Chapter 1: Introduction to Spring Boot Data: Data Persistence Made Easy

What is Spring Boot Data?

Spring Boot Data simplifies data source interactions, reducing boilerplate code and providing intuitive abstractions for efficient data storage and retrieval.

Think of Spring Boot Data as a friendly intermediary that stands between your application and your database. Rather than forcing you to write complex data access code, it provides intuitive abstractions that feel natural to Java developers. This means you can focus more on what your application does and less on the mechanics of database interactions.

Spring Boot Data achieves this by eliminating the repetitive "plumbing code" traditionally required for data operations. Instead of writing dozens of lines of SQL or database access code, you can often accomplish the same tasks with just a few lines using Spring Boot Data's conventions and annotations.

Why Use Spring Boot Data? (Benefits for Data-Centric Applications)

When building applications that work with data (which is nearly all of them!), Spring Boot Data offers several compelling advantages:

Spring Boot Data dramatically simplifies your data access layer development. Traditional data access often requires writing verbose code that maps between your Java objects and database structures. Spring Boot Data handles much of this automatically, reducing what might be hundreds of lines of code to just a few interfaces and annotations.

The reduction in boilerplate code is truly remarkable. Operations that traditionally required explicit code—such as creating SQL queries, handling connections, mapping results to objects, and managing transactions—are mostly automated. This leads to significantly cleaner and more maintainable codebases.

Spring Boot Data provides consistent abstractions across different types of data stores. Whether you're working with a relational database like MySQL, a document store like MongoDB, or a key-value store like Redis, the programming model remains largely consistent. This means you can switch between different data technologies without completely rewriting your data access code.

The rapid development capabilities cannot be overstated. With Spring Boot Data, you can implement a complete data access layer in minutes rather than hours or days. Features like automatic query generation based on method names allow you to add new data operations with minimal effort.

Beyond these primary benefits, Spring Boot Data also offers sophisticated features like pagination, sorting, and advanced querying capabilities that would be time-consuming to implement manually. It also integrates seamlessly with other Spring Boot components, creating a cohesive development experience.

Core Concepts of Spring Boot Data: Repositories, Entities, Data Sources - Gentle Introduction

To understand Spring Boot Data, let's explore its three fundamental concepts using everyday analogies:

Entities

An entity in Spring Boot Data represents a "thing" that your application manages—like a customer, a product, or an order. Technically, an entity is a Java class that maps to a database structure (such as a table in a relational database).

Think of an entity as a blueprint for a type of data you want to store. Just as a blueprint for a house specifies all its features (number of rooms, dimensions, etc.), an entity class specifies the structure of your data (what fields it has, their types, etc.).

For example, a `Customer` entity might have fields for `id`, `name`, `email`, and `address`. In your database, this would typically correspond to a table with columns for each of these fields.

```
@Entity
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;
    private String email;
    private String address;

    // Getters and setters
}
```

Repositories

Repositories provide the mechanisms to perform operations on your entities. They act as collection managers for your entities, handling the storage and retrieval details.

A helpful analogy is to think of a repository as a librarian. Just as a librarian helps you find, borrow, return, and organize books without you needing to know the library's organizational system, a repository helps your application perform CRUD operations on entities without needing to know database details.

In Spring Boot Data, you typically define repositories as interfaces that extend one of Spring's base repository interfaces. The fascinating part is that you often don't need to implement these interfaces—Spring creates the implementation automatically at runtime!

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {
    // That's it! This gives you CRUD operations for Customer entities

    // You can add custom query methods like:
    List<Customer> findByEmail(String email);
}
```

Data Sources

A data source represents the connection to your database. It manages details like connection pooling, credentials, and database-specific configuration.

Think of a data source as the dedicated phone line between your application and the database. Just as a phone line needs configuration (the correct number to dial, proper authentication), a data source needs configuration (database URL, username, password, etc.).

In Spring Boot applications, data sources are typically configured in the `application.properties` or `application.yml` file:

```
spring.datasource.url=jdbc:mysql://localhost:3306/myapp
spring.datasource.username=user
spring.datasource.password=password
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

The beauty of Spring Boot is that it often configures reasonable defaults for your data source based on the dependencies in your project. For example, if you include H2 database in your dependencies, Spring Boot will automatically configure an in-memory H2 database unless you specify otherwise.

Together, these three concepts—entities, repositories, and data sources—form the foundation of Spring Boot Data. Entities define what data you store, repositories define how you interact with that data, and data sources define where the data is stored.

Spring Data Modules Overview (High-Level - JPA, MongoDB, Redis, etc.)

Spring Data consists of a family of modules, each specialized for different types of data stores. Understanding the main modules helps you choose the right tools for your specific data needs:

Spring Data JPA

Spring Data JPA is perhaps the most widely used module, designed for working with relational databases through the Java Persistence API (JPA). It supports databases like MySQL, PostgreSQL, Oracle, and H2.

JPA provides an object-relational mapping (ORM) approach, meaning you work with Java objects rather than SQL queries. Spring Data JPA adds an additional layer of convenience on top of JPA, further simplifying database operations.

This module is ideal for applications that need to work with traditional relational databases and require robust transaction support, complex relationships between entities, and structured data.

Spring Data MongoDB

This module provides integration with MongoDB, a popular document database. MongoDB stores data in flexible, JSON-like documents, making it ideal for varying data structures or rapidly evolving schemas.

Spring Data MongoDB allows you to map your Java objects to MongoDB documents, query the database using MongoDB's powerful query language, and leverage MongoDB-specific features like geospatial queries.

This module shines in scenarios involving semi-structured data, where the rigidity of relational databases might be limiting.

Spring Data Redis

Spring Data Redis facilitates integration with Redis, an in-memory data structure store that can function as a database, cache, or message broker.

This module enables you to work with Redis data structures like strings, lists, sets, and hashes through a clean, object-oriented API. It's particularly valuable for caching, session management, real-time analytics, and messaging.

Spring Data Cassandra

For applications requiring massive scalability and high availability, Spring Data Cassandra provides integration with Apache Cassandra, a distributed NoSQL database.

Cassandra excels at handling large volumes of data across multiple data centers, making it suitable for applications with high write throughput requirements or those needing geographic distribution of data.

Spring Data Elasticsearch

This module supports Elasticsearch, a distributed search and analytics engine. It's ideal for applications requiring full-text search capabilities, complex search queries, and real-time analytics.

Other Modules

Spring Data also offers modules for other data stores like Neo4j (graph database), JDBC (direct SQL operations), and more. Each module adapts Spring Data's consistent programming model to the specific characteristics of its target data store.

The beauty of Spring Data's approach is the consistency across these modules. Once you understand how to work with one module, learning others becomes much easier since they share common patterns and abstractions. This consistency allows you to choose the right data store for each part of your application without being constrained by technology-specific knowledge barriers.

Setting up a Basic Spring Boot Data Project

1. **Navigate to Spring Initializr**: start.spring.io
2. **Configure Project**: Select Maven, Java, and Spring Boot version.
3. **Add Dependencies**: Spring Data JPA, H2 Database, Spring Web, Lombok (optional).
4. **Generate and Download**: Extract and import into your IDE.

Step 5: Configure the Database Connection

Spring Boot will automatically configure an in-memory H2 database, but let's make some customizations. Open the `src/main/resources/application.properties` file and add:

```
# H2 Database Configuration
spring.datasource.url=jdbc:h2:mem:testdb
```

```
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password

# Enable H2 Console for browser-based database access
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console

# JPA/Hibernate Configuration
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

These settings will:

- Configure an in-memory H2 database
- Enable the H2 web console for easy database inspection
- Configure Hibernate (JPA implementation) to automatically update the database schema
- Show generated SQL in the logs for learning purposes

Step 6: Create a Basic Entity Class

Create a new Java class under `src/main/java/com/example/datademo` called `Book.java`:

```
package com.example.datademo;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import lombok.Data;  // If you added Lombok

@Entity  // Marks this class as a JPA entity
@Data    // Lombok annotation to generate getters, setters, etc. (remove if not
using Lombok)
public class Book {

    @Id  // Marks this field as the primary key
    @GeneratedValue(strategy = GenerationType.AUTO)  // Automatically generate ID
values
    private Long id;

    private String title;
    private String author;
    private Integer publicationYear;

    // If not using Lombok, add getters and setters here
}
```

Step 7: Create a Repository Interface

Create a new Java interface called `BookRepository.java`:

```
package com.example.datademo;

import org.springframework.data.jpa.repository.JpaRepository;
import java.util.List;

// This interface extends JpaRepository, which provides CRUD operations
// The type parameters are: <Entity type, ID type>
public interface BookRepository extends JpaRepository<Book, Long> {

    // Spring Data JPA will automatically implement these methods based on the
method names
    List<Book> findByAuthor(String author);
    List<Book> findByPublicationYearGreaterThan(Integer year);
}
```

Step 8: Create a Service to Use the Repository

Create a new Java class called `BookService.java`:

```
package com.example.datademo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.util.List;

@Service
public class BookService {

    private final BookRepository bookRepository;

    @Autowired
    public BookService(BookRepository bookRepository) {
        this.bookRepository = bookRepository;
    }

    // Method to save a book
    public Book saveBook(Book book) {
        return bookRepository.save(book);
    }

    // Method to find all books
    public List<Book> findAllBooks() {
        return bookRepository.findAll();
    }

    // Method to find books by author
    public List<Book> findBooksByAuthor(String author) {
        return bookRepository.findByAuthor(author);
```

```
    }

    // Method to find books published after a certain year
    public List<Book> findBooksPublishedAfter(Integer year) {
        return bookRepository.findByPublicationYearGreaterThan(year);
    }
}
```

Step 9: Create a REST Controller to Expose Your Data

Create a new Java class called `BookController.java`:

```
package com.example.datademo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import java.util.List;

@RestController
@RequestMapping("/api/books")
public class BookController {

    private final BookService bookService;

    @Autowired
    public BookController(BookService bookService) {
        this.bookService = bookService;
    }

    @PostMapping
    public Book addBook(@RequestBody Book book) {
        return bookService.saveBook(book);
    }

    @GetMapping
    public List<Book> getAllBooks() {
        return bookService.findAllBooks();
    }

    @GetMapping("/author/{author}")
    public List<Book> getBooksByAuthor(@PathVariable String author) {
        return bookService.findBooksByAuthor(author);
    }

    @GetMapping("/published-after/{year}")
    public List<Book> getBooksPublishedAfter(@PathVariable Integer year) {
        return bookService.findBooksPublishedAfter(year);
    }
}
```

Step 10: Initialize Some Data for Testing

Create a new Java class called `DataInitializer.java`:

```
package com.example.datademo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

@Component
public class DataInitializer implements CommandLineRunner {

    private final BookRepository bookRepository;

    @Autowired
    public DataInitializer(BookRepository bookRepository) {
        this.bookRepository = bookRepository;
    }

    @Override
    public void run(String... args) {
        // Create and save some sample books
        Book book1 = new Book();
        book1.setTitle("The Great Gatsby");
        book1.setAuthor("F. Scott Fitzgerald");
        book1.setPublicationYear(1925);
        bookRepository.save(book1);

        Book book2 = new Book();
        book2.setTitle("To Kill a Mockingbird");
        book2.setAuthor("Harper Lee");
        book2.setPublicationYear(1960);
        bookRepository.save(book2);

        Book book3 = new Book();
        book3.setTitle("1984");
        book3.setAuthor("George Orwell");
        book3.setPublicationYear(1949);
        bookRepository.save(book3);

        System.out.println("Sample data initialized!");
    }
}
```

Step 11: Run Your Application

1. Find the main application class, usually named `DataDemoApplication.java`
2. Right-click on it and select "Run" (the exact option depends on your IDE)
3. The application will start and automatically create the database tables and insert the sample data

Step 12: Test Your Application

Open your web browser and navigate to:

1. http://localhost:8080/h2-console
    - JDBC URL: jdbc:h2:mem:testdb
    - User Name: sa
    - Password: password
    - Click "Connect" to view your database
2. http://localhost:8080/api/books
    - You should see a JSON response with the three books you initialized
3. Try using Postman or curl to test other endpoints:
    - GET http://localhost:8080/api/books/author/George%20Orwell
    - GET http://localhost:8080/api/books/published-after/1950

This provides a solid foundation for learning more advanced Spring Boot Data concepts.

"Hello Data" in Spring Boot - Your First Simple Data Application (CRUD Operations)

Now that we have our application set up, let's walk through the core CRUD (Create, Read, Update, Delete) operations using Spring Boot Data. These operations form the foundation of almost all data-driven applications.

Creating Data (Create)

Creating new records is straightforward with Spring Data repositories. In our application, when a POST request is made to `/api/books`, the `addBook` method in our controller calls the repository's `save` method:

```
@PostMapping
public Book addBook(@RequestBody Book book) {
    return bookService.saveBook(book);
}

// In BookService.java
public Book saveBook(Book book) {
    return bookRepository.save(book);
}
```

To test this, you can use a tool like Postman or curl to send a POST request:

```
POST http://localhost:8080/api/books
Content-Type: application/json

{
```

```
    "title": "The Hobbit",
    "author": "J.R.R. Tolkien",
    "publicationYear": 1937
}
```

The repository's `save` method:

1. Takes your Book object
2. Creates a new row in the database table
3. Assigns an ID to the object (if it's a new entity)
4. Returns the persisted object with its assigned ID

Reading Data (Read)

Reading data is equally simple. Our application demonstrates several ways to read data:

Finding All Records

```
@GetMapping
public List<Book> getAllBooks() {
    return bookService.findAllBooks();
}

// In BookService.java
public List<Book> findAllBooks() {
    return bookRepository.findAll();
}
```

The `findAll()` method returns all records from the corresponding table.

Finding Records by ID

Although not implemented in our example, finding a record by ID is a common operation:

```
// Add this to BookService.java
public Book findBookById(Long id) {
    return bookRepository.findById(id)
        .orElseThrow(() -> new RuntimeException("Book not found"));
}

// Add this to BookController.java
@GetMapping("/{id}")
public Book getBookById(@PathVariable Long id) {
    return bookService.findBookById(id);
}
```

The `findById()` method returns an `Optional<Book>`, which we can use to handle cases where the ID doesn't exist.

Custom Queries

Our application also demonstrates custom queries based on method names:

```
// In BookRepository.java
List<Book> findByAuthor(String author);
List<Book> findByPublicationYearGreaterThan(Integer year);

// In BookController.java
@GetMapping("/author/{author}")
public List<Book> getBooksByAuthor(@PathVariable String author) {
    return bookService.findBooksByAuthor(author);
}
```

Spring Data JPA automatically implements these methods based on their names, creating the appropriate SQL queries behind the scenes.

Updating Data (Update)

Updating existing records uses the same `save` method as creating new ones:

```
// Add this to BookService.java
public Book updateBook(Long id, Book bookDetails) {
    Book book = findBookById(id);  // First find the existing book
    book.setTitle(bookDetails.getTitle());
    book.setAuthor(bookDetails.getAuthor());
    book.setPublicationYear(bookDetails.getPublicationYear());
    return bookRepository.save(book);  // Then save the updated book
}

// Add this to BookController.java
@PutMapping("/{id}")
public Book updateBook(@PathVariable Long id, @RequestBody Book bookDetails) {
    return bookService.updateBook(id, bookDetails);
}
```

When the `save` method receives an entity with an existing ID, it performs an update rather than creating a new record.

Deleting Data (Delete)

Finally, deleting records is handled by the `delete` or `deleteById` methods:

```
// Add this to BookService.java
public void deleteBook(Long id) {
    bookRepository.deleteById(id);
}

