# API REST Productos SPRING BOOT.

Vamos a crear el APIREST que permita hacer la facturación 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 Datos
6. Cree un repositorio de datos de Spring - ProductoRepositorio.java
7. Capa de servicio (usa repositorio)
8. Creación de las API: ProductoControlador
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 agregando las dependencias Spring web, devTools y Spring Data MongoDB.

## 2. Agregar Swagger al Proyecto.

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>
```

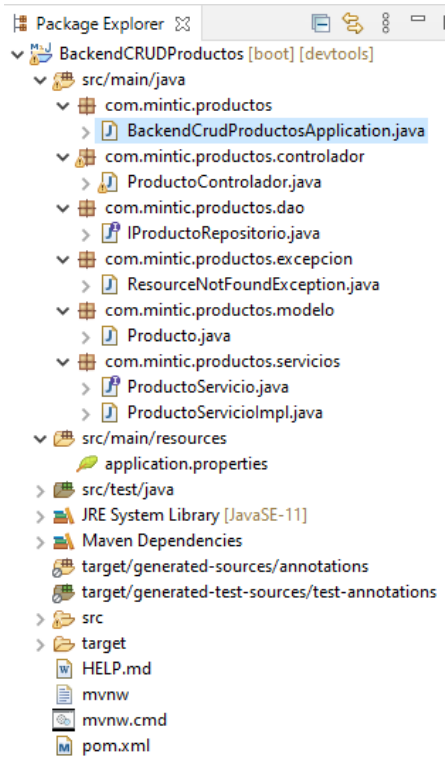
En la clase principal se agrega la anotación "@EnableSwagger2" para activar el uso de la herramienta, además se debe agregar un código que nos permite agregar una arquitectura MVC con swagger al archivo de de ejecución, la documentación de la herramienta se encuentra en el siguiente url:

El contenido del archivo sería el siguiente:

```java
package com.mintic.productos;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

import springfox.documentation.builders.PathSelectors;
import springfox.documentation.builders.RequestHandlerSelectors;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;
import springfox.documentation.swagger2.annotations.EnableSwagger2;


@SpringBootApplication
@EnableSwagger2
public class BackendCrudProductosApplication {

    @Bean
    public Docket productosApi() {
        return new Docket(DocumentationType.SWAGGER_2)
                .select()
                .apis(RequestHandlerSelectors.any())
                .paths(PathSelectors.any())
                .build();
    }

    public static void main(String[] args) {
        SpringApplication.run(BackendCrudProductosApplication.class, args);
    }

}
```

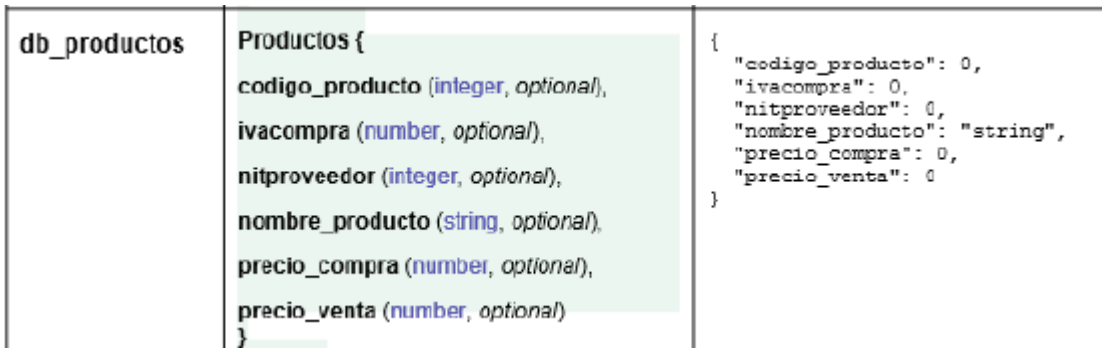
## 3. Creación de la Estructura del Proyecto.

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.

Ya es de conocimiento el manejo de spring boot con Mongo Db, basado en ello creamos la definición del url de la base de datos(Colección), se dará una estructura a la base de como se aprecia a continuación:




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

```
spring.data.mongodb.uri=mongodb://localhost:27017/db_productos00
server.port=8083

```

## 5. Creación del modelo de Datos

Ahora creemos el modelo de Datos, debemos entender que la clase Productos contiene algunos campos cuyo nombre incluye guiones de piso, por ejemplo, el nombre código_producto lo vamos a cambiar por codigoproducto para facilitar la definición del método que realice la búsqueda por código. No olvidar definir constructores, setter y getter.

```java
package com.mintic.productos.modelo;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

@Document(collection = "productos")
public class Producto {

    @Id
    private String _id;
    private int codigoproducto;
    private double ivacompra;
    private int nitproveedor;
    private String nombre_producto;
    private double precio_compra;
    private double precio_venta;
```

## 6. Cree un repositorio de datos de Spring - ProductosRepositorio.java.

A continuación, necesitamos crear ProductosRepositorio 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.

```java
package com.mintic.productos.dao;

import org.springframework.data.mongodb.repository.MongoRepository;
import org.springframework.stereotype.Repository;

import com.mintic.productos.modelo.Producto;

@Repository
public interface IProductoRepositorio extends MongoRepository<Producto, String> {


    public Producto findByCodigoproducto(int codigoproducto);


}
```

## 7. Capa de servicio (usa repositorio).

Primero vamos a crear la interface con las cabeceras de los métodos a implementar.

```java
package com.mintic.productos.servicios;

import java.util.List;

import com.mintic.productos.modelo.Producto;


public interface ProductoServicio {

    // Cabeceras de métodos a utilizar en la colección Productos
    // Crear
    Producto crearProducto(Producto producto);

    //Actualizar
    Producto updateProducto(Producto producto);

    //Listado de Productos
    List<Producto> getAllProducto();

    //Buscar Producto por _id
    Producto getProductoById(String productoId);

    //Buscar Producto por código
    Producto buscarByCodigo(int codigo);

    //Eliminar un Producto
    void deleteProducto(String productoId);

}
```

## Clase "ProductosServicioImpl".

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

```java
package com.mintic.productos.servicios;

import java.util.List;
import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import com.mintic.productos.dao.IProductoRepositorio;
import com.mintic.productos.excepcion.ResourceNotFoundException;
import com.mintic.productos.modelo.Producto;

@Service
@Transactional
public class ProductoServicioImpl implements ProductoServicio {

    @Autowired
    private IProductoRepositorio productoRepo;

    @Override
    public Producto crearProducto(Producto producto) {
        return productoRepo.save(producto);
    }

    @Override
    public Producto updateProducto(Producto producto) {
        // Se busca el cliente con el Id del objeto cliente recibido
        Optional<Producto> productoDb = this.productoRepo.findById(producto.get_id());

        if(productoDb.isPresent()) {
            // Se crea un objeto tipo Producto con los datos recuperados
            Producto productoUpdate = productoDb.get();
            // Se actualiza el valor de cada atributo con el get correspondiente de producto
            productoUpdate.set_id(producto.get_id());
            productoUpdate.setCodigoproducto(producto.getCodigoproducto());
            productoUpdate.setIvacompra(producto.getIvacompra());
            productoUpdate.setNitproveedor(producto.getNitproveedor());
            productoUpdate.setNombre_producto(producto.getNombre_producto());
            productoUpdate.setPrecio_compra(producto.getPrecio_compra());
            productoUpdate.setPrecio_venta(producto.getPrecio_venta());
            // Se actualizan los valores del objeto producto
            productoRepo.save(productoUpdate);
            return productoUpdate;
        }else {
            throw new ResourceNotFoundException("Registro no Encontrado con el Id:"+producto.get_id());
        }
    }

    @Override
    public List<Producto> getAllProducto() {
        // Crea la lista de los Productos
        return productoRepo.findAll();
    }
```

```
    @Override
    public Producto getProductoById(String productoId) {
        // Búsqueda de producto por _id
        Optional<Producto> productoDb = this.productoRepo.findById(productoId);
        if(productoDb.isPresent()) {
            return productoDb.get();
        }else {
            throw new ResourceNotFoundException("Registro no Encontrado con el Id:"+productoId);
        }
    }

    @Override
    public Producto buscarByCodigo(int codigo) {
        Producto productoDb = this.productoRepo.findByCodigoproducto(codigo);
        if(productoDb != null) {
            // Si lo encuentra lo RETORNA
            return productoDb;
        }else {
            throw new ResourceNotFoundException("Registro no Encontrado con el código:"+codigo);
        }
    }

    @Override
    public void deleteProducto(String productoId) {
        // Busca primero
        Optional<Producto> productoDb = this.productoRepo.findById(productoId);
        if(productoDb.isPresent()) {
            // Si lo encuentra lo borra
            this.productoRepo.delete(productoDb.get());
        }else {
            throw new ResourceNotFoundException("Registro no Encontrado con el Id:"+productoId);
        }
    }
}
```

- **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.

```
package com.mintic.clientes.excepcion;

import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus
public class ResourceNotFoundException extends RuntimeException{

    private static final long serialVersionUID = 1L;

    public ResourceNotFoundException(String message) {
        super(message);
    }

    public ResourceNotFoundException(String message, Throwable throwable) {
        super(message, throwable);
    }

}
```

## 8. Creación de las API: ProductosControlador.

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 ProductoControlador con las anotaciones de @RestController, @CrossOrigin y @RequestMapping("/api/productos").

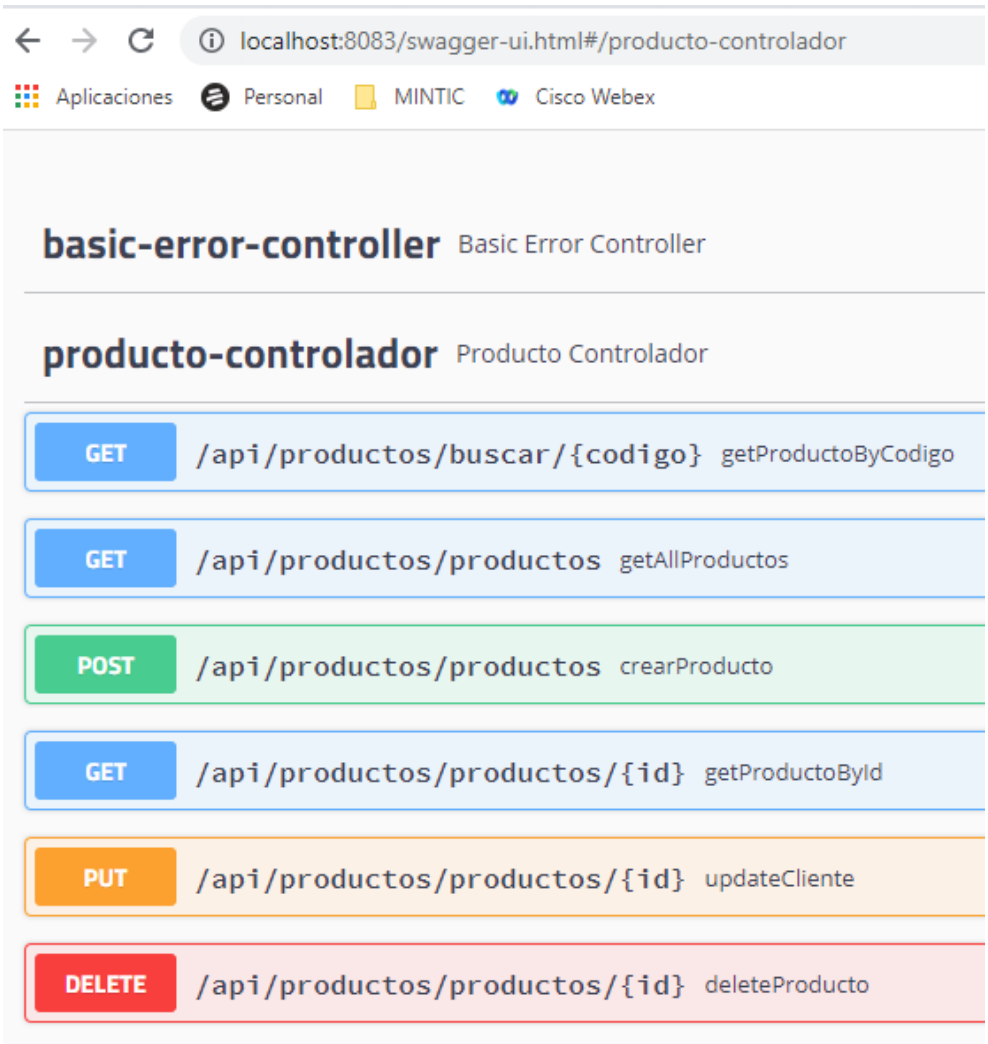
```java
package com.mintic.productos.controlador;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
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.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import com.mintic.productos.modelo.Producto;
import com.mintic.productos.servicios.ProductoServicio;

@RestController
@CrossOrigin(origins = "*", methods = { RequestMethod.POST, RequestMethod.GET, RequestMethod.PUT,
        RequestMethod.DELETE })
@RequestMapping("/api/productos")
public class ProductoControlador {

    @Autowired
    ProductoServicio productoServicio;

    @GetMapping("/productos")
    public ResponseEntity<List<Producto>> getAllProductos(){
        return ResponseEntity.ok().body(productoServicio.getAllProducto());
    }

    @GetMapping("/productos/{id}")
    public ResponseEntity<Producto> getProductoById(@PathVariable String id) {
        return ResponseEntity.ok().body(productoServicio.getProductoById(id));
    }

    @GetMapping("/buscar/{codigo}")
    public ResponseEntity<Producto> getProductoByCodigo(@PathVariable int codigo) {
        Producto productoData = productoServicio.buscarByCodigo(codigo);

        if (productoData != null) {
          return ResponseEntity.ok().body(productoData);
        } else {
          return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
    }

    @PostMapping("/productos")
    public ResponseEntity < Producto > crearProducto(@RequestBody Producto producto){
        return ResponseEntity.ok().body(this.productoServicio.crearProducto(producto));
    }

    @PutMapping("/productos/{id}")
    public ResponseEntity<Producto> updateCliente(@PathVariable String id, @RequestBody Producto producto){
        producto.set_id(id);
        return ResponseEntity.ok().body(this.productoServicio.updateProducto(producto));
    }

    @DeleteMapping("/productos/{id}")
    public HttpStatus deleteProducto(@PathVariable String id) {
        this.productoServicio.deleteProducto(id);
        return HttpStatus.OK;
    }
}
```

## 9. Ejecución de la aplicación de arranque Spring.

Cambiemos el puerto de ejecución a 8083 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.

## 10. Pruebe las API REST con Swagger.

Para probar el proyecto nos vamos a un navegador y escribimos http://localhost:8083/swagger-ui.html#/ nos debe aparecer una figura como la siguiente:




En el drive del curso pueden encontrar el archivo comprimido del proyecto, el nombre del archivo es "BackendCRUDProductos.zip".