**Proyecto con Spring Tool Suite**

**java -version EN CMD y echo %JAVA\_HOME% para validar la instalación correcta del java.**

**Agregar la ruta en JRE isntaler del JDK y en compiler cambiamos la JDK a 11 porque por defecto viene otro jdk.**

**package** com.ibero.demo;

**import** org.slf4j.Logger;

**import** org.slf4j.LoggerFactory;

**import** org.springframework.boot.CommandLineRunner;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

**public** **class** Consola01Application **implements** CommandLineRunner{

**private** **static** Logger *LOG* = LoggerFactory.*getLogger*(Consola01Application.**class**);

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(Consola01Application.**class**, args);

}

@Override

**public** **void** run(String... args) **throws** Exception {

//System.out.println("Hola coders");

*LOG*.info("Hola coders");

*LOG*.warn("Con cuidado");

}

}

**MENSAJE EN CONSOLA**

Qualifier

@Qualifier("persona2") 🡪 agregar este debajo de repository cuando se tenga conflictos con los repositorios.

Y en el service también indicar el repositorio con @Qualifier("persona2") después de @Autowired

@Repository stereotype que indica que es una clase de acceso a datos.

@Service stereotype que indica a la clase que es una lógica de negocios.

@Autowired stereotype trae una instancia definido.

**INYECCIÓN DE DEPENDENCIAS:**

Dependencias

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-thymeleaf</artifactId>

</dependency>

Asignar puertos en properties

#Puerto del servidor

server.port=9080

Clase controller

**package** com.ibero.demo.controller;

**import** org.springframework.stereotype.Controller;

**import** org.springframework.ui.Model;

**import** org.springframework.web.bind.annotation.GetMapping;

**import** org.springframework.web.bind.annotation.RequestParam;

@Controller

**public** **class** DemoController {

@GetMapping("/greeting")

**public** String greeting(@RequestParam(name = "name",required = **false**,defaultValue = "World")String name,Model model) {

model.addAttribute("name",name);

**return** "greeting";}}

Pagina de Theamlif y HTML

<!DOCTYPE HTML>

<html xmlns:th=*"http://www.thymeleaf.org"*>

<head>

<title>Spring Web con Thymeleaf</title>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=UTF-8"* />

</head>

<body>

<p th:text=*"'Hello, ' + ${name} + '!'"* />

</body>

</html>

Enviar nombre por http http://localhost:9080/greeting?name=Raul

**SPRING WEB CON THEAMLEAF**

Dependencia JPA

<!-- JPA en Spring -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<!-- Postgresql -->

<dependency>

<groupId>org.postgresql</groupId>

<artifactId>postgresql</artifactId>

<scope>runtime</scope>

</dependency>

<!-- conector JDCB for Mysql -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<scope>runtime</scope>

</dependency>

<!-- Thymeleaf -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-thymeleaf</artifactId>

</dependency>

<!-- JPA en Spring -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

--conexión de base de datos postgresqlen properties

#JPA Y CONEXIÓN POSTGRESQL

spring.jpa.database=POSTGRESQL

spring.datasource.driver-class-name=org.postgresql.Driver

spring.datasource.url=jdbc:postgresql://localhost/demo

spring.datasource.username=postgresql

spring.datasource.password=123

#Dev

spring.jpa.show-sql=true 🡪 Muestra en consola las sentencias SQL

spring.jpa.hibernate.ddl-auto=update

logging.level.org.hibernate.SQL=debug 🡪Muestra el debug del sql con spring

spring.jpa.properties.hibernate.jdbc.lob.non\_contextual\_creation=true

#JPA Y CONEXIÓN MYSQL

# DATASOURCE

spring.datasource.driverClassName=com.mysql.cj.jdbc.Driver

spring.datasource.url=jdbc:mysql://localhost:3306/iberodb?characterEncoding=UTF-8

spring.datasource.username=root

spring.datasource.password=12345

#Dialect

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5InnoDBDialect o

spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

Dialect actualizado org.hibernate.dialect.MySQLDialect

Si sucede un error de Zona Horaria agregar : ?ServerTimezone=America/Lima

Ejm: pring.datasource.url=jdbc:mysql://localhost:3306/db\_udemy?ServerTimezone=America/Lima

Error al parar la web agregar ………&useSSL=False

**SPRING DATA (JPA) CON SPRING MVC**

**DB POSTGRESQL o MYSQL**

#CREACION DE LA ENTIDAD

**package** com.ibero.demo.entity;

**import** javax.persistence.Column;

**import** javax.persistence.Entity;

**import** javax.persistence.Id;

@Entity

**public** **class** Persona {

@Id

**private** **int** idPersona;

@Column(name="nombre",length = 50)

**private** String nombre;

Métodos de accesos

}

#creamos la interfaz repository dentro del paquete repository extiende de JpaRepository

package com.ibero.demo.repository;

import org.springframework.data.jpa.repository.JpaRepository;

import com.ibero.demo.entity.Persona;

public interface IPersonaRepo extends JpaRepository<Persona, Integer>{

}

#en el controller anterior creamos un método

@GetMapping("/gredlist")

**public** String greeting(Model model) {

model.addAttribute("personas",repo.findAll());

**return** "greeting";

}

#HTML para mostrar la lista de las personas en base de datos

<table>

<th:block th:each=*"per :${personas}"*>

<tr>

<td th:text=*"${per.idPersona}"*></td>

<td th:text=*"${per.nombre}"*></td>

</tr>

</th:block>

</table>

#SPRING SECURITY DEPENDENCIA

<!-- Spring security -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

#Ejemplo poner usuario y clave en properties

#SECURITY

spring.security.user.name=benito

spring.security.user.password=123

#Entity

package com.ibero.demo.entity;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.Id;

@Entity

public class Usuario {

@Id

private int id;

@Column(name="nombre",length = 10)

private String nombre;

@Column(name="clave")

private String clave;

**Métodos de acceso**

}

#Interfaz de IUsuarioRepository

**package** com.ibero.demo.repository;

**import** org.springframework.data.jpa.repository.JpaRepository;

**import** com.ibero.demo.entity.Usuario;

**public** **interface** IUsuarioRepo **extends** JpaRepository<Usuario, Integer>{}

**Agregar spring security al proyecto**

**package** com.ibero.demo;

**import** **static** org.junit.jupiter.api.Assertions.*assertTrue*;

**import** org.junit.jupiter.api.Test;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.boot.test.context.SpringBootTest;

**import** org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

**import** com.ibero.demo.entity.Usuario;

**import** com.ibero.demo.repository.IUsuarioRepo;

@SpringBootTest

**class** Practica01ApplicationTests {

@Autowired

**private** IUsuarioRepo repo;

--para coificar en formato no leíble la contraseña

@Autowired

**private** BCryptPasswordEncoder encoder;

@Test

**void** crearUsuarioTest() {

Usuario user= **new** Usuario();

user.setId(2);

user.setNombre("Diego");

user.setClave(encoder.encode("123"));

Usuario reto = repo.save(user);

*assertTrue*(reto.getClave().equalsIgnoreCase(user.getClave()));

}

}

Si hay errores con la vista:

spring.thymeleaf.prefix=classpath:/templates/

spring.thymeleaf.suffix=.html

**Pasar un valor o cadena de caracteres a la vista desde el controlador.**

@GetMapping({"/index","/","/home"})

@RequestMapping(value = "/home", method = RequestMethod.***GET***)

**public** String showHome(Model model) {

model.addAttribute("titulo", "Hola Spring framework desde Model!");

**return** "home";

}

Recoger el texto en el index o home con thymelief.

Primero agregamos la llamada el nexspache

<html xmlns:th="http://www.thymeleaf.org">

Recién utilizar donde queremos que nos aparezca

<title th:text="${titulo}"></title>

O

<h1 th:text="${titulo}"></h1>

**O**

@RequestMapping(value = "/home", method = RequestMethod.***GET***)

**public** String showHome(ModelMap model2) {

model2.addAttribute("h1", "Hola Spring framework desde ModelMap!");

**return** "home";

}

@RequestMapping(value = "/home", method = RequestMethod.***GET***)

**public** String showHome(Map<String, Object> map) {

map.put("h2", "Hola spring desde Map");

**return** "home";

}

@RequestMapping(value = "/home2", method =RequestMethod.***GET***)

**public** ModelAndView showModel(ModelAndView model3) {

model3.addObject("titulo", "Hola desde ModelAndView Spring Framework!");

model3.setViewName("home");

**return** model3;

}

**RequestMapping sobre el controlador**

**import** org.springframework.web.bind.annotation.RequestMapping;

@RequestMapping("/app")

En el metodo

@GetMapping({"/index","/","/home"})

La ruta quedaría : [http://localhost:8080/app/home](http://localhost:8080/app/home%20/) | <http://localhost:8080/app/> | <http://localhost:8080/app/index>

Objeto model.

**package** edu.ibero.demo.models;

**public** **class** Usuario {

**private** String nombre;

**private** String apellido;

**private** String email;

Getter and setter - Constructor

Controller

@RequestMapping("/perfil")

**public** String perfiModel(Model model) {

Usuario user = **new** Usuario();

user.setNombre("Juan");

user.setApellido("Rodriguez");

model.addAttribute("usuario", user);

model.addAttribute("h1", "Perfil del usuario : " .concat(user.getNombre()));

**return** "perfil";

}

Vista Perfil

@RequestMapping("/listar")

**public** String listarView(Model model) {

List<Usuario> usuarios= **new** ArrayList();

usuarios.add(**new** Usuario("Andres","Perez","perez@gmail.com"));

usuarios.add(**new** Usuario("Mario","Gomez","mario@gmail.com"));

usuarios.add(**new** Usuario("Rildo","Mozombite","mozombite@gmail.com"));

model.addAttribute("usuarios", usuarios);

**return** "listar";

}

Directiva if de thymeleaf – En la vista de perfil agregamos th:if =”${usuario.email}”

<span th:if="${usuario.email}" th:text="${usuario.email}"></span>

Else

<span th:if="${usuario.email==null}" th:text="'El usuario no tiene correo'"></span>

Otra <div th:if="${usuarios.size()==0}">No hay usuarios en la lista!</div>

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head>

<meta charset="UTF-8">

<title>Insert title here</title>

</head>

<body>

<h1 th:text="${h1}"></h1>

<span th:text="${usuario.nombre}"></span>

<span th:text="${usuario.apellido}"></span> <br>

<span th:text="${usuario.email}"></span>

</body>

</html>

**VISTA PERFIL**

<!DOCTYPE html>

<html>

<html xmlns:th="http://www.thymeleaf.org">

<meta charset="UTF-8">

<title>Insert title here</title>

</head>

<body>

<h1 th:text="${h1}"></h1>

<h1 th:text="${h2}"></h1>

<div th:if="${usuarios.size()==0}">No hay usuarios en la lista!</div>

O

<div th:if="${usuarios.isEmpty()}">No hay usuarios en la lista!</div>

<table th:if="${not usuarios.isEmpty()}">

O

<table th:if="${usuarios.size()>0}">

<thead>

<tr>

<th>Nombres</th>

<th>Apellidos</th>

<th>Correos</th>

</tr>

</thead>

<tbody>

<tr th:each="usuario:${usuarios}">

<td th:text="${usuario.nombre}"></td>

<td th:text="${usuario.apellido}"></td>

<td th:text="${usuario.email}"></td>

</tr>

</tbody>

</table>

</body>

</html>

También se puede trabajar así:

@RequestMapping("/listar")

**public** String listarView(Model model, ModelMap model2) {

List<Usuario> usuarios = Arrays.*asList*(

**new** Usuario("Andres2", "Perez", "perez@gmail.com"),

**new** Usuario("Mario2", "Gomez", "mario@gmail.com"),

**new** Usuario("Rildo2", "Mozombite","mozomb@gmail.com"));

model.addAttribute("usuarios", usuarios);

**return** "listar";

Y el mismo HTML LISTAR.

ModelATRIBUTE --- Pasar datos comunes

@ModelAttribute("usuarios")

**public** List<Usuario> poblarUsuarios() {

List<Usuario> usuarios = Arrays.*asList*(

**new** Usuario("Andres2", "Perez", "perez@gmail.com"),

**new** Usuario("Mario2", "Gomez", "mario@gmail.com"),

**new** Usuario("Rildo2", "Mozombite", "mozom@gmail.com"));

**return** usuarios; }

Ejemplo con Params Recibimos un parámetro del url HTTP.

@GetMapping("/string")

**public** String param(@RequestParam(defaultValue = "Escribe el parametro") String texto, Model model) {

model.addAttribute("resultado","El parametro recibido es : " + texto);

**return** "params/ver";}

En el HTML

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head>

<meta charset="UTF-8">

</head>

<body>

<div th:text="${resultado}"></div>

</body>

</html>

@GetMapping("/")

**public** String index() {

**return** "params/index";

}

html

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head><meta charset="UTF-8">

<title>Enviar datos HTTP GET - URL</title></head>

<body>

<ul>

<li><a th:href="@{/params/string(texto='Hola Andres')}">/params/string?texto=Hola Andrés</a>

</li>

<li><a th:href="@{/params/mix-params(saludo='Como estás Andres',numero=7)}">/params/mix-params?saludo=Como estás Andres&numero=7</a></li>

<li><a th:href="@{/params/mix-params-request(saludo='Como estás Andres',numero=7)}">/params/mix-params-request?saludo=Como estás Andres&numero=7</a></li>

</ul>

</body>

</html>

@GetMapping("/mix-params")

**public** String param(@RequestParam String saludo,@RequestParam Integer numero, Model model) {

model.addAttribute("resultado","El saludo recibido es : '" + saludo + "' y el número es '" +numero+"'");

**return** "params/ver";}

@GetMapping("/mix-params-request")

**public** String param(HttpServletRequest request, Model model) { String saludo=request.getParameter("saludo");

Integer numero=**null**;

**try** {numero=Integer.*parseInt*(request.getParameter("numero"));}

**catch** (NumberFormatException e) {numero=0;}

model.addAttribute("resultado","El saludo recibido es : '" + saludo + "' y el número es '" +numero+"'");

**return** "params/ver";

Anotaciones @PathVariable

Creamos un controlador nuevo – con su ruta

@RequestMapping("/variables") //Sobre el controlador

@GetMapping("/string/{texto}")

**public** String variables(@PathVariable(name = "texto") String textoopc, Model model) {

model.addAttribute("h1", "Recibir parámetros de la ruta(PathVariable)");

model.addAttribute("resultado", "El texto enviado en la ruta es: " + textoopc);

**return** "variables/ver";}

Vista ver HTML

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head>

<meta charset="UTF-8">

<title th:text="${h1}"></title>

</head>

<body>

<h1 th:text="${h1}"></h1>

<div th:text="${resultado}"></div>

</body>

</html>

Varios parámetros con @PathVariable - metodo

@GetMapping("/string/{texto}/{numero}")

**public** String variables(@PathVariable(name = "texto") String textoopc,@PathVariable Integer numero, Model model) {

model.addAttribute("h1", "Recibir parámetros de la ruta(@PathVariable)");

model.addAttribute("resultado", "El texto enviado en la ruta es: " + textoopc +" y el número enviado en el path es: " + numero);

**return** "variables/ver";}

Método del index

@GetMapping("/")

**public** String index(Model model) {

model.addAttribute("h1", "Enviar parámetros de la ruta(@PathVariable)");

**return** "variables/index";}

Vista del Index

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head>

<meta charset="UTF-8">

</head>

<body>

<h1 th:text="${h1}"></h1>

<ul>

<li> <a th:href="@{/variables/string/} + 'Hola Andrés'">/variables/string/Hola Andrés</a></li>

<li> <a th:href="@{/variables/string/} + 'Que tal John'">/variables/string/Que tal John</a></li>

<li> <a th:href="@{/variables/string/} + 'saludos Andrés' + '/' + 7">/variables/string/Saludos Andrés/7</a></li>

<li> <a th:href="@{/variables/string/} + ${'saludos Andrés con llaves'} + '/' + ${'8'}">/variables/string/Saludos Andrés/8</a></li>

</ul>

</body>

</html>

Reutilizamos la vista de ver para el método que envía varias variables.

Creando Hoja de estilo CCS EN LA RUTA : /demo.spring.web/src/main/resources/static/css/style.css

h1.color-rojo{

color:red;

}

h2.color-azul{

color:blue;

}

Enlazamos la Hoja css con el HTML

<link rel="stylesheet" type="text/css" th:href="@{/css/style.css}"></link>

Agregar imagen en la ruta /demo.spring.web/src/main/resources/static/images/joins.png

<img alt="" th:src="@{/images/joins.png}"></img>

Inyectar valores usando la anotación @Value

Agregamos los valores en propesties.

texto.indexcontroller.index.titulo:Hola Spring Framework desde Aplication.properties!

texto.indexcontroller.perfil.titulo:Perfil del usuario desde Aplication.properties!

texto.indexcontroller.listar.titulo:Listado desde Aplication.properties!

Llamamos esos valores desde el controles con @Value

@Value("${texto.indexcontroller.index.titulo}")

**private** String textoIndex;

@Value("${texto.indexcontroller.perfil.titulo}")

**private** String textoPerfil;

@Value("${texto.indexcontroller.listar.titulo}")

**private** String textoListar;

en los métodos handler llamamos a las variables inyectadas

model.addAttribute("h1", "Perfil del usuario :".concat(user.getNombre())); ó

model.addAttribute("h3", textoPerfil.concat(user.getNombre())); ó

model.addAttribute("titulo", textoIndex);

Como solucionar warning ddel properties. Presionamos sobre el warning y presionar crear metadata.

