

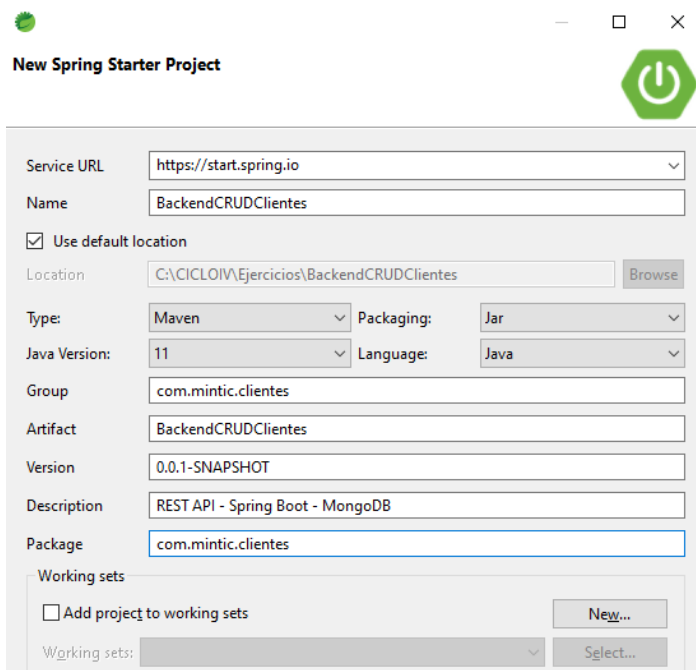
Microservicio (Backend) Clientes SPRING BOOT.

Vamos a crear un nuevo microservicio usando en el backend Spring Boot y como base de datos MongoDB, además vamos a documentar el api con swagger. El ejercicio contiene el siguiente procedimiento:

1. Creación del Proyecto.
2. Agregar Swagger al Proyecto.
3. Creación de la Estructura del Proyecto.
4. Configuración de la base de datos MongoDB
5. Creación del modelo de Cliente
6. Cree un repositorio de datos de Spring - ClienteRepository.java
7. Capa de servicio (usa repositorio)
8. Creación de las API: ClienteControlador
9. Ejecución de la aplicación de arranque Spring
10. Pruebe las API REST con Swagger.

1. Creación del Proyecto (Backend).

Vamos a crear un proyecto Spring boot con las siguientes definiciones:



New Spring Starter Project

Service URL:

Name:

☒ Use default location

Location:

Type: Packaging:

Java Version: Language:

Group:

Artifact:

Version:

Description:

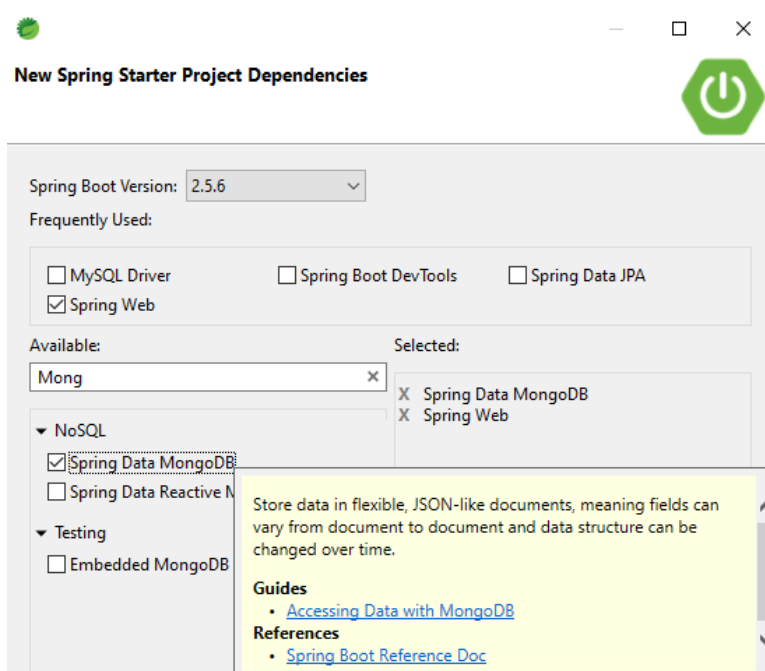
Package:

Working sets

☐ Add project to working sets

Working sets:

Por otra parte agregamos las dependencias Spring web y Spring Data MongoDB.



2. Agregar Swagger al Proyecto.

Una API definitivamente pierde su sentido sino es accesible y si no tenemos una documentación que nos ayude a entenderla.

Uno de los mayores problemas de las APIs es que en muchos casos, la documentación que les acompaña es inútil. Swagger nace con la intención de solucionar este problema. Su objetivo es estandarizar el vocabulario que utilizan las APIs. Es el diccionario API.

Cuando hablamos de Swagger nos referimos a una serie de reglas, especificaciones y herramientas que nos ayudan a documentar nuestras APIs. De esta manera, podemos realizar documentación que sea realmente útil para las personas que la necesitan. Swagger nos ayuda a crear documentación que todo el mundo entienda.

Swagger UI – La interfaz de usuario de Swagger

Swagger UI es una de las herramientas atractivas de la plataforma. Para que una documentación sea útil necesitaremos que sea navegable y que esté perfectamente organizada para un fácil acceso. Por esta razón, realizar una buena documentación puede ser realmente tedioso y consumir mucho tiempo a los desarrolladores.

Agregamos dos dependencias nuevas para gestionar el uso de swagger:

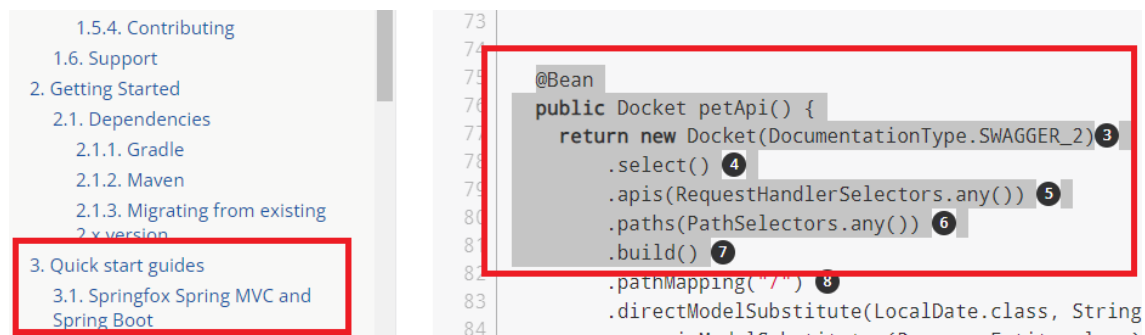
```
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger2</artifactId>
  <version>2.9.2</version>
</dependency>

<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger-ui</artifactId>
  <version>2.9.2</version>
</dependency>
```

