UTN FACULTAD REGIONAL CÓRDOBA
INGENIERÍA EN SISTEMAS DE INFORMACIÓN

Backend de Aplicaciones
Apunte de Clases

Apunte de Clases 16 - Spring Data

Introducción

Spring Data es uno de los módulos más grandes de spring framework ya que cuenta con muchos sub módulos y cuyo principal objetivo es proveer una serie de herramientas que permitan realizar integraciones muy sencillas con los diferentes entornos de **almacenamiento de datos**.

Spring Data funciona mediante la creación de interfaces que heredan de las interfaces proporcionadas por Spring. Estas interfaces heredan funcionalidades predefinidas para operaciones comunes de acceso a datos conocidas como **CRUD** (**C**reate-**R**ead-**U**pdate-**D**elete) para crear, leer, actualizar o borrar entidades de una base datos. Lo único que se debe hacer como programador es definir estas interfaces y Spring generará **en tiempo de ejecución** una clase que implementa las funcionalidades indicadas por las interfaces.

Para utilizar Spring Data, se necesita agregar la dependencia correspondiente en el archivo pom.xml del proyecto (ó al momento de inicializar el proyecto desde initializr). Luego, en el archivo **application.properties**, se configuran los datos de conexión a la base de datos, como la URL, el usuario y la contraseña.

Funcionalidades y módulos de Spring Data

El módulo **Spring Data** forma parte del ecosistema de Spring Framework y tiene como propósito principal **simplificar el acceso a datos** en aplicaciones Java. A través de su integración con tecnologías de persistencia (como JPA, JDBC, MongoDB, entre otras), reduce drásticamente el código repetitivo necesario para realizar operaciones comunes de persistencia.

🎯 Funcionalidades clave

Spring Data ofrece las siguientes funcionalidades destacadas:

- ✅ **Repositorios automáticos**: permite definir interfaces sin implementación explícita. Spring genera la implementación en tiempo de ejecución.
- 🔍 **Consultas derivadas por convención**: se pueden construir métodos de consulta dinámicos simplemente nombrando el método siguiendo ciertas reglas (por ejemplo, `findByNombre`).
- 📦 **Clases base reutilizables**: proporciona jerarquías de interfaces como `CrudRepository`, `JpaRepository`, etc., que encapsulan operaciones comunes de CRUD y paginación.
- 🕓 **Auditoría transparente**: facilita el manejo automático de campos como fecha de creación, última modificación y usuarios relacionados.
- 🤝 **Integración fluida con el core de Spring**: compatible con otros módulos como Spring Web, Spring Security y Spring Boot, simplificando el desarrollo y configuración.

📚 Principales submódulos de Spring Data

Spring Data está compuesto por varios submódulos que permiten trabajar con distintas tecnologías de almacenamiento de datos. Algunos de los más relevantes son:

- **Spring Data JPA**: Integración con JPA y Hibernate para bases de datos relacionales.
- **Spring Data JDBC**: Alternativa más ligera y directa que prescinde de algunas abstracciones de JPA.
- **Spring Data MongoDB**: Soporte para documentos en bases MongoDB.
- **Spring Data Redis**: Integración con bases de datos en memoria como Redis.
- **Spring Data Cassandra**: Soporte para bases de datos distribuidas basadas en columnas como Cassandra.
- **Spring Data Elasticsearch**: Para indexación y búsquedas full-text en Elasticsearch.
- **Spring Data LDAP**: Acceso a directorios LDAP.
- **Spring Data REST**: Exposición automática de repositorios como servicios REST.

🧩 Interfaces de repositorio más utilizadas

Spring Data proporciona una jerarquía de interfaces genéricas para construir repositorios. Las más comunes son:

Interfaz	Extiende	Descripción
`CrudRepository<T, ID>`	–	Métodos CRUD básicos (`save`, `findById`, `delete`, etc.).
`PagingAndSortingRepository<T, ID>`	`CrudRepository`	Agrega paginación y ordenamiento (`findAll(Pageable)`).
`JpaRepository<T, ID>`	`PagingAndSortingRepository`	Funciones adicionales como `flush`, `deleteInBatch`, `getOne`, etc. Ideal para JPA.
`MongoRepository<T, ID>`	–	CRUD y filtros para documentos en MongoDB.
`ElasticsearchRepository<T, ID>`	–	Indexación y búsqueda en Elasticsearch.
`Neo4jRepository<T, ID>`	–	Consultas para bases de grafos (Neo4j).
`CassandraRepository<T, ID>`	–	Soporte para tablas distribuidas (Cassandra).
`RedisRepository<T, ID>`	–	Repositorios sobre clave-valor en Redis.