Agregar otros archivos properties para los textos.

Copiamos y pegamos el properties con todo sus atributos de texto a inyectar.

textos.properties

texto.indexcontroller.index.titulo:Hola Spring Framework desde Aplication.properties!

texto.indexcontroller.perfil.titulo:Perfil del usuario:

texto.indexcontroller.listar.titulo:Listado :

En la raíz del proyecto creamos una nueva clase.

package edu.ibero.demo;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.PropertySource;

import org.springframework.context.annotation.PropertySources;

@Configuration

@PropertySources({ @PropertySource("classpath:textos.properties") })

public class TextosPropertiesConfig { }

Redirect y fordward.

@GetMapping("/")

**public** String home() {

**return** "redirect:/app/home";

También puede redirigir a una página externa.

**return** "redirect:https://www.google.com";}

Otra forma, sin reiniciar el request y sin recargar la pagina Forward solo a rutas del proyecto no externos.

@GetMapping("/")

**public** String home() { **return** "forward:/app/home"; }

Genarar el proyecto en Maven.

Abrir la ruta del proyecto CD.

Ejecutamos .\mvnw.cmd package

Ejecutamos java -jar ruta target…..

Nuevo Proyecto.

Crear Proyecto nuevo, creamos paquete controlers y models.

Creación del controlador.

package com.bolsadeideas.springboot.di.app.controllers;

@Controller

public class IndexController {

@GetMapping({"/", "", "/index"})

public String index(Model model) {

return "index"; }}

Agregar service.

**public** **class** MiServicio **implements** IServicio{

@Override

**public** String operacion() {

**return** "ejecutando algún proceso importante simple ...";}}

Llamado desde el controlador mediante una instancia del objeto.

**private** MiServicio servicio = **new** MiServicio();

quedaría el método.

@Controller

public class IndexController {

@GetMapping({"/", "", "/index"})

public String index(Model model) {

return "index"; }}

En el HTML – Sin inyección de dependencia.

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head> <meta charset="UTF-8"></head>

<body>

<div th:text="${objeto}"></div>

</body>

</html>

Aplicando inyección de dependencia con @autowired

Tenemos que hacer que nuestro servicio sea un componente este dentro del contenedor.

Agregando en la clase @Component o @service (Service aporta una semantica)

@Component

**public** **class** MiServicio {….}

Ahora inyectamos un objeto que esta registrado en el contenedor de spring con @Autowired.

@Autowired

**private** MiServicio servicio;

Si probamos funciona exactamente igual.

Inyectar mediante la interfaz.

Creamos un Interfaz del Service.

**public** **interface** IServicio {

**public** String operacion();

}

Implementamos el service.

**public** **class** MiServicio **implements** IServicio

Llamamos al recurso desde el servicio.

@Autowired

**private** IServicio servicio;

Al component también se lo puede dar un nombre.

@Component("miServicioSimple")

Inyección de dependencia via constructor y método setter.

Implemnatamos el setter de la interfaz y movemos la etiqueta

@Autowired

**public** **void** setServicio(IServicio servicio) {

**this**.servicio = servicio;}

Via constructor

@Autowired

**public** IndexController(IServicio servicio) {

**this**.servicio = servicio;}//Cuando inyectamos por constructor se puede omitir el @Autowired.

Anotación Primary : Cuando tengamos dos clases component o service para indicar el candidato que queremos inyectar por defecto agregamos @primary arriba de la clase component.

**import** org.springframework.context.annotation.Primary;

@Primary

Utilizando Qualifired.

@Autowired

@Qualifier("miServicioComplejo") 🡨 Llamamos al recurso por sus nombre con la etiqueta Qualifier.

**private** IServicio servicio;

@Qualifier También lo podemos utilizar sin Primary.

Registrando componentes usando clase de configuración y la anotación @Bean

Quitamos @component y primary de la clase component.

En la base del proyecto creamos una clase llamada AppConfig, agregamos la etiqueta @Configuratión

@Configuration

**public** **class** AppConfig {

@Bean("miServicioSimple")

@Primary **public** IServicio registrarMiServicio() { **return** **new** MiServicio(); }

@Bean("miServicioComplejo") **public** IServicio registrarMiServicioComplejo() {

**return** **new** MiServicioComplejo(); }

También en el controlador podemos usar igualmente @Qualifier.

Ejemplo Factura y relaciones entre sus componentes

Creamos un paquete domain o entity.

Creamos las clases Cliente – Factura- ItemFactura – Producto.

Clase Cliente:

@Component

@RequestScope

**public** **class** Cliente {

@Value("${cliente.nombre}")

**private** String nombre;

@Value("${cliente.apellido}")

**private** String apellido;

Getter and Setter

}

Clase Factura.

@Component

@RequestScope

**public** **class** Factura **implements** Serializable{

**private** **static** **final** **long** ***serialVersionUID*** =946004357128146951L;

@Value("${factura.descripcion}")

**private** String descripcion;

@Autowired

**private** Cliente cliente;

@Autowired

**private** List<ItemFactura> items;

@PostConstruct

**public** **void** inicializar() {

cliente.setNombre(cliente.getNombre().concat(" ").concat("José"));

descripcion = descripcion.concat(" del cliente: ").concat(cliente.getNombre());

}

@PreDestroy

**public** **void** destruir() {

System.***out***.println("Factura destruida: ".concat(descripcion));

}

Métodos Getter and Setter.

Clase ItemFactura

**public** **class** ItemFactura {

**private** Producto producto;

**private** Integer cantidad;

Getter and setter and constructor.

Método.

**public** Integer calcularImporte() {

**return** cantidad \* producto.getPrecio();}

Clase Producto.

**public** **class** Producto {

**private** String nombre;

**private** Integer precio;

Configuración de propesties para textos.

cliente.nombre: Andr&eacute;s

cliente.apellido: Guzm&aacute;n

factura.descripcion: Factura Deporte

En appConfig.

@Bean("itemsFactura")

**public** List<ItemFactura> registrarItems(){

Producto producto1 = **new** Producto("Camara Sony", 100);

Producto producto2 = **new** Producto("Bicicleta Bianchi aro 26", 200);

ItemFactura linea1 = **new** ItemFactura(producto1, 2);

ItemFactura linea2 = **new** ItemFactura(producto2, 4);

**return** Arrays.*asList*(linea1, linea2);}

@Bean("itemsFacturaOficina")

@Primary

**public** List<ItemFactura> registrarItemsOficina(){

Producto producto1 = **new** Producto("Monitor LG LCD 24", 250);

Producto producto2 = **new** Producto("Notebook Asus", 500);

ItemFactura linea1 = **new** ItemFactura(producto1, 2);

ItemFactura linea2 = **new** ItemFactura(producto2, 1);

**return** Arrays.*asList*(linea1, linea2, linea3, linea4);}

@Controller

@RequestMapping("/factura")

**public** **class** FacturaController {

@Autowired

**private** Factura factura;

@GetMapping("/ver")

**public** String ver(Model model) {

model.addAttribute("factura", factura);

model.addAttribute("titulo", "Ejemplo Factura con inyección de dependencia");

**return** "factura/ver";}}

Ciclo de Vida de un componente Spring.

**import** javax.annotation.PostConstruct;

**import** javax.annotation.PreDestroy;

Método para modificar el valor de cliente y también añade el nombre a descripción.

@PostConstruct

**public** **void** inicializar() {

cliente.setNombre(cliente.getNombre().concat(" ").concat("José"));

descripcion = descripcion.concat(" del cliente: ").concat(cliente.getNombre());}

Método para destruir el objeto.

@PreDestroy

**public** **void** destruir() {

System.***out***.println("Factura destruida: ".concat(descripcion));}

Contextos scope de spring

@RequestScope

En la clase u entity.

@SessionScope 🡪 para trabajar con sesiones marcar nuestro obketo de sesión.

@ApplicationScope 🡪

**Agregar dependencia de Validación 🡪 Desde spring 2.3.x agregar**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-validation</artifactId>

</dependency>

Para agregar archivos HTML en spring tool suite tenemos que instalar Eclipse Web Developer

Para manejo del Thymeleaf en el formulario o pagina html agergagos el XMLS

<html xmlns:th="http://www.thymeleaf.org">

Siempre las eqiquetas tiene TH emeplo th:text=${titulo}

Capturamos valores enviados desde el HTml con @RequestParam String username

O también se puede hacer @RequestParam (name=username)String username

Pasamos eos valores recibido al html resultado con MODEL.

model.addAttribute("username", username);

Mostramos en el HTML th:text="${username}"

Creamos una clase Usuario con sus atributos y sus métodos getter and setter.

**private** String username;

**private** String password;

**private** String email;

Creamos una instancia del objeto.

Usuario user= new Usuario();

Agregamos valores con el método set de los atributos.

user.setUsername(username);

user.setPassword(password);

user.setEmail(email);

Enviamos a la vista el objeto con model.

model.addAttribute("usuario", user);

Al mostrar en la vista seria th:text="${usuario.username}"

También podemos automatizar el objeto agregando el Usuario user en el parámetro del método.

Atributo email en el formulario : type="email"

**Sobre los package javax vs jakarta**

Tengan en cuenta para los siguientes videos que en Spring Boot 3 se cambia javax por jakarta, en el api de validación de Java (Java API Bean validation), por ejemplo en Spring Boot 3 se importa de:

1. import jakarta.validation.Valid;
2. import jakarta.validation.constraints.NotEmpty;
3. import jakarta.validation.constraints.Size;
4. import jakarta.validation.constraints.Email;
5. import jakarta.validation.constraints.NotNull;
6. import jakarta.validation.constraints.Pattern;
7. import jakarta.validation.constraints.Max;
8. import jakarta.validation.constraints.Min;
9. etc...

Y antes en Spring Boot 2 se importaba de:

1. import javax.validation.Valid;
2. import javax.validation.constraints.NotEmpty;
3. import javax.validation.constraints.Size;
4. import javax.validation.constraints.Email;
5. import javax.validation.constraints.NotNull;
6. import javax.validation.constraints.Pattern;
7. import javax.validation.constraints.Max;
8. import javax.validation.constraints.Min;
9. etc...

Esto se debe a que Spring Boot 2 usa por debajo (tras bambalinas) en spring framework Java EE 8 y en Spring Boot 3 por debajo usa Jakarta EE 9.

Validación del formulario usando anotaciones

@Valid Usuario user -> dentro de los parámetros del objeto.

Para manejar el resultado en el parámetro del método llamamos a BindingResult result : representa el resultado de la validación.

Siempre tiene que estar después del objeto que se valida.

Validamos con un if

**if** (result.hasErrors()) {

Map<String, String>errores = **new** HashMap<>();

result.getFieldErrors().forEach(err ->{

errores.put(err.getField(),"El campo ".concat(err.getField()).concat(" ").concat(err.getDefaultMessage())); }); model.addAttribute("error", errores);

return “form”;}

Mostramos en la vista el error.

th:if="${error != null && error.containsKey('username')}" th:text="${error.username}" value muestra nuevamente los valores cuando sucede un error th:value="${user.username}

Cuando sale el errores por error de que en el get donde carga el formulario no se esta enviando una instancia del usuario.

Agregar @Valid @ModelAttribute("user") Usuario user

El atributo object de thymeleaf en el elemento form

<input type="text" name="username" id="username" th:value="${user.username}"> O también puede ser de la siguiente manera. <input type="text" id="username" th:field="\*{username}">

En la etiqueta forme entonces llamamos al objeto que estará mapenado en cada uno de los campos.

th:object="${usuario}"

quitamos el error del controlador porque se va manejar de forma automática por el framework.

<div th:if="${#fields.hasErrors('apellido')}" th:errors="\*{apellido}"></div>

Mostrar valores de atributos del objeto model en el formulario

Al cargar el formulario en el método handler creamos un objeto de la clase y lo instanciamos con el metod set.

Usuario user = **new** Usuario();

user.setNombre("John");

user.setApellido("Doe");

user.setIdentificador("123.456.546-K");

model.addAttribute("usuario", user);-> enviamos el objeto a la vista.

Mostramos en el resultado del formulario.

<ul>

<li th:text="${usuario.identificador}"></li>

<li th:text="${usuario.nombre}"></li>

<li th:text="${usuario.apellido}"></li>

<li th:text="${usuario.username}"></li>

<li th:text="${usuario.password}"></li>

<li th:text="${usuario.email}"></li>

</ul>

La anotación @SessionAttributes para manter los datos durante el ciclo del form

Agregamos la anotación arriba de la clase controller.

@SessionAttributes("usuario")

Luego en el método pos capturamos el estado de la sesión SessionStatus status

Y cerramos la sesión. status.setComplete();

La anotación @Size para rangos y @Email para validar correo electrónico

Agregarmos en los atributos

@Email

**private** String email;

@Size(min=3,max=8) 🡪Solo para String.

**private** String username;

Personalizar el mensaje.

@NotEmpty(message = "El nombre no puede estar vacio")

@Email(message = "Correo con formato incorrecto")

Mensajes de errores personalizados usando properties

Creamos un archivo properties para los mensajes.

messages.properties

Agregamos en el contenido del propesrties e indicamos el mensaje personalizado. Esto aplicaría todas las etiquetas.

NotEmpty=El campo no puede ser vacío desde properties

También podemos personalizar un mensaje de error del campo, indicando su ruta completa.

NotEmpty.usuario.nombre=elcampo nombre no debe estar vacio prop.

Archivos properties y acentos con UTF-8

Vamos al archivo properties mensaje y cambiamos de formato.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Validación personalizada usando anotación @Pattern para expresiones regulares

@Pattern(regexp = "[0-9]{2}[.,][\\d]{3}[.,][\\d]{3}[-][A-Z]{1}")

Validación personalizada usando una clase Validation

Creamos un paquete y una clase implementando la validator y anotar con component

@Component

**class** UsuarioValidador **implements** Validator

@Override

**public** **boolean** supports(Class<?> clazz) {

**return** Usuario.**class**.isAssignableFrom(clazz);}

@Override

**public** **void** validate(Object target, Errors errors) {

Usuario usuario = (Usuario)target;

ValidationUtils.*rejectIfEmpty*(errors,"nombre","NotEmpty.usuario.nombre");

**if**(usuario.getNombre().isEmpty()) {

errors.rejectValue("nombre","NotEmpty.usuario.nombre");

**if**(usuario.getIdentificador().matches("[0-9]{2}[.,][\\d]{3}[.,][\\d]{3}[-][A-Z]{1}")==**false**) {

errors.rejectValue("identificador","pattern.usuario.identificador"); }

En el controller inyectamos el objeto.

@Autowired

**private** UsuarioValidador validador;

En el método POS agregamos.

validador.validate(user, result);

Validamos rtambien para que no acepte espacios en blanco

ValidationUtils.*rejectIfEmptyOrWhitespace*(errors,"nombre", "NotEmpty.usuario.nombre");

O también podemos requerido.usuario.nombre

Registrando una clase validador con la anotación @InitBinder

@NotBlank 🡪valida si no es espacio en blanco o nulo.

Agregamos en el properties.

requerido.usuario.nombre=El nombre no puede ser vacio.

@InitBinder

**public** **void** initBinder(WebDataBinder binder) {

binder.setValidator(validador);}🡪se pierde la validación con etiquetas.

Para que acepte las demás validaciones agregamos. binder.addValidators(validador);

Validación personalizada usando anotaciones

Creamos una clase validadora.

package edu.ibero.demo.form.app.validation;

import static java.lang.annotation.ElementType.FIELD;

import static java.lang.annotation.ElementType.METHOD;

import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Retention;

import java.lang.annotation.Target;

import jakarta.validation.Constraint;

import jakarta.validation.Payload;

@Constraint(validatedBy = IdentificadorRagexValidador.class)

@Retention(RUNTIME)

@Target({ FIELD, METHOD })

public @interface IdentificadorRagex {

String message() default "Identificador inválido";

Class<?>[] groups() default { };

Class<? extends Payload>[] payload() default { };}

Creamos otra clase.

package edu.ibero.demo.form.app.validation;

import jakarta.validation.ConstraintValidator;

import jakarta.validation.ConstraintValidatorContext;

public class IdentificadorRagexValidador implements ConstraintValidator<IdentificadorRagex, String>{

@Override

public boolean isValid(String value, ConstraintValidatorContext context) {

if(value.matches("[0-9]{2}[.,][\\d]{3}[.,][\\d]{3}[-][A-Z]{1}")) {

return true;

}

return false;

En la clase validamos con @IdentificadorRagex

Arriba del atributo de la clase.

