**CONTROLADORES REST EN SPRING BOOT**

| **Información general** | |
| --- | --- |
| **Duración estimada en minutos:** | 120 |
| **Docente:** | Carlos Andrés Florez Villarraga |
| **Guía no.** | 04 |

| **Información de la Guía** |
| --- |

**OBJETIVO**

Aprender a programar un API REST en Spring Boot a través de RestController y probar dicho servicio usando aplicaciones como Postman.

**CONCEPTOS BÁSICOS**

Bases de datos NoSQL MongoDB, Java, archivos JSON.

**CONTEXTUALIZACIÓN TEÓRICA**

Una API REST (Transferencia de Estado Representacional) es una interfaz de programación de aplicaciones que sigue las convenciones de la arquitectura REST, permitiendo así la comunicación entre sistemas de manera estandarizada.

En términos simples, una API es un conjunto de reglas y protocolos que especifica cómo deben interactuar diferentes aplicaciones o servicios entre sí. Gracias a las APIs, las aplicaciones pueden comunicarse sin necesidad de exponer todo su código fuente. Esto hace que las APIs sean una herramienta fundamental para el desarrollo de software, ya que proporcionan una forma estructurada y segura de interactuar con otras aplicaciones.

En el desarrollo de software, las APIs permiten que otras aplicaciones utilicen funcionalidades específicas de un sistema sin acceder al código fuente completo. Por ejemplo, en nuestro proyecto, una API es crucial para que el frontend (ya sea web, móvil, escritorio, etc.) se comunique de manera estandarizada y segura con el backend, que en este caso está construido en Java mediante Spring Boot.

La información en una API se transmite a través del protocolo HTTP, y generalmente en formatos como JSON (JavaScript Object Notation), HTML, XML o texto plano. De estos, JSON es el formato más utilizado, ya que es fácil de entender tanto para las máquinas como para los seres humanos, además de ser independiente del lenguaje de programación.

El protocolo HTTP es el encargado de transferir la información a través de la web, permitiendo la comunicación entre aplicaciones, servicios y servidores. Esta transferencia se lleva a cabo mediante mensajes que incluyen un método, el cual indica al servidor cómo debe manejar la solicitud. Los métodos HTTP más comunes son:

* GET: Se utiliza para obtener datos sin generar efectos secundarios en el servidor.
* POST: Generalmente se emplea para crear nuevos datos o realizar acciones que modifican el estado de los recursos.
* DELETE: Se usa para eliminar un recurso específico.
* PUT: Permite actualizar datos existentes en el servidor.

Además de los métodos, las APIs REST utilizan códigos de respuesta HTTP para indicar el resultado de una solicitud. Estos códigos son fundamentales para comprender si la solicitud fue procesada correctamente, si hubo algún error, o si se requiere alguna acción adicional. Algunos de los códigos más comunes son:

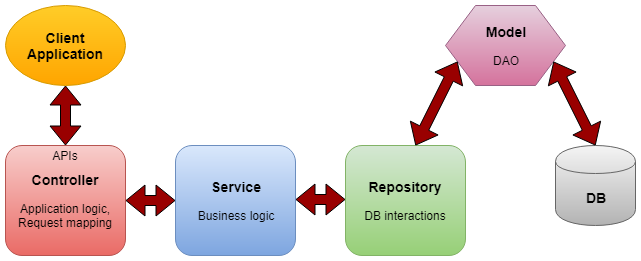
* 200 OK: La solicitud se ha procesado correctamente.
* 201 Created: El recurso solicitado se ha creado con éxito.
* 400 Bad Request: La solicitud tiene una sintaxis inválida o no puede ser procesada.
* 401 Unauthorized: La solicitud requiere autenticación.
* 404 Not Found: El recurso solicitado no se ha encontrado.
* 500 Internal Server Error: Ocurrió un error inesperado en el servidor.

Siguiendo las buenas prácticas basadas en OpenAPI, es recomendable proporcionar documentación clara y detallada sobre los códigos de respuesta de la API. Esto facilita la tarea de los desarrolladores, quienes pueden comprender rápidamente cómo manejar diferentes respuestas y posibles errores. Además, es esencial implementar una gestión adecuada de errores, proporcionando mensajes descriptivos y consistentes que no solo faciliten la depuración, sino que también contribuyan a la mejora continua de la API.