> 🔍 Cada una de estas interfaces está especializada en un tipo de base de datos o motor de persistencia. En proyectos JPA con Hibernate, lo más habitual es extender `JpaRepository`.

Esta estructura permite construir rápidamente una capa de acceso a datos potente, reutilizable y alineada con buenas prácticas, sin necesidad de escribir SQL manual para operaciones estándar.

Interfaz CrudRepository

La interfaz `CrudRepository` en Spring Data proporciona una serie de métodos CRUD (Create, Read, Update, Delete) comunes para operaciones de gestión de datos en la base de datos. Algunos de los métodos incluidos en la interfaz `CrudRepository` son:

1. `save(S entity)`: Guarda una entidad en la base de datos. Si la entidad ya existe, se actualiza; de lo contrario, se crea una nueva entrada.
2. `saveAll(Iterable<S> entities)`: Guarda una colección de entidades en la base de datos.
3. `findById(ID id)`: Recupera una entidad por su identificador (clave primaria).
4. `existsById(ID id)`: Verifica si existe una entidad con el identificador proporcionado.
5. `findAll()`: Recupera todas las entidades del tipo especificado.
6. `findAllById(Iterable<ID> ids)`: Recupera todas las entidades cuyos identificadores coinciden con los proporcionados en la lista.
7. `count()`: Obtiene el número total de entidades en la base de datos.
8. `deleteById(ID id)`: Elimina una entidad por su identificador.
9. `delete(T entity)`: Elimina una entidad de la base de datos.
10. `deleteAll()`: Elimina todas las entidades del tipo especificado en la base de datos.
11. `deleteAll(Iterable<? extends T> entities)`: Elimina una colección de entidades de la base de datos.

Estos métodos proporcionan una funcionalidad básica y común para realizar operaciones de lectura, escritura y eliminación en la base de datos. La interfaz `CrudRepository` es una de las interfaces fundamentales de Spring Data y se utiliza como base para crear repositorios personalizados que gestionan entidades específicas en la base de datos.

> Nota: la interfaz JpaRepository agrega un método que en ocasiones suele ser útil, **saveAndFlush()**.Este método, además de guardar la entidad en el contexto de persistencia, fuerza la escritura inmediata de los cambios en la base de datos. En otras palabras, realiza una operación de flush inmediatamente después de guardar la entidad. Esto significa que los cambios se escriben en la base de datos de inmediato y cualquier error de persistencia se detecta en ese momento.

Primer repositorio

Continuando con el caso de estudio citado anteriormente, se propone refactorizar la abstracción de datos inicial utilizando un repositorio de Spring Data JPA. En este caso los datos de las personas registradas al

evento serán persistidas en una base de datos MySQL con la siguiente estructura:

- id (long) [PK]
- documento (entero) [no nulo],
- nombre (varchar(50)) [no nulo],
- apellido (varchar(50)) [no nulo],
- fecha de nacimiento (datetime)
- extranjero (char) [no nulo, valor por defecto 'N']

El script de creación del esquema **personas_db** junto con la tabla **personas** se encuentra en el archivo personas.sql

Lo primero será crear un nuevo proyecto de Spring Boot mediante Spring Initializr. Tener presente incluir las dependencias de "Spring Web", "Lombok" y "Spring Data JPA".

Luego se crea una clase Java para representar la entidad **Persona** tal como se muestra en el siguiente código:

```
package com.example.personasData.models;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import java.time.LocalDateTime;
import lombok.Data;

@Entity
@Data
public class Persona {

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

    private Long documento;
    private String nombre;
    private String apellido;
    private LocalDateTime fechaNacimiento;
    private char extranjero;
}
```

> *Notas*:
>
> - tener en cuenta que si no se indica lo contrario el atributo fechaNacimiento será mapeado en la tabla como **fecha_nacimiento**.
> - Formato de fecha por defecto para MySql es 'yyyy-mm-dd'

La anotación **@Data** de Lombok genera automáticamente los métodos equals, hashCode, toString, así como los getters y setters para todos los campos de la clase. Esto reduce significativamente la cantidad de

código que necesitas escribir manualmente y hace que tu clase sea más concisa y fácil de mantener.