Validación personalizada usando anotaciones ejemplo requerido

package edu.ibero.demo.form.app.validation;

import static java.lang.annotation.ElementType.FIELD;

import static java.lang.annotation.ElementType.METHOD;

import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Retention;

import java.lang.annotation.Target;

import jakarta.validation.Constraint;

import jakarta.validation.Payload;

@Constraint(validatedBy = RequeridoValidador.class)

@Retention(RUNTIME)

@Target({ FIELD, METHOD })

public @interface Requerido {

String message() default "El campo es requerido - usando anotaciones";

Class<?>[] groups() default { };

Class<? extends Payload>[] payload() default { };

**import** org.springframework.util.StringUtils;

**import** jakarta.validation.ConstraintValidator;

**import** jakarta.validation.ConstraintValidatorContext;

**public** **class** RequeridoValidador **implements** ConstraintValidator<Requerido, String> {

@Override

**public** **boolean** isValid(String value, ConstraintValidatorContext context) {

**if**(value == **null** || /\*value.isEmpty() || value.isBlank()\*/🡪!StringUtils.*hasText*(value)) {

**return** **false**;

}**return** **true**;}

Validación de números enteros con @Max @Min y NotNul

Agregar en el model un atributo.

private Integer cuenta;

Añadimos la anotación @NotNull🡪 que es para objeto , @NotEmpty es para string. Si el aributo fuera int entonces no se puede usar @NotNull porque no seria un objeto sino un primitivo, para variables primitivas se usa @Min es recomendable que sea un objeto es decir integer.

Al atributo cuenta agregamos @NotNull, @Min y @Max.

Al momento de querer enviar una cadena de texto en el cambo cuenta indica error Failed to convert property value of type java.lang.String to required type java.lang.Integer for property cuenta; For input string: "FH"

Para suluciónar este inconveniente con el error podemos personalizarlo agregando en el properties message typeMismatch.java.lang.Integer=Debe ser un número entero

Entonces al volver a enviar la cadena dirá el mensaje que lo indicamos. Eso sucede cuando el type del campo es libre se puede escribir sea un numero o cadena es decir es text también podemos restringir que solo sea numero para ese campo cambiándolo a number al campo.

Ojo : typeMismatch.java.lang.Integer= Texto 🡪es para todos los integers.

typeMismatch.usuario.cuenta = texto 🡪es personalizado para un campo.

Validación de fechas con @NotNull y dando formato con @DateTimeForma

Agregamos la variable

**private** Date fechaNacimiento;

Validamos con @NotNull, agregamos ese campo en el formulario.

Si presionamos enviar nos mostrará un mensaje de error extenso.

Failed to convert property value of type java.lang.String to required type java.util.Date for property fechaNacimiento; Failed to convert from type [java.lang.String] to type [@jakarta.validation.constraints.NotNull java.util.Date] for value []

Para manejar ese error tenemos que agregar en el mensaje properties.

typeMismatch.java.util.Date=El campo debe ser fecha formato Año/Mes/Día

Aunque también se puede poner un mensaje en el placeholder placeholder="yyyy/mm/dd"

Pero el formulario cambia al momento de presionar enviar, podemos personalizar el formato con @DateTimeFormat(pattern = "yyyy/mm/dd")

Entonces al presionar enviar el formato quedará como lo deseamos en el pattern, también podemos personalizar el mensaje para ese campo.

Validación de fechas con input type date de HTML5 y @Past y @Future

Cambiamos el type de fecha de nacimiento del form type="date"

Podemos validar @Past para fechas pasadas de la actual, @Future para fechas futuras a partir de la fecha actual.

Formateando fechas con @InitBinder y registrando un CustomDateEditor

En InitBinder implementamos o modiicamos.

@InitBinder

**public** **void** initBinder(WebDataBinder binder) {

binder.addValidators(validador);

SimpleDateFormat dateFormat = **new** SimpleDateFormat("yyyy-MM-dd");

dateFormat.setLenient(**false**);

binder.registerCustomEditor(Date.**class**, **new** CustomDateEditor(dateFormat, **false**));}

Formateando fechas con @InitBinder y registrando un CustomDateEditor Parte 2

Con register custom editor también podemos especificar el atributo de la clase entity donde se va aplicar binder.registerCustomEditor(Date.**class**,"fechaNacimiento",**new** CustomDateEditor(dateFormat, **false**)); - Class – Campo – date format

Implementando propio filtro custom property editor para convertir a mayúscula

Primero para tener un orden creamos una clase llamada NombreMayusculaEditor y sobre escribimos un mentodo.

**public** **class** NombreMayusculaEditor **extends** PropertyEditorSupport {

@Override

**public** **void** setAsText(String text) **throws** IllegalArgumentException { setValue(text.toUpperCase().trim());}

Luego en el binder del controller agregamos una línea de binder editor custom.

binder.registerCustomEditor(String.**class**, **new** NombreMayusculaEditor());

Con eso todos los campos string se van a cambiar de minúscula a mayúscula, aplicaría todos los strings, Aplicaría a solo un campo de la siguiente manera:

binder.registerCustomEditor(String.**class**,"nombre" ,**new** NombreMayusculaEditor());

Lista select desplegable

Agregamos el campo país en el entity usuario **private** String pais;

Gettter and setter de país, indicamos con @NotEmpty que no acepte vacios

Luego creamos un método donde creamos un arreglo de países.

@ModelAttribute("paises")

**public** List<String> paises(){

**return** Arrays.*asList*("España","México","Chile","Argentina","Perú","Colombia","Venezuela"); }

Agregamos a la vista u select donde cargaran los paises.

<div>

<label for="pais">País</label>

<div>

<select id="pais" th:field="\*{pais}">

<option value="">-- Seleccionar --</option>

<option th:each="pais:${paises}" th:text="${pais}" th:value="${pais}"></option>

</select>

</div>

<div th:if="${#fields.hasErrors('pais')}" th:errors="\*{pais}"></div>

</div>

Mostramos en el resultado el país seleccionado.

<li th:text="${usuario.pais}"></li>

Formateando fechas en vistas thymeleaf

Para dar dormato a la vita de la fecha aplicamos en el resultado lo siguiente.

th:text="${#dates.format(usuario.fechaNacimiento,'dd/MM/yyyy')}"

o También podemos hacer:

th:text="${#calendars.format(usuario.fechaNacimiento,'dd-MM-yyyy')}"

porque al no dar dormato se aplica según nuestra ubicación.

Llenando lista select con Map

Creamos un método nuevo.

@ModelAttribute("paisesMap")

**public** Map<String,String> paisesMap(){

Map<String,String> paises = **new** HashMap<String, String>();

paises.put("ES", "España");

paises.put("MX", "México");

paises.put("CL", "Chile");

paises.put("AR", "Argentina");

paises.put("PE", "Perú");

paises.put("CO", "Colombia");

paises.put("VE", "Venezuela");

**return** paises;}

Cambiamos en la vista:

<div>

<label for="pais">País</label>

<div>

<select id="pais" th:field="\*{pais}">

<option value=" ">-- Seleccionar --</option>

<option

th:each="pais:${paisesMap.entrySet()}" th:text="${pais.value}" th:value="${pais.key}">

</option>

</select>

</div>

<div th:if="${#fields.hasErrors('pais')}" th:errors="\*{pais}"></div>

</div>

Al enviar el valor enviado seria el key no el value.

Llenando lista select con objetos del la clase País

Creamos una clase País.

**public** **class** Pais {

@NotNull🡪Validamos que el objeto no sea nulo.

**private** Integer id;

**private** String codigo;

**private** String nombre; Getter and Setter, Constructores.

Cambiamos la variable de la clase Usuario:

@Valid 🡪 Para que valide el campo de la clase Pais

**private** Pais pais;

Creamos un método en el constructor para la población de países.

@ModelAttribute("listaPaises")

**public** List<Pais> listaPaises(){

**return** Arrays.*asList*(

**new** Pais(1,"ES","España"),

**new** Pais(2,"MX","México"),

**new** Pais(3,"CL","Chile"),

**new** Pais(4,"AR","Argentina"),

**new** Pais(5,"PE","Perú"),

**new** Pais(6,"CO","Colombia"),

**new** Pais(7,"VE","Venezuela"));}

En el formulario realizamos poblamos el select con los datos del método del constructor.

<div>

<label for="pais.id">País</label>

<div>

<select id="pais" th:field="\*{pais.id}">

<option value=" ">-- Seleccionar --</option>

<option th:each="pais:${listaPaises}" th:text="${pais.nombre}" th:value="${pais.id}"></option>

</select>

</div>

<div th:if="${#fields.hasErrors('pais.id')}" th:errors="\*{pais.id}"></div>

</div>

En el resultado verificamos que el dado a mostrar sea el validado y el enviado.

<li th:text="${usuario.pais.id}"></li>

Añadiendo componente service de Pais

Creamos un packete Service donde creamos la interface para manejar el país.

**public** **interface** PaisService {

**public** List<Pais> listar();

**public** Pais obtenerPaisId(Integer id);}

Creamos una clase donde implementamos la interface:

@Service

**public** **class** PaisServiceImpl **implements** PaisService{

**private** List<Pais> lista;

**public** PaisServiceImpl() {

**this**.lista = Arrays.*asList*(

**new** Pais(1,"ES","España"),

**new** Pais(2,"MX","México"),

**new** Pais(3,"CL","Chile"),

**new** Pais(4,"AR","Argentina"),

**new** Pais(5,"PE","Perú"),

**new** Pais(6,"CO","Colombia"),

**new** Pais(7,"VE","Venezuela"));}

@Override

**public** List<Pais> listar() {

**return** lista;}

@Override

**public** Pais obtenerPaisId(Integer Id) {

Pais resultado = **null**;

**for** (Pais pais : **this**.lista) {

**if**(Id== pais.getId()) {

resultado = pais;

**break**;}}

**return** resultado;}}

En el controller inyectamos la interface:

@Autowired

**private** PaisService paisService;

En el método ahora llamamos a la interfaz.

@ModelAttribute("listaPaises")

**public** List<Pais> listaPaises(){

**return** paisService.listar();}

Añadiendo componente pais property editor

Creamos una nueva clase llamada PaisPropertiesEditor.

@Component 🡪ANOTAMOS PARA PODER INYECTAR

**public** **class** PaisPropertyEditor **extends** PropertyEditorSupport{

@Autowired 🡪INYECTAMOS EL SERVICE

**private** PaisService paisService;

@Override

**public** **void** setAsText(String idString) **throws** IllegalArgumentException {

**try** { Integer id = Integer.*parseInt*(idString);

**this**.setValue(paisService.obtenerPaisId(id));

}**catch** (NumberFormatException e) {setValue(**null**);}

}**else** {setValue(**null**);}}}

Inyectamos en el controller

@Autowired

**private** PaisPropertyEditor paiseditor;

Luego registramos:

binder.registerCustomEditor(Pais.**class**,"pais",paiseditor);

Cambiamos la validación de país

@NotNull

**private** Pais pais;

y eliminamos la validación del atributo de país porque ahora valida el objeto.

En el formulario validamos el objeto país no el atributo de la clase país:

<div>

<label for="pais">País</label>

<div>

<select id="pais" th:field="\*{pais}">

<option value="">-- Seleccionar --</option>

<option th:each="pais:${listaPaises}" th:text="${pais.nombre}" th:value="${pais.id}"></option>

</select>

</div>

<div th:if="${#fields.hasErrors('pais')}" th:errors="\*{pais}"></div>

</div>

Mostramos los atributos del país enviado.

<li th:text="${usuario.pais.id}"></li>

<li th:text="${usuario.pais.codigo}"></li>

<li th:text="${usuario.pais.nombre}"></li>

Implementando checkboxes

Crear un arreglo en el model

@NotEmpty🡪No puede ser vacio

**private** List<String> roles;🡪Getter and Setter

Método handler para poblar los roles.

@ModelAttribute("listaRolesString")

**public** List<String> listaRolesString(){

List<String> roles = **new** ArrayList<>();

roles.add("ROLE\_ADMIN");

roles.add("ROLE\_USER");

roles.add("ROLE\_MODERATOR");

**return** roles;}

Implementamos el checkbox de los roles

<div>

<label >Roles</label>

<div th:each="role:${listaRolesString}">

<input type="checkbox" th:field="\*{roles}" th:value="${role}">

<label th:text="${role}"></label>

</div>

<div th:if="${#fields.hasErrors('roles')}" th:errors="\*{roles}"></div>

</div>

Agregar cuando hacemos clic en el nombre del rol también se selecione el Rol.

th:for="${#ids.prev('roles')}"

Llenando checkboxes con Map

Método para poblar los roles con Map

@ModelAttribute("listaRolesMap")

**public** Map<String,String> listaRolesMap(){

Map<String,String> roles = **new** HashMap<String, String>();

roles.put("ROLE\_ADMIN", "Administrador");

roles.put("ROLE\_USER", "Usuario");

roles.put("ROLE\_MODERATOR", "Moderador");

**return** roles;}

Modificamos en el formulario:

<div>

<label>Roles</label>

<div th:each="role:${listaRolesMap.entrySet()}">

<input type="checkbox" th:field="\*{roles}" th:value="${role.key}">

<label th:text="${role.value}" th:for="${#ids.prev('roles')}"></label>

</div>

<div th:if="${#fields.hasErrors('roles')}" th:errors="\*{roles}"></div>

</div>

Llenando checkboxes con objetos del tipo Role

Creamos una clase Role:

**public** **class** Role {

**private** Integer id;

**private** String nombre;

**private** String role;🡪Getter and Setter

En la clase usuario creamos una lista de roles:

@NotEmpty

**private** List<Role> roles;

Creamos una interfaz de los métodos abstractos.

**public** **interface** RoleService {

**public** List<Role> listar();

**public** Role obtenerPorId(Integer id);}

Clase service implements.

@Service

**public** **class** RoleServiceImpl **implements** RoleService{

**private** List<Role> roles;

**public** RoleServiceImpl() {

**this**.roles = **new** ArrayList<>();

**this**.roles.add(**new** Role(1, "Administrador" , "ROLE\_ADMIN"));

**this**.roles.add(**new** Role(2, "Usuario" , "ROLE\_USER"));

**this**.roles.add(**new** Role(3, "Moderador" , "ROLE\_MODERATOR"));}

@Override

**public** List<Role> listar() {

**return** roles;}

@Override

**public** Role obtenerPorId(Integer id) {

Role roleres = **null**;

**for** (Role role : **this**.roles) {

**if**(id== role.getId()) {

roleres = role;

**break**; }} **return** roleres;}}

Inyectamos roles en el constructor:

@Autowired

**private** RoleService roleService;

Además, creamos un método en el controler:

@ModelAttribute("listaRoles")

**public** List<Role> listaRoles(){

**return** **this**.roleService.listar();}

Implementamos el roleEditor

@Component

**public** **class** RolesEditor **extends** PropertyEditorSupport{

@Autowired

**private** RoleService roleService;

@Override

**public** **void** setAsText(String text) **throws** IllegalArgumentException {

**try** {

Integer id= Integer.*parseInt*(text);

setValue(roleService.obtenerPorId(id));

} **catch** (NumberFormatException e) {

setValue(**null**); }

}}

Agregamos el controller para el editorrole:

@Autowired

**private** RolesEditor roleEditor;

en el método InitBnder agregamos :

binder.registerCustomEditor(Role.**class**,"roles",roleEditor);

Para mostrar el resultado modificamos el resultado:

<div>Roles</div>

<ul>

<li th:each="role: ${usuario.roles}" th:text="${role.id}+ ': ' + ${role.nombre} + '=>'+${role.role}"></li>

</ul>

Checkbox booleano true o false

Creamos un rol estado:

**private** Boolean estado;

En el controlador en el GET format

user.setEstado(**true**);

Agregamos un checkbox en el formulario:

<div>

<label for="estado">Habilitar</label>

<div>

<input type="checkbox" id="estado" th:field="\*{estado}"></div></div>

Mostramos en el resultado:

<li th:text="'Estado: ' + ${usuario.estado}"></li>

Radio button

Agregamos un atributo género en el models

@NotEmpty🡪Validamos que no sea vacio

**private** String genero;Getter and Setter

Agregamos un campo en el formulario:

<div>

<label >Género</label>

<div>

<input type="radio" id="genero1" th:field="\*{genero}" value="Hombre">

<label for="genero1">Hombre</label>

</div>

<div>

<input type="radio" id="genero2" th:field="\*{genero}" value="Mujer">

<label for="genero2">Mujer</label>

</div>

<div th:if="${#fields.hasErrors('genero')}" th:errors="\*{genero}"></div>

</div>

Mostramos en el HTML resultado:

<li th:text="'Género: ' + ${usuario.genero}"></li>

Para tener de forma dinámica se agregar un modelatribute:

@ModelAttribute("genero")

**public** List<String> genero(){

**return** Arrays.*asList*("Hombre","Mujer");}

Modificamos el formulario para el genero de forma dinámica:

<div>

<label>Género</label>

<div th:each="sexo:${genero}">

<input type="radio" th:field="\*{genero}" th:value="${sexo}">

<label th:for="${#ids.prev('genero')}" th:text="${sexo}"></label>

</div>

<div th:if="${#fields.hasErrors('genero')}" th:errors="\*{genero}"></div>

</div>

Input type hidden

Agregamos un atributo en la clase model:

**private** String valorSecreto;

Indicamos un valor por defecto:

user.setValorSecreto("\*\*\*Algun valor secreto\*\*\*");

También utilizamos @SessionAttributes("usuario")

Para persistir datos que no deseamos que el usuario lo visualice también podemos utilizar los input type="hidden".

