Aunque no soy un experto en esta materia, lo único que pretendo es compartir con vosotros lo que he podido aprender. Espero no aburrir mucho.

En una primera versión de este capítulo, vamos a crear una aplicación backend con spring boot y sin seguridad, para hacer un CRUD de una entidad llamada Alumnos.

No vamos a incorporar control de errores en esta primera "versión", para hacerlo mas fácil. Luego introduciremos control de errores y seguridad.

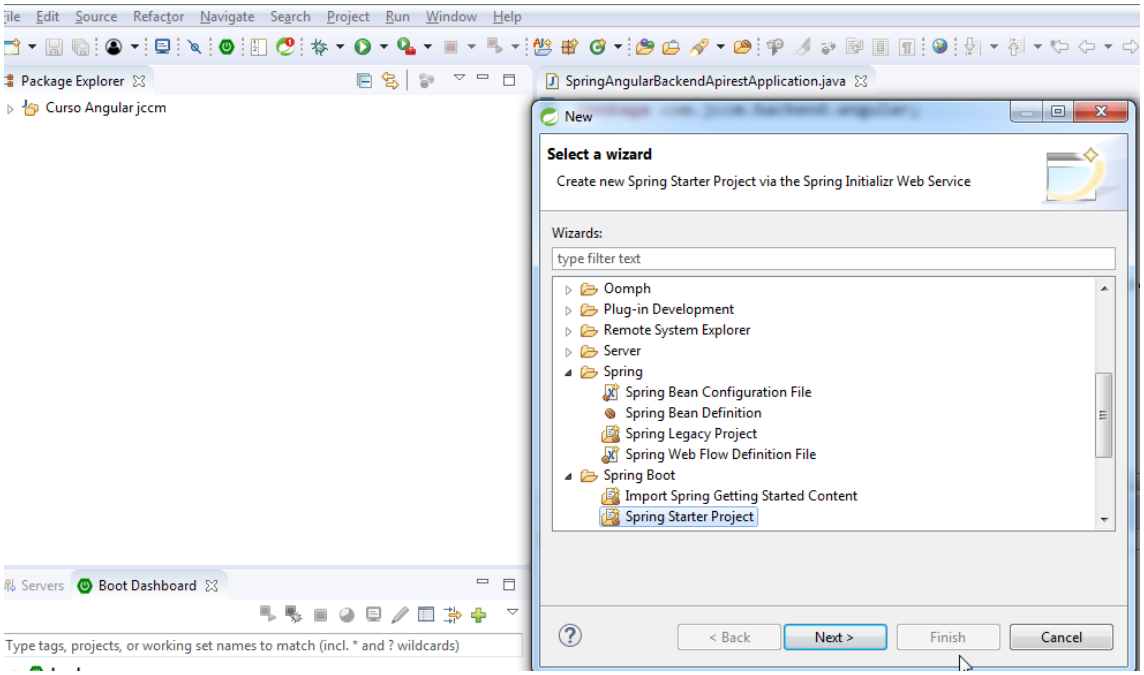
Crearemos el proyecto backend paso a paso y luego en el proyecto que tenemos de angular incorporaremos una nueva ruta donde tendremos la lista de alumnos y los botones necesarios para crear, actualizar, borrar y listar.

Creo que es un buen comienzo para la gente que quiera aprender el funcionamiento básico de estas tecnologías.

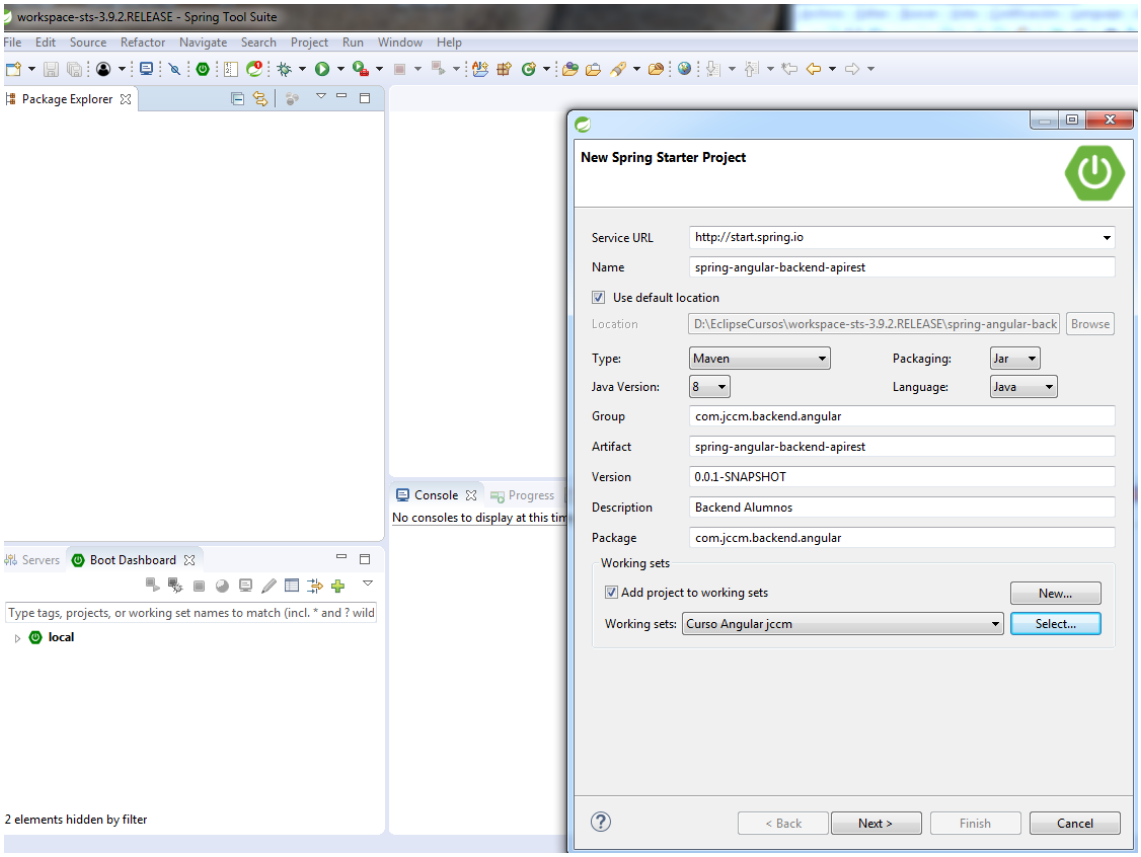
Pues manos a la obra.

1. Creación del proyecto Backend.

Abrimos el STS eclipse y vamos a File->New -> Other -> Spring Starter Project .



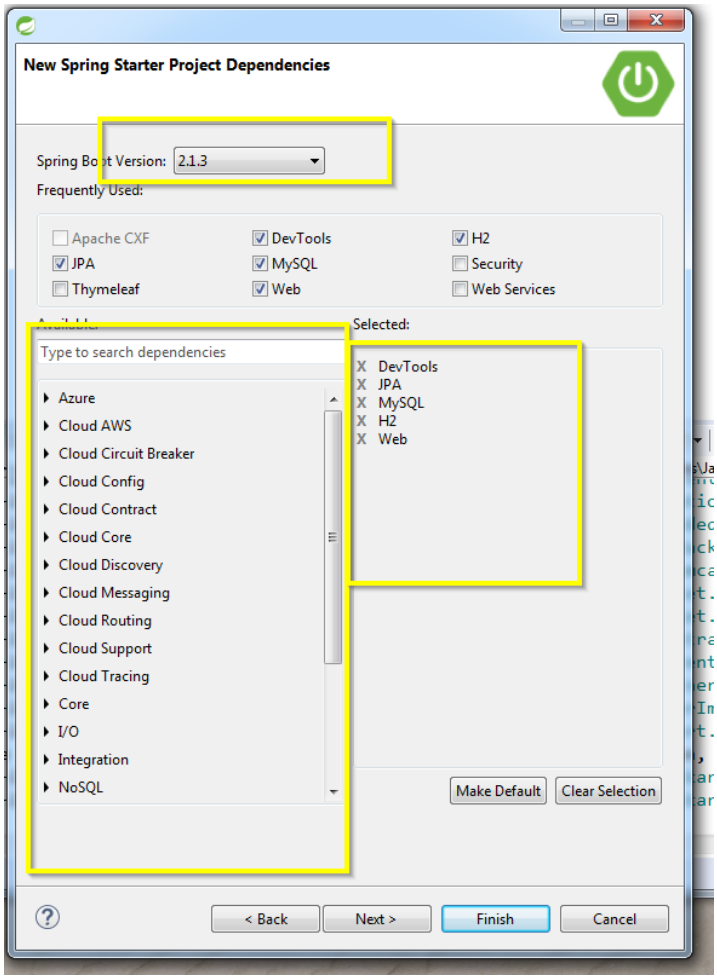
Nos saldrá un Wizard donde vamos a especificar algunas propiedades del Proyecto.