// Add this to BookController.java
@DeleteMapping("/{id}")
public void deleteBook(@PathVariable Long id) {
    bookService.deleteBook(id);
}
```

These operations demonstrate the power of Spring Boot Data—complex database operations are reduced to simple method calls, with all the underlying SQL and database interaction handled automatically.

Chapter Summary and Key Takeaways

We explored the core concepts of Spring Boot Data:

- Entities that represent the data structure
- Repositories that provide data access methods
- Data sources that manage database connections

We learned about various Spring Data modules that support different types of databases:

- Spring Data JPA for relational databases
- Spring Data MongoDB for document databases
- Spring Data Redis for key-value stores
- And several others for specialized data storage requirements

Finally, we explored the fundamental CRUD operations:

- Creating new records with the `save` method
- Reading data with `findAll`, `findById`, and custom query methods
- Updating existing records, also with the `save` method
- Deleting records with `delete` or `deleteById`

Chapter 2: Working with Relational Databases and Spring Data JPA - Core Concepts

Introduction to JPA (Java Persistence API) - Object-Relational Mapping for Spring Boot

Java Persistence API (JPA) forms the foundation for database interactions in modern Java applications, especially when working with Spring Boot. At its core, JPA addresses the fundamental challenge in enterprise applications: bridging the gap between object-oriented programming and relational databases.

Think of JPA as a translator between two different worlds. In one world, we have Java objects with inheritance, encapsulation, and polymorphism. In the other, we have relational databases with tables, rows, and foreign

keys. These worlds operate on fundamentally different principles, creating what we call an "impedance mismatch." JPA solves this problem through Object-Relational Mapping (ORM).

When we use JPA, we define special Java classes called "entities" that correspond to database tables. Each instance of an entity represents a row in that table. The JPA specification provides annotations and interfaces that tell the system how to convert between these entities and their database representation.

Spring Data JPA builds upon the standard JPA specification by adding an additional layer of abstraction. This layer dramatically reduces the amount of boilerplate code needed for database operations. Where traditional JPA requires you to write detailed data access objects (DAOs), Spring Data JPA lets you define simple interfaces that automatically implement common database operations.

The JPA `EntityManager` serves as the core interface for persistence operations. It manages the lifecycle of entity instances and provides methods for persisting, retrieving, updating, and removing entities. When using Spring Data JPA, you rarely interact directly with the `EntityManager` since repositories handle that for you, but it's still working behind the scenes.

JPQL (Java Persistence Query Language) provides a way to write database queries using a syntax similar to SQL but operating on entities rather than tables. For example, instead of writing `SELECT * FROM users`, you would write `SELECT u FROM User u` where `User` is the entity class name.

Spring Data JPA Repositories - Defining Data Access Interfaces (In-Depth)

Spring Data JPA repositories represent one of the most powerful abstractions in the Spring ecosystem. They allow you to define data access patterns without implementing them, significantly reducing boilerplate code.

Repository Hierarchy

Spring Data JPA provides a hierarchy of repository interfaces:

1. `Repository<T, ID>` - The base marker interface
2. `CrudRepository<T, ID>` - Provides basic CRUD operations
3. `PagingAndSortingRepository<T, ID>` - Adds pagination and sorting capabilities
4. `JpaRepository<T, ID>` - Extends all previous interfaces and adds JPA-specific methods

Let's look at how you might define a repository for a `Customer` entity:

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface CustomerRepository extends JpaRepository<Customer, Long> {
    // No implementation needed! Spring Data JPA provides it automatically.
}
```

With just this interface definition, you get access to methods like:

- `save(entity)` - Saves an entity
- `findById(id)` - Finds an entity by its ID

- `findAll()` - Returns all entities
- `delete(entity)` - Deletes an entity
- `count()` - Returns the count of entities

The power of Spring Data JPA repositories becomes even more apparent when we consider that we can define custom query methods without writing any implementation code:

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {
    // Finds customers by their last name
    List<Customer> findByLastName(String lastName);

    // Finds customers by email, ignoring case
    Customer findByEmailIgnoreCase(String email);

    // Counts how many active customers exist
    long countByActiveTrue();
}
```

Spring Data JPA will automatically generate the implementation for these methods based on their names. This approach eliminates the need for writing repetitive data access code.

Benefits of Repository Abstraction

The repository pattern offers several key advantages:

1. **Separation of concerns**: It isolates data access logic from business logic
2. **Reduced boilerplate**: No need to write implementation code for common operations
3. **Consistency**: Ensures consistent data access patterns across the application
4. **Testability**: Easier to mock repositories for unit testing
5. **Transaction management**: Spring automatically manages transactions
6. **Query optimization**: Spring can optimize query execution behind the scenes

JPA Entities - Mapping Java Objects to Database Tables (Comprehensive Coverage)

JPA entities are the cornerstone of ORM in Java applications. They represent the bridge between Java objects and database tables.

Basic Entity Mapping

Let's start with a basic entity:

```
import javax.persistence.*;
import java.time.LocalDate;

@Entity
@Table(name = "customers")
public class Customer {
```

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

    @Column(name = "first_name", nullable = false, length = 50)
    private String firstName;

    @Column(name = "last_name", nullable = false, length = 50)
    private String lastName;

    @Column(unique = true)
    private String email;

    @Column(name = "date_of_birth")
    private LocalDate dateOfBirth;

    @Column(columnDefinition = "boolean default true")
    private boolean active;

    // Constructors, getters, and setters omitted for brevity
}
```

Let's break down the annotations:

- `@Entity`: Marks the class as a JPA entity, making it persistable
- `@Table`: Specifies the database table name (optional, defaults to class name)
- `@Id`: Marks the field as the primary key
- `@GeneratedValue`: Specifies how the primary key is generated
- `@Column`: Customizes the column mapping with options like name, nullable, unique, length

Relationship Mappings

Relationships between entities are a critical aspect of database design. JPA provides several annotations to define these relationships:

One-to-One Relationship

```
@Entity
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // Other fields...

    @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    @JoinColumn(name = "address_id", referencedColumnName = "id")
    private Address address;
}
```

```
@Entity
public class Address {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String street;
    private String city;
    private String state;
    private String zipCode;

    // Bidirectional relationship (optional)
    @OneToOne(mappedBy = "address")
    private Customer customer;
}
```

In this example, the `Customer` entity has a one-to-one relationship with the `Address` entity. The `@JoinColumn` annotation specifies the foreign key column in the `customers` table.

One-to-Many and Many-to-One Relationships

```
@Entity
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // Other fields...

    @OneToMany(mappedBy = "customer", cascade = CascadeType.ALL, orphanRemoval =
true)
    private List<Order> orders = new ArrayList<>();

    // Helper method to maintain both sides of the relationship
    public void addOrder(Order order) {
        orders.add(order);
        order.setCustomer(this);
    }

    public void removeOrder(Order order) {
        orders.remove(order);
        order.setCustomer(null);
    }
}

@Entity
@Table(name = "orders")
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
```

```
    private LocalDate orderDate;
    private String status;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "customer_id")
    private Customer customer;
}
```

Here, a `Customer` can have many `Order` entities, represented by the `@OneToMany` annotation. Conversely, each `Order` belongs to one `Customer`, represented by the `@ManyToOne` annotation.

Many-to-Many Relationships

```
@Entity
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private BigDecimal price;

    @ManyToMany(mappedBy = "products")
    private Set<Order> orders = new HashSet<>();
}

@Entity
@Table(name = "orders")
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // Other fields...

    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    @JoinTable(
        name = "order_product",
        joinColumns = @JoinColumn(name = "order_id"),
        inverseJoinColumns = @JoinColumn(name = "product_id")
    )
    private Set<Product> products = new HashSet<>();

    // Helper methods
    public void addProduct(Product product) {
        products.add(product);
        product.getOrders().add(this);
    }

    public void removeProduct(Product product) {
```

```
        products.remove(product);
        product.getOrders().remove(this);
    }
}
```

In a many-to-many relationship, a join table is required. The `@JoinTable` annotation specifies the join table name and its foreign key columns.

Embedded Objects

Sometimes, you want to group related fields but don't need them as separate entities. JPA provides `@Embedded` and `@Embeddable` for this purpose:

```
@Entity
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // Other fields...

    @Embedded
    private ContactInfo contactInfo;
}

@Embeddable
public class ContactInfo {
    private String email;
    private String phone;

    @AttributeOverride(name = "streetName", column = @Column(name =
"home_street"))
    @AttributeOverride(name = "city", column = @Column(name = "home_city"))
    @Embedded
    private Address homeAddress;

    @AttributeOverride(name = "streetName", column = @Column(name =
"work_street"))
    @AttributeOverride(name = "city", column = @Column(name = "work_city"))
    @Embedded
    private Address workAddress;
}

@Embeddable
public class Address {
    private String streetName;
    private String city;
    private String state;
    private String zipCode;
}
```

Here, `ContactInfo` and `Address` are embeddable objects. They don't have their own tables but are stored as columns in the `customers` table. The `@AttributeOverride` annotation allows you to customize column names when embedding the same type multiple times.

Understanding Cascade Types and Fetch Types

Cascade types define how operations should propagate from a parent entity to its associated entities:

- `CascadeType.PERSIST`: When persisting a parent, persist its children
- `CascadeType.MERGE`: When merging a parent, merge its children
- `CascadeType.REMOVE`: When removing a parent, remove its children
- `CascadeType.REFRESH`: When refreshing a parent, refresh its children
- `CascadeType.DETACH`: When detaching a parent, detach its children
- `CascadeType.ALL`: Apply all cascade types

Fetch types determine when associated entities are loaded:

- `FetchType.EAGER`: Load associated entities immediately when the parent is loaded
- `FetchType.LAZY`: Load associated entities only when they are accessed

The choice between eager and lazy loading significantly impacts performance. As a rule of thumb, use lazy loading for most relationships to avoid loading unnecessary data, but be aware of the potential for LazyInitializationException when accessing lazy associations outside a transaction.

For example:

```
// Eager loading (loads all orders with the customer)
@OneToMany(mappedBy = "customer", fetch = FetchType.EAGER)
private List<Order> orders;

// Lazy loading (loads orders only when accessed)
@OneToMany(mappedBy = "customer", fetch = FetchType.LAZY)
private List<Order> orders;
```

Orphan removal is another important concept. When set to `true`, it removes child entities when they are no longer referenced by their parent:

```
@OneToMany(mappedBy = "customer", orphanRemoval = true)
private List<Order> orders;
```

With this setting, if you remove an order from the customer's orders collection and save the customer, the order will be deleted from the database.

Data Sources and Database Configuration in Spring Boot

Spring Boot simplifies database configuration through auto-configuration and sensible defaults. Let's explore how to configure data sources in a Spring Boot application.

Basic Configuration with application.properties

For most applications, configuring a data source is as simple as adding a few properties:

```
# Database Connection
spring.datasource.url=jdbc:mysql://localhost:3306/myapp
spring.datasource.username=dbuser
spring.datasource.password=dbpass
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# Hibernate Configuration
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
```

Or using YAML format:

```
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/myapp
    username: dbuser
    password: dbpass
    driver-class-name: com.mysql.cj.jdbc.Driver
  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true
    properties:
      hibernate:
        format_sql: true
        dialect: org.hibernate.dialect.MySQL8Dialect
```

Let's break down these properties:

- `spring.datasource.url`: The JDBC URL for connecting to the database
- `spring.datasource.username` and `spring.datasource.password`: Database credentials
- `spring.datasource.driver-class-name`: The JDBC driver class (often optional in Spring Boot)
- `spring.jpa.hibernate.ddl-auto`: Controls how Hibernate manages the database schema (options: none, validate, update, create, create-drop)
- `spring.jpa.show-sql`: Displays SQL queries in the logs
- `spring.jpa.properties.hibernate.format_sql`: Formats SQL for better readability
- `spring.jpa.properties.hibernate.dialect`: Specifies the Hibernate dialect for your database

Embedded Databases for Development and Testing

Spring Boot provides excellent support for embedded databases like H2, HSQLDB, and Derby. These are useful for development and testing:

```
# H2 Database
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

When using an embedded database like H2, Spring Boot automatically initializes it with schema.sql and data.sql files if they are present in the classpath.

Connection Pooling

Connection pooling is crucial for performance in production applications. Spring Boot configures HikariCP by default, which is a high-performance JDBC connection pool:

```
# HikariCP Connection Pool Configuration
spring.datasource.hikari.maximum-pool-size=10
spring.datasource.hikari.minimum-idle=5
spring.datasource.hikari.idle-timeout=300000
spring.datasource.hikari.connection-timeout=20000
spring.datasource.hikari.max-lifetime=1200000
```

These settings control the behavior of the connection pool:

- `maximum-pool-size`: Maximum number of connections in the pool
- `minimum-idle`: Minimum number of idle connections
- `idle-timeout`: Time in milliseconds after which an idle connection is closed
- `connection-timeout`: Maximum time to wait for a connection from the pool
- `max-lifetime`: Maximum lifetime of a connection in the pool

Multiple Data Sources

For applications requiring multiple data sources, you'll need to configure them manually:

```
@Configuration
public class DataSourceConfig {

    @Primary
    @Bean(name = "primaryDataSource")
    @ConfigurationProperties("app.datasource.primary")
    public DataSource primaryDataSource() {
        return DataSourceBuilder.create().build();
    }

    @Bean(name = "secondaryDataSource")
    @ConfigurationProperties("app.datasource.secondary")
```

```
    public DataSource secondaryDataSource() {
        return DataSourceBuilder.create().build();
    }
}
```

With corresponding properties:

```
# Primary DataSource
app.datasource.primary.url=jdbc:mysql://localhost:3306/primary_db
app.datasource.primary.username=dbuser1
app.datasource.primary.password=dbpass1

# Secondary DataSource
app.datasource.secondary.url=jdbc:postgresql://localhost:5432/secondary_db
app.datasource.secondary.username=dbuser2
app.datasource.secondary.password=dbpass2
```

Working with multiple data sources requires additional configuration for entity managers and transaction managers, which goes beyond the scope of this chapter but is an important consideration for complex applications.

Basic CRUD Operations with Spring Data JPA Repositories - Save, Find, Delete, etc. (Practical Examples)

Now let's examine how to perform basic CRUD (Create, Read, Update, Delete) operations using Spring Data JPA repositories.

Creating and Updating Entities

The `save()` method is used for both creating new entities and updating existing ones:

```
@Service
@Transactional
public class CustomerService {

    private final CustomerRepository customerRepository;

    public CustomerService(CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
    }

    // Create a new customer
    public Customer createCustomer(Customer customer) {
        // Business validation logic
        if (customer.getEmail() == null || customer.getEmail().isEmpty()) {
            throw new IllegalArgumentException("Email is required");
        }

        // Save the customer
```

```
        return customerRepository.save(customer);
    }

    // Update an existing customer
    public Customer updateCustomer(Long id, Customer updatedCustomer) {
        // Find the existing customer
        Customer existingCustomer = customerRepository.findById(id)
            .orElseThrow(() -> new EntityNotFoundException("Customer not found
with ID: " + id));

        // Update the customer properties
        existingCustomer.setFirstName(updatedCustomer.getFirstName());
        existingCustomer.setLastName(updatedCustomer.getLastName());
        existingCustomer.setEmail(updatedCustomer.getEmail());

        // Save the updated customer
        return customerRepository.save(existingCustomer);
    }
}
```

A few important points to note:

1. The `save()` method returns the saved entity, which may be useful for getting generated IDs.
2. For updates, it's recommended to first retrieve the entity, modify it, and then save it. This ensures that only the specified fields are updated.
3. The `@Transactional` annotation ensures that operations are performed within a transaction.

Reading Entities

Spring Data JPA provides several methods for retrieving entities:

```
@Service
public class CustomerService {

    private final CustomerRepository customerRepository;

    public CustomerService(CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
    }

    // Find customer by ID
    public Customer getCustomerById(Long id) {
        return customerRepository.findById(id)
            .orElseThrow(() -> new EntityNotFoundException("Customer not found
with ID: " + id));
    }