Poblando pais y roles de forma automática:

Modificamos el método get donde se carga la vista con los valores por defecto:

user.setPais(**new** Pais(3,"CL","Chile"));

Al revisar el formulario no se muestra el país seleccionado eso porque nosotros lo estamos mandando un objeto a la vista y en el formulario esta con el id th:value="${pais.id}", entonces para que el formulario pueda validar y comparar el id tenermos que convertir ese valor a integer a string, entonces en la clase Models de País implementamos el método ToString:

@Override 🡪Esto es en el select

**public** String toString() {

**return** **this**.id.toString();

}

Ahora tratamos de hacer lo mismo conlos roles pero esto es diferente porque roles es un arreglo, entonces implementamos un valor por defecto en el método get del formulario:

user.setRoles(Arrays.*asList*(**new** Role(2,"Usuario","ROLE\_USER")));

Modificamos el formulario:

<div>

<label>Roles</label>

<div th:each="role:${listaRoles}">

<input type="checkbox" th:field="\*{roles}" th:value="${role.id}"

th:checked="${#lists.contains(usuario.roles,role)}">

<label th:text="${role.nombre}" th:for="${#ids.prev('roles')}"></label>

</div>

<div th:if="${#fields.hasErrors('roles')}" th:errors="\*{roles}"></div>

</div>

Implementamos el método iquals para comparar un lista y seleccionar el valor por defecto:

@Override

**public** **boolean** equals(Object obj) {

Role role= (Role)obj;

**return** role.id!=**null** && **this**.id.equals(role.getId());}

Como buenas parcticas consultamos si la instancia del objeto es la misma y también si el contexto es lo mismo:

@Override

**public** **boolean** equals(Object obj) {

**if** (**this**==obj) {🡪si es igual en instancia o refencia retornamos true

**return** **true**;

}

**if** (!(obj **instanceof** Role)) {🡪Consultamos si la instancia son los mismos, sino es retornamos false.

**return** **false**;

}🡪 si es instancia del role

Role role= (Role)obj;

**return** role.id!=**null** && **this**.id.equals(role.getId()); }

Redirect después del POST procesar

Modificamos el método Post que procesa los datos del formulario:

@PostMapping("/form")

**public** String ProcesarForm(@Valid Usuario user, BindingResult result, Model model) {

**if** (result.hasErrors()) {

model.addAttribute("titulo", "Resultado del Formulario");

**return** "form";}

**return** "redirect:/ver"; }

Creamos un método handler:

@GetMapping("/ver")

**public** String ver(@SessionAttribute("usuario")Usuario user, Model model,SessionStatus status) {

model.addAttribute("titulo", "Resultado del Formulario");

status.setComplete();

**return** "resultado";}

Si refrescamos la pagina resultado ahora no sdara un error porque ahora el usuario no existe debido a sessionatribute complete, es decir se elimina ese objeto, pero para manejar ese error en el parámetro modificamos @SessionAttribute(name = "usuario", required = false)Usuario user

Ahora consultamos si usuario es null entonces que redirija al formulario o a donde deseamos.

**if**(user == **null**) {

**return** "redirect:/form";}

Entonces el método quedara de la siguiente manera:

@GetMapping("/ver")

**public** String ver(@SessionAttribute(name = "usuario", required = **false**)Usuario user, Model model,SessionStatus status) {

**if**(user == **null**) {**return** "redirect:/form";}

model.addAttribute("titulo", "Resultado del Formulario");

status.setComplete();

**return** "resultado";}

Añadiendo estilos CSS al formulario con Bootstrap

Ingresar a la Web: <https://getbootstrap.com/>

Agregamos los CDN de Boostrap, por ejemplo vamos a usar la versión 4.4.1:

<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@4.4.1/dist/css/bootstrap.min.css" integrity="sha384-Vkoo8x4CGsO3+Hhxv8T/Q5PaXtkKtu6ug5TOeNV6gBiFeWPGFN9MuhOf23Q9Ifjh" crossorigin="anonymous">

Agramamos class=”clase de boostrap” para dar un formato según la clase, ejemplo:

class="form-group" o class="form-control"

Para agregar un archivo css dlocal del proyecto:

<link rel="stylesheet" th:href="/css/bootstrap.min.css">

Introduccón a los interceptores Http

Los interceptores son útiles cuando se requiere aplicar una funcionalidad especifica para ciertas peticiones HTTP y es transversal a nuestra aplicación.

¿Cómo funciona un interceptor? Los interceptores deben implementar la interfza handlerInterceptor o extender de la clase abstracta HandlerInterceptorAdapter.

Método Boolean preHandler(): Cuando este método retorna true, continua con la ejecución del controlador y si tiene más interceptores asociados continua con la ejecución en cada, cuando retorna false, se asume que el interceptor se hace cargo del request(por ejemplo, redirige a una determinada vista o ruta) y no continua ejecutando el controlador ni los demás interceptores.

Método void postHandler(): implementa algo después de que se haya invocado el método handler del controlador.

Método void afterCompletion(): implementa algo cuando finaliza el proceso, es decir, después de cargar y renderizar la vista.

Crear un package en la raíz: interceptors.

Crear una Clase TiempoTranscurridoInterceptor que implemente a HandlerInterceptor

Anotamos con @Component para que se pueda inyectar en una clase configuración.

Clase completa :

**import** org.slf4j.Logger;

**import** org.slf4j.LoggerFactory;

@Component("tiempoTranscurridoInterceptor")

**public** **class** TiempoTranscurridoInterceptor **implements** HandlerInterceptor {

**private** **static** **final** Logger ***log*** = LoggerFactory.*getLogger*(TiempoTranscurridoInterceptor.**class**);

@Override

**public** **boolean** preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)**throws** Exception {

***log***.info("TiempoTranscurridoInterceptor: preHandle() entrando...");

**long** tiempoInicio = System.*currentTimeMillis*();

request.setAttribute("tiempoInicio", tiempoInicio);

Random random = **new** Random();

Integer demora = random.nextInt(500);

Thread.*sleep*(demora);

**return** **true**;}

@Override

**public** **void** postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,

ModelAndView modelAndView) **throws** Exception {

**long** tiempoFin = System.*currentTimeMillis*();

**long** tiempoInicio = (Long) request.getAttribute("tiempoInicio");

**long** tiempoTranscurrido = tiempoFin - tiempoInicio;

**if** (modelAndView != **null** && handler **instanceof** TiempoTranscurridoInterceptor) {

modelAndView.addObject("tiempoTranscurrido", tiempoTranscurrido);}

***log***.info("TiempoTranscurridoInterceptor: postHandle() saliendo...");} }

Registrando interceptor en la configuración de spring

Crear una clase configuración llamando MvcConfig

**package** edu.ibero.demo.form.app;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.beans.factory.annotation.Qualifier;

**import** org.springframework.context.annotation.Configuration;

**import** org.springframework.web.servlet.HandlerInterceptor;

**import** org.springframework.web.servlet.config.annotation.InterceptorRegistry;

**import** org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration

**public** **class** MvcConfig **implements** WebMvcConfigurer {

@Autowired

@Qualifier("tiempoTranscurridoInterceptor")

**private** HandlerInterceptor tiempoTranscurridoInterceptor;

@Override

**public** **void** addInterceptors(InterceptorRegistry registry) {

registry.addInterceptor(tiempoTranscurridoInterceptor); }}

Mo

Mostramos el resultado de los milisegundos transcurridos:

<div class="alert alert-success">

<span th:text="'Tiempo transcurrido: ' + ${tiempoTranscurrido} + 'en milisegundos'"></span>

</div>

Como saber que recursos estamos utilizando:

***log***.info("Intercetando : " + handler);

Podemos restringir mas:

**if** (handler **instanceof** HandlerMethod && modelAndView != **null**) {

modelAndView.addObject("tiempoTranscurrido", tiempoTranscurrido);}

Mostramos en consola cual es el método que se esta usando:

**if** (handler **instanceof** HandlerMethod) {

HandlerMethod metodo = (HandlerMethod) handler;

***log***.info("Es un metodo del controlador : " + metodo.getMethod().getName());}

Implementando fragment interceptor para reutilizar código en thymeleaf

Creamos un directorio donde almacenar los componentes o plantillas, dentro de templates > componentes > plantillaComponentes.

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<body>

<section th:fragment="interceptor">

<div class="alert alert-success">

<span th:text="'Tiempo transcurrido: ' + ${tiempoTranscurrido} + ' en milisegundos'"></span>

</div>

</section>

</body>

</html>

Reutilizamos esa plantilla en las vistas donde se van a visualizar:

<section th:replace="componentes/plantillaTiempo :: interceptor"></section>

Para aplicar el interceptor solo a una método modificamos el método addInterceptors:

registry.addInterceptor(tiempoTranscurridoInterceptor).addPathPatterns("/form/\*\*");

También podemos validar si el tiempo es diferente de null que se muestre de lo contrario que no muestre.

th:if="${tiempoTranscurrido != null}"

También podemos omitir según el tipo del método es decir si es método post que no aplique el interceptor:

**if**(request.getMethod().equalsIgnoreCase("post")) {

**return** **true**; }

Solucionamos el error nullpointer del tiempotranscurrido:

**if**(!request.getMethod().equalsIgnoreCase("post")) {**return** ; }

Retornando false en un interceptor

Si el método preHandler retorna un false lo ideal es redirigir a un login o algún mensaje que muestre al usuario:

response.sendRedirect(request.getContextPath().concat("/login"));

**return** **false**;

Interceptores HTTP : Horario De acceso a clientes

1.- Creando proyecto

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamenteDependencias:

Crear un paquete controllers y dentro una calase controller HorarioControllers.

**package** com.ibero.interceptors.app.controllers;

**import** org.springframework.beans.factory.annotation.Value;

**import** org.springframework.stereotype.Controller;

**import** org.springframework.ui.Model;

**import** org.springframework.web.bind.annotation.GetMapping;

@Controller

**public** **class** HorarioController {

@Value("${config.horario.apertura}")

**private** Integer apertura;

@Value("${config.horario.cierre}")

**private** Integer cierre;

@GetMapping({"/","/index"})

**public** String index(Model model) {

model.addAttribute("titulo", "Bienvenido al horario de atención a clientes. ^-^");

**return** "index"; }

@GetMapping("/cerrado")

**public** String cerrado(Model model) {

StringBuilder mensaje = **new** StringBuilder("Cerrado, por favor visitenos desde las");

mensaje.append(", atendemos desde las ");

mensaje.append(apertura);

mensaje.append(" hrs.");

mensaje.append("Hasta las ");

mensaje.append(cierre);

mensaje.append(" hrs.");

mensaje.append("Gracias por su visita.");

model.addAttribute("titulo", "Fuera del horario de atención. -\_-");

model.addAttribute("mensaje", mensaje.toString());

**return** "cerrado"; } }

Creamos un recurso en el aplicationproperties con la apertura y el cierre:

config.horario.apertura=14

config.horario.cierre=23

Creamos un paquete Interceptors, donde creamos una clase interceptors:

**package** com.ibero.interceptors.app.interceptors;

**import** java.util.Calendar;

**import** org.springframework.beans.factory.annotation.Value;

**import** org.springframework.stereotype.Component;

**import** org.springframework.web.method.HandlerMethod;

**import** org.springframework.web.servlet.HandlerInterceptor;

**import** org.springframework.web.servlet.ModelAndView;

**import** jakarta.servlet.http.HttpServletRequest;

**import** jakarta.servlet.http.HttpServletResponse;

@Component("horariointerceptor")

**public** **class** HorarioInterceptors **implements** HandlerInterceptor {

@Value("${config.horario.apertura}")

**private** Integer apertura;

@Value("${config.horario.cierre}")

**private** Integer cierre;

@Override

**public** **boolean** preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)

**throws** Exception {

Calendar calendar = Calendar.*getInstance*();

**int** hora = calendar.get(calendar.***HOUR\_OF\_DAY***);

**if**(hora >= apertura && hora < cierre) {

StringBuilder mensaje = **new** StringBuilder("Bienvenidos al horario de atención a clientes");

mensaje.append(", atendemos desde las ");

mensaje.append(apertura);

mensaje.append(" hrs.");

mensaje.append("Hasta las ");

mensaje.append(cierre);

mensaje.append(" hrs.");

mensaje.append("Gracias por su visita.");

request.setAttribute("mensaje", mensaje.toString());

**return** **true**;}

response.sendRedirect(request.getContextPath().concat("/cerrado"));

**return** **false**;}

@Override

**public** **void** postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,

ModelAndView modelAndView) **throws** Exception {

String mensaje = (String) request.getAttribute("mensaje");

**if**(modelAndView != **null** && handler **instanceof** HandlerMethod) {

modelAndView.addObject("horario", mensaje);} } }

Creamos una clase MvcConfig donde registraremos el interceptor:

**package** com.ibero.interceptors.app;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.beans.factory.annotation.Qualifier;

**import** org.springframework.context.annotation.Configuration;

**import** org.springframework.web.servlet.HandlerInterceptor;

**import** org.springframework.web.servlet.config.annotation.InterceptorRegistry;

**import** org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration

**public** **class** MvcConfig **implements** WebMvcConfigurer{

@Autowired

@Qualifier("horariointerceptor")

**private** HandlerInterceptor horariointerceptor;

@Override

**public** **void** addInterceptors(InterceptorRegistry registry) {

registry.addInterceptor(horariointerceptor).excludePathPatterns("/cerrado");}}

HTML del index:

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head>

<meta charset="UTF-8">

<title th:text="${titulo}"></title>

</head>

<body>

<h1 th:text="${titulo}"></h1>

<div th:text="${horario}"></div>

</body>

</html>

HTML de cerrado:

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head>

<meta charset="UTF-8">

<title th:text="${titulo}"></title>

</head>

<body>

<h1 th:text="${titulo}"></h1>

<div th:text="${mensaje}"></div>

</body>

</html>

Manejo de Errores

Creando nueva aplicación y controlador

Dependencias:

Spring boot devtools.

Thymeleaf

Spring web

Crear un paquete controlador llamado controllers, luego crear una clase llamada ErrorsController.

**package** com.ibero.errors.app.controllers;

**import** org.springframework.stereotype.Controller;

**import** org.springframework.web.bind.annotation.GetMapping;

@Controller

**public** **class** ErrorController {

@GetMapping("/index")

**public** String index() {

**return** "index"; }

}

Al ejecutar la aplicación e ingresar a la ruta [http://localhost:8080/index lanzará un error 500](http://localhost:8080/index%20lanzará%20un%20error%20500).

Personalizando error 404 y 500

Para personalizar los errores, en spring boot todo tiene una configuración predefinidad pero también maneja convención por sobre configuración, es decir, podemos customizar los errores(personalizar).

Crear una carpeta llamada error y el html 500.html, según la plantilla podemos modificar el diseño de la vista.

Agregamos el thymeleaf xmlns:th=<http://www.thymeleaf.org> y utilizamos los objetos que trae thymeleaf por defecto:

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head>

<meta charset="UTF-8">

<title>Ha ocurrido un error Ooops!</title>

</head>

<body>

<h3>Ha ocurrido un error Ooops! -\_-</h3>

<div th:text="${error}"></div> 🡪 Indica el error

<div th:text="${message}"></div>🡪Indica el mensaje del error

<div th:text="${status}"></div>🡪Indica el error sea 500

<div th:text="${timestamp}"></div>🡪Indica la fecha

<div th:text="${trace}"></div>🡪Indica la traza del error

</body>

</html>

Entonces cualquier error 500 se va mostrar en la vista que estamos implementando, pero también podemos personalizar mas a detalle ese error es decir si fuera un error AritmeticException que muestre una vista de ese tipo de error y así personalizar cada error.

Ahora creamos un html llamado 404.html y cualquier error que sea 404 se mostrara en esa vista.

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head>

<meta charset="UTF-8">

<title>Lo sentimos, página no encontrada 404!</title>

</head>

<body>

<h3>Ooops! página no encontrada 404! -\_-</h3>

<div th:text="${error}"></div>🡪 Indica el error

<div th:text="${status}"></div>🡪Indica el error sea 404

<div th:text="${timestamp}"></div>🡪 Indica la fecha

<div th:text="${path}"></div>🡪Indica la ruta

</body>

</html>

Anotación @ControllerAdvice para manejo de excepciones

Creamos un nuevo controlador Llamado ErrorHandlerController anotado con el estereotypo ControllerAdvice.

**package** com.ibero.errors.app.controllers;

**import** java.util.Date;

**import** org.springframework.http.HttpStatus;

**import** org.springframework.ui.Model;

**import** org.springframework.web.bind.annotation.ControllerAdvice;

**import** org.springframework.web.bind.annotation.ExceptionHandler;

@ControllerAdvice

**public** **class** ErrorHandlerController {

@ExceptionHandler(ArithmeticException.**class**)//Tambien podemos poner varias excepciones con {}

**public** String aritmeticaError(ArithmeticException ex, Model model) {

model.addAttribute("error", "error de aritmética -\_-");

model.addAttribute("message", ex.getMessage());

model.addAttribute("status", HttpStatus.***INTERNAL\_SERVER\_ERROR***.value());

model.addAttribute("timestamp", **new** Date());

**return** "error/aritmetica";} }

Creamos el html dentro de la carpeta error, tal como lo llamamos en el método de retorno.

