

Arquitecturas de Software para Aplicaciones Empresariales

Desarrollo de servicios web RestFul con Spring y prueba con Postman o Thunder



PROGRAMA DE INGENIERÍA ELECTRÓNICA Y TELECOMUNICACIONES

Ing. Daniel Eduardo Paz Perafán (danielp@Unicauca.edu.co)

Ing. Pablo A. Magé (pimage@Unicauca.edu.co)

TECNOLOGIAS PARA EL DESARROLLO DE JAVA EE



- ❖ EL framework Spring proporciona un modelo integral de programación y configuración para aplicaciones empresariales modernas basadas en Java o Groovy. Tiene un gran conjunto de módulos.
- ❖ El framework de Spring es muy potente, **pero su configuración inicial, uso y la preparación de las aplicaciones para producción** son tareas bastante tediosas. Spring Boot es una herramienta que facilita el uso del framework.
- ❖ Maven permite importar artefactos (librerías) mediante un fichero **POM.xml** (Project Object Model).
- ❖ Apache Tomcat es un servidor de aplicaciones, el cual va a permitir alojar los servicios web REST.



Spring Boot



Apache Tomcat

DESARROLLO DE UN SERVICIO WEB RESTFUL



IDE A UTILIZAR

Para crear un proyecto en Spring, trabajaremos con el entorno visual studio code. Para configurar el entorno con un conjunto de extensiones que permitan trabajar con el framework spring deben seguir los pasos planteados en el siguiente link <https://code.visualstudio.com/docs/java/java-spring-boot>.

Los proyectos se trabajaron con la versión 17 del JDK.

- Link a la página de Pivotal: <https://spring.io/tools>
- Pivotal, es la página oficial de los creadores del framework Spring.
- En la pagina web encontramos herramientas para trabajar con Spring.

Spring Tools 4 for Eclipse

The all-new Spring Tool Suite 4.
Free. Open source.

4.8.1 - LINUX 64-BIT

4.8.1 - MACOS 64-BIT

4.8.1 - WINDOWS 64-BIT

Spring Tools 4 for Visual Studio Code

Free. Open source.


SPRING TOOLS 4
VSCode Marketplace

Spring Tools 4 for Theia

Free. Open source.

SPRING TOOLS 4
Installation for Theia

EXTENSIONES BASICAS




Spring Boot Extension Pack v0.2.1

VMware vmware.com | 2,138,280 | ★★★★★ (16)

A collection of extensions for developing Spring Boot applications

[Disable](#) [Uninstall](#) ⚙️

This extension is enabled globally.




Extension Pack for Java v0.25.15

Microsoft microsoft.com | 25,411,551 | ★★★★★☆ (72)

Popular extensions for Java development that provides Java IntelliSense, debugging, testin...

[Disable](#) [Uninstall](#) [Switch to Pre-Release Version](#) ⚙️

This extension is enabled globally.



Maven for Java v0.44.0

Microsoft microsoft.com | 27,177,473 | ★★★★★☆ (15)

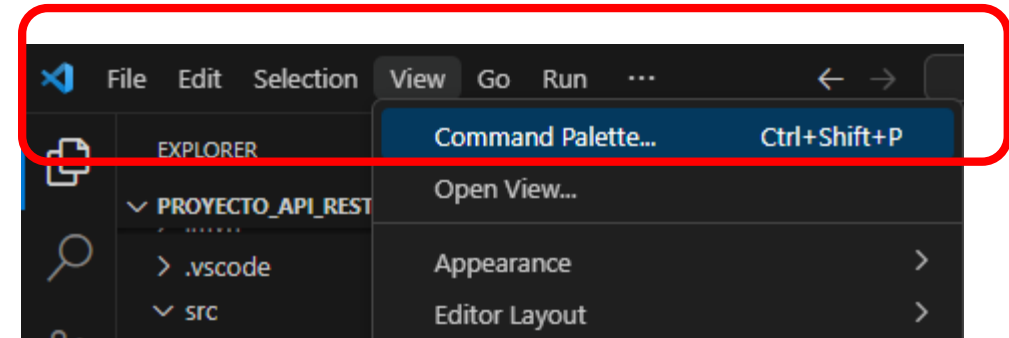
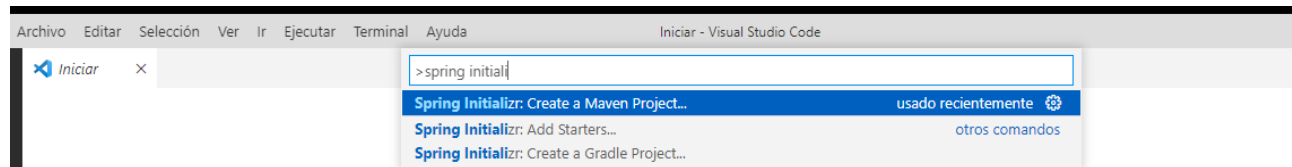
Manage Maven projects, execute goals, generate project from archetype, improve user exp...

[Disable](#) [Uninstall](#) [Switch to Pre-Release Version](#) ⚙️

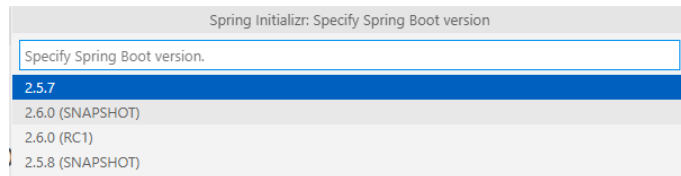
This extension is enabled globally.

CREAR UN PROYECTO DE TIPO SPRING BOOT

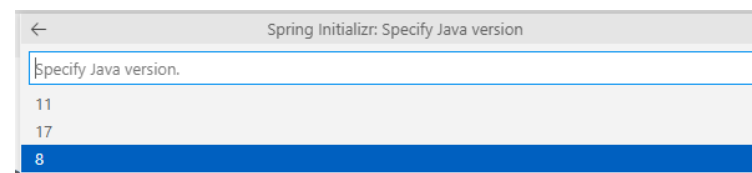
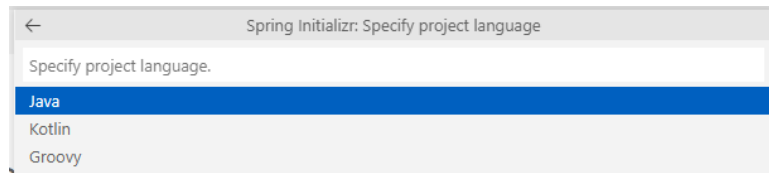
Paso 1



