BackEndI
Profesor: Christian Ariel Díaz

Módulo 3. Framework para el desarrollo ágil de aplicaciones

Sesión 21. Introducción a Spring Boot
Junio 7 de 2022

Protocolo HTTP

Request
- Method
- URL
- Header (cabecera)
- Body

Response
- Status
- Header
- Body

Framework vs Librería

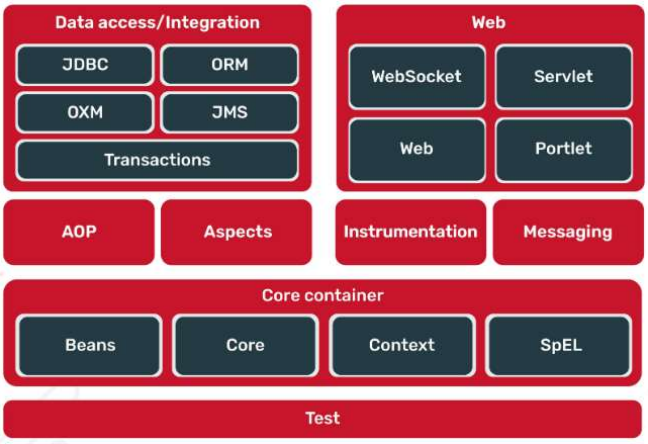
Librería
Es un conjunto de funcionalidades que resuelven necesidades específicas de un determinado proyecto.

Framework
Es un conjunto de archivos y pautas que definen la estructura sobre cómo hacer el desarrollo de un proyecto software.

Spring
- Microservicios
- Reactivo
- Nube
- Aplicaciones Web
- Serverless
- Orientado a eventos
- Procesamiento en lotes (Batch)

Spring Platform
Es un conjunto de proyectos open source desarrollados en Java con el objetivo de agilizar el desarrollo de aplicaciones. Cuenta con varias herramientas que facilitan el trabajo desde el acceso a datos, infraestructura, creación de aplicaciones web, microservicios, etc. Ej: Spring Boot, Spring Framework, Spring Data, Spring Security, Spring Batch, etc.

Spring Framework
Es un framework para el desarrollo de aplicaciones y contenedor de inversión de control. Consta de diferentes módulos que se agrupan como se muestra:



Spring Boot
Es una extensión de Spring framework que permite la creación fácil y rápida de aplicaciones web listas para producción con el concepto de just run (solo ejecutar). Requiere una mínima configuración y se complementa con muchos proyectos de Spring Platform y librerías de terceros.

Características de Spring Boot

- Web Server incorporado
Spring Boot trae el servidor Tomcat ya incorporado sin necesidad de instalación previa —también es posible usar Jetty y Undertow—.

- Puerto 8080
De manera predeterminada, el servidor incorporado escucha las solicitudes HTTP en el puerto 8080. Esto quiere decir que, cuando se arranca el servidor, se debe accedee a la URL http://localhost:8080/ para ver la aplicación. Es posible configurar otro puerto y otras propiedades, diferentes a las que se setean por defecto, colocándolas directamente en el archivo de configuración: application.properties.

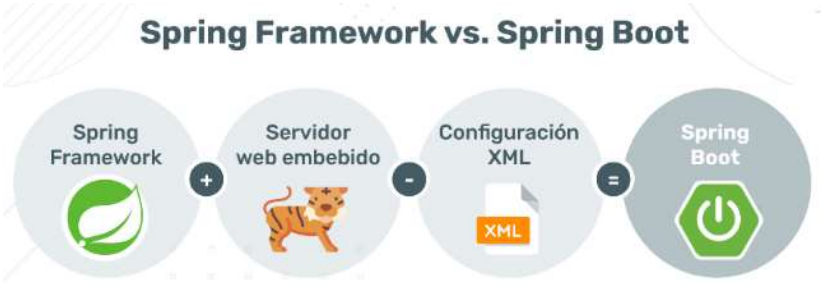
- Spring Boot Starters
Los iniciadores (starters) se usan para limitar la cantidad de configuración manual de las dependencias requeridas para un proyecto. Básicamente son dependencias de Maven que se registran en el archivo POM.xml.

Al instalar los starters, Spring Boot se encarga de hacer encajar las dependencias de tal forma que estas puedan utilizarse de forma natural en la aplicación con sus versiones correspondientes. Comienzan con spring-boot-starter-*, donde * es el tipo de aplicación que se quiere desarrollar.

Los más populares:

- spring-boot-starter-web
Se utiliza para desarrollar servicios web de RESTful con Spring MVC y Tomcat como el contenedor de aplicaciones incorporado.

- spring-boot-starter-jdbc
Se utiliza para el agrupamiento de conexiones JDBC. Se basa en la implementación del grupo de conexiones JDBC de Tomcat.



Spring Boot Initializr

Spring Initializr es una pequeña utilidad web que permite crear un esqueleto de un proyecto de Spring Boot con las opciones que se desee configurar. Permite elegir el proveedor de dependencias (Maven, Gradle, etc.), el lenguaje a utilizar (Java, Kotlin, etc.), el tipo de empaquetado (Jar o War), las dependencias/librerías requeridas, entre otras configuraciones iniciales.

Estructura del Proyecto
Se crean los siguientes archivos y directorios:

- pom.xml
- application.properties
- carpeta target (luego de construir el proyecto)
- profile (definir entorno de trabajo)

Anotaciones
Una anotación sirve para dar información al compilador o a las herramientas de desarrollo para hacer algo. Una anotación siempre va delante de una clase, método, o declaración.

Ejemplos:
@Autor(nombre = “DH", fecha = “14/03/2019")
@Deprecated
@OtraAnotacion("valor")

@Deprecated
Se usa para indicar que un método es obsoleto.

@Override
Le informa al compilador que sobreescribirá el método de la clase padre. Esto es aplicable cuando existe herencia de clases.

@SuppressWarnings
Le dice al compilador que elimine las advertencias que se pueden mostrar.

Definiendo la Clase principal
Toda aplicación en java debe contener una clase principal con un método main. Dicho método, en caso de implementar una aplicación con Spring, deberá llamar al método run de la clase SpringApplication.

```
@Configuration
@EnableAutoConfiguration
@ComponentScan
public class Application {
	public static void main(String[] args) throws Exception {
		SpringApplication.run(Application.class, args);
	}
}
```

Spring Boot permite sustituir las etiquetas @Configuration, @EnableAutoConfiguration y @ComponentScan por @SpringBootApplication.

Sesión 22. Patrón MVC
Junio 8 de 2022

Patrón de diseño: Conjunto de reglas que definen la arquitectura de un sistema.

Ventajas del MVC

- Separar la lógica de la aplicación de su representación.
- Facilitar la reutilización de código.
- Simplificar el desarrollo de cada capa.
- Mayor velocidad de desarrollo en equipo.
- Facilidad para realizar pruebas unitarias.

Vista
Conforma la interfaz gráfica de la aplicación y contiene todos los elementos que son visibles al usuario. A través de ella, el usuario interactúa enviando y solicitando información al servidor.

Controlador
Conforma la capa intermedia entre las vistas y los modelos. Sus responsabilidades son procesar los datos que recibe de los modelos y elegir la vista correspondiente en función de aquellos datos.

Modelo
Conforma y contiene la lógica de la aplicación. Sus responsabilidades son conectarse con la base de datos, realizar consultas y administrar lo que se conoce como la lógica de negocio.

Spring MVC y Spring Boot MVC

¿Cómo trabaja Spring MVC?
Spring MVC cuenta con un componente conocido como el DispatcherServlet. Es el responsable de delegar y coordinar el control entre varias interfaces en la fase de ejecución durante una petición HTTP, utilizando el patrón de diseño controlador frontal.

Esto significa que todas las solicitudes que vienen de un recurso en una aplicación serán manejadas por un solo controlador y luego enviadas al controlador apropiado para ese tipo de solicitud. El controlador frontal puede usar otros ayudantes para lograr el mecanismo de reenvío.

Handler Mapping

Después de recibir una solicitud HTTP, el DispatcherServlet consulta el HandlerMapping, que según la URL solicitada, sabrá a qué controlador debe llamar.

Controller

El controlador invocado toma la solicitud, ejecuta los métodos apropiados, y luego de llevar a cabo toda función de la lógica empresarial definida, devolverá el nombre lógico de la vista que debe mostrarse al DispatcherServlet.

ViewResolver

El DispatcherServlet tendrá la ayuda de un ViewResolver, que proporciona una asignación entre los nombres de las vistas lógicas y el nombre físico de las vistas reales.

View

Finalmente, el FrontController (el DispatcherServlet) redirige la petición hacia la vista, que muestra los resultados de la operación realizada.

Spring Boot MVC

Definiendo un Controller

La anotación @Controller indica que una clase particular cumple la función de un controlador.

El patrón Controlador sirve como intermediario entre una interfaz y el algoritmo que implementa, de tal forma que es la que recibe los datos del usuario y la que los envía a las distintas clases según el método llamado.

El trabajo del Controlador es:

- Obtener los parámetros HTTP (si los hay).
- Disparar la lógica del negocio.
- Colocar el resultado en un ámbito accesible a la vista y cederle el control a este.

@RequestMapping
Utilizamos el @RequestMapping para relacionar una URL a una clase completa o a un método del controlador particular.

Ejemplo:

```
@Controller
@RequestMapping("/hello")
public class HelloController {
	@RequestMapping(method = RequestMethod.GET)
	public String printHello(ModelMap model) {
		model.addAttribute("message", "Hello Spring MVC Framework!");
		return "hello";
	}
}
```

- Al usar @RequestMapping("/hello"), asociamos una URL a la clase HelloController, es decir, indicamos que todos los métodos de manejo en este controlador son relativos a la ruta: "/hello". http://localhost:8080/hello

Para que Spring sepa qué método del controlador debe procesar la petición HTTP, se puede especificar qué método HTTP se asocia al método Java:

@RequestMapping(method = RequestMethod.GET)

Por tanto, si se hace una llamada a la misma URL con POST produciría un error HTTP de tipo 404 porque no hay nada asociado a dicha petición.

Otra forma:

```
@Controller
public class HelloController {
	@RequestMapping(value = "/hello", method = RequestMethod.GET)
```

```
	public String printHello(Model model) {
		model.addAttribute("message","Hello Spring MVC Framework!");
		return "hello";
	}
}
```

Parámetros
Los parámetros de URL contienen una clave y un valor separados por un signo igual (=) y unidos por un signo de unión (&). El primer parámetro siempre se ubica después del signo de interrogación
en una URL.

Ejemplo:
http://example.com/listaOfertas?mes=1&user=google

Obtener parámetros HTTP

```
@Controller
public class ListaOfertasController {
	@RequestMapping(value= "/listaOfertas", method= RequestMethod.GET)
	public String procesar(Model model, @RequestParam("mes") int mes) {
		model.addAttribute("message","Hello Spring MVC Framework!");
		model.addAttribute("mes", mes);
		return "ofertas";
	}
}
```

@RequestParam asocia y convierte un parámetro HTTP a un parámetro Java.