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head>

<meta charset="UTF-8">

<title th:text="${error}"></title>

</head>

<body>

<h3 th:text="${error}"></h3>

<div th:text="${message}"></div>

<div th:text="${status}"></div>

<div th:text="${timestamp}"></div>

<div th:text="${trace}"></div>

</body>

</html>

Personalizando página de error para la excepción NumberFormatException

Que pasa si agregamos la siguiente línea de código:

Integer valor = Integer.parseInt("10x");🡪Lanzará un error de mensaje de NumberFormatExpeción.

Para manejar ese error creamos un nuevo método en el ErrorHandlerController:

@ExceptionHandler(NumberFormatException.**class**)

**public** String numberFormatError(NumberFormatException ex, Model model) {

model.addAttribute("error", "error: Formato de numero incorrecto -\_-");

model.addAttribute("message", ex.getMessage());

model.addAttribute("status", HttpStatus.***INTERNAL\_SERVER\_ERROR***.value());

model.addAttribute("timestamp", **new** Date());

**return** "error/number-error";}

Creamos la vista del método que manejará el error:

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head>

<meta charset="UTF-8">

<title th:text="${error}"></title>

</head>

<body>

<h3 th:text="${error}"></h3>

<div th:text="${message}"></div>

<div th:text="${status}"></div>

<div th:text="${timestamp}"></div>

<div th:text="${trace}"></div>

</body>

</html>

Si tenemos la misma vista para diferentes error se crea un html con el nombre que gustamos para manejar el error que serán diferentes pero mostrados en la misma vista, solo cambiamos en el return de cada método:

**return** "error/generica";

Manejando excepción personalizada creando clases Model y Service

Creamos un paquete llamado models dentro del paquete creamos una clase usuario:

**package** com.ibero.errors.app.models;

**public** **class** Usuario {

**private** Integer id;

**private** String nombre;

**private** String apellido;

Cosntructor vacio

Constructor con atributos.

Métodos getter and setter

Creamos un paquete Service y dentro una interfaz y una clase del usuario:

**package** com.ibero.errors.app.services;

**import** java.util.List;

**import** com.ibero.errors.app.models.Usuario;

**public** **interface** UsuarioService {

**public** List<Usuario> ListarUsuario();

**public** Usuario Buscarxid(Integer id);}

Implementamos la interfaz creada en una clase llamada UsuarioServiceImpl

**package** com.ibero.errors.app.services;

**import** java.util.ArrayList;

**import** java.util.List;

**import** org.springframework.stereotype.Service;

**import** com.ibero.errors.app.models.Usuario;

@Service

**public** **class** UsuarioServiceImpl **implements** UsuarioService {

**private** List<Usuario> listarUsuario;

**public** UsuarioServiceImpl() {

**this**.listarUsuario = **new** ArrayList<>();

**this**.listarUsuario.add(**new** Usuario(1, "Andrés", "Guzmán"));

**this**.listarUsuario.add(**new** Usuario(2, "Victor", "Hurtado"));

**this**.listarUsuario.add(**new** Usuario(3, "Samuel", "Goméz"));

**this**.listarUsuario.add(**new** Usuario(4, "Rodirgo", "Rodriguez"));

**this**.listarUsuario.add(**new** Usuario(5, "Pedro", "Gareca"));

**this**.listarUsuario.add(**new** Usuario(6, "Cristian", "Giusepe"));

**this**.listarUsuario.add(**new** Usuario(7, "Monica", "Garcia"));}

@Override

**public** List<Usuario> ListarUsuario() {

**return** **this**.listarUsuario;}

@Override

**public** Usuario Buscarxid(Integer id) {

Usuario resultado = **null**;

**for** (Usuario u: **this**.listarUsuario) {

**if**(u.getId().equals(id)) {

resultado = u;

**break**;}}

**return** resultado;}}

Añadiendo método handler ver

Inyectamos el service en el controller :

@Autowired

private UsuarioService usuarioService;

Método Handler ver :

@GetMapping("/ver/{id}")

**public** String ver(@PathVariable Integer id,Model model) {

Usuario usuario = usuarioService.Buscarxid(id);

model.addAttribute("usuario", usuario);

model.addAttribute("titulo","Detalle usuario: ".concat(usuario.getNombre()));

**return** "ver";}

Creamos la vista ver:

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head>

<meta charset="UTF-8">

<title th:text="${titulo}"></title>

</head>

<body>

<h3 th:text="${titulo}"></h3>

<ul>

<li th:text="${usuario.id}"></li>

<li th:text="${usuario.nombre}"></li>

<li th:text="${usuario.apellido}"></li>

</ul>

</body>

</html>

Creando clase Exception personalizada

Crear un paquete llamada errors, dentro del paquete crear una clase llamada UsuarioNoEncontradoException.

**package** com.ibero.errors.app.errors;

**public** **class** UsuarioNoEncontradoException **extends** RuntimeException {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**public** UsuarioNoEncontradoException(String id) {

**super**("usuario con ID: ".concat(id).concat(" no existe en el sistema"));}}

Modificamos el controller, agregando las siguientes líneas de código:

**if**(usuario == **null**) {

**throw** **new** UsuarioNoEncontradoException(id.toString());}

Con eso tendríamos el error captura ahora para personalizar más tenemos que agregar un método en controlleradvice:

@ExceptionHandler(UsuarioNoEncontradoException.**class**)

**public** String usuarioNoEncontradoError(UsuarioNoEncontradoException ex, Model model) {

model.addAttribute("error", "error: usuario no encontrado -\_-");

model.addAttribute("message", ex.getMessage());

model.addAttribute("status", HttpStatus.***INTERNAL\_SERVER\_ERROR***.value());

model.addAttribute("timestamp", **new** Date());

**return** "error/generica";}

Si deseamos personalizar mas la vista del error tenemos que crear un nuevo html.

Lanzamiento de exepción usando Api Optional de Java 8

Agregamos un método en la interfaz

**public** Optional<Usuario> Buscarxidopc(Integer id);

Implementamos en el service implements:

@Override

**public** Optional<Usuario> Buscarxidopc(Integer id) {

Usuario usuario = Buscarxid(id);

**return** Optional.*ofNullable*(usuario);}

Modificamos el método handler del controller:

@GetMapping("/ver/{id}")

**public** String ver(@PathVariable Integer id,Model model) {

Usuario usuario = usuarioService.Buscarxidopc(id).orElseThrow(()->**new** UsuarioNoEncontradoException(id.toString()));

model.addAttribute("usuario", usuario);

model.addAttribute("titulo","Detalle usuario: ".concat(usuario.getNombre()));

**return** "ver";}

Spring MVC y Data JPA : Desarrollando una aplicación desde cero

Creando el proyecto

Dependencias:

* Spring boot devtools
* Spring data JPA
* H2 database
* Mysql Driver
* Thymeleaf
* Spring web

[Importante] Sobre los package javax vs jakarta

Tengan en cuenta para los siguientes videos que en Spring Boot 3 se cambia javax por jakarta, por poner un ejemplo en JPA Spring Boot 3 se importa de:

1. import jakarta.persistence.Column;
2. import jakarta.persistence.Entity;
3. import jakarta.persistence.GeneratedValue;
4. import jakarta.persistence.GenerationType;
5. import jakarta.persistence.Id;
6. import jakarta.persistence.Table;

Y antes en Spring Boot 2 se importaba de:

1. import javax.persistence.Column;
2. import javax.persistence.Entity;
3. import javax.persistence.GeneratedValue;
4. import javax.persistence.GenerationType;
5. import javax.persistence.Id;
6. import javax.persistence.Table;

Esto se debe a que Spring Boot 2 usa por debajo (tras bambalinas) en spring framework Java EE 8 y en Spring Boot 3 por debajo usa Jakarta EE 9.

Creando la clase Entidad JPA anotada con @Entity

**package** com.ibero.app.entity;

**import** java.io.Serializable;

**import** java.util.Date;

**import** jakarta.persistence.Column;

**import** jakarta.persistence.Entity;

**import** jakarta.persistence.GeneratedValue;

**import** jakarta.persistence.GenerationType;

**import** jakarta.persistence.Id;

**import** jakarta.persistence.Table;

**import** jakarta.persistence.Temporal;

**import** jakarta.persistence.TemporalType;

@Entity

@Table(name = "clientes")

**public** **class** Cliente **implements** Serializable {

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

**private** Long id;

**private** String nombre;

**private** String apellido;

**private** String email;

@Column(name = "create\_at")

@Temporal(TemporalType.***DATE***)

**private** Date createAt;

Getter and Setter.

Creando la clase de Acceso a Datos o DAO con Entity Manager - @Repository

¿Qué es un DAO? En ingeniería de software, un data Access Object – DAO es una clase que implementa y provee una interfaz común para acceder y trabajar con los datos, independiente de las tecnologías a utilizar JDBC, JPA, Hibernate, TopLink, OpenJpa,Codo,Eclipselink,iBatis o JDO.

Esta interface tiene que tener todos los métodos necesarios para recuperar y almacenar los datos (contrato de imlementación) con las operaciones básicas: listar,obtener por id,guardar, eliminar etc.

Creamos un paquete Dao, dentro del paquete la interface llamada IClienteDao

**package** com.ibero.app.dao;

**import** java.util.List;

**import** com.ibero.app.entity.Cliente;

**public** **interface** IClienteDao {

**public** List<Cliente> findAll();}

Clase entity y el entitymanager: en este caso y mientras la entidad sea manejada/asociada por el contexto de persistencia – también se las conoce como entidades atachadas o ttached entities y el estado – valores de la propiedad – de la entidad serán automáticamente sincronizadas con la DB mediante el método flush().

Luego implementamos la interface DAO con la clase ClienteDaoImpl.

**package** com.ibero.app.dao.impl;

**import** java.util.List;

**import** org.springframework.stereotype.Repository;

**import** org.springframework.transaction.annotation.Transactional;

**import** jakarta.persistence.EntityManager;

**import** jakarta.persistence.PersistenceContext;

@Repository

**public** **class** ClienteDaoImpl **implements** IClienteDao {

@PersistenceContext

**private** EntityManager em;

@SuppressWarnings("unchecked")

@Transactional(readOnly=**true**)> Marca el metodo que es solo de lectura

@Override

**public** List<Cliente> findAll() {

**return** em.createQuery("from Cliente").getResultList();}}

Creando controlador con la acción handler listar

Creamos un nuevo paquete llamamdo controller donde creamos una clase llamada ClienteController:

**package** com.ibero.app.controller;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Controller;

**import** org.springframework.ui.Model;

**import** org.springframework.web.bind.annotation.RequestMapping;

**import** org.springframework.web.bind.annotation.RequestMethod;

@Controller

**public** **class** ClienteController {

@Autowired

**private** IClienteDao clienteDao;

@RequestMapping(value = "/listar", method = RequestMethod.***GET***)

**public** String ListarCliente(Model model) {

model.addAttribute("titulo", "Listado de clientes :");

model.addAttribute("clientes", clienteDao.findAll());

**return** "index"; }}

Creando la vista listar.html

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head>

<meta charset="UTF-8"/>

<title th:text="${titulo}"></title>

</head>

<body>

<h1 th:text="${titulo}"></h1>

<table>

<thead>

<tr>

<th>Código</th>

<th>Nombre</th>

<th>Apellido</th>

<th>Email</th>

<th>Fecha</th>

</tr>

</thead>

<tbody>

<tr th:each="cliente: ${clientes}">

<td th:text="${cliente.id}"></td>

<td th:text="${cliente.nombre}"></td>

<td th:text="${cliente.apellido}"></td>

<td th:text="${cliente.email}"></td>

<td th:text="${cliente.createAt}"></td>

</tr>

</tbody>

</table>

</body>

</html>

En recursos del proyecto tenemos que crear un archivo llamado import.sql y agregamos los datos:

INSERT INTO clientes(nombre,apellido,email,create\_at) values('Andres','Guzman','andres@gmail.com','2017-08-28');

INSERT INTO clientes(nombre,apellido,email,create\_at) values('Pedro','Gomez','pedro@gmail.com','2018-09-22');

Decorando la vista listar con estilos CSS Bootstrap

Agregar el CDN de boostrap:

<!-- Bootstrap CSS -->

<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@4.3.1/dist/css/bootstrap.min.css" integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T" crossorigin="anonymous">

<!-- Optional JavaScript -->

<!-- jQuery first, then Popper.js, then Bootstrap JS -->

<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js" integrity="sha384-q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo" crossorigin="anonymous"></script>

<script src="https://cdn.jsdelivr.net/npm/popper.js@1.14.7/dist/umd/popper.min.js" integrity="sha384-UO2eT0CpHqdSJQ6hJty5KVphtPhzWj9WO1clHTMGa3JDZwrnQq4sF86dIHNDz0W1" crossorigin="anonymous"></script>

<script src="https://cdn.jsdelivr.net/npm/bootstrap@4.3.1/dist/js/bootstrap.min.js" integrity="sha384-JjSmVgyd0p3pXB1rRibZUAYoIIy6OrQ6VrjIEaFf/nJGzIxFDsf4x0xIM+B07jRM" crossorigin="anonymous"></script>

La consola H2 Database

Para utilizar la consola H2 data base primero habilitamos en el properties del proyecto:

#Configuración de consola de H2 DataBase

spring.h2.console.enabled=true

Luego ingresamos a la ruta de la consola <http://localhost:8080/h2-console>

también podemos modificar los valores por defecto:

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

spring.datasource.url=jdbc:h2:mem:Clientesdb

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=

Agregando crear en Repositorio Dao y Controlador

Agregamos un nuevo método en la interface IClienteDao:

public void guardar(Cliente cliente);

Implementamos en ClienteDaoImpl:

@Transactional

@Override

**public** **void** guardar(Cliente cliente) {

em.persist(cliente);}

Entity Manager:

Sus métodos más importantes son:

\*persist(object entity) : almacena un objeto entity en el contexto de persistencia y en la base de datos.

Merge(object entity): Actualiza las modificaciones en una entidad devolviendo un objeto entity manejado por el contexto doctrine.

\*remove(object entity): Elimina la entidad

Find(class<T> entity, object primarykey):Busca la entidad a través de su clave primaria.

Implementamos los métodos handler de carga del formulario y el método que procese los datos enviados del formulario:

@RequestMapping(value = "/form", method = RequestMethod.***GET***)

**public** String showForm(Map<String, Object> model) {

Cliente cliente = **new** Cliente();

model.put("titulo", "Registro de Clientes");

model.put("clientes",cliente );

**return** "index";}

@RequestMapping(value = "/form", method = RequestMethod.***POST***)

**public** String processForm(Cliente cliente) {

clienteDao.guardar(cliente);

**return** "redirect:/listar";}

Creando la vista crear y formulario

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head>

<meta charset="UTF-8" />

<!-- Bootstrap CSS -->

<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@4.3.1/dist/css/bootstrap.min.css"

integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T" crossorigin="anonymous">

<title th:text="${titulo}"></title>

</head>

<body>

<nav class="navbar navbar-expand-lg navbar-light bg-light">

<a class="navbar-brand" href="#">Navbar</a>

<button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarSupportedContent"

aria-controls="navbarSupportedContent" aria-expanded="false" aria-label="Toggle navigation">

<span class="navbar-toggler-icon"></span>

</button>

<div class="collapse navbar-collapse" id="navbarSupportedContent">

<ul class="navbar-nav mr-auto">

<li class="nav-item">

<a class="nav-link" th:href="@{/listar}">Home <span class="sr-only">(current)</span></a>

</li>

<li class="nav-item">

<a class="nav-link" th:href="@{/form}">Nuevo Cliente</a>

</li>

</ul>

</div>

</nav>

<h1 th:text="${titulo}"></h1>

<form th:action="@{/form}" th:object="${cliente}" method="post">

<table>

<tr>

<td>Nombre</td>

<td><input type="text" th:field="\*{nombre}" class="form-control"/></td>

</tr>

<tr>

<td>Apellido</td>

<td><input type="text" th:field="\*{apellido}" class="form-control" /></td>

</tr>

<tr>

<td>Correo</td>

<td><input type="email" th:field="\*{email}" class="form-control"/></td>

</tr>

<tr>

<td><input type="submit" value="Crear cliente" class="btn btn-success" /></td>

</tr>

</table>

<input type="hidden" th:field="\*{id}" />

</form>

</body>

</html>

Antes de ejecutar tenemos que agregar el método en el entity prepersiste:

@PrePersist

**public** **void** prePersist() {

createAt = **new** Date();}

Mejorando formulario con estilos Bootstrap CSS

<div class="container">

<form th:action="@{/form}" th:object="${cliente}" method="post">

<div class="form-group row">

<label class="col-sm-2 col-form-label">Nombre</label>

<div class="col-sm-6">

<input type="text" th:field="\*{nombre}" class="form-control" />

</div>

</div>

<div class="form-group row">

<label class="col-sm-2 col-form-label">Apellidos</label>

<div class="col-sm-6">

<input type="text" th:field="\*{apellido}" class="form-control" />

</div>

</div>

<div class="form-group row">

<label class="col-sm-2 col-form-label">Correo</label>

<div class="col-sm-6">

<input type="text" th:field="\*{email}" class="form-control" />

</div>

</div>

<div class="form-group row">

<input type="submit" value="Crear Cliente" class="btn btn-primary"/>

</div>

<input type="hidden" th:field="\*{id}" />

</form>

Agregando campo fecha en formulario

Agregar el campo fecha en el formulario:

<div class="form-group row">

<label class="col-sm-2 col-form-label">Fecha</label>

<div class="col-sm-6">

<input type="text" class="form-control" th:field="\*{createAt}" />

</div>

</div>

Si mandamos 09/02/2021 funciona correctamente pero si mandamos de la siguiente forma 09-02-2023 no funciona porque lo recibe y piensa que es un string y no puede convertir a fecha, entonces para solucionar el problema se realiza lo siguiente:

@DateTimeFormat(pattern = "yyyy-MM-dd")

Entonces va depender del formato que lo otorguemos en el campo fecha del entity con DateFormat.