    // Find all customers
    public List<Customer> getAllCustomers() {
        return customerRepository.findAll();
    }
```

```
    // Find customers with pagination
    public Page<Customer> getCustomersPage(int page, int size) {
        Pageable pageable = PageRequest.of(page, size,
Sort.by("lastName").ascending());
        return customerRepository.findAll(pageable);
    }

    // Check if customer exists
    public boolean customerExists(Long id) {
        return customerRepository.existsById(id);
    }

    // Count customers
    public long countCustomers() {
        return customerRepository.count();
    }
}
```

Key observations:

1. The `findById()` method returns an `Optional<T>`, allowing for null-safe handling of the result.
2. For paginated results, use the `findAll(Pageable)` method with a `PageRequest`.
3. The `existsById()` method efficiently checks for existence without loading the entity.
4. The `count()` method returns the total count of entities.

Deleting Entities

Spring Data JPA provides multiple ways to delete entities:

```
@Service
@Transactional
public class CustomerService {

    private final CustomerRepository customerRepository;

    public CustomerService(CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
    }

    // Delete by ID
    public void deleteCustomer(Long id) {
        if (!customerRepository.existsById(id)) {
            throw new EntityNotFoundException("Customer not found with ID: " +
id);
        }
        customerRepository.deleteById(id);
    }

    // Delete entity
    public void deleteCustomer(Customer customer) {
```

```
        customerRepository.delete(customer);
    }

    // Delete multiple customers
    public void deleteCustomers(List<Customer> customers) {
        customerRepository.deleteAll(customers);
    }

    // Delete all customers
    public void deleteAllCustomers() {
        customerRepository.deleteAll();
    }
}
```

Important considerations:

1. It's good practice to check if the entity exists before deleting it by ID.
2. For bulk deletions, `deleteAll(Iterable)` is more efficient than calling `delete()` for each entity.
3. Be careful with `deleteAll()` as it removes all entities from the repository.

Bulk Operations

Spring Data JPA supports bulk operations for improved performance when dealing with multiple entities:

```
@Service
@Transactional
public class CustomerService {

    private final CustomerRepository customerRepository;

    public CustomerService(CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
    }

    // Save multiple customers
    public List<Customer> saveAllCustomers(List<Customer> customers) {
        return customerRepository.saveAll(customers);
    }

    // Find multiple customers by IDs
    public List<Customer> getCustomersByIds(List<Long> ids) {
        return customerRepository.findAllById(ids);
    }
}
```

These bulk operations are more efficient than performing individual operations, especially for large datasets.

Query Methods in Spring Data JPA - Automatic Query Generation from Method Names (Detailed Explanation)

One of Spring Data JPA's most powerful features is its ability to generate queries automatically based on method names. This feature allows you to define query methods without writing any query code.

Query Method Naming Conventions

The structure of a query method follows this pattern:

```
findBy[Property][Condition][AndOr][Property][Condition]...
```

For example:

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {
    // Find by a single property
    List<Customer> findByLastName(String lastName);

    // Find by multiple properties with AND
    List<Customer> findByFirstNameAndLastName(String firstName, String lastName);

    // Find by multiple properties with OR
    List<Customer> findByFirstNameOrLastName(String firstName, String lastName);
}
```

Supported Keywords for Conditions

Spring Data JPA supports a wide range of keywords for defining conditions:

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {
    // Equality conditions
    List<Customer> findByActive(boolean active);

    // Negation
    List<Customer> findByActiveNot(boolean active);

    // Case insensitive comparison
    List<Customer> findByEmailIgnoreCase(String email);

    // Pattern matching
    List<Customer> findByLastNameLike(String pattern);
    List<Customer> findByEmailStartingWith(String prefix);
    List<Customer> findByFirstNameEndingWith(String suffix);
    List<Customer> findByLastNameContaining(String text);

    // Comparison operators
    List<Customer> findByAgeLessThan(int age);
    List<Customer> findByAgeGreaterThan(int age);
    List<Customer> findByAgeLessThanEqual(int age);
    List<Customer> findByAgeGreaterThanEqual(int age);
```

```
    // Range queries
    List<Customer> findByAgeBetween(int start, int end);

    // NULL checks
    List<Customer> findByEmailIsNull();
    List<Customer> findByEmailIsNotNull();

    // Collection queries
    List<Customer> findByRolesIn(Collection<String> roles);
    List<Customer> findByRolesNotIn(Collection<String> roles);
}
```

Sorting and Limiting Results

You can specify sorting or result limits directly in the method name:

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {
    // Ordering results
    List<Customer> findByLastNameOrderByFirstNameAsc(String lastName);
    List<Customer> findByActiveOrderByLastNameDescFirstNameAsc(boolean active);

    // Limiting results
    Customer findFirstByOrderByLastNameAsc();
    Customer findTopByOrderByAgeDesc();
    List<Customer> findTop5ByActive(boolean active);
    List<Customer> findFirst10ByLastName(String lastName);
}
```

Alternative Query Method Prefixes

Besides `findBy`, Spring Data JPA supports other prefixes for different operations:

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {
    // Count results
    long countByActive(boolean active);

    // Check existence
    boolean existsByEmail(String email);

    // Delete operations
    void deleteByLastName(String lastName);

    // Remove operations (same as delete)
    long removeByActive(boolean active);
}
```

Working with Relationships

You can navigate through relationships in query methods:

```
public interface OrderRepository extends JpaRepository<Order, Long> {
    // Find orders by customer property
    List<Order> findByCustomerId(Long customerId);
    List<Order> findByCustomerLastName(String lastName);

    // More complex nested properties
    List<Order> findByCustomerAddressCity(String city);

    // Find by collection properties
    List<Order> findByProductsName(String productName);
    List<Order> findByProductsNameContaining(String productNamePart);
}
```

Date and Time Queries

Working with dates requires special attention:

```
public interface OrderRepository extends JpaRepository<Order, Long> {
    List<Order> findByOrderDateAfter(LocalDate date);
    List<Order> findByOrderDateBefore(LocalDate date);
    List<Order> findByOrderDateBetween(LocalDate startDate, LocalDate endDate);
}
```

Using @Query for Complex Queries

When query methods become too complex, you can use the `@Query` annotation:

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {
    // JPQL query
    @Query("SELECT c FROM Customer c WHERE c.lastName LIKE %:name%")
    List<Customer> findCustomersWithLastNameLike(@Param("name") String name);

    // Native SQL query
    @Query(value = "SELECT * FROM customers WHERE active = ?1", nativeQuery =
true)
    List<Customer> findActiveCustomersNative(boolean active);

    // Update query
    @Modifying
    @Query("UPDATE Customer c SET c.active = :active WHERE c.lastLogin < :date")
    int updateActivityStatus(@Param("active") boolean active, @Param("date")
LocalDate date);
}
```

The `@Query` annotation allows you to write both JPQL and native SQL queries. Use `@Param` for named parameters or positional parameters (e.g., `?1`). For update or delete operations, add the `@Modifying` annotation.

Practical Examples and Code Demonstrations

Let's look at a complete example of a simple blogging application with users, posts, and comments:

Entity Definitions

```
@Entity
@Table(name = "users")
public class User {

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

    @Column(nullable = false, length = 50)
    private String name;

    @Column(nullable = false, unique = true)
    private String email;

    @Column(nullable = false)
    private String password;

    @OneToMany(mappedBy = "author", cascade = CascadeType.ALL, orphanRemoval =
true)
    private List<Post> posts = new ArrayList<>();

    @OneToMany(mappedBy = "author", cascade = CascadeType.ALL, orphanRemoval =
true)
    private List<Comment> comments = new ArrayList<>();

    // Constructors, getters, and setters
}

@Entity
@Table(name = "posts")
public class Post {

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

    @Column(nullable = false, length = 100)
    private String title;

    @Column(nullable = false, columnDefinition = "TEXT")
    private String content;
```

```
    @Column(name = "created_at", nullable = false)
    private LocalDateTime createdAt;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "author_id", nullable = false)
    private User author;

    @OneToMany(mappedBy = "post", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Comment> comments = new ArrayList<>();

    @ManyToMany
    @JoinTable(
        name = "post_tags",
        joinColumns = @JoinColumn(name = "post_id"),
        inverseJoinColumns = @JoinColumn(name = "tag_id")
    )
    private Set<Tag> tags = new HashSet<>();

    // Constructors, getters, and setters
}

@Entity
@Table(name = "comments")
public class Comment {

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

    @Column(nullable = false, columnDefinition = "TEXT")
    private String content;

    @Column(name = "created_at", nullable = false)
    private LocalDateTime createdAt;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "post_id", nullable = false)
    private Post post;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "author_id", nullable = false)
    private User author;

    // Constructors, getters, and setters
}

@Entity
@Table(name = "tags")
public class Tag {

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

```
    @Column(nullable = false, unique = true, length = 30)
    private String name;

    @ManyToMany(mappedBy = "tags")
    private Set<Post> posts = new HashSet<>();

    // Constructors, getters, and setters
}
```

Repository Definitions

```
public interface UserRepository extends JpaRepository<User, Long> {
    Optional<User> findByEmail(String email);
    boolean existsByEmail(String email);
}

public interface PostRepository extends JpaRepository<Post, Long> {
    List<Post> findByAuthorId(Long authorId);
    List<Post> findByAuthorEmail(String email);
    List<Post> findByTitleContainingOrContentContaining(String title, String
content);
    List<Post> findByCreatedAtAfter(LocalDateTime date);
    List<Post> findByTagsName(String tagName);

    @Query("SELECT p FROM Post p WHERE p.createdAt > :date ORDER BY p.createdAt
DESC")
    List<Post> findRecentPosts(@Param("date") LocalDateTime date);
}

public interface CommentRepository extends JpaRepository<Comment, Long> {
    List<Comment> findByPostId(Long postId);
    List<Comment> findByAuthorId(Long authorId);
    List<Comment> findByPostIdOrderByCreatedAtDesc(Long postId);

    @Query("SELECT c FROM Comment c WHERE c.post.id = :postId AND c.createdAt >
:date")
    List<Comment> findRecentCommentsByPostId(@Param("postId") Long postId,
@Param("date") LocalDateTime date);
}

public interface TagRepository extends JpaRepository<Tag, Long> {
    Optional<Tag> findByName(String name);
    List<Tag> findByNameIn(Collection<String> names);
}
```

Service Layer Implementation

Now let's implement a service layer that uses these repositories:

```
@Service
@Transactional
public class BlogService {

    private final UserRepository userRepository;
    private final PostRepository postRepository;
    private final CommentRepository commentRepository;
    private final TagRepository tagRepository;

    // Constructor injection
    public BlogService(UserRepository userRepository,
                       PostRepository postRepository,
                       CommentRepository commentRepository,
                       TagRepository tagRepository) {
        this.userRepository = userRepository;
        this.postRepository = postRepository;
        this.commentRepository = commentRepository;
        this.tagRepository = tagRepository;
    }

    // User operations
    public User registerUser(User user) {
        if (userRepository.existsByEmail(user.getEmail())) {
            throw new IllegalStateException("Email already in use: " +
user.getEmail());
        }
        return userRepository.save(user);
    }

    public User getUserById(Long userId) {
        return userRepository.findById(userId)
            .orElseThrow(() -> new EntityNotFoundException("User not found with
ID: " + userId));
    }

    // Post operations
    public Post createPost(Long authorId, Post post, Set<String> tagNames) {
        User author = getUserById(authorId);
        post.setAuthor(author);
        post.setCreatedAt(LocalDateTime.now());

        // Process tags
        if (tagNames != null && !tagNames.isEmpty()) {
            Set<Tag> tags = new HashSet<>();

            for (String tagName : tagNames) {
                Tag tag = tagRepository.findByName(tagName)
                    .orElseGet(() -> {
                        Tag newTag = new Tag();
                        newTag.setName(tagName);
                        return tagRepository.save(newTag);
                    });
                tags.add(tag);
```

```
            }

            post.setTags(tags);
        }

        return postRepository.save(post);
    }

    public Post getPostById(Long postId) {
        return postRepository.findById(postId)
            .orElseThrow(() -> new EntityNotFoundException("Post not found with
ID: " + postId));
    }

    public List<Post> getPostsByAuthor(Long authorId) {
        return postRepository.findByAuthorId(authorId);
    }

    public List<Post> searchPosts(String searchTerm) {
        return postRepository.findByTitleContainingOrContentContaining(searchTerm,
searchTerm);
    }

    // Comment operations
    public Comment addComment(Long postId, Long authorId, String content) {
        Post post = getPostById(postId);
        User author = getUserById(authorId);

        Comment comment = new Comment();
        comment.setContent(content);
        comment.setCreatedAt(LocalDateTime.now());
        comment.setPost(post);
        comment.setAuthor(author);

        return commentRepository.save(comment);
    }

    public List<Comment> getCommentsByPost(Long postId) {
        return commentRepository.findByPostIdOrderByCreatedAtDesc(postId);
    }

    // Additional utility methods
    public List<Post> getRecentPosts(int days) {
        LocalDateTime date = LocalDateTime.now().minusDays(days);
        return postRepository.findRecentPosts(date);
    }

    public List<Tag> getPopularTags() {
        // This would typically involve more complex logic or a custom query
        return tagRepository.findAll();
    }
}
```

Database Configuration

Let's include a sample configuration for development and production environments:

```
# application.yml
spring:
  profiles:
    active: dev

---
# Development profile
spring:
  config:
    activate:
      on-profile: dev
  datasource:
    url: jdbc:h2:mem:blogdb
    username: sa
    password: password
    driver-class-name: org.h2.Driver
  h2:
    console:
      enabled: true
      path: /h2-console
  jpa:
    hibernate:
      ddl-auto: create-drop
    show-sql: true
    properties:
      hibernate:
        format_sql: true

---
# Production profile
spring:
  config:
    activate:
      on-profile: prod
  datasource:
    url: jdbc:mysql://localhost:3306/blogdb
    username: ${DB_USERNAME}
    password: ${DB_PASSWORD}
    driver-class-name: com.mysql.cj.jdbc.Driver
    hikari:
      maximum-pool-size: 10
      minimum-idle: 5
      idle-timeout: 300000
      connection-timeout: 20000
  jpa:
    hibernate:
      ddl-auto: validate
    properties:
```

```
    hibernate:
      dialect: org.hibernate.dialect.MySQL8Dialect
  show-sql: false
flyway:
  enabled: true
  locations: classpath:db/migration
```

In this configuration:

- The development profile uses an H2 in-memory database with automatic schema creation.
- The production profile uses MySQL with validated schemas managed by Flyway migrations.
- Sensitive information like database credentials in production are externalized as environment variables.

Chapter Summary and Key Takeaways

In this chapter, we've explored the core concepts of working with relational databases using Spring Data JPA. Here are the key takeaways:

1. **JPA and Object-Relational Mapping**
    - JPA provides a standard for ORM in Java applications
    - It bridges the gap between object-oriented code and relational databases
    - Spring Data JPA builds upon JPA to simplify data access
2. **Spring Data JPA Repositories**
    - Repositories provide a powerful abstraction for data access
    - They eliminate boilerplate code through interface-based programming
    - Different repository types offer various levels of functionality (CrudRepository, JpaRepository)
3. **JPA Entities and Relationships**
    - Entities map Java objects to database tables
    - Relationships between entities (One-to-One, One-to-Many, Many-to-Many) model database relationships
    - Cascade types and fetch types control how operations propagate and when data is loaded
    - Embedded objects allow for grouping related fields without separate tables
4. **Data Source Configuration**
    - Spring Boot simplifies database configuration with sensible defaults
    - Connection pooling is essential for performance in production applications
    - Multiple data sources can be configured for complex applications
5. **CRUD Operations**
    - Spring Data JPA provides methods for creating, reading, updating, and deleting entities
    - Transactions ensure data consistency
    - Bulk operations improve performance for multiple entities
6. **Query Methods**

- Method names automatically generate queries without writing SQL or JPQL
- A rich set of keywords supports various query conditions
- Complex queries can be defined using the `@Query` annotation

Chapter 3: Advanced Spring Data JPA - Custom Queries, Projections, and Specifications

Introduction

As applications grow in complexity, the basic query methods provided by Spring Data JPA repositories might not be sufficient. In this chapter, we'll explore advanced features that enable you to craft sophisticated data access patterns, optimize performance, and maintain clean, maintainable code even as your data requirements become more complex.

These advanced features build upon the foundation we established in previous chapters, taking your Spring Data JPA skills to the next level. By mastering these techniques, you'll be equipped to handle real-world data access challenges with elegance and efficiency.

@Query Annotation - Writing Custom JPQL and Native SQL Queries

While derived query methods offer convenience, they can become unwieldy for complex queries. The `@Query` annotation provides a powerful alternative by allowing you to write custom queries directly in your repository interfaces.

JPQL Queries with @Query

JPQL (Java Persistence Query Language) is an object-oriented query language that operates on entity objects rather than database tables. Here's how to use it with the `@Query` annotation:

```
public interface EmployeeRepository extends JpaRepository<Employee, Long> {

    @Query("SELECT e FROM Employee e WHERE e.department.name = :departmentName AND
e.salary > :minSalary")
    List<Employee> findEmployeesByDepartmentAndSalaryGreaterThan(
            @Param("departmentName") String department,
            @Param("minSalary") Double salary);

    @Query("SELECT e FROM Employee e JOIN e.projects p WHERE p.name =
:projectName")
    List<Employee> findEmployeesAssignedToProject(@Param("projectName") String
projectName);
}
```

In the examples above:

- The first query retrieves employees by department name and minimum salary
- The second query demonstrates a join operation to find employees assigned to a specific project

Native SQL Queries

When JPQL isn't sufficient, you can use native SQL queries by setting the `nativeQuery` attribute to `true`:

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

    @Query(value = "SELECT * FROM products p WHERE p.price BETWEEN :minPrice AND
:maxPrice ORDER BY p.price",
           nativeQuery = true)
    List<Product> findProductsInPriceRange(
            @Param("minPrice") Double minPrice,
            @Param("maxPrice") Double maxPrice);

    @Query(value = "SELECT p.* FROM products p " +
                   "JOIN product_categories pc ON p.id = pc.product_id " +
                   "JOIN categories c ON pc.category_id = c.id " +
                   "WHERE c.name = :category AND p.stock_quantity > 0",
           nativeQuery = true)
    List<Product> findInStockProductsByCategory(@Param("category") String
category);
}
```

Native SQL queries are particularly useful when:

- You need database-specific features not supported by JPQL
- You're working with complex SQL constructs like window functions or CTEs
- You need to optimize performance with database-specific query hints

Named vs. Indexed Parameters

Spring Data JPA supports both named and indexed parameters in `@Query` annotations:

```
// Named parameters (recommended for readability and maintainability)
@Query("SELECT u FROM User u WHERE u.email = :email")
User findByEmailNamedParam(@Param("email") String email);

// Indexed parameters (position-based)
@Query("SELECT u FROM User u WHERE u.email = ?1")
User findByEmailIndexedParam(String email);
```

Named parameters are generally preferred as they:

- Improve readability
- Allow parameter reuse within the same query
- Are less prone to errors when modifying queries

@Modifying for Data Modification Operations

The `@Modifying` annotation is required for queries that modify the database state:

```
@Modifying
@Query("UPDATE Employee e SET e.salary = e.salary * (1 + :percentage/100) WHERE
e.department.id = :departmentId")
int giveRaiseToDepartment(@Param("percentage") double percentage,
@Param("departmentId") Long departmentId);

@Modifying
@Query("DELETE FROM Product p WHERE p.lastOrderDate < :date")
int removeDiscontinuedProducts(@Param("date") LocalDate date);
```

Important considerations when using `@Modifying`:

- The method should return `void`, `int`, or `Integer` (the latter two will return the number of affected rows)
- These operations typically need to be wrapped in a transaction (using `@Transactional`)
- By default, persistence contexts are not automatically cleared after a modifying query; use `clearAutomatically = true` if needed

```
@Modifying(clearAutomatically = true)
@Query("UPDATE User u SET u.failedLoginAttempts = 0 WHERE u.id = :userId")
void resetFailedLoginAttempts(@Param("userId") Long userId);
```

Projections in Spring Data JPA - Selecting Specific Attributes

Projections allow you to retrieve only the data you need instead of fetching entire entity objects, which can significantly improve performance for large entities or when you only need a subset of fields.

Interface-based Projections

Closed Projections

Closed projections explicitly define the properties to retrieve:

```
public interface EmployeeRepository extends JpaRepository<Employee, Long> {
    List<EmployeeNameView> findByDepartmentName(String departmentName);
}

// Closed projection interface
public interface EmployeeNameView {
    String getFirstName();
    String getLastName();
    // You can even add derived properties
    default String getFullName() {
        return getFirstName() + " " + getLastName();
    }
}
```

The repository method will automatically return objects containing only the requested properties, reducing network overhead and memory usage.

Open Projections

Open projections use SpEL (Spring Expression Language) expressions:

```
public interface EmployeeNameView {
    @Value("#{target.firstName + ' ' + target.lastName}")
    String getFullName();

    @Value("#{target.department.name}")
    String getDepartmentName();
}
```

Open projections are more flexible but less efficient than closed projections because they require loading the entire entity to evaluate the expressions.

Class-based (DTO) Projections

Class-based projections use Data Transfer Objects (DTOs) to hold specific entity attributes:

```
public interface EmployeeRepository extends JpaRepository<Employee, Long> {
    @Query("SELECT new com.example.dto.EmployeeProjection(e.id, e.firstName,
e.lastName, e.department.name) " +
           "FROM Employee e WHERE e.department.id = :departmentId")
    List<EmployeeProjection>
findEmployeeProjectionsByDepartmentId(@Param("departmentId") Long departmentId);
}

// DTO class
public class EmployeeProjection {
    private Long id;
    private String firstName;
    private String lastName;
    private String departmentName;

    // Constructor, getters, setters
    public EmployeeProjection(Long id, String firstName, String lastName, String
departmentName) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
        this.departmentName = departmentName;
    }

    // Getters and setters...
}
```

Class-based projections offer greater control over the structure of returned data and are useful for complex mappings or when you need to transform data during retrieval.

Dynamic Projections

Dynamic projections allow you to specify the return type at runtime:

```
public interface UserRepository extends JpaRepository<User, Long> {
    <T> List<T> findByActiveTrue(Class<T> type);
}
```

With this approach, you can use the same repository method to retrieve different projections:

```
// Get full entities
List<User> fullUsers = userRepository.findByActiveTrue(User.class);

// Get just usernames
List<UsernameOnly> usernames =
userRepository.findByActiveTrue(UsernameOnly.class);

// Get detailed view
List<UserDetailView> detailedUsers =
userRepository.findByActiveTrue(UserDetailView.class);
```

This flexibility is powerful for creating reusable repository methods that can adapt to different use cases.

Specifications in Spring Data JPA - Dynamic and Complex Queries

Specifications provide a type-safe, programmatic way to build complex queries using the JPA Criteria API. They're particularly useful for dynamic queries where conditions are determined at runtime.

Setting Up Specifications

To use specifications, your repository needs to extend `JpaSpecificationExecutor`:

```
public interface ProductRepository extends
    JpaRepository<Product, Long>,
    JpaSpecificationExecutor<Product> {
    // Regular repository methods
}
```

Creating Simple Specifications

Here's how to create individual specifications:

```
public class ProductSpecifications {

    public static Specification<Product> hasCategory(String category) {
        return (root, query, criteriaBuilder) -> {
            if (category == null) {
                return criteriaBuilder.conjunction();  // Always true predicate if
no category
            }
            return criteriaBuilder.equal(root.get("category"), category);
        };
    }

    public static Specification<Product> priceBetween(Double min, Double max) {
        return (root, query, criteriaBuilder) -> {
            if (min == null && max == null) {
                return criteriaBuilder.conjunction();
            }

            if (min == null) {
                return criteriaBuilder.lessThanOrEqualTo(root.get("price"), max);
            }

            if (max == null) {
                return criteriaBuilder.greaterThanOrEqualTo(root.get("price"),
min);
            }

            return criteriaBuilder.between(root.get("price"), min, max);
        };
    }

    public static Specification<Product> inStock() {
        return (root, query, criteriaBuilder) ->
            criteriaBuilder.greaterThan(root.get("stockQuantity"), 0);
    }
}
```

Combining Specifications

The power of specifications comes from combining them:

```
// Create a service that uses specifications
@Service
public class ProductService {

    private final ProductRepository productRepository;

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

```
    public List<Product> findProducts(String category, Double minPrice, Double
maxPrice, Boolean inStockOnly) {
        Specification<Product> spec = Specification.where(null);  // Start with an
empty specification

        if (category != null) {
            spec = spec.and(ProductSpecifications.hasCategory(category));
        }

        if (minPrice != null || maxPrice != null) {
            spec = spec.and(ProductSpecifications.priceBetween(minPrice,
maxPrice));
        }

        if (Boolean.TRUE.equals(inStockOnly)) {
            spec = spec.and(ProductSpecifications.inStock());
        }

        return productRepository.findAll(spec);
    }
}
```

Complex Joins and Aggregations with Specifications

For more complex scenarios involving joins and nested properties:

```
public static Specification<Order> hasProductInCategory(String category) {
    return (root, query, cb) -> {
        Join<Order, OrderItem> orderItems = root.join("orderItems");
        Join<OrderItem, Product> product = orderItems.join("product");
        return cb.equal(product.get("category"), category);
    };
}

public static Specification<Employee> hasSkill(String skillName) {
    return (root, query, cb) -> {
        // To avoid duplicate results in the final query
        if (Long.class != query.getResultType()) {
            query.distinct(true);
        }
        Join<Employee, Skill> skills = root.join("skills");
        return cb.equal(skills.get("name"), skillName);
    };
}
```

Implementing a Generic Specification Builder

For even more flexibility, you can create a generic specification builder:

```
public class GenericSpecificationBuilder<T> {

    private final List<SearchCriteria> criteria = new ArrayList<>();

    public GenericSpecificationBuilder<T> with(String key, String operation,
Object value) {
        criteria.add(new SearchCriteria(key, operation, value));
        return this;
    }

    public Specification<T> build() {
        if (criteria.size() == 0) {
            return null;
        }

        Specification<T> result = new GenericSpecification<>(criteria.get(0));

        for (int i = 1; i < criteria.size(); i++) {
            result = result.and(new GenericSpecification<>(criteria.get(i)));
        }

        return result;
    }

    // Helper class to hold search criteria
    private static class SearchCriteria {
        private String key;
        private String operation;
        private Object value;

        // Constructor, getters, setters...
    }

    // Implementation of individual specifications
    private static class GenericSpecification<T> implements Specification<T> {
        private final SearchCriteria criteria;

        // Constructor

        @Override
        public Predicate toPredicate(Root<T> root, CriteriaQuery<?> query,
CriteriaBuilder cb) {
            switch (criteria.getOperation()) {
                case "eq":
                    return cb.equal(root.get(criteria.getKey()),
criteria.getValue());
                case "gt":
                    return cb.greaterThan(root.get(criteria.getKey()),
                                          criteria.getValue().toString());
                // More operations...
                default:
                    return null;
            }
```

```
        }
    }
}
```

This approach allows clients to build specifications dynamically based on user input:

```
// Example usage of the generic builder
GenericSpecificationBuilder<Product> builder = new GenericSpecificationBuilder<>
();

if (categoryParam != null) {
    builder.with("category", "eq", categoryParam);
}

if (minPriceParam != null) {
    builder.with("price", "gt", minPriceParam);
}

Specification<Product> spec = builder.build();
List<Product> products = productRepository.findAll(spec);
```

Derived Delete Queries - Efficient Deletion Strategies

Spring Data JPA allows you to define derived delete methods in your repositories:

```
public interface OrderRepository extends JpaRepository<Order, Long> {

    // Delete orders by status
    void deleteByStatus(OrderStatus status);

    // Delete orders created before a specific date
    long deleteByCreatedDateBefore(LocalDate date);

    // Delete orders with a specific customer and status
    @Modifying
    @Transactional
    int deleteByCustomerIdAndStatus(Long customerId, OrderStatus status);
}
```

Important considerations:

- Methods can return `void`, `long`, or `int` to indicate the number of deleted entities
- For bulk deletions, methods should be annotated with `@Modifying` and run within a transaction
- Cascading relationships will be respected for entity-based deletions but not for bulk deletions

Bulk Delete vs. Entity-Based Delete

Spring Data JPA offers two approaches to deletions:

1. **Entity-based deletion**: Loads entities into memory and performs cascading deletes

```
// Entity-based deletion (loads entities first)
List<Product> products = productRepository.findByCategory("Obsolete");
productRepository.deleteAll(products);
```

2. **Bulk deletion**: Executes a direct DELETE query without loading entities

```
// Bulk deletion (direct SQL, no entity loading)
@Modifying
@Query("DELETE FROM Product p WHERE p.category = :category")
int deleteProductsByCategory(@Param("category") String category);
```

Choose the appropriate strategy based on:

- Volume of data (bulk operations are more efficient for large datasets)
- Need for cascading (entity-based deletions respect cascade rules)
- Audit requirements (entity-based deletions trigger entity lifecycle events)

Auditing in Spring Data JPA - Automatic Timestamping and User Tracking

Spring Data JPA provides built-in support for auditing entities, keeping track of when they were created or modified and by whom.

Enabling Auditing

First, enable auditing by adding the `@EnableJpaAuditing` annotation to a configuration class:

```
@Configuration
@EnableJpaAuditing
public class JpaConfig {

    // Optionally define a bean to provide the current auditor (user)
    @Bean
    public AuditorAware<String> auditorProvider() {
        return () -> Optional.ofNullable(SecurityContextHolder.getContext())
                .map(SecurityContext::getAuthentication)
                .filter(Authentication::isAuthenticated)
                .map(Authentication::getName);
    }
}
```

Creating Auditable Entities

Next, create a base class for auditable entities:

```
@MappedSuperclass
@EntityListeners(AuditingEntityListener.class)
public abstract class Auditable {

    @CreatedDate
    @Column(name = "created_date", nullable = false, updatable = false)
    private LocalDateTime createdDate;

    @LastModifiedDate
    @Column(name = "last_modified_date")
    private LocalDateTime lastModifiedDate;

    @CreatedBy
    @Column(name = "created_by", updatable = false)
    private String createdBy;

    @LastModifiedBy
    @Column(name = "last_modified_by")
    private String lastModifiedBy;

    // Getters and setters
}
```

Then extend this base class in your entities:

```
@Entity
public class Customer extends Auditable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String email;

    // Other fields, getters, setters...
}
```

Custom Auditing Fields

You can also create custom audit fields when the standard ones don't meet your needs:

```
@Entity
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // Standard fields
```

```
    // Custom audit field
    @PrePersist
    public void prePersist() {
        orderNumber = "ORD-" + System.currentTimeMillis();
        orderDate = LocalDateTime.now();
    }

    @PreUpdate
    public void preUpdate() {
        lastUpdateTimestamp = LocalDateTime.now();
    }
}
```

Transactions with Spring Data JPA - Declarative Transaction Management

Proper transaction management is crucial for maintaining data integrity, especially for operations that involve multiple entities or steps.

Basic Transaction Management

The simplest way to apply transaction management is with the `@Transactional` annotation:

```
@Service
public class OrderService {

    private final OrderRepository orderRepository;
    private final InventoryService inventoryService;

