Desarrollo de backends con Spring

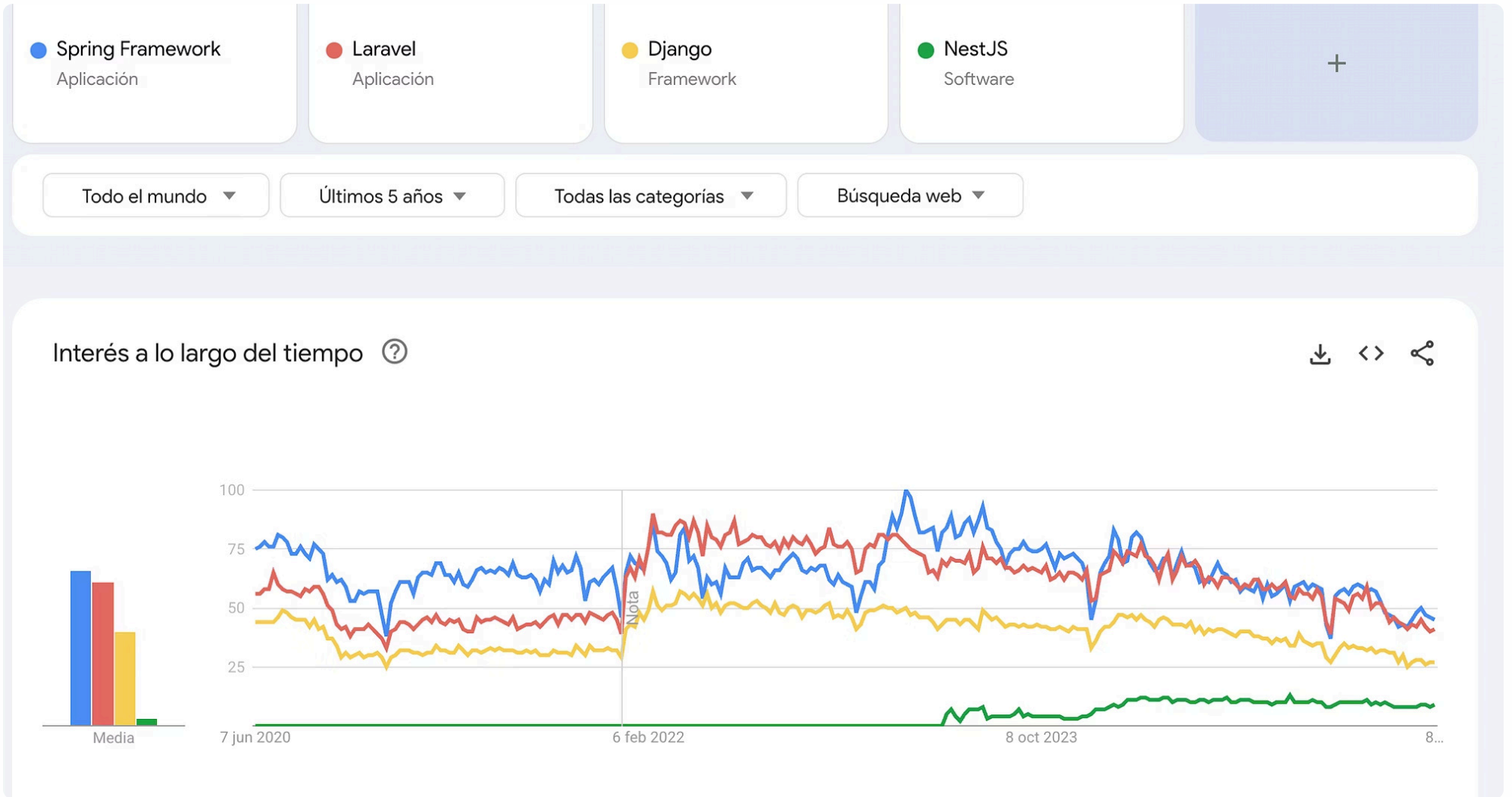
Solución solida para el desarrollo de backends

 **por Ivan Ruiz Rube**

Agenda

1. Arquitectura por capas
2. Diseño de API
3. Spring MVC
4. Spring HATEOAS
5. Spring Data JPA
6. Spring Data REST

Comparativa frameworks de back-end



Principios y patrones de diseño de software

Fundamentos que guían el desarrollo de aplicaciones robustas y mantenibles.

Código Limpio

Mejores prácticas en cada lenguaje de programación.

Principios básicos del diseño OO

SOLID: Responsabilidad única, Abierto-cerrado, Sustitución de Liskov, Segregación de interfaces, Inversión de dependencias.

Alta cohesión, bajo acoplamiento

Patrones de Diseño

GoF: Factory, Singleton, Observer, Strategy, Decorator.

Patrones arquitectónicos

Pipeline, MVC, plug-ins, event-driven, capas (clean, hexagonales, ports and adapter, onion..).

Arquitectura por Capas

Organización estructurada del software que separa responsabilidades y favorece la mantenibilidad.

