayamos planificado el proyecto, veremos que no todas las tareas o recursos involucrados van a tener el mismo efecto a la hora de conseguir los objetivos. Por lo que nuestra atención se deberá centrar en aquellos que sean más relevantes. Esto se entiende bien cuando planificamos con Camino Crítico, ya que las tareas dentro de este camino son las que marcan la duración total del proyecto, por lo que pasan a ser más relevantes que el resto.

Definir correctamente los objetivos del proyecto

"No se puede controlar lo que no se puede medir", por tanto para controlar si estamos consiguiendo o no los objetivos del proyecto, debemos definir estos de forma que sean medibles.

Objetivo general

Los objetivos generales son aquellos que se formulan para completar una meta o señalar el fin de un proyecto o investigación.

El objetivo general en el caso de un proyecto siempre va relacionado con el título del mismo, es decir, es el mismo título con la diferencia que se encuentra redactado en infinitivo.

Es por ello que el objetivo general es el paso más sencillo de formular al momento de realizar una investigación a diferencia del planteamiento del problema y los marcos legales, metodológicos y conceptuales.

Incluso es más sencillo de redactar que los objetivos específicos.

Al formular el objetivo general es conveniente tomar en cuenta que debe ser:

- **Cualitativo:** Debe indicar calidad.
- **Integral:** Debe ser capaz de integrar los objetivos específicos y para ello debe estar encabezado por un verbo de mayor nivel.
- **Terminal:** Debe definir un plazo.

Objetivos específicos

Se trata de objetivos que provienen del objetivo general, son objetivos redactados con un verbo de menor nivel al verbo en el que fue redactado el objetivo general del que provienen.

Los objetivos específicos son una serie de fines que llevan a cumplir el objetivo general.

Los objetivos específicos deben tener:

Claridad: El lenguaje en el que se encuentran redactados debe ser claro y preciso.

Factibilidad: Los objetivos deben poder ser cumplidos a través de la metodología seleccionada.

Pertinencia: Deben relacionarse lógicamente con el problema a solucionar.

Diferencia entre objetivo general y específico

- Un objetivo general precisa la finalidad de un estudio, investigación, organización o empresa.
- Los objetivos generales son actividades a cumplir para llevar a cabo el objetivo general.

En nuestro caso vamos a pensar que nuestro sistema va a quedar completamente operativo e intslado, entonces por ejemplo para el tinder de mascotas un objetivo sería:

Tener 1500 usuarios de la aplicación al finalizar diciembre del 2020

Y un objetivo específico sería:

Invertir 100 pesos en campañas de Marketing para fidelizar la aplicación.

Etapa 2. Análisis de requisitos y Diseño

En esta etapa vamos a “entrevistar” al ideador del proyecto para poder determinar qué es lo que debe hacer el Sistema, una vez relevados los datos debemos validar con el ideador las funcionalidades detectadas.

La toma de requerimientos es de las etapas mas importantes, ya que todo nuestro sistema se fundamentará en el relevamiento que hagamos en esta etapa.

Una vez definida las funcionalidades, estas se documentan mediante Casos de Uso. Qué son los casos de uso?

Casos de Uso:

En ingeniería del software, un caso de uso es una técnica para la captura de requisitos potenciales de un nuevo sistema o una actualización de software. Cada caso de uso proporciona uno o más escenarios que indican cómo debería interactuar el sistema con el usuario o con otro sistema para conseguir un objetivo específico. Normalmente, en los casos de usos se evita el empleo de jergas técnicas, prefiriendo en su lugar un lenguaje más cercano al usuario final. En ocasiones, se utiliza a usuarios sin experiencia junto a los analistas para el desarrollo de casos de uso.

En otras palabras, un caso de uso es una secuencia de interacciones que se desarrollarán entre un sistema y sus actores en respuesta a un evento que inicia un actor principal sobre el propio sistema. Los diagramas de casos de uso sirven para especificar la comunicación y el comportamiento de un sistema mediante su interacción con los usuarios y/u otros sistemas. O lo que es igual, un diagrama que muestra la relación entre los actores y los casos de uso en un sistema. Una relación es una conexión entre los elementos del modelo, por ejemplo la especialización y la generalización son relaciones. Los diagramas de casos de uso se utilizan para ilustrar los requerimientos del sistema al mostrar cómo reacciona a eventos que se producen en su ámbito o en él mismo

Características

Los casos de uso evitan típicamente la jerga técnica, prefiriendo la lengua del usuario final o del experto del campo del saber al que se va a aplicar. Los casos del uso son a menudo elaborados en colaboración por los analistas de requerimientos y los clientes.

Cada caso de uso se centra en describir cómo alcanzar una única meta o tarea de negocio. Desde una perspectiva tradicional de la ingeniería de software, un caso de uso describe una característica del sistema. Para la mayoría de proyectos de software, esto significa que quizás a veces es necesario especificar diez o centenares de casos de uso para definir completamente el nuevo sistema. El grado de la formalidad de un proyecto particular del software y de la etapa del proyecto influenciará el nivel del detalle requerido en cada caso de uso.

Los casos de uso pretenden ser herramientas simples para describir el comportamiento del software o de los sistemas. Un caso de uso contiene una descripción textual de todas las maneras que los actores previstos podrían trabajar con el software o el sistema. Los casos de uso no describen ninguna funcionalidad interna (oculta al exterior) del sistema, ni explican cómo se implementará. Simplemente muestran los pasos que el actor sigue para realizar una tarea.