En la clase principal se agrega la anotación “@EnableSwagger2” para activar el uso de la herramienta, además se debe agregar un código que nos permite agregar una arquitectura MVC con swagger al archivo de mismo ejecución de ejecución, la documentación de la herramienta se encuentra en el siguiente url:

<https://springfox.github.io/springfox/docs/current/#springfox-spring-mvc-and-spring-boot>

En el siguiente gráfico se ilustra el contenido de la página en la sección de implementación de spring boot:



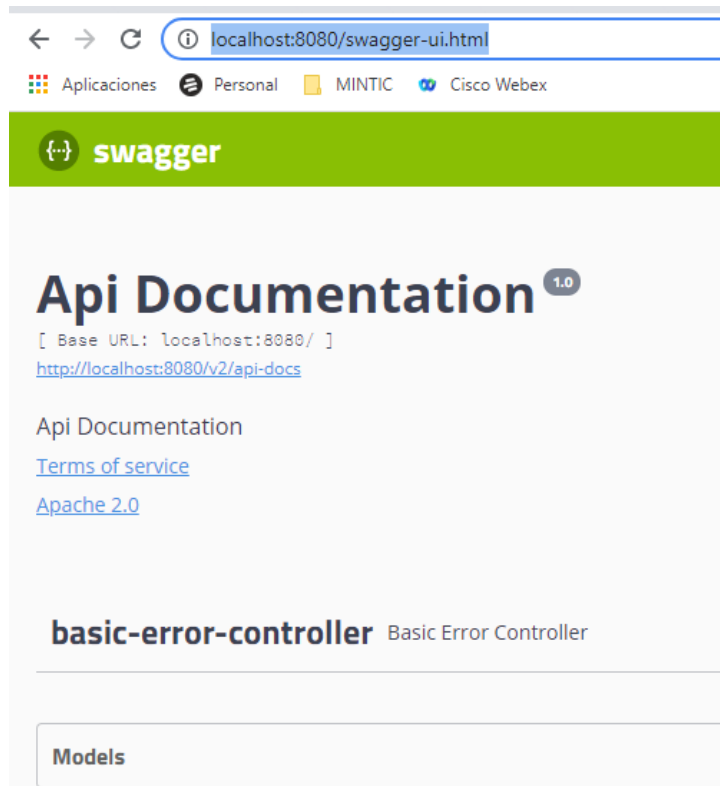
El contenido del archivo sería el siguiente:

```

1 package com.mintic.clientes;
2
3 import org.springframework.boot.SpringApplication;
12
13 @SpringBootApplication
14 @EnableSwagger2
15 public class BackendCrudClientesApplication {
16
17     @Bean
18     public Docket clientesApi() {
19         return new Docket(DocumentationType.SWAGGER_2)
20             .select()
21             .apis(RequestHandlerSelectors.any())
22             .paths(PathSelectors.any())
23             .build();
24     }
25
26     public static void main(String[] args) {
27         SpringApplication.run(BackendCrudClientesApplication.class, args);
28     }
29
30 }

```

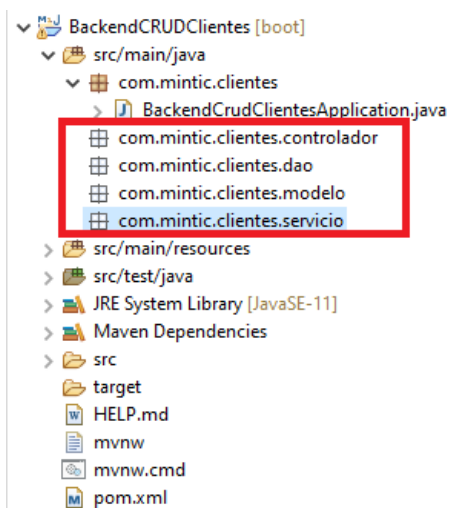
Ahora vamos a probar ejecutando el proyecto, al ejecutar sobre el url y puerto correspondiente marcaría un error de página no encontrada, si se desea visualizar la interfaz de swagger se debe agregar al url "/swagger-ui.html", el resultado se puede apreciar en la siguiente figura:



Vamos a crear el proyecto con servicios Rest con spring boot, si desean revisar más en detalle pueden encontrar una explicación en la siguiente página: <https://spring.io/guides/tutorials/rest/>.

3. Creación de la Estructura del Proyecto.

Agregamos los packages para crear los archivos necesarios en el proyecto como se aprecia en la figura:



4. Configuración de la base de datos MongoDB.

Spring Boot intenta configurar automáticamente la mayoría de las cosas según las dependencias que ha agregado en el archivo pom.xml.

Dado que hemos agregado una dependencia spring-boot-starter-mongodb, Spring Boot intenta construir una conexión con MongoDB leyendo la configuración de la base de datos del archivo application.properties. Respetando las recomendaciones de la universidad en el documento guía del proyecto como se muestra en la figura se dará una estructura a la base de como se aprecia a continuación:

Colección	Modelo	Valor de Ejemplo
db_clientes	<pre> Clientes { cedula_cliente (integer, optional), direccion_cliente (string, optional), email_cliente (string, optional), nombre_cliente (string, optional), telefono_cliente (string, optional) } </pre>	<pre> { "cedula_cliente": 0, "direccion_cliente": "string", "email_cliente": "string", "nombre_cliente": "string", "telefono_cliente": "string" } </pre>

Abra el archivo `application.properties` y agregue las siguientes propiedades de MongoDB :

```
1 # Propiedades MongoDB
2 spring.data.mongodb.uri=mongodb://localhost:27017/db_clientes
3
4
```

Según nuestra configuración, Mongo DB se ejecuta localmente en el puerto predeterminado 27017.

Tenga en cuenta que necesitamos crear la base de datos "db_clientes" usando el siguiente comando: **use db_clientes**.

Si la base de datos " db_clientes " no está presente en MongoDB , creará una nueva.