La anotación **@Entity** se utiliza para marcar una clase de Java como una entidad persistente, lo que significa que los objetos de esta clase se guardarán y se recuperarán en una base de datos. Una entidad se mapea generalmente a una tabla en una base de datos relacional. La anotación @Entity debe colocarse encima de una clase que represente un objeto que deseas almacenar en la base de datos.

La anotación **@Id** se utiliza para marcar una propiedad en una clase como la clave primaria de la entidad. La clave primaria es un campo único que identifica de manera única una fila en una tabla de la base de datos. Cada entidad debe tener una propiedad marcada con @Id. Combinada con esta anotación se utiliza **@GeneratedValue** para indicar cómo se generará automáticamente el valor de la clave primaria cuando se inserte una nueva entidad en la base de datos. Puede tomar diferentes estrategias de generación, como GenerationType.IDENTITY, GenerationType.SEQUENCE, GenerationType.AUTO, entre otras, dependiendo del sistema de gestión de bases de datos que estés utilizando.

En la capa de acceso a datos creamos un **repositorio** CRUD para la entidad mediante el siguiente código:

```
import org.springframework.data.repository.CrudRepository;

public interface PersonaRepository extends CrudRepository<Persona, Long> {
    // Spring Data automáticamente proporciona métodos CRUD básicos
}
```

> Notar que solo se necesita extender de la interfaz CrudRepository indicando la clase y el tipo de datos del campo que será mapeado como clave primaria (CrudRepository<Persona, Long>), sin necesidad de escribir ningún código SQL para las operaciones básicas de manipulación de la entidad.

Por último se configura el conector Java-MySql como una dependencia más del proyecto agregando en el pom.xml del proyecto:

```
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <scope>runtime</scope>
</dependency>
```

A su vez es necesario definir los parámetros de conexión hacia el origen de datos en el archivo application.properties, tal como se muestra en el siguiente código: `

```
spring.datasource.url=jdbc:mysql://localhost:3333/personas_db
spring.datasource.username=111mil
spring.datasource.password=111mil
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

donde:

- **spring.datasource.url**: Esta propiedad define la URL de conexión a la base de datos. En este caso, estás utilizando MySQL como base de datos y conectándote a **localhost** en el puerto **3333** a una base de datos llamada **personas_db**.
- **spring.datasource.username**: Aquí especificas el nombre de usuario utilizado para autenticarte en la base de datos.
- **spring.datasource.password**: Esta propiedad contiene la contraseña asociada al nombre de usuario para la autenticación en la base de datos. Siempre tener en cuenta que almacenar contraseñas en texto plano en archivos de configuración no es una buena práctica de seguridad. En entornos de producción, se recomienda utilizar soluciones de gestión de secrets para manejar las contraseñas de manera segura.
- **spring.datasource.driver-class-name**: Indica la clase del controlador de la base de datos que se utilizará para establecer la conexión. En este caso, estás utilizando el controlador com.mysql.cj.jdbc.Driver.

El código completo del proyecto se encuentra en Spring boot - Personas Data.

> Notas:
>
> - Para probar la ejecución de los servicios expuestos en la API, por defecto se publica en **localhost:8080/api/personas**. Si se quiere cambiar el puerto es necesario agregar el archivo .properties del proyecto la siguiente línea:
>
> ```
> server.port=xxxx
> ```
>
> - Para probar las funcionalidades de la API Personas se deja una colección de pruebas POSTMAN en el siguiente link.
> - Si al intentar registrar una persona se obtiene un código de estado 400 (BAD_REQUEST) posiblemente el campo fechaNacimiento se este mandando en un formato incorrecto.

Consultas dinámicas

Spring Data provee una herramienta en la cual se construyen consultas dinámicas únicamente utilizando un nombramiento de los métodos de un repositorio. Cuando se definen métodos en los repositorios con nombres que comienzan con findBy, Spring Data JPA analiza el nombre del método y genera automáticamente la consulta SQL correspondiente en función del nombre y los parámetros del método. Éstas consultas también suelen ser nombradas como **consultas derivas**.

La estructura general de un método que utiliza findBy es la siguiente:

```
List<Entity> findByPropertyName(ParameterType parameter);
```

Donde:

- **Entity**: Es la entidad JPA sobre la que deseas hacer la consulta.

- **PropertyName**: Es el nombre de un atributo o campo de la entidad por el cual deseas filtrar.
- **ParameterType**: Es el tipo de dato del parámetro que pasas al método para establecer el criterio de búsqueda.

Por ejemplo, continuando con el ejercicio anterior, si se necesita recuperar todas las personas por nombre es posible agregar el siguiente método en la definición de la interfaz de repositorio:

```
 List<Persona> findByNombre(String nombre);
En este caso, el método **findByNombre** generará automáticamente una
consulta SQL que busca todas las personas cuyo nombre coincida con el
valor proporcionado.

