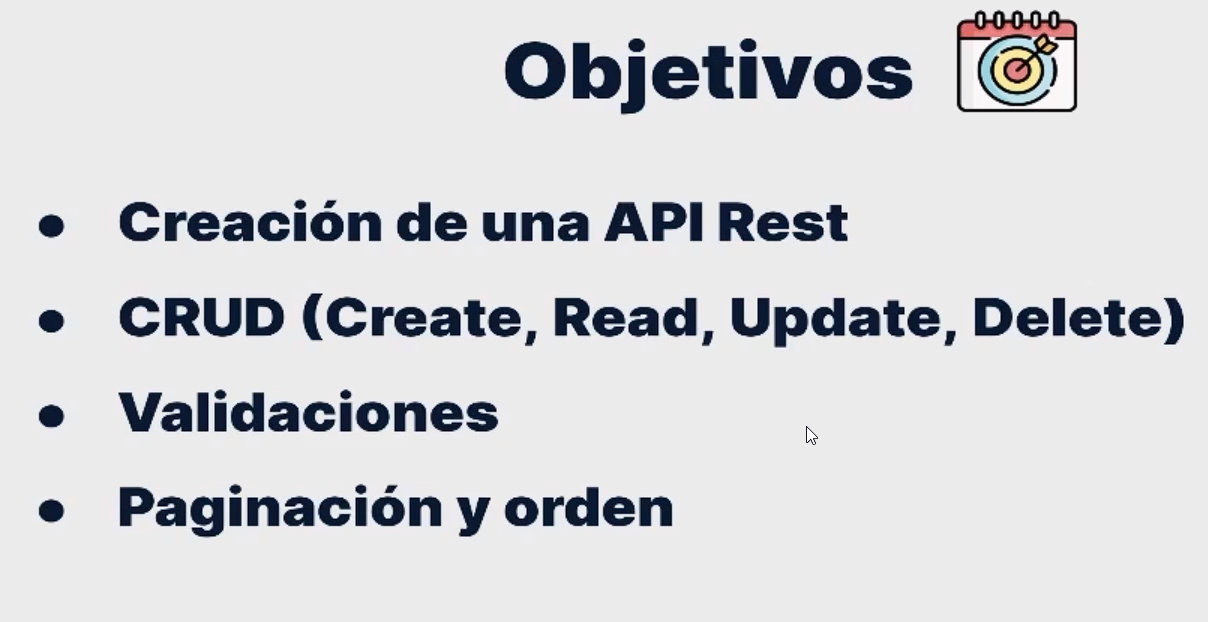
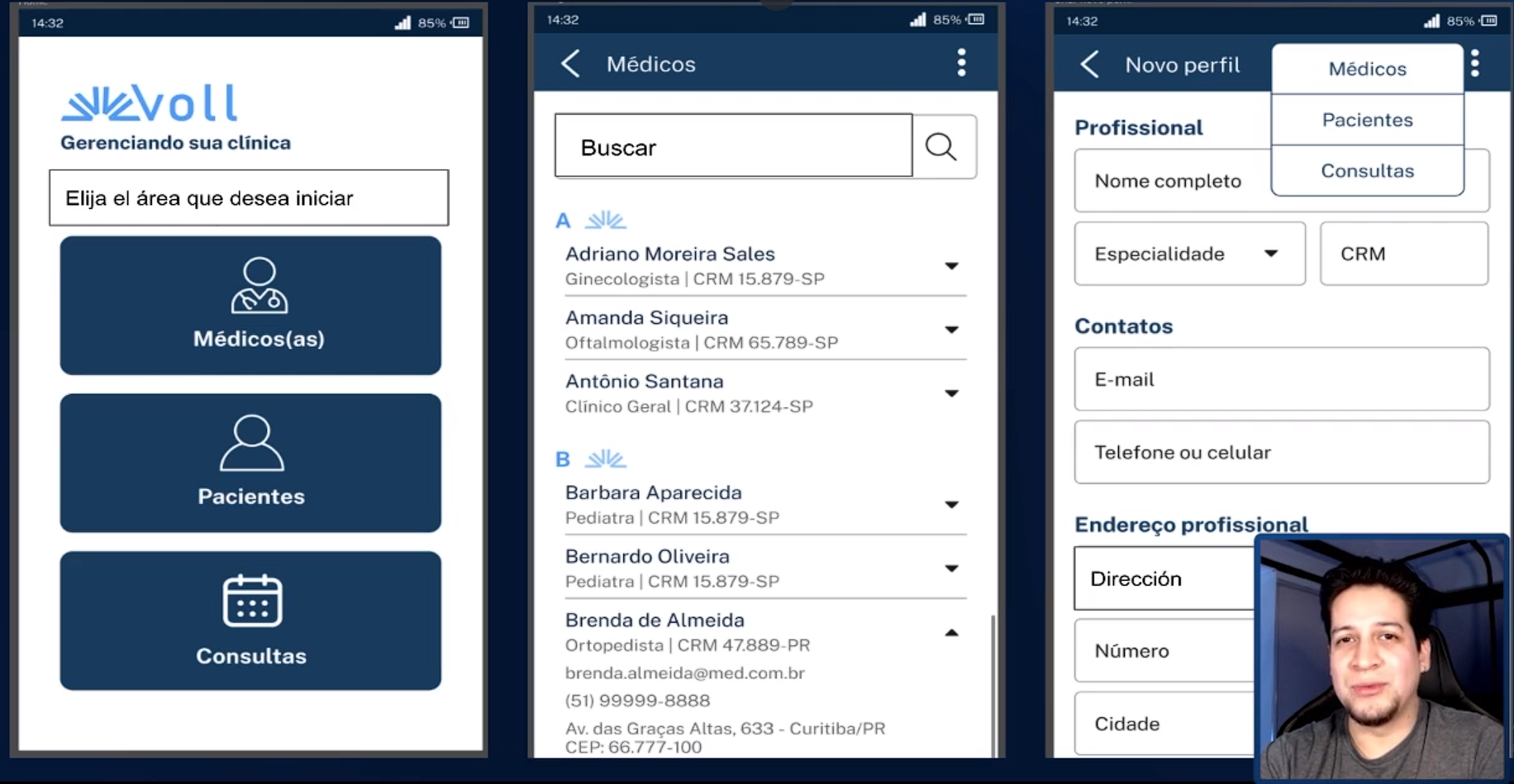
urso de  
Spring Boot 3: desarrollar una API Rest en Java