5. Creación del modelo de Cliente

Ahora creemos el modelo de Cliente que se asignará a un documento en la base de datos de MongoDB . Cree un nuevo modelo de paquete dentro del packages y agregue un archivo `Cliente.java` dentro del paquete del modelo con el siguiente contenido:

```
1 package com.mintic.clientes.modelo;
2
3 import org.springframework.data.annotation.Id;
4 import org.springframework.data.mongodb.core.mapping.Document;
5
6 @Document(collection = "clientes")
7 public class Cliente {
8
9     @Id
10     private String _id;
11
12     private int cedula;
13
14     private String direccion;
15
16     private String email;
17
18     private String nombre;
19
20     private String telefono;
21
22     public String get_id() {
23         return _id;
24     }
25
26     public void set_id(String _id) {
27         this._id = _id;
28     }
29 }
```

```
30 public int getCedula() {  
31     return cedula;  
32 }  
33  
34 public void setCedula(int cedula) {  
35     this.cedula = cedula;  
36 }  
37  
38 public String getDireccion() {  
39     return direccion;  
40 }  
41  
42 public void setDireccion(String direccion) {  
43     this.direccion = direccion;  
44 }  
45  
46 public String getEmail() {  
47     return email;  
48 }  
49  
50 public void setEmail(String email) {  
51     this.email = email;  
52 }  
53  
54 public String getNombre() {  
55     return nombre;  
56 }  
57  
58 public void setNombre(String nombre) {  
59     this.nombre = nombre;  
60 }  
61  
62 public String getTelefono() {  
63     return telefono;  
64 }  
65  
66 public void setTelefono(String telefono) {  
67     this.telefono = telefono;  
68 }  
69  
70  
71 }
```

6. Cree un repositorio de datos de Spring - ClienteRepository.java.

A continuación, necesitamos crear ProductRepository para acceder a los datos de la base de datos, en esta interface se agrega como extensión una clase "MongoRepository" que contiene los métodos básicos de gestión y ejecución de la Base de Datos.

```
1 package com.mintic.clientes.dao;  
2  
3 import org.springframework.data.mongodb.repository.MongoRepository;  
4  
5  
6  
7 @Repository  
8 public interface IClienteRepositorio extends MongoRepository<Cliente, String> {  
9  
10  
11 }
```

7. Capa de servicio (usa repositorio).

La capa de servicio es opcional; aún se recomienda realizar lógica de negocio adicional, si corresponde. Generalmente, nos conectaremos con el repositorio aquí para operaciones CRUD. Vamos a crear la interface con las cabeceras de los métodos a implementar.

```
1 package com.mintic.clientes.servicio;
2
3 import java.util.List;
4
5 import com.mintic.clientes.modelo.Cliente;
6
7 public interface ClienteServicio {
8
9     Cliente crearCliente(Cliente cliente);
10
11     Cliente updateProduct(Cliente cliente);
12
13     List<Cliente> getAllCliente();
14
15     Cliente getClienteById(String clienteId);
16
17     void deleteCliente(String id);
18
19 }
```

Clase "ClienteServicioImpl".

Ahora implementamos los métodos declarados en una clase llamada "ClienteServicioImpl", los métodos se desarrollan instanciando un objeto de la interface de servicio IClienteServicio.

```
1 package com.mintic.clientes.servicio;
2
3 import java.util.List;
4 import java.util.Optional;
5
6 import org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.stereotype.Service;
8 import org.springframework.transaction.annotation.Transactional;
9
10 import com.mintic.clientes.dao.IClienteRepositorio;
11 import com.mintic.clientes.excepcion.ResourceNotFoundException;
12 import com.mintic.clientes.modelo.Cliente;
13
14 @Service
15 @Transactional
16 public class ClienteServicioImpl implements ClienteServicio {
17
18     @Autowired
19     private IClienteRepositorio clienteRepo;
20
21
22     @Override
23     public Cliente crearCliente(Cliente cliente) {
24         return clienteRepo.save(cliente);
25     }
26 }
```



```

27 @Override
28 public Cliente updateCliente(Cliente cliente) {
29     Optional<Cliente> clienteDb = this.clienteRepo.findById(cliente.get_id());
30
31     if(clienteDb.isPresent()) {
32         Cliente clienteUpdate = clienteDb.get();
33         clienteUpdate.set_id(cliente.get_id());
34         clienteUpdate.setCedula(cliente.getCedula());
35         clienteUpdate.setDireccion(cliente.getDireccion());
36         clienteUpdate.setEmail(cliente.getEmail());
37         clienteUpdate.setNombre(cliente.getNombre());
38         clienteUpdate.setTelefono(cliente.getTelefono());
39         clienteRepo.save(clienteUpdate);
40         return clienteUpdate;
41     }else {
42         throw new ResourceNotFoundException("Registro no Encontrado con el Id:"+cliente.get_id());
43     }
44 }
45
46 @Override
47 public List<Cliente> getAllCliente() {
48     return clienteRepo.findAll();
49 }
50
51 @Override
52 public Cliente getClienteById(String clienteId) {
53     Optional<Cliente> clienteDb = this.clienteRepo.findById(clienteId);
54     if(clienteDb.isPresent()) {
55         return clienteDb.get();
56     }else {
57         throw new ResourceNotFoundException("Registro no Encontrado con el Id:"+clienteId);
58     }
59 }
60
61 @Override
62 public void deleteCliente(String clienteId) {
63     Optional<Cliente> clienteDb = this.clienteRepo.findById(clienteId);
64     if(clienteDb.isPresent()) {
65         this.clienteRepo.delete(clienteDb.get());
66     }else {
67         throw new ResourceNotFoundException("Registro no Encontrado con el Id:"+clienteId);
68     }
69 }
70
71 }

```

• Clase "ResourceNotFoundException".