    @Autowired
    public OrderService(OrderRepository orderRepository, InventoryService
inventoryService) {
        this.orderRepository = orderRepository;
        this.inventoryService = inventoryService;
    }

    @Transactional
    public Order placeOrder(Order order) {
        // Validate order
        validateOrder(order);

        // Reserve inventory
        inventoryService.reserveInventory(order.getItems());

        // Save order
        Order savedOrder = orderRepository.save(order);

        // Process payment
        processPayment(order);

        return savedOrder;
```

```
    }

    // Other methods...
}
```

In this example, if any step fails (e.g., payment processing), the entire transaction is rolled back, preventing inventory from being reserved without a corresponding order.

Transaction Propagation

Transaction propagation defines how transactions relate to each other when one transactional method calls another:

```
@Service
public class OrderService {

    @Transactional
    public Order placeOrder(Order order) {
        // This method has its own transaction
        return processOrder(order);
    }

    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public Order processPayment(Order order) {
        // This will always run in a new transaction
        // Even if called from placeOrder
    }

    @Transactional(propagation = Propagation.MANDATORY)
    public void updateInventory(Order order) {
        // This method requires an existing transaction
        // Will throw an exception if called outside a transaction
    }

    @Transactional(propagation = Propagation.SUPPORTS)
    public Order findOrderDetails(Long orderId) {
        // Will run in a transaction if one exists
        // Otherwise runs non-transactionally
    }
}
```

Common propagation options:

- `REQUIRED` (default): Use current transaction or create a new one if none exists
- `REQUIRES_NEW`: Always create a new transaction, suspending the current one if it exists
- `SUPPORTS`: Use current transaction if it exists, otherwise run non-transactionally
- `MANDATORY`: Must be run within an existing transaction, otherwise throws an exception
- `NEVER`: Must be run outside a transaction, otherwise throws an exception

Transaction Isolation Levels

Isolation levels determine how changes made by one transaction are visible to other concurrent transactions:

```
@Transactional(isolation = Isolation.READ_COMMITTED)
public void updateProduct(Product product) {
    // This method uses READ_COMMITTED isolation
}

@Transactional(isolation = Isolation.SERIALIZABLE)
public void transferFunds(Account from, Account to, BigDecimal amount) {
    // This method uses SERIALIZABLE isolation for maximum safety
}
```

Common isolation levels (from least to most restrictive):

- `READ_UNCOMMITTED`: Can read uncommitted changes from other transactions (dirty reads)
- `READ_COMMITTED`: Can only read committed changes from other transactions
- `REPEATABLE_READ`: Ensures repeated reads of the same data return the same results
- `SERIALIZABLE`: Complete isolation, transactions behave as if they were executed serially

Higher isolation levels provide stronger guarantees but may impact performance due to increased locking.

Transaction Rollback Rules

You can specify when a transaction should be rolled back:

```
@Transactional(rollbackFor = {InventoryException.class})
public Order placeOrder(Order order) throws OrderException, InventoryException {
    // This transaction will roll back for InventoryException
    // But not for OrderException (unless it's a RuntimeException)
}

@Transactional(noRollbackFor = {ValidationException.class})
public void processPayment(Payment payment) {
    // This transaction won't roll back for ValidationException
    // But will for other runtime exceptions
}
```

By default, transactions roll back for runtime exceptions but not for checked exceptions.

Practical Example: Building an Advanced E-commerce Repository

Let's put everything together in a comprehensive example for an e-commerce application:

```
// Entity
@Entity
@Table(name = "products")
```

```java
@EntityListeners(AuditingEntityListener.class)
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String description;
    private BigDecimal price;
    private Integer stockQuantity;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "category_id")
    private Category category;

    @ManyToMany
    @JoinTable(
        name = "product_tags",
        joinColumns = @JoinColumn(name = "product_id"),
        inverseJoinColumns = @JoinColumn(name = "tag_id")
    )
    private Set<Tag> tags = new HashSet<>();

    @CreatedDate
    private LocalDateTime createdDate;

    @LastModifiedDate
    private LocalDateTime lastModifiedDate;

    // Getters, setters, etc.
}

// Projections
public interface ProductSummary {
    Long getId();
    String getName();
    BigDecimal getPrice();
    Integer getStockQuantity();

    @Value("#{target.category.name}")
    String getCategoryName();
}

// Specifications
public class ProductSpecifications {
    public static Specification<Product> nameContains(String name) {
        return (root, query, cb) -> {
            if (name == null) {
                return cb.conjunction();
            }
            return cb.like(cb.lower(root.get("name")), "%" + name.toLowerCase() +
"%");
        };
    }
```

```
    public static Specification<Product> inCategory(Long categoryId) {
        return (root, query, cb) -> {
            if (categoryId == null) {
                return cb.conjunction();
            }
            return cb.equal(root.get("category").get("id"), categoryId);
        };
    }

    public static Specification<Product> hasTags(List<String> tagNames) {
        return (root, query, cb) -> {
            if (tagNames == null || tagNames.isEmpty()) {
                return cb.conjunction();
            }

            query.distinct(true);
            Join<Product, Tag> tags = root.join("tags");
            return tags.get("name").in(tagNames);
        };
    }

    public static Specification<Product> priceInRange(BigDecimal min, BigDecimal
max) {
        return (root, query, cb) -> {
            if (min == null && max == null) {
                return cb.conjunction();
            }
            if (min == null) {
                return cb.lessThanOrEqualTo(root.get("price"), max);
            }
            if (max == null) {
                return cb.greaterThanOrEqualTo(root.get("price"), min);
            }
            return cb.between(root.get("price"), min, max);
        };
    }

    public static Specification<Product> inStock() {
        return (root, query, cb) -> cb.greaterThan(root.get("stockQuantity"), 0);
    }
}

// Repository
public interface ProductRepository extends JpaRepository<Product, Long>,
JpaSpecificationExecutor<Product> {
    // Custom queries
    @Query("SELECT p FROM Product p WHERE p.stockQuantity < :threshold AND
p.category.id = :categoryId")
    List<Product> findLowStockProducts(@Param("threshold") int threshold,
@Param("categoryId") Long categoryId);

    // Projections
    <T> List<T> findByCategory_Id(Long categoryId, Class<T> projection);
```

```
    // Derived delete query
    @Modifying
    @Transactional
    int deleteByLastModifiedDateBefore(LocalDateTime date);

    // Native query
    @Query(value = "SELECT p.* FROM products p " +
                   "JOIN product_tags pt ON p.id = pt.product_id " +
                   "JOIN tags t ON pt.tag_id = t.id " +
                   "WHERE t.name IN :tagNames " +
                   "GROUP BY p.id " +
                   "HAVING COUNT(DISTINCT t.id) = :tagCount",
           nativeQuery = true)
    List<Product> findByAllTagNames(@Param("tagNames") List<String> tagNames,
@Param("tagCount") int tagCount);
}

// Service
@Service
public class ProductService {

    private final ProductRepository productRepository;

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

    @Transactional(readOnly = true)
    public Page<ProductSummary> searchProducts(
            String name,
            Long categoryId,
            List<String> tags,
            BigDecimal minPrice,
            BigDecimal maxPrice,
            boolean inStockOnly,
            Pageable pageable) {

        Specification<Product> spec = Specification.where(null);

        if (name != null && !name.trim().isEmpty()) {
            spec = spec.and(ProductSpecifications.nameContains(name));
        }

        if (categoryId != null) {
            spec = spec.and(ProductSpecifications.inCategory(categoryId));
        }

        if (tags != null && !tags.isEmpty()) {
            spec = spec.and(ProductSpecifications.hasTags(tags));
        }

        if (minPrice != null || maxPrice != null) {
```

```
            spec = spec.and(ProductSpecifications.priceInRange(minPrice,
maxPrice));
        }

        if (inStockOnly) {
            spec = spec.and(ProductSpecifications.inStock());
        }

        return productRepository.findAll(spec, pageable)
                .map(product -> convertToSummary(product));
    }

    @Transactional
    public void updateProductStock(Long productId, int quantityChange) {
        Product product = productRepository.findById(productId)
                .orElseThrow(() -> new ProductNotFoundException(productId));

        int newQuantity = product.getStockQuantity() + quantityChange;
        if (newQuantity < 0) {
            throw new InsufficientStockException(productId,
product.getStockQuantity(), quantityChange);
        }

        product.setStockQuantity(newQuantity);
        productRepository.save(product);
    }

    @Transactional
    @Scheduled(cron = "0 0 1 * * ?")  // Run at 1 AM every day
    public void cleanupDiscontinuedProducts() {
        LocalDateTime sixMonthsAgo = LocalDateTime.now().minusMonths(6);
        int deleted =
productRepository.deleteByLastModifiedDateBefore(sixMonthsAgo);
        log.info("Deleted {} discontinued products", deleted);
    }

    // Private helper methods...
}
```

Chapter Summary and Key Takeaways

In this chapter, we explored advanced features of Spring Data JPA that enable sophisticated data access patterns:

- **@Query Annotation**: Provides flexibility to write custom JPQL and native SQL queries when derived methods aren't sufficient. Use `@Modifying` for update and delete operations.
- **Projections**: Optimize data retrieval by selecting only the attributes you need, with support for interface-based, class-based, and dynamic projections.
- **Specifications**: Enable type-safe, programmatic building of complex queries, particularly useful for dynamic filtering scenarios. Can be combined for powerful query composition.

- **Derived Delete Queries**: Offer a convenient way to define deletion operations based on entity properties, with support for both entity-based and bulk deletion strategies.
- **Auditing**: Simplify tracking of entity creation and modification with built-in support for timestamps and user tracking.
- **Transaction Management**: Ensure data integrity through declarative transaction control, with support for various propagation and isolation levels.

Chapter 4: Working with Document Databases - Spring Data MongoDB

Introduction to NoSQL Document Databases - MongoDB Concepts

In the world of database management systems NoSQL databases emerged as a response to the changing needs of modern applications, particularly those requiring flexibility, scalability, and handling of unstructured or semi-structured data.

Document databases represent one of the most popular categories of NoSQL databases, with MongoDB being the leading implementation. Unlike relational databases that organize data in tables with fixed schemas, document databases store data in flexible, JSON-like documents.

Core MongoDB Concepts

Documents: The fundamental unit of data in MongoDB. A document is analogous to a row in a relational database but has a flexible schema. Each document is essentially a JSON-like structure (technically BSON, which is binary JSON) containing field-value pairs. For example:

```
{
  "_id": "5f8a716b1cd7f23c9d3a9876",
  "title": "MongoDB Basics",
  "author": {
    "firstName": "John",
    "lastName": "Smith"
  },
  "tags": ["mongodb", "nosql", "document-database"],
  "comments": [
    {
      "user": "alex",
      "text": "Great explanation!",
      "date": "2023-02-15T10:30:00Z"
    }
  ]
}
```

Collections: Groups of documents, similar to tables in relational databases. However, documents within a collection can have different structures—there's no requirement that all documents follow the same schema.

Databases: Containers for collections, analogous to schemas in relational databases.

Comparing Document Databases with Relational Databases

Feature	Relational Databases (MySQL, PostgreSQL)	Document Databases (MongoDB)
Data Model	Tables with rows and columns	Collections of JSON-like documents
Schema	Fixed schema, predefined structure	Flexible schema, dynamic structure
Relationships	Foreign keys, joins	Embedded documents or references
Query Language	SQL	JSON-based query language
Scalability	Vertical scaling (bigger servers)	Horizontal scaling (more servers)
Use Cases	Complex transactions, structured data	Semi-structured data, rapid development
Data Integrity	ACID transactions	BASE principles (eventual consistency)

Schema Flexibility

One of the most significant advantages of document databases is schema flexibility. In a relational database, adding a new column typically requires altering the table, which can be problematic for large datasets. In MongoDB, you can simply start including new fields in your documents without modifying the entire collection.

This flexibility is particularly valuable in scenarios where:

- Your data structure evolves over time
- Different entities have varying attributes
- You're dealing with external data sources with inconsistent structures
- You need to rapidly iterate on your data model during development

Let's consider an example where we're building a product catalog. Some products might have color options, while others have size variations, and still others have technical specifications. In a relational model, you'd need multiple tables with foreign keys, or a complex table with many nullable columns. In MongoDB, each product can have its own unique structure:

```
// A clothing product
{
  "_id": "product1",
  "name": "Cotton T-Shirt",
  "category": "Clothing",
  "price": 19.99,
  "colors": ["Red", "Blue", "Black"],
  "sizes": ["S", "M", "L", "XL"]
}
```

```
// An electronics product
{
  "_id": "product2",
  "name": "Smartphone",
  "category": "Electronics",
  "price": 699.99,
  "specifications": {
    "processor": "Octa-core",
    "memory": "8GB",
    "storage": "128GB"
  },
  "warranty": "2 years"
}
```

Both documents coexist in the same collection despite having different structures, a flexibility that would be challenging to achieve in a relational database.

Spring Data MongoDB - Simplifying MongoDB Interaction in Spring Boot

Spring Data MongoDB extends the familiar Spring Data paradigm we explored with JPA to provide a seamless integration between Spring Boot applications and MongoDB databases. Just as Spring Data JPA simplified our interaction with relational databases, Spring Data MongoDB offers similar abstractions and conveniences for working with MongoDB.

Key Features of Spring Data MongoDB

1. **Object-Document Mapping (ODM)**: Maps Java objects to MongoDB documents and vice versa, similar to how JPA handles object-relational mapping.
2. **Repository Support**: Provides a repository abstraction that eliminates boilerplate code for common database operations.
3. **Query Methods**: Allows defining query methods by method name convention, similar to Spring Data JPA.
4. **Template Operations**: Offers `MongoTemplate` for more complex operations, analogous to `JdbcTemplate` but for MongoDB.
5. **Annotation-based Mapping**: Uses annotations like `@Document` and `@Field` to define how Java objects map to MongoDB documents.
6. **Aggregation Framework Support**: Provides Java-based support for MongoDB's powerful aggregation capabilities.

Spring Data MongoDB lets us work with MongoDB while maintaining the consistency and familiarity of the Spring ecosystem. This means you can leverage your existing Spring knowledge while incorporating the flexibility and scalability benefits of MongoDB.

Setting up MongoDB and Spring Boot MongoDB Project

Let's walk through the process of setting up a MongoDB instance and configuring a Spring Boot project to work with it.

Setting Up MongoDB

You have several options for running MongoDB:

Option 1: Local Installation

1. Download MongoDB Community Edition from the official MongoDB website.
2. Follow the installation instructions for your operating system.
3. Start the MongoDB server (typically using the `mongod` command).

Option 2: Docker Container If you're familiar with Docker, this is a convenient option:

```
docker run --name mongodb -d -p 27017:27017 mongo
```

Option 3: MongoDB Atlas (Cloud Service)

1. Create an account on MongoDB Atlas.
2. Create a new cluster (they offer a free tier).
3. Set up database access credentials and network access rules.
4. Get your connection string from the Atlas dashboard.

Configuring a Spring Boot Project

1. Create a new Spring Boot project, either through Spring Initializr (https://start.spring.io/) or your IDE, selecting the following dependencies:
    - Spring Web
    - Spring Data MongoDB
2. Add the MongoDB dependency to your `pom.xml`:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
```

3. Configure the MongoDB connection in your `application.properties` or `application.yml`:

```
# For local MongoDB
spring.data.mongodb.host=localhost
spring.data.mongodb.port=27017
spring.data.mongodb.database=bookstore

# OR, for MongoDB Atlas or other remote MongoDB
```

```
#
spring.data.mongodb.uri=mongodb+srv://username:password@cluster.mongodb.net/bookst
ore
```

4. Verify the connection by creating a simple test class:

```
@SpringBootApplication
public class MongoDbApplication implements CommandLineRunner {

    @Autowired
    private MongoTemplate mongoTemplate;

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

    @Override
    public void run(String... args) throws Exception {
        // This will throw an exception if the connection fails
        System.out.println("Successfully connected to MongoDB: "
                           + mongoTemplate.getDb().getName());
    }
}
```

MongoDB Documents and Collections - Mapping Java Objects to Documents

In Spring Data MongoDB, we use annotations to define how our Java objects map to MongoDB documents. This concept is similar to how we used JPA annotations to map entities to relational tables, but with adaptations specific to MongoDB's document model.