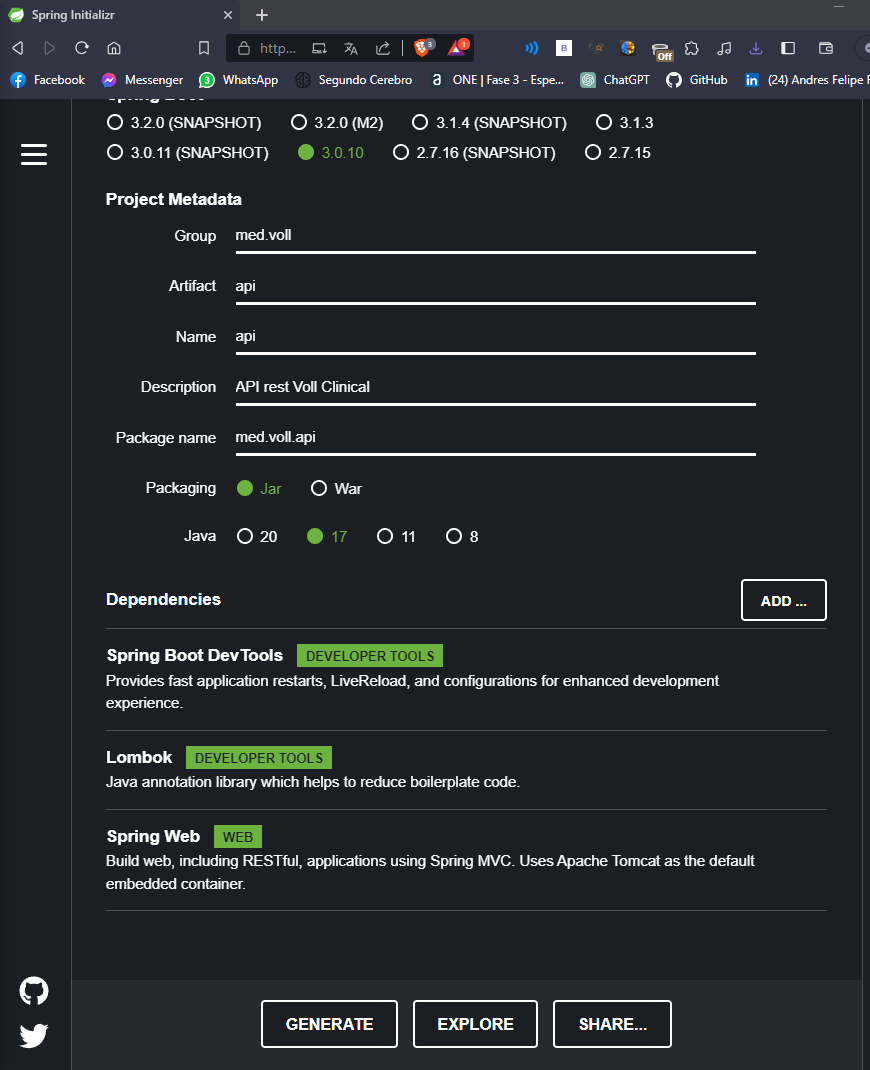
**Creación del Proyecto:**







Generando Proyecto:



Estamos utilizando Java 17 en este curso, sin embargo, generalmente Oracle sugiere instalar la versión más actual y no tiene problema usar la versión más reciente pues así evita los problemas de compatibilidad.

Además, puede instalar una versión Java y utilizar otra versión en el IDE de su máquina. Vamos a mostrar dos ejemplos de cómo cambiar la versión de Java en un proyecto.

**Un ejemplo con Intellij IDEA:**

Seleccione la opción 'File' en el menú principal.

Seleccione 'Project Structure'.

En 'Project Settings' en la parte 'Project' seleccione la opción deseada de Java en 'SDK' Y seleccione Ok.

Recuerde que es posible añadir nuevas versiones y utilizar como desee

Un ejemplo con Eclipse:

Seleccione el proyecto con el botón derecho y seleccione la opción ‘Properties’.

Seleccione la sección ‘Java Compiler’ y desmarque la opción de la parte ‘JDK Compliance’.

Después es posible cambiar la versión de Java en la parte ‘Compiler compliance level’.

Seleccione la sección ‘**Java Compiler**'y desmarque la opción de la parte ‘J**DK Compliance**’.- Después es posible cambiar la versión de Java en la parte ‘Compiler compliance level’.

Spring y Spring Boot no son lo mismo con diferentes nombres.

**Spring** es un **framework** para desarrollar aplicaciones en Java, creado a mediados de 2002 por Rod Johnson, que se ha vuelto muy popular y adoptado en todo el mundo debido a su simplicidad y facilidad de integración con otras tecnologías.

El framework se desarrolló de forma **modular**, en el que cada recurso que proporciona está representado por un módulo, que se puede agregar a una aplicación según sea necesario. Con esto, en cada aplicación podemos agregar solo los módulos que tengan sentido, haciéndola así más liviana. Hay varios módulos en Spring, cada uno con un propósito diferente, tales como: el módulo **MVC**, para desarrollar aplicaciones Web y API's Rest; el módulo de **Security**, para manejar el control de autenticación y autorización de las aplicaciones; y el módulo **Transactions**, para gestionar el control transaccional.

Sin embargo, uno de los mayores problemas de las aplicaciones que usaban Spring era la parte de configuración de sus módulos, que se hacía íntegramente con archivos XML, y después de unos años el framework también comenzó a soportar configuraciones a través de clases Java, utilizando principalmente anotaciones. En ambos casos, dependiendo del tamaño y complejidad de la aplicación, así como de la cantidad de módulos Spring utilizados en ella, dichas configuraciones eran bastante extensas y difíciles de mantener.

Además, iniciar un nuevo proyecto con Spring era una tarea bastante complicada, debido a la necesidad de realizar este tipo de configuraciones en el proyecto.

Precisamente para solventar tales dificultades, a mediados de 2014 se creó un nuevo módulo Spring, denominado **Boot**, con el objetivo de agilizar la creación de un proyecto que utilice Spring como framework, así como simplificar las configuraciones de sus módulos.

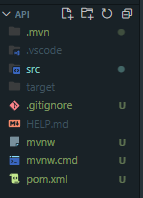
El lanzamiento de Spring Boot fue un hito para el desarrollo de aplicaciones Java, ya que hizo más simple y ágil esta tarea, facilitando mucho la vida de las personas que utilizan el lenguaje Java para desarrollar sus aplicaciones.

A lo largo del curso, aprenderemos a desarrollar una aplicación usando Spring Boot, junto con varios otros módulos de Spring, de una manera simple y productiva.