**Actualización: para Spring Boot 2.3.0 o superior agregar dependencia validation**

Para validar es muy importante agregar la siguiente dependencia en el pom.xml del proyecto Spring Boot:

1. <dependency>
2. <groupId>org.springframework.boot</groupId>
3. <artifactId>spring-boot-starter-validation</artifactId>
4. </dependency>

Muy importante, es una actualización desde la versión de Spring Boot 2.3.0 o superior! agregar esta dependencia de spring-boot-starter-validation para validar!

Agregando reglas de validación usando anotaciones

Agregamos el validador en cada atributos a validar : @NotEmpty : El elemento anotado no debe ser nulo ni estar vacío.

@NotNull : El elemento anotado no debe ser nulo. Acepta cualquier tipo.

@Email : La cadena debe ser una dirección de correo electrónico bien formada. La semántica exacta de lo que constituye una dirección de correo electrónico válida se deja a los proveedores de Jakarta Bean Validation. Acepta CharSequence.

Los elementos nulos se consideran válidos.

@Size : El tamaño del elemento anotado debe estar entre los límites especificados (incluidos).

Es agregar en el controlador en los parámetros del método post valida estos atributos simplemente llamando en el parámetro @Valid.

@Valid : Marca una propiedad, parámetro de método o tipo de retorno de método para la validación en cascada.

Las restricciones definidas en el objeto y sus propiedades se validan cuando se valida la propiedad, el parámetro del método o el tipo de retorno del método.

Este comportamiento se aplica de forma recursiva.

También podemos agregar @ModelAttribute : Anotación que vincula un parámetro de método o valor de retorno de un método a un atributo de modelo con nombre, expuesto a una vista web. Compatible con clases de controlador con métodos @RequestMapping.

ADVERTENCIA: El enlace de datos puede provocar problemas de seguridad al exponer partes del gráfico de objetos a las que los clientes externos no deben acceder ni modificar. Por lo tanto, el diseño y uso del enlace de datos debe considerarse cuidadosamente en términos de seguridad. Para obtener más detalles, consulte las secciones dedicadas al enlace de datos para Spring Web MVC y Spring WebFlux en el manual de referencia.

@RequestMapping(value = "/form", method = RequestMethod.***POST***)

**public** String processForm(@Valid Cliente cliente,BindingResult result,Model model) {

**if**(result.hasErrors()) {

model.addAttribute("titulo", "Registro de Clientes");

**return** "form";}

clienteDao.guardar(cliente);

**return** "redirect:listar";}

Mostramos en el formulario el error de la siguiente forma:

<div class="form-text text-danger" th:if="${#fields.hasErrors('apellido')}" th:errors="\*{apellido}"> </div>

También podemos pintar el input frente a un error aplicando lo siguiente en el input:

<input type="text" th:field="\*{apellido}" class="form-control" th:errorclass="'form-control alert-danger'"/>

Agregando mensajes de errores personalizados

Copiamos el archivo properties y renombramos a messages.properties, debe estar UTF-8:

Agregamos la anotación que se encuentra en el atributo ejemplo:

NotEmpty=El campo no puede ser vacío desde properties 🡪Esto se aplicaría a todos.

Ahora para aplicar a una variable única agregamos en el properties de la siguiente manera:

NotEmpty.cliente.nombre= Nombre de campo requerido-> donde notempty es la anotación, cliente es la clase y nombre es el atributo de la clase.

typeMismatch.java.lang.Integer=Debe ser un número entero 🡪Para indicar si el formato del atributo no coincide.

Agregando mensajes de errores sobre el formulario

Arriba del formulario agregamos las siguientes líneas:

<div th:object="${cliente}" th:remove="tag">

<ul th:if="${#fields.hasErrors('\*')}" class="alert alert-danger">

<li th:each="err: ${#fields.errors('\*')}" th:text="${err}">

</li>

</ul>

</div>

Agregando editar

Agregamos en la interface el método a implementar:

public Cliente findOne(Long id);

Implementamos el método:

@Override

**public** Cliente findOne(Long id) {

**return** em.find(Cliente.**class**, id);}

Implementamos el método editar en el controlador:

@RequestMapping(value = "/form/{id}", method = RequestMethod.***GET***)

**public** String editarForm(@PathVariable(value = "id") Long id,Map<String, Object> model) {

Cliente cliente = **null**;

**if**(id > 0) {

cliente = clienteDao.findOne(id);

}**else** {

**return** "redirect:/listar";}

model.put("titulo", "Editar Cliente");

model.put("cliente",cliente );

**return** "form";}

Agregamos el link para el editar :

<td> <a class="btn btn-primary btn-xs" th:href="@{/form/} + ${cliente.id}">Editar</a></td>

Si tratamos de modificar el valor de ese cliente y guardamos, brindara un error, para resolver el error modificamos el metod sabe o guardar:

@Transactional

@Override

**public** **void** guardar(Cliente cliente) {

**if**(cliente.getId() != **null** && cliente.getId() > 0) {

em.merge(cliente);

}**else** {

em.persist(cliente);}}

También podemos modificar el hidden por sessionatribute:

@SessionAttributes("cliente")🡪Arriba del controlador.

Y en em método post de gauradar los datos cerramos el estado de la sesión:

SessionStatus status🡪Dentro de los parámetros del método.

Adentro del método completamos la sesión:

status.setComplete();

Agregando eliminar

Agregamos un método abstracto en la interfaz:

**public** **void** delete(Long id);

Implementamos en la Clase de la interface dao.

@Transactional

@Override

**public** **void** delete(Long id) {

Cliente cliente = findOne(id);

em.remove(cliente);}

Creamos el metod handler eliminar:

@RequestMapping(value = "/eliminar/{id}", method = RequestMethod.***GET***)

**public** String eliminarCliente(@PathVariable(value = "id") Long id) {

**if**(id > 0) {

clienteDao.delete(id);

}

**return** "redirect:/listar";}

Agregando la clase Service

@Service : Esta anotación sirve como una especialización de @Component, lo que permite que las clases de implementación se detecten automáticamente mediante el escaneo de classpath.

Implementa el patron de diseño fast cade.

Service es una fachada del Dao.

Creamos el paquete service dentro del paquete creamos la interfaz IClienteService:

**public** **interface** IClienteService {

**public** List<Cliente> findAll();

**public** **void** guardar(Cliente cliente);

**public** Cliente findOne(Long id);

**public** **void** delete(Long id);

}

Luego creamos la Clase ClienteServiceImpl implementa la interface IClienteService:

@Service

**public** **class** ClienteServiceImpl **implements** IClienteService{

@Autowired

**private** IClienteDao clienteDao;

@Override

@Transactional(readOnly = **true**)

**public** List<Cliente> findAll() {

**return** clienteDao.findAll(); }

@Transactional

@Override

**public** **void** guardar(Cliente cliente) {

clienteDao.guardar(cliente); }

@Transactional(readOnly = **true**)

@Override

**public** Cliente findOne(Long id) {

**return** clienteDao.findOne(id); }

@Transactional

@Override

**public** **void** delete(Long id) {

clienteDao.delete(id); } }

Inyectamos en el controlador la interface IClineteService, cambiamos la inyección del IClienteDao:

@Autowired

**private** IClienteService clienteService;

Cambiamos en los demás métodos handler.

Implementando el Dao con la interface CrudRepository

Eliminamos del proyecto la clase Dao Implementes

Modificamos la interface IClienteDao:

**public** **interface** IClienteDao **extends** CrudRepository<Cliente, Long> {}

Modificamos la clase Service IClienteService:

@Service

**public** **class** ClienteServiceImpl **implements** IClienteService{

@Autowired

**private** IClienteDao clienteDao;

@Override

@Transactional(readOnly = **true**)

**public** List<Cliente> findAll() {

**return** (List<Cliente>) clienteDao.findAll(); }

@Transactional

@Override

**public** **void** guardar(Cliente cliente) {

clienteDao.save(cliente); }

@Transactional(readOnly = **true**)

@Override

**public** Cliente findOne(Long id) {

**return** clienteDao.findById(id).orElse(**null**); }

@Transactional

@Override

**public** **void** delete(Long id) {

clienteDao.deleteById(id);} }

Instalando MySQL <https://www.mysql.com/> 🡪 Descargar la versión gratuita comuinity.

Después de instalar creamos un nuevo schema de base de datos en mysql.

**Recursos sobre persistencia y base de datos en Spring**

Descargar archivo PDF que contienen un resumen sobre base de datos en Spring usando JPA, conceptos que iremos abarcando a lo largo del curso.

Importante leer!!! para complementar con lo que iremos viendo en las próximas clases!

[Abrir PDF](ppt_base_de_datos.pdf)

Bonus track: Layout usando fragments en Thymeleaf

Creamos un folder llamado layout dentro de /src/main/resources/layout dentro creamos un archivo llamado layout.html:

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head th:fragment="head">

</head>

<body>

<header th:fragment="header">

</header>

<footer th:fragment="footer" class="container">

</footer>

</body>

</html>

En las plantillas HTML agregamos :

<head th:replace="layout/layout :: head"></head>

Solo cambiamos el nombre del fragment.

Deprecado: th:replace="layout/layout :: head"

Actual : th:replace="~{layout/layout :: footer}"

Bonus track: Layout usando fragments en Thymeleaf - Parte II

Vamos a modificar el listar con los card

Agregamos lo siguiente:

<div class="container py-4">

<div class="card">

<div class="card-header" th:text="${titulo}"></div>

<div class="card-body">

<table class="table container">

<thead>

<tr> <th>Código</th>

<th>Nombre</th>

<th>Apellido</th>

<th>Email</th>

<th>Fecha</th>

<th>Acciones</th> </tr>

</thead>

<tbody>

<tr th:each="cliente: ${clientes}">

<td th:text="${cliente.id}"></td>

<td th:text="${cliente.nombre}"></td>

<td th:text="${cliente.apellido}"></td>

<td th:text="${cliente.email}"></td>

<td th:text="${cliente.createAt}"></td>

<td><a class="btn btn-primary btn-xs" th:href="@{/form/} + ${cliente.id}">Editar</a><a class="btn btn-danger btn-xs" th:href="@{/eliminar/} + ${cliente.id}"onclick="return confirm('¿Estas seguro de eliminar?')">Eliminar</a></td>

</tr></tbody></table></div></div></div>

Mas cambios:

class="card bg-dark text-white"

class="card-header "

class="card-body text-white"

class="table text-white"

Creamos dos directorios en la ruta /src/main/resources/static/ y agregamos los recursos css y js en sus respectivas fólderes.

Cambiamos las rutas en el layout:

<!-- Bootstrap CSS -->

<link rel="stylesheet" th:href="@{/css/bootstrap.min.css}">

<!-- Optional JavaScript -->

<!-- jQuery first, then Popper.js, then Bootstrap JS -->

<script th:src="@{/js/jquery-3.3.1.min.js}"></script>

<script th:src="@{/js/popper.min.js}"></script>

<script th:src="@{/js/bootstrap.min.js}"></script>

Bonus track: Flash Messenger

Primero modificamos el layout agregando los div para los mensajes al cliente:

<div class="alert alert-success" th:if="${success != null}" th:text="${success}"></div>

<div class="alert alert-danger" th:if="${error != null}" th:text="${error}"></div>

<div class="alert alert-warning" th:if="${warning != null}" th:text="${warning}"></div>

<div class="alert alert-info" th:if="${info != null}" th:text="${info}"></div>

Actualización: para Spring Boot 3 o superior para PagingAndSortingRepository

Tengan en cuenta para los siguientes videos que en Spring Boot 3 la interface **PagingAndSortingRepository** ya no extiende de **CrudRepository** sino que extiende de **Repository**y no incluye los métodos del Crud.

Tenemos dos soluciones:

La primera es en vez de usar PagingAndSortingRepository extendemos de JpaRepository**:**

1. public interface IClienteDao extends JpaRepository<Cliente, Long>{
2. }

Mas <https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html>

La segunda solución es extender de ambas separado por coma, de PagingAndSortingRepository y CrudRepository:

1. public interface IClienteDao extends PagingAndSortingRepository<Cliente, Long>, CrudRepository<Cliente, Long>{
2. }

Saludos!

Bonus track: Paginador desde cero - Usando PagingAndSortingRepository - Parte I

Primero modificado la interface Dao:

IClienteDao **extends** CrudRepository<Cliente, Long>, PagingAndSortingRepository<Cliente, Long>{

Agregamos un nuevo método en IClienteService:

public Page<Cliente> findAll(Pageable pageable);

Implementamos el método en ClienteServiceImpl implements IClienteService

@Override

@Transactional(readOnly = **true**)

**public** Page<Cliente> findAll(Pageable pageable) {

**return** clienteDao.findAll(pageable);}

Modificamos el método del contraloador :

@RequestMapping(value = "/listar", method = RequestMethod.***GET***)

**public** String ListarCliente(@RequestParam(name = "page",defaultValue = "0")**int** page, Model model) {

Pageable pageRequest = PageRequest.*of*(page, 4);

Page<Cliente> clientes = clienteService.findAll(pageRequest);

model.addAttribute("titulo", "Listado de clientes :");

model.addAttribute("clientes", clientes);

**return** "index"; }

Bonus track: Paginador desde cero - La clase PageRender y controlador - Parte II

Creamos un paquete útil.paginator dentro del paquete crear una clase llamada PageRender :

**import** org.springframework.data.domain.Page;

**public** **class** PageRender<T> {

**private** String url;

**private** Page<T> page;

**private** **int** totalPaginas;

**private** **int** numElementosPorPagina;

**public** PageRender(String url, Page<T> page) {

**this**.url = url;

**this**.page = page;

numElementosPorPagina = page.getSize();

totalPaginas = page.getTotalPages();

paginaActual = page.getNumber();

**int** desde,hasta;

**if**(totalPaginas <= numElementosPorPagina) {

desde = 1;

hasta = totalPaginas;

}**else** {

**if**(paginaActual <= numElementosPorPagina/2) {

desde = 1;

hasta = numElementosPorPagina;

}**else** **if**(paginaActual >= totalPaginas - numElementosPorPagina/2) {

desde = totalPaginas - numElementosPorPagina + 1;

hasta = numElementosPorPagina;

}**else** {

desde = paginaActual - numElementosPorPagina/2;

hasta = numElementosPorPagina;}}}}

Creamo su clase PageItems:

**public** **class** PageItem {

**private** **int** numero;

**private** **boolean** actual;

**public** PageItem(**int** numero, **boolean** actual) {

**this**.numero = numero;

**this**.actual = actual;}

**public** **int** getNumero() {

**return** numero; }

**public** **boolean** getActual() {

**return** actual; }}

agregamos una colección de las paginas ítems:

**private** List<PageItem> paginas;

Agregamos en e constructor:

**this**.paginas = **new** ArrayList<PageItem>();

agregar dentro del contrsuctor :

**for**(**int** i = 0; i < hasta; i++) {

paginas.add(**new** PageItem(desde + i, paginaActual == desde + i)); }

métodos :

**public** **boolean** isFirst() {

**return** page.isFirst();}

**public** **boolean** isLast() {

**return** page.isLast();}

**public** **boolean** isHasNext(){

**return** page.hasNext();}

**public** **boolean** isHasPrevious(){

**return** page.hasPrevious();}

Getter de los atributos.

public String getUrl()

public int getTotalPaginas()

public int getPaginaActual()

public List<PageItem> getPaginas()

Bonus track: Paginador desde cero - La vista fragmento paginator-nav - Parte III

Agregamos :

PageRender<Cliente> pageRender = **new** PageRender< Cliente>("/listar", clientes);

Y enviamos:

model.addAttribute("page", pageRender);

Creamos un recurso html llamado paginator-nav.html:

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<body>

<nav th:fragment="paginator">

<ul>

<li>

<span th:if="${page.first}">Primera</span>

<a th:if="${not page.first}" th:href="@{${page.url}(page=0)}">Primera</a>

</li>

<li>

<span th:if="${not page.hasPrevious}">Anterior</span>

<a th:if="${page.hasPrevious}" th:href="@{${page.url}(page=${page.paginaActual-2})}">Anterior</a>

</li>

<li th:each="item : ${page.paginas}">

<span th:if="${item.actual}" th:text="${item.numero}"></span>

<a th:if="${not item.actual}" th:href="@{${page.url}(page=${item.numero-1})}" th:text="${item.numero}"></a>

</li>

<li>

<span th:if="${not page.hasNext}">Siguiente</span>

<a th:if="${page.hasNext}" th:href="@{${page.url}(page=${page.paginaActual})}">Siguiente</a>

</li>

<li>

<span th:if="${page.last}">Ultimo</span>

<a th:if="${not page.last}" th:href="@{${page.url}(page=${page.totalPaginas-1})}">Ultimo</a>

</li>

</ul>

</nav>

</body>

Bonus track: Paginador desde cero - Agregando estilos Bootstrap - Parte IV

Modificamos el Nav del paginator-nav.html:

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<body>

<nav th:fragment="paginator">

<ul class="pagination">

<li class="page-item" th:class="${page.first?'page-item disabled':'page-item'}">

<span class="page-link" th:if="${page.first}">Primera</span>

<a class="page-link" th:if="${not page.first}" th:href="@{${page.url}(page=0)}">Primera</a>

</li>

<li class="page-item" th:class="${not page.hasPrevious?'page-item disabled':'page-item'}">

<span class="page-link" th:if="${not page.hasPrevious}">Anterior</span>

<a class="page-link" th:if="${page.hasPrevious}" th:href="@{${page.url}(page=${page.paginaActual-2})}">Anterior</a>

</li>

<li class="page-item" th:each="item : ${page.paginas}" th:class="${item.actual? 'page-item active':'page-item'}">

<span class="page-link" th:if="${item.actual}" th:text="${item.numero}"></span>

<a class="page-link" th:if="${not item.actual}" th:href="@{${page.url}(page=${item.numero-1})}" th:text="${item.numero}"></a>

</li>

<li class="page-item" th:class="${not page.hasNext?'page-item disabled':'page-item'}">

<span class="page-link" th:if="${not page.hasNext}">Siguiente</span>

<a class="page-link" th:if="${page.hasNext}" th:href="@{${page.url}(page=${page.paginaActual})}">Siguiente</a>

</li>

<li class="page-item" th:class="${page.last?'page-item disabled':'page-item'}">

<span class="page-link" th:if="${page.last}">Ultimo</span>

<a class="page-link" th:if="${not page.last}" th:href="@{${page.url}(page=${page.totalPaginas-1})}">Ultimo</a>

</li>

</ul>

</nav>

</body>

</html>

Configuraciones y preparando la aplicación : uploads de imagenes

Agregamos en configuración del proyecto:

Agregar en el formulario la etiqueta:

enctype="multipart/form-data"🡪en el form

Configurar el aplicatión properties:

#Img

spring.servlet.multipart.max-file-size= 10MB

spring.servlet.multipart.max-request-size=10MB

En la clase entity agregamos el atributo foto:

**private** String foto;

Generamos los getter and setter.