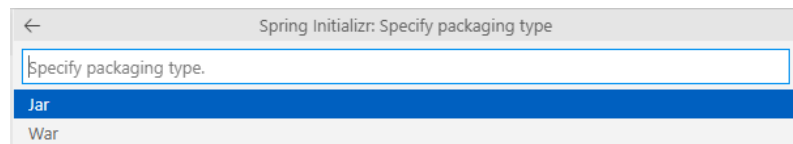
Paso 2



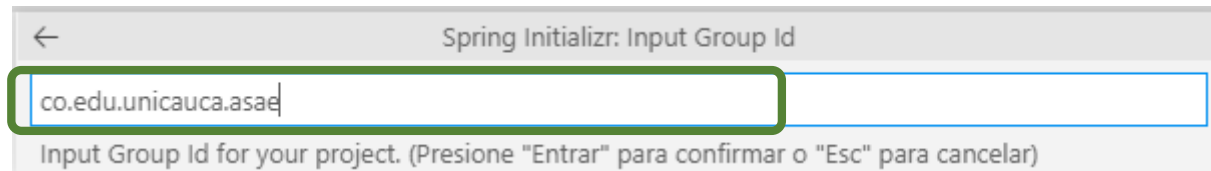
Paso 3



Paso 4



Paso 5



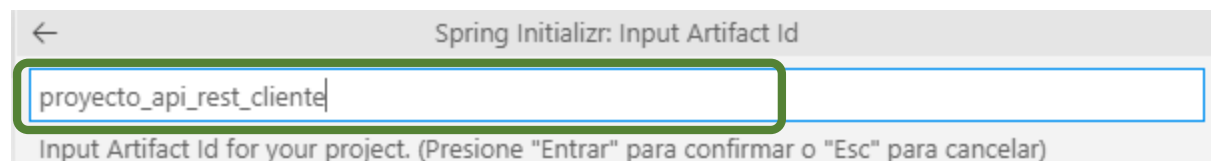
← Spring Initializr: Input Group Id

co.edu.unicauca.asae

Input Group Id for your project. (Presione "Entrar" para confirmar o "Esc" para cancelar)

Group; Identificará su proyecto de forma única en todos los proyectos, por lo que debemos aplicar un esquema de nomenclatura. Tiene que seguir las reglas del nombre del paquete, lo que significa que tiene que ser al menos como un nombre de dominio que usted controla, y puede crear tantos subgrupos como desee.

Paso 6



← Spring Initializr: Input Artifact Id

proyecto_api_rest_cliente

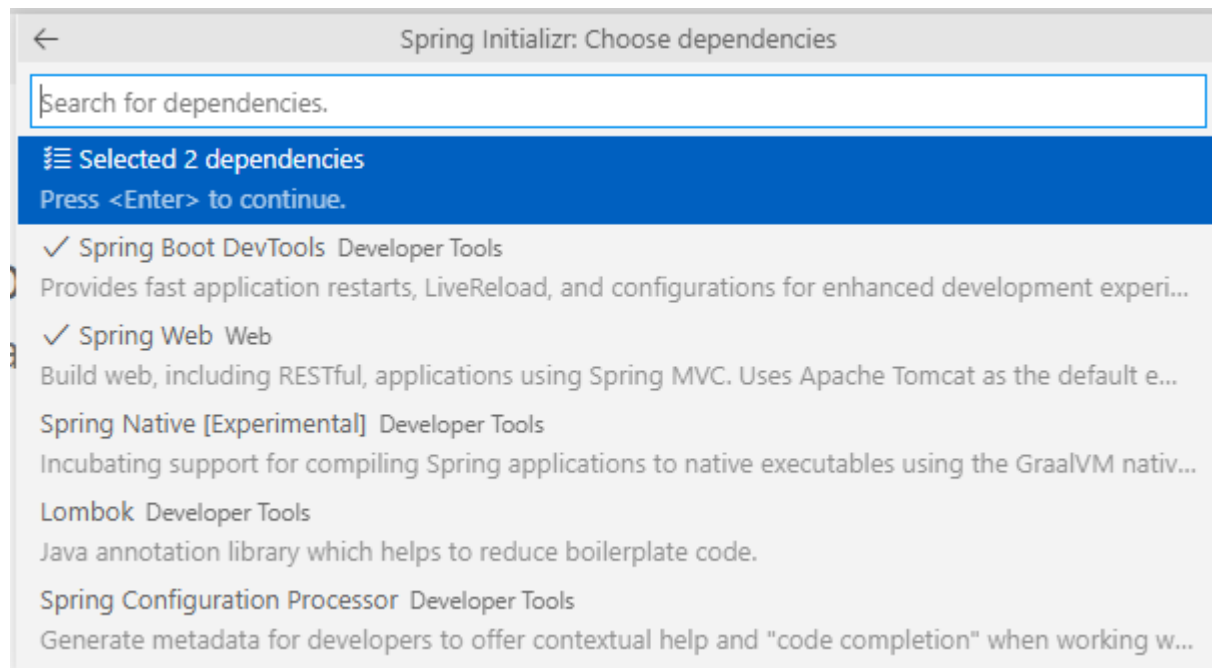
Input Artifact Id for your project. (Presione "Entrar" para confirmar o "Esc" para cancelar)

Artefact: es el nombre que se asignara al jar sin una versión específica. Puede elegir el nombre que desee.

Dependencias

Paso 7

Para nuestro proyecto hemos elegido las siguientes dependencias:



Spring Boot DevTools

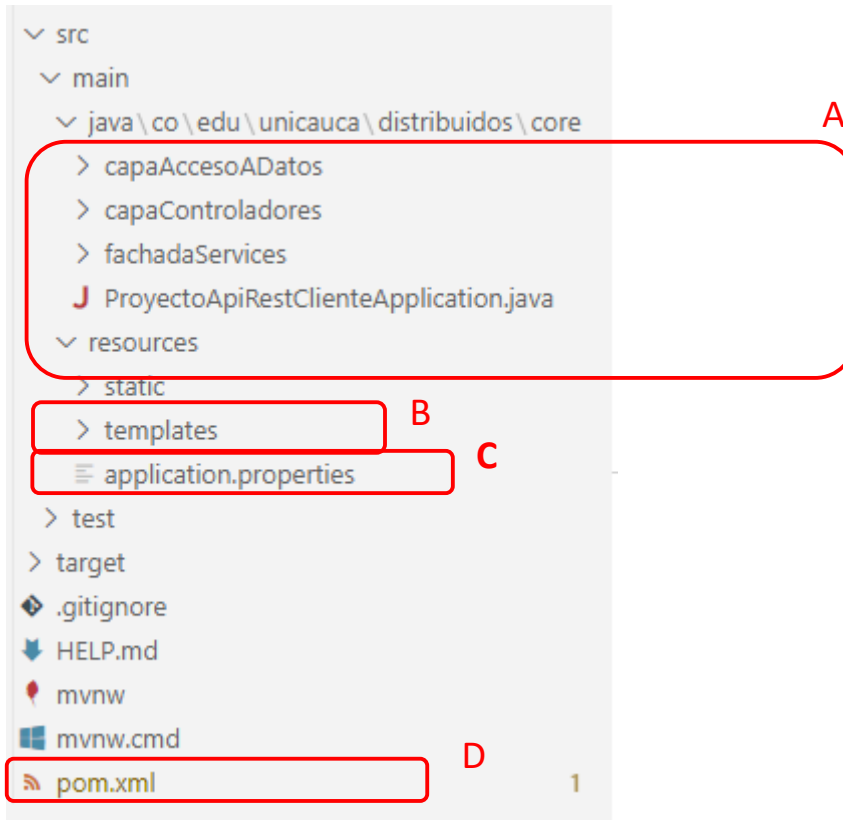
HERRAMIENTAS DE DESARROLLO

Proporciona reinicios rápidos de aplicaciones, LiveReload y configuraciones para una experiencia de desarrollo mejorada.

Spring Web

WEB

Cree aplicaciones web, incluidas RESTful, utilizando Spring MVC. Utiliza Apache Tomcat como contenedor integrado predeterminado.



A: Dentro del paquete core se ubican los controladores, clases del modelo asociadas a la lógica de negocio y el acceso a datos.

B: En la carpeta templates se almacenan las vistas de la aplicación.

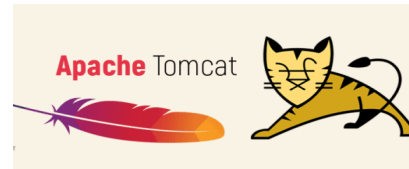
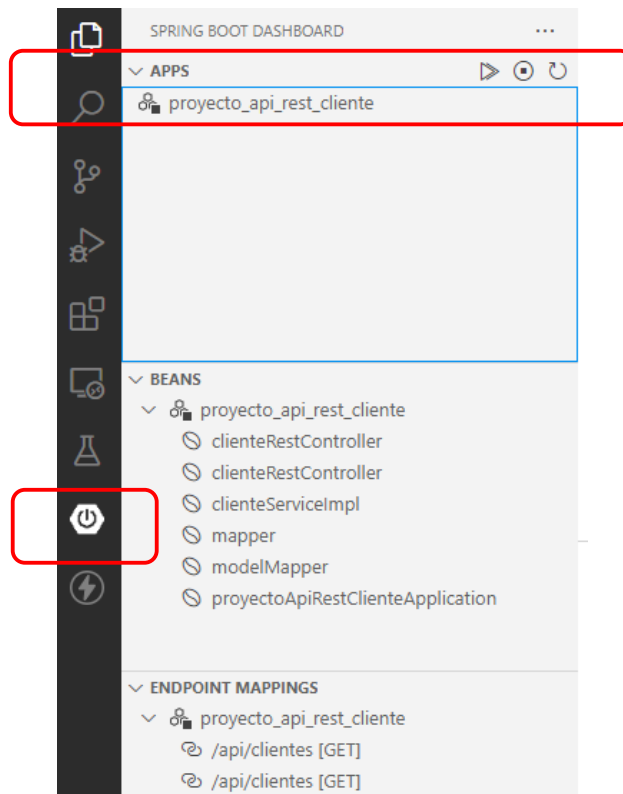
C: En el archivo `application.properties` se ubican propiedades que permiten establecer el puerto de escucha del servidor de aplicaciones, la cadena de conexión a la base de datos, el driver de conexión a la base de datos.

```
spring.datasource.url=jdbc:mysql://localhost/bdgestionempleados?serverTimezone=GMT
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.hibernate.ddl-auto=update
spring.thymeleaf.cache=true
server.port=9000
```

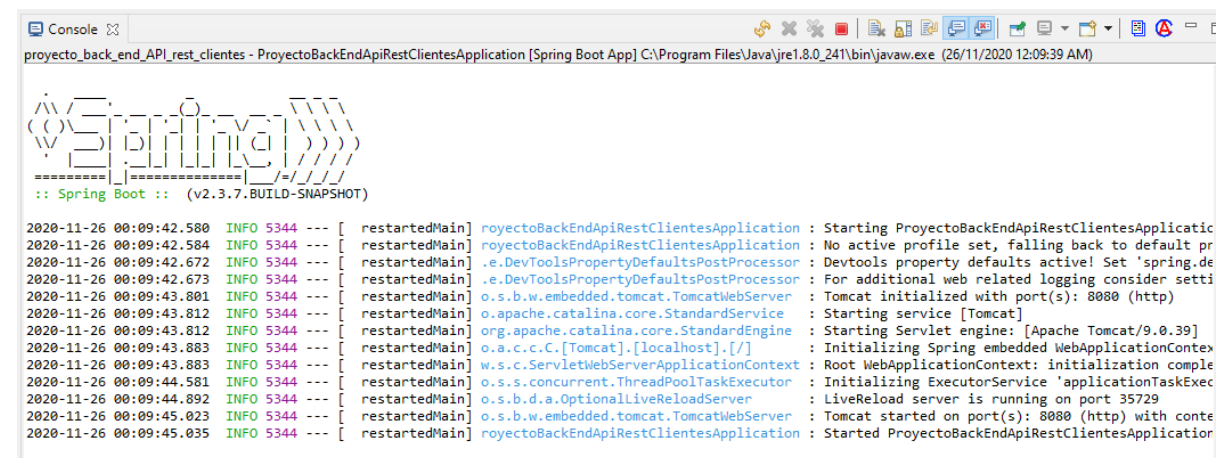
D: En el archivo `pom.xml` se encuentran alojadas las dependencias que utiliza el proyecto.

LANZAR LA APLICACIÓN WEB

El servidor de aplicaciones integrado en el IDE es Apache Tomcat.

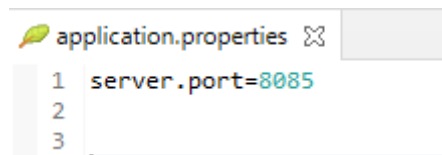


Log resultado de la ejecución



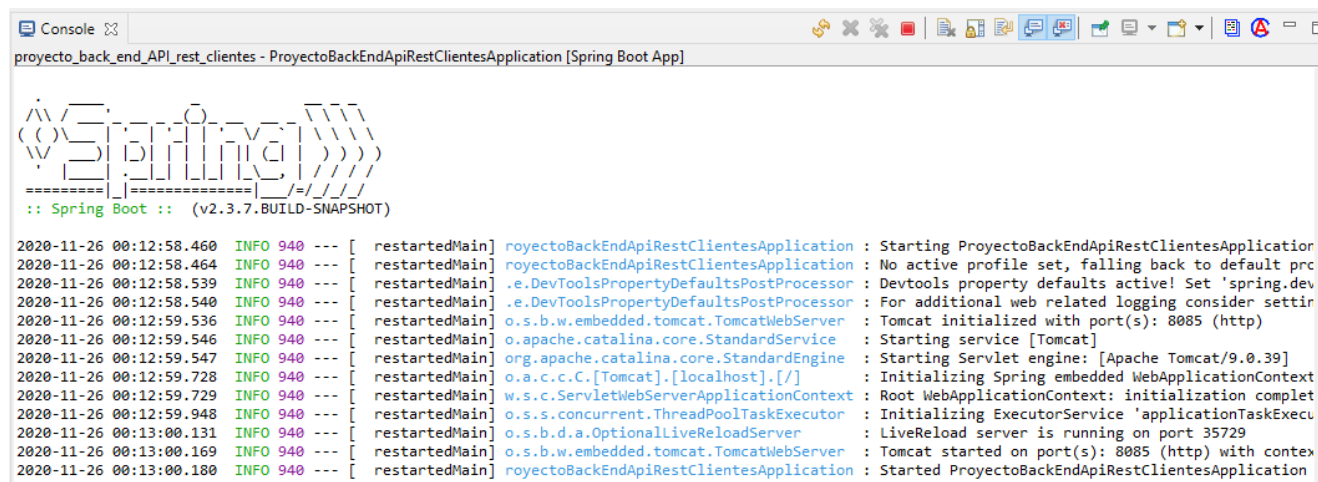
CAMBIAR EL PUERTO DE ESCUCHA

De forma predeterminada, el servidor integrado se inicia en el puerto 8080. Para cambiar el puerto de escucha debemos agregar la siguiente propiedad al archivo application.properties.



```
application.properties
1 server.port=8085
2
3
```

Log resultado de la ejecución



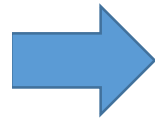
```
Console
proyecto_back_end_API_rest_clientes - ProyectoBackEndApiRestClientesApplication [Spring Boot App]

:: Spring Boot :: (v2.3.7.BUILD-SNAPSHOT)

2020-11-26 00:12:58.460 INFO 940 --- [ restartedMain] royectoBackEndApiRestClientesApplication : Starting ProyectoBackEndApiRestClientesApplication
2020-11-26 00:12:58.464 INFO 940 --- [ restartedMain] royectoBackEndApiRestClientesApplication : No active profile set, falling back to default pro
2020-11-26 00:12:58.539 INFO 940 --- [ restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : Devtools property defaults active! Set 'spring.dev
2020-11-26 00:12:58.540 INFO 940 --- [ restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : For additional web related logging consider settin
2020-11-26 00:12:59.536 INFO 940 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8085 (http)
2020-11-26 00:12:59.546 INFO 940 --- [ restartedMain] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2020-11-26 00:12:59.547 INFO 940 --- [ restartedMain] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.39]
2020-11-26 00:12:59.728 INFO 940 --- [ restartedMain] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2020-11-26 00:12:59.729 INFO 940 --- [ restartedMain] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization complet
2020-11-26 00:12:59.948 INFO 940 --- [ restartedMain] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecu
2020-11-26 00:13:00.131 INFO 940 --- [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on port 35729
2020-11-26 00:13:00.169 INFO 940 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8085 (http) with contex
2020-11-26 00:13:00.180 INFO 940 --- [ restartedMain] royectoBackEndApiRestClientesApplication : Started ProyectoBackEndApiRestClientesApplication
```

Crear los siguientes paquetes.

```
✓ java \ co \ edu \ unicauca \ distribuidos \ core  
  > capaAccesoADatos  
  > capaControladores  
  > fachadaServices
```



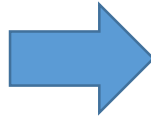
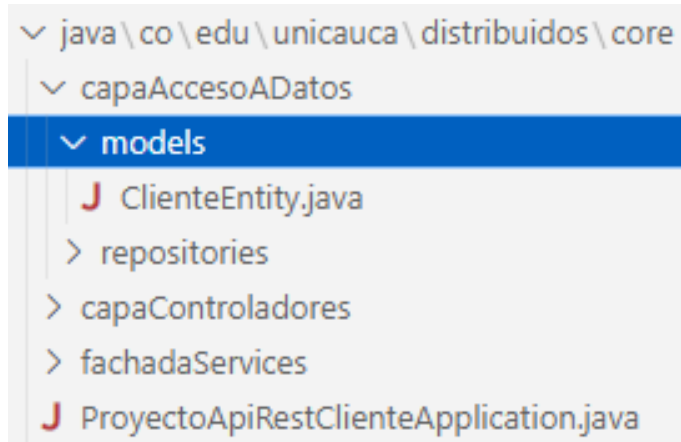
En el paquete **capaControladores** se crea el controlador que ofrecerá los servicios web.

En el paquete **capaAccesoADatos** se crean las clases e interfaces que dan acceso al medio de persistencia.

En el paquete **fachadaServices** se crean las clases e interfaces que se conectan con las clases del paquete **repositories**. El controlador utilizará las clases del paquete **services** con el objetivo de que no conozca el medio de persistencia.

En el paquete **models** se almacenan las clases asociadas a las entidades que se gestionan.

Crear la clase ClienteEntity en el paquete models



```
import java.util.Date;

import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.Setter;

@Getter @Setter @AllArgsConstructor
public class ClienteEntity {
    private Integer id;
    private String nombre;
    private String apellido;
    private String email;
    private Date createdAt;

    public ClienteEntity()
    {

    }
}
```

Anotaciones

El carácter de signo (@) indica al compilador que lo que sigue es una anotación. **@Service**, **@Repository** son anotaciones que permite que spring cree automáticamente objetos de una clase. Las anotaciones de Java permiten modificar la funcionalidad del programa.

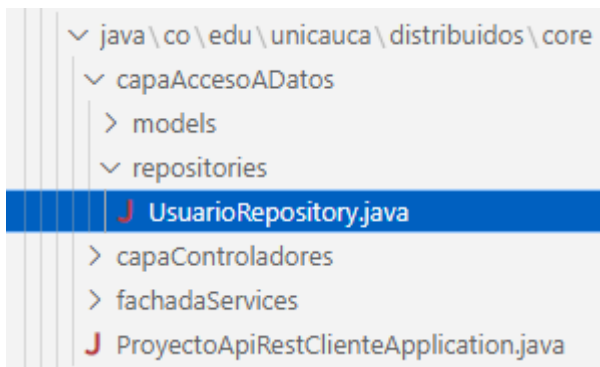
En la mayoría de las aplicaciones típicas, tenemos distintas capas como acceso a datos, presentación, servicio, negocios, etc.

Además, en cada capa tenemos varios beans. Para detectar estos beans automáticamente, Spring usa anotaciones de escaneo de classpath . Luego, registra cada bean en ApplicationContext .

A continuación, se ofrece una descripción general rápida de algunas de estas anotaciones:

- **@Component** es un estereotipo genérico para cualquier componente administrado por Spring.
- **@Service** anota clases en la capa de servicio.
- **@Repository** anota clases en la capa de persistencia, que actuará como un repositorio de la base de datos.

Crear la clase UsuarioRepository en el paquete repositories



```
@Repository
public class UsuarioRepository {

    private ArrayList<ClienteEntity> listaDeClientes;

    public UsuarioRepository()
    {
        this.listaDeClientes= new ArrayList<ClienteEntity>();
        cargarClientes();
    }
}
```

El carácter de signo (@) indica al compilador que lo que sigue es una anotación. @Service es una anotación que permite que spring cree automáticamente objetos de una clase. Las anotaciones de Java permiten modificar la funcionalidad del programa.

En la clase **UsuarioRepository** se deben crear los métodos para listar clientes, registrar cliente, consultar cliente, actualizar cliente y eliminar cliente. Además, debe crear un método para almacenar algunos clientes de prueba.

```
public List<ClienteEntity> findAll()
{
    System.out.println(x: "Invocando a listarclientes");
    return this.listaDeClientes;
}
```

```
public ClienteEntity save(ClienteEntity cliente)
{
    System.out.println(x: "Invocando a almacenar cliente");
    ClienteEntity objCliente=null;
    if (this.listaDeClientes.add(cliente))
    {
        objCliente=cliente;
    }
    return objCliente;
}
```

```
public ClienteEntity findById(Integer id)
{
    System.out.println(x: "Invocando a consultar un cliente");
    ClienteEntity objCliente=null;

    for (ClienteEntity cliente : listaDeClientes) {
        if(cliente.getId()==id)
        {
            objCliente=cliente;
            break;
        }
    }

    return objCliente;
}
```

```
public ClienteEntity update(Integer id, ClienteEntity cliente)
{
    System.out.println(x: "Invocando a actualizar un cliente");
    ClienteEntity objCliente=null;

    for (int i = 0; i < this.listaDeClientes.size(); i++) {
        if(this.listaDeClientes.get(i).getId()==id)
        {
            this.listaDeClientes.set(i,cliente);
            objCliente=cliente;
            break;
        }
    }

    return objCliente;
}
```

En la clase `UsuarioRepository` se deben crear los métodos para listar clientes, registrar cliente, consultar cliente, actualizar cliente y eliminar cliente. Además, debe crear un método para almacenar algunos clientes de prueba.

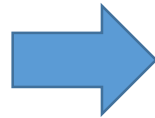
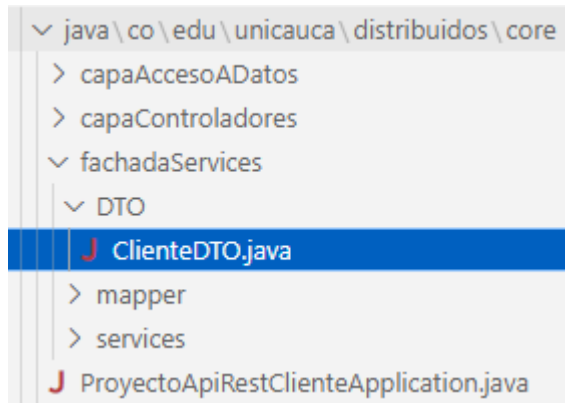
```
public boolean delete(Integer id)
{
    System.out.println(x: "Invocando a eliminar un cliente");
    boolean bandera=false;

    for (int i = 0; i < this.listaDeClientes.size(); i++) {
        if(this.listaDeClientes.get(i).getId()==id)
        {
            this.listaDeClientes.remove(i);
            bandera=true;
            break;
        }
    }

    return bandera;
}
```

```
private void cargarClientes()
{
    ClienteEntity objCliente1= new ClienteEntity(id: 1, nombre: "Juan", apellido: "Perez", em
this.listaDeClientes.add(objCliente1);
    ClienteEntity objCliente2= new ClienteEntity(id: 2, nombre: "Catalina", apellido: "Lopez"
this.listaDeClientes.add(objCliente2);
    ClienteEntity objCliente3= new ClienteEntity(id: 3, nombre: "Sandra", apellido: "Sanchez"
this.listaDeClientes.add(objCliente3);
    ClienteEntity objCliente= new ClienteEntity(id: 4, nombre: "Andres", apellido: "Perez", e
this.listaDeClientes.add(objCliente);
}
```

En el paquete `services` debe existir una clase que sigue el patrón DTO



```
import java.util.Date;

import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
@AllArgsConstructor
public class ClienteDTO {
    private Integer id;
    private String nombre;
    private String apellido;
    private String email;
    private Date createAt;

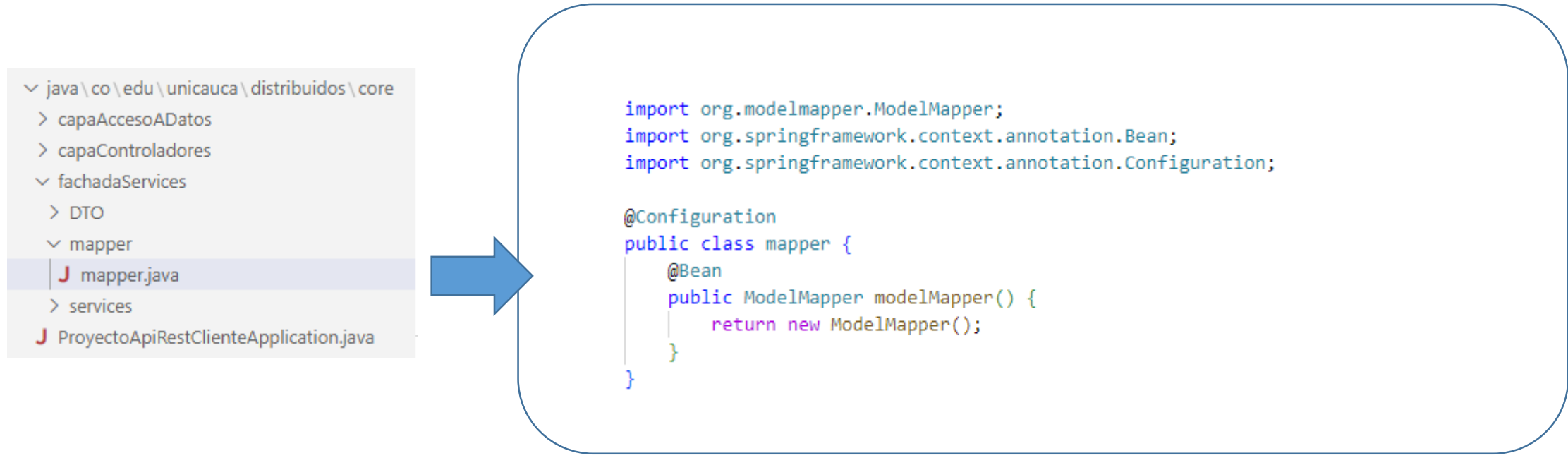
    public ClienteDTO() {

    }
}
```

En el paquete services debe crear una clase mapper que genera un objeto que permite mapear Entity a DTO.

@configuration indica que una clase tiene métodos que crean beans con **@Bean**, los beans creados serán singleton

@Bean permite crear beans

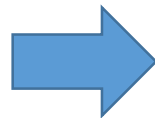
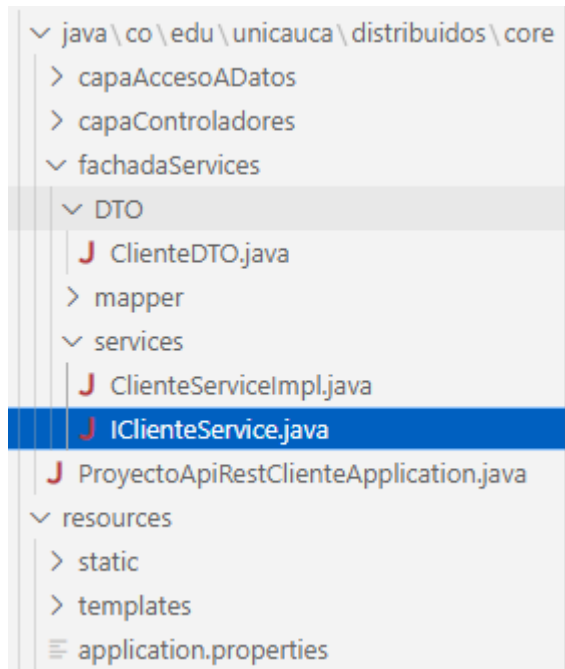


```
▼ java\co\edu\unicauca\distribuidos\core
  > capaAccesoADatos
  > capaControladores
  ▼ fachadaServices
    > DTO
    ▼ mapper
      J mapper.java
    > services
  J ProyectoApiRestClienteApplication.java
```

```
import org.modelmapper.ModelMapper;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class mapper {
    @Bean
    public ModelMapper modelMapper() {
        return new ModelMapper();
    }
}
```

En el paquete services debe crear una interface con la definición de los servicios que pueden ser consumidos por el controlador. La interface se denomina IClienteService.



```
public interface IClienteService {  
  
    public List<ClienteDTO> findAll();  
    public ClienteDTO findById(Integer id);  
    public ClienteDTO save(ClienteDTO cliente);  
    public ClienteDTO update(Integer id, ClienteDTO cliente);  
    public boolean delete(Integer id);  
}
```

En el paquete services cree una clase que implemente la interface IClienteService.

```
@Service
public class ClienteServiceImpl implements IClienteService {

    @Autowired
    private UsuarioRepository servicioAccesoBaseDatos;

    @Autowired
    private ModelMapper modelMapper;

    @Override
    public List<ClienteDTO> findAll() {

        List<ClienteEntity> clientesEntity= this.servicioAccesoBaseDatos.findAll();
        List<ClienteDTO> clientesDTO=this.modelMapper.map(clientesEntity, new TypeTo
        return clientesDTO;
    }
}
```

La anotación **@Service** permite que spring cree automáticamente objetos de una clase.

La anotación **@Autowired** hace que spring inyecte automáticamente un objeto de la clase UsuarioRepository, el objeto creado es almacenado en un contenedor de Beans y posteriormente es inyectado en el atributo servicioAccesoADatos..

En el paquete controllers cree una clase denominada **ClienteRestController**. Utilice las anotaciones **@RestController** y **@RequestMapping**.

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

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

}
```

La anotación **@RestController** indica que los métodos del controlador serán servicios que siguen el modelo REST.

La anotación **@RequestMapping** indica que las rutas para acceder a los servicios que siguen el modelo REST deben tener el prefijo api.

La URL de la petición es **localhost:8085/api/cliente/**

En el controlador denominado `ClienteRestController` declare un atributo de tipo `IClienteService` y utilice la notación `@Autowired` con el objetivo que Spring cree automáticamente un objeto de la clase que implementa a la interface `IClienteService`.

```
import java.util.List;

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


    @Autowired
    private IClienteService clienteService;
```

La anotación `@Autowired` hace que spring cree automáticamente un objeto de la clase que implementa la interface `IClienteService`, posteriormente el objeto es almacenado en un contenedor de Beans y posteriormente es inyectado en el atributo `clienteService`..

Servicio web para listar clientes

La anotación **@GetMapping** asocia un método con un servicio REST que recibe peticiones mediante el verbo Get

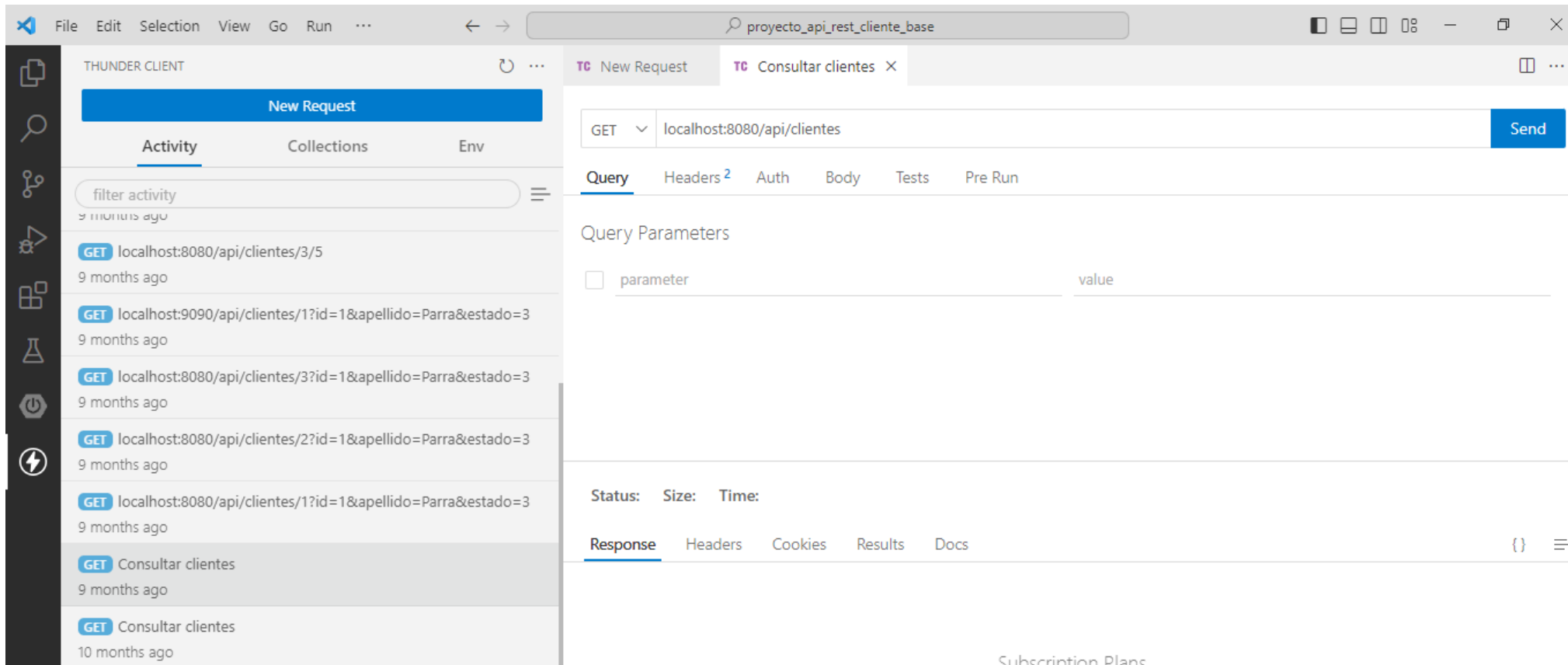
La ruta **/clientes** permite acceder al servicio web que retorna todo el listado de clientes.



```
@GetMapping("/clientes")
public List<ClienteDTO> index() {
    return clienteService.findAll();
}
```

El método utiliza el objeto inyectado el cual permite el acceso al medio de persistencia

Podemos utilizar clientes como postmap o thunder para probar las APIs



PRUEBA AL SERVICIO DE LISTAR CLIENTES

UNIVERSIDAD DEL CAUCA – FIET
DEPARTAMENTO DE SISTEMAS

La URL de la petición es `localhost:8085/api/clientes/`

Seleccionamos el verbo Get

Enviamos la petición

No enviamos nada en el cuerpo de la petición

Se listan los clientes en formato JSON que el servicio web retorna

Seleccionamos el formato JSON

The screenshot shows a REST client interface with the following elements:

- Request Bar:** Method: GET, URL: localhost:8085/api/clientes/. Buttons: Send, Save.
- Request Body:** Tab: Body, Format: raw. Content: 1.
- Response Section:** Status: 200 OK, Time: 4.88 s, Size: 533 B. Buttons: Save Response.
- Response Body:** Tab: Body, Format: JSON. Content:

```
{
  "id": 1,
  "nombre": "Juan",
  "apellido": "Perez",
  "email": "juan@unicauca.edu.co",
  "createAt": "2020-11-26T06:41:56.400+00:00"
},
{
  "id": 2,
  "nombre": "Catalina",
  "apellido": "Lopez",
  "email": "catalina@unicauca.edu.co",

```

Blue arrows indicate the workflow: selecting the GET method, sending the request, and selecting the JSON response format. A red box highlights the JSON format selector in the response body.

Como se representan en JSON los datos

UNIVERSIDAD DEL CAUCA – FIET
DEPARTAMENTO DE SISTEMAS

Clase persona

```
package Ejemplo2.modelo;

public class Persona {
    private String nombres;
    private String apellidos;
    private int edad;

    public Persona(String nombres, String apellidos, int edad)
    {...5 lines }

    public String getNombres() {...3 lines }

    public void setNombres(String nombres) {...3 lines }

    public String getApellidos() {...3 lines }

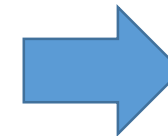
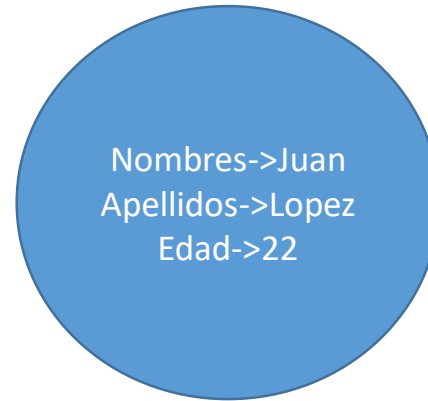
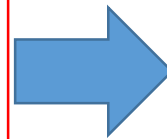
    public void setApellidos(String apellidos) {...3 lines }

    public int getEdad() {...3 lines }

    public void setEdad(int edad) {...3 lines }
}
```

Objeto de la clase persona

```
Persona objPersona= new Persona("Juan", "Lopez",22);
```



El estado de un objeto son los valores que toman los atributos de un objeto

JSON generado

```
{
  "nombres": "Juan",
  "apellidos": "Lopez",
  "edad": 22
}
```

JSON

```
{  
  "nombres": "Juan",  
  "apellidos": "Lopez",  
  "edad": 22  
}
```

- ❖ JSON representa objetos de manera textual mediante **parejas clave=valor**
- ❖ JSON requiere usar comillas dobles para las cadenas y los nombres de las claves. Las comillas simples no son válidas.
- ❖ JSON es sólo un formato de datos, contiene sólo nombres de claves y valores, no métodos
- ❖ Un objeto se representa como una secuencia de parejas clave=valor encerradas entre llaves { }.
- ❖ Las claves son cadenas de texto entre comillas " ".

Visualizar un JSON

UNIVERSIDAD DEL CAUCA – FIET
DEPARTAMENTO DE SISTEMAS

<https://jsoneditoronline.org/>

The screenshot displays the JSON Editor Online interface, which is split into two main panels: 'New document 1' on the left and 'New document 2' on the right. Both panels have a top bar with a 'code' tab selected and a 'tree' tab. The left panel shows a JSON document in code view, with line numbers 1 through 28. The right panel shows the same JSON document in tree view, with a search bar and a 'Diff' button. The JSON data is as follows:

```
{
  "nombres": "Juan",
  "apellidos": "Lopez",
  "edad": 22,
  "objDireccion": {
    "calle": "Calle 5 no 25 A 32",
    "ciudad": "Popayán",
    "departamento": "Cauca",
    "codigoPostal": "9024"
  },
  "telefonos": [
    {
      "tipo": "Teléfono de casa",
      "numero": "8302529"
    },
    {
      "tipo": "Teléfono movil",
      "numero": "3125467832"
    },
    {
      "tipo": "Teléfono trabajo",
      "numero": "3152753458"
    }
  ]
}
```

Servicio web para consultar un cliente

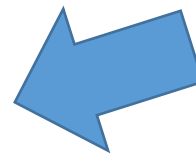
La anotación **@GetMapping** asocia un método con un servicio REST que recibe peticiones mediante el verbo Get



La ruta **/clientes/{id}** permite acceder al servicio web que consulta un cliente. La ruta recibe un parámetro el cual es el id del cliente a consultar.



```
@GetMapping("/clientes/{id}")
public ClienteDTO show(@PathVariable Integer id) {
    ClienteDTO objCliente = null;
    objCliente = clienteService.findById(id);
    return objCliente;
}
```



La anotación **@PathVariable** asocia el parámetro de un método con el parámetro de la URL

<https://www.baeldung.com/spring-pathvariable>




El método utiliza el objeto inyectado el cual permite el acceso al medio de persistencia

Servicio web para crear un cliente

La anotación **@PostMapping** asocia un método con un servicio REST que recibe peticiones mediante el verbo Post

La ruta **/clientes** permite acceder al servicio web que crea un cliente

La anotación **@RequestBody** asocia el parámetro de un método con los datos almacenados en el cuerpo de la petición.



```
@PostMapping("/clientes")
public ClienteDTO create(@RequestBody ClienteDTO cliente) {
    ClienteDTO objCliente = null;
    objCliente = clienteService.save(cliente);
    return objCliente;
}
```

El método utiliza el objeto inyectado el cual permite el acceso al medio de persistencia

Servicio web para actualizar un cliente

La anotación **@PutMapping** asocia un método con un servicio REST que recibe peticiones mediante el verbo Put.



La ruta **/clientes/{id}** permite acceder al servicio web que actualiza un cliente. El parámetro id representa el id del cliente a actualizar.



La anotación **@RequestBody** asocia el parámetro de un método con los datos almacenados en el cuerpo de la petición.