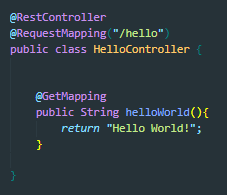
La versión 3 de Spring Boot se lanzó en noviembre de 2022 y trae algunas características nuevas en comparación con la versión anterior. Entre las principales novedades se encuentran:

* Compatibilidad con Java 17
* Migración de especificaciones Java EE a Jakarta EE
* Compatibilidad con imágenes nativas
* Puede ver la lista completa de las novedades de Spring Boot versión 3 en el sitio web: Spring Boot 3.0 Release Notes

Estructura del Proyecto:



Hello World:





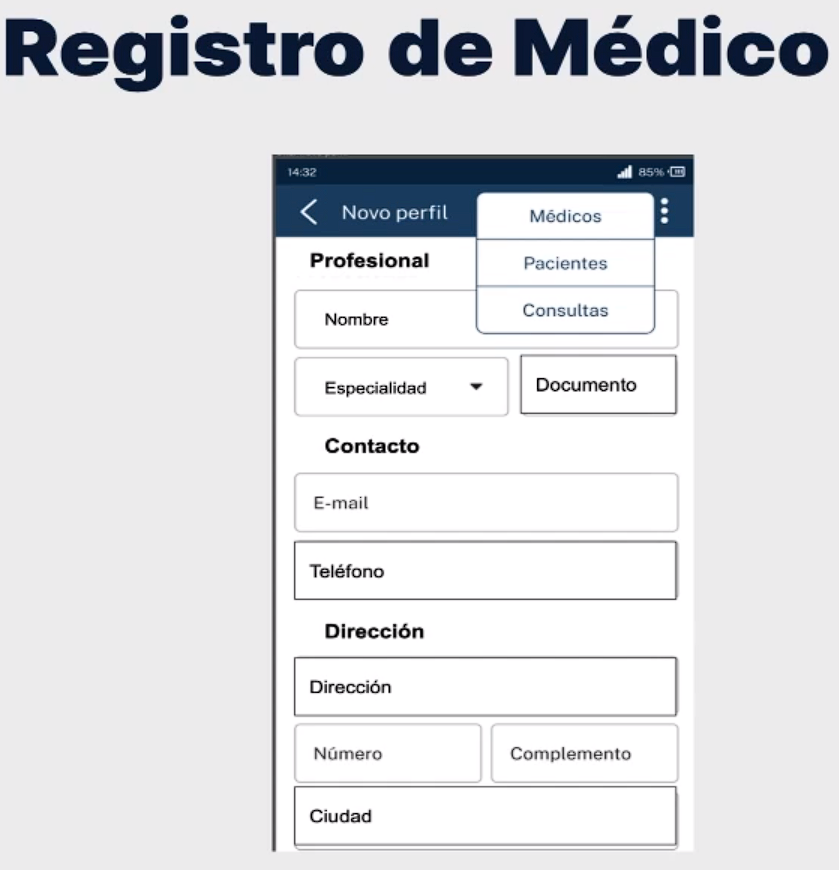
Lo que Aprendimos:

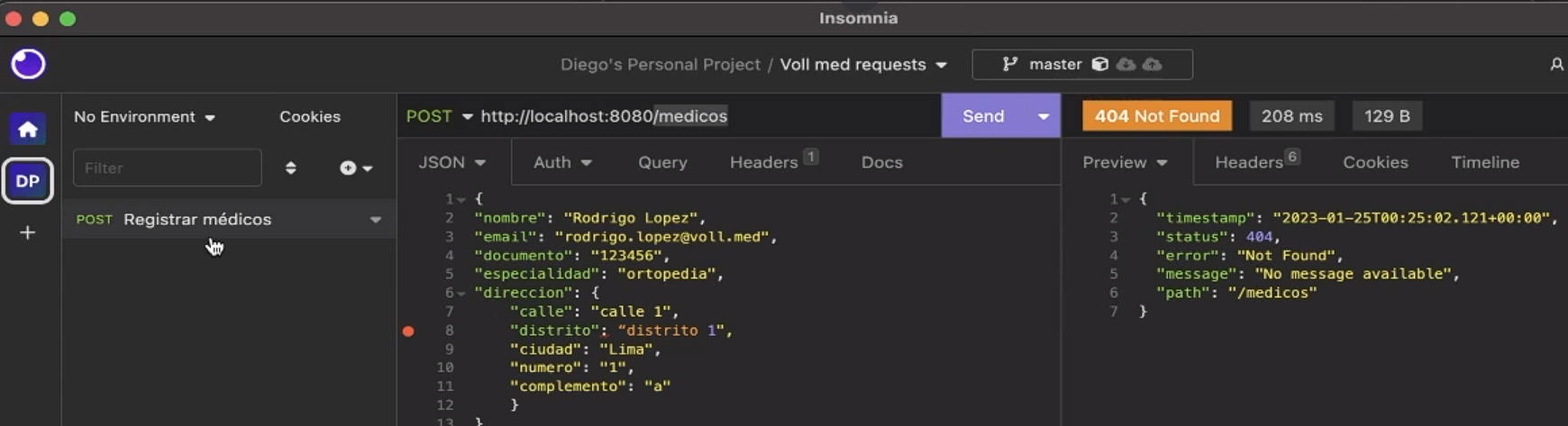
**En esta clase, aprendiste a:**

* Crear un proyecto Spring Boot usando el sitio web Spring Initializr;
* Importar el proyecto a IntelliJ y ejecutar una aplicación Spring Boot a través de la clase que contiene el método main;
* Crear una clase Controller y mapear una URL en él usando las anotaciones @RestController y @RequestMapping;
* Realizar una solicitud de prueba en el navegador accediendo a la URL mapeada en el Controller.

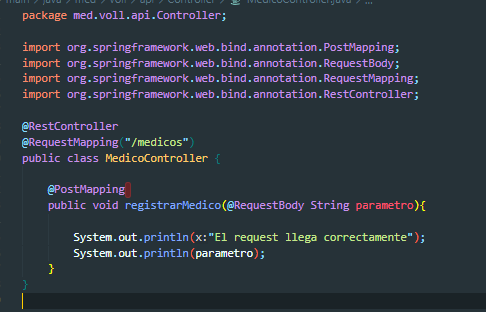
**Request POST:**

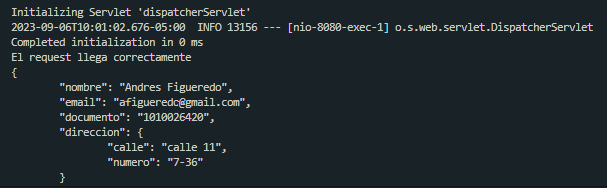
Enviando datos:

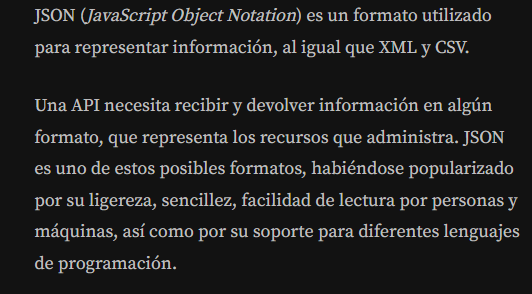


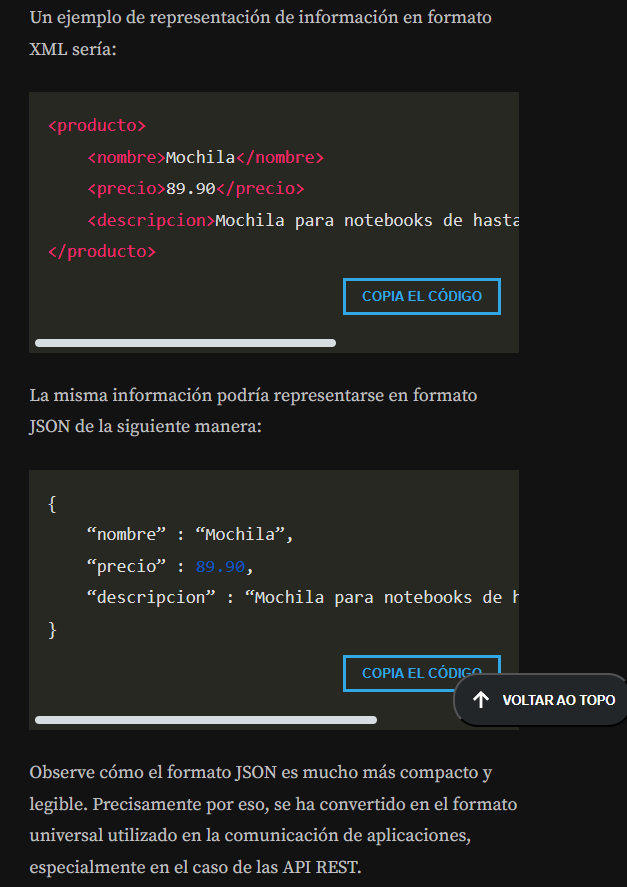


Recibiendo datos:



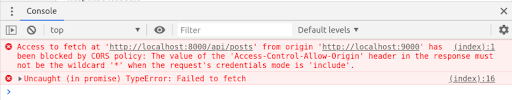






CORS:

Cuando desarrollamos una API y queremos que todos sus recursos estén disponibles para cualquier cliente HTTP, una de las cosas que nos viene a la mente es CORS (*Cross-Origin Resource Sharing*), en Español, “Intercambio de recursos con diferentes orígenes” Si aún no te ha pasado, no te preocupes, es normal tener errores de CORS al consumir y poner a disposición las APIs.

Pero al fin y al cabo, ¿qué es CORS, qué provoca errores y cómo evitarlos en nuestras APIs con Spring Boot?

**CORS**

CORS es un mecanismo utilizado para agregar encabezados HTTP que le indican a los navegadores que permitan que una aplicación web se ejecute en un origen y acceda a los recursos desde un origen diferente. Este tipo de acción se denomina *cross-origin HTTP request*. En la práctica, les dice a los navegadores si se puede acceder o no a un recurso en particular.

Pero, ¿por qué ocurren los errores? ¡Es hora de entender!

**Same-origin policy**

Por defecto, una aplicación Front-end, escrita en JavaScript, solo puede acceder a los recursos ubicados en el mismo origen de la solicitud. Esto sucede debido a la política del mismo origen (*same-origin policy*), que es un mecanismo de seguridad de los navegadores que restringe la forma en que un documento o script de un origen interactúa con los recursos de otro. Esta política tiene como objetivo detener los ataques maliciosos.

Dos URL comparten el mismo origen si el protocolo, el puerto (si se especifica) y el host son los mismos. Comparemos posibles variaciones considerando la URL https://cursos.alura.com.br/category/programacao:

| **URL** | **Resultado** | **Motivo** |
| --- | --- | --- |
| <https://cursos.alura.com.br/category/front-end> | Mismo origen | Solo camino diferente |
| <http://cursos.alura.com.br/category/programacao> | Error de CORS | Protocolo diferente (http) |
| <https://faculdade.alura.com.br:80/category/programacao> | Error de CORS | Host diferente |

Ahora, la pregunta sigue siendo: ¿qué hacer cuando necesitamos consumir una API con una URL diferente sin tener problemas con CORS? Como, por ejemplo, cuando queremos consumir una API que se ejecuta en el puerto 8000 desde una aplicación React que se ejecuta en el puerto 3000. ¡Compruébalo!

Al enviar una solicitud a una API de origen diferente, la API debe devolver un header llamado **Access-Control-Allow-Origin**. Dentro de ella es necesario informar los diferentes orígenes que serán permitidas de consumir la API, en nuestro caso: Access-Control-Allow-Origin: http://localhost:3000.

Puede permitir el acceso desde cualquier origen utilizando el símbolo \* (asterisco): Access-Control-Allow-Origin: \*. Pero esta no es una medida recomendada, ya que permite que fuentes desconocidas accedan al servidor, a menos que sea intencional, como en el caso de una API pública. Ahora veamos cómo hacer esto en Spring Boot correctamente.

**Habilitación de diferentes orígenes en Spring Boot**

Para configurar el CORS y permitir que un origen específico consuma la API, simplemente cree una clase de configuración como la siguiente:

@Configuration

**public** **class** **CorsConfiguration** **implements** **WebMvcConfigurer** {

@Override

**public** **void** **addCorsMappings**(CorsRegistry registry) {

registry.addMapping("/\*\*")

.allowedOrigins("http://localhost:3000")

.allowedMethods("GET", "POST", "PUT", "DELETE", "OPTIONS", "HEAD", "TRACE", "CONNECT");