Las propiedades que hemos puesto son el nombre, que le llamamos spring-angular-backend-apirest, el tipo de proyecto, que elegimos Maven, el tipo de empaquetado para los despliegues, que al no tener vistas de tipo jsp, vamos a elegir jar. La versión de java, pues elegimos la 8 por ejemplo, aunque se podría elegir la 11 si la tienen instalada.

También configuramos el Group y el package de la aplicación por ejemplo con el nombre com.jccm.backend.angular, aunque se puede configurar con otros nombres.

Damos siguiente y nos permite elegir las dependencias del proyecto.



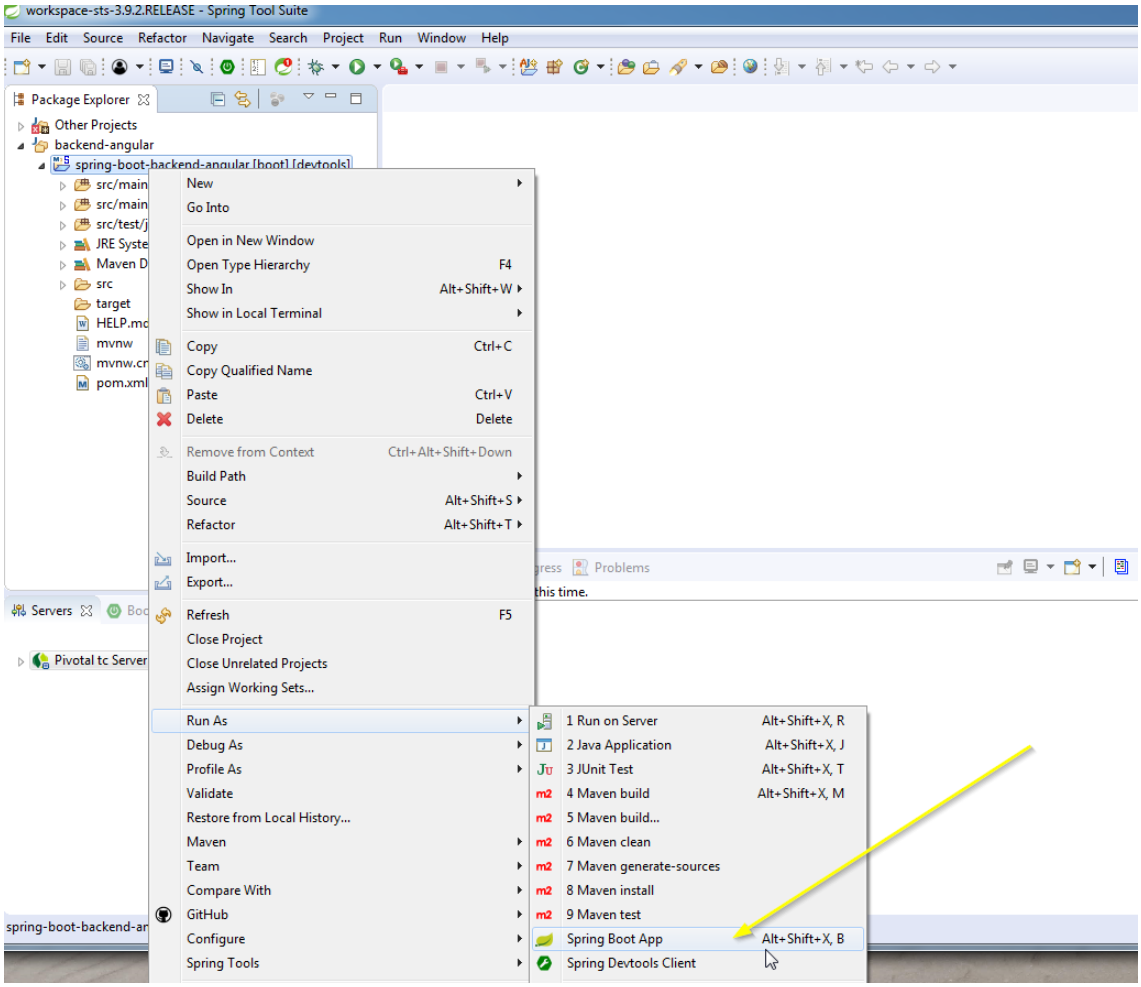
En las dependencias, lo primero que nos da a elegir es la versión de Spring Boot.

Se recomienda elegir la última estable de la versión 2.x, ya que estas son las que utilizan spring framework 5 y las versiones 1.x, utilizan spring framework 4.

Luego están las dependencias que vamos a elegir y que están en el cuadro de la derecha, en estos, elegimos JPA, para poder trabajar con clases mapeadas a tablas de la base de datos, elegiremos DevTools, que es una dependencia que en tiempo de desarrollo, nos permite que vaya reiniciando cada vez que hacemos algún cambio, también vamos a elegir las dependencias de las bases de datos H2, que es la que vamos a utilizar y la de MySql por si lo usamos. Por último utilizaremos la dependencia Web, que incluye las anotaciones del tipo @GetMapping o @RestController, que son las encargadas de responder las peticiones http.

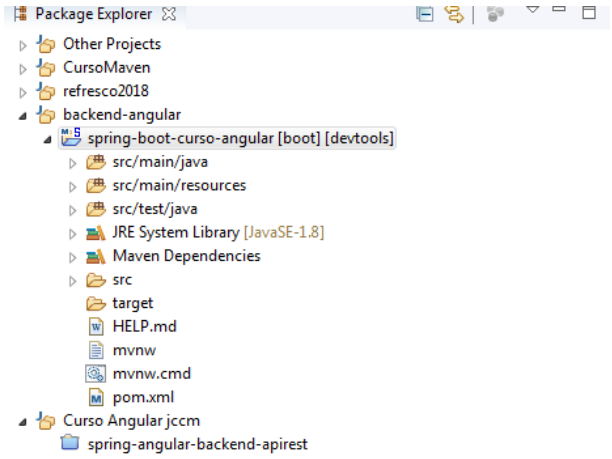
Damos finalizar y finalizar y ya tenemos nuestro proyecto.