Otras anotaciones:
@GetMapping, @PostMapping, @PutMapping, @DeleteMapping.

Ejemplo

```
@Controller
public class HelloController {
	@GetMapping ("/hello")
	public String printHello(Model model) {
		model.addAttribute("message", "Hello Spring MVC Framework!");
		return "hello";
	}
	@PostMapping("/guardar")
	public String guardarProducto(@RequestBody Employee employee) {
		return "has hecho una peticion post";
	}
}
```

Tecnología spara la capa de presentación

Spring MVC admite muchos tipos de vistas para diferentes tecnologías de presentación, a menudo denominadas plantillas de vistas.

Estas tecnologías se describen como templates porque proporcionan un lenguaje de marcado para exponer los atributos del modelo dentro de la vista durante la representación del lado del servidor.

Ejemplos:
Freemaker / Jackson / JSTL / Velocity / Thymeleaf / Views / XML / JSP / Excel / Tiles / RSS / PDF / Markup / XSTL / Groovy / Marshaling / Script

Sincrónico

REST:
https://www.freecodecamp.org/news/rest-api-tutorial-rest-client-rest-service-and-api-calls-explained-with-code-examples/

Sesión 23

@Autowired
Genera inyección de dependencias

Construir una API RestFul – Guía paso a paso
https://www.ma-no.org/es/programacion/como-construir-una-api-restful-guia-paso-a-paso

Sesión 24. Taller de coding sobre el trabajo integrador.
Junio 10 de 2022.

Sesión 25. API Rest

APIs Web
Se utilizan para crear aplicaciones que necesitan ofrecer interoperabilidad como parte de sus características. Es decir, permiten comunicarse con otras aplicaciones sin depender del sistema operativo y los lenguajes de programación.

Una API Web es una colección de estándares y protocolos que las aplicaciones y los sistemas usan para intercambiar datos a través de Internet.

SOAP
SOAP (Simple Object Access Protocol) es un protocolo estándar que define cómo dos objetos en diferentes procesos pueden comunicarse por medio de un intercambio de datos específico utilizando archivos (XML). Por lo general, opera con el protocolo HTTP, pero puede ser enviado por POP3, TCP, colas de mensajería (JMS, MQ, etc.) o FTP.

REST
REST (Representational State Transfer) es una arquitectura de software que proporciona muchas características y protocolos subyacentes que rigen el comportamiento de los clientes y los servidores.

Un servicio REST no tiene estado (es stateless), que quiere decir que no se puede llamar a un servicio REST, pasarle unos datos y esperar que los recuerde en la siguiente petición, porque no lo hará.

De ahí el nombre: el estado lo mantiene el cliente y, por lo tanto, es el cliente quien debe pasar el estado en cada llamada.

Características:
- Requier emenos recursos del servidor.
- Se centra en la escalabilidad y rendimiento a gran escala.
- Funciona basándose en los métodos HTTP.

¿Qué significa RESTFul?
Hace referencia a un servicio web que implementa la arquitectura REST:
- implementa la arquitectura Cliente-Servidor.
- Está basado en protocolo HTTP.

- No mentiene estado.
- Se accede a los recursos por medio de una URL.

JSON
JavaScript Object Notation. Es un formato de texto ligero para el intercambio de datos.

Ejemplo:
```
{
	"nombre" : "Homero",
	"apellido" : "Simpson",
	"hijos" : ["Bart", "Lisa", "Maggie"],
	"edad": 40,
	"esposa" : true
}
```

Anotaciones en el Controller

Múltiples parámetros con el método GET

@GetMapping

Ejemplo:

```
@RestController
public class HelloRestController {

	@GetMapping(path = "{name}/{lastname}/{age}")
	public String sayHello( @PathVariable String name,
				@PathVariable String lastName,
				@PathVariable Integer age) {
		return "Hola, " + name + " " + lastName + " Tu edad es: " + age;
	}
}
```

@PathVariable
Es una anotación que permite extraer información que es parte de la estructura de la URI pero que no se trata como una par nombre=valor.

```
@GetRequest("/user/{userId}")
	public User getUser(@PathVariable("userId") String userId){
	//...
}
```

@RequestParam
Es una anotación que permite recibir parámetros desde una ruta mediante el método GET, para trabajar con ellos e incluso poder emitir una respuesta que

dependa de los parámetros que sean obtenidos. Cada uno de los parámetros generalmente se ubican después de un signo ? Y están anidados por un &.
Ejemplo:
http://localhost:8080/student?name=”Horacio”&lastname=”Quiroga”

```
@GetMapping(path = "/student/")
	public Student findStudent( @RequestParam String name,
	                            @RequestParam String lastName) {
	//...
}
```

@PathVariable	@RequestParam
Se usa para recuperar valores de la propia URI.	Se usa para recuperar parámetros de consulta. Los obtiene identificándolos luego del '?' en la URL.
Forma de la petición: http://localhost:8080/employee/Juan/Lopez	**Forma de la petición:** http://localhost:8080/employee?name=Juan&lastname=Lopez

`@GetRequest("/employee/{name}/{lastname}")`
`public Employee getEmployee(@PathVariable("name") String name`
`@PathVariable("lastname") String lastName)`
`{`
`return new Employee(name, lastName);`
`}`
	`@GetMapping(path = "/employee/")`
	`public Student findEmployee(@RequestParam String name,`
	`@RequestParam String lastName) {`
	`return new Employee(name, lastName);`
	`}`

Múltiples parámetros con el método POST

@PostMapping
Se usa para mapear solicitudes HTTP POST en métodos de controlador.

Ejemplo:

```
@RestController
@RequestMapping("/")
public class HelloRestController {
	@PostMapping(path = "/employee")
	public void addEmployee(@RequestBody Employee employee) {
		//...
	}
}
```

Payload
Como parte de una solicitud POST o PUT se puede enviar una carga útil (Payload) de datos al servidor en el cuerpo (body) de la solicitud. El contenido del cuerpo puede ser cualquier objeto JSON válido.

@RequestBody

Se utiliza para vincular una solicitud HTTP (HTTP Request) con un objeto en un parámetro en un método del controlador.

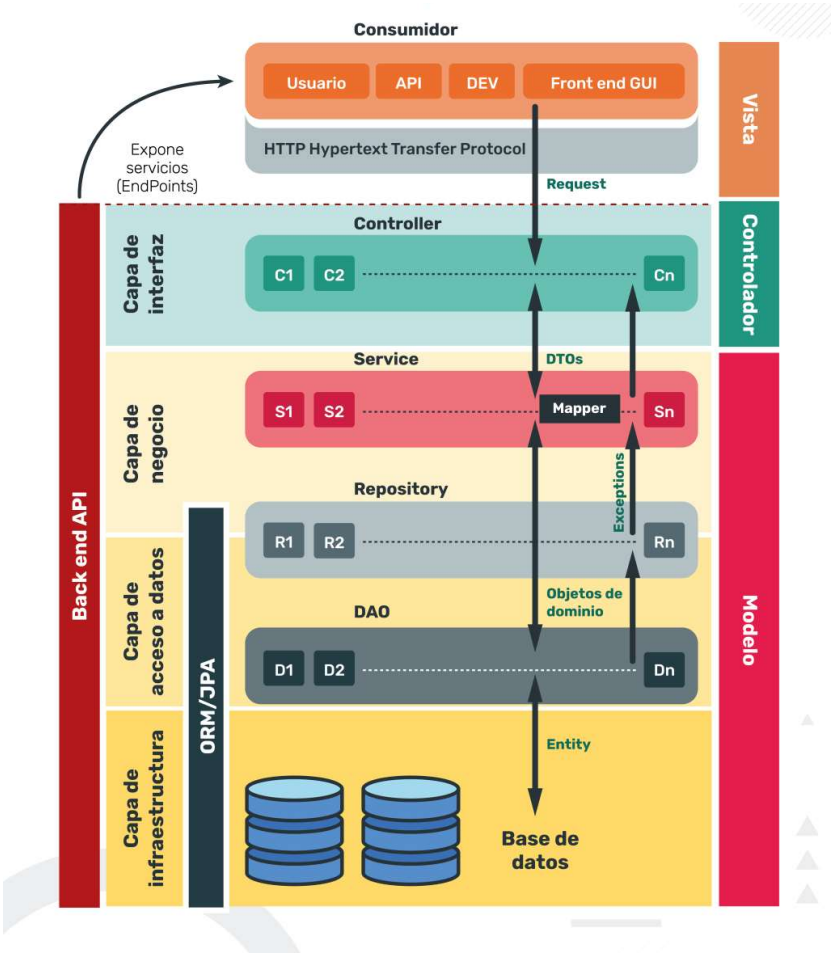
```
public class Employee{
	private Long id;
	private String name;
	private String lastName;
}

@RestController
@RequestMapping("/")
public class HelloRestController {
	@PostMapping(path = "/employee")
	public void handle(@RequestBody Employee employee) {
		//...
	}
}
```

@ResponseBody

Es una anotación utilizada para indicar el contenido de una respuesta HTTP (response) dentro de su cuerpo. Una respuesta HTTP no puede contener objetos Java, por lo que @ResponseBody se encarga de transformar los objetos a formato JSON o XML.

```
@GetMapping(path = "/orders/")
@ResponseBody
public List<Order> getOrders(){
	return orderService.getAllOrders();
}
```

Diseño de Capas Web API



Inyección de dependencias: Spring Boot se encarga de instanciar las clases.

Ejemplo:

```java
@RestController
@RequestMapping("/medicamentos")
public class MedicamentoController {

    private MedicamentoService medicamentoService = new MedicamentoService(new MedicamentoDaoH2(new ConfiguracionJDBC()));

    @PostMapping("/registrar")
    public Medicamento guardar(@RequestBody Medicamento medicamento){
        return medicamentoService.guardar(medicamento);
    }
    @GetMapping("/{id}")
    public Medicamento buscar(@PathVariable("id") Integer identificador){
        return medicamentoService.buscar(identificador);
    }
}
```

Para acceder al primer endpoint:
http://localhost:8080/medicamentos/registrar

Para el segundo endpoint:
http://localhost:8080/medicamentos/1

Ejercicio

Edad de una persona
Se necesita desarrollar un API que recibe como parámetro tres valores: Día, Mes y Año. Dichos valores corresponden a la fecha de nacimiento de una persona y deben ser valores enteros. Por ejemplo, un ingreso válido sería: 10/01/1990.

Para este caso es necesario que la fecha de nacimiento se pase mediante la URL del navegador. Por ejemplo: http://localhost:8080/10/01/1990. Como resultado, la API deberá devolver la edad de la persona.