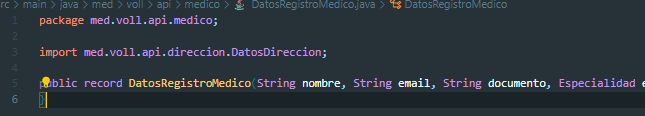
}

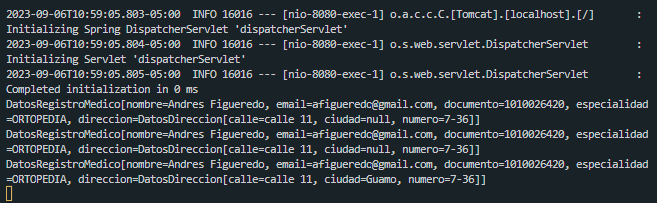
}

COPIA EL CÓDIGO

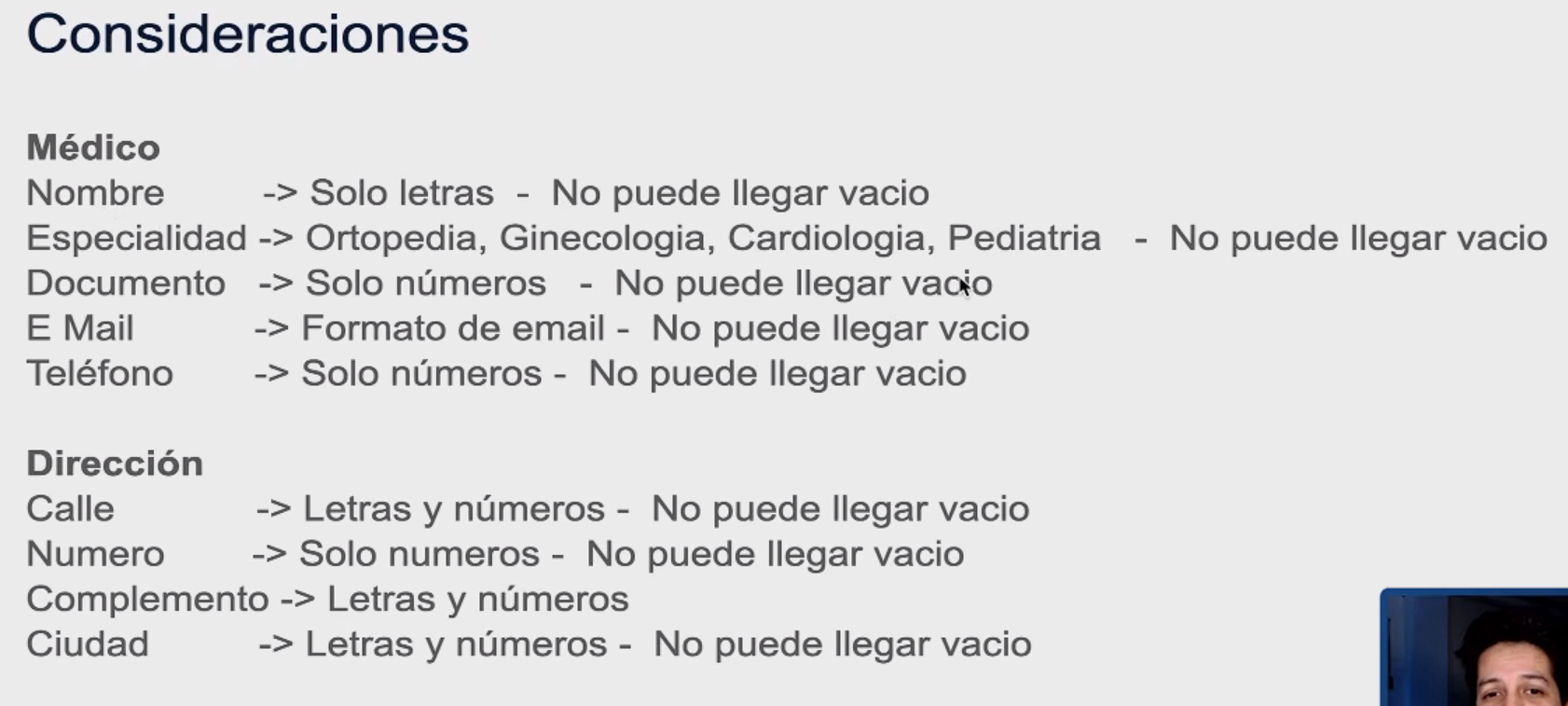
[**http://localhost:3000**](http://localhost:3000/) sería la dirección de la aplicación Front-end y **.allowedMethods** los métodos que se permitirán ejecutar. Con esto, podrás consumir tu API sin problemas desde una aplicación front-end.

DTO java Record:





DATA TRANSFER OBJECT



Lanzado oficialmente en Java 16, pero disponible experimentalmente desde Java 14. **Record** es un recurso que le permite representar una clase inmutable, que contiene solo atributos, constructor y métodos de lectura, de una manera muy simple y ágil.

Este tipo de clase encaja perfectamente para representar clases DTO, ya que su objetivo es únicamente representar datos que serán recibidos o devueltos por la API, sin ningún tipo de comportamiento.

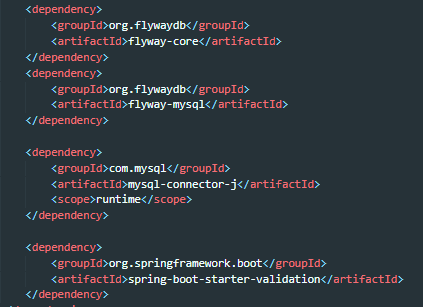
Lo que Aprendimos:

**En esta clase, aprendiste a:**

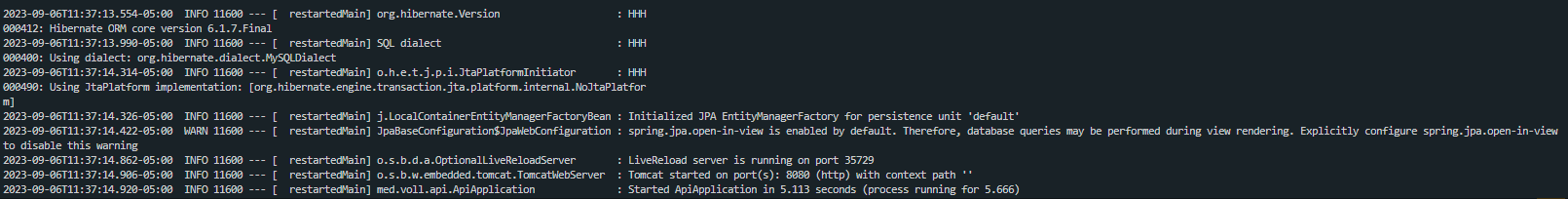
* Mapear solicitudes POST en una clase Controller;
* Enviar solicitudes POST a la API usando Insomnia;
* Enviar datos a la API en formato JSON;
* Utilizar la anotación @RequestBody para recibir datos del cuerpo de la solicitud en un parámetro en el Controller;
* Use el padrón **DTO (*Data Transfer Object*)**, a través de Java Records, para representar los datos recibidos en una solicitud POST.

**Spring Data JPA:**

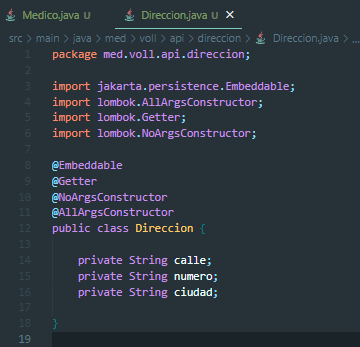
Agregando dependencias:

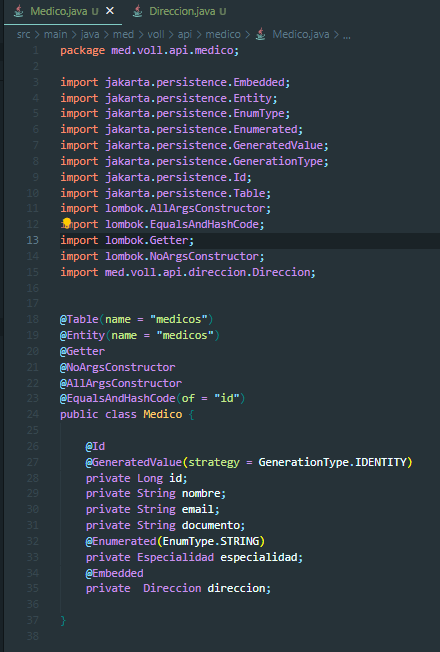




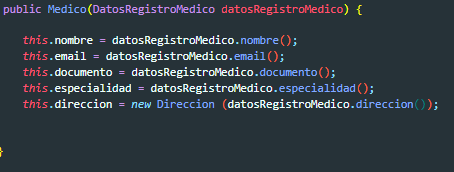


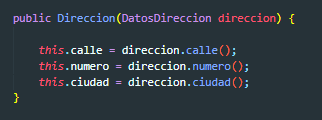
Entidades JPA:

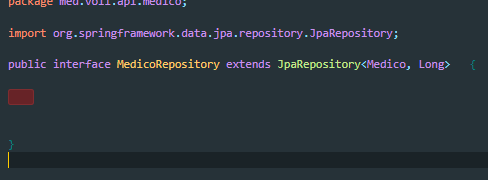


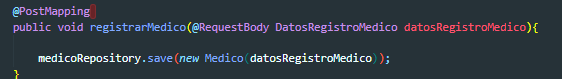


Interfaces y repositorio:









En algunos proyectos Java, dependiendo de la tecnología elegida, es común encontrar clases que siguen el patrón **DAO**, usado para aislar el acceso a los datos. Sin embargo, en este curso usaremos otro patrón, conocido como **Repositorio**.

Pero entonces pueden surgir algunas preguntas: ¿cuál es la diferencia entre los dos enfoques y por qué esta elección?

**Patrón DAO**

El patrón de diseño DAO, también conocido como **Data Access Object**, se utiliza para la persistencia de datos, donde su objetivo principal es separar las reglas de negocio de las reglas de acceso a la base de datos. En las clases que siguen este patrón, aislamos todos los códigos que se ocupan de conexiones, comandos SQL y funciones directas a la base de datos, para que dichos códigos no se esparzan a otras partes de la aplicación, algo que puede dificultar el mantenimiento del código y también el intercambio de tecnologías y del mecanismo de persistencia.

El repositorio es un mecanismo para encapsular el almacenamiento, recuperación y comportamiento de búsqueda, que emula una colección de objetos.

En pocas palabras, un repositorio también maneja datos y oculta consultas similares a DAO. Sin embargo, se encuentra en un nivel más alto, más cerca de la lógica de negocio de una aplicación. Un repositorio está vinculado a la regla de negocio de la aplicación y está asociado con el agregado de sus objetos de negocio, devolviéndolos cuando es necesario.

Pero debemos estar atentos, porque al igual que en el patrón DAO, las reglas de negocio que están involucradas con el procesamiento de información no deben estar presentes en los repositorios. Los repositorios no deben tener la responsabilidad de tomar decisiones, aplicar algoritmos de transformación de datos o brindar servicios directamente a otras capas o módulos de la aplicación. Mapear entidades de dominio y proporcionar funcionalidades de aplicación son responsabilidades muy diferentes.

Un repositorio se encuentra entre las reglas de negocio y la capa de persistencia:

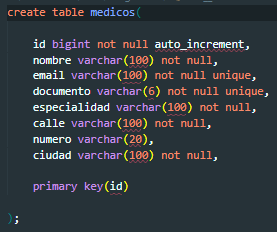
1. Proporciona una interfaz para las reglas comerciales donde se accede a los objetos como una colección;
2. Utiliza la capa de persistencia para escribir y recuperar datos necesarios para persistir y recuperar objetos de negocio.

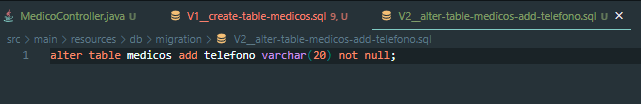
Por lo tanto, incluso es posible utilizar uno o más DAOs en un repositorio.

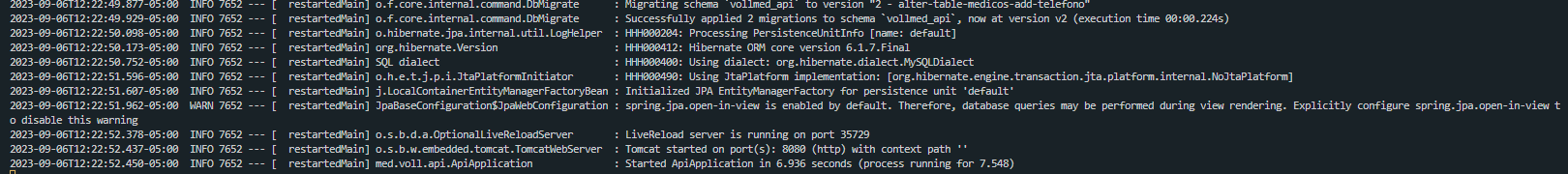
**¿Por qué el padrón repositorio en lugar de DAO usando Spring?**

El patrón de repositorio fomenta un diseño orientado al dominio, lo que proporciona una comprensión más sencilla del dominio y la estructura de datos. Además, al usar el repositorio de Spring, no tenemos que preocuparnos por usar la API de JPA directamente, simplemente creando los métodos, que Spring crea la implementación en tiempo de ejecución, lo que hace que el código sea mucho más simple, pequeño y legible.

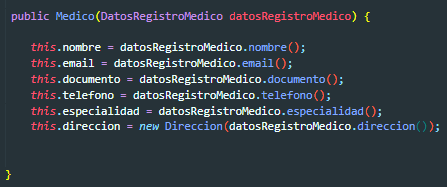
Migraciones flyway:

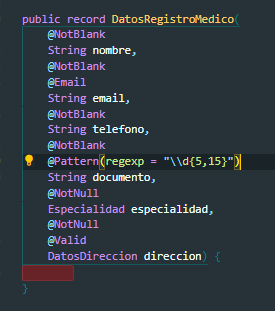






Validación:





Como se explicó en el video anterior, el Bean Validation se compone de varias anotaciones que se deben agregar a los atributos en los que queremos realizar las validaciones. Hemos visto algunas de estas anotaciones, como @NotBlank, que indica que un atributo String no puede ser nulo o vacío.

Nueva migración:



**Para saber más: Lombok**

Lombok, como se dijo anteriormente, es una biblioteca de Java especialmente enfocada en la reducción de código y en la productividad en el desarrollo de proyectos en ese lenguaje.

Él utiliza la idea de anotaciones (familiar a Spring ¿no?) para generar códigos en el tiempo de compilación. Pero recuerde que no vemos el código generado, y tampoco es posible cambiar lo que se ha generado.

**Para saber más: Anotación @Autowired en Spring**

Traducido del inglés, la palabra Autowired sería ''un cable automático''. En el contexto del framework Spring, que utiliza como una de sus bases el patrón de diseño “Inyección de Dependencias”, la idea sirve para definir una inyección automática en un determinado componente del proyecto Spring, ese componente puede ser atributos, métodos e incluso constructores.

Esta anotación se permite con la ayuda de la anotación @SpringBootApplication, en el archivo de configuración de Spring, disponible cada vez que se crea un proyecto Spring.

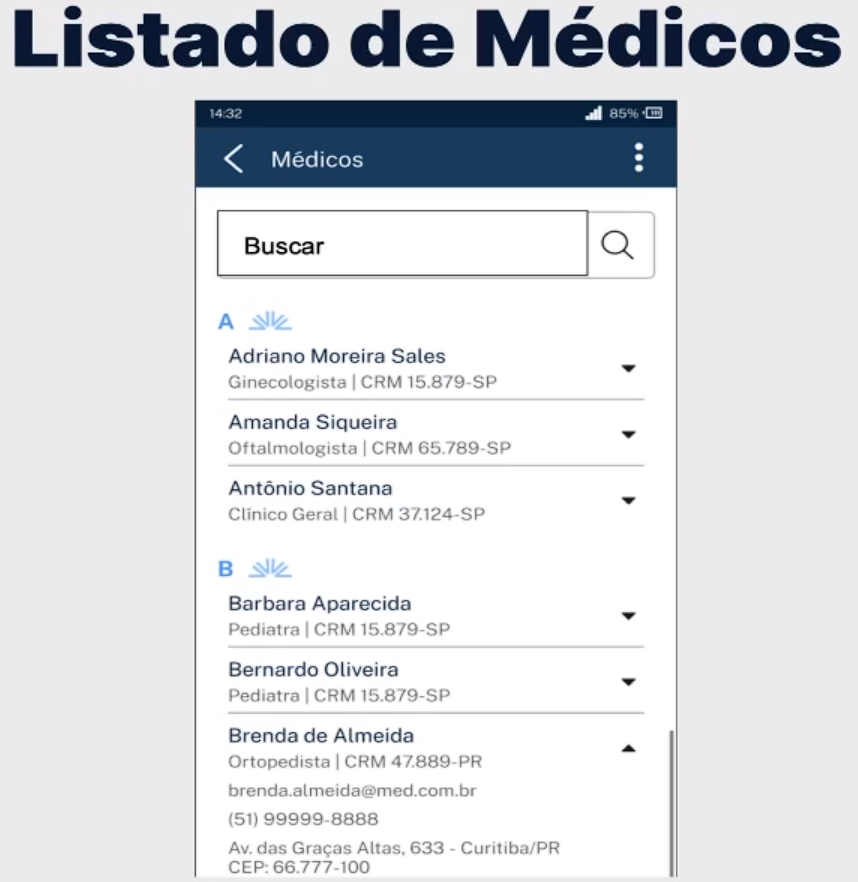
Al marcar un componente con la anotación @Autowired le estamos diciendo a Spring que el componente es un punto donde se debe inyectar una dependencia, en otras palabras, el componente se inyecta en la clase que lo posee, estableciendo una colaboración entre componentes.

Lo que Aprendimos:

**En esta clase, aprendiste a:**

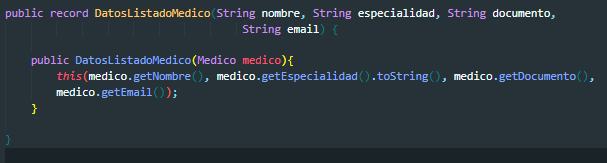
* Agregar nuevas dependencias en el proyecto;
* Asignar una entidad JPA y crear una interfaz de Repositorio para ella;
* Utilizar Flyway como herramienta de migración de proyectos;
* Realice validaciones con Bean Validation usando algunas de sus anotaciones, como @NotBlank.

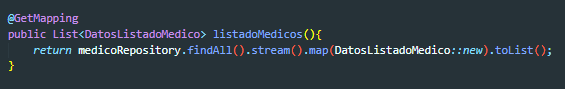
**Request GET:**



Produciendo datos:

Probando lista:





Usamos DTO para representar los datos que recibimos y devolvemos a través de la API, pero probablemente se esté preguntando "¿Por qué, en lugar de crear un DTO, no devolvemos directamente la entidad JPA en el Controller?". Para hacer esto, simplemente cambie el método list en el Controller a:

De esa forma, el código sería más ligero y no necesitaríamos crear el DTO en el proyecto.

Pero, ¿es esto realmente una buena idea?

**Problemas de recepción/devolución de la entidad JPA**

De hecho, es mucho más simple y cómodo no usar DTO, sino tratar directamente con entidades JPA en los Controllers. Sin embargo, este enfoque tiene algunas desventajas, incluida la vulnerabilidad de la aplicación a los ataques de **Mass Assignment**.

Uno de los problemas es el hecho de que, al devolver una entidad JPA en un método del Controller, Spring generará el JSON que contiene **todos** sus atributos, y este no siempre es el comportamiento que queremos.

Eventualmente podemos tener atributos que no queremos que sean devueltos en el JSON, ya sea por razones de seguridad, en el caso de datos *sensibles*, o incluso porque no son utilizados por clientes API.

**Uso de la anotación @JsonIgnore**

En esta situación, podríamos usar la anotación @JsonIgnore, que nos ayuda a ignorar ciertas propiedades de una clase Java cuando se serializa en un objeto JSON.

En el ejemplo anterior, el atributo 'salario' de la clase 'Empleado' no se mostrará en las respuestas JSON y el problema estaría resuelto.

Sin embargo, puede haber algún otro endpoint de la API en el que necesitemos enviar el salario de los empleados en el JSON, en cuyo caso tendríamos problemas, ya que con la anotación @JsonIgnore tal atributo **nunca** se enviará en el JSON, y al eliminar la anotación se enviará el atributo **siempre**. Por lo tanto, perdemos la flexibilidad de controlar cuándo se deben enviar ciertos atributos en el JSON y cuándo no.

**DTO**

El patrón DTO (*Data Transfer Object*) es un patrón arquitectónico que se usó ampliamente en aplicaciones Java distribuidas (arquitectura cliente/servidor) para representar los datos que eran enviados y recibidos entre aplicaciones cliente y servidor.

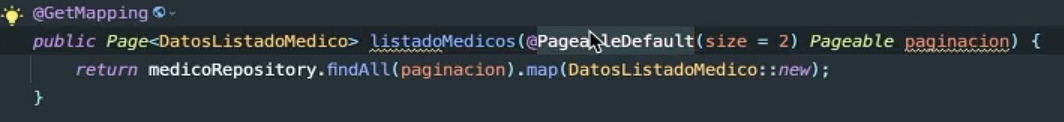
El patrón DTO puede (y debe) usarse cuando no queremos exponer todos los atributos de alguna entidad en nuestro proyecto, una situación similar a los salarios de los empleados que discutimos anteriormente. Además, con la flexibilidad y la opción de filtrar qué datos se transmiten, podemos ahorrar tiempo de procesamiento.

**Bucle infinito que causa StackOverflowError**

Al devolver un objeto de tipo 'Producto' en el Controller, Spring tendría problemas para generar el JSON de este objeto, lo que provocaría una excepción de tipo 'StackOverflowError'. Este problema ocurre porque el objeto producto tiene un atributo de tipo Categoría, que a su vez tiene un atributo de tipo Lista<Producto>, lo que provoca un bucle infinito en el proceso de serialización a JSON.