En el inport de la base de datos agregamos la comuna fotos:

INSERT INTO clientes( nombre, apellido, email, create\_at, foto) values('Andres' ,'Guzman' ,'andres@gmail.com' ,'2017-08-28','');

Agregando campo file en formulario y procesar MultipartFile en el Controlador

Mofificamos la vista agregando el campo para subir la foto en el formulario :

<div class="form-group row">

<label for="file" class="col-sm-2 col-form-label">Foto</label>

<div class="col-sm-6">

<input type="file" name="file" class="form-control"/>

<div class="form-text text-danger" th:if="${#fields.hasErrors('foto')}"

th:errors="\*{foto}"></div></div></div>

Creamos un folder con el nombre uploads dentro de statics

Modificamos el controlador:

@RequestMapping(value = "/form", method = RequestMethod.***POST***)

**public** String processForm(@Valid Cliente cliente,BindingResult result,Model model,@RequestParam("file") MultipartFile foto,RedirectAttributes flash) {

**if**(result.hasErrors()) {

model.addAttribute("titulo", "Registro de Clientes");

**return** "form";

}

//Procesar foto

**if**(!foto.isEmpty()) {

Path directorioRecursos = Paths.*get*("src//main//resources//static/uploads");

String rootPath = directorioRecursos.toFile().getAbsolutePath();

**try** {

**byte**[] bytes = foto.getBytes();

Path rutaCompleta = Paths.*get*(rootPath+"//"+foto.getOriginalFilename());

Files.*write*(rutaCompleta, bytes);

flash.addFlashAttribute("info", "Has subido correctamente '"+foto.getOriginalFilename()+"'");

cliente.setFoto(foto.getOriginalFilename());

} **catch** (IOException e) {e.printStackTrace();}}

String mensajeFlash = (cliente.getId() != **null**)?"Cliente editado con éxito":"Cliente creado con éxito";

clienteService.guardar(cliente);

flash.addFlashAttribute("success",mensajeFlash);

**return** "redirect:listar";}

Agregando método handler ver imagen en el Controlador

Creamos un método handler ver:

@RequestMapping(value = "/ver/{id}", method = RequestMethod.***GET***)

**public** String ShowDetalleCliente(@PathVariable(value="id") Long id,Map<String,Object> model, RedirectAttributes flash) {

Cliente cliente = clienteService.findOne(id);

**if**(cliente == **null**) {

flash.addFlashAttribute("error", "El cliente mp existe en la base de datos");

**return** "redirect:/listar";

}

model.put("cliente", cliente);

model.put("titulo", "Detalle cliente : " + cliente.getNombre());

**return** "ver"; }

Creando la vista ver para visualizar la imagen

Crear un recurso html llamado ver:

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head th:replace="~{layout/layout :: head}"></head><body>

<header th:replace="~{layout/layout :: header}"></header>

<div class="container py-4">

<div class="card bg-light text-dark">

<div class="card-header font-weight-bold" th:text="${titulo}"></div>

<div class="card-body text-dark">

<img th:if="${#strings.length(cliente.foto)>0}" class="img-thumbnail rounded float-left" style="max-width: 25%;" th:src="@{'/uploads/'+${cliente.foto}}"

th:alt="${cliente.foto}"/>

<ul class="list-group">

<li class="list-group-item" th:text="${cliente.nombre}"></li>

<li class="list-group-item" th:text="${cliente.apellido}"></li>

<li class="list-group-item" th:text="${cliente.email}"></li>

<li class="list-group-item" th:text="${cliente.createAt}"></li>

</ul></div></div></div><footer th:replace="~{layout/layout :: footer}"></footer></body></html>

Modidificamos el listar en el TD id y agregamos un link para mostrar el detalle del id:

<td><a class="btn btn-primary btn-xs" th:href="@{/ver/} + ${cliente.id}" th:text="${cliente.id}"></a></td>

<td th:text="${cliente.nombre}"></td>

Agregar directorio uploads externo al proyecto (Resource Handler)

Eliminamos el folder uploads.

Modificamos la ruta del controlador:

//Procesar foto

**if**(!foto.isEmpty()) {

String rootPath = "C://Temp//uploads";//<--ruta Windonws

**try** {

**byte**[] bytes = foto.getBytes();

Path rutaCompleta = Paths.*get*(rootPath+"//"+foto.getOriginalFilename());

Files.*write*(rutaCompleta, bytes);

flash.addFlashAttribute("info", "Has subido correctamente '"+foto.getOriginalFilename()+"'");

cliente.setFoto(foto.getOriginalFilename());

} **catch** (IOException e) {

e.printStackTrace();}}

Crear una clase de configuración en la raíz del proyecto llamado MvcConfig:

**package** com.ibero.app;

@Configuration

**public** **class** MvcConfig **implements** WebMvcConfigurer{

implementamos el método:

@Override

**public** **void** addResourceHandlers(ResourceHandlerRegistry registry) {

WebMvcConfigurer.**super**.addResourceHandlers(registry);

registry.addResourceHandler("/uploads/\*\*").addResourceLocations("file:/C:/Temp/uploads/"); }}

Antes de ejecutar el proyecto primero no olvidemos de crear el directorio o carpeta C:\Temp\uploads.

Agregar directorio absoluto y externo en raíz del proyecto

Creamos un folder en la raíz del proyecto en el nombre del proyecto hacemos new y créate forlder lo llamamos al recurso uploads.

Modificamos el controlador, el método guardar del formulario:

@RequestMapping(value = "/form", method = RequestMethod.***POST***)

**public** String processForm(@Valid Cliente cliente,BindingResult result,Model model,@RequestParam("file") MultipartFile foto,RedirectAttributes flash) {

**if**(result.hasErrors()) {

model.addAttribute("titulo", "Registro de Clientes");

**return** "form";}

//Procesar foto

**if**(!foto.isEmpty()) {

String uniqueFile = UUID.*randomUUID*().toString() + "\_" + foto.getOriginalFilename();

Path rootPath = Paths.*get*("uploads").resolve(uniqueFile);

Path rootAbsolutePath = rootPath.toAbsolutePath();

**try** {

Files.*copy*(foto.getInputStream(), rootAbsolutePath);

flash.addFlashAttribute("info", "Has subido correctamente '" + uniqueFile + "'");

cliente.setFoto(uniqueFile);

} **catch** (IOException e) {

e.printStackTrace();}}

String mensajeFlash = (cliente.getId() != **null**)?"Cliente editado con éxito":"Cliente creado con éxito";

clienteService.guardar(cliente);

flash.addFlashAttribute("success",mensajeFlash);

**return** "redirect:listar";}

Path rootPath = Paths.*get*("uploads").resolve(uniqueFile);🡪 Ruta relativo al proyecto.

Path rootAbsolutePath = rootPath.toAbsolutePath();🡪Path absoluto

Modificamos la clase de configuración :

@Configuration

**public** **class** MvcConfig **implements** WebMvcConfigurer {

@Override

**public** **void** addResourceHandlers(ResourceHandlerRegistry registry) {

WebMvcConfigurer.**super**.addResourceHandlers(registry);

String resourcesPath = Paths.*get*("uploads").toAbsolutePath().toUri().toString();

registry.addResourceHandler("/uploads/\*\*").addResourceLocations(resourcesPath);}}

Cargando imagen programáticamente en la respuesta HTTP - con ResponseEntity

Eliminamos la clase de configuración MvcConfigure.

Ahora no funcionara al desear ver el detalle del cliente no cargara la foto, porque no lo encuentra.

Creamos un nuevo método para manejo de las imágenes en el controlador:

@RequestMapping(value = "/uploads/{filename:.+}", method = RequestMethod.***GET***)

**public** ResponseEntity<Resource> showPhoto(@PathVariable String filename) {//import ...core.io.Resource;

Path pathFoto = Paths.*get*("uploads").resolve(filename).toAbsolutePath();

LOG.info("pathFoto :" + pathFoto);

Resource recurso = **null**;

**try** {

recurso = **new** UrlResource(pathFoto.toUri());

**if**(!recurso.exists() || !recurso.isReadable()) {

**throw** **new** RuntimeException("Error : no se puede cargar la imagen: " + pathFoto.toString());}} **catch** (MalformedURLException e) {

e.printStackTrace();}

//headers->springframework.http.HttpHeaders;

**return** ResponseEntity.*ok*().header(HttpHeaders.***CONTENT\_DISPOSITION***, "attachment; filename=\"" + recurso.getFilename()+"\"").body(recurso);}

Eliminar archivo de imagen

Modificamos el método del controlador eliminar:

@RequestMapping(value = "/eliminar/{id}", method = RequestMethod.***GET***)

**public** String eliminarCliente(@PathVariable(value = "id") Long id,RedirectAttributes flash) {

**if**(id > 0) {

Cliente cliente = clienteService.findOne(id);

clienteService.delete(id);

flash.addFlashAttribute("success", "Cliente eliminado con éxito");

Path rootPath = Paths.*get*("uploads").resolve(cliente.getFoto()).toAbsolutePath();

File archivo = rootPath.toFile(); //->Importe de File IO

**if**(archivo.exists() && archivo.canRead()) {

**if(archivo.delete()) {**

flash.addFlashAttribute("info", "¡Foto " + cliente.getFoto() +"eliminada con exito!");}}}

**return** "redirect:/listar";}

Creamos un atributo constante para manejo de UPLOADS:

**private** **final** **static** String ***UPLOADS\_FORLDER***="uploads";

Modificamos editar :

//Para que sobreescriba la foto

**if**(cliente.getId() != **null** && cliente.getId() > 0 && cliente.getFoto()!= **null** && cliente.getFoto().length() > 0) {

Path rootPath = Paths.*get*(***UPLOADS\_FORLDER***).resolve(cliente.getFoto()).toAbsolutePath();

File archivo = rootPath.toFile(); //->Importe de File IO

**if**(archivo.exists() && archivo.canRead()) {

archivo.delete();}}

método Completo:

@RequestMapping(value = "/form", method = RequestMethod.***POST***)

**public** String processForm(@Valid Cliente cliente,BindingResult result,Model model,@RequestParam("file") MultipartFile foto,RedirectAttributes flash) {

**if**(result.hasErrors()) {model.addAttribute("titulo", "Registro de Clientes");**return** "form";}

//Procesar foto

**if**(!foto.isEmpty()) {

//Para que sobreescriba la foto

**if**(cliente.getId() != **null** && cliente.getId() > 0 && cliente.getFoto()!= **null** && cliente.getFoto().length() > 0) {

Path rootPath = Paths.*get*(***UPLOADS\_FORLDER***).resolve(cliente.getFoto()).toAbsolutePath();

File archivo = rootPath.toFile(); //->Importe de File IO

**if**(archivo.exists() && archivo.canRead()) {

archivo.delete();}}//fin

String uniqueFile = UUID.*randomUUID*().toString() + "\_" + foto.getOriginalFilename();

Path rootPath = Paths.*get*(***UPLOADS\_FORLDER***).resolve(uniqueFile);

Path rootAbsolutePath = rootPath.toAbsolutePath();

LOG.info("rootPath : " + rootPath);

LOG.info("rootAbsolutePath " + rootAbsolutePath);

**try** {

Files.*copy*(foto.getInputStream(), rootAbsolutePath);

flash.addFlashAttribute("info", "Has subido correctamente '" + uniqueFile + "'");

cliente.setFoto(uniqueFile);

} **catch** (IOException e) {

e.printStackTrace();}}

String mensajeFlash = (cliente.getId() != **null**)?"Cliente editado con éxito":"Cliente creado con éxito";

clienteService.guardar(cliente);

flash.addFlashAttribute("success",mensajeFlash);

**return** "redirect:listar";}

Cuando se edita la foto no esta eliminando la foto anterior si no se duplica para evitar que la foto nos este llegando nulla se persiste agregando en la anotación @SessionAttributes("cliente") y en el método SessionStatus status al final cerrar la sesión para eliminar datos en memoria con status.setComplete();

Con eso se corrige el error de duplicar las fotos.

La clase UploadFileService

Creamos una interface llamada IUploadFileService:

**import** org.springframework.core.io.Resource;

**import** org.springframework.web.multipart.MultipartFile;

**public** **interface** IUploadFileService {

**public** Resource load(String filename)**throws** MalformedURLException;

**public** String copy(MultipartFile file)**throws** IOException;

**public** **boolean** delete(String filename);}

Crear la clase que implementa la interface llamada IUploadFileServiceImpl :

@Service

**public** **class** IUploadFileServiceImpl **implements** IUploadFileService {

**private** **final** Logger LOG = LoggerFactory.*getLogger*(getClass());

**private** **final** **static** String ***UPLOADS\_FORLDER*** = "uploads";

@Override

**public** Resource load(String filename) **throws** MalformedURLException {

Path pathFoto = getPath(filename);

LOG.info("pathFoto :" + pathFoto);

Resource recurso = **null**;

recurso = **new** UrlResource(pathFoto.toUri());

**if** (!recurso.exists() || !recurso.isReadable()) {

**throw** **new** RuntimeException("Error : no se puede cargar la imagen: " + pathFoto.toString());}

**return** recurso;}

@Override

**public** String copy(MultipartFile file) **throws** IOException {

String uniqueFile = UUID.*randomUUID*().toString() + "\_" + file.getOriginalFilename();

Path rootPath = getPath(uniqueFile);

LOG.info("rootPath : " + rootPath);

Files.*copy*(file.getInputStream(), rootPath);

**return** uniqueFile;}

@Override

**public** **boolean** delete(String filename) {

Path rootPath = getPath(filename);

File archivo = rootPath.toFile(); //->Importe de File IO

**if**(archivo.exists() && archivo.canRead()) {

**if**(archivo.delete()) {**return** **true**;}}**return** **false**;}

**public** Path getPath(String filename) {

**return** Paths.*get*(***UPLOADS\_FORLDER***).resolve(filename).toAbsolutePath();}}

Modificamos los métodos del controlador:

Inyectamos el service:

@Autowired

**private** IUploadFileService uploadFileService;

@RequestMapping(value = "/eliminar/{id}", method = RequestMethod.***GET***)

**public** String eliminarCliente(@PathVariable(value = "id") Long id,RedirectAttributes flash) {

**if**(id > 0) {Cliente cliente = clienteService.findOne(id);

clienteService.delete(id);

flash.addFlashAttribute("success", "Cliente eliminado con éxito");

**if**(uploadFileService.delete(cliente.getFoto())) {

flash.addFlashAttribute("info", "Foto " + cliente.getFoto() + " eliminada con exito!");}}

**return** "redirect:/listar";}

@RequestMapping(value = "/form", method = RequestMethod.***POST***)

**public** String processForm(@Valid Cliente cliente,BindingResult result,Model model,@RequestParam("file") MultipartFile foto,

RedirectAttributes flash, SessionStatus status) {

**if**(result.hasErrors()) {

model.addAttribute("titulo", "Registro de Clientes");

**return** "form";}

//Procesar foto

**if**(!foto.isEmpty()) {

**if**(cliente.getId() != **null**

&& cliente.getId() > 0

&& cliente.getFoto() != **null**

&& cliente.getFoto().length() > 0) {

uploadFileService.delete(cliente.getFoto());}

String uniqueFile = **null**;

**try** {

uniqueFile = uploadFileService.copy(foto);

} **catch** (IOException e) {

e.printStackTrace();}

flash.addFlashAttribute("info", "Has subido correctamente '" + uniqueFile + "'");

cliente.setFoto(uniqueFile);}

String mensajeFlash = (cliente.getId() != **null**)?"Cliente editado con éxito":"Cliente creado con éxito";

clienteService.guardar(cliente);

status.setComplete();

flash.addFlashAttribute("success",mensajeFlash);

**return** "redirect:listar";}

@RequestMapping(value = "/uploads/{filename:.+}", method = RequestMethod.***GET***)

**public** ResponseEntity<Resource> showPhoto(@PathVariable String filename) {//import ...core.io.Resource;

Resource recurso= **null**;

**try** {

recurso = uploadFileService.load(filename);

} **catch** (MalformedURLException e) {

e.printStackTrace();}

//headers->springframework.http.HttpHeaders;

**return** ResponseEntity.*ok*().header(HttpHeaders.***CONTENT\_DISPOSITION***, "attachment; filename=\"" + recurso.getFilename()+"\"").body(recurso);}

Inicializando directorio uploads automáticamente usando CommandLineRunner

Modificamos la interface IUploadFileService agregamos 2 metodos:

**public** **void** deleteAll();

**public** **void** init()**throws** IOException;

Implementamos los métodos en la clase implementes de la interface IUploadFileServiceImpl

@Override

**public** **void** deleteAll() {

FileSystemUtils.*deleteRecursively*(Paths.*get*(***UPLOADS\_FORLDER***).toFile());}

@Override

**public** **void** init() **throws** IOException {

Files.*createDirectory*(Paths.*get*(***UPLOADS\_FORLDER***));}

Modificamos la clase del proyecto :

**public** **class** SpringJpaDataApplication **implements** CommandLineRunner

Agregamos el método:

@Override

**public** **void** run(String... args) **throws** Exception {

uploadFileService.deleteAll();

uploadFileService.init();}

Análisis y Diseño OO con UML Diagrama de Clases del Dominio

Diagrama

Descripción generada automáticamente

Asociaciones: ManyToOne Bidireccional - Clases Entity Factura y Cliente

Modificamos la entidad Clientes agregando el atributo de la clase cliente:

**private** List<Factura> facturas;

Generar el getter y setter de el atributo.

