Antes de meternos en el asunto, he de decir, que el fichero llamado application.resources, que está en la carpeta resources, lo he renombrado y se llama application.yml.

Ese cambio, viene porque próximamente vamos a incorporar (si todo va bien) la validación con el cas de la junta y nuestro compañero Miguel Angel Tercero lo tiene así.

Por lo tanto, el fichero en cuestión quedaría de la forma

```
# Configuración del datasource
spring:
  datasource:
    url: jdbc:h2:mem:alumnosdb
    username: user
    password: 1234
    driver-class-name: org.h2.Driver
  h2:
    console:
      enabled: true
  jpa:
    hibernate:
      ddl-auto: create-drop



logging:
  level:
    org:
      hibernate:
        SQL: debug
```

Aunque van a tener el código fuente en un fichero rar .

Ahora Sí

En este documento, vamos a gestionar un poco los errores en el backed, de forma que por ejemplo no pueda guardar un alumno si falta algún dato e incluso que informe que un alumno no existe si realmente no existe.

Para que se vea mejor este ejemplo, vamos a modificar un poco la clase Alumno, incorporando una propiedad mas y poniendo las etiquetas correspondientes para que esas propiedades sean por ejemplo obligatorias.

Teníamos la clase Alumno de la forma siguiente:

```
package com.jccm.backend.angular.models.entity;

import java.io.Serializable;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.Table;


@Entity

@Table(name="alumnos")

public class Alumno implements Serializable {


	private static final long serialVersionUID = 1L;

	@Id

	@GeneratedValue( strategy=GenerationType.IDENTITY)

	private Long id;

	private String nombre;

	private String apellidos;

	private String email;

}
```

Pues ahora, cambiamos la clase Alumno añadiendo una propiedad adicional que se llamará numescolar y que además será obligatoria.

Por lo tanto, vamos a cambiar la clase Alumno y nos va a quedar así, en la que ponemos que el nombre va a ser obligatorio, el número escolar que va a ser obligatorio y que además tiene que ser único en la base de datos. De tal forma que la clase alumnos quedará así:

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

	@Column( nullable = false )
	private String nombre;

	private String apellidos;

	@Column( nullable = false, unique = true)
	private String numescolar;

	private String email;
```

Añadiendo a esto los Getters y Setter.

Una vez que hemos hecho esto, y antes de que se nos olvide, teníamos en la carpeta resources un fichero llamado import.sql que contenía datos de ejemplo para cuando iniciamos la aplicación. Como hemos cambiado la clase entity Alumnos, pues modificamos también ese fichero, para que cuando se levante la aplicación guarde en la base de datos registros de alumnos que ya incluyen un número escolar.

```
INSERT INTO alumnos (nombre,  apellidos, numescolar, email ) VALUES('Juan', 'Pérez de Tudela', '258741' ,'juanperez@gmail.com');
INSERT INTO alumnos (nombre,  apellidos, numescolar, email ) VALUES('Alicia', 'Sánchez Gómez', '258744' , 'alicia@gmail.com');
INSERT INTO alumnos (nombre,  apellidos, numescolar, email ) VALUES('Hernán', 'Cortés de Monroy', '258731'
,'hernancortes@gmail.com');
INSERT INTO alumnos (nombre,  apellidos, numescolar, email ) VALUES('Carlos', 'García', '258671' ,'carlitos@gmail.com');
INSERT INTO alumnos (nombre,  apellidos, numescolar, email ) VALUES('Juan', 'Pérez de Tudela','348741' , 'juanperez@gmail.com');
INSERT INTO alumnos (nombre,  apellidos, numescolar, email ) VALUES('Alicia', 'Sánchez Gómez', '257841' ,'alicia@gmail.com');
INSERT INTO alumnos (nombre,  apellidos, numescolar, email ) VALUES('Hernán', 'Cortés de Monroy', '748741'
,'hernancortes@gmail.com');
INSERT INTO alumnos (nombre,  apellidos, numescolar, email ) VALUES('Carlos', 'García', '299641' ,'carlitos@gmail.com');
```

Ahora vamos con el lío principal…

Así es como tenemos hasta ahora el controlador. Ya sabemos que es el encargado de llamar a los métodos de los servicios para listar, guardar, editar y borrar.

Ahora tenemos el controlador de la siguiente forma.