Basic Document Mapping

Let's create a simple `Book` class and map it to a MongoDB collection:

```
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;
import org.springframework.data.mongodb.core.mapping.Field;
import java.util.List;
import java.util.Date;

@Document(collection = "books")
public class Book {

    @Id
    private String id;  // MongoDB uses String IDs by default

    private String title;
```

```
    private String author;

    @Field("publication_date")  // Custom field name in document
    private Date publicationDate;

    private double price;

    private List<String> categories;

    // Constructor, getters, and setters
    // ...
}
```

Key Annotations

1. `@Document`: Marks a class as a domain object to be persisted to MongoDB. The optional `collection` attribute specifies the name of the collection.
2. `@Id`: Designates a field as the document's ID. In MongoDB, the ID field is named `_id` and is automatically created if not provided.
3. `@Field`: Specifies the name of the field as it will appear in the MongoDB document. This is useful when the Java property name differs from the desired document field name.
4. `@Transient`: Indicates that a field should not be persisted to the database.

Embedded Documents

MongoDB excels at representing hierarchical data structures through embedded documents. Spring Data MongoDB makes it easy to map these relationships:

```
@Document(collection = "books")
public class Book {

    @Id
    private String id;

    private String title;

    // Embedding an Author object
    private Author author;

    // Embedding a list of Review objects
    private List<Review> reviews;

    // Other fields...
}

public class Author {
    private String firstName;
    private String lastName;
```

```
    private String biography;
    // Getters, setters...
}

public class Review {
    private String username;
    private int rating;
    private String comment;
    private Date date;
    // Getters, setters...
}
```

When saved, this structure creates a document with nested objects:

```
{
  "_id": "60c73a5b2b50d43bdca4567d",
  "title": "Understanding MongoDB",
  "author": {
    "firstName": "Jane",
    "lastName": "Doe",
    "biography": "Database expert and author"
  },
  "reviews": [
    {
      "username": "reader123",
      "rating": 5,
      "comment": "Excellent book!",
      "date": "2023-03-10T14:30:00Z"
    },
    {
      "username": "tech_guru",
      "rating": 4,
      "comment": "Very informative",
      "date": "2023-03-15T09:45:00Z"
    }
  ]
}
```

This embedded structure eliminates the need for joins, as all related data is stored within a single document. This approach is particularly efficient for data that is always accessed together.

Document References

For cases where embedding isn't appropriate (e.g., when the related data is large or shared across multiple documents), MongoDB supports references between documents. Spring Data MongoDB provides the `@DBRef` annotation for this purpose:

```
@Document(collection = "books")
public class Book {
```

```
    @Id
    private String id;

    private String title;

    // Reference to a Publisher document
    @DBRef
    private Publisher publisher;

    // Other fields...
}

@Document(collection = "publishers")
public class Publisher {

    @Id
    private String id;

    private String name;

    private String address;

    // Other publisher details...
}
```

When deciding between embedding and referencing, consider:

- Embedding: For "has-a" relationships where the child objects are always accessed with the parent and are unique to each parent.
- Referencing: For "is-a" relationships where the child objects are shared across multiple parents or are large enough to impact performance if embedded.

Spring Data MongoDB Repositories - Data Access for MongoDB

Spring Data MongoDB repositories follow the same pattern as Spring Data JPA repositories but are adapted for MongoDB's document model. They provide a high-level abstraction that eliminates boilerplate code for common data access operations.

Creating a MongoDB Repository

To create a MongoDB repository, we extend one of Spring Data MongoDB's repository interfaces:

```
import org.springframework.data.mongodb.repository.MongoRepository;

public interface BookRepository extends MongoRepository<Book, String> {

    // Custom query methods go here
    // ...
}
```

The `MongoRepository` interface takes two type parameters:

1. The domain class (entity) the repository manages
2. The type of the entity's ID field

By extending `MongoRepository`, our `BookRepository` automatically inherits a comprehensive set of methods for CRUD operations, including:

- `save()` - Save or update a document
- `findById()` - Find a document by ID
- `findAll()` - Retrieve all documents
- `delete()` - Delete a document
- `count()` - Count documents

Differences from JPA Repositories

While Spring Data MongoDB repositories share many similarities with JPA repositories, there are some notable differences:

1. **ID Generation**: MongoDB automatically generates IDs (typically ObjectIds) if not provided, whereas JPA often relies on database-specific sequences or identity columns.
2. **Query Language**: JPA repositories use JPQL (Java Persistence Query Language), while MongoDB repositories use MongoDB's query language.
3. **Native Queries**: JPA uses `@Query` with JPQL or SQL, whereas MongoDB uses `@Query` with JSON-like query documents.
4. **Transactions**: MongoDB has different transaction semantics compared to relational databases, especially in versions prior to 4.0.
5. **No Entity Manager**: MongoDB repositories operate on a `MongoTemplate` rather than an `EntityManager`.

Basic CRUD Operations with Spring Data MongoDB Repositories - Save, Find, Delete, etc.

Let's explore how to perform basic CRUD operations using Spring Data MongoDB repositories.

Creating and Saving Documents

```
@Service
public class BookService {

    @Autowired
    private BookRepository bookRepository;

    public Book addBook(Book book) {
        // If id is null, MongoDB will generate one
        return bookRepository.save(book);
```

```
    }

    public List<Book> addBooks(List<Book> books) {
        return bookRepository.saveAll(books);
    }
}
```

Reading Documents

```
@Service
public class BookService {

    @Autowired
    private BookRepository bookRepository;

    public Optional<Book> findBookById(String id) {
        return bookRepository.findById(id);
    }

    public List<Book> findAllBooks() {
        return bookRepository.findAll();
    }

    public Page<Book> findBooksWithPagination(int page, int size) {
        Pageable pageable = PageRequest.of(page, size);
        return bookRepository.findAll(pageable);
    }
}
```

Updating Documents

```
@Service
public class BookService {

    @Autowired
    private BookRepository bookRepository;

    public Book updateBook(Book book) {
        // Verify the book exists
        if (bookRepository.existsById(book.getId())) {
            return bookRepository.save(book);
        }
        throw new BookNotFoundException("Book not found with id: " +
book.getId());
    }

    public Book updateBookPrice(String id, double newPrice) {
        Optional<Book> bookOpt = bookRepository.findById(id);
        if (bookOpt.isPresent()) {
```

```
            Book book = bookOpt.get();
            book.setPrice(newPrice);
            return bookRepository.save(book);
        }
        throw new BookNotFoundException("Book not found with id: " + id);
    }
}
```

Deleting Documents

```
@Service
public class BookService {

    @Autowired
    private BookRepository bookRepository;

    public void deleteBook(String id) {
        bookRepository.deleteById(id);
    }

    public void deleteBooks(List<String> ids) {
        List<Book> booksToDelete = bookRepository.findAllById(ids);
        bookRepository.deleteAll(booksToDelete);
    }

    public void deleteAllBooks() {
        bookRepository.deleteAll();
    }
}
```

Example: Complete CRUD REST Controller

```
@RestController
@RequestMapping("/api/books")
public class BookController {

    @Autowired
    private BookService bookService;

    @PostMapping
    public ResponseEntity<Book> createBook(@RequestBody Book book) {
        Book savedBook = bookService.addBook(book);
        return new ResponseEntity<>(savedBook, HttpStatus.CREATED);
    }

    @GetMapping("/{id}")
    public ResponseEntity<Book> getBookById(@PathVariable String id) {
        return bookService.findBookById(id)
                .map(book -> new ResponseEntity<>(book, HttpStatus.OK))
```

```
                .orElseGet(() -> new ResponseEntity<>(HttpStatus.NOT_FOUND));
    }

    @GetMapping
    public ResponseEntity<List<Book>> getAllBooks() {
        List<Book> books = bookService.findAllBooks();
        return new ResponseEntity<>(books, HttpStatus.OK);
    }

    @PutMapping("/{id}")
    public ResponseEntity<Book> updateBook(@PathVariable String id, @RequestBody
Book book) {
        book.setId(id);
        try {
            Book updatedBook = bookService.updateBook(book);
            return new ResponseEntity<>(updatedBook, HttpStatus.OK);
        } catch (BookNotFoundException e) {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteBook(@PathVariable String id) {
        bookService.deleteBook(id);
        return new ResponseEntity<>(HttpStatus.NO_CONTENT);
    }
}
```

Querying in Spring Data MongoDB - Query Methods and MongoTemplate

Spring Data MongoDB offers two main approaches to querying:

1. Query methods defined in repository interfaces
2. The more flexible `MongoTemplate` for complex operations

Query Methods in MongoDB Repositories

Similar to Spring Data JPA, Spring Data MongoDB allows you to define query methods by method name convention. The repository implementation automatically translates these method names into MongoDB queries.

```
public interface BookRepository extends MongoRepository<Book, String> {

    // Find by a single property
    List<Book> findByTitle(String title);

    // Find by multiple properties
    List<Book> findByAuthorAndPrice(String author, double price);

    // Using comparison operators
```

```
    List<Book> findByPriceGreaterThan(double price);
    List<Book> findByPriceBetween(double minPrice, double maxPrice);

    // Working with dates
    List<Book> findByPublicationDateAfter(Date date);

    // String operations
    List<Book> findByTitleContaining(String titlePart);
    List<Book> findByTitleStartingWith(String prefix);

    // Working with embedded documents
    List<Book> findByAuthorFirstName(String firstName);

    // Working with collections/arrays
    List<Book> findByCategoriesContaining(String category);

    // Ordering results
    List<Book> findByAuthorOrderByPublicationDateDesc(String author);

    // Limiting results
    List<Book> findTop5ByOrderByPriceDesc();
}
```

Using @Query Annotation

For more complex queries, you can use the `@Query` annotation with MongoDB's JSON query syntax:

```
public interface BookRepository extends MongoRepository<Book, String> {

    @Query("{ 'price': { $lt: ?0 }, 'categories': ?1 }")
    List<Book> findCheapBooksByCategory(double maxPrice, String category);

    @Query("{ 'author.lastName': ?0 }")
    List<Book> findBooksByAuthorLastName(String lastName);

    @Query(value = "{ 'title': { $regex: ?0, $options: 'i' } }",
           fields = "{ 'title': 1, 'author': 1, 'price': 1 }")
    List<Book> findBooksByTitleRegexProjected(String titleRegex);
}
```

The `@Query` annotation accepts:

- A `value` parameter with the MongoDB query document
- An optional `fields` parameter for projection (limiting returned fields)

Using MongoTemplate

While repositories are convenient for standard operations, `MongoTemplate` provides more flexibility for complex queries and operations:

```
@Service
public class BookService {

    @Autowired
    private MongoTemplate mongoTemplate;

    public List<Book> findExpensiveBooksByAuthor(String author, double minPrice) {
        Query query = new Query();
        query.addCriteria(Criteria.where("author").is(author)
                                  .and("price").gte(minPrice));
        return mongoTemplate.find(query, Book.class);
    }

    public Book updateBookCategories(String id, List<String> newCategories) {
        Query query = new Query(Criteria.where("id").is(id));
        Update update = new Update().set("categories", newCategories);
        return mongoTemplate.findAndModify(query, update, Book.class);
    }

    public Book saveBookWithCustomId(Book book, String customId) {
        book.setId(customId);
        return mongoTemplate.save(book);
    }

    public long countBooksPublishedAfter(Date date) {
        Query query = new Query(Criteria.where("publicationDate").gt(date));
        return mongoTemplate.count(query, Book.class);
    }

    public void removeOutOfStockBooks() {
        Query query = new Query(Criteria.where("inStock").is(false));
        mongoTemplate.remove(query, Book.class);
    }
}
```

Advanced Query Techniques with MongoTemplate

```
@Service
public class BookService {

    @Autowired
    private MongoTemplate mongoTemplate;

    // Complex criteria
    public List<Book> findBooksByCriteriaAndSort(String titlePattern,
                                                 List<String> categories,
                                                 double minPrice,
                                                 double maxPrice) {
        Query query = new Query();
```

```
        Criteria criteria = new Criteria();

        // Title pattern
        if (titlePattern != null && !titlePattern.isEmpty()) {
            criteria.and("title").regex(titlePattern, "i");
        }

        // Categories
        if (categories != null && !categories.isEmpty()) {
            criteria.and("categories").in(categories);
        }

        // Price range
        if (minPrice > 0 || maxPrice > 0) {
            Criteria priceCriteria = Criteria.where("price");
            if (minPrice > 0) {
                priceCriteria.gte(minPrice);
            }
            if (maxPrice > 0) {
                priceCriteria.lte(maxPrice);
            }
            criteria.and("price").is(priceCriteria);
        }

        query.addCriteria(criteria);

        // Sorting
        query.with(Sort.by(Sort.Direction.DESC, "publicationDate"));

        return mongoTemplate.find(query, Book.class);
    }

    // Working with subdocuments and arrays
    public List<Book> findBooksWithHighRatings(double minRating) {
        Query query = new Query();
        query.addCriteria(Criteria.where("reviews.rating").gte(minRating));
        return mongoTemplate.find(query, Book.class);
    }

    // Multiple updates
    public void applyDiscountToExpensiveBooks(double threshold, double
discountPercentage) {
        Query query = new Query(Criteria.where("price").gt(threshold));

        // Using multiply to apply a percentage discount
        Update update = new Update().multiply("price", (100 - discountPercentage)
/ 100);

        mongoTemplate.updateMulti(query, update, Book.class);
    }
}
```

Indexing in MongoDB with Spring Data MongoDB - Improving Query Performance

Indexes are crucial for improving query performance in MongoDB, just as they are in relational databases. Spring Data MongoDB provides annotation-based support for creating and managing indexes.

Creating Simple Indexes

You can define indexes directly on your document classes using the `@Indexed` annotation:

```
@Document(collection = "books")
public class Book {

    @Id
    private String id;

    @Indexed
    private String title;

    @Indexed
    private String author;

    @Indexed(direction = IndexDirection.DESCENDING)
    private Date publicationDate;

    @Indexed(name = "price_index")
    private double price;

    // Other fields...
}
```

Compound Indexes

For queries that involve multiple fields, compound indexes can provide significant performance improvements:

```
@Document(collection = "books")
@CompoundIndex(name = "author_title", def = "{'author': 1, 'title': 1}")
@CompoundIndex(name = "category_price", def = "{'categories': 1, 'price': -1}")
public class Book {
    // Fields...
}
```

The `def` attribute uses MongoDB's index definition syntax, where `1` indicates ascending order and `-1` indicates descending order.

Unique Indexes

To enforce uniqueness constraints, use the `unique` attribute:

```
@Document(collection = "books")
public class Book {

    @Id
    private String id;

    @Indexed(unique = true)
    private String isbn;

    // Other fields...
}
```

Text Indexes

MongoDB offers text indexes for full-text search capabilities:

```
@Document(collection = "books")
@TextIndexed(name = "book_text_index",
             fields = {
                @TextIndexed.Field(value = "title", weight = 3),
                @TextIndexed.Field(value = "description", weight = 2),
                @TextIndexed.Field(value = "author", weight = 1)
             })
public class Book {
    // Fields...
}
```

This creates a text index that allows full-text search across the specified fields, with different weights to prioritize matches in certain fields.