Si se desea buscar personas por nombre y apellido, es posible combinar los
atributos en el nombre del método:

```Java
public interface PersonaRepository extends JpaRepository<Persona, Long> {
    List<Persona> findByNombreAndApellido(String nombre, String apellido);
}
```

> Nota: cabe resaltar el conector **And** entre los nombres de atributos. Si por ejemlo el método se nombrara findByNombreOrApellido, entonces la búsqueda sería por nombre o apellido coincidentes con los parámetros recibidos.

Para más detalle de palabras clave de repositorio consultar aquí

Además de findBy, Spring Data JPA también admite otros prefijos, como **countBy**, **deleteBy**, entre otros. Estos prefijos permiten realizar operaciones comunes como contar o eliminar registros según ciertos criterios sin necesidad de escribir consultas SQL completas.

Es importante tener en cuenta que si se necesita realizar consultas más complejas que no pueden ser generadas automáticamente por los métodos findBy, Spring Data JPA también permite definir consultas personalizadas utilizando la anotación @Query. Esto es más flexibilidad para expresar consultas específicas en lenguaje SQL o JPQL (Java Persistence Query Language).

Consultas mediante @Query

La anotación **@Query** en Spring Data JPA se utiliza para declarar consultas personalizadas que se ejecutarán en la base de datos. Esta anotación permite escribir consultas en lenguaje de consulta específico (como JPQL o SQL nativo) en lugar de depender únicamente de los métodos generados automáticamente por Spring Data JPA. SI las consultas están escritas en SQL puro suelen nombrarse como **consultas nativas**.

En esencia, **@Query** permite definir una consulta personalizada que será utilizada por Spring Data JPA para acceder a los datos en la base de datos. Esto puede ser útil cuando las consultas requeridas son más complejas o no pueden expresarse fácilmente utilizando los métodos de consulta generados automáticamente.

La anotación @Query se puede aplicar en dos niveles:
```

- **Nivel** de método en el **repositorio**: Puedes aplicar @Query directamente en un método en tu interfaz de repositorio para indicar una consulta específica para ese método.
- **Nivel de entidad** en un repositorio personalizado: También puedes definir un método abstracto en una interfaz de repositorio y proporcionar la implementación en un repositorio personalizado que utilice la anotación @Query.

Por ejemplo supondiendo que se desea agregar al repositorio de Personas una consulta por nombre, apellido y edad, y se desea buscar personas cuyas edades estén dentro de un rango dado y cuyos nombres contengan cierta cadena:

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import java.util.List;

public interface PersonaRepository extends JpaRepository<Persona, Long> {

    @Query("SELECT p FROM Persona p WHERE p.fechaNacimiento BETWEEN
:fechaInicio AND :fechaFin AND LOWER(p.nombre) LIKE %:nombre%")
    List<Persona> buscarPorRangoFechasYNombre(LocalDateTime fechaInicio,
LocalDateTime fechaFin, String nombre);
}
```

En este ejemplo, se utiliza la anotación @Query para definir la consulta personalizada. La consulta está escrita en JPQL (Java Persistence Query Language), que es un lenguaje similar a SQL pero específico para JPA. La consulta puede leerse como:

- **SELECT p FROM Persona p**: seleccionar entidades Persona y nombrarlas como **p** para su uso en la consulta.
- **WHERE p.edad BETWEEN :minEdad AND :maxEdad**: se está filtrando las personas cuyas edades están dentro del rango especificado por minEdad y maxEdad.
- **AND LOWER(p.nombre) LIKE %:nombre%**: se está agregando otra condición al filtro para que los nombres coincidan (en minúsculas) con la cadena nombre.

Los parámetros **:minEdad**, **:maxEdad** y **:nombre** son marcadores de posición que se vinculan a los parámetros del método buscarPorEdadYNombre.

Para más detalles sobre JPQL consultar la referencia aquí.