Este problema se puede resolver usando la anotación @JsonIgnore o usando las anotaciones @JsonBackReference y @JsonManagedReference, pero también se puede evitar usando un DTO que represente solo los datos que se deben devolver en el JSON.

Paginación:



Ordenación:



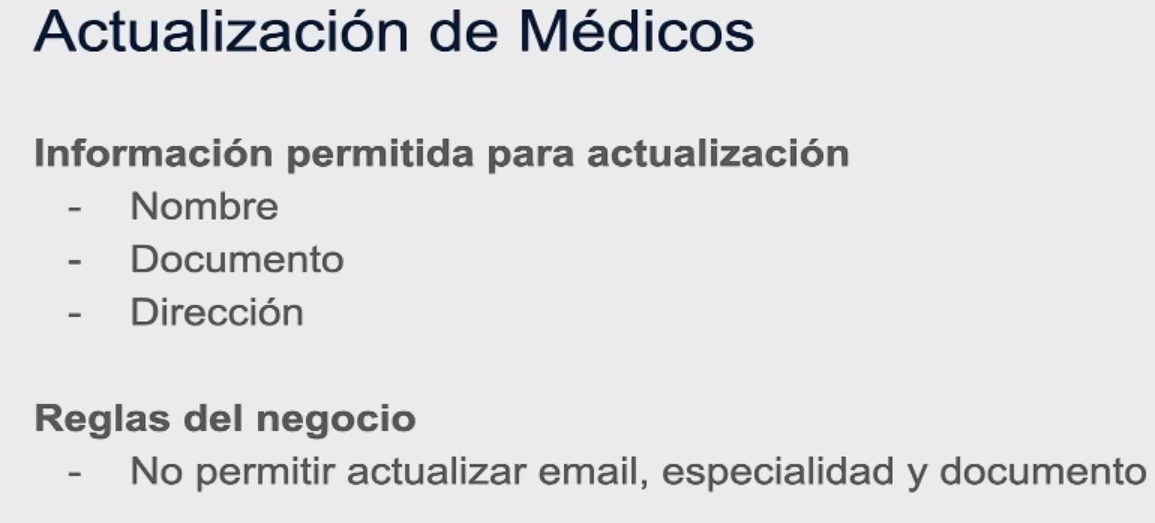
Lo que Aprendimos:

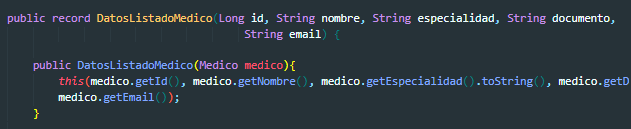
**En esta clase, aprendiste a:**

* Usar la anotación @GetMapping para mapear métodos en los Controllers que producen datos;
* Usar la interfaz Pageable de Spring para realizar consultas con paginación;
* Controlar la paginación y el ordenamiento de los datos devueltos por la API con los parámetros page, size y sort;
* Configurar el proyecto para que los comandos SQL se visualicen en la consola.

**Request PUT y DELETE:**

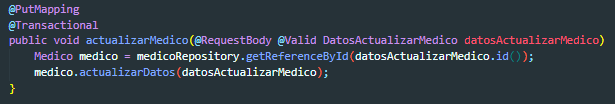
Request PUT:

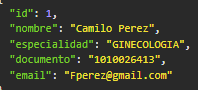






Actualizando datos:





Elegir entre el método HTTP PUT o PATCH es una pregunta común que surge cuando estamos desarrollando APIs y necesitamos crear un endpoint para la actualización de recursos. Comprendamos las diferencias entre las dos opciones y cuándo usar cada una.

**PUT**

El método PUT reemplaza todos los datos actuales de un recurso con los datos enviados en la solicitud, es decir, estamos hablando de una actualización completa. Entonces, con él, hacemos la actualización completa de un recurso en una sola solicitud.

**PATCH**

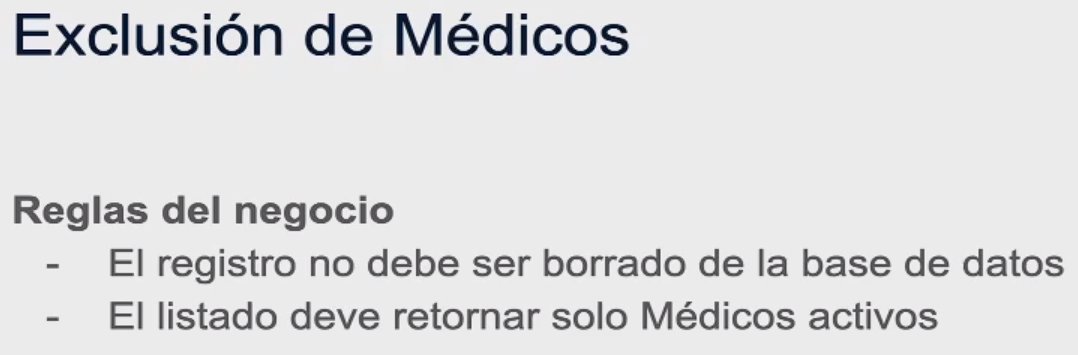
El método PATCH, a su vez, aplica modificaciones **parciales** a un recurso. Por lo tanto, es posible modificar solo una parte de un recurso. Con PATCH, entonces, realizamos actualizaciones parciales, lo que flexibiliza las opciones de actualización.

**¿Cuál elegir?**

En la práctica, es difícil saber qué método usar, porque no siempre sabremos si un recurso se actualizará parcial o completamente en una solicitud, a menos que lo verifiquemos, algo que no se recomienda.

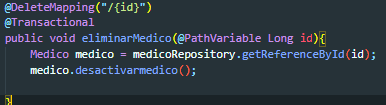
Entonces, lo más común en las aplicaciones es usar el método PUT para las solicitudes de actualización de recursos en una API, que es nuestra elección en el proyecto utilizado a lo largo de este curso.

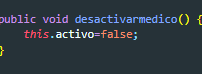
Request DELETE:

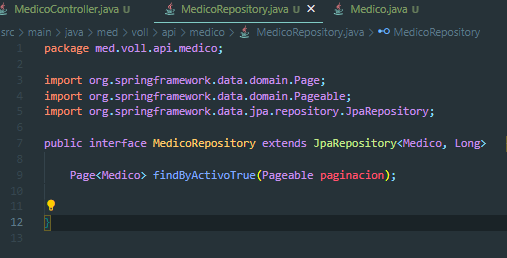




Exclusión lógica:







Lo que Aprendimos:

**En esta clase, aprendiste a:**

* Mapear solicitudes PUT con la anotación @PutMapping;
* Escribir un código para actualizar la información de un registro en la base de datos;
* Mapear solicitudes DELETE con la anotación @DeleteMapping;
* Mapear parámetros dinámicos en la URL con la anotación @PathVariable;
* Implementar el concepto de exclusión lógica utilizando un atributo booleano.