DTO, Lombok y Manejo de errores

¿Qué es un DTO?

Los DTO (Data Transfer Object) son un tipo de objetos que sirven únicamente para transportar datos. EL DTO contiene las propiedades del objeto. Datos que pueden tener su origen en una o más entidades de información.

Un DTO normalmente no provee lógica de negocios o validaciones de ningún tipo. Algunos autores remarcan que el DTO debe ser inmutable, dado que sus cambios no deben reflejarse en el sistema.

Lombok

Lombok es una librería para Java que permite, mediante anotaciones, generar constructores y los métodos getters, setters, equals, hashcode y toString, entre otras cosas.

Algunas anotaciones de Lombok son:

@AllArgsConstructor

@NoArgsConstructor

@Getter

@Setter

@ToString

@EqualsAndHashCode

@Data

Records

A partir de Java 14 se agregaron de forma experimental, y en Java 17 con una versión estable, las clases Records que funcionan parecido a Lombok.

Solo se le debe indicar qué atributos contendrá la clase y esta genera automáticamente los métodos getter, hashCode, equals, toString, el constructor con todos los atributos.

A diferencia de Lombok la clase Record está pensada para hacer objetos inmutables, inicialmente se la pensó como un sustituto a los DTO.

Java Api Validations

- Spring implementa la validación definida en el JSR-303 (Java Specification Requests) cuya implementación más popular es Hibernate Validation

- Proporciona una serie de anotaciones que realizan validaciones específicas sobre atributos
- Para incluirla debemos agregar, al pom.xml, la siguiente depedencia

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

Anotaciones más usadas

@NotNull

@Size(min= , max =)

@Min(value="") / @Max (value="")

@Past / @PastOrPresent

@Future / @FutureOrPresent

@Digits(integer=, fraction=)

@Email

@NotBlank

@NotEmpty

Otras anotaciones

Hay anotaciones que no son parte del estándar pero que están disponibles en la librería

- @Url
- @Range(min= , max=)
- @Length(min=, max=)
- @ISBN
- @CrediCardNumber
- @Mod11Check

Validar en Spring Boot

- Para comprobar si un objeto cumple con sus requisitos debemos agregarle a un @RequestBody la anotación @Valid

- Si la validación falla lanzará una MethodArgumentNotValidException

- De forma predeterminada Spring devolverá una respuesta con código http 400 (Bad Request)

Validar PathVariable o RequestParam

```java
@RestController
@Validated
@RequestMapping("/api/test")
public class ValidarParametros {

  @GetMapping("/m1/{id}")
  ResponseEntity<String> validatePathVariable(
      @PathVariable("id") @Min(5) int id) {
    return ResponseEntity.ok("valid");
  }

  @GetMapping("/m2")
  ResponseEntity<String> validateRequestParameter(
      @RequestParam("param") @Min(5) int param) {
    return ResponseEntity.ok("valid");
  }
}
```

http://localhost:9011/api/test/m1/4

http://localhost:9011/api/test/m2?param=99

Manejo de Excepciones

- Para manejar excepciones se requieren armar bloques try-catch

```
try{
    dto= competenciaService.getByAnio(anio);
}
catch(NoEncontradoException e){
    return ResponseEntity.status(HttpStatus.UNPROCESSABLE_ENTITY).body(null);
}
catch(Exception e){
    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(null);
}

return ResponseEntity.ok().body(dto);
```

Manejo de Excepción

- Para evitar que las clases se llenen de bloques try-catch Spring nos da un controller especial llamado CotrollerAdviced.

- Los ControllerAdviced son un handler de expceciones, este handler escucha los sucesos de excepciones que les definamos y nos permiten encapsular el comportamiento para esas excepciones. Con el ejemplo anterior.

```java
@ExceptionHandler(EntidadNoEncontradaException.class)  no usages
public ResponseEntity<?> handlerEntidadNoEncontradoException(EntidadNoEncontradaException exception, HttpServletRequest request){

    log.error("Entidad no encontrda en " + request.getRequestURL().toString() + "exception "
            + exception.getLocalizedMessage());

    ApiErrorDTO apiErrorDTO = new ApiErrorDTO(
        exception.getLocalizedMessage(),
        LocalDate.now());

    return ResponseEntity.status(HttpStatus.NOT_FOUND).body(apiErrorDTO);
}

 @ExceptionHandler(MethodArgumentNotValidException.class)  no usages
 public ResponseEntity<?> handlerMethodArgumentNotValidExcpetion(MethodArgumentNotValidException exception, HttpServletRequest request){

     log.error("Body request incorrecto en  " + request.getRequestURL().toString() + "exception "
             + exception.getLocalizedMessage());

     ApiErrorDTO apiErrorDTO = new ApiErrorDTO(
         exception.getLocalizedMessage(),
         LocalDate.now());

     return ResponseEntity.badRequest().body(apiErrorDTO);
 }
```

Manejo de Excepción

```java
@ControllerAdvice  1 usage
public class GlobalHandlerException {

    private Logger log = LoggerFactory.getLogger(GlobalHandlerException.class);  5 usages

    @ExceptionHandler(Exception.class)  no usages
    public ResponseEntity<?> handlerGenericException(Exception exception, HttpServletRequest request){

        log.error("Error no controlado en  " + request.getRequestURL().toString() + "exception "
                + exception.getLocalizedMessage());

        ApiErrorDTO apiErrorDTO = new ApiErrorDTO(
            mensaje: "Error del servidor, vuelva a intentarlo mas tarde",
            LocalDate.now());

        return ResponseEntity.internalServerError().body(apiErrorDTO);

    }
```

¿Cómo definir excepciones personalizadas?

- Para crear una excepción personalizada solo basta extender una clase de Exception o RuntimeException

- En el constructor debemos utilizar super para poder usar el constructor de la clase padre

```
public class EntidadNoEncontradaException extends RuntimeException {

    public EntidadNoEncontradaException(String message) { super(message); }

}
```