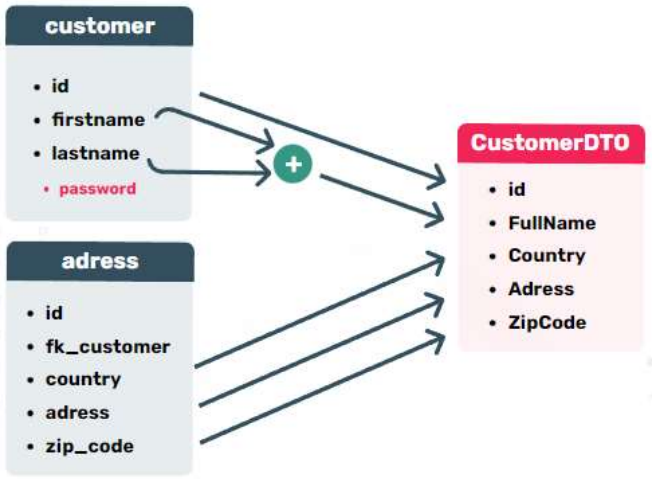
Realizar los pasos necesarios para implementar un método en el controlador que mapee correctamente el path ingresado y que devuelva la información requerida.

Sesión 26. Integradora 8.
Junio 13 de 2022

Sesión 27. API Rest II
Junio 14 de 2022

Data Transfer Object

El patrón DTO tiene como finalidad crear un objeto plano (POJO) con una serie de atributos que puedan ser enviados o recuperados del servidor en una sola invocación, de tal forma que un DTO puede contener información de múltiples fuentes o tablas y agruparla en una única clase simple.



- Un DTO se conforma de una serie de atributos que pueden o no estar compuestos por más de una fuente de datos.
- La información puede ser pasada de un lado a otro intacta o ser derivada de más de un campo.
- Permite omitir información que el usuario no requiere como la password.

El DTO es un objeto plano que debe cumplir varias reglas:
- Solo lectura. Evitar tener operaciones de negocio o métodos que realicen cálculos sobre los datos. Solo se deben tener los métodos GET y SET de los atributos del DTO.
- Serializable. Deben ser serializables tanto la clase como todos los atributos que contenga el DTO. Atributos de tipo Date o Calendar no tienen una forma estándar para serializarse y deben ser evitados.

Jackson

Es una librería que facilita el mapeo de los atributos entre una entidad y un DTO y viceversa.

```
<dependency>
	<groupId>com.fasterxml.jackson.core</groupId>
	<artifactId>jackson-databind</artifactId>
	<version>2.9.9</version>
</dependency>
```

```
// Entidad Usuario
public class Usuario {
	private Integer id;
	private String nombreUsuario;
	private String nombre;
	private String apellido;
	private String dni;
	private String contrasena;
	public Usuario(Integer id, String nombreUsuario, String nombre, String
	apellido, String dni, String contrasena) {
		this.id = id;
		this.nombreUsuario = nombreUsuario;
		this.nombre = nombre;
		this.apellido = apellido;
		this.dni = dni;
		this.contrasena = contrasena;
	}
	//Getters y Setters
}

// DTO UsuarioDTO
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
@JsonIgnoreProperties(ignoreUnknown = true)
public class UsuarioDTO {
	private String nombreUsuario;
	private String nombre;
	private String apellido;
	public UsuarioDTO(String nombreUsuario, String nombre, String apellido
	{
		this.nombreUsuario = nombreUsuario;
		this.nombre = nombre;
		this.apellido = apellido;
	}
	//Getters y Setters
}
```

La anotación @JsonIgnoreProperties(ignoreKnown = true) sirve para indicarle a Jackson que ignore el resto de atributos que tiene la entidad y que no están en el DTO. Además, Jackson requiere que haya un constructuor vacío.

```
@Test
public void entidadADTO() {
	Usuario = new Usuario(1,"user99","Juan","Perez",12345678,"abcd1234");
	ObjectMapper mapper = new ObjectMapper();
	UsuarioDTO usuarioDTO = mapper.convertValue(usuario,UsuarioDTO.class);
	Assert.assertNotNull(usuarioDTO);
}
```

El primer parámetro de convertValue es la entidad y el segundo es el tipo de dato al que se tienen que convertir los datos.

ResponseEntity

ResponseEntity hereda de la clase HttpEntity y agrega un HttpStatus código de estado. Normalmente es usada en los sevicios Rest dentro de los controladores. ResponseEntity maneja toda la respuesta HTTP incluyendo el cuerpo, la cabecera y código de estado.

Ejemplo:

```
@GetMapping(‘’/hola’’)
ResponseEntity<String> holaMundo() {
        return new ResponseEntity <>(‘’Hola Mundo desde una respuesta
        HTTP!’’, HttpStatus.OK) ;
}
```

El tipo de parámetro en genérico, lo cual permite en este caso retornar un String; además se retorna un código de estado.

Ejemplo:

```
@GetMapping(‘’/verificar/{correo}’’)
ResponseEntity<String> verificarCorreo(@PathVariable String correo) {
        if (!EmailValidator.getInstance().isValid(correo) ) {
                return new ResponseEntity <>(‘’Formato debe ser:
                ejemplo@correo.dominio’’,
                HttpStatus.BAD_REQUEST) ;
        }
        return new ResponseEntity<> ( ‘’Su correo es: ‘’+ correo, HttpStatus.OK) ;
}
```

Ejemplo

Configurar el header:

```
@GetMapping(‘’/cabecera/{cliente}’’)
ResponseEntity<String> cabeceraPersonalizada(@PathVariable String cliente)
{
        HttpHeaders cabecera = new HttpHeaders ();
        cabecera.add( ‘’Estado Cliente’’, ‘’Cliente’’+ cliente + ‘’: habilitado’’ );
        return new ResponseEntity <>(‘’Bienvenido ’’ + correo, cabecera,
        HttpStatus.OK);
}
```

ResponseEntity provee dos clases anidadas de tipo interface: BodyBuilder y HeadersBuilder y un método estático para acceder a estas interfaces.

Ejemplo

```
@GetMapping(‘’/HolaMundo’’)
ResponseEntity<String> HolaMundo2() {
        return ResponseEntity.ok(‘’HolaMundo !’’);
}
```

Ejemplo

```
@GetMapping(‘’/verificarCorreo/{correo}’’)
ResponseEntity<String> verificarCorreo2(@PathVariable String correo) {
	if (!EmailValidator.getInstance().isValid(correo) ) {
		return ResponseEntity.badRequest().body(‘’Error ! Formato
		correcto:
		ejemplo@correo.dominio’’);
	}
	return ResponseEntity.status(200).body(‘’Correo: ‘’+ correo);
}
```

Ejemplo

```
@GetMapping(‘’/cabeceraCustomizada/{cliente}’’)
ResponseEntity<String> cabeceraPersonalizada2(@PathVariable String cliente)
{
	return ResponseEntity.ok()
	.header(‘’Estado Cliente’’, ‘’Cliente’’ + cliente ‘’: habilitado’’)
	.body(Bienvenido cliente: ‘’ + cliente);
}
```

Alternativas

@ResponseBody

En las aplicaciones clásicas de Spring MVC, los endpoints generalmente devuelven páginas HTML renderizadas. A veces solo necesitamos devolver los datos reales. En tales casos podemos marcar el método del controlador con @ResponseBody, y Spring trata el valor de resultado del método como el propio cuerpo de respuesta HTTP.

@ResponseStatus

Cuando un endpoint regresa con éxito, Spring proporciona una respuesta HTTP 200 (OK). Si el endpoint arroja una excepción, Spring busca un controlador de excepciones que indique qué estado HTTP usar. Podemos marcar estos métodos con @ResponseStatus y, por lo tanto, Spring regresa con un estado HTTP personalizado.

Manipular la respuesta directamente

Spring también nos permite acceder directamente al objeto javax.servlet.http.HttpServletResponse; solo tenemos que declararlo como argumento de método:

```
@GetMapping("/manual")
void manual (HttpServerletResponse response) throws IOException {
	response.setHeader("Custom - Header", "foo");
	response.setStatus(200);
	response.getWriter().println("Hello Word!");
}
```

Ejercicio
Dada la entidad Película compuesta por:

- Título (String).
- Categoría (String).
- Premios (Integer).
- Id (Integer).

Desarrollar una API que permita:

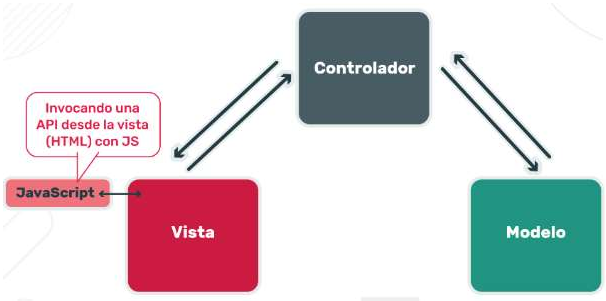
- Buscar todas las películas guardadas. Queremos que se vea un listado solamente con el título y la categoría: para este punto necesitamos utilizar un DTO. PATH: /películas Método : GET
- Eliminar una película por id. En caso de que la película exista y se haya podido eliminar correctamente, retornar con un código de estatus 200 utilizando ResponseEntity, en caso de no encontrar la película enviar status 404. PATH: /{id} Método : DELETE

Para este ejercicio no hace falta usar base de datos, se pueden hacer las operaciones sobre una lista.

Sesión 28. Consumir APIs
Junio 15 de 2022

Consumir API desde la vista.

Esta arquitectura se utiliza para poder traer o modificar datos del front end sin tener que refrescar toda la pantalla (AJAX).



Fetch()

Ejemplo Método GET:

Recibe como primer parámetro la URL del endpoint al cual se hace el llamado asincrónico. Al no saber cuándo se completa la petición, el servidor devuelve una promesa.

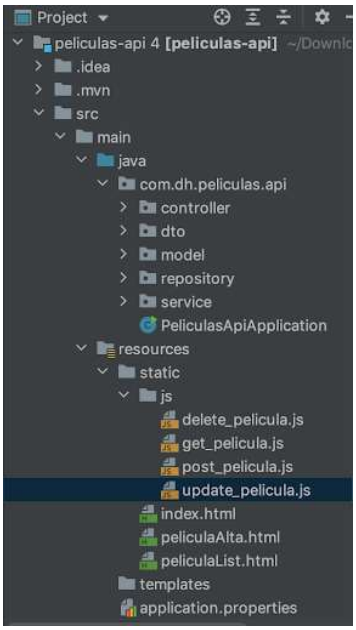
```
fetch("https://restcountries.eu/rest/v2/")
.then(function(response){
	return response.json();
})
.then(function(data){
	console.log(data)
})
.catch(function(error){
	console.error(error)
})
```

Ejemplo Método POST:

```
fetch(url, {
	method: “POST”,
	body: JSON.stringify(data),
	headers: {
		“Content-Type”: “application/json”
	}
})
.then(function(res) {
	return res.json();
})
.then(function(data) {
	console.log(data);
})
```

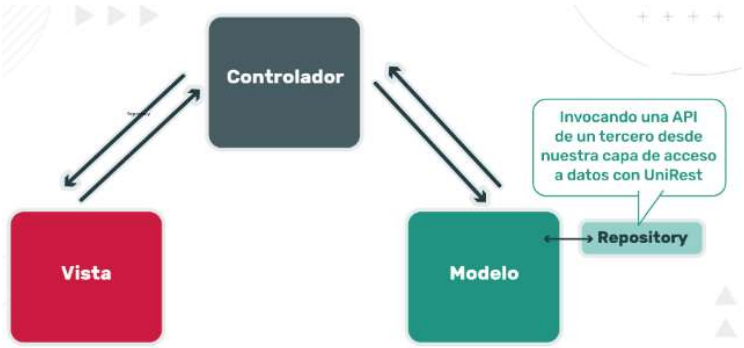
```
.catch(function(err) {
        console.log("Error! " + err);
})
```

Ejemplo



Ejercicio
Continuando con nuestro proyecto, realizar la vista que se comunica con la API de odontólogos. Desarrollar un formulario web en el que podamos dar de alta nuevos odontólogos en el sistema. Nuestro chapter lead nos solicita para este caso utilizar JavaScript y enviar las peticiones al rest controller mediante AJAX con Fetch.

