default CacheManager (ConcurrentMapCacheManager).

1. **Open pom.xml:**
   - Locate your project's pom.xml file in the root directory.
2. **Add the Caching Starter Dependency:**
   - Add the following inside the <dependencies> section:

```
<!-- pom.xml -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
```

3. **Reload Maven Project:**
   - Your IDE will prompt you to reload the project to download the new libraries. Confirm this action.

Step 2: Enable Caching in Your Application

To activate Spring's caching functionality, you need to enable it in your main Spring Boot application class.

1. **Open src/main/java/com/example/demo/DemoApplication.java:**
2. **Add @EnableCaching annotation:**
   - Add the @EnableCaching annotation to your main application class.

```
// src/main/java/com/example/demo/DemoApplication.java
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cache.annotation.EnableCaching; // Import this

@SpringBootApplication
@EnableCaching // Enable Spring's caching capabilities
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringApplication.class, args);
    }
```

```
}
```

- **Explanation:** This annotation tells Spring to process caching annotations on your beans and create the necessary AOP proxies to intercept method calls for caching.

Step 3: Update Product Model for Caching

For objects to be effectively cached (especially in distributed caches like Redis, though not strictly required for ConcurrentMapCacheManager), they should implement Serializable.

1. **Open src/main/java/com/example/demo/model/Product.java:**
2. **Implement Serializable:**
   - Add implements Serializable to your Product class.

```
// src/main/java/com/example/demo/model/Product.java
package com.example.demo.model;

import jakarta.persistence.*;
import java.io.Serializable; // Import Serializable

@Entity
@Table(name = "PRODUCTS")
public class Product implements Serializable { // IMPORTANT: Implement
Serializable
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "NAME", nullable = false, length = 255)
    private String name;

    @Column(name = "PRICE", nullable = false)
    private double price;

    // Constructors
    public Product() {}
    public Product(String name, double price) {
        this.name = name;
        this.price = price;
    }
    // Added for ProductService's initial data setup
```

```
    public Product(Long id, String name, double price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }

    // Getters and Setters
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public double getPrice() { return price; }
    public void setPrice(double price) { this.price = price; }

    @Override
    public String toString() {
        return "Product{id=" + id + ", name='" + name + "', price=" + price + '}';
    }
}
```

- **Explanation:** Implementing Serializable ensures that Spring can convert your Product objects into a byte stream, which is necessary for storing them in various cache providers.

Step 4: Create a ProductService with Caching Logic

We will create a new ProductService layer. This service will encapsulate the business logic and apply caching annotations. It will interact with your ProductRepository (JPA-based) to fetch/save data to the actual database.

1. **Create a new package:** If you don't have one, create a service package inside src/main/java/com/example/demo (e.g., com.example.demo.service).
2. **Create ProductService.java:** Inside the service package, create ProductService.java and add the following code:

```
// src/main/java/com/example/demo/service/ProductService.java
package com.example.demo.service;


import com.example.demo.model.Product;
import com.example.demo.repository.ProductRepository; // Import your JPA repository
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cache.annotation.CacheEvict; // For cache eviction
import org.springframework.cache.annotation.CachePut; // For cache updates
import org.springframework.cache.annotation.Cacheable; // For read-through
caching
import org.springframework.stereotype.Service; // Mark as a service component
import org.springframework.transaction.annotation.Transactional; // For
transactional operations

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

@Service // Marks this class as a Spring service component
public class ProductService {

    private final ProductRepository productRepository; // Inject the JPA repository

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

    // @Cacheable: Cache the result of this method.
    // If "products" cache has key 'productId', return cached value.
    // Otherwise, execute method, store result, then return.
    @Cacheable(value = "products", key = "#productId")
    @Transactional(readOnly = true) // Read-only transaction for fetching
    public Optional<Product> getProductById(Long productId) {
        System.out.println("--- ProductService: Fetching product from DATABASE for
ID: " + productId + " ---");
        // Simulate a delay to clearly see caching effect
```

```
        try {
            Thread.sleep(1500);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        return productRepository.findById(productId);
    }

    // @CachePut: Always execute method, then update cache with returned value.
    // Key is product.id. This ensures the cache is updated with the latest data
    // after a create or update operation.
    @CachePut(value = "products", key = "#product.id")
    @Transactional // Transaction for saving
    public Product saveProduct(Product product) {
        System.out.println("--- ProductService: Saving/Updating product in
DATABASE: " + product.getName() + " ---");
        // Simulate a delay
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        return productRepository.save(product); // Delegates to JPA repository
    }

    // @CacheEvict: Evict an entry from the cache after method execution.
    // Key is productId. This invalidates the cache entry when a product is deleted.
    @CacheEvict(value = "products", key = "#productId")
    @Transactional // Transaction for deleting
    public void deleteProduct(Long productId) {
        System.out.println("--- ProductService: Deleting product from DATABASE for
```

```
ID: " + productId + " ---");
        // Simulate a delay
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        productRepository.deleteById(productId); // Delegates to JPA repository
    }

    // @CacheEvict with allEntries=true: Evict all entries from the specified cache.
    // Useful for a full cache refresh or when a mass update/delete occurs.
    @CacheEvict(value = "products", allEntries = true)
    public void clearAllProductsCache() {
        System.out.println("--- ProductService: Clearing ALL entries from 'products'
cache ---");
        // No database interaction here, just cache clearing
    }

    // Method to get all products (not cached in this example, but could be)
    @Transactional(readOnly = true)
    public List<Product> getAllProducts() {
        System.out.println("--- ProductService: Fetching ALL products from
DATABASE ---");
        // Simulate a delay
        try {
            Thread.sleep(800);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        return productRepository.findAll();
```

```
    }
}
```