🧩 Uso de `JpaRepository` y comparación con `CrudRepository`

Aunque `CrudRepository` es suficiente para implementar operaciones básicas como guardar, buscar y eliminar entidades, Spring recomienda utilizar `JpaRepository` por su mayor funcionalidad y compatibilidad con JPA y Hibernate.

Diferencias principales

Característica	`CrudRepository`	`JpaRepository`
Métodos CRUD básicos	✅	✅
Paginación y ordenamiento	❌	✅ *(hereda de `PagingAndSortingRepository`)*
`saveAll`, `flush`, `deleteInBatch`	❌	✅
Soporte nativo para `@Query` y proyecciones	❌	✅

Ejemplo con `JpaRepository`:

```
public interface PersonaRepository extends JpaRepository<Persona, Long> {
    List<Persona> findByNombreContainingIgnoreCase(String nombre);
}
```

> Esto permite escribir consultas como findByNombreContainingIgnoreCase("juan"), sin escribir SQL ni JPQL, aprovechando la convención de nombres.

✅ Buenas prácticas al trabajar con Spring Data JPA

- 📦 **Extender `JpaRepository` en lugar de `CrudRepository`**: brinda soporte adicional como paginación, ordenamiento y métodos adicionales (`saveAll`, `flush`, `deleteAllInBatch`, etc.).
- 🧠 **Mantener la lógica de negocio en los servicios (`@Service`)**: los repositorios deben centrarse únicamente en el acceso a datos. Evitá colocar lógica de negocio en las interfaces de repositorio o sus implementaciones.
- 🧈 **Anotar con `@Transactional` los métodos que lo requieran**: especialmente cuando se combinan múltiples operaciones de persistencia que deben ejecutarse como una unidad atómica.
- ⚠️ **Evitar `findAll()` o `deleteAll()` sin restricciones** en tablas con grandes volúmenes de datos: preferir paginación (`findAll(Pageable)`) o condiciones específicas.
- 🔐 **No exponer directamente entidades JPA en la API**: crear DTOs (Data Transfer Objects) para definir la estructura de los datos que se intercambian con el frontend.
- 🧪 **Utilizar `@DataJpaTest` o bases embebidas (como H2)** para realizar pruebas unitarias sobre la capa de acceso a datos.
- 🧩 **Documentar consultas complejas** cuando se use `@Query`, especialmente si incluyen funciones agregadas, joins o filtros múltiples.
- 🧹 **Controlar los nombres en métodos derivados** (`findByNombreAndApellido`, etc.): deben seguir las convenciones de Spring para evitar errores en tiempo de ejecución.
- 👄 **Proteger campos sensibles con `@JsonIgnore`** o usar `@JsonView` si se serializan entidades en APIs REST, evitando exponer datos innecesarios o privados.
- 📚 **Separar los paquetes lógicamente**: agrupar entidades, repositorios, servicios y controladores en paquetes distintos para mantener un proyecto limpio y mantenible.

🛠️ Implementación de un Custom Repository en Spring Data JPA

Spring Data JPA permite definir interfaces de repositorio que Spring implementa automáticamente. Sin embargo, en ciertos casos necesitamos lógica más compleja que no puede expresarse mediante métodos derivados ni con `@Query`. Para estos escenarios, podemos **crear un repositorio personalizado** (custom repository).

Este patrón combina la potencia de Spring Data con la flexibilidad del `EntityManager` de JPA.

🎯 Objetivo del ejemplo

Queremos contar cuántas personas cumplen años en un mes determinado. Esta lógica requiere manipulación sobre una función de fecha (`MONTH()`), lo cual es difícil de expresar con una query derivada.

🧩 Paso 1: Definir una interfaz con el método personalizado

Creamos una interfaz con el nombre `PersonaRepositoryCustom` y declaramos el método que vamos a implementar:

```
public interface PersonaRepositoryCustom {
    long contarCumpleanierosDelMes(int mes);
}
```

Esta interfaz **no extiende de `JpaRepository` ni `CrudRepository`**, ya que será utilizada como extensión personalizada.