Consumir APIs desde el BackEnd



Esta arquitectura se utiliza cuando tengamos que invocar una API de un tercero, por ejemplo, una API de AFIP, Facebook, Twitter, etcétera. En estos

casos es recomendable consumir y trabajar con estas APIs desde la capa de acceso a datos.

Unirest
Unirest es una librería que nos permite conectarnos como clientes a servicios REST, no solo está disponible para Java, sino que también se utiliza en muchos lenguajes, como Node.js, .Net, Python, Ruby, entre otros.

Implementación

Agrgar la dependencia en el POM:

```
<dependency>
	<groupId>com.mashape.unirest</groupId>
	<artifactId>unirest-java</artifactId>
	<version>1.4.9</version>
</dependency>
```

Petición GET simple:

```
HttpResponse<JsonNode> jsonResponse =
Unirest.get("http://www.mocky.io/v2/5a9ce37b3100004f00ab5154")
.asJson();
```

Petición GET con parámetros:

```
HttpResponse<JsonNode> jsonResponse =
Unirest.get("http://www.mocky.io/v2/5a9ce37b3100004f00ab5154")
.queryString("apiKey", "123")

HttpResponse<JsonNode> jsonResponse =
Unirest.get("http://www.mocky.io/v2/5a9ce37b3100004f00ab5154/
(userId)")
.routeParam("userId", "123")
```

Petición PUT

```
Map<String, Object> fields = new HashMap<>();
fields.put("name", "Sam Baeldung");
fields.put("id", "PSP123");
HttpResponse<JsonNode> jsonResponse =
Unirest.put("http://www.mocky.io/v2/5a9ce7853100002a00ab515e")
.fields(fields)
.asJson();
```

Petición POST

```
Map<String, Object> fields = new HashMap<>();
fields.put("name", "Sam Baeldung");
fields.put("id", "PSP123");
HttpResponse<JsonNode> jsonResponse =
Unirest.post("http://www.mocky.io/v2/5a9ce7853100002a00ab515e")
.fields(fields)
.asJson();
```

Petición DELETE

Revisar.

```
Map<String, Object> fields = new HashMap<>();
fields.put("name", "Sam Baeldung");
fields.put("id", "PSP123");
HttpResponse<JsonNode> jsonResponse =
Unirest.delete("http://www.mocky.io/v2/5a9ce7853100002a00ab515e")
.fields(fields)
.asJson();
```

Ejemplo Live Coding

Para convertir de objeto a Json y viceversa:

```
public  class Jsons {

    public static String asJsonString(Object object){
        try{
            ObjectMapper objectMapper = getObjectMapper();
            objectMapper.setSerializationInclusion(JsonInclude.Include.NON_NULL);
            objectMapper.configure(SerializationFeature.FAIL_ON_EMPTY_BEANS,  state: false);
            return objectMapper.writeValueAsString(object);
        }catch (Exception e){
            throw new RuntimeException(e);
        }

    }

    public static  <T> T objectFromString(Class<T> aClass, String value) throws JsonProcessingException {
        return getObjectMapper().readValue(value, aClass);
    }
    public static ObjectMapper getObjectMapper() {
        return new ObjectMapper().registerModule(new ParameterNamesModule())
                .configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES,  state: false)
                .registerModule(new Jdk8Module())
                .registerModule(new JavaTimeModule());
    }
}
```

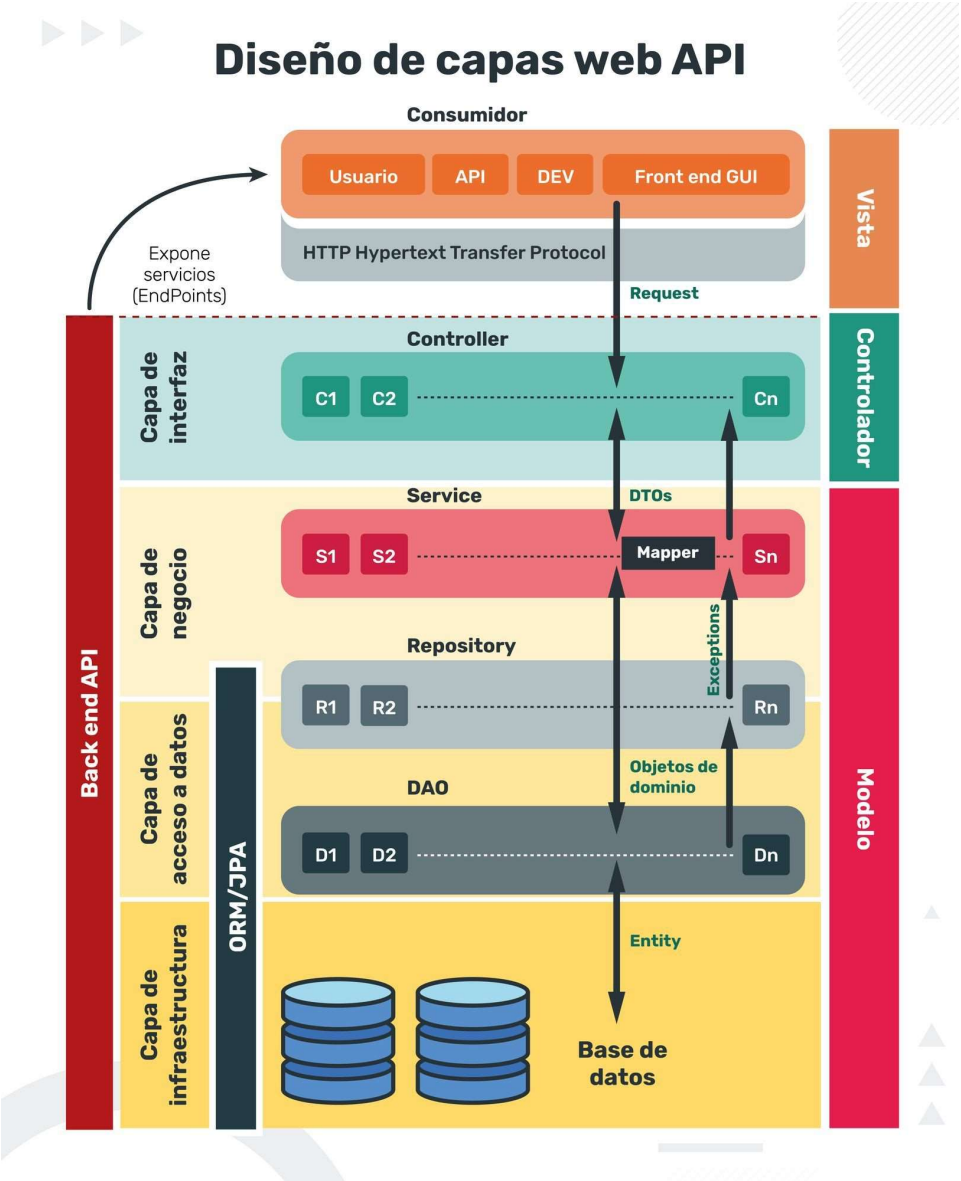
.

Sesión 29. Integradora 9
Junio 16 de 2022

Sesión 30. Inyección de dependencias
Junio 17 de 2022

Estructura de los Proyectos en Aplicaciones Empresariales

Arqueitectura Multicapa



Controller

Encargada de atender una solicitud desde el momento en que es interceptada hasta la generación de la respuesta y su transmisión. Lo que hace un controlador es llamar a una o más funciones de la capa de servicio. También gestiona la deserialización de una solicitud y la serialización de la respuesta, a través de la capa DTO. Evitar pasar un DTO como parámetros a la capa de servicio (o recibirlos de ella) permite que el sistema esté altamente desacoplado y mantiene la capa de servicio completamente independiente de

la representación de datos. Cada clase de controlador debe estar marcada con la anotación @RestController.

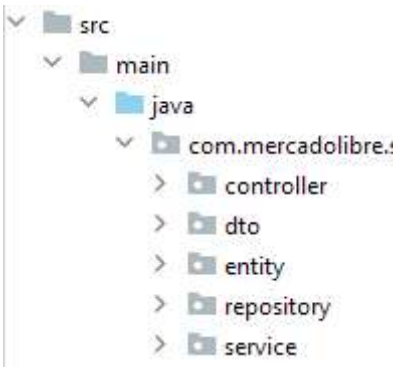
Service
La incorporación de una capa entre las clases dentro del paquete de repositorio y los controladores incrementa el nivel de desacoplamiento y hace que cada función del controlador sea más fácil de administrar y leer. La capa de servicio también se puede utilizar para definir las políticas de autorización que puede requerir cada operación. En conclusión, la capa de servicio es el lugar donde ubicar cualquier operación de lógica de negocio que uno o más controladores necesiten. Cada clase de servicio debe marcarse con la anotación @Service.

DTO (Data Transfer Object)
Los objetos de transferencia de datos se utilizan para desacoplar la representación de datos (la vista) de los objetos del modelo.

Repository
También podemos encontrarla como DAO (Data Access Object). Es la capa de persistencia de datos. Facilita el trabajo con diferentes tecnologías de acceso a datos, como JDBC, Hibernate o JPA, de manera consistente. Cambiar entre las tecnologías de persistencia antes mencionadas se vuelve fácil gracias a esta capa. Cada clase de repositorio debe estar marcada con la anotación @Repository.

Entity (o Model)
Una entidad representa una tabla almacenada en una base de datos. Cada instancia de una entidad representa una fila en la tabla. Cada clase de entidad debe estar marcada con la anotación @Entity.

Estructura del Proyecto
Cada capa debe incluirse en un paquete específico. Cada uno de estos tiene que tener el mismo nombre que la capa. De esta forma, encontrar una clase se vuelve muy fácil e intuitivo.



Inyección de Dependencias

¿Qué es un POJO?

POJO o Plain Old Java Object es un concepto que se comenzó a utilizar con los frameworks no intrusivos, como Spring y Hibernate. Es una instancia de una clase que no extiende ni implementa nada en especial.

- No debe extender clases preespecificadas.
- No debe implementar interfaces preespecificadas.
- No debe tener anotaciones preespecificadas.
- No hay restricción sobre el acceso-modificador de los atributos.
- No es necesario incluir ningún constructor en él.