Creamos el método de agregar factura al cliente:

**public** **void** addFactura(Factura factura) {

facturas.add(factura);}

Generamos un constructor de la clase:

**public** Cliente() {

facturas = **new** ArrayList<Factura>();}

Creamos la relación con la entidad factura:

@OneToMany🡪Un cliente muchas facturas

**private** List<Factura> facturas;

@OneToMany(mappedBy = "cliente" 🡪Mapeado por indicamos el atributo de la otra clase, donde se harán la referencia.

,fetch = FetchType.***LAZY🡪 Lazy: La carga de los objetos de la relación se producen a demanda, es decir, cuando un cliente los solicita.***

,cascade = CascadeType.***ALL🡪 Esto quiere decir que insertamos, actualizamos o eliminamos una entidad, también se aplican estas operaciones a la entidad que se relaciona***)

**private** List<Factura> facturas;

Creamos la entidad Factura con sus atributos y métodos de acceso.

@Entity 🡪que representan datos que se pueden conservar en la base de datos.

@Table(name = "facturas")

**public** **class** Factura **implements** Serializable{

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

**private** Long id;

**private** String descripción;

**private** String observacion;

@Temporal(TemporalType.***DATE***)

@Column(name = "create\_at")

**private** Date createAt;

@ManyToOne(fetch = FetchType.***LAZY***)

**private** Cliente cliente;

@PrePersist🡪se utiliza para configurar una devolución de llamada para eventos pre-persist (pre-inserción) de la entidad.

**public** **void** prePersist() {

createAt = **new** Date();}

Métodos de acceso getter and setter.

Asociaciones: OneToMany Unidireccional - Clases Entity Factura y ItemFactura

Creamos la entidad ItemFacura:

@Entity

@Table(name = "facturas\_items")

**public** **class** ItemFactura **implements** Serializable{

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

**private** Long id;

**private** Integer cantidad;

**public** Long calcularImporte() {

**return** cantidad.longValue();}

Getter y setter.

Modificamos la clase Factura:

Agregamos:

@OneToMany(fetch = FetchType.***LAZY***,cascade = CascadeType.***ALL***)

@JoinColumn(name = "factura\_id")

**private** List<ItemFactura> items;

Inicializamos Items:

**public** Factura() {

**this**.items = **new** ArrayList<ItemFactura>();}

método agregar de itemsfactura:

**public** **void** addItemFactura(ItemFactura item) {

**this**.items.add(item);}

Asociaciones: ManyToOne Unidireccional - Clases Entity ItemFactura y Producto

Creamos la entidad Producto:

@Entity

@Table(name = "productos")

**public** **class** Producto **implements** Serializable{

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

**private** Long id;

**private** String nombre;

**private** Double precio;

@Temporal(TemporalType.***DATE***)

@Column(name = "create\_at")

**private** Date createAt;

@PrePersist

**public** **void** prePersist() {createAt = **new** Date();}

En itemFactura agregamos :

@ManyToOne(

fetch = FetchType.***LAZY***,

cascade = CascadeType.***ALL***)

@JoinColumn(name = "producto\_id")

**private** Producto producto;

**public** Double calcularImporte() {

**return** cantidad.doubleValue()\*producto.getPrecio();}

En factura creamos el método:

**public** Double getTotal() {

Double total = 0.0;

**int** size = items.size();

**for**(**int** i=0;i<size; i++) {

total +=items.get(i).calcularImporte();

}

**return** total;}

Creando controlador FacturaController con la acción handler crear

@Controller

@RequestMapping("/factura")

@SessionAttributes("cliente")

**public** **class** FacturaController {

@Autowired

**private** IClienteService clienteService;

@RequestMapping(value = "/form/{clienteId}", method = RequestMethod.***GET***)

**public** String crearFactura(@PathVariable(value = "clienteId") Long clienteId,

Map<String,Object> model,

RedirectAttributes flash) {

Cliente cliente = clienteService.findOne(clienteId);

**if**(cliente == **null**) {

flash.addFlashAttribute("error", "El cliente no existe en la base de datos");

**return** "redirect:/cliente/listar";}

Factura factura = **new** Factura();

factura.setCliente(cliente);

model.put("factura",factura );

model.put("titulo","Crear factura" );

**return** "factura/form";}}

Links para acceder al formulario factura

En listar Cliente creamos un link :

<td>

<a class="btn btn-success btn-xs" th:href="'/factura/form/'+ ${cliente.id}" th:text="'crear factura'"></a>

</td>

También en el HTML ver, donde se visualiza el detalle del cliente:

<h4 class="card-title">

<a class="btn btn-primary btn-xs" th:href="'/factura/form/'+ ${cliente.id}" th:text="'crear factura'"></a></h4>

En form factura agregamos un link de regreso a detalle del cliente:

<h4 class="card-title" >

<a class="btn btn-primary btn-xs" th:href="'/ver/'+ ${factura.cliente.id}" th:text="'&laquo; volver'"></a></h4>

Escribiendo código JavaScript para el Autocomplete usando jQuery UI

Descarga el jquery iu desde <https://jqueryui.com/>

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Descomprime lo descargado y solo pega los archivos en la imagen indicado en el proyecto.

Interfaz de usuario gráfica

Descripción generada automáticamente

Agregar en layout las librerías copiadas:

<link rel="stylesheet" th:href="@{/css/jquery-ui.min.css}">

<script th:src="@{/js/jquery-ui.min.js}"></script>

Creamos un input de buscar producto

<div class="form-group row">

<label for="buscar\_producto" class="col-sm-2 col-form-label" th:text="'Buscar producto'"></label>

<div class="col-sm-6">

<input text="text" name="buscar\_producto" id="buscar\_producto" class="form-control"/>

</div>

</div>

Crear un folder llamado JS y crear un html llamado autocomplete-productos.

<!DOCTYPE html>

<html>

<head xmlns:th="http://www.thymeleaf.org">

<body>

<script type="text/javascript" th:fragment="javascript">

$(document).ready(**function**(){

$("#buscar\_producto").autocomplete({

source:**function**(request,response){

$.ajax({

url:"/factura/cargar-productos/"+ request.term,

dataType: "json",

data: {

term: request.term

},

success: **function**(data){

response($.map(data,**function**(item){

return {

value: item.id,

label: item.nombre,

precio: item.precio };}));},});},

select: **function**(event,ui){

$("#buscar\_producto").val(ui.item.label);return false;}});});

</script>

</body>

</html>

Consulta JPA para buscar productos en el Autocomplete

Creó la interface:

**public** **interface** IProductoDao **extends** CrudRepository<Producto, Long>{

@Query("select p from Producto p where p.nombre like %?1%")

**public** List<Producto> findByName(String term);}

Se agrego un método en la interface IClienteService:

**public** List<Producto> findByName(String term);

En la implementación IClienteServiceimpl:

@Autowired

**private** IProductoDao productoDao;

@Override

**public** List<Producto> findByName(String term) {

**return** productoDao.findByName(term);}

key : ghp\_F8iaXvk6W3mx6E1jcE9NaJmitf73Qu0e0J78

En el controller agregamos:

@GetMapping(value="/cargar-productos/{term}", produces = {"application/json"})

**public** @ResponseBody List<Producto> cargarProductos(@PathVariable String term){ **return** clienteService.findByName(term);}

Creando la plantilla fragment para crear los items de la factura con jQuery

plantilla-items.html

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<body>

<table th:fragment="itemsFactura" class="d-none">

<tbody id="plantillaItemsFactura">

<tr id="row\_{ID}">

<td class="d-none"><input type="hidden" value="{ID}" name="item\_id[]"/></td>

<td>{NOMBRE}</td> <td>{PRECIO}</td>

<td><input type="number" value="1" name="cantidad[]" id="cantidad\_{ID}" class="form-control col-sm-4"/></td></tr></tbody></table></body></html>

Incluir en el form de factura : form.html.

<!-- Lineas de factura Items-->

<table th:replace="factura/plantilla-items :: itemsFactura"></table>

Creando las líneas de la Factura usando jQuery y la plantilla de ítems:

Modificamos el form.html:

<table id="cargarItemProducto" class="table table-sm table-striped table-hover">

<thead><tr>

<th>Nombre</th>

<th>Precio</th>

<th>Cantidad</th></tr>

</thead>

<tbody></tbody>

</table>

Modificar el JavaScript:

select: **function** (event, ui) {

//$("#buscar\_producto").val(ui.item.label);

**var** linea = $("#plantillaItemsFactura").html();

linea = linea.replace(/{ID}/g,ui.item.value);

linea = linea.replace(/{NOMBRE}/g,ui.item.label);

linea = linea.replace(/{PRECIO}/g,ui.item.precio);

$("#cargarItemProducto tbody").append(linea);

return false;}

Calculando el total de la línea o importe

Modificamos el form.html

<th>Total</th>

Modificar plantilla-item.html: AGREGAMOS LO SIGUIENTE:

<td><span id="total\_importe\_{ID}">0</span></td>

Modificar el autocomplete JS:

**var** itemsHelper = {

calcularImporte: **function**(id,precio,cantidad){

$("#total\_importe\_" + id).html(parseInt(precio) \* parseInt(cantidad));

}

}

Luego cargamos:

$("#cargarItemProducto tbody").append(linea);

itemsHelper.calcularImporte(ui.item.value,ui.item.precio,1);

En plantilla-Items modificamos la línea:

<td><input type="number" value="1" name="cantidad[]" id="cantidad\_{ID}" class="form-control col-sm-4"

onchange="itemsHelper.calcularImporte({ID},{PRECIO},this.value);"/></td>

Incrementando cantidad de un producto existente en el detalle de la factura

Modificamos el autocomplete-productos HTML:

**var** itemsHelper = {

calcularImporte: **function**(id,precio,cantidad){

$("#total\_importe\_" + id).html(parseInt(precio) \* parseInt(cantidad));

},

hasProducto: **function**(id){

**var** resultado = false;

$('input[name="item\_id[]"]').each(**function**(){

if(parseInt(id) == parseInt($(this).val())){

resultado = true;}});return resultado;},

incrementaCantidad: **function**(id,precio){

**var** cantidad = $("#cantidad\_"+id).val()?parseInt($("#cantidad\_"+id).val()):0;

$("#cantidad\_"+id).val(++cantidad);

this.calcularImporte(id,precio,cantidad);}}

En selec del mismo archivo agrego las líneas:

select: **function** (event, ui) {

//$("#buscar\_producto").val(ui.item.label);

if(itemsHelper.hasProducto(ui.item.value,ui.item.precio)){

itemsHelper.incrementaCantidad(ui.item.value,ui.item.precio);

return false;}

**var** linea = $("#plantillaItemsFactura").html();

linea = linea.replace(/{ID}/g,ui.item.value);

linea = linea.replace(/{NOMBRE}/g,ui.item.label);

linea = linea.replace(/{PRECIO}/g,ui.item.precio);

$("#cargarItemProducto tbody").append(linea);

itemsHelper.calcularImporte(ui.item.value,ui.item.precio,1);

return false;}

Importante dar formato a los JS para que realice el cambio del desarrollo.

Eliminar linea de la factura

Agregamos en el form.html:

<th>Eliminar</th>

Creamos un nuevo método en el JS:

,eliminarLineaFactura: **function**(id){

$("#row\_" + id).remove();}

En plantillas item agregamos:

<!-- Link eliminar-->

<td><a href="#" class="btn btn-danger btn-xs" onclick="itemsHelper.eliminarLineaFactura({ID});">X</a></td>

Calculando Gran Total de la Factura

Agregamos después de la table del form de facturación la siguiente línea:

<!-- Calculando el gran Total-->

<h5> importe Total : <span class="barge barge-secondary" id="gran\_total">0</span></h5>

Agrego otra función:

,

calcularGranTotal: **function**(){

**var** total = 0;

$('span[id^="total\_importe\_"]').each(**function**(){

total += parseInt($(this).html());

});

$('#gran\_total').html(total);}

Al eliminar y calcular el importe también realizo el cálculo del gran total:

//Calcular el gran importe

this.calcularGranTotal();🡪 Llamo donde deseo aplicar este calculo.

Actualizamos un poco la base de datos:

UPDATE `db\_udemy`.`productos` SET `nombre` = 'Panasonic smart TV LCD 4K UHD 43\"' WHERE (`id` = '1');

Creando Crud Repository Factura e implementando save en clase Service

Creamso un dao llamado IFacturaDao:

**public** **interface** IFacturaDao **extends** CrudRepository<Factura, Long>{}

Agregamos un metodo abstracto en interfaz service:

**public** **void** guardarFactura(Factura factura);

Implementamos el método de la interfaz en el service implements:

Inyectamos el valor:

@Autowired

**private** IFacturaDao facturaDao;

@Override

@Transactional

**public** **void** guardarFactura(Factura factura) {

facturaDao.save(factura);

Implementando método para buscar productos en el serviceservice

Agregamos un método abstracto en el cliente service:

**public** Producto findProductoById(Long id);

Implementamos el método en el service implemente:

@Override

@Transactional(readOnly = **true**)

**public** Producto findProductoById(Long id) {

**return** productoDao.findById(id).orElse(**null**);}

En item factura agregamos el getter y setter del objeto producto.

Implementando método handler guardar en FacturaController

@PostMapping("/form")

**public** String guardarFactura(Factura factura,

@RequestParam(name ="item\_id[]",required= **false**) Long[] itemId,

@RequestParam(name = "cantidad[]", required=**false**)Integer[] cantidad,

RedirectAttributes flash,

SessionStatus status){

**for**(**int** i=0;i<itemId.length;i++) {

Producto producto = clienteService.findProductoById(itemId[i]);

ItemFactura linea = **new** ItemFactura();

linea.setCantidad(cantidad[i]);

linea.setProducto(producto);

factura.addItemFactura(linea);}

clienteService.guardarFactura(factura);

status.setComplete();

flash.addFlashAttribute("success","Factura creada con éxito!");

**return** "redirect:/ver/" + factura.getCliente().getId();}

Modificamos el javascript autocomplete:

$("form").submit(**function**(){

$("#plantillaItemsFactura").remove();

return;

});

Creando método ver detalle en FacturaController

Modificamos la intyerface :

IClienteService:

**public** Factura findFacturaById(Long id);

Implementamos en la interface implemente :

ClienteServiceImpl

@Override

**public** Factura findFacturaById(Long id) {

**return** facturaDao.findById(id).orElse(**null**);}

Creamos un método handler en FacturaController:

@RequestMapping(value = "/ver/{id}", method = RequestMethod.***GET***)

**public** String ShowDetalleFactura(@PathVariable(value="id") Long id,

Map<String,Object> model,

RedirectAttributes flash) {

Factura factura = clienteService.findFacturaById(id);

**if**(factura == **null**) {

flash.addFlashAttribute("error", "¡La factura no existe en la base de datos!");

**return** "redirect:/listar";}

model.put("factura", factura);

model.put("titulo", "Factura: ".concat(factura.getDescripcion()));

**return** "factura/ver";}