- **Explanation:**
  - @Service: Marks this class as a Spring service.
  - @Autowired ProductRepository: Injects your JPA repository.
  - @Cacheable(value = "products", key = "#productId"): Caches the result of getProductById in a cache named "products" using the productId as the key.
  - @CachePut(value = "products", key = "#product.id"): Ensures that after saveProduct executes (for both new and updated products), the cache entry for that product ID is updated with the latest data.
  - @CacheEvict(value = "products", key = "#productId"): Removes the specific product from the cache after deleteProduct is called.
  - @CacheEvict(value = "products", allEntries = true): Clears the entire "products" cache.
  - System.out.println statements are added to visually confirm when the actual database call (via productRepository) is made.
  - Thread.sleep() calls simulate network/DB latency, making the caching effect more noticeable.
  - @Transactional: Ensures database operations are handled within a transaction.

Step 5: Modify ProductController to use ProductService

Now, we'll update your ProductController to use the new ProductService for all product-related operations.

1. **Open src/main/java/com/example/demo/controller/ProductController.java:**
2. **Make the following changes:**
   - **Replace** the ProductRepository injection with ProductService.
   - **Update all CRUD methods** to call the corresponding methods on productService.

- **Add a new endpoint** to trigger the clearAllProductsCache() method.
- For simplicity and focus on caching, the custom query endpoints from the previous JPA activity will be removed from the controller (they still exist in ProductRepository but won't be exposed here).

```
// src/main/java/com/example/demo/controller/ProductController.java
package com.example.demo.controller;

import com.example.demo.model.Product;
import com.example.demo.service.ProductService; // Import the new service
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

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

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

    private final ProductService productService; // Inject the ProductService

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

    // CREATE Product (POST /api/products)
    @PostMapping
    public ResponseEntity<Product> createProduct(@RequestBody Product
```

```
product) {
        Product savedProduct = productService.saveProduct(product); //
Delegates to service (with @CachePut)
        return new ResponseEntity<>(savedProduct, HttpStatus.CREATED);
    }

    // READ All Products (GET /api/products)
    @GetMapping
    public ResponseEntity<List<Product>> getAllProducts() {
        List<Product> products = productService.getAllProducts(); // Delegates to
service
        return new ResponseEntity<>(products, HttpStatus.OK);
    }

    // READ Product by ID (GET /api/products/{id})
    @GetMapping("/{id}")
    public ResponseEntity<Product> getProductById(@PathVariable Long id) {
        Optional<Product> product = productService.getProductById(id); //
Delegates to service (with @Cacheable)
        return product.map(value -> new ResponseEntity<>(value, HttpStatus.OK))
                .orElseGet(() -> new
ResponseEntity<>(HttpStatus.NOT_FOUND));
    }

    // UPDATE Product (PUT /api/products/{id})
    @PutMapping("/{id}")
    public ResponseEntity<Product> updateProduct(@PathVariable Long id,
@RequestBody Product productDetails) {
        // First, check if the product exists in the DB (or cache)
        Optional<Product> productOptional = productService.getProductById(id);
        if (productOptional.isPresent()) {
```

```
            Product existingProduct = productOptional.get();
            // Update the fields from the request body
            existingProduct.setName(productDetails.getName());
            existingProduct.setPrice(productDetails.getPrice());
            Product updatedProduct = productService.saveProduct(existingProduct);
// Delegates to service (with @CachePut)
            return new ResponseEntity<>(updatedProduct, HttpStatus.OK);
        } else {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND); // Product not
found to update
        }
    }

    // DELETE Product (DELETE /api/products/{id})
    @DeleteMapping("/{id}")
    public ResponseEntity<HttpStatus> deleteProduct(@PathVariable Long id) {
        // It's good practice to check existence before deleting,
        // though deleteById in repository would also handle not found gracefully.
        if (productService.getProductById(id).isPresent()) { // Check existence
(might be cached)
            productService.deleteProduct(id); // Delegates to service (with
@CacheEvict)
            return new ResponseEntity<>(HttpStatus.NO_CONTENT); // Successful
deletion
        } else {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND); // Product not
found to delete
        }
    }

    // NEW Endpoint: Clear all entries from the "products" cache
```

```
    // POST /api/products/clear-cache
    @PostMapping("/clear-cache")
    public ResponseEntity<String> clearProductCache() {
        productService.clearAllProductsCache(); // Delegates to service (with
@CacheEvict(allEntries=true))
        return new ResponseEntity<>("Product cache cleared!", HttpStatus.OK);
    }

    // Removed custom query endpoints from previous activity for focus on
caching.
    // They can be re-added if needed, but caching logic would apply to the
service layer.
}
```

- **Explanation:** The controller now interacts with the ProductService, which in turn handles the caching logic before delegating to the ProductRepository for actual database operations.