Para poder administrar las excepciones creamos una clase que permita emitir un mensaje en tiempo de ejecución, el contenido se puede apreciar en las siguientes líneas, se crea sobre un packages excepcion.

```

1 package com.mintic.clientes.excepcion;
2
3 import org.springframework.web.bind.annotation.ResponseStatus;
4
5 @ResponseStatus
6 public class ResourceNotFoundException extends RuntimeException{
7
8     private static final long serialVersionUID = 1L;
9
10    public ResourceNotFoundException(String message) {
11        super(message);
12    }
13
14    public ResourceNotFoundException(String message, Throwable throwable) {
15        super(message, throwable);
16    }
17
18 }

```

8. Creación de las API: ClienteControlador.

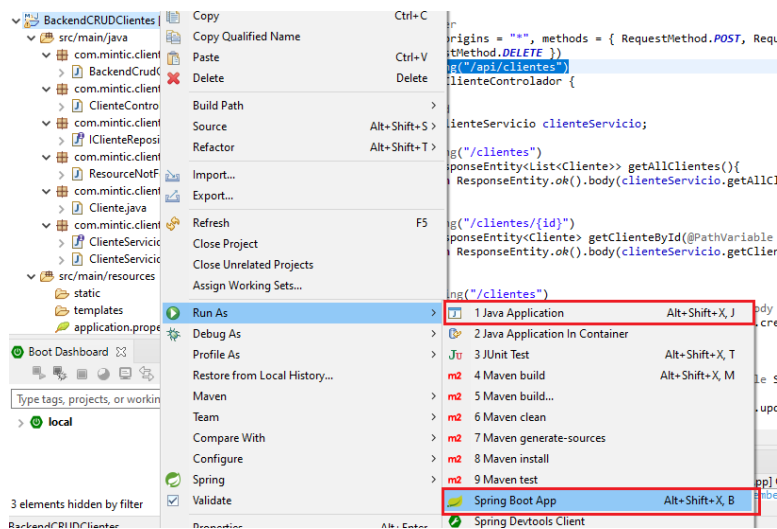
En el controlador creamos las API que van a poder utilizar los clientes con los métodos requeridos usando la interface del servicio e implementando las soluciones que permitan construir el CRUD completo de clientes.

Creamos una clase llamada ClienteControlador con las anotaciones de `@RestController`, `@CrossOrigin` y `@RequestMapping("/api/clientes")`, este es el puente para que el cliente llame a cada uno de los métodos.

```
1 package com.mintic.clientes.controlador;
2
3 import java.util.List;
4
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.http.HttpStatus;
7 import org.springframework.http.ResponseEntity;
8 import org.springframework.web.bind.annotation.CrossOrigin;
9 import org.springframework.web.bind.annotation.DeleteMapping;
10 import org.springframework.web.bind.annotation.GetMapping;
11 import org.springframework.web.bind.annotation.PathVariable;
12 import org.springframework.web.bind.annotation.PostMapping;
13 import org.springframework.web.bind.annotation.PutMapping;
14 import org.springframework.web.bind.annotation.RequestBody;
15 import org.springframework.web.bind.annotation.RequestMapping;
16 import org.springframework.web.bind.annotation.RequestMethod;
17 import org.springframework.web.bind.annotation.RestController;
18
19 import com.mintic.clientes.modelo.Cliente;
20 import com.mintic.clientes.servicio.ClienteServicio;
21
22 @RestController
23 @CrossOrigin(origins = "*", methods = { RequestMethod.POST, RequestMethod.GET, RequestMethod.PUT,
24     RequestMethod.DELETE })
25 @RequestMapping("/api/clientes")
26 public class ClienteControlador {
27
28     @Autowired
29     private ClienteServicio clienteServicio;
30
31     @GetMapping("/clientes")
32     public ResponseEntity<List<Cliente>> getAllClientes(){
33         return ResponseEntity.ok().body(clienteServicio.getAllCliente());
34     }
35
36     @GetMapping("/clientes/{id}")
37     public ResponseEntity<Cliente> getClienteById(@PathVariable String id) {
38         return ResponseEntity.ok().body(clienteServicio.getClienteById(id));
39     }
40
41     @PostMapping("/clientes")
42     public ResponseEntity < Cliente > crearCliente(@RequestBody Cliente cliente){
43         return ResponseEntity.ok().body(this.clienteServicio.crearCliente(cliente));
44     }
45
46     @PutMapping("/clientes/{id}")
47     public ResponseEntity<Cliente> updateCliente(@PathVariable String id, @RequestBody Cliente cliente){
48         cliente.set_id(id);
49         return ResponseEntity.ok().body(this.clienteServicio.updateCliente(cliente));
50     }
51
52     @DeleteMapping("/clientes/{id}")
53     public HttpStatus deleteProduct(@PathVariable String id) {
54         this.clienteServicio.deleteCliente(id);
55         return HttpStatus.OK;
56     }
57
58 }
```

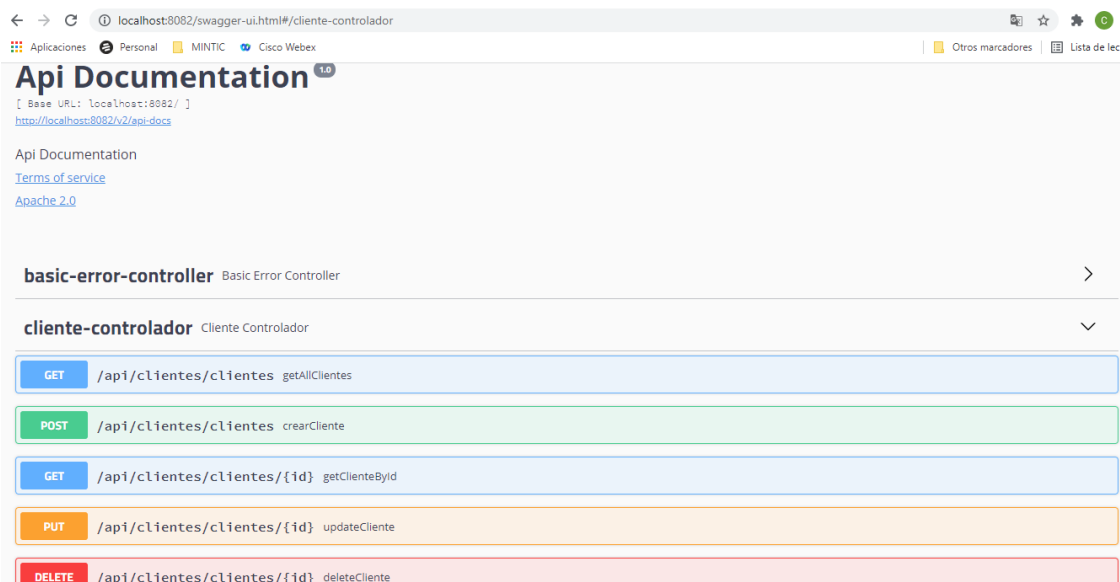
9. Ejecución de la aplicación de arranque Spring.