Programmatic Index Creation

In addition to annotations, you can create indexes programmatically using `MongoTemplate`:

```
@PostConstruct
public void initIndexes() {
    mongoTemplate.indexOps(Book.class)
        .ensureIndex(new Index().on("publisher", Direction.ASC)
                                .on("publicationYear", Direction.DESC)
                                .named("publisher_year_idx"));
}
```

Index Management Considerations

1. **Index Size**: Indexes consume disk space and memory, so create only indexes that improve your common queries.
2. **Write Performance**: While indexes speed up queries, they slow down write operations, as each index needs to be updated.
3. **Background Creation**: For large collections, consider creating indexes in the background to avoid blocking operations.
4. **Regular Evaluation**: Regularly evaluate your index usage with MongoDB's tools like `db.collection.getIndexes()` and `explain()`.
5. **TTL Indexes**: For time-series data, consider Time-To-Live (TTL) indexes to automatically expire documents after a certain period.

```
@Document(collection = "sessions")
public class UserSession {

    @Id
    private String id;

    @Indexed(expireAfterSeconds = 3600) // Expires after 1 hour
    private Date lastActivity;

    // Other fields...
}
```

Aggregation Framework in MongoDB with Spring Data MongoDB

MongoDB's Aggregation Framework is a powerful tool for data processing and analysis, allowing you to perform complex operations on your data. Spring Data MongoDB provides a fluent API to work with aggregations.

Basic Aggregation Operations

```
@Service
public class BookAnalyticsService {

    @Autowired
    private MongoTemplate mongoTemplate;

    // Group books by author and count
    public List<AuthorBookCount> countBooksByAuthor() {
        TypedAggregation<Book> aggregation = Aggregation.newAggregation(
            Book.class,
            Aggregation.group("author").count().as("bookCount"),
            Aggregation.project("bookCount").and("author").previousOperation(),
            Aggregation.sort(Sort.Direction.DESC, "bookCount")
        );
```

```
        AggregationResults<AuthorBookCount> results =
            mongoTemplate.aggregate(aggregation, AuthorBookCount.class);

        return results.getMappedResults();
    }
}

public class AuthorBookCount {
    private String author;
    private int bookCount;

    // Getters and setters...
}
```

More Complex Aggregation Example

```
@Service
public class BookAnalyticsService {

    @Autowired
    private MongoTemplate mongoTemplate;

    // Calculate average book price by category with additional statistics
    public List<CategoryPriceStatistics> calculatePriceStatsByCategory() {
        TypedAggregation<Book> aggregation = Aggregation.newAggregation(
            Book.class,
            // Unwind categories array to create a document for each category
            Aggregation.unwind("categories"),

            // Group by category and calculate statistics
            Aggregation.group("categories")
                .avg("price").as("averagePrice")
                .min("price").as("minPrice")
                .max("price").as("maxPrice")
                .count().as("bookCount"),

            // Add calculated fields
            Aggregation.project("averagePrice", "minPrice", "maxPrice",
"bookCount")
                .and("categories").previousOperation()
                .andExpression("maxPrice - minPrice").as("priceRange"),

            // Sort by average price
            Aggregation.sort(Sort.Direction.DESC, "averagePrice")
        );

        AggregationResults<CategoryPriceStatistics> results =
            mongoTemplate.aggregate(aggregation, CategoryPriceStatistics.class);

        return results.getMappedResults();
```

```
    }
}

public class CategoryPriceStatistics {
    private String categories;
    private double averagePrice;
    private double minPrice;
    private double maxPrice;
    private int bookCount;
    private double priceRange;

    // Getters and setters...
}
```

Common Aggregation Stages

The MongoDB Aggregation Framework provides numerous stages for processing data:

- `$match`: Filters documents (like a WHERE clause)
- `$group`: Groups documents by a specified expression
- `$project`: Reshapes documents (selecting, renaming, computing fields)
- `$sort`: Sorts documents
- `$limit`/`$skip`: Limits or skips a number of documents
- `$unwind`: Deconstructs an array field to create a document for each element
- `$lookup`: Performs a left outer join to another collection
- `$count`: Counts the number of documents
- `$addFields`: Adds new fields to documents

Spring Data MongoDB provides methods corresponding to each of these stages in its `Aggregation` class.

Practical Examples and Code Demonstrations

Let's put everything together in a comprehensive example of a book management system using Spring Data MongoDB.

Domain Model

```
@Document(collection = "books")
@CompoundIndex(name = "author_title", def = "{'author.lastName': 1, 'title': 1}")
public class Book {

    @Id
    private String id;

    @Indexed
    private String title;

    private Author author;

    @Field("publication_date")
```

```
    private Date publicationDate;

    @Indexed
    private double price;

    private boolean available;

    private List<String> categories;

    private List<Review> reviews;

    @DBRef
    private Publisher publisher;

    // Getters, setters, etc.
}

public class Author {
    private String firstName;
    private String lastName;
    private String biography;

    // Getters, setters, etc.
}

public class Review {
    private String username;
    private int rating; // 1-5
    private String comment;
    private Date date;

    // Getters, setters, etc.
}

@Document(collection = "publishers")
public class Publisher {

    @Id
    private String id;

    @Indexed(unique = true)
    private String name;

    private String
    private String address;

    private String website;

    private List<String> genres;

    // Getters, setters, etc.
}
```

Repository Interfaces

```
public interface BookRepository extends MongoRepository<Book, String> {

    // Basic query methods
    List<Book> findByTitle(String title);
    List<Book> findByAuthorLastName(String lastName);
    List<Book> findByPriceBetween(double minPrice, double maxPrice);
    List<Book> findByCategoriesContaining(String category);
    List<Book> findByAvailableTrue();

    // Advanced query methods
    List<Book> findByAuthorLastNameAndPriceLessThan(String lastName, double
price);
    List<Book> findByPublicationDateAfterAndCategoriesContaining(Date date, String
category);

    // Custom queries
    @Query("{ 'reviews.rating': { $gte: ?0 } }")
    List<Book> findByMinimumRating(int rating);

    @Query(value = "{ 'author.lastName': ?0 }", fields = "{ 'title': 1, 'price': 1
}")
    List<Book> findBooksByAuthorLastNameProjected(String lastName);

    @Query("{ 'publisher.$id': ?0 }")
    List<Book> findByPublisherId(String publisherId);
}

public interface PublisherRepository extends MongoRepository<Publisher, String> {

    Optional<Publisher> findByName(String name);

    List<Publisher> findByGenresContaining(String genre);
}
```

Service Layer

```
@Service
public class BookService {

    @Autowired
    private BookRepository bookRepository;

    @Autowired
    private PublisherRepository publisherRepository;

    @Autowired
    private MongoTemplate mongoTemplate;
```

```
    // Basic CRUD operations
    public List<Book> findAllBooks() {
        return bookRepository.findAll();
    }

    public Optional<Book> findBookById(String id) {
        return bookRepository.findById(id);
    }

    public Book saveBook(Book book) {
        return bookRepository.save(book);
    }

    public void deleteBook(String id) {
        bookRepository.deleteById(id);
    }

    // Business operations
    public Book addReviewToBook(String bookId, Review review) {
        Book book = bookRepository.findById(bookId)
            .orElseThrow(() -> new RuntimeException("Book not found"));

        if (book.getReviews() == null) {
            book.setReviews(new ArrayList<>());
        }

        review.setDate(new Date());
        book.getReviews().add(review);

        return bookRepository.save(book);
    }

    public List<Book> findBooksByAuthorAndCategory(String authorLastName, String
category) {
        Query query = new Query();

        Criteria criteria = new Criteria();

        if (authorLastName != null && !authorLastName.isEmpty()) {
            criteria.and("author.lastName").is(authorLastName);
        }

        if (category != null && !category.isEmpty()) {
            criteria.and("categories").in(category);
        }

        query.addCriteria(criteria);

        return mongoTemplate.find(query, Book.class);
    }

    public void updateBookAvailability(String id, boolean available) {
        Query query = new Query(Criteria.where("id").is(id));
        Update update = new Update().set("available", available);
```

```
        mongoTemplate.updateFirst(query, update, Book.class);
    }

    public void applyDiscountToCategory(String category, double
discountPercentage) {
        Query query = new Query(Criteria.where("categories").is(category));
        Update update = new Update().multiply("price", (100 - discountPercentage)
/ 100);
        mongoTemplate.updateMulti(query, update, Book.class);
    }

    // Aggregation operations
    public List<CategoryStatistics> getBookStatisticsByCategory() {
        TypedAggregation<Book> aggregation = Aggregation.newAggregation(
            Book.class,
            // Unwind categories to create a document for each category
            Aggregation.unwind("categories"),

            // Group by category
            Aggregation.group("categories")
                .count().as("count")
                .avg("price").as("averagePrice")
                .sum("price").as("totalValue"),

            // Project fields and add calculation
            Aggregation.project("count", "averagePrice", "totalValue")
                .and("categories").previousOperation(),

            // Sort by count descending
            Aggregation.sort(Sort.Direction.DESC, "count")
        );

        AggregationResults<CategoryStatistics> results =
            mongoTemplate.aggregate(aggregation, CategoryStatistics.class);

        return results.getMappedResults();
    }

    public List<BooksByPublisher> countBooksByPublisher() {
        TypedAggregation<Book> aggregation = Aggregation.newAggregation(
            Book.class,
            Aggregation.lookup("publishers", "publisher", "_id",
"publisherDetails"),
            Aggregation.unwind("publisherDetails"),
            Aggregation.group("publisherDetails.name").count().as("bookCount"),
            Aggregation.project("bookCount").and("publisher").previousOperation(),
            Aggregation.sort(Sort.Direction.DESC, "bookCount")
        );

        AggregationResults<BooksByPublisher> results =
            mongoTemplate.aggregate(aggregation, BooksByPublisher.class);

        return results.getMappedResults();
    }
```

```
}

public class CategoryStatistics {
    private String categories;
    private long count;
    private double averagePrice;
    private double totalValue;

    // Getters, setters, etc.
}

public class BooksByPublisher {
    private String publisher;
    private long bookCount;

    // Getters, setters, etc.
}
```

REST Controller

```
@RestController
@RequestMapping("/api/books")
public class BookController {

    @Autowired
    private BookService bookService;

    @GetMapping
    public ResponseEntity<List<Book>> getAllBooks() {
        return ResponseEntity.ok(bookService.findAllBooks());
    }

    @GetMapping("/{id}")
    public ResponseEntity<Book> getBookById(@PathVariable String id) {
        return bookService.findBookById(id)
            .map(ResponseEntity::ok)
            .orElse(ResponseEntity.notFound().build());
    }

    @PostMapping
    public ResponseEntity<Book> createBook(@RequestBody Book book) {
        Book savedBook = bookService.saveBook(book);
        return ResponseEntity.status(HttpStatus.CREATED).body(savedBook);
    }

    @PutMapping("/{id}")
    public ResponseEntity<Book> updateBook(@PathVariable String id, @RequestBody
Book book) {
        book.setId(id);
        Book updatedBook = bookService.saveBook(book);
        return ResponseEntity.ok(updatedBook);
```

```
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteBook(@PathVariable String id) {
        bookService.deleteBook(id);
        return ResponseEntity.noContent().build();
    }

    @PostMapping("/{id}/reviews")
    public ResponseEntity<Book> addReview(@PathVariable String id, @RequestBody
Review review) {
        Book updatedBook = bookService.addReviewToBook(id, review);
        return ResponseEntity.ok(updatedBook);
    }

    @PutMapping("/{id}/availability")
    public ResponseEntity<Void> updateAvailability(@PathVariable String id,
                                                   @RequestParam boolean available) {
        bookService.updateBookAvailability(id, available);
        return ResponseEntity.noContent().build();
    }

    @GetMapping("/search")
    public ResponseEntity<List<Book>> searchBooks(
            @RequestParam(required = false) String author,
            @RequestParam(required = false) String category) {
        List<Book> books = bookService.findBooksByAuthorAndCategory(author,
category);
        return ResponseEntity.ok(books);
    }

    @GetMapping("/statistics/categories")
    public ResponseEntity<List<CategoryStatistics>> getCategoryStatistics() {
        return ResponseEntity.ok(bookService.getBookStatisticsByCategory());
    }

    @GetMapping("/statistics/publishers")
    public ResponseEntity<List<BooksByPublisher>> getPublisherStatistics() {
        return ResponseEntity.ok(bookService.countBooksByPublisher());
    }

    @PostMapping("/categories/{category}/discount")
    public ResponseEntity<Void> applyCategoryDiscount(
            @PathVariable String category,
            @RequestParam double percentage) {
        bookService.applyDiscountToCategory(category, percentage);
        return ResponseEntity.noContent().build();
    }
}
```

Chapter Summary and Key Takeaways

In this chapter, we've explored Spring Data MongoDB, which extends the Spring Data paradigm to work with MongoDB document databases. Here are the key takeaways:

Document Databases vs. Relational Databases

1. **Schema Flexibility**: Document databases like MongoDB allow for flexible, evolving data structures, whereas relational databases enforce a fixed schema.
2. **Hierarchical Data**: MongoDB excels at storing hierarchical data through embedded documents, eliminating the need for joins.
3. **Scalability**: MongoDB is designed for horizontal scaling across multiple servers, while traditional relational databases often scale vertically.

Spring Data MongoDB Components

1. **Object-Document Mapping**: Maps Java classes to MongoDB documents using annotations like `@Document`, `@Id`, and `@Field`.
2. **Repository Abstraction**: Provides ready-to-use data access methods through interfaces extending `MongoRepository`.
3. **Query Methods**: Allows defining query operations through method naming conventions that are converted to MongoDB queries.
4. **MongoTemplate**: Offers a more flexible, programmatic API for complex operations beyond basic CRUD.
5. **Index Support**: Provides annotation-based index configuration for optimizing query performance.
6. **Aggregation Framework**: Enables complex data analysis through MongoDB's powerful aggregation pipeline.

When to Choose MongoDB

MongoDB and Spring Data MongoDB are particularly well-suited for:

1. **Evolving Data Models**: When your data schema is likely to change frequently during development.
2. **Hierarchical Data**: When your data naturally forms hierarchical structures with nested objects.
3. **High Write Throughput**: When your application requires high-volume write operations.
4. **Horizontal Scalability**: When you need to scale your database across multiple servers.
5. **Semi-structured Data**: When dealing with data that doesn't fit neatly into a tabular format.

Chapter 5: Working with Key-Value Stores - Spring Data Redis

Introduction to NoSQL Key-Value Stores - Redis Concepts

NoSQL databases emerged as an alternative to traditional relational databases to address the scaling and flexibility challenges encountered in modern applications. Among the various NoSQL database types, key-value stores represent one of the simplest yet most powerful models.

Understanding Key-Value Stores

At their core, key-value stores function like large distributed hash tables. Each data item is stored as a key-value pair, where the key serves as a unique identifier used to retrieve the associated value. This simple model offers significant advantages:

1. **Simplicity**: The data model is straightforward and intuitive.
2. **Performance**: Key-based lookups are extremely fast, often with O(1) time complexity.
3. **Scalability**: The simple data model makes horizontal scaling more straightforward than with relational databases.
4. **Flexibility**: Values can store various data types without enforcing a schema.

Redis - An Advanced Key-Value Store

Redis (Remote Dictionary Server) extends the basic key-value store model with rich data structures and additional features:

1. **In-Memory Processing**: Redis primarily operates in memory, offering exceptional performance with typical operations completing in less than a millisecond.