Ejemplo

```
public class Alumno {
	private String name;
	private String lastName;

	public int yearsOld;

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public String getLastName() {
		return lastName;
	}

	public void setLastName(String lastName) {
		this.lastName = lastName;
	}
}
```

@Component
Es una anotación de nivel de clase que indica que la clase con esta anotación es un componente o Bean de Spring. Spring Framework detecta automáticamente las clases anotadas con @Component.

Ejemplo:

```
@Component
public class SoyUnComponente {
	// …
}
```

Esta annotation posee otras especificaciones más concretas: @Repository, @Service y @Controller.

@Repository
Si la clase va a contener todo el código necesario para la persistencia de datos, se recomienda usar @Repository en vez de @Component.

@Service
Sirve para indicar que una clase pertenece a la capa de negocio, es decir, donde se codifica la lógica de negocios de la aplicación y se oculta la complejidad de la misma al resto de las capas. @Service es una especialización de @Component, añade un valor semántico que indica la utilidad de la clase anotada como @Service para la capa de negocio.

@Controller
Para la capa de presentación se utiliza @Controller en vez de @Component.

Inyección de dependencias

Es una técnica mediante la cual un objeto proporciona las dependencias de otro objeto.

¿Qué es una dependencia?
Si la clase A usa una funcionalidad de la clase B, se dice que la clase A debende de la clase B.
En Java, para utilizar un método de otra clase, se requiere primero instanciar dicha clase. La inyección de dependencias permite que la clase A usea la clase B sin necesidad de instanciarla.

¿Cómo se implementa?
- Inyección por constructor.
Dependencias de una clase provistas por el constructor.

- Inyección por Setter
Se utiliza un método Set para proveer la dependencia.

- Inyección mediante interfaz
Se crea una interfaz con un método para inyectar las dependencias.

Ejemplo

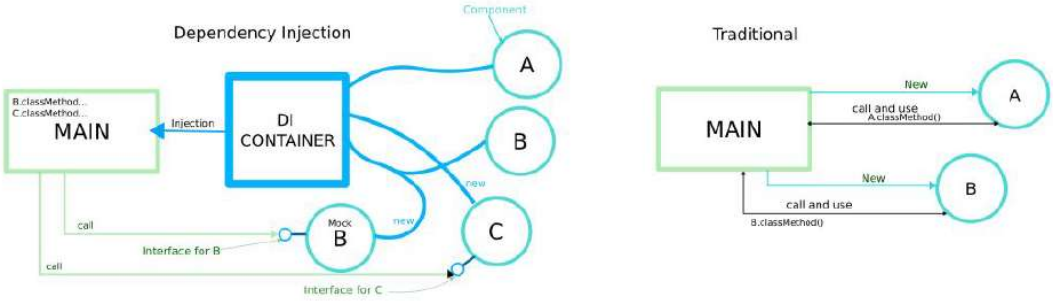
```
public class CocinaController {

	private CocineroService unCocinero = new CocineroService();

	public String prepararPlato(String ingredientes,String menu) {
		unCocinero.setIngredientes(ingredientes);
		unCocinero.setMenu(menu);
		String platoListo = unCocinero.getPlatoListo();
```

```
            return platoListo;
        }
}
```

IoC (Inversion of Control)

Contenedor de Inversión de Control de Inversión

Spring se basa en el principio de Inversión de Control (IoC) o Patrón Hollywood («No nos llames, nosotros te llamaremos») consiste en que el contenedor controla la creación de los objetos, define su ciclo de vida y resuelve las dependencias entre ellos (inyección de dependencias).

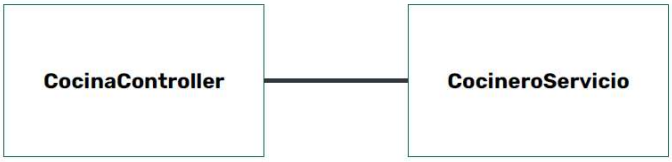


El contenedor de Inversión de control es responsable de crear instancias, configurar y ensamblar los objetos (beans). El contenedor obtiene sus instrucciones leyendo los metadatos de configuración, los cuales se representan a través de las anotaciones.

Inversión de control se refiere a todo aquel diseño de software cuyo propósito obedece a la necesidad de querer controlar el flujo de ejecución de este, de forma automática y transparente, es decir, ceder el control de ese flujo a un "agente externo", normalmente un framework.

Ventajas:
- Permite ampliar/modificar la funcionalidad del sistema sin necesidad de modificar las clases.
- Proporciona modularidad (aparte de la que proporciona nativamente el lenguaje).
- Evita la dependencia entre clases.

Rol de las Interfaces en la Inyección de Dependencias
La inyección de dependencias puede realizarse referenciando directamente las clases de dichas dependencias:



Sin embargo, esta solución crea un fuerte acoplamiento entre las clases. Para evitarlo, se puede agregar una interfaz que permite tener diferentes implementaciones sin necesidad de modificar la clase Controller:

Métodos para Inyectar dependencias:

1. Por constructor
Es el más recomendado, pues otorga la visibilidad necesaria de las inyecciones que tiene la clase y es más fácil de hacer debug.

```
@Controller
public class CocinaController {
	private CocineroService unCocineroService;
	/**
	* Constructor de la clase CocinaController
	*/
	@Autowired // Anotacion opcional desde version 4.3
	public CocinaController(CocineroService cocineroService) {
		this.unCocineroService = cocineroService;
	}
	public String prepararPlato(String ingredientes,String menu) {
		this.unCocineroService.setIngredientes(ingredientes);
		this.unCocineroService.setMenu(menu);
		String platoListo = this.unCocineroService.getPlatoListo();
		return platoListo;
	}
}
```

2. Por método Setter

Ejemplo:

```
@Controller
public class CocinaController {
	private CocineroService unCocineroService;
	@Autowired
```

```
	/*Setter*/
	public void setCocineroService (CocineroService cocineroService) {
		this.unCocineroService = cocineroService;
	}
	public String prepararPlato(String ingredientes,String menu) {
		this.unCocineroService.setIngredientes(ingredientes);
		this.unCocineroService.setMenu(menu);
		String platoListo = this.unCocineroService.getPlatoListo();
		return platoListo;
	}
}
```

3. Directo en la propiedad

```
@Controller
public class CocinaController {

	@Autowired
	private CocineroService uncocineroService;

	public String prepararPlato (String ingredientes, String menu) {
		unCocineroService.setIngredientes(ingredientes);
		unCocineroService.setMenu(menu);
		String platoListo = unCocineroService.getPlatoListo();
		return platoListo;
	}
}
```

Contenedor de inversión de Control de Spring

Un tema clave en Spring es el contenedor de inversión de control, que es compatible con la interfaz ApplicationContext. Spring crea este "espacio" en la aplicación donde se pueden colocar algunas instancias de objetos como grupos de conexión de bases de datos, clientes HTTP, etc. Estos objetos, llamados beans, se pueden usar más tarde en otras partes de la aplicación, comúnmente a través de una interfaz para abstraer su código de implementaciones específicas.

Un bean es un objeto que Spring administra en tiempo de ejecución con el contenedor de inversión de control. Estos se crean y agregan a un "repositorio de objetos" para que puedan ser obtenidos más tarde.

El mecanismo para hacer referencia a uno de estos beans desde el ApplicationContext en otras clases es la inyección de dependencia, y en Spring esto es posible a través de la configuración XML o anotaciones de código.

Bean

En Spring, los objetos que forman la columna vertebral de la aplicación y que son administrados por el contenedor de Inversión de control se denominan beans.

Un bean es un objeto que es instanciado, ensamblado y administrado por un contenedor de inversión de control. Dicho de otra forma: un bean es simplemente uno de los muchos objetos de la aplicación.

Contenedor de inversión de control Spring

La interfaz org.springframework.context.ApplicationContext representa el contenedor de inversión de control y es responsable de crear instancias, configurar y ensamblar los beans.

El contenedor obtiene sus instrucciones sobre qué objetos instanciar, configurar y ensamblar leyendo los metadatos de configuración. Los metadatos de configuración se representan en XML o a través de las anotaciones.

Anotaciones para inyección de dependencias

@Autowired
Sirve para inyectar un componente, configuración, servicios y beans, usando la autodetección de Spring.

```
@Controller
public class CocinaController {

    @Autowired
    private CocineroService unCocineroService;

   public String prepararPlato(String ingredientes,String menu) {
        unCocineroService.setIngredientes(ingredientes);
        unCocineroService.setMenu(menu);
        String platoListo = unCocineroService.getPlatoListo();
        return platoListo;
    }
}
```

@Qualifier
Sirve para especificar a Spring cuál de las implementaciones de una interfaz debe usar:

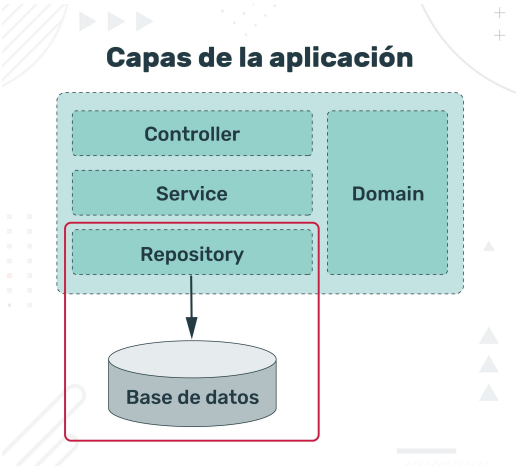
```
public class CocinaController {

        @Autowired
        @Qualifier("ChefServiceImpl")
        private CocineroService chefService;
```

```
        @Autowired
        @Qualifier("PlancheroServiceImpl")
        private CocineroService plancheroService;
        // …
}

@Service("ChefServiceImpl")
public class ChefServiceImpl implements CocineroService{
        // …
}

@Service("PlancheroServiceImpl")
public class PlancheroServiceImpl implements CocineroService{
        // …
}
```

Para incluir automáticamente getters y setters

```
import lombok.Getter;
import lombok.Setter;

@Getter @Setter
public class Book {
        // …
}
```

Sesión 31. ORM
Junio 21 de 2022



Bases de datos relacionales

Es un tipo de base de datos que cumple con el modelo relacional. El lenguaje más común para construir las consultas a bases de datos relacionales es el SQL. Las bases de datos relacionales (SQL) siguen las garantías de ACID, por sus siglas en inglés: atomicity, consistency, isolation y durability.

Atomicidad
Las transacciones tiene éxito o fallan como una unidad completa.

Consistencia
Los datos siempre cambian entre estados válidos.

Aislamiento
Asegura que la concurrencia no cause efectos secundarios.

Durabilidad
Después de una transacción, el estado persiste incluso en el evento de una falla del sistema.

Para garantizar estas características, las base de datos relacionales no pueden manejar adecuadamente la escalabilidad horizontal (múltiples nodos distribuidos), lo que significa que no escalan tan bien.