Cambiamos el puerto de ejecución a 8082 y después para ejecutar y probar el proyecto podemos acudir al archivo de inicio del mismo y ejecutar como una aplicación java o en su defecto podemos "Run As" Spring Boot App como se puede observar en la figura:



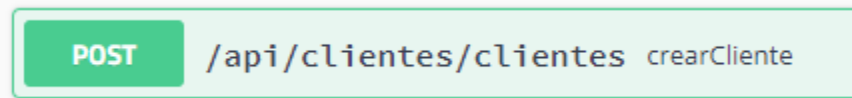
10. Pruebe las API REST con Swagger.

Para probar el proyecto nos vamos a un navegador y escribimos <http://localhost:8082/swagger-ui.html#/> nos debe aparecer una figura como la siguiente:



Podemos probar cada uno de los métodos en forma directa en el navegador y reemplazamos con ello el uso de Postman.

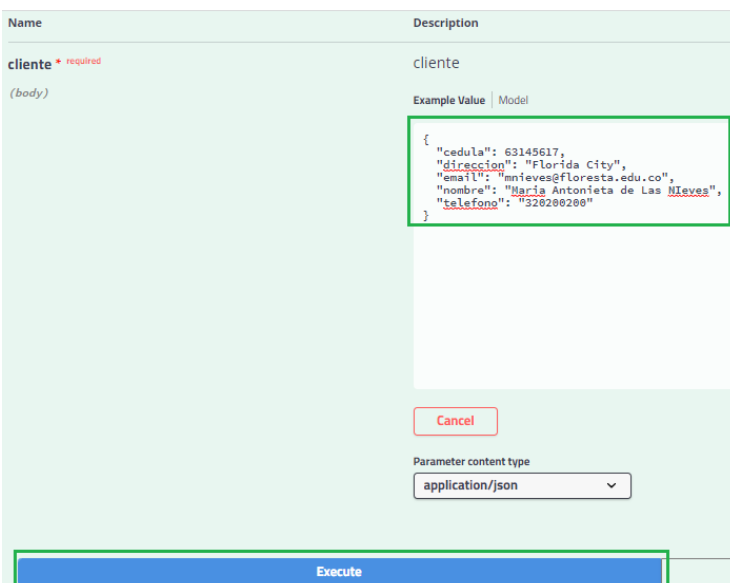
- **Método POST: crearCliente.**



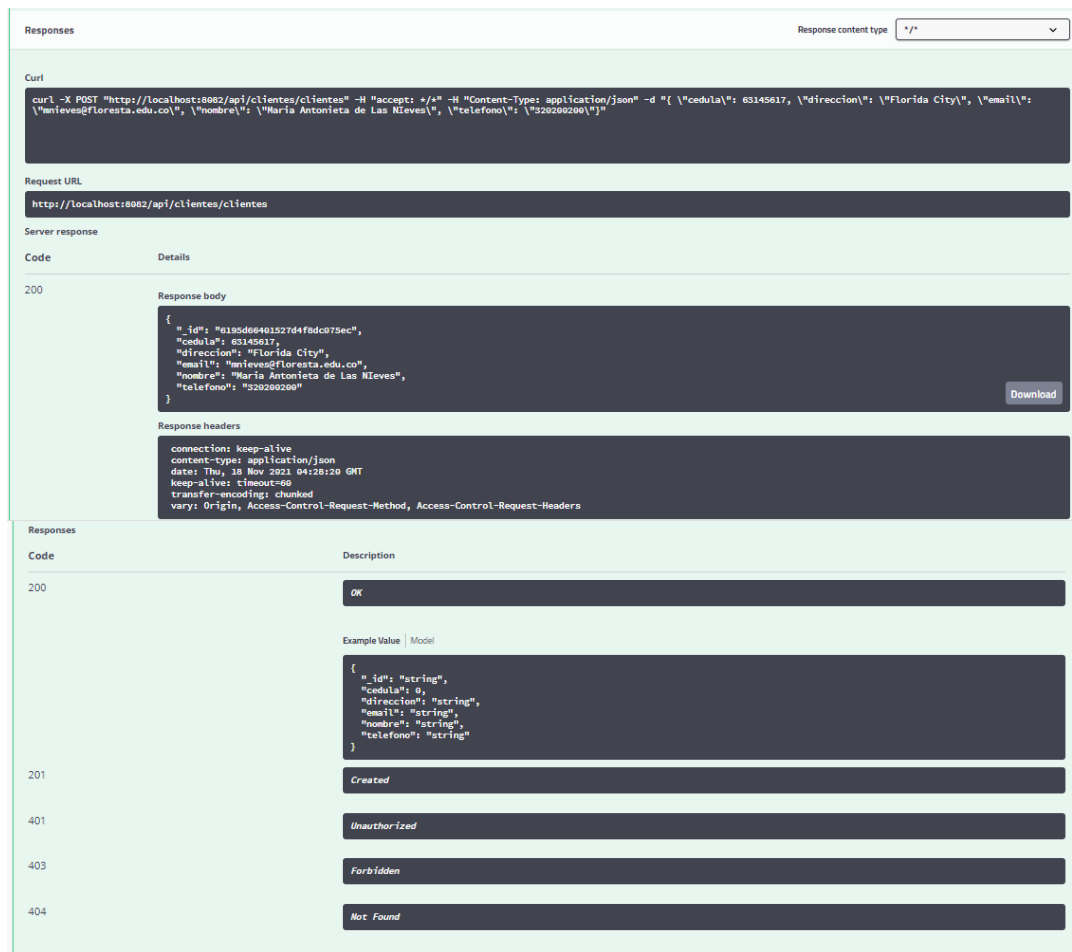
Vamos a iniciar con el método post para crear un cliente, debemos seleccionar el botón de Try it Out, escribimos los datos del nuevo cliente sin escribir algún Id con el fin de que MongoDB lo genere:



Después presionamos el botón de ejecutar (Execute), debajo aparecerá la respuesta correspondiente.