🛠️ Paso 2: Implementar la lógica con `EntityManager`

Creamos una clase con el mismo nombre que el repositorio principal seguido de `Impl`, es decir: `PersonaRepositoryImpl`.

```
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
import org.springframework.stereotype.Repository;

@Repository
public class PersonaRepositoryImpl implements PersonaRepositoryCustom {

    @PersistenceContext
    private EntityManager em;

    @Override
    public long contarCumpleanierosDelMes(int mes) {
        return em.createQuery(
            "SELECT COUNT(p) FROM Persona p WHERE FUNCTION('MONTH',
p.fechaNacimiento) = :mes", Long.class)
            .setParameter("mes", mes)
            .getSingleResult();
    }
}
```

Detalles importantes:

- `@PersistenceContext` inyecta el `EntityManager`, que permite ejecutar consultas JPQL o SQL.
- Se usa `FUNCTION('MONTH', ...)` para invocar una función de fecha (compatible con H2, MySQL, PostgreSQL).
- El método retorna un `long` con la cantidad de resultados.

🔗 Paso 3: Extender ambas interfaces en el repositorio principal

Ahora conectamos todo en la definición de nuestro repositorio principal:

```
public interface PersonaRepository extends JpaRepository<Persona, Long>,
PersonaRepositoryCustom {
    // Métodos derivados + método personalizado
}
```

Esto le indica a Spring que, además de los métodos de `JpaRepository`, debe buscar la implementación de `PersonaRepositoryCustom` en una clase llamada `PersonaRepositoryImpl`.

💡 Paso 4: Usar desde un servicio de aplicación

Podemos invocar nuestro método personalizado desde un `@Service` de Spring:

```
@Service
public class PersonaService {

    private final PersonaRepository personaRepo;

    public PersonaService(PersonaRepository personaRepo) {
        this.personaRepo = personaRepo;
    }

    public long obtenerCantidadDeCumpleañerosDelMes(int mes) {
        return personaRepo.contarCumpleanierosDelMes(mes);
    }
}
```

✅ Ventajas del enfoque

- Permite mantener la lógica de acceso a datos encapsulada y aislada del servicio.
- Evita saturar la interfaz principal con lógica específica o consultas complejas.
- Se puede testear de manera independiente.
- Es extensible: podés definir múltiples interfaces `XRepositoryCustom` según tu necesidad.

Este patrón es especialmente útil para:

- Consultas con agregaciones (`COUNT`, `SUM`, `AVG`, etc.)
- Consultas dinámicas o condicionales

- Invocación de stored procedures
- Búsquedas con múltiples filtros opcionales

Ejemplo: Entradas Kempes

El siguiente ejemplo corresponde a un servicio desarrollado con Springboot y Spring Data para registrar las entradas nominadas a un evento en el Estadio Kempes. El servicio toma los datos de la persona y genera la entrada con los datos para el evento recibido también como argumento. Este ejemplo será retomado en clases posteriores para abordar temas de enrutamiento dinámico a un punto de entrada cuando se tiene un ecosistema de microservicios.

Tomando el mismo esquema de base de datos del ejemplo anterior, es necesario el siguiente comando DDL para la creación de la tabla entradas:

```
CREATE TABLE IF NOT EXISTS `personas_db`.`entradas` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `apellido_nombre` INT NOT NULL,
  `id_evento` INT NOT NULL,
  `socio` CHAR(1) NOT NULL DEFAULT 'N',
  PRIMARY KEY (`id`))
ENGINE = InnoDB;
```

El proyecto Springboot permite crear una API Rest que expone solo un punto de acceso por método POST y recibe los datos de la entrada a registrar. El siguiente ejemplo muestra el formato JSON de la entrada:

```
{
  "evento": {
    "id_evento": 1,
    "nombre": "Concierto de Verano",
    "fecha": "2023-08-15",
    "hora": "19:30"
  },
  "persona": {
    "nombreCompleto": "Juan Pérez",
    "esSocio": "S"
  }
}
```

El código completo del proyecto se encuentra en Spring Data - Entradas Kemples.

UTN FACULTAD REGIONAL CÓRDOBA
INGENIERÍA EN SISTEMAS DE INFORMACIÓN

Backend de Aplicaciones
Apunte de Clases

Apunte de Clases 17 - Proyectos Spring - Elementos transversales

Este documento continúa el trabajo realizado sobre el ejemplo práctico de una API REST de gestión de personas, implementada con Spring Boot, Spring Web y Spring Data JPA. A partir de esta base, se desarrollan los elementos transversales clave para el diseño profesional de aplicaciones backend.

1. División en capas y responsabilidades

Una buena arquitectura en aplicaciones Spring Boot se basa en la **separación clara de responsabilidades**. Esto permite mantener el código modular, testable y fácil de mantener.