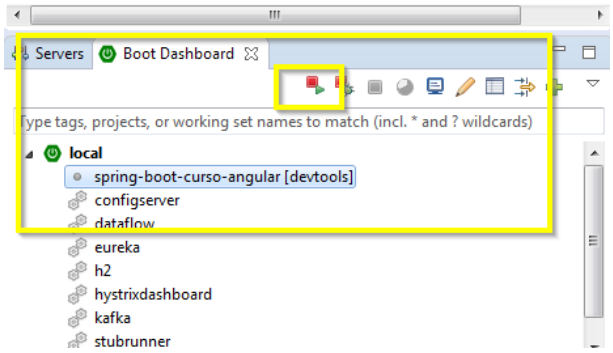
Deberíamos poder probarlo para ver que ha ido bien dando sobre el proyecto con el botón derecho y run as spring boot app



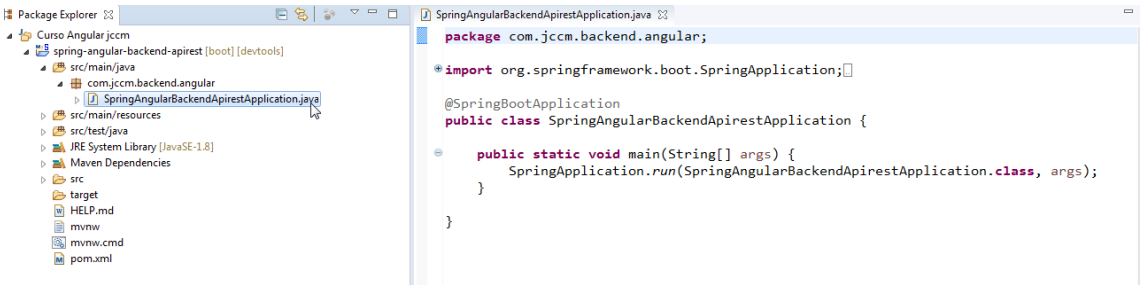
Lo normal es que no de errores y si estamos sin errores, es que vamos por buen camino.

También podemos desplegar con el acceso directo en el Boot Dashboard





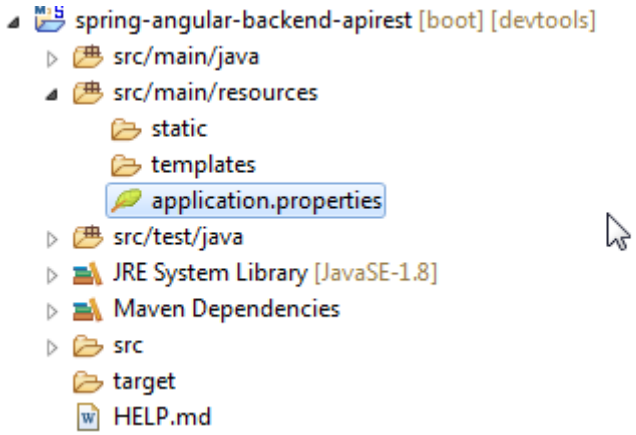
Esta es la clase principal del proyecto, que es la encargada de levantar el servidor apache-tomcat que lleva embebido



El siguiente paso es crear la configuración de la base de datos que vamos a utilizar.

De momento vamos a configurar la base de datos H2, que viene con java.

Para ello, abrimos el ficheros application.resources que está en la carpeta src/main/resources

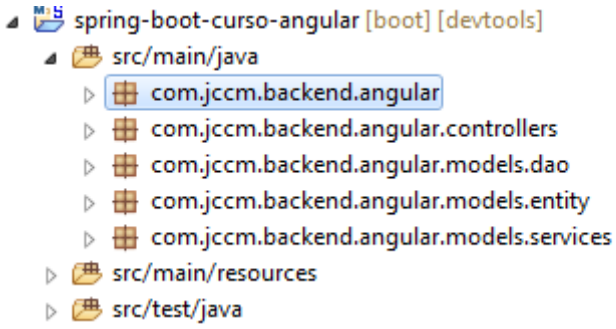


Y pegamos este código

```
# Configuración del datasource
spring.datasource.url=jdbc:h2:mem:alumnosdb
spring.datasource.username=user
spring.datasource.password=1234
spring.datasource.driver-class-name=org.h2.Driver
spring.h2.console.enabled=true

spring.jpa.hibernate.ddl-auto=create-drop
logging.level.org.hibernate.SQL=debug
```

En el que le decimos la url de la base de datos, las credenciales, el driver, que active la consola, que cuando se inicie el proyecto y cuando se baje, que se creen y destruyan los objetos de la base de datos y además, que muestre las sentencias SQL que ejecuta.

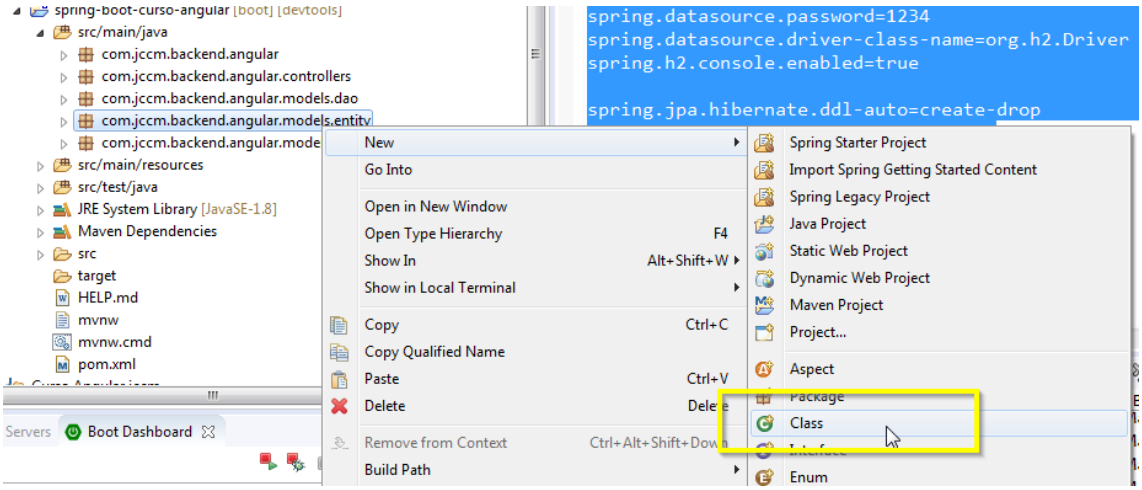
Después de esto, vamos a crear la estructura siguiente de paquetes, para poner en cada uno de ellos la capa correspondiente del proyecto. Por un lado las entidades, por otro la capa de acceso a datos, por otro lado la capa de servicios y por último los controladores que son los que interactúan con las peticiones http y el modelo.



Para crear estos paquetes es simple. Nos ponemos en el paquete principal y con File-> New->Package vamos creando los paquetes para que nos queden como en la imagen superior.

Ahora vamos a codificar.

1. Creando la primera Entity. Será una clase Alumno que estará mapeada a una tabla de la base de datos gracias a Jpa.
   Sobre el paquete ….entity, damos botón derecho y New->Class



   Saldrá un Wizard y llamaremos a esa Clase Alumno

   La clase alumno, tendrá de momento 4 propiedades que serán el Id, el nombre, apellidos y email.
   De forma, que quedará así:

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

public class Alumno {


	private Long id;

	private String nombre;

	private String apellidos;

	private String email;



}
```

Ahora, vamos a preparar esta clase, para que pueda ser mapeada.
Lo primero es implementar el Serializable, además anotar la clase con @Entity para decir al motor de persistencia que es un clase entidad. Luego, la anotamos con @Table y el nombre de la tabla a la que corresponderá en la base de datos.
Anotamos también el Id, para informar que será nuestro campo clave y la estrategia de generación de código que tiene que ser utilizada.

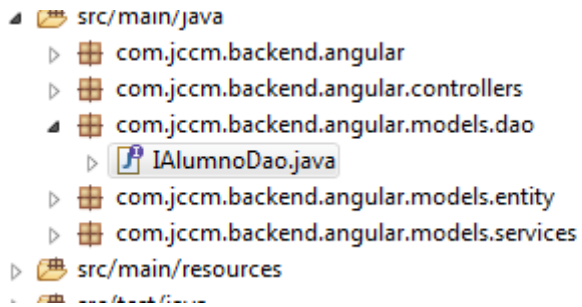
Luego, dando al botón derecho en la misma clase y dando Source -> Generate Getters y Setters, generaremos los get y set de las propiedades de la clase, quedando por lo tanto de la forma:

```
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;

	public Long getId() {
		return id;
	}

	public void setId(Long id) {
		this.id = id;
	}

	public String getNombre() {
		return nombre;
	}

	public void setNombre(String nombre) {
		this.nombre = nombre;
	}

	public String getApellidos() {
		return apellidos;
	}

	public void setApellidos(String apellidos) {
		this.apellidos = apellidos;
	}

	public String getEmail() {
		return email;
	}

	public void setEmail(String email) {
		this.email = email;
	}


}
```

El siguiente paso, es crear el acceso a datos, El dao o el repositorio de datos.
Para ello, en el paquete dao, vamos a crear una interfaz en la extenderemos de la interfaz CrudRepository, que nos ofrece ya spring data jpa y que trae los métodos necesarios básicos para crear, borrar, modificar, etc.

Crearmos la interfaz IAlumnoDao en el paquete ….dao



Luego en el código, tendremos esto
package com.jccm.backend.angular.models.dao;

import org.springframework.data.repository.CrudRepository;

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

public interface IAlumnoDao extends CrudRepository<Alumno, Long> {

}

Aquí le decimos que extienda de CrudRepository y que tiene que utilizar la entidad Alumno y que su campo clave es un Long.

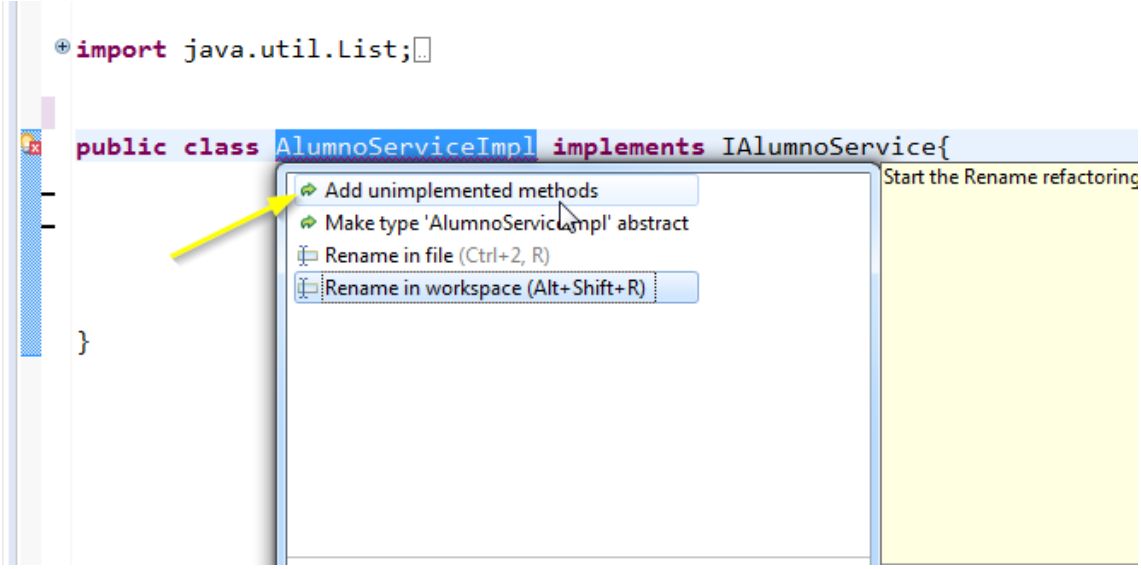
(para importar los paquetes en eclipse, se pulsa CTRL+Shift+O)

El siguiente paso, es crear el servicio que utilizará el controlador, para ello, en el paquete ….services, creamos la interfaz con las firmas de los métodos que tendrá la implementación del servicio. Llamaremos a la interfaz IAlumnoService y tendrá el siguiente contenido.

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

import java.util.List;

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

public interface IAlumnoService {

	public List<Alumno> findAll();
```

```
public Alumno save ( Alumno alumno );

public void delete ( Long id );

public Alumno findById(Long id);

}
```

Después de esto en la mismo paquete, creamos la implementación de esta interfaz. Para ello, creamos una clase con el nombre AlumnoServiceImpl e implementamos la interfaz IAlumnoService. Al poner que implementa la interfaz, nos muestra un error



Y al pulsar, nos dice que implementemos los métodos que están sin implementar.
Una vez que los escribe por nosotros, los tenemos que sobreescribir con el siguiente código que ahora explico:

```
@Service
public class AlumnoServiceImpl implements IAlumnoService{

	@Autowired
	private IAlumnoDao alumnoDao;


	@Override
	@Transactional(readOnly=true)
	public List<Alumno> findAll() {
		return (List<Alumno>) alumnoDao.findAll();
	}


	@Override
	@Transactional
	public Alumno save(Alumno alumno) {
		return alumnoDao.save(alumno);
	}


	@Override
	@Transactional
	public void delete(Long id) {
		alumnoDao.deleteById(id);

	}


	@Override
	@Transactional(readOnly=true)
	public Alumno findById(Long id) {
		return alumnoDao.findById(id).orElse(null);
	}



	}
```

Hemos anotado con @Service, para informar al contenedor de spring que es un componente de tipo servicio.
Además hemos anotado como @Transactional los métodos de la clase diferenciando los que son de solo lectura.
También hemos inyectado el Dao/Repositorio anotando con @Autowired, para que busque una implementación de la interfaz IAlumnoDao.

Los métodos que nos ofrece el dao, ya vienen predefinidos en CrudRepository, por eso es tan simple su utilización , por ejemplo utilizar el findById, save, etc.
Así, tenemos el servicio y por último creamos el controlador.

En el paquete ….. controllers, creamos una clase que la llamaremos por ejemplo AlumnosRestController

Debe quedar así y ahora explico lo mas importante de cada método.

```
@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);
    }
```

Cómo se puede ver, es una clase, que hemos anotado con la anotación @RestController y además con @RequestMapping ("/api"). De esta forma, todas las peticiones http rest, pasarán por /api.

Con @Autowired, hemos inyectado el servicio de Alumnos.

El primer método es el index. En este médoto se devuelve una lista de objetos de tipo alumnos. Ese método está anotado con @GetMapping("/alumnos"), para informar que será una petición de tipo Get y que será a la url /api/alumnos.

El segundo método es VerAlumno. Como puede verse, este método devuelve un objeto de tipo Alumno. Para ello, le hemos anotado con @GetMapping("/alumnos({id}"), para decirle que será una petición de tipo get y que además será a la url /api/alumnos/id, donde el id, será el id del alumno y que recoge el método con @PathVariable Long id (ojo que se tiene que llamar igual el id del método como el de la configuración del @GetMapping). Una vez que se tiene el id, con el método findById, devolvemos el Alumno.

El tercer método es el guardar. Como puede verse, devolverá un objeto de tipo Alumno que será el que se ha guardado en la base de datos. Le anotamos con @PostMapping, pero además, le anotamos con @ResponseStatus para cuando termine de ejecutarse el método devuelva ese estado. Esto lo vamos a probar pronto con PostMan.

El cuarto método es actualizar. Vemos que está mapeado también a la url "alumno/{id}" al igual que el método de ver un alumno pero está anotado con @PutMappig, para informar que es esa url pero con el verbo Put.

El último método es borrar, que no devuelve nada y lo único que hace es borrar el alumno que viene por el parámetro id. Está anotado con @DeleteMapping.

Antes de continuar, vamos a poblar nuestra base de datos con unos datos de pruebas, para no tener que estar grabando datos.