Es importante destacar que cada mensaje HTTP contiene una cabecera, que define los metadatos relacionados con el mensaje. Estos metadatos indican al servidor, la aplicación o el navegador web el propósito del mensaje, ayudando a establecer una comunicación más eficiente.

**API REST en Spring Boot**

En la siguiente imagen podemos ver una arquitectura muy resumida del acceso a la información desde la capa de presentación (cliente).



El proceso de petición desde el cliente a los servicios de Spring Boot sigue el modelo de arquitectura Cliente-Servidor y se basa en el protocolo HTTP (o HTTPS). Aquí hay un resumen general del proceso:

* Un cliente (como un navegador web o una aplicación móvil) inicia una solicitud enviando una petición HTTP al servidor Spring Boot.
* En el servidor Spring Boot, las peticiones son manejadas por controladores. Los controladores están anotados con @Controller, @RestController u otras anotaciones específicas de Spring.
* El controlador realiza las operaciones necesarias en función de la lógica de negocios. Puede interactuar con servicios, acceder a bases de datos, realizar operaciones y preparar datos para enviar como respuesta.
* Generación de la respuesta: Una vez que el controlador ha completado su lógica, genera una respuesta. La respuesta generada por el controlador se envía de vuelta al cliente a través del protocolo HTTP.
* El cliente (por ejemplo, un navegador web o una aplicación móvil) recibe la respuesta y realiza acciones según la lógica de la aplicación.

Para más información:

* <https://spring.io/guides/gs/rest-service>
* <https://developer.mozilla.org/es/docs/Web/HTTP/Methods>
* <https://www.redhat.com/es/topics/api/what-is-a-rest-api>

**Aplicaciones para probar la API**

Cada funcionalidad de un proyecto debe ser testeado para verificar su correcto funcionamiento. Una API REST se puede probar de múltiples formas, pero existe una herramienta muy útil que se llama Postman (<https://www.postman.com/>) que nos brinda la posibilidad de probar de manera rápida y sencilla cualquier servicio bajo diferentes arquitecturas, entre ellas REST. Postman nos permite enviar peticiones HTTP REST sin necesidad de desarrollar un cliente, ahorrándonos tiempo. Así mismo existe Swagger (<https://swagger.io/>) que al igual que Postman permite probar de manera sencilla toda nuestra API.

**PRECAUCIONES Y RECOMENDACIONES**

Recuerde verificar que tiene instalado el JDK de Java (preferiblemente la versión 21). Asegúrese de que tiene el servidor de MongoDB (mongod) en ejecución.

**ARTEFACTOS**

Se requiere tener instalado IntelliJ IDEA en su versión Ultimate, así como MongoDB y el controlador de versiones GIT.

**EVALUACIÓN O RESULTADO**

Se espera que el estudiante aprenda a programar un API REST en Spring Boot usando los estándares actuales, como por ejemplo respuestas en formato JSON y manejo adecuado de errores.

| **Procedimiento** |
| --- |

1. Asegúrese que tiene la siguiente dependencia añadida en el archivo build.grade.kts del proyecto de Spring Boot.

| *implementation*("org.springframework.boot:spring-boot-starter-web") |
| --- |

1. Cree una clase que se llame SaludoControlador. Para esto primero cree un paquete que se llame co.edu.uniquindio.proyecto.controladores. A esta clase añádale la anotación @RestController. Y haga que quede así:

| package co.edu.uniquindio.proyecto.controladores;  import org.springframework.web.bind.annotation.GetMapping;  import org.springframework.web.bind.annotation.PathVariable;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RestController;  @RestController  @RequestMapping("/saludo") // Prefijo común para todas las rutas del controlador  public class SaludoControlador {  @GetMapping  public String saludar(){  return "Hola, bienvenido a la aplicación";  }  @GetMapping("/{nombre}")  public String saludarNombre(@PathVariable String nombre){  return "Hola %s, bienvenido a la aplicación".formatted(nombre);  }  } |
| --- |

Con base en el código anterior, podemos destacar lo siguiente:

@RestController: Indica que esta clase es un controlador REST, lo que significa que manejará solicitudes HTTP y devolverá respuestas en formato de texto o JSON.