Capa de presentación

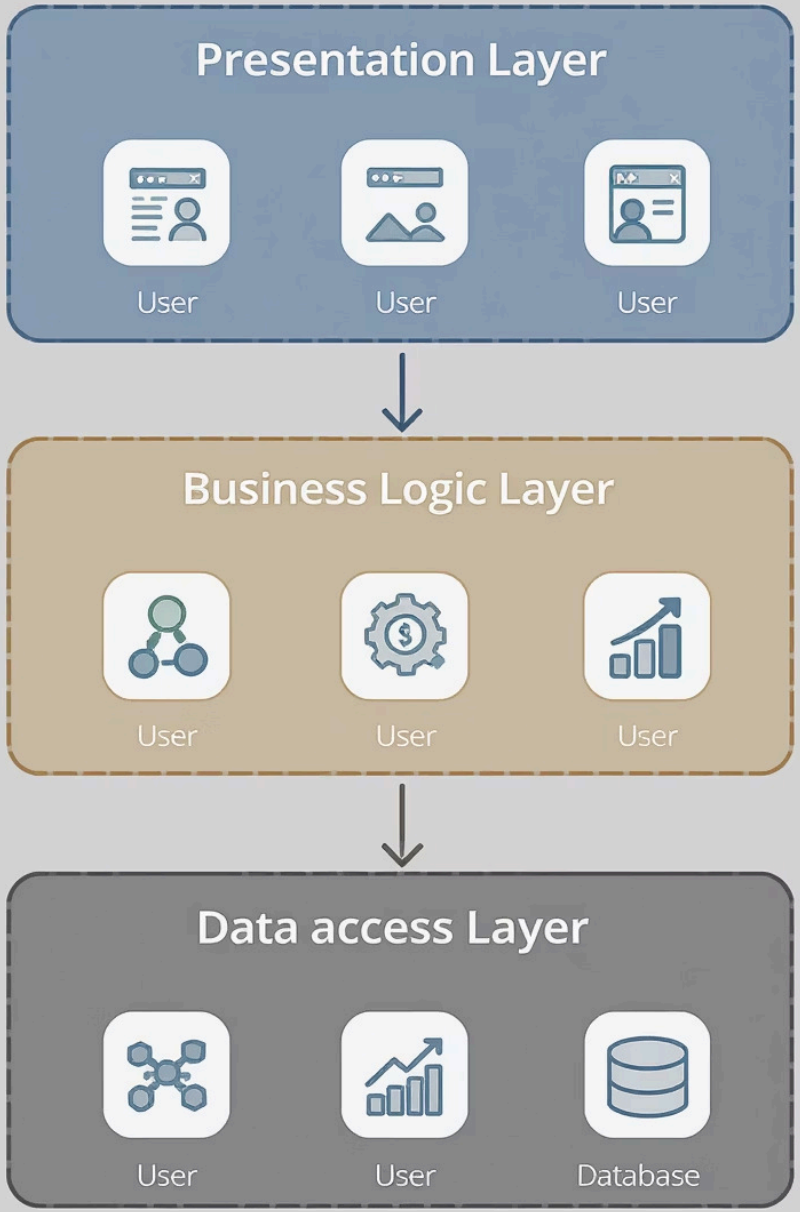
Interacción con usuarios o sistemas externos

Capa de negocio

Entidades del dominio, casos de uso y reglas del negocio

Capa de acceso a datos / infraestructura

Persistencia y comunicación con bases de datos.



Capa de Presentación

Es la parte más visible de la aplicación y actúa como punto de entrada al sistema. Es una capa dependiente del protocolo de comunicación

Interacción externa

Maneja las solicitudes de los usuarios o de otros sistemas, mediante peticiones sobre controladores REST, la vista en una app web, una tarea programada (schedule), comandos de una CLI, etc.)

Validación de entradas

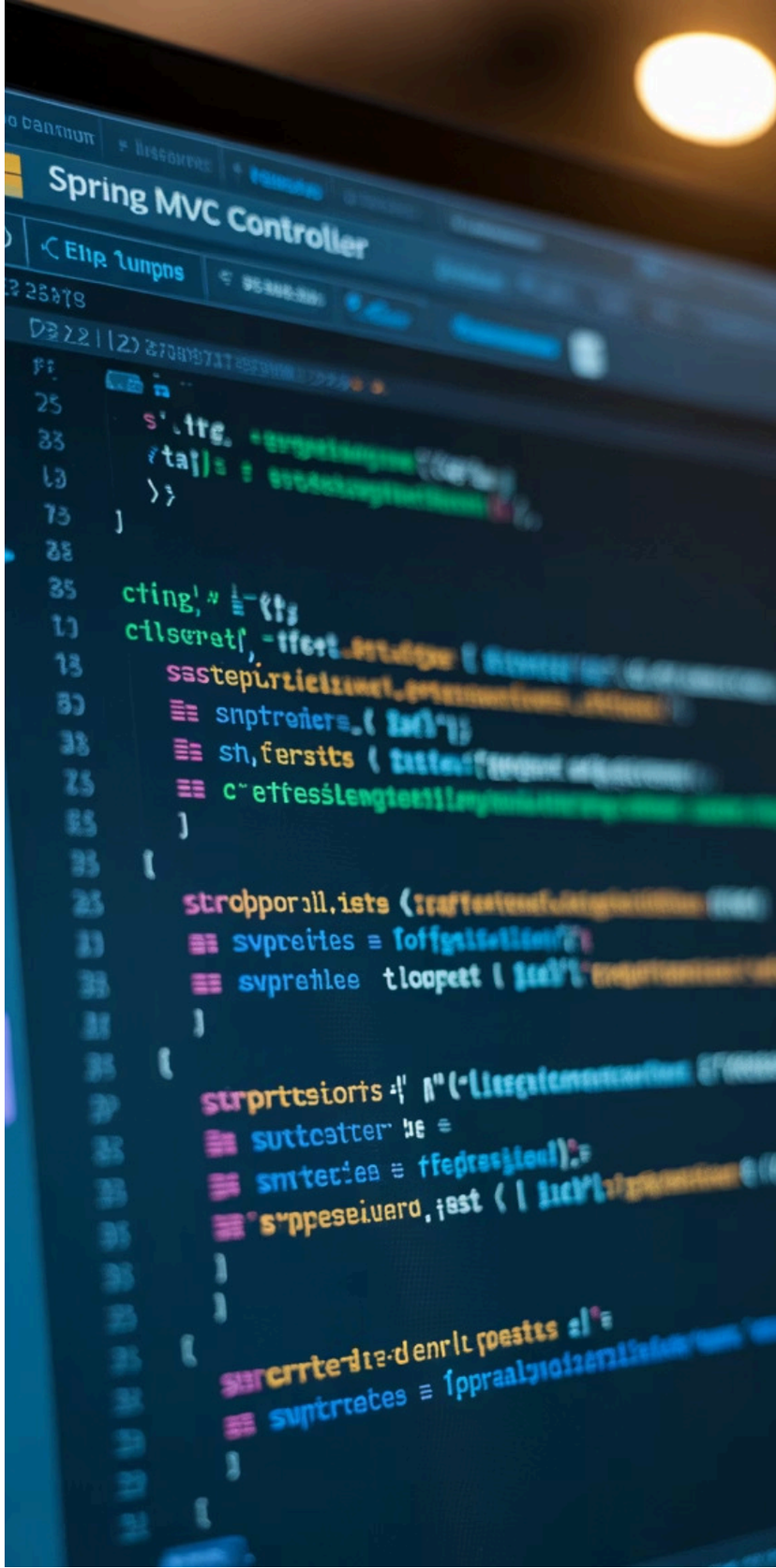
Verifica que los datos recibidos cumplan con el formato esperado.