Discrepancia de Impendancia Objeto-Relacional (o desajuste de paradigma)

Los modelos de objeto y los modelos relacionales no funcionan muy bien juntos. Los RDBMS representan datos en un formato tabular (por ejemplo una hoja cálculo), mientras que los lenguajes orientados a objetos, como Java, los representan como un gráfico interconectado de objetos. Cargar y almacenar

gráficos de objetos usando una base de datos relacional tabular nos expone a 5 problemas de discrepancia:

Granularidad
A veces, un modelo de objetos tendrá más clases que el número de tablas correspondientes en la base de datos —decimos que el modelo de objetos es más granular que el modelo relacional—.

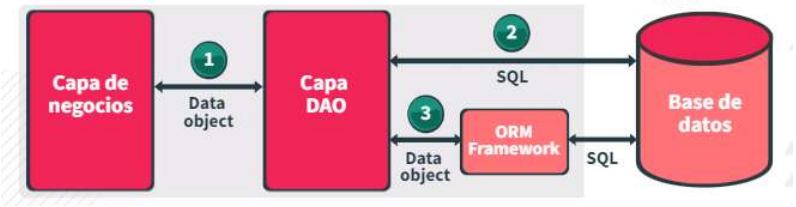
Herencia
La herencia es un paradigma natural en los lenguajes de programación orientados a objetos. Sin embargo, los RDBMS no definen nada similar en general.

Identidad
Un RDBMS define exactamente una noción de "igualdad": la clave primaria. Java, sin embargo, define tanto la identidad de objeto a == b como la igualdad de objeto a.equals (b).

Asociaciones
Se representan como referencias unidireccionales en lenguajes orientados a objetos, mientras que los RDBMS utilizan la noción de claves foráneas. Si se necesitan relaciones bidireccionales en Java, se debe definir la asociación dos veces.

Navegación de datos
La forma en que accede a los datos en Java es fundamentalmente diferente a la forma en que lo hace en una base de datos relacional. En Java, navega de una asociación a otra recorriendo la red de objetos. Esta no es una forma eficaz de recuperar datos de una base de datos relacional. Por lo general, desea minimizar el número de consultas SQL y, por lo tanto, cargar varias entidades a través de JOIN y seleccionar las entidades de destino antes de comenzar a recorrer la red de objetos.

ORM (Object relational Mapping)
Es un mecanismo que permite interactuar con una base de datos sin la necesidad de usar código SQL. Los ORMs se encargan de traducir las instrucciones en el lenguaje de programación que se esté utilizando utilizando a sentencias SQL que el gestor de base de datos pueda entender. Hay muchos ORM para Java: EJB, JDO, JPA. Mientras que estos son especificaciones, Hibernate, es una implementación.



.

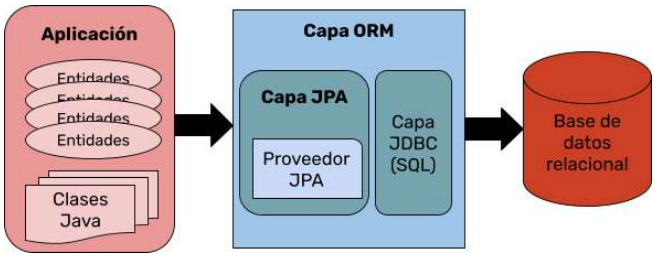
Ventajas y desventajas

Ventajas
- No se requiere usar SQL.
- El ORM permite interactuar con diferentes DBMS.
- Se pueden realizar transacciones, migraciones, triggers, y trabajar con vistas y stored procedures.

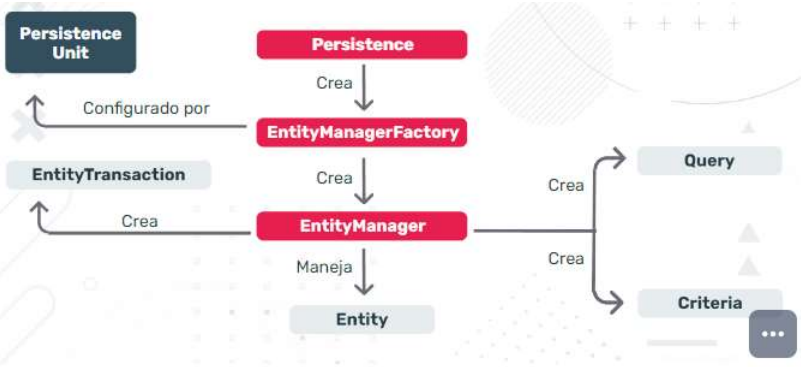
Desventajas
- En algunos casos, puede llegar a ser más compeljo que utilizar el lenguaje SQL.
- Si se trata de un servicio REST pequeño, puede ser más práctico utilizar una alternativa como JDBC.

¿Qué es JPA?
JPA o Java Persistance API es una colección de clases y métodos que almacenan de forma persistente grandes cantidades de datos en una BD. No es un framework, sino que define un conjunto de conceptos que pueden ser implementados por cualquier herramienta o framework.
JPA busca solucionar la problemática planteada al intentar traducir un modelo orientado a objetos a un modelo relacional.

Al usar JPA se crea un mapa de la BD a los objetos del modelo de datos de la aplicación. La conexión entre la BD relacional y la aplicación es gestionada por JDBC (Java database connectivity API). JPA es una API open source y existen diferentes proveedores que lo implementan, siendo utilizado en productos como por ejemplo Hibernate y Spring Data JPA.



Arquitectura de JPA:



.

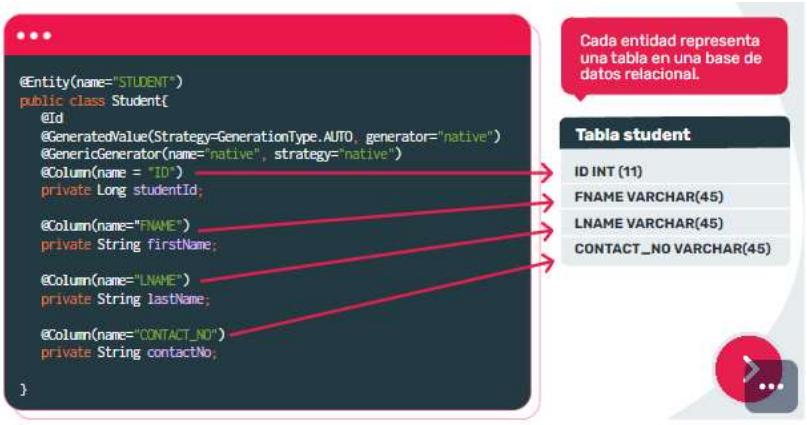
EntityManagerFactory
Es una clase Factory que crea y gestiona múltiples instancias del Entity Manager. Suele haber uno por base de datos.

EntityManager
Es una interfaz que gestiona las operaciones de persistencia en los objetos. Crea y remueve instancias de persistencia. Encuentra entidades por su clave primaria. Permite que las queries sean ejecutadas sobre entidades.

Entity
Una entidad es un objeto de persistencia. JPA utiliza anotaciones o xml para mapear entidades a una BD relacional.

EntityTransaction
Es un conjunto de operaciones SQL que son comitteadas o rolled backed como una unidad simple. Cualquier modificación iniciada por el EntityManager se coloca dentro de una transacción.

Query
Es una interfaz utilizada para controlar la ejecución de las consultas. Un EntityManager ayuda a crear un objeto Query, su implementación dependerá del proveedor de persistencia.

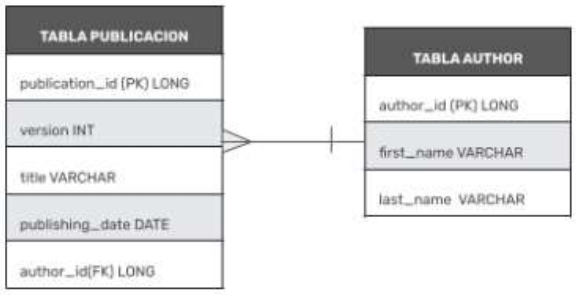


Criteria
Permite construir queries SQL usando objetos Java. Es posible realizar consultas tipadas seguras, que pueden ser chequeadas en tiempo de compilación.

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();

//create query object
CriteriaQuery<Student> query = cb.createQuery(Student.class);

//get object representing 'from' part
Root<Student> studentRoot = query.from(Student.class);

//combine 'select' and 'from' parts, equivalent to 'SELECT s FROM STUDENT s;'
```

```
query.select(studentRoot);
TypedQuery<Student> typedQuery = entityManager.createQuery(query);

typedQuery.getResultList()
      .forEach(s -> System.out.println(s.getFirstName()));
```

Mapeando del DER a clases de Java




```
// Clase Author
@Entity
public class Author{

      @Id
      @GeneratedValue(strategy=generationType.SEQUENCE)
      private Long id;

      @Column(name = "first_name")
      private String firstName;

      @Column(name = "last_name")
      private String lastName;

      @OnetoMany(mappedBy = "author")
      private Set <Publicacion> publications = HashSet<Publicacion>();

}

// Clase Publicacion
@Entity
public abstract class Publication {

@Id
@GeneratedValue(strategy=generationType.SEQUENCE)
private Long id;
private int version;
private String title;

@ManytoOne(fetch = FetchType.LAZY)
private Author author;
```

```
@Column(name = "publishing_date")
private LocalDate publishingDate;

}
```

Spring Data
Spring Framework tiene múltiples módulos disponibles para trabajar con bases de datos, agrupados en la familia Spring Data: JDBC, Cassandra, Hadoop, Elasticsearch, entre otros.

Uno de ellos es Spring Data JPA, que abstrae el acceso a las bases de datos utilizando la API de persistencia de Java en un estilo de programación basado en Spring.

Spring Boot da un paso adicional con un iniciador dedicado que usa la configuración automática y algunas herramientas adicionales para iniciar rápidamente el acceso a la base de datos: el módulo spring-boot-starterdata-jpa. También puede autoconfigurar bases de datos integradas como H2.

