

Vamos primero a ver como he hecho el código del controller y del back de springboot.

```
6
7  @RestController
8  @RequestMapping("/api")
9
0  public class MainController {
1
2      public ArrayList<Moroso> lista = new ArrayList<>();
3
4
5      @GetMapping("/principal")
6      public ArrayList<Moroso> obtenerLista() {
7          return lista;
8      }
9
0
1      @PostMapping("/add")
2      public void addMoroso(@RequestBody Moroso moroso){
3          lista.add(moroso);
4      }
5
6      @DeleteMapping("/remove/{id}")
7      public void removeMoroso(@PathVariable("id") String id){
8          for (Moroso moroso : lista) {
9              if(moroso.getId().equals(id)){
10                  lista.remove(moroso);
11              }
12          }
13      }
14 }
```

@RestController porque el postman se supone que está en otro dominio distinto al del back.

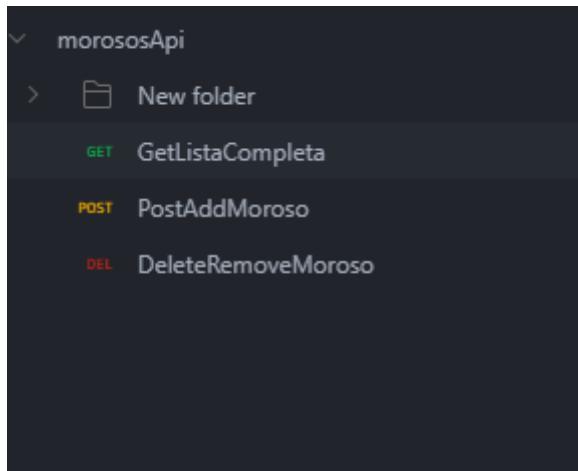
@RequestMapping(/api) para indicar cual es el endpoint principal de mi servidor.

Como es un ejercicio de prueba, ya se que no debería de ser así, esta vez no he definido un @Service, no es indispensable para realizar las pruebas, así que lo que he hecho es definir primero el arrayList dentro del controller para almacenar los objetos con los que vamos a trabajar.

Con `@GetMapping(/principal)` indicamos el endpoint la que nos vamos a conectar al hacer peticiones cuando hacemos un get. Nos devuelve el array completo.

con `@PostMapping(/add)` marco el endpoint al que postman se tiene que dirigir al hacer peticiones Post.

`@DeleteMapping(/remove/{id})` marcamos el endpoint de eliminar. Con id pasamos como parámetro la id del objeto que queremos eliminar.



En postman debemos de crear las peticiones con las que vamos a trabajar. Primero ya hemos definidos nuestras variables de enviroment para que pueda ir dirigiéndose a los endpoint necesarios.

Primero vamos a hacer un get para ver que obtenemos.

A screenshot of the Postman request interface. At the top, it shows a GET request to `((baseUrl))((morososPath))/principal`. The 'Body' tab is selected, and the raw JSON response is shown as an empty object: `{}`. Below the request, the response pane shows a status of 200 OK with a time of 74 ms and a size of 166 B. The response body is also empty. The bottom navigation bar includes tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL, PORTS, and POSTMAN CONSOLE.

Este es el resultado. Status 200 ok pero no devuelve nada porque no hay nada.

Hay que hacer primero un post.

```

POST [baseURL][insecurePath]/add
Params Authorization Headers (9) Body Scripts Settings
none form-data x-www-form-urlencoded raw binary GraphQL JSON
1
2   "id": 1,
3   "nombre": "Pepe Pérez",
4   "dni": "23456789B",
5   "email": "luis@example.com",
6   "telefono": "600333444",
7   "importe": "300.0",
8   "concepto": "Servicio impagado"
9 }

```

Body Cookies Headers (4) Test Results

Pretty Raw Preview Text ⌂

Status: 200 OK Time: 65 ms Size: 123 B

Hacemos primero el post de un objeto JSON y vemos la respuesta, 200 ok.

Ahora volvemos a hacer un GET para comprobar que se ha guardado en nuestro array.

```

Params Authorization Headers (5) Test Results
none form-data x-www-form-urlencoded raw binary GraphQL JSON
1

```

Body Cookies Headers (5) Test Results

Pretty Raw Preview JSON ⌂

Status: 200 OK Time: 12 ms Size: 316 B

```

[{"id": "1", "nombre": "Pepe Pérez", "dni": "23456789B", "email": "luis@example.com", "telefono": "600333444", "importe": "300.0", "concepto": "Servicio impagado"}]

```

El nuevo GET da un resultado de OK y además nos responde con el mismo objeto que tenemos. Tendremos que hacer más POST para las siguientes partes.

```

Params Authorization Headers (9) Body Scripts Settings
none form-data x-www-form-urlencoded raw binary GraphQL JSON
1
2   "id": 2,
3   "nombre": "Lucía Gómez",
4   "dni": "12345678B",
5   "email": "lucia@example.com",
6   "telefono": "60011123",
7   "importe": "225.5",
8   "concepto": "Alquiler"
9 }

```

Body Cookies Headers (4) Test Results

Pretty Raw Preview Text ⌂

Status: 200 OK Time: 9 ms Size: 123 B

Nuevo Post para añadir un nuevo elemento.

```

1
[{"id":1,"nombre":"Pepe Pérez","dni":"23456789B","email":"luis@example.com","telefono":"600333444","importe":"300.0","concepto":"Servicio impagado"}, {"id":2,"nombre":"Lucía Gómez","dni":"12345678B","email":"lucia@example.com","telefono":"600111223","importe":"225.5","concepto":"Alquiler"}]

```

Al hacer el GET tenemos más elementos en la colección. Ahora hay que eliminar 1 para comprobar que funciona el elemento DELETE

```

1
{
  "id": 1,
  "nombre": "Pepe Pérez",
  "dni": "23456789B",
  "email": "luis@example.com",
  "telefono": "600333444",
  "importe": "300.0",
  "concepto": "Servicio impagado"
}

```

Probamos eliminando el primer objeto, que así no hay sospechas de que pueda usar la foto del primer GET. Ahora al hacer un nuevo GET debería de salirnos el segundo elemento de nuestra colección.

GET {{baseUrl}}/{{morosoPath}}/principal

Params Authorization Headers (7) Body Scripts Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

Send | Send with file | Code Cookies | Beautify

1

Body Cookies Headers (5) Test Results

Pretty Raw Preview JSON ↻

1 [{"id": "2", "nombre": "Lucía Gómez", "dni": "123456788", "email": "lucia@example.com", "telefono": "600111223", "importe": "225.5", "concepto": "Alquiler"}]

Status: 200 OK Time: 16 ms Size: 310 B

Body Cookies Headers (5) Test Results

Pretty Raw Preview JSON ↻

1 [{"id": "2", "nombre": "Lucía Gómez", "dni": "123456788", "email": "lucia@example.com", "telefono": "600111223", "importe": "225.5", "concepto": "Alquiler"}]

Status: 200 OK Time: 16 ms Size: 310 B

Como vemos, con el nuevo GET, obtenemos el segundo elemento de nuestra colección.