Delegación

Delega la lógica de negocio a la siguiente capa. Los controladores **no** deben contener lógica de negocio, sino invocar métodos de la capa de servicio.

Formateo de respuestas

Devuelve respuestas en el formato adecuado (JSON/HTML). Maneja códigos de estado HTTP apropiados y estructuras de respuesta consistentes.



Capa de Negocio

La capa de negocio implementa el **core** de la lógica de negocio de la aplicación, actuando como intermediario entre la capa de presentación y la de datos.

Entidades

Clases que representan los dominios de información del sistema.

No deben ser clases anémicas (clases solo getters y setters), sino que deben incluir las validaciones e invariantes necesarias para mantener su integridad y exponer acciones propias del objeto para evitar inconsistencias.

Servicios

Clases que encapsulan toda la complejidad de la lógica de negocio exponiendo interfaces (input ports) simplificadas (fachadas).

Se trata de operaciones que afectan a varias entidades o que necesitan acceder a repositorios para persistir la información y mantener la consistencia del dominio.

Gestionan transacciones para asegurar la integridad de los datos cuando se realizan múltiples operaciones y realizar rollbacks automáticos en caso de excepciones.

También pueden implementar validaciones complejas, cálculos y operaciones de transformación de datos para respetar las reglas de negocio

Aseguran que todas las operaciones cumplan con los requisitos funcionales.

Arquitectura limpia

El core debería ser independiente de la tecnología de interfaz de usuario y del protocolo de comunicación utilizado.

No debe tener dependencias con ningún tipo de infraestructura externa: sistemas de bases de datos, sistemas de almacenamiento en bloques, sistemas de pago electrónico, colas de mensajeria, clientes de correo electrónico, plataformas de notificaciones, geolocalización, etc.

Los cambios en la infraestructura no afectan a esta capa, si creamos un nivel de indirección mediante interfaces (output ports) que nos aíslen de esos 'detalles'.

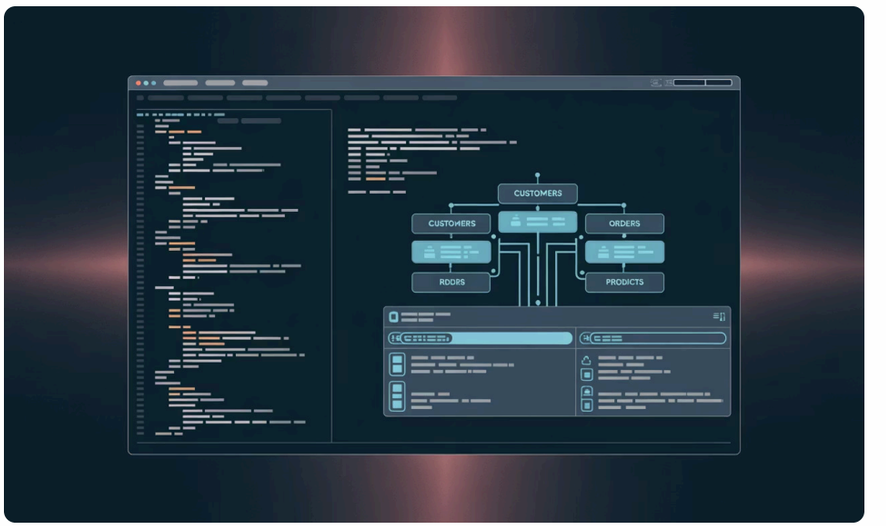
Facilita las pruebas unitarias mediante el uso de mocks o stubs de esas dependencias

Capa de Acceso a Datos (infraestructura)

Responsabilidades

La capa de acceso a datos ofrece las operaciones de persistencia, consultas, actualizaciones y eliminaciones de datos.

Proporciona abstracción sobre la tecnología de almacenamiento subyacente, ofreciendo una una interfaz uniforme para trabajar con diferentes fuentes de datos.



Active Record

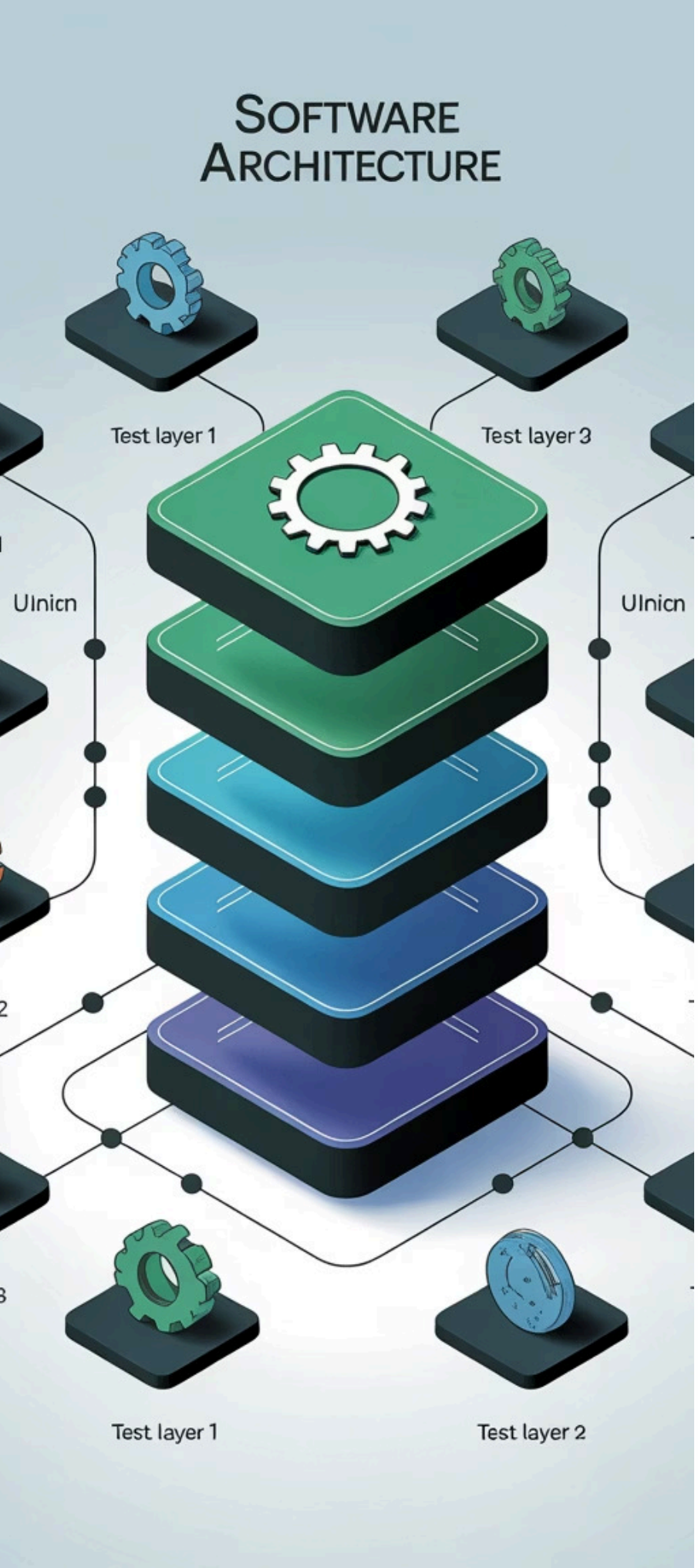
Cada objeto de dominio incluye métodos propios de persistencia (por ejemplo, `.save()` o `.delete()`), de modo que el modelo "se conoce a sí mismo" y sabe cómo guardarse en la base de datos.