Para ello, nos vamos a la carpeta src/main/resources y creamos un fichero llamado import.sql (tiene que llamarse obligatoriamente así para que importe los datos)

Incorporamos al archivo import.sql este contenido:

```
/* Unos pocos datos de alumnos */
INSERT INTO alumnos (nombre, apellidos, email ) VALUES('Juan', 'Pérez de Tudela', 'juanperez@gmail.com');
INSERT INTO alumnos (nombre, apellidos, email ) VALUES('Alicia', 'Sánchez Gómez', 'alicia@gmail.com');
INSERT INTO alumnos (nombre, apellidos, email ) VALUES('Hernán', 'Cortés de Monroy',
'hernancortes@gmail.com');
INSERT INTO alumnos (nombre, apellidos, email ) VALUES('Carlos', 'García', 'carlitos@gmail.com');
```

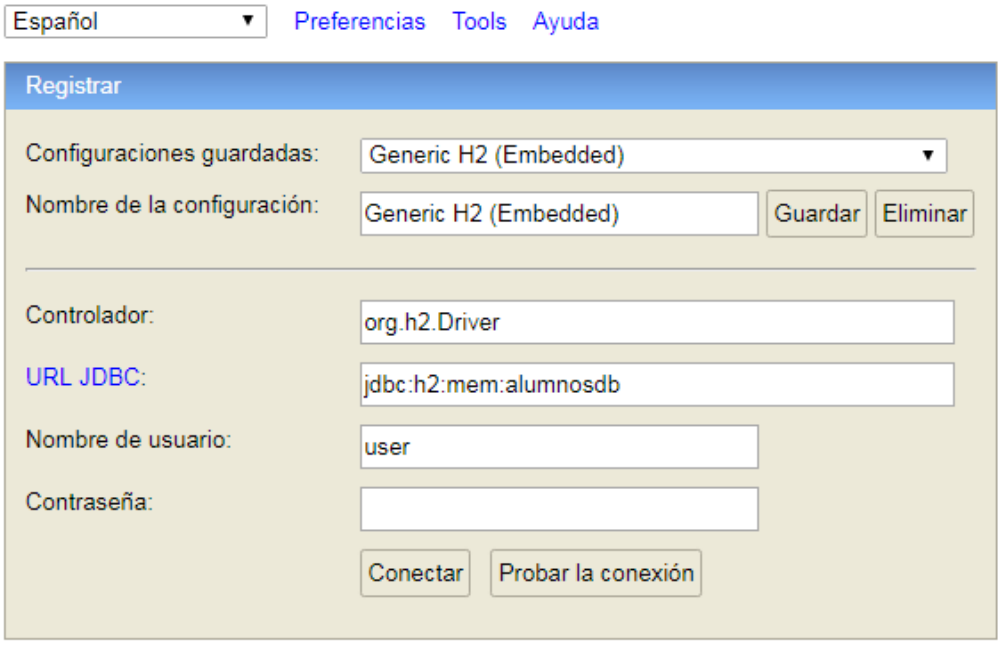
Llegó la hora, vamos a probar el backend.

Primero desplegamos y nos fijamos en la consola que no tenemos errores.

Antes de ir a postman, podemos ver si se han grabado los datos en la base de datos.

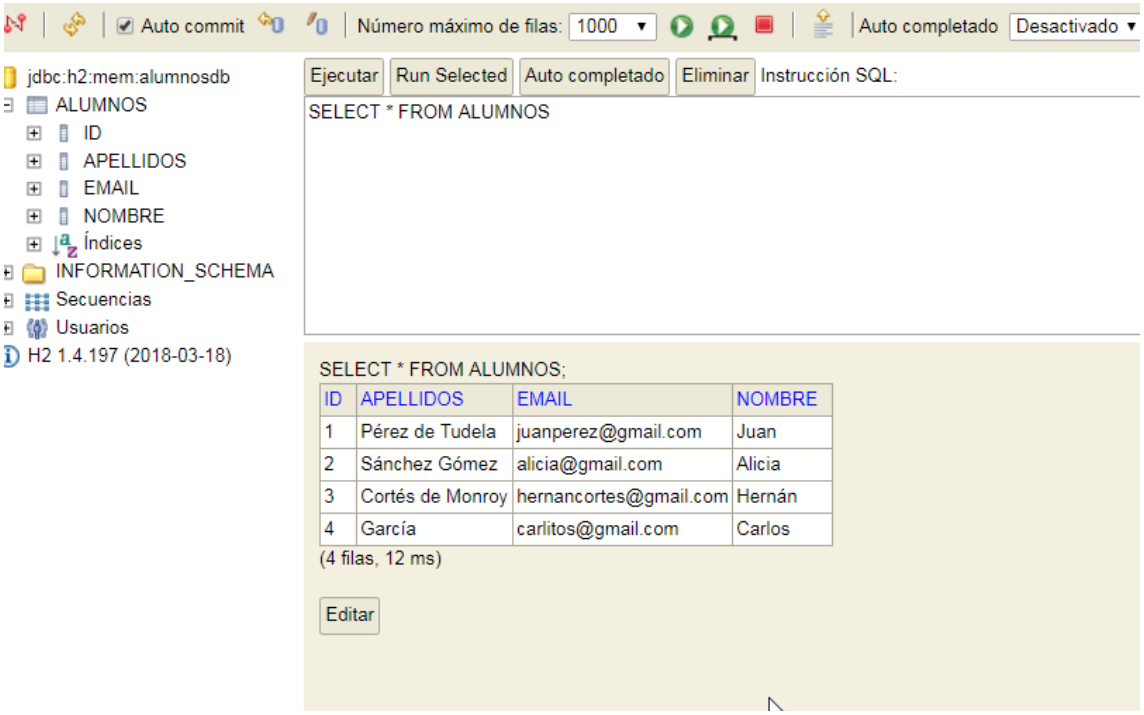
Para ello y con nuestro servidor levantado, vamos a la url: http://localhost:8080/h2-console

Y nos sale una ventana como esta



Donde ponemos la url de la base de datos, el nombre de usuario y la clave que pusimos que era 1234

Ya ponemos ver el dashboard y comprobar que efectivamente están los alumnos que hemos incorporado.