El resultado se puede apreciar en la siguiente figura:



The screenshot displays a REST client interface with the following details:

- Response content type:** */*
- Curl:** `curl -X POST "http://localhost:8082/api/clientes/clientes" -H "accept: */*" -H "Content-Type: application/json" -d '{"cedula": 63145617, "direccion": "Florida City", "email": "Mnieves@foresta.edu.co", "nombre": "Maria Antonieta de Las Nieves", "telefono": "329299299"}'`
- Request URL:** `http://localhost:8082/api/clientes/clientes`
- Server response:**
 - Code:** 200
 - Response body:**

```
{
  "_id": "6195d66401527d4f8dc075ec",
  "cedula": 63145617,
  "direccion": "Florida City",
  "email": "Mnieves@foresta.edu.co",
  "nombre": "Maria Antonieta de Las Nieves",
  "telefono": "329299299"
}
```
 - Response headers:**

```
connection: keep-alive
content-type: application/json
date: Thu, 18 Nov 2021 04:28:20 GMT
keep-alive: timeout=60
transfer-encoding: chunked
vary: Origin, Access-Control-Request-Method, Access-Control-Request-Headers
```
- Responses Table:**

Code	Description
200	OK
201	Created
401	Unauthorized
403	Forbidden
404	Not Found

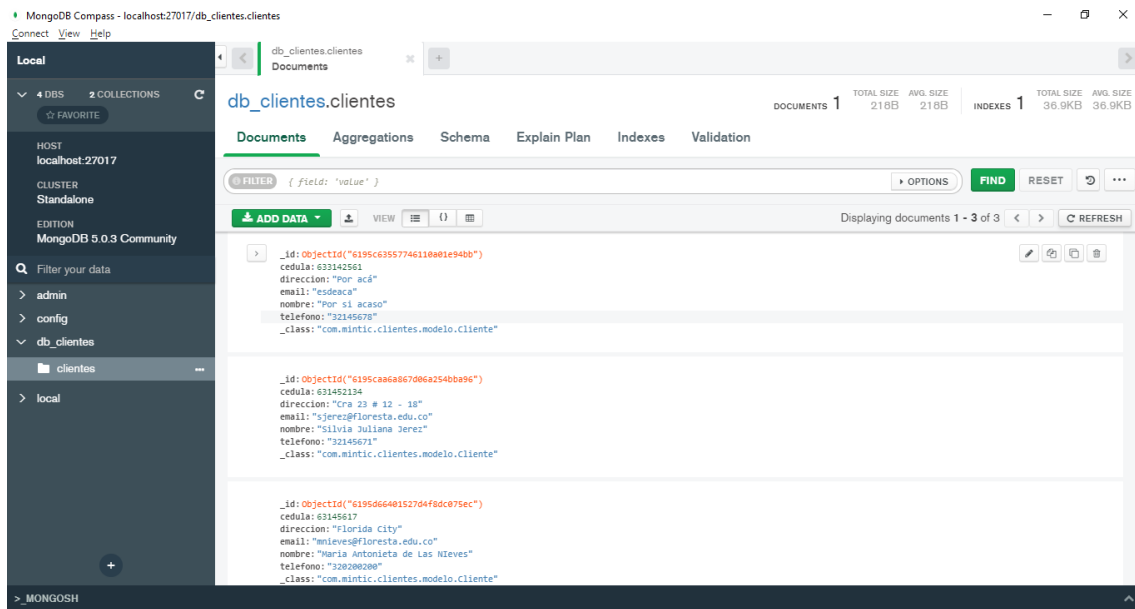
Podemos revisar sobre la Base de Datos de MongoDB en la consola o usando la interfaz de Compass.

Si usamos la consola podemos ejecutar las instrucciones que nos permita visualizar si la base de datos existe y podemos acceder a ella como lo muestra la gráfica:

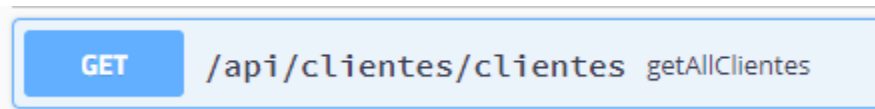
```
> show dbs
admin      0.000GB
config     0.000GB
db_clientes 0.000GB
local      0.000GB
> use db_clientes
switched to db db_clientes
```

```
> show collections
clientes
> db.clientes.find()
{ "_id" : ObjectId("6195c63557746110a01e94bb"), "cedula" : 633142561, "direccion" : "Por acá", "email" : "esdeaca", "nombre" : "Por si acaso", "telefono" : "32145678", "_class" : "com.mintic.clientes.modelo.Cliente" }
{ "_id" : ObjectId("6195caa6a867d06a254bba96"), "cedula" : 631452134, "direccion" : "Cra 23 # 12 - 18", "email" : "sjerez@floresta.edu.co", "nombre" : "Silvia Juliana Jerez", "telefono" : "32145671", "_class" : "com.mintic.clientes.modelo.Cliente" }
{ "_id" : ObjectId("6195d66401527d4f8dc075ec"), "cedula" : 63145617, "direccion" : "Florida City", "email" : "mnieves@floresta.edu.co", "nombre" : "Maria Antonieta de Las Nieves", "telefono" : "320200200", "_class" : "com.mintic.clientes.modelo.Cliente" }
> db.clientes.find().pretty()
{
  "_id" : ObjectId("6195c63557746110a01e94bb"),
  "cedula" : 633142561,
  "direccion" : "Por acá",
  "email" : "esdeaca",
  "nombre" : "Por si acaso",
  "telefono" : "32145678",
  "_class" : "com.mintic.clientes.modelo.Cliente"
}
{
  "_id" : ObjectId("6195caa6a867d06a254bba96"),
  "cedula" : 631452134,
  "direccion" : "Cra 23 # 12 - 18",
  "email" : "sjerez@floresta.edu.co",
  "nombre" : "Silvia Juliana Jerez",
  "telefono" : "32145671",
  "_class" : "com.mintic.clientes.modelo.Cliente"
}
{
  "_id" : ObjectId("6195d66401527d4f8dc075ec"),
  "cedula" : 63145617,
  "direccion" : "Florida City",
  "email" : "mnieves@floresta.edu.co",
  "nombre" : "Maria Antonieta de Las Nieves",
  "telefono" : "320200200",
  "_class" : "com.mintic.clientes.modelo.Cliente"
}
}
```

Si revisamos Compass también podemos visualizar el estado actual de la Base de Datos:



- **Método GET: getAllClientes.**



Al probar el método `getAllClientes` podemos visualizar el resultado con la lista de todos los clientes:

Responses

Curl

```
curl -X GET "http://localhost:8082/api/clientes/clientes" -H "accept: */*"
```