Data Access Object (DAO)

Es una clase dedicada exclusivamente a encapsular las operaciones CRUD y las consultas SQL o de ORM para una entidad concreta, separando la lógica de acceso a datos del resto de la aplicación.

Repository

Similar al DAO, pero pudiendo trabajar con objetos agregados y usando métodos con nombres expresivos del dominio



Beneficios de la Arquitectura en Capas

Desacoplamiento

Permite **desacoplar** componentes para facilitar cambios. Las responsabilidades quedan claramente separadas, reduciendo las dependencias entre módulos y minimizando el impacto de modificaciones futuras en el sistema.

Intercambiabilidad

Puedes modificar una capa sin afectar a las demás. Esto facilita la actualización de tecnologías, implementación de nuevos requisitos o corrección de errores de forma aislada, manteniendo la estabilidad del sistema completo.

Testabilidad

Facilita pruebas unitarias de cada capa aislada. Es posible utilizar objetos mock o simulados para aislar la capa bajo prueba, aumentando la cobertura y fiabilidad de los tests mientras se reduce la complejidad de los mismos.

Especialización

Permite asignar equipos por especialidad. Los desarrolladores pueden concentrarse en áreas específicas según su experiencia: frontend, lógica de negocio o acceso a datos, optimizando la productividad y calidad del desarrollo.

Diseño de API REST

Buenas prácticas

¿Qué es una API?

Una API define cómo interactúan los componentes de software entre sí.

1. **API (Application Programming Interface):** Conjunto de funciones que un componente ofrece para ser utilizado por otro.
2. **API del sistema operativo:** Permiten desarrollar aplicaciones nativas aprovechando funcionalidades del SO.
3. **Tipos de API:** Invocaciones a procedimientos locales (interfaces de Java) o remotos (a través de un protocolo de comunicación de red).
4. **Servicios web:** Fundamentales en arquitecturas modernas distribuidas y microservicios. Utilizan HTTP. Proporciona independencia tecnológica entre el cliente y el servidor.
5. **REST:** Arquitectura que usa HTTP para comunicación entre clientes y servidores más común hoy día

Servucios REST

Significado

REpresentational **S**tate **T**ransfer es el estándar predominante para APIs web modernas.

Protocolo Web

Aprovecha HTTP, el mismo protocolo que usamos para navegar por internet.

Formato JSON

Utiliza JSON para transmitir datos, un formato ligero y fácil de procesar.

Ligereza

Significativamente más liviano que alternativas como SOAP, reduciendo la sobrecarga.

Alternativas tecnológicas a REST

SOAP

Protocolo basado en XML, ampliamente adoptado en entornos empresariales y sistemas legacy. Ofrece robustez pero con mayor sobrecarga.

gRPC

Desarrollado por Google, utiliza Protocol Buffers para serialización de datos. Destaca por su alta eficiencia y comunicación bidireccional.

Apache Thrift

Creado por Facebook, implementa formato binario para intercambio de datos. Optimizado para comunicaciones de alto rendimiento entre servicios.

Principios fundamentales de REST

Recursos identificados por URI

Toda la información se modela como recursos accesibles mediante identificadores únicos (URIs).

Comunicación sin estado

El servidor no almacena información sobre el cliente entre peticiones.

Métodos HTTP estándar

Se utilizan GET, POST, PUT, DELETE para interactuar con los recursos.

Cacheo de peticiones

Algunas respuestas pueden almacenarse en caché para mejorar el rendimiento.

Media Types

El formato de los datos (JSON, XML) se especifica mediante media types.

Recursos

1 **Datos**

Información principal del recurso que representa su contenido actual en el sistema. Los recursos no son estáticos.

Metadatos

Describen las características del recurso como tipo, formato o fecha de modificación.

Enlaces

Implementan HATEOAS permitiendo a clientes descubrir acciones disponibles y navegar entre recursos relacionados.

El estado de los recursos, en un instante dado, viene determinado por los datos, sus metadatos y los enlaces HATEOAS.

Verbos HTTP

Los verbos HTTP definen las operaciones fundamentales para interactuar con recursos en una API REST. Cada verbo tiene un propósito específico.

GET

Solicita representación de un recurso específico. Solo recupera datos sin modificar el servidor.

POST

Crea un nuevo recurso. Envía datos al servidor para procesarlos y almacenarlos.

PUT

Reemplaza completamente un recurso existente con la información proporcionada.

PATCH

Aplica modificaciones parciales a un recurso sin reemplazarlo completamente.

DELETE

Elimina el recurso especificado del servidor de forma permanente.

Invocaciones al API

Las APIs REST siguen un modelo de comunicación estructurado donde cada elemento tiene un propósito específico.