Capas principales y sus responsabilidades

📍 Controller

- Se compone de componentes anotados con `@RestController` y `@RequestMapping`.
- Expone endpoints REST utilizando anotaciones como `@GetMapping`, `@PostMapping`, etc.
- Mapea parámetros de entrada desde path, query params o body hacia objetos Java.
- Retorna las respuestas HTTP correctamente estructuradas (status codes, headers, body).
- No debe contener lógica de negocio, sino delegar esa responsabilidad al servicio.
- Puede aplicar transformación entre DTOs y entidades.

Ejemplo de responsabilidad:

> Decidir si una respuesta HTTP será 200 OK, 404 Not Found o 201 Created según el resultado del servicio.

⚙️ Service

- Centraliza la lógica de negocio: validaciones, cálculos, transformaciones y reglas del dominio.
- Orquesta acciones sobre múltiples repositorios si es necesario.
- Incluso puede orquestar conexiones a otros microservicios
- Puede aplicar patrones de diseño como Strategy, State, Chain of Responsibility u otros para modelar reglas complejas.
- Detecta y lanza excepciones semánticas del negocio (por ejemplo: `PersonaDuplicadaException`).

Ejemplo de responsabilidad:

> Antes de crear una persona, verificar si ya existe otra con el mismo DNI. Aplicar una regla de negocio sobre un estado para permitir o rechazar una operación.

🗃️ Repository

- Encapsula la relación con la base de datos o esquema de persistencia de cada entidad o modelo.
- Define la interfaz de acceso a datos usando `JpaRepository`, `CrudRepository` u otras variantes.
- Se encarga de leer y persistir entidades en la base de datos.
- No contiene lógica de negocio, aunque puede contener consultas complejas personalizadas.

Ejemplo de responsabilidad:

> Buscar personas por barrio, por nombre, o entre dos fechas de nacimiento.

📦 Model

- Representa entidades del dominio que se mapean a tablas de la base de datos.
- Incluye anotaciones JPA como `@Entity`, `@Id`, `@ManyToOne`, etc.
- Puede incluir validaciones estructurales (ej: `@NotNull`, `@Size`).

Ejemplo de responsabilidad:

> Definir que el atributo `nombre` de una persona es obligatorio y no puede superar los 60 caracteres.

🔄 DTO (opcional)

- Separa los objetos de entrada/salida del API de las entidades internas del sistema.
- Permite ocultar o transformar campos, enriquecer con datos adicionales, o validar formatos específicos.

Ejemplo de responsabilidad:

> Enviar solo nombre completo, edad y ciudad en la respuesta del endpoint público sin exponer el ID o fecha de nacimiento exacta.

Estructura de paquetes sugerida

```
utnfc.isi.back.spring.personas
├── controllers  --> PersonaController.java
├── services     --> PersonaService.java
├── repositories --> PersonaRepository.java
├── models       --> Persona.java
├── dtos         --> PersonaDTO.java (si aplica)
└── exceptions --> GlobalExceptionHandler.java, ServiceException.java
```

Flujo típico de una petición HTTP

1. El cliente realiza una petición HTTP al controlador.
2. El `@RestController` delega en el servicio la ejecución de la lógica.
3. El servicio consulta o modifica entidades a través del/os repositorio/s.
4. El resultado se transforma en una respuesta (`ResponseEntity`) y se devuelve.

🔁 2. Manejo de respuestas HTTP con `ResponseEntity`

Spring facilita el retorno de respuestas HTTP bien formadas usando `ResponseEntity`, que encapsula:

- Código de estado (`200`, `201`, `204`, `400`, `404`, etc.)
  - Ver HHTP Cats
- Headers personalizados (si es necesario)
- Cuerpo de respuesta (normalmente un objeto serializado como JSON)

Un ejemplo concreto típico al obtener una persona por ID desde el controlador sería:

```
@GetMapping("/{id}")
public ResponseEntity<Persona> obtenerPorId(@PathVariable Long id) {
    Optional<Persona> persona = personaService.buscarPorId(id);
    // Equivalente a: persona.map(p -> ResponseEntity.ok(p))
    return persona
        .map(ResponseEntity::ok)
        .orElseGet(() -> ResponseEntity.notFound().build());
}
```

> Este enfoque aprovecha el uso de `Optional` y devuelve de forma clara:
>
> - `200 OK` con el objeto si existe.
> - `404 Not Found` si no se encontró el recurso.