Step 6: Run and Test the Application

Now, run your Spring Boot application and observe the caching behavior through API calls. Pay close attention to your console output to see when database calls are made versus when data is served from the cache.

1. **Run the application:**
   - **From IDE:** Right-click on your main application class (e.g., DemoApplication.java) and select "Run 'DemoApplication.main()'".
   - **From Command Line (Maven):** Open your terminal, navigate to the root directory of your project, and run:
     mvn spring-boot:run
   - **Observe Console Output:** Look for messages like "--- ProductService: Fetching product from DATABASE..." (cache miss) and the absence of such

messages (cache hit).

2. **Initial Data Check (Optional):**
    - If your database is empty, create a few products using the POST /api/products endpoint first.
    - Example: POST http://localhost:8080/api/products with {"name": "New Laptop", "price": 1200.00}. Note the ID returned.
3. **Test Caching (@Cacheable):**
    - **First GET request for a product:**
        - **Method:** GET
        - **URL:** http://localhost:8080/api/products/{ID_OF_AN_EXISTING_PRODUCT} (e.g., http://localhost:8080/api/products/1)
        - **Observe:** You should see --- ProductService: Fetching product from DATABASE... in your console, and a noticeable delay (due to Thread.sleep).
        - **Expected Response:** 200 OK with the product data.
    - **Second GET request for the *same* product:**
        - **Method:** GET
        - **URL:** http://localhost:8080/api/products/{SAME_ID_AS_ABOVE} (e.g., http://localhost:8080/api/products/1)
        - **Observe:** The response should be almost immediate. You should **NOT** see --- ProductService: Fetching product from DATABASE... in your console. This indicates a **cache hit**.
    - **Third GET request for a *different* product:**
        - **Method:** GET
        - **URL:** http://localhost:8080/api/products/{ID_OF_A_DIFFERENT_PRODUCT} (e.g., http://localhost:8080/api/products/2)
        - **Observe:** You should see --- ProductService: Fetching product from DATABASE... again, as this product is not yet in the cache.
4. **Test Cache Update (@CachePut):**

- **Update an existing product (e.g., ID 1):**
  - **Method:** PUT
  - **URL:** http://localhost:8080/api/products/1
  - **Headers:** Content-Type: application/json
  - **Body (raw, JSON):** {"name": "Updated Laptop Pro", "price": 1600.00}
  - **Observe:** You should see --- ProductService: Saving/Updating product in DATABASE...
  - **Expected Response:** 200 OK with the updated product.
  - **Now, immediately GET the same product (ID 1):**
    - **Method:** GET
    - **URL:** http://localhost:8080/api/products/1
    - **Observe:** The response should be immediate, and you should **NOT** see a database fetch log. This shows @CachePut successfully updated the cache.

5. **Test Cache Eviction (@CacheEvict):**
   - **Delete a product (e.g., ID 2):**
     - **Method:** DELETE
     - **URL:** http://localhost:8080/api/products/2
     - **Observe:** You should see --- ProductService: Deleting product from DATABASE...
     - **Expected Response:** 204 No Content.
     - **Now, immediately GET the deleted product (ID 2):**
       - **Method:** GET
       - **URL:** http://localhost:8080/api/products/2
       - **Observe:** You should now see --- ProductService: Fetching product from DATABASE... (because it was evicted) and then a 404 Not Found response, as it's no longer in the database.
6. **Test Clear All Cache (@CacheEvict(allEntries=true)):**
   - **Create a few more products** or GET existing ones to populate the cache.
   - **Clear the entire product cache:**
     - **Method:** POST

- **URL:** http://localhost:8080/api/products/clear-cache
- **Observe:** You should see --- ProductService: Clearing ALL entries from 'products' cache ---
- **Expected Response:** 200 OK.
- **Now, GET any product you previously cached (e.g., ID 1 if it still exists in DB):**
  - **Method:** GET
  - **URL:** http://localhost:8080/api/products/1
  - **Observe:** You should now see --- ProductService: Fetching product from DATABASE... again, indicating the cache was completely cleared.

You have successfully implemented caching in your Spring Boot application using Spring's caching abstraction and annotations. You've learned how to:

- Enable caching in your application.
- Mark your entities as Serializable.
- Apply @Cacheable for read-through caching.
- Use @CachePut to update cache entries after data modification.
- Employ @CacheEvict to invalidate cache entries upon deletion or for full cache clearing.
- Observe the performance benefits of caching by reducing direct database access.

This activity demonstrates how caching can significantly boost the performance and responsiveness of your Spring Boot applications.