```
<dependencies>
	<dependency>
		<groupid>org.springframework.boot</groupid>
		<artifactid>spring-boot-starter-data-jpa</artifactid>
	</dependency>
	<dependency>
		<groupid>com.h2database</groupid>
		<artifactid>h2</artifactid>
		<scope>runtime</scope>
	</dependency>
</dependencies>
```

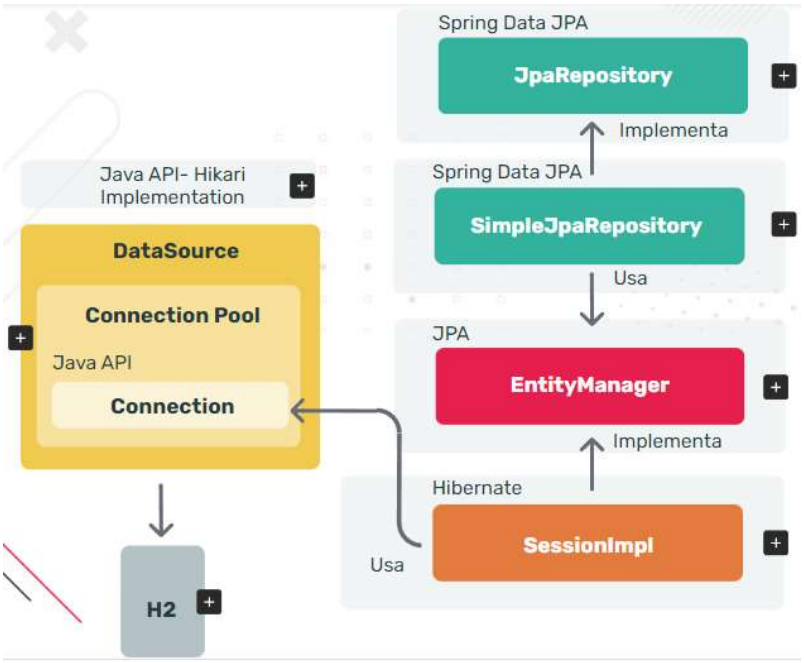
Hibernate es la implementación de referencia para JPA en Spring Boot. Eso significa que el iniciador trae las dependencias de Hibernate adentro. También incluye los artefactos principales de JPA y la dependencia con su módulo principal, Spring Data JPA.

Stack de Tecnologías de Spring Boot Data JPA

JpaRepository
Además del EntityManager de JPA, Spring Data JPA define una interfaz JpaRepository con los métodos más comunes que necesitamos usar normalmente: FIND, GET, DELETE, UPDATE, etcétera.

SimpleJpaRepository
La clase SimpleJpaRepository es la implementación predeterminada en Spring y usa EntityManager por debajo. Esto significa que no necesitamos usar el

estándar JPA puro ni Hibernate para realizar operaciones de base de datos en nuestro código, ya que podemos usar estas abstracciones de Spring.



EntityManager
SessionImpl, a través de su árbol de jerarquía, implementa la interfaz JPA EntityManager. JPA EntityManager es parte de la especificación JPA. Su implementación, en Hibernate, es lo que hace el ORM completo.

SessionImpl
Hibernate usa estas API —y, por lo tanto, la implementación de HikariCP en nuestra aplicación— para conectarse a la base de datos. El tipoJPA en Hibernate para administrar la base de datos es la clase SessionImpl, que incluye mucho código para realizar declaraciones, ejecutar consultas, manejar las conexiones de la sesión, etcétera.

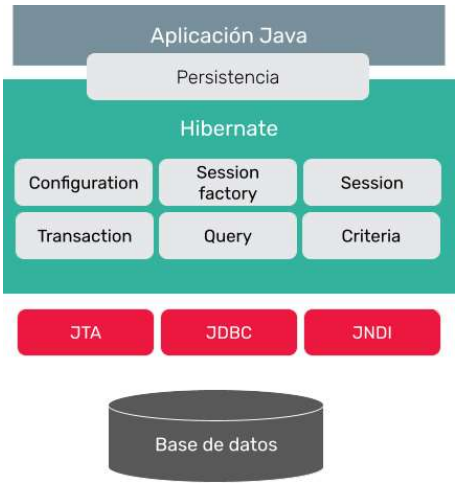
Java API-Hikari implementation
Spring Boot viene con HikariCP, que es una de las implementaciones más populares de los grupos de conexiones DataSource porque es liviana y tiene un buen rendimiento.

Connection Pool
Allí podemos encontrar las interfaces DataSource, Connection y algunas otras para recursos agrupados como ConnectionPoolDataSource. Existen múltiples implementaciones de estas APIs de diferentes proveedores.

H2
Hay algunas APIs principales de Java para manejar bases de datos SQL en los paquetes java.sql y javax.sql.

Hibernate

Es un framework ORM que ayuda a lograr la persistencia de datos. Específicamente, se ocupa de la persistencia de datos en lo que respecta a las bases de datos relacionales (RDBMS). Hibernate mapea las clases Java en tablas de BD y provee mecanismos para consultar datos.



Características

- Es open source.
- Tiene alta performance.
- Posee queries independientes de la BD: con HQL (Hibernate Query Language) es posible generar queries independientes de la BD utilizada. Antes de Hibernate si la BD era cambiada, era necesario cambiar todas las queries en la aplicación.
- Crea automáticamente las tablas.
- Simplifica joins complejos: es posible traer datos de múltiples tablas.
- Provee estadísticas a través del query caché, y sobre el status de la BD.

Anotaciones

Son una forma potente de añadir metadata para el mapeo de objetos y tablas relacionales. Esta metadata se inyecta en la clase POJO de Java a la par del código. Las siguientes anotaciones pertenecen al estándar JPA y son utilizadas por Hibernate:

@Entity

Se etiqueta a la clase como un Bean del tipo entity que va a ser mapeado por el ORM con una tabla de la BD.

@Table

Especifica detalles de la tabla que va a ser usada para persistir la entidad en la BD. Con el atributo «name» se puede explicitar el nombre de la tabla a la que debe asociarse la clase.

@Id
Cada Bean del tipo entity va a tener una Primary Key (que puede ser simple o compuesta). Esta especifica cuál es el índice, permite que la BD genere un nuevo valor con cada operación de inserción.

@GeneratedValue
Si no se utilizamos esta anotación la aplicación es responsable de gestionar por sí misma el campo @Id. El atributo strategy = GenerationType puede tener los siguientes valores:

- AUTO
Por defecto. El tipo de ID generada puede ser numérica o UUID.

@GeneratedValue(strategy=generationType.AUTO)

- IDENTITY
Asigna claves primarias para las entidades que utilizan una columna de identidad, son autoincrementales.

@GeneratedValue(strategy=generationType.IDENTITY)

- SEQUENCE
Asigna claves primarias para las entidades utilizando una secuencia que puede ser customizada.

@GeneratedValue(strategy=generationType.SEQUENCE)

- TABLE
Asigna claves primarias para las entidades utilizando una tabla de la BD, guardando en una tabla el último valor de clave primaria.

@GeneratedValue(strategy=generationType.TABLE)

@Column
Especifica los detalles de una columna para indicar a qué atributo o campo será mapeada. Puede utilizar varios atributos:

- name
- length
- nullable
- unique

Ejemplo:

```
@Entity
public class User {
        @Column(length = 3)
        private String firstName;
```

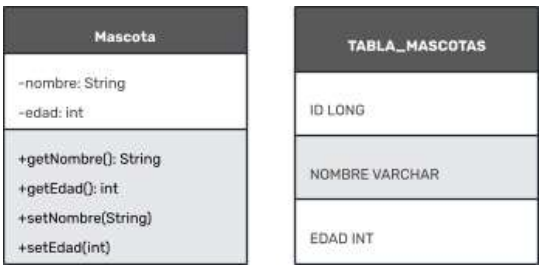
```
	// ...
}
```

Ejemplo:

```
@Entity
public class User {
	// ...
	@Column(length = 5)
	@Size(min = 3, max = 5)
	private String city;
	// ...
}
```

Ejercicio
A partir de la siguiente clase y tabla:

- Desarrollar la clase en Java.
- Construir la tabla en H2.
- Armar todo el mecanismo de persistencia utilizando Hibernate.



.

Sesión 32. Integradora 10.
Junio 22 de 2022

Sesión 33. Spring Data – Hibernate
Junio 23 de 2022

Mapeo de Relaciones

Anotaciones

Relación	Anotación
Uno a uno	@OneToOne
Uno a muchos	@OneToMany
Muchos a uno	@ManyToOne
Muchos a muchos	@ManyToMany

Relación Uno a Uno:

Navegabilidad Unidireccional:

```
// Clase Usuario
@Entity
@Table(name = "usuarios")
public class Usuario {
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	@Column(name = "id")
	private Long id;

	@OneToOne(cascade = CascadeType.ALL)
	@JoinColumn(name = "id_direccion" , referencedColumnName = "id")
	private Direccion direccion;

	// Add the getters and setters methods
}


// Clase Direccion
@Entity
@Table(name = "direcciones")
public class Direccion {

	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	@Column(name = "id")
	private Long id;

	// Add getters and setters
}
```

Navegabilidad bidireccional:

```
// Clase Direccion
@Entity
@Table(name = "direcciones")
public class Direccion {

	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	@Column(name = "id")
	private Long id;

	@OnetoOne(mappedBy = "direccion")
	private Usuario usuario;

// Add getters and setters
}
```

Relación Uno a Muchos

Navegabilidad Unidireccional

```
// Clase Carrito
@Entity
@Table(name = "carritos")
public class Carrito{

	@Id
	@GeneratedValue(strategy=generationType.IDENTITY)
	@Column(name = "id")
	private Long id;

	@OnetoMany(cascade = CascadeType.ALL)
	@JoinColumn(name = "carrito_id")
	private Set <Item> items;

	// Add getters and setters
}
```

La anotación @JoinColumn estará ubicada del lado "one" de la relación, pero hace referencia a la tabla en la base de datos del "many" de esta.

```
// Clase Item
@Entity
@Table(name = "items")
public class Item {

	@Id
```

```
    @GeneratedValue(strategy = generationType.IDENTITY)
    @Column(name = "id")
    private Long id;


    // Add getters and setters
}
```

Se puede observar que Item no tiene asociado ningún atributo del tipo Carrito, ya que la navegabilidad de la relación es de Carrito hacia Item solamente.

Navegabilidad bidireccional

```
// Clase Carrito
@Entity
@Table(name = "carritos")
public class Carrito{

    @Id
    @GeneratedValue(strategy=generationType.INDENTITY)
    @Column(name = "id")
    private Long id;

    @OnetoMany(mappedBy = "carrito")
    @JoinIgnore
    private Set <Item> items;

    // Add getters and setters
}
```

El atributo mappedBy se usa para indicar que el atributo "carrito" del lado de la clase Item es quien establece la relación y el atributo JsonIgnore se utiliza porque —en una relación bidireccional— si este objeto viaja en formato JSON, podría entrar en un bucle infinito.

```
// Clase Item
@Entity
@Table(name = "items")
public class Item {

    @Id
    @GeneratedValue(strategy = generationType.IDENTITY)
    @Column(name = "id")
    private Long id;

    @ManyToOne
    @JoinColumn(name = "carrito_id", nullable = false)
    private Carrito carrito;
```

```
        // Add getters and setters
}
```

Relación Muchos a Muchos

Navegabilidad Unidireccional

```
// Clase Contacto
@Entity
@Table(name = "contactos")
public class Contacto {

        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        @Column(name = "id")
        private Long id;

        @ManyToMany
        @JoinTable(
                name = "likes",
                joinColumns = @JoinColumn(name = "id_contacto"),
                inverseJoinColumns = @JoinColumn(name = "id_foto")
        )

        private Set <Foto> likedFotos;

        // Add getters and setters
}
```

- @JoinTable: Se usa para utilizar la tabla que asocia a las dos tablas de la base de datos. Con el atributo name se define el nombre de esta tabla.

- joinColumns: Se usa para definir la columna en la tabla asociativa que apunta al id de la clase propietaria.

- inverseJoinColumns: Se define la columna en la tabla externa que apunta al id de la tabla inversa de la asociación.