Un caso de uso debe:

- describir una tarea del negocio que sirva a una meta de negocio
- tener un nivel apropiado del detalle

- ser bastante sencillo como que un desarrollador lo elabore en un único lanzamiento Situaciones que pueden darse:
- Un actor se comunica con un caso de uso (si se trata de un actor primario la comunicación la iniciará el actor, en cambio si es secundario, el sistema será el que inicie la comunicación).
- Un caso de uso extiende otro caso de uso.
- Un caso de uso utiliza otro caso de uso.

Ventajas

La técnica de caso de uso tiene éxito en sistemas interactivos, ya que expresa la intención que tiene el actor (su usuario) al hacer uso del sistema.

Como técnica de extracción de requerimiento permite que el analista se centre en las necesidades del usuario, qué espera éste lograr al utilizar el sistema, evitando que la gente especializada en informática dirija la funcionalidad del nuevo sistema basándose solamente en criterios tecnológicos.

A su vez, durante la extracción (elicitation en inglés), el analista se concentra en las tareas centrales del usuario describiendo por lo tanto los casos de uso que mayor valor aportan al negocio. Esto facilita luego la priorización del requerimiento.

Limitaciones

Los casos de uso pueden ser útiles para establecer requisitos de comportamiento, pero no establecen completamente los requisitos funcionales ni permiten determinar los requisitos no funcionales. Los casos de uso deben complementarse con información adicional como reglas de negocio, requisitos no funcionales, diccionario de datos que complementen los requerimientos del sistema. Sin embargo la ingeniería del funcionamiento especifica que cada caso crítico del uso debe tener un requisito no funcional centrado en el funcionamiento asociado.

Buenas prácticas y recomendaciones de uso

Los casos de uso, como el resto de los requisitos, deben tener una redacción cuidada para evitar problemas de interpretación. En general, algunas recomendaciones a tener en cuenta son:

- El caso de uso debe describir qué debe hacer el sistema a desarrollar en su interacción con los actores y no cómo debe hacerlo. Es decir, debe describir sólo comportamiento observable externamente, sin entrar en la funcionalidad interna del sistema.
- El nombre del caso de uso debe ilustrar el objetivo que pretende alcanzar el actor al realizarlo.
- El caso de uso debe describir interacciones con los actores sin hacer referencias explícitas a elementos concretos de la interfaz de usuario del sistema a desarrollar.
- La invocación de unos casos de uso desde otros casos de uso (lo que se conoce como inclusión, o extensión si es condicional, en UML), sólo debe usarse como un mecanismo para evitar repetir una determinada secuencia de pasos que se repite en varios casos de uso. Nunca debe usarse para expresar posibles menús de la interfaz de usuario.

- Se debe ser cuidadoso al usar estructuras condicionales en la descripción del caso de uso, ya que los clientes y usuarios no suelen estar familiarizados con este tipo de estructuras, especialmente si son complejas.
- Se debe intentar que todos los casos de uso de una misma ERS estén descritos al mismo nivel de detalle.
- En los diagramas de casos de uso, debe evitarse que se crucen las líneas que unen los actores a los casos de uso.
- En caso de usar github, discutir el nombre de las variables que se van a usar y dejarlas guardadas en un pdf para evitar problemas y conflictos en el repositorio.

A continuación les dejo unos ejemplos de un caso de uso del tinder de mascotas.

Recuerden identificar TODAS las funcionalidades que quieran incluir en el Sistema y diagramarlo como caso de uso. (Para el tinder de mascotas por ejemplo: Login de Usuario, Registración Usuario, Crear mascota, Buscar mascota cercana, Like mascota, etc)La idea es hacer un caso de uso de manera sencilla.

CU 1. Login del usuario

CU 1. Login del usuario	
Descripción	El usuario ingresa al sistema mediante usuario y clave
Precondición	(Acá va algo sólo si debe existir algo antes que esta acción)Por ejemplo: El usuario debe estar dado de alta.
Acciones (Siempre se describe el camino feliz, osea la situación exitosa)	1- El usuario ingresa nombre y clave 2- El usuario presiona Aceptar 3- El sistema valida que exista un usuario con esas credenciales. 4- El sistema muestra la pantalla principal de la aplicación. (Debemos colocar paso a paso la interacción entre el/los actores y el sistema)
Postcondición	El usuario se encuentra logueado y listo para usar el sistema ¿Qué pasa cuando termina el caso de uso?
Nota	Acá va alguna aclaración
Actores	Usuario Personas que interactúan con el Sistema
Alternativas	4.1- El sistema muestra pantalla de error con el mensaje “Sus credenciales son incorrectas, intente nuevamente.” Acá van las acciones en el caso de que el camino no sea feliz, por ejemplo, que pasa si el usuario o la clave con incorrectas?

CU 2. Registrar usuario

CU 2. Registrar usuario	
Descripción	El usuario nuevo desea registrarse para poder usar el sistema
Precondición	-

Acciones	1- El usuario ingresa a la pantalla de registración. 2- El sistema le solicita los datos personales. 3- El usuario ingresa los datos solicitados y presiona “Registrar”. 4- El sistema envía un mail confirmando la registración exitosa.
Postcondición	El usuario se registrado y listo para loguearse.
Nota	-
Actores	Usuario
Alternativas	-