Ahora sí, vamos a POSTMAN

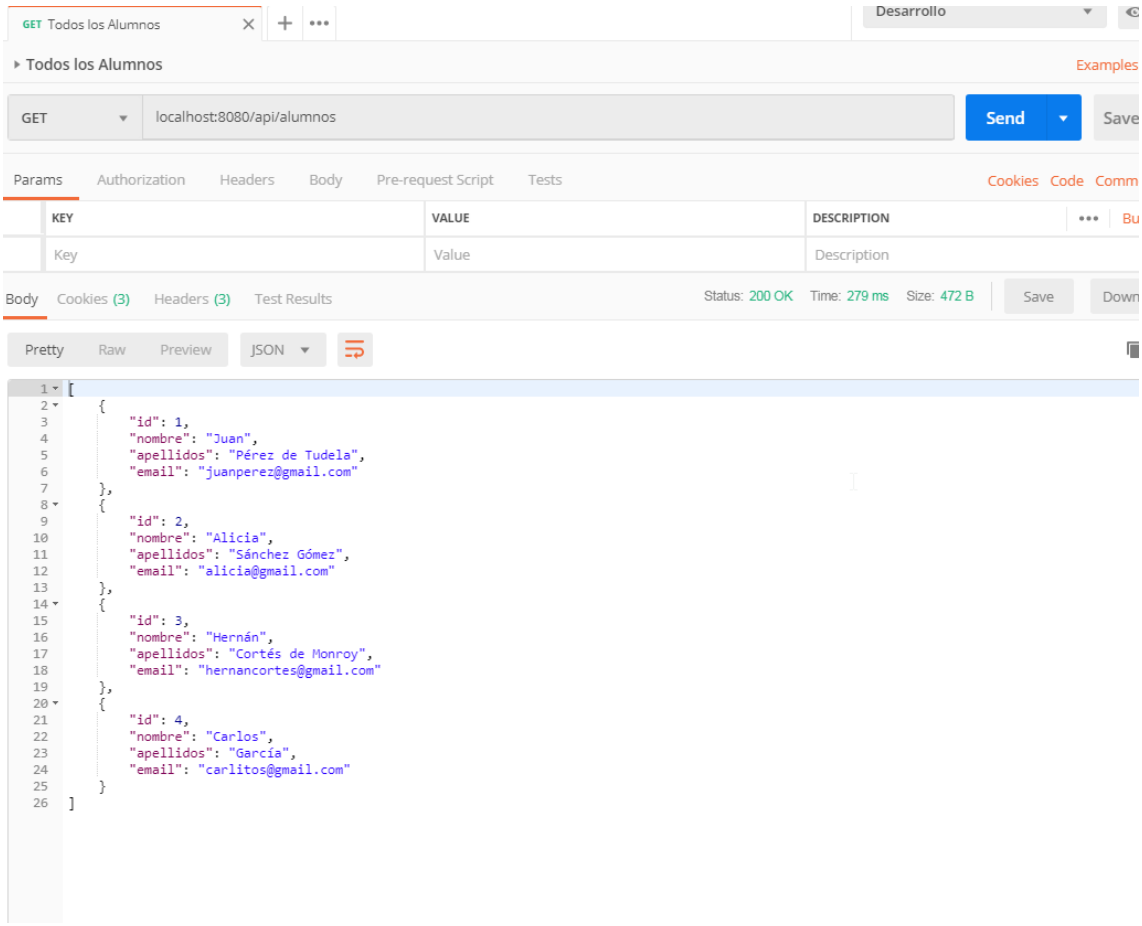
Con el servidor levantado, ejecutamos postman y vamos a ir probando cada uno de los métodos de nuestro servicio rest.

Esta es la ventana de postman donde vamos a poder poner la url del servicio, el verbo utilizado y el botón enviar.

Probamos primero el método para obtener todos los alumnos y que es con el verbo GET.

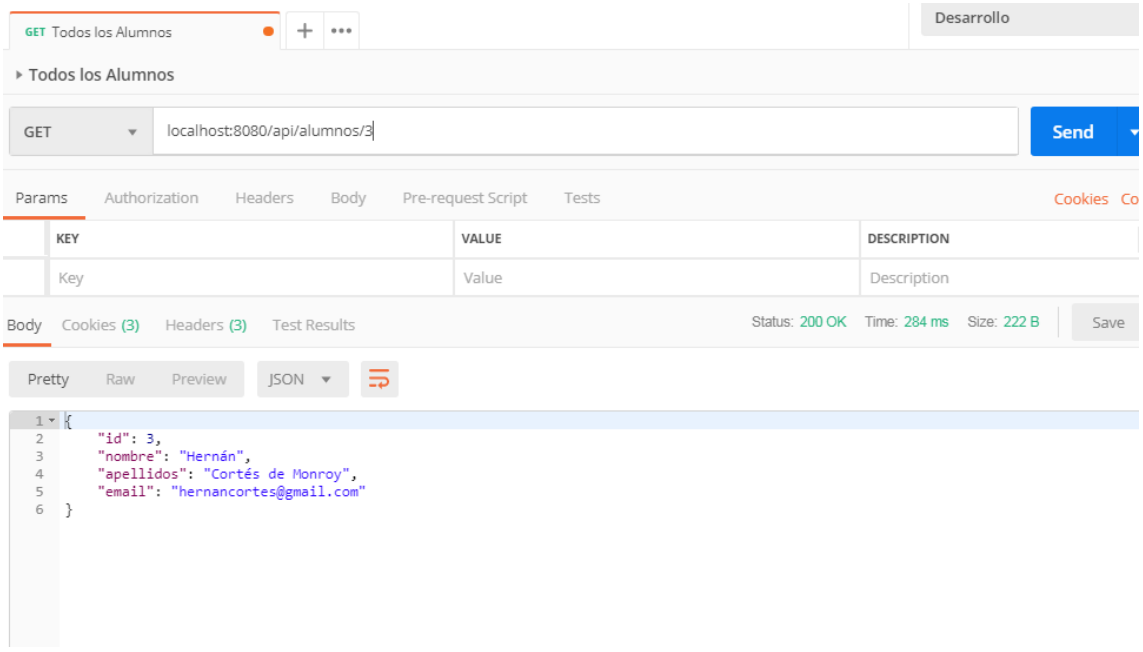
Para ello, ponemos en la url localhost:8080/api/alumnos y el verbo Get y pulsamos Send

Nos tiene que salir un json con todos los alumnos:



Probamos el obtener un alumno.

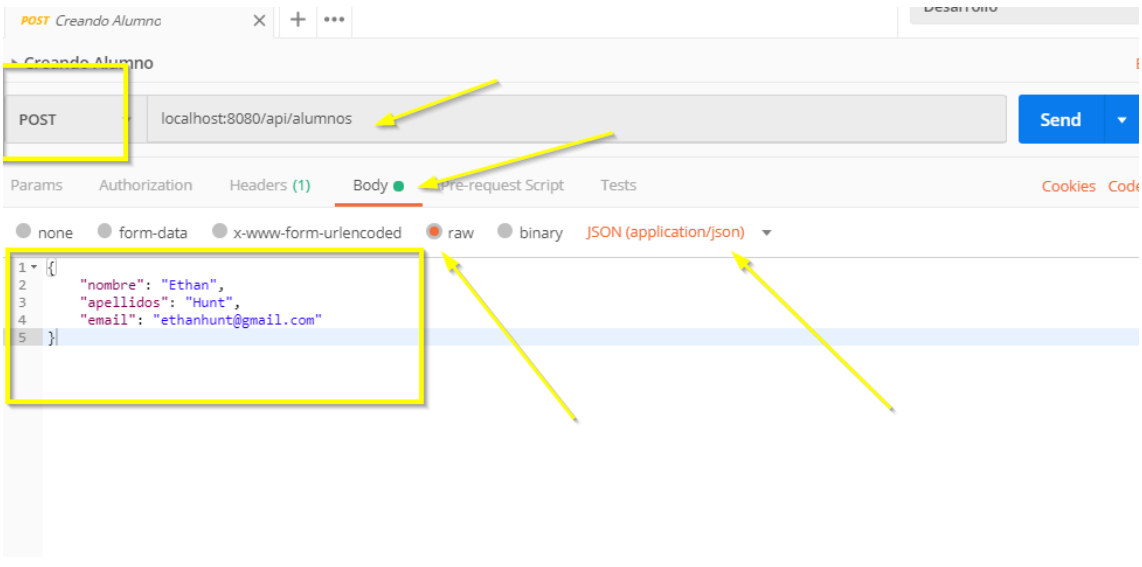
Por ejemplo para el alumno número 3 será: url localhost:8080/api/alumnos/3 y el verbo Get y pulsamos Send.