```
package com.jccm.backend.angular.controllers;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

import com.jccm.backend.angular.models.entity.Alumno;
import com.jccm.backend.angular.models.services.IAlumnoService;


@RestController
@RequestMapping("/api")
public class AlumnosRestController {

	@Autowired
	private IAlumnoService alumnoService;

	@GetMapping("/alumnos")
	public List<Alumno> index(){
		return alumnoService.findAll();
	}

	@GetMapping("/alumnos/{id}")
	public Alumno verAlumno(@PathVariable Long id) {

		return alumnoService.findById(id);
```

```
	}

	@PostMapping("/alumnos")
	@ResponseStatus(HttpStatus.CREATED)
	public Alumno guardar(@RequestBody Alumno alumno) {
		return alumnoService.save(alumno);

	}

	@PutMapping("/alumnos/{id}")
	@ResponseStatus(HttpStatus.CREATED)
	public Alumno actualizar(@RequestBody Alumno alumno,
@PathVariable Long id) {

		Alumno alumnoActual = alumnoService.findById(id);

		alumnoActual.setApellidos( alumno.getApellidos() );
		alumnoActual.setEmail( alumno.getEmail() );
		alumnoActual.setNombre( alumno.getNombre() );

		return alumnoService.save(alumnoActual);

	}

	@DeleteMapping("/alumnos/{id}")
	@ResponseStatus(HttpStatus.NO_CONTENT)
	public void borrar(@PathVariable Long id) {
		alumnoService.delete(id);
	}

}
```

Como se puede ver, por ejemplo el método verAlumno(), devuelve un objeto de tipo Alumno. Por ejemplo, en el método guardar(), aparte de devolver un objeto de tipo Alumno, se está devolviendo un código de respuesta http, para informar que ha sido creado correctamente.

Vamos a ir paso a paso.

Primero, la cabecera de la clase.

La vamos a dejar así, donde

```
@CrossOrigin( origins = {"http://localhost:4200"})
@RestController
@RequestMapping("/api")
public class AlumnosRestController {
```

Aquí hemos implementado cors, para especificar las ips o dominios permitidos. Hemos puesto nuestro dominio de Angular. También se podrían especificar los métodos permitidos, pero por defecto de dejan todos.

El método Listar alumnos, lo dejamos como estaba.

```
@GetMapping("/alumnos")
	public List<Alumno> index(){
		return alumnoService.findAll();
	}
```

El método verAlumno, le cambiamos sustancialmente, quedando así:

```
@GetMapping("/alumnos/{id}")
	public ResponseEntity<?> verAlumno(@PathVariable Long id) {

		Map<String, Object> response = new HashMap<>();

		Alumno alumno = null;

		try {
			alumno = alumnoService.findById(id);
		} catch (DataAccessException e) {
			response.put("mensaje", "Error en el servidor = " +
e.getMostSpecificCause());
			return new ResponseEntity<Map<String, Object>>( response,
HttpStatus.INTERNAL_SERVER_ERROR);
		}

		if ( alumno == null ) {
			response.put("mensaje", "No existe el alumno con id = " +
id.toString());
			return new ResponseEntity<Map<String, Object>>( response,
HttpStatus.NOT_FOUND);
		}

		response.put("mensaje", "ok");
		response.put("alumno", alumno);