1. La **URL** identifica el recurso: http://myserver.com/api/v1/courses/1
2. El **método HTTP** (GET, POST, PUT, DELETE) define la operación a realizar
3. El **cuerpo de la petición** contiene los datos en formato JSON
4. Los **códigos de estado HTTP** indican el resultado: 2xx (éxito), 4xx (error cliente), 5xx (error servidor)

Este modelo estandarizado facilita la interoperabilidad entre sistemas distribuidos.

Ejemplo

Petición al API

Las peticiones especifican el recurso, método y formato de datos.

```
POST /blog/posts
Accept: application/json
Content-Type: application/json

{"title":"Hello World!",
"body":"This is my first post!"}
```

Respuesta del servidor

Spring MVC genera respuestas con códigos HTTP apropiados y datos estructurados.

```
HTTP/1.1 201 Created
Content-Type: application/json

{"id":"1","title":"Hello World!",
"body":"This is mí first post!"}
```

Spring MVC gestiona este flujo completo: recibe peticiones HTTP, las procesa mediante controladores y devuelve respuestas formateadas con los códigos de estado correspondientes.

Buenas prácticas en APIs REST (I)

La seguridad es un aspecto crítico en el diseño de APIs REST modernas. Implementarla correctamente protege datos y garantiza integridad del servicio.

Comunicaciones seguras

Implementa HTTPS para todas las conexiones. Utiliza certificados válidos y configura TLS correctamente.

Autorización granular

Define permisos específicos por recurso. Implementa. políticas de acceso RBAC (basado en roles), ACL (listas de control), etc.

Autenticación OAuth2

Permite delegación de acceso segura mediante tokens. Ideal para APIs con múltiples clientes y niveles de acceso.

API Keys

Solución simple para identificar aplicaciones cliente. Combínala con HTTPS y políticas de caducidad adecuadas.

Buenas prácticas en APIs REST (I)

Utilizar nombres de recursos, nunca verbos

Usa **/books** en lugar de ~~/getAllBooks~~. Los recursos deben ser sustantivos.

Usar nombres en plural

Utiliza **/books** en lugar de /book para representar colecciones de recursos.

Mantener GET como operación de solo lectura

GET jamás debe modificar el estado del servidor. Para cambios, usa PUT o PATCH.

Buenas prácticas en APIs REST (III)

Versionar tu API

Incluye el número de versión en la URL para facilitar su evolución y no romper clientes existentes. **/api/v1/books**.

Usar subrecursos para relaciones

Expresa jerarquías mediante URL anidadas como **/cars/711/drivers** para representar relaciones.

Cabeceras HTTP

Usa **Accept: application/json** para indicar el formato deseado en respuestas.

Especifica **Content-Type: application/json** para declarar el formato de envío.

Mantener consistencia

Sigue el mismo patrón de nombrado en toda la API para mejorar usabilidad.

Estas prácticas mejoran la interoperabilidad y evitan ambigüedades en la comunicación. Un API bien diseñado hace explícitas sus expectativas mediante convenciones estandarizadas.

Buenas prácticas en APIs REST (IV)

Los **códigos de estado** HTTP comunican claramente el resultado de una petición al cliente.

Código	Nombre	Significado
200	OK	Petición completada con éxito
201	Created	Recurso creado correctamente
204	No Content	Recurso eliminado sin contenido a devolver
304	Not Modified	Recurso no modificado, usar versión en caché
400	Bad Request	Petición incorrecta o malformada
401	Unauthorized	Falta autenticación para acceder al recurso
403	Forbidden	Cliente autenticado sin permisos suficientes
404	Not Found	Recurso solicitado no encontrado
500	Internal Server Error	Error en servidor. Debe evitarse en producción

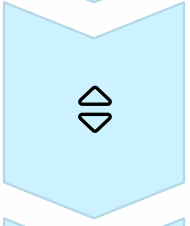
Un API REST debe usar estos códigos de forma consistente para mantener una comunicación efectiva con los clientes.

Buenas prácticas en APIs REST (V)

Los **parámetros de consulta** permiten a los clientes refinar sus peticiones API sin modificar el endpoint principal y optimizar consultas (existen otras alternativas para esto, como GraphQL)

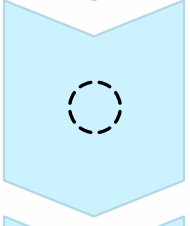
Filtrado

Usar parámetros específicos como **/cars?color=red** para obtener solo coches rojos.



Ordenación

Emplear prefijos **+/-** para indicar dirección: **/cars?sort=-manufacturer,+model**.



Selección

Permitir elegir campos específicos: **/cars?fields=manufacturer,model,id**.

Paginación

Implementar **offset/limit** o **page/size** para controlar resultados mostrados.

Estas técnicas mejoran el rendimiento y reducen la transferencia de datos innecesarios entre cliente y servidor.

Buenas prácticas en APIs REST (VI)

Usar **HATEOAS** (Hypermedia as the Engine of Application State) en los servicios REST permite descubrir las acciones disponibles en tiempo real.

Respuestas autoexplicativas

Cada respuesta incluye enlaces que indican qué acciones puede realizar el cliente a continuación.

Desacoplamiento

El cliente no necesita conocer la estructura del API de antemano.

Flexibilidad

Facilita la evolución del API sin romper clientes existentes.

Spring HATEOAS proporciona herramientas para implementar este principio de forma elegante.

Spring MVC - Objetivo

Aplicaciones y servicios web

Framework web basado en el patrón Modelo-Vista-Controlador para implementar servicios web REST y aplicaciones web (en combinación con algún motor de plantillas).

Separación de responsabilidades

Facilita la separación de responsabilidades: controladores gestionan peticiones, servicios procesan la lógica de negocio, vistas representan la presentación.

Spring MVC - Instalación

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Spring MVC - Componentes clave

DispatcherServlet

Punto de entrada para todas las peticiones HTTP.

Controllers

Clases con métodos anotados que manejan rutas (@Controller / @RestController).

Model

Objeto de datos que se envía a la vista.

ViewResolvers

Resuelven vistas (JSP, Thymeleaf, etc.).

@xxxMapping

Anotaciones para mapear Mapean URLs a métodos Java.