Request URL

```
http://localhost:8082/api/clientes/clientes
```

Server response

Code	Details
200	<p>Response body</p> <pre>[{ "_id": "6195c63557746110a01e94bb", "cedula": 633142561, "direccion": "Por acá", "email": "esdeaca", "nombre": "Por si acaso", "telefono": "32145678" }, { "_id": "6195caa6a867d06a254bba96", "cedula": 631452134, "direccion": "Cra 23 # 12 - 18", "email": "sjerez@floresta.edu.co", "nombre": "Silvia Juliana Jerez", "telefono": "32145671" }, { "_id": "6195d66401527d4f8dc075ec", "cedula": 63145617, "direccion": "Florida City", "email": "mnieves@floresta.edu.co", "nombre": "María Antonieta de Las Nieves", "telefono": "320290290" }]</pre> <p>Response headers</p> <pre>connection: keep-alive content-type: application/json date: Thu, 18 Nov 2021 04:37:05 GMT keep-alive: timeout=60 transfer-encoding: chunked vary: Origin, Access-Control-Request-Method, Accept</pre>

Responses	
Code	Description
200	OK
<div>Example Value Model</div> <pre>[{ "id": "string", "cedula": 0, "direccion": "string", "email": "string", "nombre": "string", "telefono": "string" }]</pre>	
401	Unauthorized
403	Forbidden
404	Not Found

- **Método GET: getClientById(id).**

Ahora probamos del método de búsqueda por id getClientById

GET

/api/clientes/clientes/{id} getClientById

GET

/api/clientes/clientes/{id} getClientById

Parameters

Name	Description
id <small>required</small>	id
string (path)	<input type="text" value="6195caa6a867d06a254bba96"/>

Execute

Responses

La respuesta será como se muestra:

Responses

Curl

```
curl -X GET "http://localhost:8082/api/clientes/clientes/6195caa6a867d06a254bba96" -H "accept: */*"
```

Request URL

```
http://localhost:8082/api/clientes/clientes/6195caa6a867d06a254bba96
```

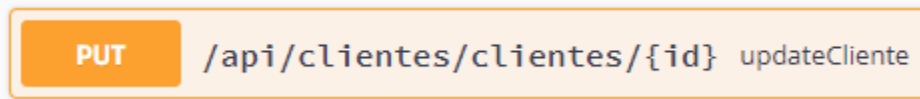
Server response

Code	Details
200	<div>Response body</div> <pre>{ "_id": "6195caa6a867d06a254bba96", "cedula": 631452134, "direccion": "Cra 23 # 12 - 18", "email": "sjerez@floresta.edu.co", "nombre": "Silvia Juliana Jerez", "telefono": "32145671" }</pre> <div>Response headers</div> <pre>connection: keep-alive content-type: application/json date: Thu, 18 Nov 2021 04:43:30 GMT keep-alive: timeout=60 transfer-encoding: chunked vary: Origin, Access-Control-Request-Method, Access-Control-Request-Headers</pre>

Responses

Code	Description
200	<div>OK</div> <div>Example Value Model</div> <pre>{ "_id": "string", "cedula": 0, "direccion": "string", "email": "string", "nombre": "string", "telefono": "string" }</pre>
401	Unauthorized
403	Forbidden
404	Not Found

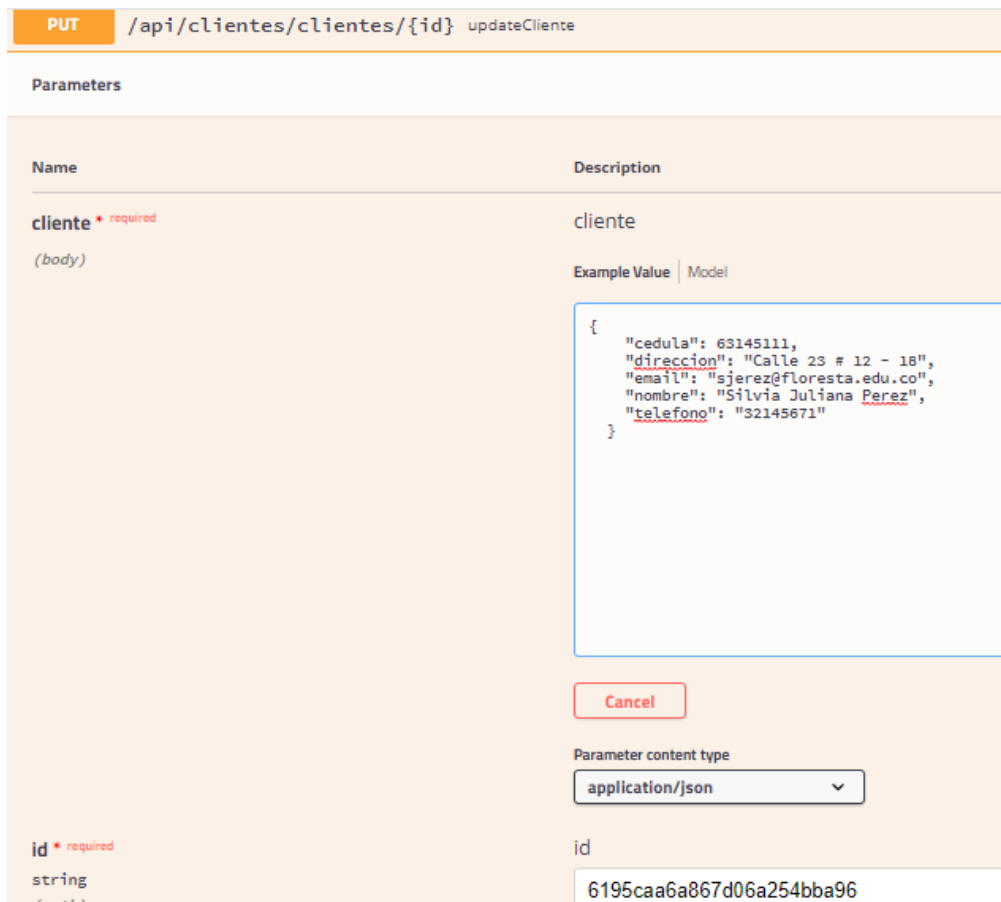
- **Método PUT: updateCliente(id).**



Ahora vamos a probar la actualización y tomamos como guía el siguiente documento:

```
> {
  "_id": ObjectId("6195caa6a867d06a254bba96"),
  "cedula": 631452134,
  "direccion": "Cra 23 # 12 - 18",
  "email": "sjerez@floresta.edu.co",
  "nombre": "Silvia Juliana Jerez",
  "telefono": "32145671",
  "_class": "com.mintic.clientes.modelo.Cliente"
}
```

Seleccionamos el método en el navegador y escribimos los nuevos valores y no debemos que el parámetro de ejecución es el id que debemos escribir en la entrada respectiva, es más en el objeto Json no escribimos el Id.