		return new ResponseEntity<Map<String, Object>>(response, HttpStatus.OK);
```

```
    }
```

En este método hemos hecho varias cosas.
Lo primero es que ya no devuelve un objeto Alumno, si no que devuelve un objeto ResponseEntity, que nos permite devolver "cualquier cosa".

Entonces, hemos puesto que devolvemos un Objeto <?> tipo ResponseEntity para devolver cualquier cosa.

Hemos creado un mapa de java que tiene un par de valores String, Objet.

Luego, hemos buscado al alumno en el try / cacth y puede pasar que la base de datos no responda y nos de un error en el servidor y por lo tanto, devolvemos un mensaje con el error que devuelve el servidor y si el servidor funciona, puede ser que alumno no exista, entonces devolvemos un mensaje diciendo que el alumno no se encuentra.

En ambos casos, devolveremos también el httpStatus con el código correspondiente. O un error de servidor o que no se encuentra el recurso solicitado.

Si todo ha ido bien, ponemos en nuestro mapa de java un mensaje de Ok y al alumno encontrado.

Por último, devolvemos el mapa de java y el código de respuesta http Ok (código 200).

Los demás métodos son semejantes, quedando la clase esta clase de la forma:

```
package com.jccm.backend.angular.controllers;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.dao.DataAccessException;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
```

```
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.jccm.backend.angular.models.entity.Alumno;
import com.jccm.backend.angular.models.services.IAlumnoService;


@CrossOrigin( origins = {"http://localhost:4200"})
@RestController
@RequestMapping("/api")
public class AlumnosRestController {

	@Autowired
	private IAlumnoService alumnoService;

	@GetMapping("/alumnos")
	public List<Alumno> index(){
		return alumnoService.findAll();
	}

	@GetMapping("/alumnos/{id}")
	public ResponseEntity<?> verAlumno(@PathVariable Long id) {

		Map<String, Object> response = new HashMap<>();

		Alumno alumno = null;

		try {
			alumno = alumnoService.findById(id);
		} catch (DataAccessException e) {
			response.put("mensaje", "Error en el servidor =
" + e.getMostSpecificCause());
			return new ResponseEntity<Map<String, Object>>(
response, HttpStatus.INTERNAL_SERVER_ERROR);
		}

		if ( alumno == null ) {
			response.put("mensaje", "No existe el alumno con
id = " + id.toString());
			return new ResponseEntity<Map<String, Object>>(
response, HttpStatus.NOT_FOUND);
		}
```

```
            response.put("mensaje", "ok");
            response.put("alumno", alumno);

            return new ResponseEntity<Map<String,
Object>>(response, HttpStatus.OK);
      }

      @PostMapping("/alumnos")
      public ResponseEntity<?> guardar(@RequestBody Alumno
alumno) {

            Map<String, Object> response = new HashMap<>();

            Alumno nuevoAlumno = null;

            try {
                  nuevoAlumno = alumnoService.save(alumno);
            } catch (DataAccessException e) {
                  response.put("mensaje", "Error al crear al
alumno. = " + e.getMostSpecificCause());
                  return new ResponseEntity<Map<String, Object>>(
response, HttpStatus.INTERNAL_SERVER_ERROR);
            }

            response.put("mensaje", "ok");
            response.put("alumno", nuevoAlumno);

            return new ResponseEntity<Map<String,
Object>>(response, HttpStatus.CREATED);

      }

      @PutMapping("/alumnos/{id}")
      public ResponseEntity<?> actualizar(@RequestBody Alumno
alumno, @PathVariable Long id) {

            Alumno alumnoActual = null;
            Alumno alumnoActualizado = null;

            Map<String, Object> response = new HashMap<>();

            try {
                  alumnoActual = alumnoService.findById(id);
```

```
                if ( alumnoActual == null ) {
                      response.put("mensaje", "Error no se
encuentra al alumno con id  = " + id.toString());
                      return new ResponseEntity<Map<String,
Object>>( response, HttpStatus.NOT_FOUND);
                }

           } catch (DataAccessException e) {
                response.put("mensaje", "Error al actualizar al
alumno con id  = " + id.toString());
                return new ResponseEntity<Map<String, Object>>(
response, HttpStatus.INTERNAL_SERVER_ERROR);
           }

           alumnoActual.setApellidos( alumno.getApellidos() );
           alumnoActual.setEmail( alumno.getEmail() );
           alumnoActual.setNombre( alumno.getNombre() );
           alumnoActual.setNumescolar(  alumno.getNumescolar());

           try {
                alumnoActualizado =
alumnoService.save(alumnoActual);
           } catch (DataAccessException e) {
                response.put("mensaje", "Error al actualizar al
u usuario = " + e.getMostSpecificCause());
                return new ResponseEntity<Map<String, Object>>(
response, HttpStatus.INTERNAL_SERVER_ERROR);
           }

           response.put("mensaje", "ok");
           response.put("alumno", alumnoActualizado);

           return new ResponseEntity<Map<String,
Object>>(response, HttpStatus.CREATED);