Spring MVC - Ejemplo (I)

La **clase** que representa el modelo de datos en nuestra aplicación Spring MVC:

```
public class Author {
    @Id
    private UUID id;          // Identificador único universal
    private String nif;       // Número de identificación fiscal
    private String name;      // Nombre del autor
    private String surname;   // Apellido del autor
    private String email;     // Correo electrónico de contacto

    // Constructor, getters, setters y otros métodos omitidos por brevedad
}
```

Spring MVC - Ejemplo (II)

El controlador que maneja las peticiones HTTP y coordina la respuesta

```
//La anotación @RestController combina @Controller y @ResponseBody para servicios REST.
@RestController
@RequestMapping("/api/v1/authors")  // Define la ruta base del recurso
public class AuthorController {

    // Almacenamiento en memoria de autores usando UUID como clave
    private final Map store = new ConcurrentHashMap<>();

    // Los métodos del controlador para GET, POST, PUT, DELETE
    // se implementarán aquí para gestionar las peticiones
}
```

Spring MVC - Ejemplo (III)

Implementación de los **endpoints GET** en nuestro controlador:

```
@GetMapping
public Collection<Author> listAll() {
    return store.values();
}



@GetMapping("/{id}")
@ResponseStatus(HttpStatus.OK)
public Author getById(@PathVariable UUID id) {
    Author author = store.get(id);
    if (author == null) {
        throw new ResponseStatusException(HttpStatus.NOT_FOUND, "No author " + id);
    }
    return author;
}
```

Spring MVC - Ejemplo (IV)

Implementación de los **endpoints POST** en nuestro controlador:

```
@PostMapping
public Author create(@RequestBody Author input) {
    validateInput(input);

    Author newAuthor = new Author(
            input.getNif().trim(),
            input.getName().trim(),
            input.getSurname().trim(),
            input.getEmail().trim()
    );

    store.put(newAuthor.getId(), newAuthor);

    return newAuthor;

}
```

Spring MVC - Ejemplo (V)

Implementación de los **endpoints POST** en nuestro controlador, utilizando un envoltorio **ResponseEntity** para proporcionar información adicional en la respuesta HTTP, como el código de estado y la ubicación del recurso creado.

```
@PostMapping
public ResponseEntity<Author> create(@RequestBody Author input) {
    validateInput(input);

    Author newAuthor = new Author(
            input.getNif().trim(),
            input.getName().trim(),
            input.getSurname().trim(),
            input.getEmail().trim()
    );

    store.put(newAuthor.getId(), newAuthor);

    URI location = URI.create("/api/v1/authors/" + newAuthor.getId());
    return ResponseEntity.created(location).body(newAuthor);
}
```

Spring MVC - Ejemplo (VI)

Implementación de los **endpoints PUT** en nuestro controlador

```
@PutMapping("/{id}")
@ResponseStatus(HttpStatus.OK)
public Author update(@PathVariable UUID id, @RequestBody Author input) {
    Author existingAuthor = store.get(id);
    if (existingAuthor == null) {
        throw new ResponseStatusException(HttpStatus.NOT_FOUND, "No author " + id);
    }
    validateInput(input);

    existingAuthor.setNif(input.getNif().trim());
    existingAuthor.setName(input.getName().trim());
    existingAuthor.setSurname(input.getSurname().trim());
    existingAuthor.setEmail(input.getEmail().trim());

    return existingAuthor;
}
```

Spring MVC - Ejemplo (VII)

Implementación de los **endpoints DELETE** en nuestro controlador

```
@DeleteMapping("/{id}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void delete(@PathVariable UUID id) {
    Author removed = store.remove(id);
    if (removed == null) {
        throw new ResponseStatusException(HttpStatus.NOT_FOUND, "No author " + id);
    }
}
```

Spring MVC - Ejemplo (VIII)

Implementación de un método auxiliar para **validar** los datos de entrada

```
private void validateInput(Author author) {
    if (author.getNif() == null || author.getNif().trim().isEmpty()) {
        throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "NIF cannot be empty");
    }
    if (author.getName() == null || author.getName().trim().isEmpty()) {
        throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "Name cannot be empty");
    }
    if (author.getSurname() == null || author.getSurname().trim().isEmpty()) {
        throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "Surname cannot be empty");
    }
    if (author.getEmail() == null || author.getEmail().trim().isEmpty()) {
        throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "Email cannot be empty");
    }
}
```

Spring HATEOAS - Objetivo

APIs REST con hipermedia —> Restful

Facilitar la creación de APIs REST enriquecidas con hipermedia (HATEOAS = Hypermedia As The Engine Of Application State).

Enlaces en recursos

Proporciona utilidades para incluir enlaces (links) en las representaciones de los recursos, guiando al cliente sobre las posibles acciones.

Spring HATEOAS - Instalación

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-hateoas</artifactId>
</dependency>
```

Spring HATEOAS - Componentes clave

EntityModel

Clase contenedora que encapsula un objeto de dominio individual y le añade enlaces de hipermedia (links). Permite transformar cualquier objeto en un recurso RESTful con capacidades HATEOAS.

CollectionModel

Envoltorio especializado para colecciones de recursos que agrega enlaces tanto a nivel de colección como para cada elemento individual. Facilita la navegación entre recursos relacionados.

Spring HATEOAS - Ejemplo (I)

Implementación de un endpoint que devuelve un recurso individual **enriquecido** con hipervínculos que facilitan la navegación entre recursos relacionados.

```
@GetMapping("/{id}")
@ResponseStatus(HttpStatus.OK)
public EntityModel<Book> getById(@PathVariable UUID id) {
    Book book = store.get(id);
    if (book == null) {
        throw new ResponseStatusException(HttpStatus.NOT_FOUND, "No book found with id " + id);
    }
    return assembler.toModel(book);
}
```

Spring HATEOAS - Ejemplo (II)

Implementación de un **ModelAssembler** para transformar entidades en recursos HATEOAS. Este componente enriquece las respuestas REST con enlaces hipermedia que facilitan la navegación entre recursos relacionados.