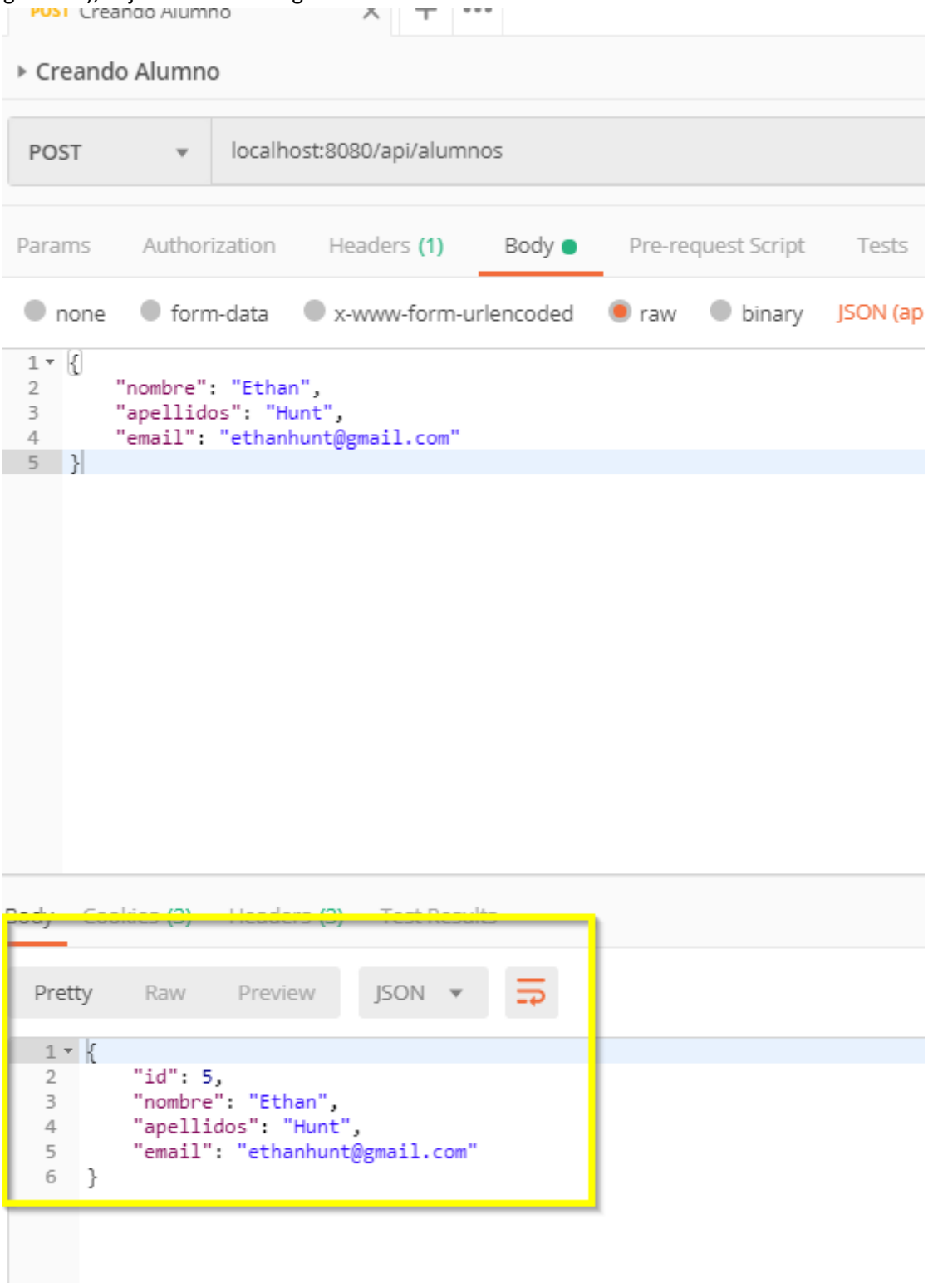
Probamos ahora el método crear.

Para ello:

1. Utilizamos el verbo POST
2. La url es: localhost:8080/api/alumnos y ahora, vamos a enviar el alumno nuevo dando a body, raw y tipo json . Luego en la ventana, ponemos el json del alumno nuevo, así:

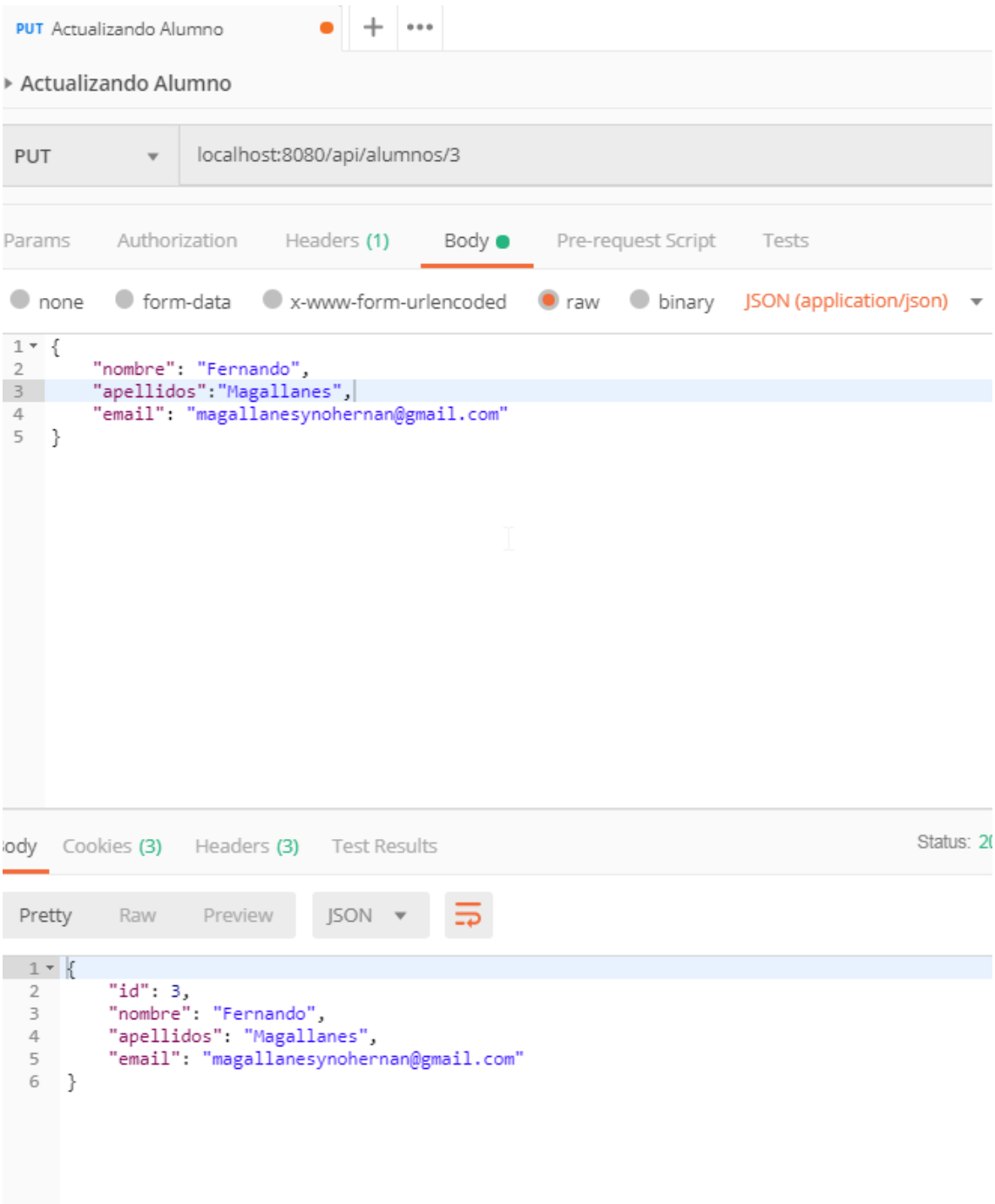


Cuando damos a enviar, nos devuelve (como estaba configurado en el método guardar), el json del alumno guardado.