     }

     @DeleteMapping("/alumnos/{id}")
     public ResponseEntity<?> borrar(@PathVariable Long id) {

           Map<String, Object> response = new HashMap<>();
```

```
            try {
                  // No hace falta buscar el alumno, ya que spring
data lo busca al hacer el delete
                  alumnoService.delete(id);

            } catch (DataAccessException e) {
                  response.put("mensaje", "Error al borrar al
alumno con id  = " + id.toString());
                  return new ResponseEntity<Map<String, Object>>(
response, HttpStatus.INTERNAL_SERVER_ERROR);
            }

            response.put("mensaje", "ok");

            return new ResponseEntity<Map<String,
Object>>(response, HttpStatus.OK);

      }

}
```

Creo que ya es más o menos entendible.

Ahora es momento de levantar el servidor y probar todos los métodos con Postman.

Las pruebas son las mismas, lo que pasa es que por ejemplo en el método crear, hay que añadir el valor para el número escolar.

Por ejemplo, el crear según está, poniendo:

url: localhost:8080/api/alumnos

Tipo: Post

Body-> Raw-> Json / aplication

```
{
   "nombre": "Ethan",
   "apellidos": "Hunt",
   "email": "ethanhunt@gmail.com"
}
```

Nos dará el error:

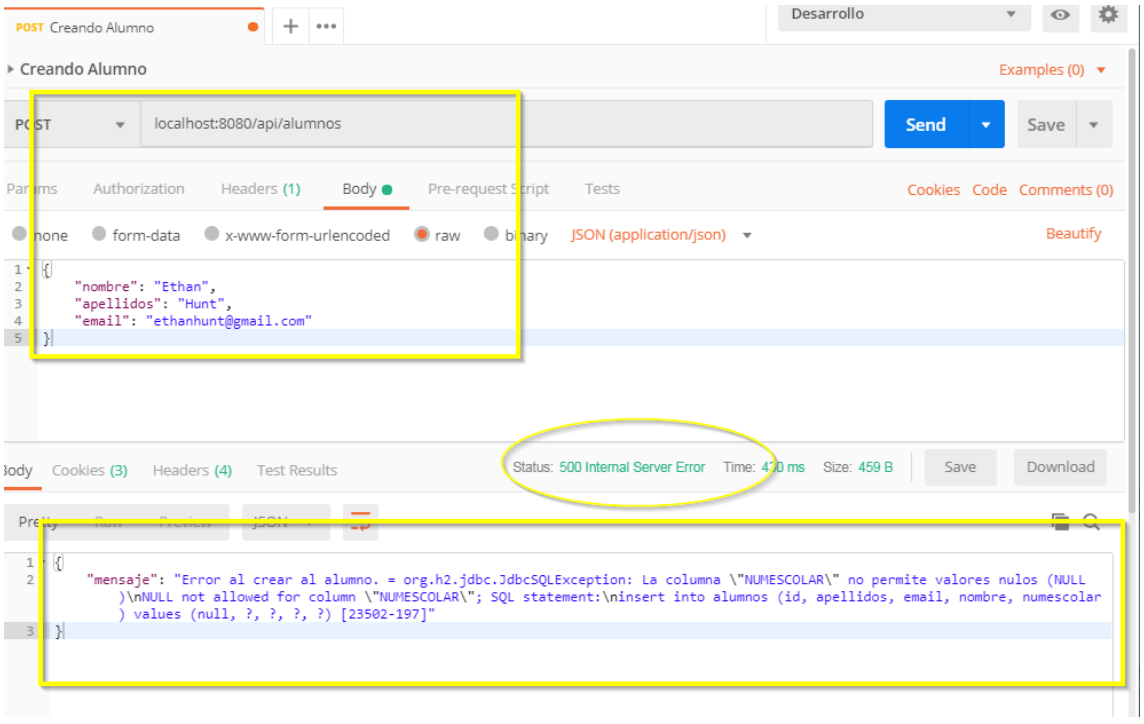
{

"mensaje": "Error al crear al alumno. = org.h2.jdbc.JdbcSQLException: La columna \"NUMESCOLAR\" no permite valores nulos (NULL)\nNULL not allowed for column \"NUMESCOLAR\"; SQL statement:\ninsert into alumnos (id, apellidos, email, nombre, numescolar) values (null, ?, ?, ?, ?) [23502-197]"

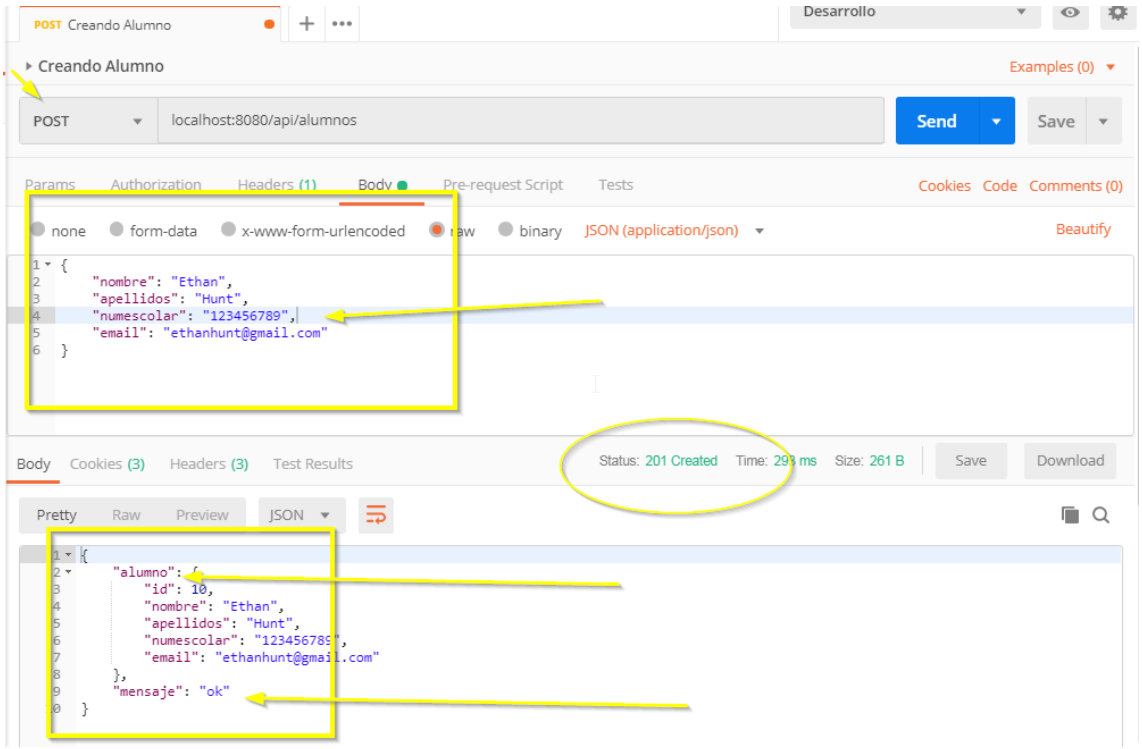
}

Y además un http response de que es un error del servidor.

Claro, no se puede insertar null en el número escolar.



Ahora, vamos a poner el campo que nos falta.



Cómo podemos ver, ahora nos devuelve un status 201, que está creado, y un Json con un objeto Alumno y con un campo mensaje = Ok. Que es lo que habíamos hecho en el backend.

Los demás métodos es probarlo y jugar un poco con cada uno de ellos.

Si tenéis alguna duda, pues me podéis mandar un correo y con gusto y si lo puedo resolver, os lo comento.

En el próximo documento, vamos incorporar unos alert con SweetAlert para controlar los errores en nuestra aplicación de Angular.

Espero que les guste.