```
@Component
public class BookModelAssembler implements RepresentationModelAssembler<Book, EntityModel<Book>> {


    @Override
    public EntityModel<Book> toModel(Book book) {
      WebMvcLinkBuilder self = linkTo(methodOn(BookController.class).getById(book.getId()));

      EntityModel<Book> model = EntityModel.of(book);

        // Agrega link self al modelo
        model.add(self.withSelfRel());

// Agrega link al autor
model.add(linkTo(methodOn(AuthorController.class).getById(book.getAuthorId())).withRel("author").withType("GE
T"));

// Agrega link para editar el libro
model.add(linkTo(methodOn(BookController.class).update(book.getId(), null)).withRel("update").withType("PUT"));

        return model;
    }
}
```

Spring HATEOAS - Ejemplo (III)

Implementación de un endpoint que devuelve una lista de objetos con soporte HATEOAS.

```
@GetMapping
public CollectionModel<EntityModel<Book>> listAll() {
List<EntityModel<Book>> books = store.values().stream() //
    .map(assembler::toModel)
    .collect(Collectors.toList());

// Add links to the collection model
return CollectionModel.of(books, List.of(
   linkTo(methodOn(BookController.class).listAll()).withSelfRel(),
   linkTo(methodOn(BookController.class).create(null)).withRel("create").withType("POST"));
  }
}
```

Spring Data - Objetivo

Simplificación de acceso a datos

Simplificar la capa de acceso a datos mediante repositorios genéricos y abstracciones sobre distintos almacenes (relacionales y no relacionales).

Eliminación de código boilerplate

Elimina gran parte del código boilerplate de DAO, proporcionando interfaces con métodos predefinidos para CRUD y consultas basadas en nombres de métodos.

Spring Data - Módulos

Spring Data JPA

Simplifica el acceso a bases de datos relacionales con JPA

Spring Data R2DBC

Ideado para trabajar con drivers reactivos de bases de datos

Spring Data MongoDB

Para trabajar con bases de datos MongoDB

Spring Data Elasticsearch

Integración con el motor de búsqueda Elasticsearch

Spring Data Redis

Permite trabajar con esta bd en memoria

Spring Data for Apache Cassandra

Para su uso con esta base de datos de alto rendimiento

Mas integraciones en https://spring.io/projects/spring-data

Spring Data JPA - Características

Implementación para bases relacionales

Implementación concreta de Spring Data para JPA/Hibernate, que facilita el mapeo objeto-relacional en bases de datos relacionales SQL.

Repositorios automáticos

Provee repositorios automáticos, manejo de transacciones, y generación de consultas JPA sin necesidad de escribir las implementaciones.

Sincronización Automática

Spring Data mantiene la coherencia entre clases y tablas, configurando la propiedad hibernate.ddl-auto.

Generación de Esquema

Crea, actualiza o valida automáticamente las tablas según las definiciones de entidades, no requiriendo SQL adicional.

Spring Data JPA - Instalación

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<!-- Driver de la BD --<
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <scope>runtime</scope>
</dependency>
```

Spring Data JPA - Repositorios

Repository

Interfaz marcador vacía que define el tipo de dominio (T) y el tipo de identificador (ID). No aporta métodos por sí misma.

CrudRepository

Operaciones CRUD básicas: save(), findById(), findAll(), delete().

PagingAndSortingRepository

Añade paginación y ordenación.

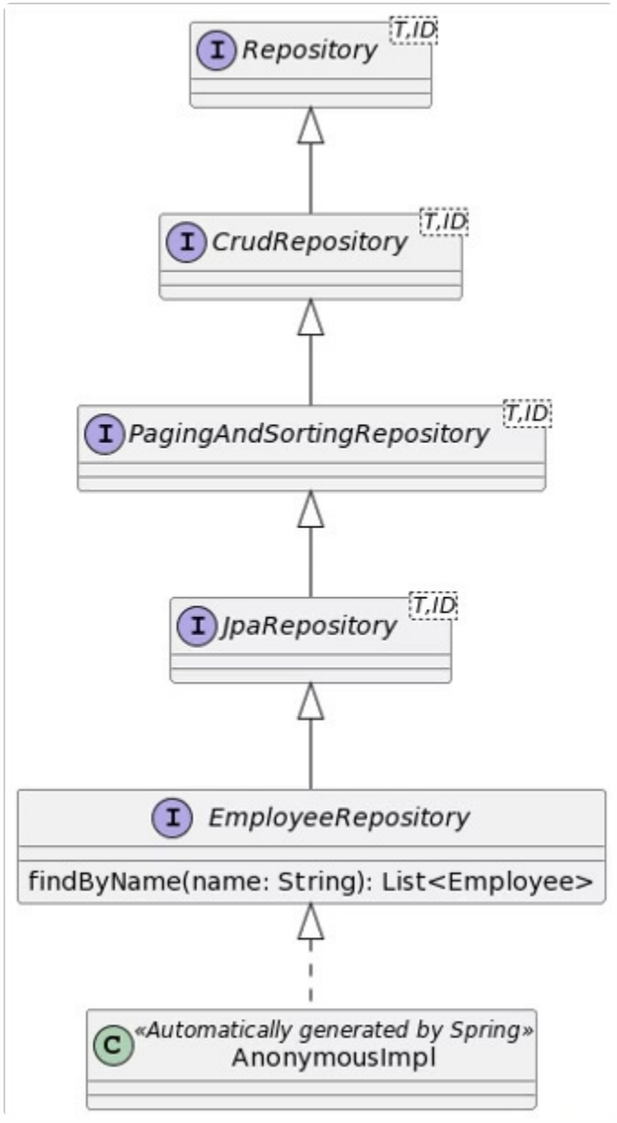
JpaRepository

Añade métodos para persistir en lote (e.j: saveAll) y sincronizar (flush)

«YourRepository»

Indicando la signatura de los métodos de consulta (y siguiendo una determinada nomenclatura), Spring puede implementar dichos métodos de acceso a datos

Soporte para consultas personalizadas en SQL o JPQL con @Query



JPA - anotaciones principales

Las anotaciones JPA permiten definir el mapeo objeto-relacional en Spring Data JPA.

@Entity

Marca una clase como entidad persistente, asociándola automáticamente con una tabla.

@Table

Personaliza el nombre de la tabla y otros atributos como schema e índices.

@Id

Define el campo que será la clave primaria de la entidad.

@Column

Personaliza la columna correspondiente: nombre, longitud, restricciones de nulidad, unicidad.

@GeneratedValue

Configura la estrategia de generación de valores para claves primarias.

@OneToMany

Establece una relación uno-a-muchos entre entidades (un autor tiene muchos libros).

@ManyToOne

Define una relación muchos-a-uno entre entidades (muchos libros pertenecen a un autor).

@ManyToMany

Configura una relación muchos-a-muchos entre entidades (estudiantes y cursos).

@OneToOne

Establece una relación uno-a-uno entre entidades (usuario y perfil).

Spring Data JPA - Ejemplo (I)

El archivo **application.properties** permite configurar la conexión a la base de datos y el comportamiento de JPA:

```
spring.datasource.url=jdbc:mysql://localhost:3306/cursoJava
spring.datasource.username=cursoJava
spring.datasource.password=cursoJava
spring.jpa.hibernate.ddl-auto= create-drop | create | update | validate
spring.jpa.show-sql=true
```

Donde **hibernate.ddl-auto** puede ser:

- **create-drop:** Crea el esquema al inicio y lo elimina al finalizar (ideal para pruebas)
- **create:** Crea el esquema eliminando datos existentes
- **update:** Actualiza el esquema preservando datos existentes
- **validate:** Solo valida que el esquema coincida con las entidades

Spring Data JPA - Ejemplo (II)

Las **entidades JPA** son clases Java que representan tablas en la base de datos. Habitualmente, cada instancia de una entidad corresponde a una fila en la tabla.

```
@Entity
@Table(name = "COURSES")
public class Course {
    @Id
    @GeneratedValue
    @JdbcTypeCode(SqlTypes.VARCHAR)
    private UUID id;

    @Column(unique = true)
    private String code;

    private String title;

    private int credits;

    // getters y setters
}
```

La anotación @Entity marca la clase como una entidad JPA, mientras que @Table especifica el nombre de la tabla en la base de datos.

Spring Data JPA - Ejemplo (III)

La interfaces de **Repositorio** permite definir operaciones de acceso a datos sin implementar código. Spring Data JPA genera automáticamente las implementaciones en tiempo de ejecución basándose en las convenciones de nomenclatura de los métodos.

```
public interface CourseRepository extends JpaRepository<Course, UUID> {

    // Heredamos save, findAll, findById, count, etc.

    List<Course> findByTitleContainingIgnoreCase(String title);

    List<Course> findByCreditsGreaterThan(int credits);

}
```

Spring Data JPA - Ejemplo (IV)

La clase de **Servicio** permite implementar la lógica de negocio y actúa como intermediario entre el controlador y el repositorio.

```
@Service
public class CourseService {

    private final CourseRepository courseRepository;

    public CourseService(CourseRepository courseRepository) {
        this.courseRepository = courseRepository;
    }

    public Course create(Course course) {
        return courseRepository.save(course);
    }

    public List<Course> readAll() {
        return courseRepository.findAll();
    }

    public List<Course> readLongCourses() {
        return courseRepository.findByCreditsGreaterThan(5);
    }


    public Optional<Course> readOne(UUID id) {
        return courseRepository.findById(id);
    }


    public Course update(Course course) {
        return courseRepository.save(course);
    }


    public void delete(UUID id) {
        courseRepository.deleteById(id);
    }
}
```

Spring Data JPA - Ejemplo (V)

El **Controlador REST** actúa como punto de entrada para las peticiones HTTP, definiendo los endpoints de la API y delegando las operaciones al servicio.

```
@RestController
@RequestMapping("/courses")
public class CourseController {

    private final CourseService courseService;

    public CourseController(CourseService courseService) {
        this.courseService = courseService;
    }

    @GetMapping
    public List<Course> getAllCourses() {
        return courseService.readAll();
    }

    @GetMapping("/{id}")
    public ResponseEntity<Course> getCourseById(@PathVariable UUID id) {
        Optional<Course> course = courseService.readOne(id);
        if (course.isEmpty()) {
            return ResponseEntity.notFound().build();
        }
        return ResponseEntity.ok(course.get());
    }

    @PostMapping
    public Course createCourse(@RequestBody Course course) {
        return courseService.create(course);
    }

    @PutMapping("/{id}")
    public ResponseEntity<Course> updateCourse(@PathVariable UUID id, @RequestBody Course courseDetails) {
        Optional<Course> course = courseService.readOne(id);
        if (course.isEmpty()) {
            return ResponseEntity.notFound().build();
        }

        course.get().setCode(courseDetails.getCode());
        course.get().setTitle(courseDetails.getTitle());
        course.get().setCredits(courseDetails.getCredits());

        Course updatedCourse = courseService.update(course.get());
        return ResponseEntity.ok(updatedCourse);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteCourse(@PathVariable UUID id) {
        Optional<Course> course = courseService.readOne(id);
        if (course.isEmpty()) {
            return ResponseEntity.notFound().build();
        }

        courseService.delete(id);
        return ResponseEntity.noContent().build();
    }
}
```

Spring Data REST - Objetivo

Exposición automática de repositorios

Exponer repositorios Spring Data automáticamente como endpoints REST sin necesidad de escribir controladores personalizados.

Recursos RESTful con HATEOAS

Convierte entidades gestionadas por Repository en recursos RESTful, incluyendo soporte para HATEOAS y operaciones CRUD.

Spring Data REST - Componentes clave

@RepositoryRestResource

Anota repositorios para personalizar rutas o relaciones.

Proyecciones (@Projection)

Permiten definir vistas parciales de entidades (incluir/ocultar campos).

Spring Data REST - Ejemplo (I)

Configuramos application.properties

```
spring.data.rest.detection-strategy=annotated
spring.data.rest.basePath=/api
```

Spring Data REST - Ejemplo (II)

```
@RepositoryRestResource(collectionResourceRel = "students", path = "students")

public interface StudentRepository extends JpaRepository<Student, Long> {

    // Ejemplo de método de búsqueda personalizado:
    List<Student> findByLastNameIgnoreCase(String lastName);
}
```

Resumen

A lo largo de esta presentación, hemos explorado el desarrollo de backends con Spring, centrándonos en sus potentes módulos.

Desde la arquitectura en capas hasta las implementaciones prácticas con Spring MVC, HATEOAS, Data JPA y Data REST.

Arquitectura en Capas

Separación clara de responsabilidades para aplicaciones robustas y mantenibles.

API REST

Diseño de interfaces escalables siguiendo principios REST y buenas prácticas.

Gestión de Datos

Simplificación del acceso a datos con repositorios y exposición automática REST.