PUT /api/clientes/clientes/{id} updateCliente

Parameters

Name	Description
cliente * required (body)	cliente

Example Value | Model

```
{
  "cedula": 63145111,
  "direccion": "Calle 23 # 12 - 18",
  "email": "sjerez@floresta.edu.co",
  "nombre": "Silvia Juliana Perez",
  "telefono": "32145671"
}
```

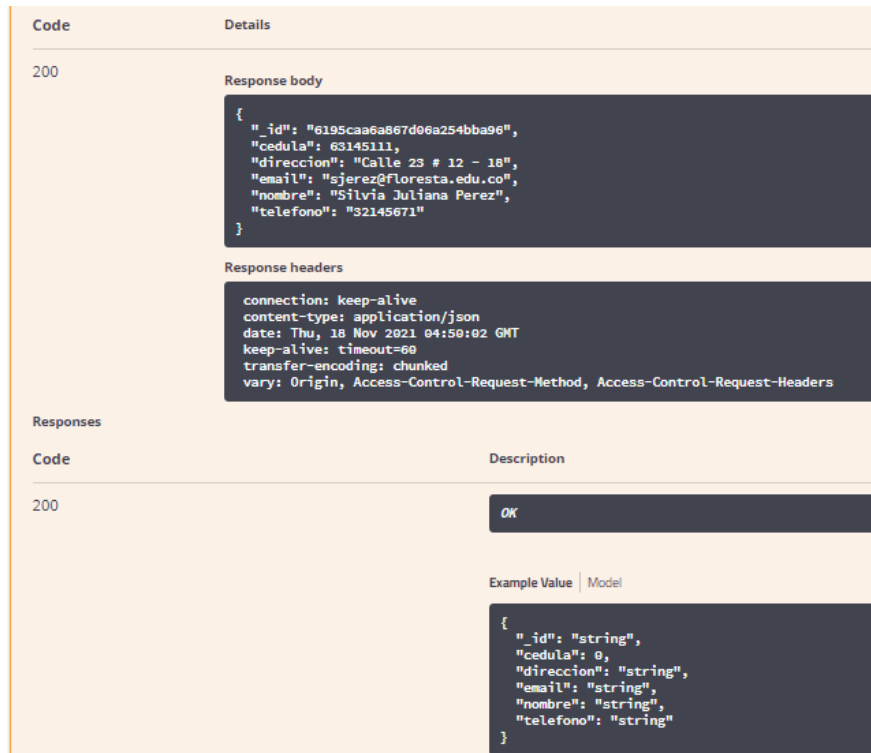
Cancel

Parameter content type
application/json

id * required
string
(path)

6195caa6a867d06a254bba96

El resultado se puede apreciar en la gráfica:



The screenshot shows a REST client interface with a 200 status code. The response body is a JSON object representing a client record. The response headers include connection, content-type, date, keep-alive, transfer-encoding, and vary.

Response body:

```
{
  "_id": "6195caa6a867d06a254bba96",
  "cedula": 63145111,
  "direccion": "Calle 23 # 12 - 18",
  "email": "sjerez@floresta.edu.co",
  "nombre": "Silvia Juliana Perez",
  "telefono": "32145671"
}
```

Response headers:

```
connection: keep-alive
content-type: application/json
date: Thu, 18 Nov 2021 04:50:02 GMT
keep-alive: timeout=60
transfer-encoding: chunked
vary: Origin, Access-Control-Request-Method, Access-Control-Request-Headers
```

Responses table:

Code	Description
200	OK

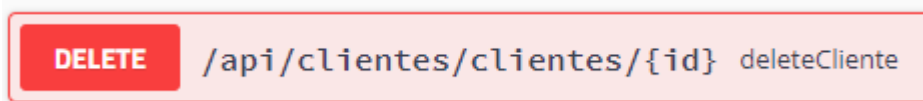
Example Value | Model:

```
{
  "_id": "string",
  "cedula": 0,
  "direccion": "string",
  "email": "string",
  "nombre": "string",
  "telefono": "string"
}
```

Podemos revisar sobre el Shell de MongoDB y el resultado sería el siguiente:

```
> db.clientes.find().pretty()
{
  "_id" : ObjectId("6195c63557746110a01e94bb"),
  "cedula" : 633142561,
  "direccion" : "Por acá",
  "email" : "esdeaca",
  "nombre" : "Por si acaso",
  "telefono" : "32145678",
  "_class" : "com.mintic.clientes.modelo.Cliente"
}
{
  "_id" : ObjectId("6195caa6a867d06a254bba96"),
  "cedula" : 63145111,
  "direccion" : "Calle 23 # 12 - 18",
  "email" : "sjerez@floresta.edu.co",
  "nombre" : "Silvia Juliana Perez",
  "telefono" : "32145671",
  "_class" : "com.mintic.clientes.modelo.Cliente"
}
{
  "_id" : ObjectId("6195d66401527d4f8dc075ec"),
  "cedula" : 63145617,
  "direccion" : "Florida City",
  "email" : "mnieves@floresta.edu.co",
  "nombre" : "Maria Antonieta de Las Nieves",
  "telefono" : "320200200",
  "_class" : "com.mintic.clientes.modelo.Cliente"
}
```

- **Método DELETE: deleteCliente.**



Por último, vamos a probar el método de eliminar, para ello vamos a hacerlo con el siguiente documento copiando el `_id` y pegándolo en el navegador:

```

1  _id: ObjectId("6195c63557746110a01e94bb")
2  cedula : 633142561
3  direccion : "Por acá //"
4  email : "esdeaca //"
5  nombre : "Por si acaso //"
6  telefono : "32145678 //"
7  _class : "com.mintic.clientes.modelo.Cliente //"

```

El escribir o pegar el valor del id podemos ejecutar el método y probarlo.



La respuesta en la pantalla de swagger se puede visualizar en el siguiente gráfico:

