

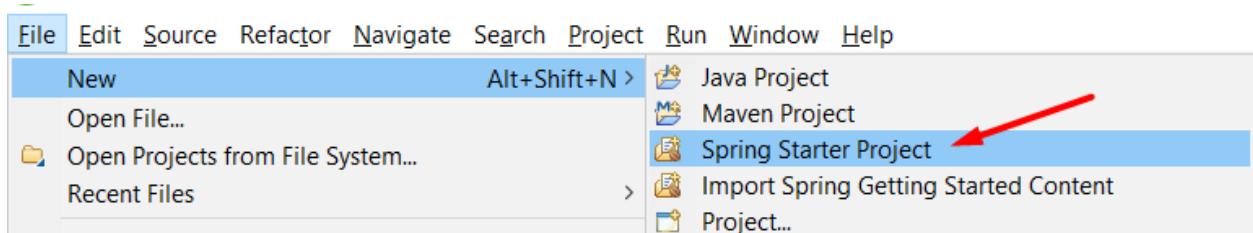
PROYECTO “cache-demo”

Utiliza **Spring Cache** para mejorar el rendimiento almacenando en caché de los productos y evitar accesos innecesarios a la base de datos.

Paso 1: Crear el Proyecto en STS

Abrir Spring Tool Suite (STS).

Ir a **File → New → Spring Starter Project**.



Completar los datos del proyecto:

- **Name:** cache-demo
- **Type:** Maven
- **Java Version:** 17
- **Group:** com.example
- **Artifact:** cache-demo
- **Packaging:** Jar

A screenshot of the 'New Spring Starter Project' dialog box in STS. The 'Service URL' is set to 'https://start.spring.io'. The 'Name' field contains 'cache-demo'. The 'Use default location' checkbox is checked. The 'Location' field shows 'C:\Users\JR\Documents\workspace\cache-demo'. The 'Type' is 'Maven', 'Packaging' is 'Jar', 'Java Version' is '17', and 'Language' is 'Java'. The 'Group' is 'com.example', 'Artifact' is 'cache-demo', 'Version' is '0.0.1-SNAPSHOT', 'Description' is 'Demo project for Spring Boot', and 'Package' is 'com.example.cachedemo'. There is a 'Browse' button next to the location field.

- **Dependencies:**
 - **Spring Web** (Para la API REST).
 - **Spring Boot DevTools** (Para recarga en vivo).
 - **Spring Cache Abstraction** (Para @Cacheable y @CacheEvict).

A screenshot of the 'Add Dependencies' dialog box in STS. It shows a list of dependencies with checkboxes. The checked dependencies are 'Spring Boot DevTools' and 'Spring Web'. The unchecked dependencies are 'H2 Database', 'Spring Boot Actuator', 'Spring Data JPA', 'Lombok', 'PostgreSQL Driver', 'Spring Cache Abstraction', and 'Spring Web Services'.

Hacer clic en **Finish** para generar el proyecto.

Paso 2: Configurar application.properties

Abre el archivo **src/main/resources/application.properties** y agrega:



```
spring.application.name=cache-demo
```

Explicación:

- **spring.application.name:** Define el nombre de la aplicación Spring Boot. En este caso, la aplicación se llama gardening-store. Este nombre puede ser utilizado para identificar la aplicación en logs, métricas, etc.



```
# PostgreSQL Configuration
```

```
spring.datasource.url=jdbc:postgresql://aws-0-us-west-1.pooler.supabase.com:6543/postgres
```

```
spring.datasource.username=postgres.wluwoborplxwkdfppdck
```

```
spring.datasource.password=admin
```

```
spring.datasource.driver-class-name=org.postgresql.Driver
```

Explicación:

- **spring.datasource.url:** Especifica la URL de conexión a la base de datos PostgreSQL.
- **spring.datasource.username:** Es el nombre de usuario para conectarse a la base de datos.
- **spring.datasource.password:** Es la contraseña para el usuario de la base de datos.
- **spring.datasource.driver-class-name:** Especifica el driver JDBC que se utilizará para conectarse a la base de datos.



```
# JPA Configuration
```

```
spring.jpa.generate-ddl=true
```

```
spring.jpa.hibernate.ddl-auto=update
```

```
spring.jpa.show-sql=true
```

```
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
```

Explicación:

- **spring.jpa.generate-ddl:** Habilita la generación automática del esquema de la base de datos a partir de las entidades JPA.
- **spring.jpa.hibernate.ddl-auto:** Define el comportamiento de Hibernate con respecto a la generación del esquema de la base de datos. update significa que Hibernate actualizará el esquema de la base de datos si es necesario, pero no borrará datos existentes.
- **spring.jpa.show-sql:** Habilita la impresión de las sentencias SQL generadas por Hibernate en la consola.
- **spring.jpa.properties.hibernate.dialect:** Especifica el dialecto de SQL que Hibernate debe usar para generar las consultas SQL.

Paso 3: Crear la Entidad Product

Creamos una clase Product para simular productos en nuestra API.



```
package com.example.cachedemo.model;

import jakarta.persistence.*;
import lombok.*;

@Entity
@Table(name = "products")
@Getter @Setter
@NoArgsConstructor @AllArgsConstructor
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false)
    private Double price;
}
```

Paso 4: Crear el Repository

Creamos el archivo **ProductRepository.java**:



```
package com.example.cachedemo.repository;

import com.example.cachedemo.model.Product;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface ProductRepository extends JpaRepository<Product, Long> {
}
```

Paso 5: Crear el service

Crear el archivo **ProductService.java**:



```
package com.example.cachedemo.service;

import com.example.cachedemo.model.Product;
import com.example.cachedemo.repository.ProductRepository;
import org.springframework.cache.annotation.CacheEvict;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Service;

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

@Service
public class ProductService {

    private final ProductRepository productRepository;

    public ProductService(ProductRepository productRepository) {
        this.productRepository = productRepository;
    }

    @Cacheable(value = "products", key = "#id")
    public Optional<Product> getProductById(Long id) {
        System.out.println("Recuperando producto de la base de datos... ID: " + id);
        return productRepository.findById(id);
    }

    public Product addProduct(Product product) {
        return productRepository.save(product);
    }

    @CacheEvict(value = "products", key = "#id")
    public void deleteProduct(Long id) {
        System.out.println("Eliminando producto de la base de datos... ID: " + id);
        productRepository.deleteById(id);
    }

    public List<Product> getAllProducts() {
        return productRepository.findAll();
    }
}
```

Explicación:

- **@Cacheable(value = "products", key = "#id")**: Antes de buscar el producto en la base de datos, Spring Cache verifica si ya está en la caché.
 - Si está en la caché, lo devuelve sin consultar la base de datos.
 - Si **NO** está en la caché, lo recupera de la base de datos y lo almacena en caché.
- **@CacheEvict(value = "products", key = "#id")**: Elimina el producto de la caché cuando se borra de la base de datos.
 - Evita que un producto eliminado siga estando en caché y sea devuelto en futuras consultas.

Paso 6: Crear el Controlador REST

Crear ProductController:



```
package com.example.cachedemo.controller;

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

import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import com.example.cachedemo.model.Product;
import com.example.cachedemo.service.ProductService;

@RestController
@RequestMapping("/products")
class ProductController {

    private final ProductService productService;

    public ProductController(ProductService productService) {
        this.productService = productService;
    }

    @GetMapping("/{id}")
    public ResponseEntity<Optional<Product>> getProduct(@PathVariable Long id) {
        return ResponseEntity.ok(productService.getProductById(id));
    }

    @PostMapping
    public ResponseEntity<Product> addProduct(@RequestBody Product product) {
        return ResponseEntity.ok(productService.addProduct(product));
    }


    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteProduct(@PathVariable Long id) {
        productService.deleteProduct(id);
        return ResponseEntity.noContent().build();
    }

    @GetMapping
    public ResponseEntity<List<Product>> getAllProducts() {
        return ResponseEntity.ok(productService.getAllProducts());
    }
}
```

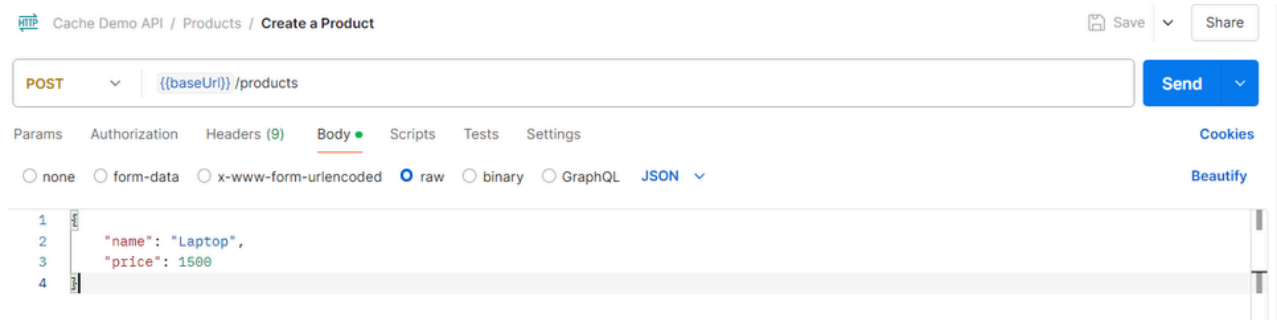
Paso 8: Ejecutar y Probar

Crear un producto (POST):

- **POST** http://localhost:8080/products



```
{  
  "name": "Laptop",  
  "price": 1500  
}
```



Obtener el producto por Id, a la **primera llamada lo consulta de la base de datos y a la segunda del cache** (GET):

- **GET** http://localhost:8080/products/5 (GET)



Elimina el producto por Id, **del cache** (DELETE):

- **DELETE** http://localhost:8080/products/4