```
// Clase Foto
@Entity
@Table(name = "fotos")
public class Foto {

        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        @Column(name = "id")
```

```
        private Long id;

        // All getters and setters
}
```

Navegabilidad Bidireccional

```
// Clase Contacto
@Entity
@Table(name = "contactos")
public class Contacto {

        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        @Column(name = "id")
        private Long id;

        @ManyToMany
        @JoinTable(
                name = "likes",
                joinColumns = @JoinColumn(name = "id_contacto"),
                inverseJoinColumns = @JoinColumn(name = "id_foto")
        )

        @JsonIgnore
        private Set <Foto> likedFotos;

        // Add getters and setters
}


// Clase Foto
@Entity
@Table(name = "fotos")
public class Foto {

        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        @Column(name = "id")
        private Long id;

        @ManyToMany(mappedBy = "likedCourses")
        private Set <Contacto> contactos;

        // All getters and setters
}
```

JoinColumn, Cascade y Fetch type

@JoinColumn

Relación Uno a Uno
En una relación, @OneToOne puede utilizarse para indicar que una columna en la entidad padre se referencia a una clave primaria en la entidad hijo.

```
@Entity
public class Office {
	@OneToOne(fetch = FetchType.LAZY)
	@JoinColumn(name = "addressId")
	private Address address;
}
```

Relación Uno a Muchos
En una relación @OneToMany podemos usar el atributo "mappedBy" para indicar que la columna referencia a otra entidad.

```
@Entity
public class Employee {
	@Id
	private Long id;

	@OneToMany(fetch = FetchType.LAZY, mappedBy = "employee")
	private List  emails;
}

@Entity
public class Email {
	@ManyToOne(fetch = FetchType.LAZY)
	@JoinColumn(name = "employee_id")
	private Employee employee;
}
```

@JoinColumns
Puede utilizarse cuando se desea crear joins de múltiples columnas.

```
@Entity
public class Office{

	@ManyToOne(fetch = FetchType.LAZY)
	@JoinColumns({name = "student_id"
		@JoinColumn(name = "ADDR_ID", referencedColumnName = "ID"),
		@JoinColumn(name = "ADDR_ZIP", referencedColumnName = "ZIP")
	})
	private Address address;
}
```

Cascading

La mayoría de las veces las relaciones entre entidades dependen de la existencia de otra entidad. Sin una la otra no podría existir, y si modificamos una deberíamos modificar ambas. Por lo tanto, si realizamos una acción sobre una entidad, la misma acción debe aplicarse a la entidad asociada.

Las más utilizadas son:

Objeto Literal	Json
CascadeType.ALL	Realiza la propagación de todas las operaciones.
CascadeType.PERSIST	Propaga las operaciones de persistencia desde una entidad padre a sus hijas.
CascadeType.MERGE	Propaga las operaciones de combinación de una entidad padre a una hija.
CascadeType.REMOVE	Propaga las operaciones de borrado de una entidad padre a una hija.

```
@Entity
public class Person {
	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	private int id;
	private String name;
	@OneToMany(mappedBy = “person”, cascade = CascadeType.ALL)
	private List addresses;
}

@Entity
public class Address {
	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	private int id;
	private String street;
	private int houseNumber;
	private String city;
	private int zipCode;
	@ManyToOne(fetch = FetchType.LAZY)
	private Person person;
}
```

Fetch Type

Es requerido especificar un Fetch Type al usar cualquiera de las asociaciones (relaciones) ya que define cuando se recupera la información de la BD y se carga en la memoria.

EAGER: Eager Loading o Carga Ansiosa

Es un patrón de diseño en el cual la inicialización de los datos ocurre en el momento. La carga en memoria de registros y sus registros asociados mapeados a través de sus entidades se realizan en una consulta.
En escenarios donde hay carga de información innecesaria puede generar problemas de rendimiento.

LAZY: Lazy Loading o Carga Perezosa
la carga en memoria de registros y sus registros asociados mapeados a través de sus entidades no se inicializará y cargará hasta una llamada explícita al método get.
Proporciona tiempos más pequeños de carga y menor consumo de memoria con respecto a Eager Loading.

Ejemplo:
@OnetoMany(mappedBy = "cart", cascade = CascadeType.ALL, fetch = FetchType.EAGER)
private Set <Item> items;

Ejercicio
Crear un proyecto fútbol con dos entidades:

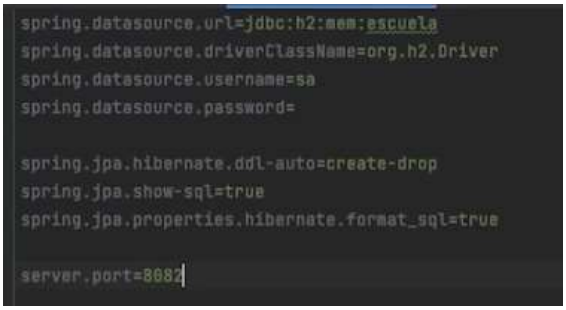
Jugador:
Nombre: String.
Puesto: String (Delantero, defensor, etc.).
Número: int.

Equipo:
Nombre: String.
Ciudad: String.

Hacer una relación 1:M entre equipo y jugadores y generar la BD desde Springboots (usar H2 en memoria).

LiveCoding:

- Agregar constructor vacío.
- Es buena práctica agregar la anotación @Table
- Cascada. Ej: Cascade_Type.ALL. Si se elimina una escuela, por cascada se eliminan todos los alumnos.



Configuración de H2 en application.properties:

- server.port=8082 (en application.properties) para cambiar el puerto por defecto.

@IdClass(value=PersonKey.class)
Para asociar una clave compuesta definida en otra clase.

Preguntas:
- @SequenceGenerator. ¿Cuándo se usa?
- ¿Por qué usar HashSet en lugar de ArrayList?
- ¿Cuándo se utiliza navegación unidireccional y cuándo navegación bidireccional?

Sesión 34. HQL
Junio 24 de 2022

¿Qué es HQL?
HQL o Hibernate Query Language es un lenguaje de consultas que proporciona Hibernate similar a SQL Standard.

Características:
- Los tipos de datos son los de Java
- Las consultas son independientes del lenguaje SQL
- Las consultas son independientes del modelo de tablas
- Trabaja con clases y atributos
- Se puede tratar con colecciones de Java
- Se puede navegar entre distintos objetos en la propia consulta

Ejemplo:

HQL:
FROM Empleado e WHERE e.id = 1

SQL:
SELECT ID, NOMBRE, APELLIDO FROM empleados WHERE ID = 1

Sintaxis de HQL

FROM
FROM Usuario

AS
FROM Usuario AS u

SELECT
SELECT u.nombre FROM Usuario u

WHERE
FROM Usuario u WHERE u.id = 999

ORDER BY
FROM Usuario u WHERE u.id > 10 ORDER BY u.nombre DESC

GROUP BY
SELECT SUM(P.salary), P.firstName FROM Person P

UPDATE
“UPDATE Employee SET salary =: salary” + “WHERE id = empId”

DELETE
“DELETE FROM Employee” + “WHERE id =: empId”

INSERT
“INSERT INTO Person(firstName, lastName, salary)” + “SELECT firstName, lastName, salary FROM old_person”

Funciones agregadas
AVG, SUM, MIN, MAX, COUNT(*), COUNT, COUNT(DISTINCT…), COUNT(ALL…)

Ejemplos

```
@Repository
public interface UsuarioRepository extends JpaRepository<usuarioModel, Long> {
	@Query(“select u from UsuarioModel u where u.userName = ?1 order by u.userName”)
}
```

Nombres de métodos para consultas (buenas prácticas)

Tipo de búsqueda	Ejemplo de nombre de método
And	**findBy**Lastname**And**Firstname
Or	**findBy**Lastname**Or**Firstname
Is, Equals	**findBy**Firstname, **findBy**Firstname**Is**, **findBy**Firstname**Equals**
Between	**findBy**StartDate**Between**
LessThan	**findBy**Age**LessThan**
LessThanEqual	**findBy**Age**LessThanEqual**
GreaterThan	**findBy**Age**GreaterThan**
GreaterThanEqual	**findBy**Age**GreaterThanEqual**
After	**findBy**StartDate**After**
Before	**findBy**StartDate**Before**
IsNull	**findBy**Age**IsNull**
IsNotNull,NotNull	**findBy**Age**NotNull**, **findBy**Age**IsNotNull**
Like	**findBy**Firstname**Like**
NotLike	**findBy**Firstname**NotLike**
StartingWith	**findBy**Firstname**StartingWith**
EndingWith	**findBy**Firstname**EndingWith**
Containing	**findBy**Firstname**Containing**
OrderBy	**findBy**Age**OrderBy**Lastname**Desc**

Ubicación de consultas en Spring

- Dentro del model (@NameQuery): Poco recomedado
- Dentro de repository (@Query): Ideal
- Dentro del service (EntityManager): Poco recomendado

Ejemplo: Consulta en Repository (recomendado)

```
@Repository
public interface UsuarioRepository extends JpaRepository<UsuarioModel,
Long> {
	@Query("select u from UsuarioModel u where u.userName = ?1 order
	by u.userName")
	List<UsuarioModel> findUsuarioModelByName(String name);
}

@service
public class UsuarioService{

	@Autowired
	UsuarioRepository usuarioRepository; // Inyección de dependencia

	public ArrayList<UsuarioModel> obtenerUsuarios(){
		return (ArrayList<UsuarioModel>) usuarioRepository.findAll();
	}

	public ArrayList<UsuarioModel> obtenerUsuariosPorNombre(String
	nombre){
		return (ArrayList<UsuarioModel>)
		// Se llama al método definido en Repository:
		usuarioRepository.findUsuarioModelByName(nombre);
	}

	public UsuarioModel guardarUsuario(UsuarioModel nuevoUsuario){
		return usuarioRepository.save(nuevoUsuario);
	}
}
```

Ejemplo: Consulta en Service (no recomendado)

```
@service
public class UsuarioService{

	// Inyectar EntityManager @PersistenceContext.
	@PersistenceContext
	public EntityManager entityManager;

	public ArrayList<UsuarioModel> obtenerUsuariosPorNombre2(String
	nombre){
		// A partir del entityManger crear la query.
		ArrayList<UsuarioModel> users =
		(ArrayList<UsuarioModel>)entityManager
		.createQuery("select u from UsuarioModel u where u.userName
		like ?1 order by u.userName")
```

```
                .setParameter(1 ,nombre)
                .getResultList();
                return users;
        }
}
```

Ejercicio
- Crear un proyecto en Spring Boot con Spring Data, que contenga una entidad Jugador con los siguientes atributos:

Nombre:String
ClubFavorito: String

- Crear el ID y una secuencia para guardar el ID.

- En el DAO agregar:
Un método para crear un nuevo jugador.
Una consulta para obtener todos los jugadores.
Una consulta para obtener un jugador por nombre.

Para los métodos que se heredan del repositorio, solo hace falta agregar un comentario.

Sesión 35. Integradora 11.
Junio 27 de 2022

Entrega del Proyecto: Viernes 8 de julio.
No incluye HQL.

Sesión 36. Taller de Coding
Junio 27 de 2022

Sesión 37. Taller de Coding
Junio 28 de 2022