Ejemplos comunes

```
return ResponseEntity.ok(persona); // 200 OK
return ResponseEntity.status(HttpStatus.CREATED).body(persona); // 201
Created
return ResponseEntity.noContent().build(); // 204 No Content
```

> Se recomienda evitar retornar directamente objetos sin `ResponseEntity`, ya que esto dificulta controlar los status codes y headers de forma explícita.

3. Validaciones y manejo de errores globales

Las validaciones se implementan en el modelo utilizando anotaciones como:

```
@NotBlank(message = "El nombre es obligatorio")
private String nombre;

@Past(message = "La fecha debe ser anterior a hoy")
private LocalDate fechaNacimiento;
```

El controlador puede recibir estas validaciones automáticamente con `@Valid`:

```
@PostMapping
public ResponseEntity<Persona> crear(@Valid @RequestBody Persona persona)
{ ... }
```

Para capturar y estructurar los errores de forma global, se utiliza un `@ControllerAdvice`:

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<Map<String, String>>
handleValidationErrors(MethodArgumentNotValidException ex) {
        Map<String, String> errores = new HashMap<>();
        ex.getBindingResult().getFieldErrors().forEach(error ->
            errores.put(error.getField(), error.getDefaultMessage()));
        return ResponseEntity.badRequest().body(errores);
    }

    @ExceptionHandler(ServiceException.class)
    public ResponseEntity<String> handleServiceException(ServiceException
ex) {
        return
ResponseEntity.status(HttpStatus.BAD_REQUEST).body(ex.getMessage());
    }
}
```

Esto permite respuestas JSON coherentes incluso ante errores, mejorando la experiencia de uso de la API y facilitando el debugging.

📁 4. Documentación del API con Swagger / OpenAPI

Para facilitar la exploración y el entendimiento de una API REST, es altamente recomendable documentarla usando **OpenAPI**. En el ecosistema Spring, la forma más simple y moderna de hacerlo es utilizando la biblioteca **springdoc-openapi**.

🧩 ¿Qué es springdoc-openapi?

Es una biblioteca que analiza automáticamente las anotaciones de tus controladores (`@RestController`, `@RequestMapping`, etc.) y genera una especificación OpenAPI 3.0, accesible desde una interfaz web (Swagger UI).

🧠 Dependencia Maven

Agregar al `pom.xml`:

```
<dependency>
  <groupId>org.springdoc</groupId>
```

```
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
  <version>2.3.0</version>
</dependency>
```

🚀 Acceso a Swagger UI

Una vez iniciada la aplicación, se puede acceder a la documentación navegando a:

```
http://localhost:8080/swagger-ui.html
```

Esto abre una interfaz gráfica para:

- Visualizar todos los endpoints expuestos
- Probar peticiones directamente desde el navegador
- Ver descripciones, parámetros, request body y responses

📌 Personalización básica

Podés agregar descripciones, ejemplos y tags usando anotaciones como:

```
@Operation(summary = "Buscar persona por ID", description = "Devuelve una
persona si existe")
@ApiResponses(value = {
    @ApiResponse(responseCode = "200", description = "Persona
encontrada"),
    @ApiResponse(responseCode = "404", description = "No se encontró la
persona")
})
@GetMapping("/{id}")
public ResponseEntity<Persona> obtenerPorId(@PathVariable Long id) { ... }
```

Para usar estas anotaciones, agregá al `pom.xml`:

```
<dependency>
  <groupId>io.swagger.core.v3</groupId>
  <artifactId>swagger-annotations</artifactId>
  <version>2.2.20</version>
</dependency>
```

🧠 Ventajas

- Cero configuración: Swagger se genera automáticamente.
- Mejora la comunicación entre desarrolladores backend y frontend.
- Permite probar la API sin necesidad de Postman o Curl.
- Puede exportarse a JSON/YAML para documentación formal.

🧪 5. Testing unitario e integrado

Se muestra cómo testear cada capa:

- Repositorios con `@DataJpaTest`
- Servicios con Mockito y pruebas de lógica
- Controladores con `@WebMvcTest` y MockMvc

No avanzamos aquí con Test unitarios a niveles de servidor porque no lo vamos requerir en el TPI pero se deja constancia de su existencia para el propio avance y estudio del estudiante.

> En este documento no se incluirán detalles sobre seguridad o logging, ya que serán abordados en unidades posteriores con materiales específicos.