```
@PutMapping("/clientes/{id}")
public ClienteDTO update(@RequestBody ClienteDTO cliente, @PathVariable Integer id) {
    ClienteDTO objCliente = null;
    ClienteDTO clienteActual = clienteService.findById(id);
    if(clienteActual!=null)
    {
        objCliente = clienteService.update(id,cliente);
    }
    return objCliente;
}
```

Servicio web para eliminar un cliente

La anotación **@DeleteMapping** asocia un método con un servicio REST que recibe peticiones mediante el verbo Delete.



La ruta **/clientes/{id}** permite acceder al servicio web que elimina un cliente. El parámetro id representa el id del cliente a eliminar.

```
@DeleteMapping("/clientes/{id}")
public Boolean delete(@PathVariable Integer id) {
    Boolean bandera=false;
    ClienteDTO clienteActual = clienteService.findById(id);
    if(clienteActual!=null)
    {
        bandera = clienteService.delete(id);
    }
    return bandera;
}
```

PRUEBAS CON POSTMAN DEL SERVICIO WEB RESTFUL



PRUEBA AL SERVICIO DE LISTAR CLIENTES

UNIVERSIDAD DEL CAUCA – FIET
DEPARTAMENTO DE SISTEMAS

La URL de la petición es `localhost:8085/api/clientes/`

Seleccionamos el verbo Get

Enviamos la petición

No enviamos nada en el cuerpo de la petición

Se listan los clientes en formato JSON que el servicio web retorna

Seleccionamos el formato JSON

The screenshot shows a REST client interface with the following elements:

- Request Bar:** Method: GET, URL: localhost:8085/api/clientes/. Buttons: Send, Save.
- Request Body:** Tab: Body, Format: raw. Content: 1.
- Response Section:** Status: 200 OK, Time: 4.88 s, Size: 533 B. Buttons: Save Response.
- Response Body:** Tab: Body, Format: JSON. Content:

```
{
  "id": 1,
  "nombre": "Juan",
  "apellido": "Perez",
  "email": "juan@unicauca.edu.co",
  "createAt": "2020-11-26T06:41:56.400+00:00"
},
{
  "id": 2,
  "nombre": "Catalina",
  "apellido": "Lopez",
  "email": "catalina@unicauca.edu.co",

```

Blue arrows indicate the workflow: selecting the GET method, sending the request, and selecting the JSON format for the response. A red box highlights the JSON format selector in the response body.

PRUEBA AL SERVICIO DE CONSULTAR CLIENTE

UNIVERSIDAD DEL CAUCA – FIET
DEPARTAMENTO DE SISTEMAS

La URL de la petición es `localhost:8085/api/clientes/`, el parámetro corresponde al cliente a consultar.

Seleccionamos el verbo Get

No enviamos nada en el cuerpo de la petición

Cliente consultado

The screenshot shows a REST client interface with the following components:

- Method and URL:** A dropdown menu shows 'GET' and the URL bar contains 'localhost:8085/api/clientes/1'.
- Request Body:** The 'Body' tab is selected, showing 'none' as the content type. The body text area is empty.
- Response:** The 'Body' tab is selected, showing a JSON response. The response is formatted as JSON, and the 'JSON' button is highlighted with a red box.
- Status Bar:** The status bar at the bottom right shows 'Status: 200 OK', 'Time: 87 ms', and 'Size: 281 B'.

The JSON response is as follows:

```
{
  "id": 1,
  "nombre": "Juan",
  "apellido": "Perez",
  "email": "juan@unicauca.edu.co",
  "createAt": "2020-11-26T06:41:56.400+00:00"
}
```

PRUEBA AL SERVICIO DE CREAR CLIENTE

UNIVERSIDAD DEL CAUCA – FIET
DEPARTAMENTO DE SISTEMAS

La URL de la petición es `localhost:8085/api/clientes/`

Seleccionamos el verbo Post

The screenshot shows a REST client interface with the following components:

- Method and URL:** A dropdown menu shows "POST" and the URL field contains "localhost:8085/api/clientes/".
- Body Tab:** The "Body" tab is selected, showing a JSON payload:

```
{  "id": 4,  "nombre": "Andrea",  "apellido": "Lopez",  "email": "adnrea@unicauca.edu.co",  "createAt": "2020-05-26T06:41:56.400+00:00"}
```
- Response Section:** Below the request, the "Body" tab shows the response in "Pretty" format:

```
{  "id": 4,  "nombre": "Andrea",  "apellido": "Lopez",  "email": "adnrea@unicauca.edu.co",  "createAt": "2020-05-26T06:41:56.400+00:00"}
```
- Status Bar:** At the bottom right, it displays "Status: 200 OK", "Time: 103 ms", and "Size: 285 B".

Enviamos en el cuerpo de la petición los datos del cliente a registrar

Cliente registrado

PRUEBA AL SERVICIO DE ACTUALIZAR CLIENTE

UNIVERSIDAD DEL CAUCA – FIET
DEPARTAMENTO DE SISTEMAS

Seleccionamos el verbo Put

La URL de la petición es `localhost:8085/api/clientes/` y el parámetro indica el id del cliente a actualizar.

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

- Method:** PUT
- URL:** localhost:8085/api/clientes/3
- Body Type:** JSON
- Body Content:**

```
1 {  
2   "id": 6,  
3   "nombre": "Camilo",  
4   "apellido": "Perez",  
5   "email": "camilo@unicauca.edu.co",  
6   "createAt": "2020-12-26T06:41:56.400+00:00"  
7 }
```
- Response:** Status: 200 OK, Time: 45 ms, Size: 285 B
- Response Body (Pretty):**

```
1 {  
2   "id": 6,  
3   "nombre": "Camilo",  
4   "apellido": "Perez",  
5   "email": "camilo@unicauca.edu.co",  
6   "createAt": "2020-12-26T06:41:56.400+00:00"  
7 }
```

Enviamos en el cuerpo de la petición los datos del cliente a actualizar

Cliente actualizado

PRUEBA AL SERVICIO DE ELIMINAR CLIENTE

UNIVERSIDAD DEL CAUCA – FIET
DEPARTAMENTO DE SISTEMAS

Seleccionamos el verbo Delete

La URL de la petición es `localhost:8085/api/clientes/` y el parámetro indica el id del cliente a eliminar.

En el cuerpo de la petición no enviamos nada.

Respuesta de la eliminación

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

- Request Bar:** Shows the method `DELETE` and the URL `localhost:8085/api/clientes/4`. Buttons for `Send` and `Save` are on the right.
- Request Tabs:** Includes `Params`, `Authorization`, `Headers (6)`, `Body` (selected), `Pre-request Script`, `Tests`, and `Settings`. There are also links for `Cookies` and `Code`.
- Request Body:** Under the `Body` tab, the format is set to `JSON`. The body content is empty, with a line number `1` visible in the left margin.
- Response Section:** Located at the bottom, it includes tabs for `Body` (selected), `Cookies`, `Headers (5)`, and `Test Results`. The status bar shows `Status: 200 OK`, `Time: 11 ms`, and `Size: 168 B`, with a `Save Response` button.
- Response Body:** Under the `Body` tab, the format is set to `Pretty`. The response content is `1 true`.

Muchas gracias
Preguntas