@RequestMapping("/saludo") para que todas las rutas comiencen con /saludo. Esto permite que las rutas queden más organizadas y evitamos posibles conflictos con otras rutas en la aplicación.

@GetMapping: Especifica que los métodos dentro del controlador responderán a solicitudes HTTP GET.

El primer método se “activa” cuando un cliente hace una solicitud GET a la raíz del controlador (/saludo). Por ejemplo: <http://localhost:8080/saludo> Devuelve el mensaje: "Hola, bienvenido a la aplicación".

El segundo método se “activa” cuando un cliente hace una solicitud GET a la raíz del controlador (/saludo), con un parámetro en la URL. Por ejemplo: <http://localhost:8080/saludo/Carlos> Devuelve el mensaje “Hola Carlos, bienvenido a la aplicación”.

**NOTA**: El puerto por defecto es el 8080, pero puede variar según la configuración del proyecto en el archivo application.properties.

1. Para probar estas peticiones se recomienda crear un archivo en la carpeta test/resources que se llame por ejemplo saludo.http y puede quedar así:

| ### Saludo  GET http://localhost:8080/saludo  ### Saludo con un nombre dado por parámetro  GET http://localhost:8080/saludo/Carlos |
| --- |

1. Pruebe cada ruta (para esto el servidor debe estar en ejecución).
2. Cree una nueva clase que se llame UsuarioControlador. En el paquete co.edu.uniquindio.proyecto.controladores. A esta clase añádale la anotación @RestController. Así:

| package co.edu.uniquindio.proyecto.controladores;  import org.springframework.web.bind.annotation.RestController;  @RestController  public class UsuarioControlador {    } |
| --- |

**NOTA:** Cambie proyecto por el nombre de su proyecto (o el nombre del paquete que ya haya creado previamente (así mismo con el resto de la guía).

1. Adicione a dicha clase la url (endpoint) que reconocerá el controlador para dar respuesta a las peticiones HTTP. Agregue lo siguiente debajo de @RestController.

| @RequestMapping("/api/usuarios") |
| --- |

Esto significa que cuando se pida la url <http://localhost:8080/api/usuarios> será el controlador REST que estamos programando quien responda ante las peticiones realizadas.

1. Cree una variable de tipo UsuarioServicio en la clase UsuarioControlador.

| private UsuarioServicio usuarioServicio; |
| --- |

Para que esto funcione, debe crear una interface con el nombre UsuarioServicio en el paquete co.edu.uniquindio.proyecto.servicios. La idea es que esta interface tenga la definición de los métodos que deben ser implementados con su respectiva lógica de negocio. Por ahora solo nos interesa hacer la definición de los métodos (no su implementación).

1. Cree un método para cada servicio de negocio de los usuarios que sea pertinente para la API. **A manera de ejemplo** se hará el controlador implementando solamente los métodos para hacer un CRUD básico en la entidad. Estos métodos se deben crear en la clase UsuarioControlador.

| public void crear(CrearUsuarioDTO cuenta) throws Exception{  }  public void editar(EditarUsuarioDTO cuenta) throws Exception{  }  public void eliminar(String id) throws Exception{  }  public UsuarioDTO obtener(String id) throws Exception{  }  public List<UsuarioDTO> listarTodos(){  } |
| --- |

**NOTA:** Cada método del controlador debe ser equivalente a los métodos del servicio de los usuarios (tanto en su retorno, su nombre, sus excepciones y sus parámetros). Es necesario crear DTOs que encapsulan los datos tanto de entrada como de salida de la API. Estos DTOs deben estar en el paquete: co.edu.uniquindio.proyecto.dto.

1. Cree los DTOs con los atributos correspondientes para cada método del CRUD. Un DTO es una clase que encapsula la información necesaria para cada requisito específico de la aplicación, separando la lógica de negocio de la estructura de los datos que se intercambian entre el cliente y el servidor.

Dado que los DTO son objetos que no tienen lógica y solo sirven para encapsular atributos, podemos hacer uso de los record de Java, entonces, por ejemplo el DTO CrearUsuarioDTO se puede crear así:

| package co.edu.uniquindio.proyecto.dto;  public record CrearUsuarioDTO(  String nombre,  String telefono,  String ciudad,  String direccion,  String email,  String password  ) {  } |
| --- |

Para más información sobre DTO y los records de Java:

* <https://www.arquitecturajava.com/java-record-class-y-jdk-14/>
* <https://www.arquitecturajava.com/data-transfer-object-dto-un-concepto-clave/>
* <https://reactiveprogramming.io/blog/es/patrones-arquitectonicos/dto>

1. También se debe hacer validaciones sobre cada uno de los atributos de los DTOs que tienen información que se registra (o actualiza) en el backend (en una base de datos por ejemplo). Por lo tanto, con las validaciones, el DTO CrearUsuarioDTO debe quedar así:

| package co.edu.uniquindio.proyecto.dto;  import jakarta.validation.constraints.Email;  import jakarta.validation.constraints.NotBlank;  import org.hibernate.validator.constraints.Length;  public record CrearUsuarioDTO(  @NotBlank @Length(max = 100) String nombre,  @Length(max = 10) String telefono,  @NotBlank @Length(max = 100) String ciudad,  @NotBlank @Length(max = 100) String direccion,  @NotBlank @Length(max = 50) @Email String email,  @NotBlank @Length(min = 7, max = 20) String password  ) {  } |
| --- |

El DTO debe tener las validaciones pertinentes para garantizar una buena integridad de los datos. Todos los DTO que represente un objeto que será guardado en la base de datos debe tener las validaciones.

Para más información sobre las anotaciones de validaciones:

* <https://javaee.github.io/javaee-spec/javadocs/javax/validation/constraints/package-summary.html>

1. Cree el DTO EditarUsuarioDTO con las validaciones pertinentes. Adicionalmente, cree el DTO UsuarioDTO, el cual será utilizado para devolver información al cliente. Dado que este último DTO es una respuesta generada por el backend, no requiere validaciones, ya que los datos ya fueron previamente procesados y verificados.

Para EditarUsuarioDTO, puede estructurarse de la siguiente manera (se asume que el usuario no puede modificar su correo electrónico ni su contraseña; para cambiar la contraseña, deberá utilizar otro mecanismo):

| package co.edu.uniquindio.proyecto.dto;  import jakarta.validation.constraints.NotBlank;  import org.hibernate.validator.constraints.Length;  public record EditarUsuarioDTO(  @NotBlank String id,  @NotBlank @Length(max = 100) String nombre,  @NotBlank @Length(max = 100) String ciudad,  @NotBlank @Length(max = 100) String direccion,  @Length(max = 10) String telefono  ) {  } |
| --- |

1. Como se explicó previamente, a cada método de UsuarioControlador se le debe añadir una anotación de acuerdo al método HTTP que deba dar respuesta. Para obtener se usa GET, para guardar o registrar se usa POST, para borrar DELETE y para modificar PUT.

| @PostMapping  public void crear(CrearUsuarioDTO cuenta) throws Exception{  }  @PutMapping  public void editar(EditarUsuarioDTO cuenta) throws Exception{  }  @DeleteMapping("/{id}")  public void eliminar(String id) throws Exception{  }  @GetMapping("/{id}")  public UsuarioDTO obtener(String id) throws Exception{  return null;  }  @GetMapping  public List<UsuarioDTO> listarTodos(){  return null;  } |
| --- |

1. Algunos métodos necesitan información que puede venir por la url o en el cuerpo de la petición. Para esos casos se debe indicar explícitamente como va a llegar dicha información. Si es por la url se debe usar la anotación @PathVariable y si es dentro del cuerpo de la petición se usa @RequestBody, para este último caso se recomienda que vaya acompañado de la anotación @Valid (para que tenga en cuenta las anotaciones de validación de los atributos de los DTO).

Por ejemplo, si deseo obtener la información de todo un objeto, lo más obvio sería enviar el código (id) de dicho objeto y con base en ese valor retornar toda la información restante. Por lo tanto en la url se puede enviar el código. Pero, para el registro de un usuario se debe enviar todo el objeto que se quiera guardar, por lo tanto lo ideal es escribir esta información en el cuerpo de la petición.

**NOTA:** Para el caso de @PathVariable, el nombre del atributo debe coincidir con el nombre que está entre llaves en la url del mapping. Por ejemplo, si tenemos @PathVariable int id, entonces la url sería "/{id}", como se observa en el código anterior.

Se puede implementar cada uno de los métodos del controlador REST haciendo uso de las anotaciones correspondientes y del servicio de negocio de los usuarios, así:

| @PostMapping  public void crear(@Valid @RequestBody CrearUsuarioDTO cuenta) throws Exception{  usuarioServicio.crear(cuenta);  }  @PutMapping  public void editar(@Valid @RequestBody EditarUsuarioDTO cuenta) throws Exception{  usuarioServicio.editar(cuenta);  }  @DeleteMapping("/{id}")  public void eliminar(@PathVariable String id) throws Exception{  usuarioServicio.eliminar(id);  }  @GetMapping("/{id}")  public UsuarioDTO obtener(@PathVariable String id) throws Exception{  return usuarioServicio.obtener(id);  }  @GetMapping  public List<UsuarioDTO> listarTodos(){  return usuarioServicio.listarTodos();  } |
| --- |

Fíjese que los métodos del controlador no tienen lógica de negocio, simplemente se hace el llamado correspondiente a cada método que está programado en los servicios y se le pasan los mismos parámetros.

1. Como vimos en los métodos anteriores, algunos métodos son void, otros lanzan excepciones que puedan aparecer al momento de realizar alguna operación de consulta, registro, etc. Lo ideal es retroalimentar al usuario siempre, cuando todo funcione bien informarle y cuando algo falla también.

Por ejemplo, para la creación de una nueva cuenta de usuario podemos modificarlo de la siguiente manera para que siempre haya una respuesta.

| @PostMapping  public ResponseEntity<MensajeDTO<String>> crear(@Valid @RequestBody CrearUsuarioDTO cuenta) throws Exception{  usuarioServicio.crear(cuenta);  return ResponseEntity.*ok*(new MensajeDTO<>(false, "Su registro ha sido exitoso"));  } |
| --- |

El objeto ResponseEntity nos permite asignar un código de respuesta (status) y un contenido, para este caso estamos devolviendo un mensaje en formato JSON. Se debe crear un record MensajeDTO, dicho record tendrá un atributo de tipo boolean que indica si hay error o no, y otro atributo de tipo T (genérico) que representará la respuesta como tal. En este caso T será de tipo String. El objetivo del DTO MensajeDTO es que todas las respuestas del servidor tengan la misma estructura.

En este otro ejemplo, si existe el codigo de la cuenta de usuario retornamos el detalle de la misma, este detalle está en el objeto UsuarioDTO:

| @GetMapping("/{id}")  public ResponseEntity<MensajeDTO<UsuarioDTO>> obtener(@PathVariable String id) throws Exception{  UsuarioDTO info = usuarioServicio.obtener(id);  return ResponseEntity.*ok*(new MensajeDTO<>(false, info));  } |
| --- |

Este sería el código para eliminar la cuenta de un usuario con el ajuste indicado anteriormente.

| @DeleteMapping("/{id}")  public ResponseEntity<MensajeDTO<String>> eliminar(@PathVariable String id) throws Exception{  usuarioServicio.eliminar(id);  return ResponseEntity.*ok*(new MensajeDTO<>(false, "Cuenta eliminada exitosamente"));  } |
| --- |

Este sería el código para listar todas las cuentas de los usuarios.

| @GetMapping  public ResponseEntity<MensajeDTO<List<UsuarioDTO>>> listarTodos(){  List<UsuarioDTO> lista = usuarioServicio.listarTodos();  return ResponseEntity.*ok*(new MensajeDTO<>(false, lista));  } |
| --- |

Y finalmente, este sería el método de editar su perfil:

| @PutMapping  public ResponseEntity<MensajeDTO<String>> editar(@Valid @RequestBody EditarUsuarioDTO cuenta) throws Exception{  usuarioServicio.editar(cuenta);  return ResponseEntity.*ok*(new MensajeDTO<>(false, "Cuenta editada exitosamente"));  } |
| --- |

Códigos de estado HTTP: <https://developer.mozilla.org/es/docs/Web/HTTP/Status>

1. El manejo de las excepciones se debe programar en una nueva clase a la que llamaremos RestExceptionHandler. Básicamente esta clase mostrará un mensaje de error personalizado al usuario según el tipo de excepción que arroje el servicio (cualquier servicio).

Dentro del paquete co.edu.uniquindio.proyecto.excepciones cree la clase RestExceptionHandler con la anotación @RestControllerAdvice así:

| package co.edu.uniquindio.proyecto.excepciones;  import co.edu.uniquindio.proyecto.dto.MensajeDTO;  import co.edu.uniquindio.proyecto.dto.ValidacionDTO;  import org.springframework.http.ResponseEntity;  import org.springframework.validation.BindingResult;  import org.springframework.validation.FieldError;  import org.springframework.web.bind.MethodArgumentNotValidException;  import org.springframework.web.bind.annotation.ExceptionHandler;  import org.springframework.web.bind.annotation.RestControllerAdvice;  import org.springframework.web.servlet.resource.NoResourceFoundException;  import java.util.ArrayList;  import java.util.List;  @RestControllerAdvice  public class RestExceptionHandler {  @ExceptionHandler(NoResourceFoundException.class)  public ResponseEntity<MensajeDTO<String>> noResourceFoundExceptionHandler (NoResourceFoundException ex){  return ResponseEntity.*status*(404).body( new MensajeDTO<>(true, "El recurso no fue encontrado") );  }  @ExceptionHandler(Exception.class)  public ResponseEntity<MensajeDTO<String>> generalExceptionHandler (Exception e){  return ResponseEntity.*internalServerError*().body( new MensajeDTO<>(true, e.getMessage()) );  }  @ExceptionHandler(MethodArgumentNotValidException.class)  public ResponseEntity<MensajeDTO<List<ValidacionDTO>>> validationExceptionHandler ( MethodArgumentNotValidException ex ) {  List<ValidacionDTO> errores = new ArrayList<>();  BindingResult results = ex.getBindingResult();  for (FieldError e: results.getFieldErrors()) {  errores.add( new ValidacionDTO(e.getField(), e.getDefaultMessage()) );  }  return ResponseEntity.*badRequest*().body( new MensajeDTO<>(true, errores) );  }  } |
| --- |

Cada método de la clase RestExceptionHandler responde a una excepción que puede ser arrojada en cualquier momento por cualquier servicio de negocio. Para diferenciar cada método se hace uso de la anotación @ExceptionHandler. Acá una breve descripción de cada una:

* NoResourceFoundException: Excepción que se lanza cuando se trata de acceder a un recurso que no existe.
* Exception: Excepción general de Java. Lo ideal es tener excepciones propias para cada tipo de error y no ponerle a todas las excepciones simplemente Exception.
* MethodArgumentNotValidException: Excepción que se lanza cuando no se cumple alguna validación puesta en los DTO (@NotNull, @Email, @Max, etc..). Para este caso debe crear un nuevo DTO con el nombre: ValidacionDTO, este record debe debe tener dos String: campo y mensaje.

1. Pruebe cada ruta, para esto, cree un nuevo archivo en la carpeta test/resources que se llame usuarios.http y que tenga lo siguiente:

| ### Obtener todos los usuarios  GET http://localhost:8080/api/usuarios  ### Obtener un usuario por su id  GET http://localhost:8080/api/usuarios/1  ### Crear un usuario  POST http://localhost:8080/api/usuarios  *Content-Type*: application/json  {  "nombre": "Carlos",  "ciudad": "Armenia",  "direccion": "Calle 123",  "email": "carlos@email.com",  "password": "123456"  }  ### Actualizar un usuario  PUT http://localhost:8080/api/usuarios  *Content-Type*: application/json  {  "id": 1,  "nombre": "Carlos",  "ciudad": "Armenia",  "direccion": "Nueva dirección"  }  ### Eliminar un usuario  DELETE http://localhost:8080/api/usuarios/1 |
| --- |

Lo más probable es que falle ya que la interface UsuarioServicio aún no tiene ninguna implementación, por lo tanto puede hacer una implementación básica con datos “quemados” para probar o incluso dejar los métodos vacíos. Para esto, cree la clase UsuarioServicioImpl en el paquete co.edu.uniquindio.proyecto.servicios.impl.

| package co.edu.uniquindio.proyecto.servicios.impl;  import co.edu.uniquindio.proyecto.servicios.UsuarioServicio;  import org.springframework.stereotype.Service;  import java.util.List;  @Service  public class UsuarioServicioImpl implements UsuarioServicio {  //*TODO implementar todos los métodos de la interfaz*  } |
| --- |

@Service es una anotación de Spring que indica que esta clase es un servicio. Se usa en la capa de servicio para manejar la lógica de negocio.