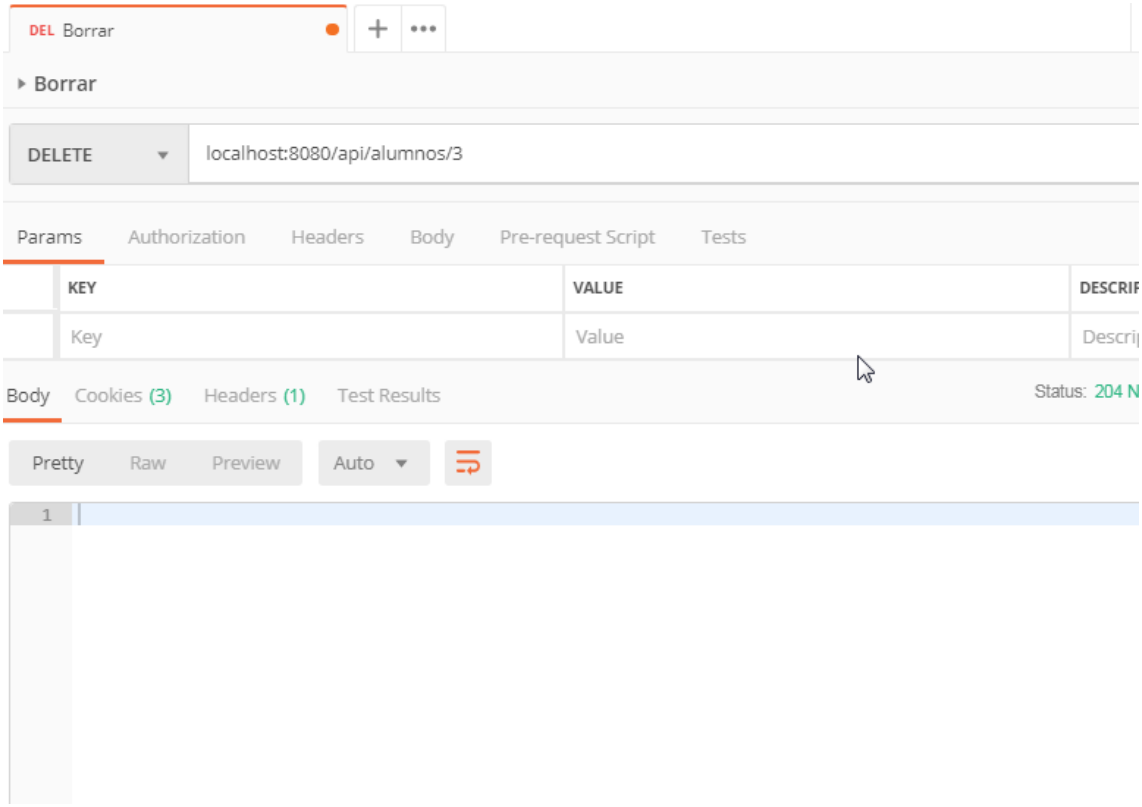


Si ahora probamos el método obtener todos los alumnos, pues veremos que está este nuevo.

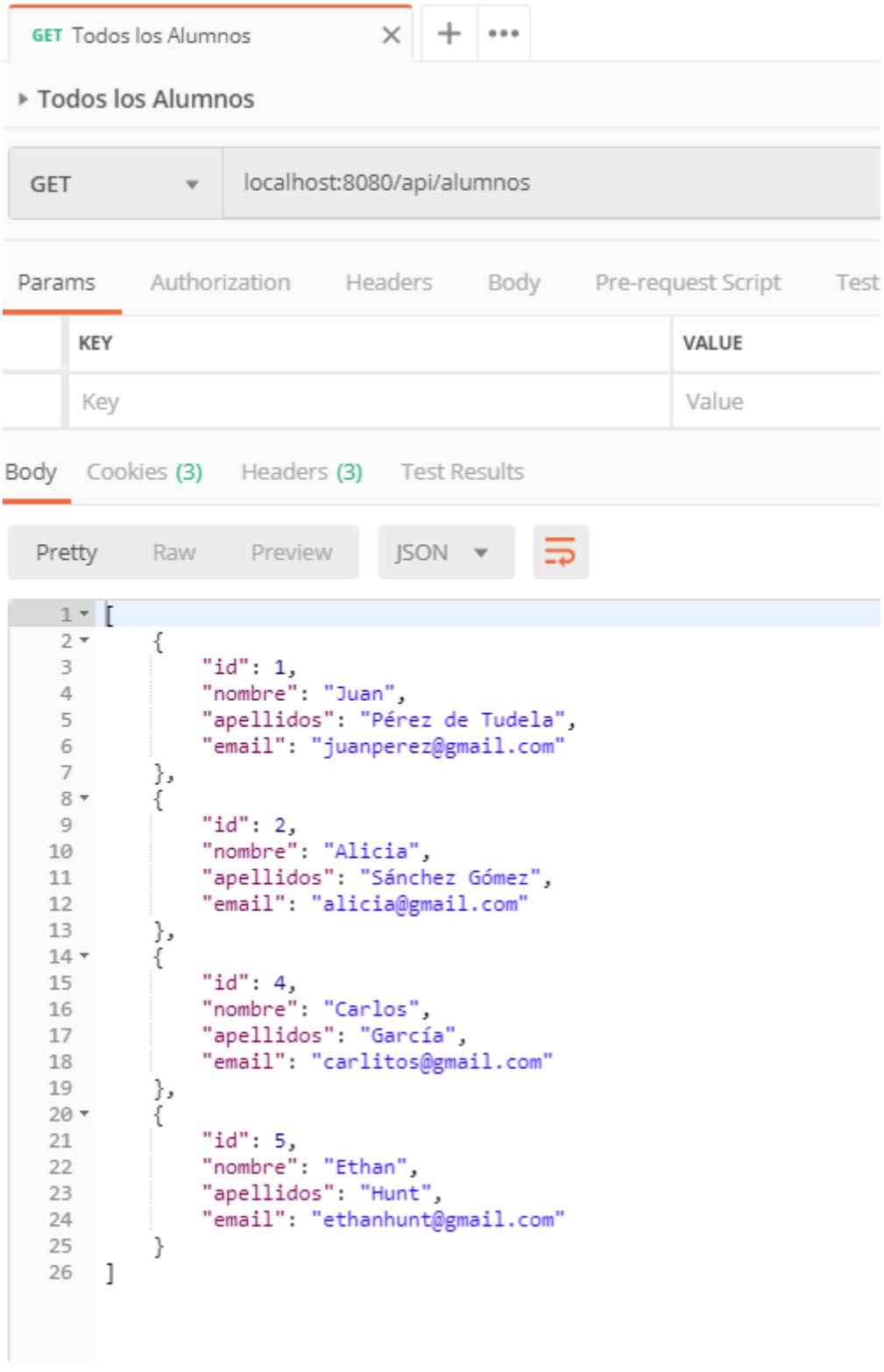
Probamos ahora el método actualizar.
Es muy parecido, el verbo es PUT, la url es parecida, ya que ponemos el id del alumno que vamos a modificar y además, al igual que el POST, vamos a poner en el body, los datos del alumno modificado, así:



Para borrar, es muy simple, el verbo es DELETE y la url es /alumnos/el número, por ejemplo para borrar al alumno 3, será:



No devuelve nada, pero si miramos si existe este alumno en todos los alumnos, por ejemplo, pues no sale.


GET Todos los Alumnos
Todos los Alumnos
GET
localhost:8080/api/alumnos
Params
Authorization
Headers
Body
Pre-request Script
Test
KEY
VALUE
Key
Value
Body
Cookies (3)
Headers (3)
Test Results
Pretty
Raw
Preview
JSON
[
{
"id": 1,
"nombre": "Juan",
"apellidos": "Pérez de Tudela",
"email": "juanperez@gmail.com"
},
{
"id": 2,
"nombre": "Alicia",
"apellidos": "Sánchez Gómez",
"email": "alicia@gmail.com"
},
{
"id": 4,
"nombre": "Carlos",
"apellidos": "García",
"email": "carlitos@gmail.com"
},
{
"id": 5,
"nombre": "Ethan",
"apellidos": "Hunt",
"email": "ethanhunt@gmail.com"
}
]

Más adelante, modificaremos todos estos métodos para incorporar el control de errores, para informar si existe o no el alumno a borrar, a modificar, etc…