1. Modifique la clase UsuarioControlador para que use la implementación de la interfaz hecha en el punto anterior. Para esto, debe hacer que la variable usuarioServicio sea final, así:

| private final UsuarioServicio usuarioServicio; |
| --- |

Para que esta variable se inicialice, se debe agregar la anotación @RequiredArgsConstructor (de Lombok) a nivel de la clase (debajo de la anotación @RestController).

Esta anotación genera automáticamente un constructor con los argumentos requeridos. Se aplica a los atributos marcados como final, lo que significa que Spring los inyectará automáticamente cuando lo requiera. Esto evita el código repetitivo de escribir un constructor manualmente.

Si no usamos la anotación @RequiredArgsConstructor, nos tocaría escribir el constructor así:

| public UsuarioControlador(UsuarioServicio usuarioServicio) {  this.usuarioServicio = usuarioServicio;  } |
| --- |

1. Pruebe cada ruta y verifique su correcto funcionamiento, incluyendo los casos en los que falle. (para esto el servidor debe estar en ejecución).
2. Modifique el método listarTodos de la clase UsuarioControlador para que permita obtener usuarios filtrados por su nombre y por su ciudad (ambos opcionales). En este caso, se aconseja hacer algo así:

| @GetMapping  public ResponseEntity<MensajeDTO<List<UsuarioDTO>>> listarTodos(  @RequestParam(required = false) String nombre,  @RequestParam(required = false) String ciudad  ) {  List<UsuarioDTO> usuarios = usuarioServicio.listarTodos(nombre, ciudad);  return ResponseEntity.*ok*(new MensajeDTO<>(false, usuarios));  } |
| --- |

@RequestParam se usa para enviar parámetros por la url a la petición. En este caso, estas peticiones son válidas:

| ### Obtener todos los usuarios que tengan un nombre dado  GET http://localhost:8080/api/usuarios?nombre=Carlos  ### Obtener todos los usuarios cuya ciudad de residencia sea Armenia  GET http://localhost:8080/api/usuarios?ciudad=Armenia  ### Obtener todos los usuarios cuya ciudad de residencia sea Armenia y su nombre sea Carlos  GET http://localhost:8080/api/usuarios?ciudad=Armenia&nombre=Carlos |
| --- |

**NOTA:** Se recomienda crear un DTO cuando se requieren múltiples filtros y utilizar @ModelAttribute en lugar de @RequestParam individuales, esto para mantener un código más limpio y organizado.

1. Modifique el servicio listarTodos para que acepte los filtros, cree algunos datos de prueba y verifique que el filtro funciona correctamente.
2. Cree una nueva clase y programe un nuevo @RestController para manejar la parte de los reportes (alertas) del proyecto final (puede empezar con el CRUD). Consulte las buenas prácticas para crear las rutas y los códigos de respuesta. Y haga el respectivo archivo de pruebas (.http).
3. Programe los demás @RestController para todos requerimientos del proyecto final. Consulte las buenas prácticas para crear las rutas y los códigos de respuesta.
4. Piense qué rutas deben estar protegidas (se requiere autenticación para usarla)
5. Para probar los diferentes métodos de los controladores podemos hacerlo de muchas maneras, en esta guía veremos tres: Postman, Swagger (OpenAPI) y pruebas unitarias con MockMvc. No es necesario que use los tres métodos para probar la API, **con que use uno es suficiente**.

| **Para la próxima clase** |
| --- |

* ¿Cómo se pueden obtener respuestas paginadas? Investigar sobre PageRequest y Pageable.
* ¿Qué otras herramientas sirven para probar APIs?
* ¿Qué es Spring Security y qué aporta en el backend?
* Investigue sobre GraphQL, ¿para qué sirve? ¿Qué ventajas y desventajas tiene frente a REST?