2. **Persistence Options**: Despite being memory-based, Redis provides durability through:

    - RDB: Point-in-time snapshots at specified intervals
    - AOF: Append-only file that logs every write operation
    - Combined approaches for balanced durability and performance

3. **Rich Data Structures**: Unlike simple key-value stores, Redis supports multiple value types:

    - **Strings**: Simple text or binary data (up to 512MB per string)
    - **Lists**: Ordered collections of strings, ideal for implementing queues
    - **Sets**: Unordered collections of unique strings
    - **Sorted Sets**: Sets with an associated score for ordering elements
    - **Hashes**: Maps between string fields and string values, perfect for representing objects
    - **Bitmaps and HyperLogLogs**: Specialized data structures for space-efficient operations

4. **Additional Features**:

    - **Pub/Sub Messaging**: Support for publish/subscribe communication patterns
    - **Lua Scripting**: Custom operations with Lua scripts
    - **Transactions**: Execution of command groups in isolation
    - **Automatic Expiration**: Time-to-live settings for keys

Comparing Database Models

Feature	Relational DB	Document DB	Key-Value Store (Redis)
Data Model	Tables, rows, columns	JSON/BSON documents	Key-value pairs with rich data structures
Schema	Fixed, rigid	Flexible, schemaless	Schemaless
Query Language	SQL	JSON-based query language	Command-based operations

Feature	Relational DB	Document DB	Key-Value Store (Redis)
Relationships	Foreign keys, joins	Embedded documents, references	Limited relationship support
Transaction Support	ACID compliant	Varies (often document-level)	Limited (multi-key operations)
Performance	Moderate to high	High for document queries	Extremely high for key operations
Use Cases	Complex queries, transactions	Semi-structured data, content management	Caching, real-time analytics, messaging

When to Choose Redis

Redis excels in scenarios requiring:

- **Caching**: Its in-memory nature makes it ideal for caching frequently accessed data
- **Session Storage**: Fast read/write of session data with automatic expiration
- **Real-time Analytics**: Counters, statistics, and leaderboards
- **Messaging**: Pub/sub features for real-time communication
- **Rate Limiting**: Tracking request counts with automatic expiration
- **Geospatial Applications**: Native support for proximity searches

Redis is less suitable when you need:

- Complex transactions spanning many operations
- Advanced querying capabilities without predetermined access patterns
- Storage of very large datasets exceeding available memory

Spring Data Redis - Simplifying Redis Interaction in Spring Boot

Spring Data Redis provides a high-level abstraction for Redis interaction within Spring applications, significantly reducing the complexity of working with Redis while maintaining its power.

Spring Data Redis Architecture

Spring Data Redis sits on top of low-level Redis drivers (primarily Lettuce or Jedis) and offers:

1. **Connection Management**: Handles the complexities of connection pooling and lifecycle management
2. **Exception Translation**: Converts Redis-specific exceptions into Spring's consistent DataAccessException hierarchy
3. **Template-based API**: Provides RedisTemplate and StringRedisTemplate for simplified operations
4. **Object Mapping**: Maps Java objects to Redis data structures
5. **Repository Support**: Offers repository abstractions similar to other Spring Data modules
6. **Messaging Support**: Simplifies working with Redis Pub/Sub

Key Components

1. **RedisConnectionFactory**: Creates and manages Redis connections

2. **RedisTemplate**: The primary interface for Redis operations
3. **Repository Support**: Optional Redis-specific repository interfaces
4. **Serializers/Converters**: For transforming objects to/from Redis storage format
5. **Caching Support**: Integration with Spring's caching abstraction

Integration with Spring Boot

Spring Boot provides auto-configuration for Redis through the `spring-boot-starter-data-redis` dependency, which:

- Automatically configures a RedisConnectionFactory based on application properties
- Creates a RedisTemplate and StringRedisTemplate bean
- Sets up Redis repositories if enabled
- Configures Redis as a cache provider when Spring caching is enabled

This tight integration makes incorporating Redis into Spring Boot applications remarkably straightforward.

Setting up Redis and Spring Boot Redis Project

Let's set up a complete Redis environment for Spring Boot development.

Setting Up Redis

Option 1: Local Installation

For Linux/macOS:

```
# Install using package manager
# For Ubuntu/Debian
sudo apt-get update
sudo apt-get install redis-server

# For macOS using Homebrew
brew install redis

# Start Redis server
sudo systemctl start redis  # Linux
brew services start redis   # macOS
```

For Windows: Windows isn't officially supported, but you can use:

- Microsoft's port: https://github.com/microsoftarchive/redis
- Redis in WSL (Windows Subsystem for Linux)

Option 2: Docker Container

```
# Pull and run Redis container
docker pull redis
docker run --name my-redis -p 6379:6379 -d redis
```

```
# To access Redis CLI in container
docker exec -it my-redis redis-cli
```

Option 3: Redis Cloud Services

- Redis Labs (Redis Cloud)
- AWS ElastiCache
- Azure Cache for Redis
- Google Cloud Memorystore

Creating a Spring Boot Redis Project

1. **Generate a Spring Boot project** using Spring Initializr (https://start.spring.io/) with:
    - Spring Web
    - Spring Data Redis
    - Lombok (optional, for reducing boilerplate)
2. **Configure Redis connection** in `application.properties`:

```
# Redis connection settings
spring.data.redis.host=localhost
spring.data.redis.port=6379
# Optional password if Redis requires authentication
# spring.data.redis.password=your_password
# Optional database index (0-15, default is 0)
# spring.data.redis.database=0

# Connection pool settings
spring.data.redis.lettuce.pool.max-active=8
spring.data.redis.lettuce.pool.max-idle=8
spring.data.redis.lettuce.pool.min-idle=0
```

3. **Create Redis configuration class**:

```
package com.example.redisdemo.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;
import
org.springframework.data.redis.serializer.GenericJackson2JsonRedisSerializer;
import org.springframework.data.redis.serializer.StringRedisSerializer;

@Configuration
public class RedisConfig {
```

```
    @Bean
    public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory
connectionFactory) {
        RedisTemplate<String, Object> template = new RedisTemplate<>();
        template.setConnectionFactory(connectionFactory);

        // Use StringRedisSerializer for keys
        template.setKeySerializer(new StringRedisSerializer());

        // Use Jackson JSON serializer for values
        template.setValueSerializer(new GenericJackson2JsonRedisSerializer());

        // Also set serializers for hash keys and values
        template.setHashKeySerializer(new StringRedisSerializer());
        template.setHashValueSerializer(new GenericJackson2JsonRedisSerializer());

        template.afterPropertiesSet();
        return template;
    }
}
```

4. **Verify connectivity** with a simple test controller:

```
package com.example.redisdemo.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class TestController {

    @Autowired
    private RedisTemplate<String, Object> redisTemplate;

    @GetMapping("/test/{value}")
    public String testRedis(@PathVariable String value) {
        // Set a value in Redis
        redisTemplate.opsForValue().set("test-key", value);

        // Retrieve and return the value
        return "Stored in Redis: " + redisTemplate.opsForValue().get("test-key");
    }
}
```

Redis Data Structures and Spring Data Redis - Mapping Java Objects to Redis Data Structures

One of Redis's core strengths is its diverse data structures. Spring Data Redis provides specialized operations for each of these structures through the RedisTemplate's operation sets.

Creating a Domain Model

Let's start by defining a simple Product class that we'll map to various Redis data structures:

```
package com.example.redisdemo.model;

import java.io.Serializable;
import java.math.BigDecimal;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class Product implements Serializable {
    private static final long serialVersionUID = 1L;

    private String id;
    private String name;
    private String description;
    private BigDecimal price;
    private String category;
    private int stockQuantity;
}
```

Working with Redis String Operations

Redis Strings can store serialized objects, text, numbers, or binary data. They're the simplest data type but also the most versatile:

```
// Service class for Product operations using Redis Strings
@Service
public class ProductStringService {

    private final RedisTemplate<String, Object> redisTemplate;
    private final static String KEY_PREFIX = "product:";

    @Autowired
    public ProductStringService(RedisTemplate<String, Object> redisTemplate) {
        this.redisTemplate = redisTemplate;
    }

    // Save a product as a serialized string
    public void saveProduct(Product product) {
        redisTemplate.opsForValue().set(KEY_PREFIX + product.getId(), product);
    }
```

```
    // Get a product by its ID
    public Product getProduct(String id) {
        return (Product) redisTemplate.opsForValue().get(KEY_PREFIX + id);
    }

    // Delete a product
    public void deleteProduct(String id) {
        redisTemplate.delete(KEY_PREFIX + id);
    }

    // Set expiration time for a product
    public void setProductExpiry(String id, long timeout, TimeUnit unit) {
        redisTemplate.expire(KEY_PREFIX + id, timeout, unit);
    }

    // Atomic operations - increment stock quantity
    public Long incrementStock(String id, int amount) {
        return redisTemplate.opsForValue().increment(KEY_PREFIX + id + ":stock",
amount);
    }
}
```

Working with Redis Hash Operations

Hashes are maps between string fields and string values, making them perfect for representing objects with named fields:

```
@Service
public class ProductHashService {

    private final RedisTemplate<String, Object> redisTemplate;
    private final static String KEY_PREFIX = "product:hash:";

    @Autowired
    public ProductHashService(RedisTemplate<String, Object> redisTemplate) {
        this.redisTemplate = redisTemplate;
    }

    // Save product as a hash (field-value pairs)
    public void saveProduct(Product product) {
        String key = KEY_PREFIX + product.getId();
        Map<String, Object> fields = new HashMap<>();
        fields.put("id", product.getId());
        fields.put("name", product.getName());
        fields.put("description", product.getDescription());
        fields.put("price", product.getPrice().toString());
        fields.put("category", product.getCategory());
        fields.put("stockQuantity", product.getStockQuantity());

        redisTemplate.opsForHash().putAll(key, fields);
```

```
    }

    // Get product from hash representation
    public Product getProduct(String id) {
        String key = KEY_PREFIX + id;
        Map<Object, Object> fields = redisTemplate.opsForHash().entries(key);

        if (fields.isEmpty()) {
            return null;
        }

        Product product = new Product();
        product.setId((String) fields.get("id"));
        product.setName((String) fields.get("name"));
        product.setDescription((String) fields.get("description"));
        product.setPrice(new BigDecimal((String) fields.get("price")));
        product.setCategory((String) fields.get("category"));

product.setStockQuantity(Integer.parseInt(fields.get("stockQuantity").toString()))
;

        return product;
    }

    // Update a specific field in the hash
    public void updateProductField(String id, String field, Object value) {
        redisTemplate.opsForHash().put(KEY_PREFIX + id, field, value);
    }

    // Check if a product exists
    public boolean productExists(String id) {
        return redisTemplate.hasKey(KEY_PREFIX + id);
    }
}
```

Working with Redis List Operations

Lists in Redis are ordered collections of strings, useful for implementing queues, timelines, or recent items:

```
@Service
public class ProductListService {

    private final RedisTemplate<String, Object> redisTemplate;

    @Autowired
    public ProductListService(RedisTemplate<String, Object> redisTemplate) {
        this.redisTemplate = redisTemplate;
    }

    // Add product to a category list (from left)
    public void addProductToCategory(String category, Product product) {
```

```
        redisTemplate.opsForList().leftPush("category:" + category, product);
    }

    // Get recent products in a category (with range)
    public List<Product> getRecentProductsInCategory(String category, int start,
int end) {
        return redisTemplate.opsForList()
                .range("category:" + category, start, end)
                .stream()
                .map(item -> (Product) item)
                .collect(Collectors.toList());
    }

    // Add product to recently viewed list with size limit
    public void addToRecentlyViewed(String userId, Product product) {
        String key = "user:" + userId + ":recent";
        // Remove product if it already exists (to prevent duplicates)
        redisTemplate.opsForList().remove(key, 0, product);
        // Add to front of list
        redisTemplate.opsForList().leftPush(key, product);
        // Trim list to keep only 10 most recent items
        redisTemplate.opsForList().trim(key, 0, 9);
    }

    // Get user's recently viewed products
    public List<Product> getRecentlyViewed(String userId) {
        String key = "user:" + userId + ":recent";
        return redisTemplate.opsForList()
                .range(key, 0, -1)
                .stream()
                .map(item -> (Product) item)
                .collect(Collectors.toList());
    }
}
```

Working with Redis Set Operations

Sets are unordered collections of unique strings, perfect for tagging, relationships, or unique collections:

```
@Service
public class ProductSetService {

    private final RedisTemplate<String, Object> redisTemplate;

    @Autowired
    public ProductSetService(RedisTemplate<String, Object> redisTemplate) {
        this.redisTemplate = redisTemplate;
    }

    // Add product to a category set
    public void addProductToCategory(String category, String productId) {
```

```
        redisTemplate.opsForSet().add("category:set:" + category, productId);
    }

    // Get all products in a category
    public Set<String> getProductsInCategory(String category) {
        return redisTemplate.opsForSet()
                .members("category:set:" + category)
                .stream()
                .map(Object::toString)
                .collect(Collectors.toSet());
    }

    // Remove product from category
    public void removeProductFromCategory(String category, String productId) {
        redisTemplate.opsForSet().remove("category:set:" + category, productId);
    }

    // Find products in multiple categories (intersection)
    public Set<String> getProductsInAllCategories(String... categories) {
        if (categories.length == 0) {
            return Collections.emptySet();
        }

        String[] keys = Arrays.stream(categories)
                .map(cat -> "category:set:" + cat)
                .toArray(String[]::new);

        return redisTemplate.opsForSet()
                .intersect(Arrays.asList(keys))
                .stream()
                .map(Object::toString)
                .collect(Collectors.toSet());
    }

    // Find products in any of the categories (union)
    public Set<String> getProductsInAnyCategory(String... categories) {
        if (categories.length == 0) {
            return Collections.emptySet();
        }

        String[] keys = Arrays.stream(categories)
                .map(cat -> "category:set:" + cat)
                .toArray(String[]::new);

        return redisTemplate.opsForSet()
                .union(Arrays.asList(keys))
                .stream()
                .map(Object::toString)
                .collect(Collectors.toSet());
    }
}
```

Working with Redis Sorted Set Operations

Sorted Sets combine sets (unique elements) with a score for each element, enabling ordered collections:

```
@Service
public class ProductSortedSetService {

    private final RedisTemplate<String, Object> redisTemplate;

    @Autowired
    public ProductSortedSetService(RedisTemplate<String, Object> redisTemplate) {
        this.redisTemplate = redisTemplate;
    }

    // Add product to price index (sorted by price)
    public void indexProductByPrice(Product product) {
        redisTemplate.opsForZSet().add(
            "products:by-price",
            product.getId(),
            product.getPrice().doubleValue()
        );
    }

    // Get products sorted by price (low to high)
    public List<String> getProductsSortedByPrice(int offset, int count) {
        Set<Object> results = redisTemplate.opsForZSet()
                .range("products:by-price", offset, offset + count - 1);
        return results.stream()
                .map(Object::toString)
                .collect(Collectors.toList());
    }

    // Get products sorted by price (high to low)
    public List<String> getProductsSortedByPriceDesc(int offset, int count) {
        Set<Object> results = redisTemplate.opsForZSet()
                .reverseRange("products:by-price", offset, offset + count - 1);
        return results.stream()
                .map(Object::toString)
                .collect(Collectors.toList());
    }

    // Get products within a price range
    public List<String> getProductsInPriceRange(double minPrice, double maxPrice)
{
        Set<Object> results = redisTemplate.opsForZSet()
                .rangeByScore("products:by-price", minPrice, maxPrice);
        return results.stream()
                .map(Object::toString)
                .collect(Collectors.toList());
    }

    // Increment product views and maintain "most viewed" leaderboard
```

```
    public void incrementProductViews(String productId) {
        redisTemplate.opsForZSet().incrementScore("products:most-viewed",
productId, 1);
    }

    // Get most viewed products
    public List<String> getMostViewedProducts(int limit) {
        Set<Object> results = redisTemplate.opsForZSet()
                .reverseRange("products:most-viewed", 0, limit - 1);
        return results.stream()
                .map(Object::toString)
                .collect(Collectors.toList());
    }
}
```

RedisTemplate and StringRedisTemplate - Core Components for Redis Operations

RedisTemplate is the central component of Spring Data Redis, abstracting away the complexities of working with Redis while providing a rich API for all Redis operations.

Understanding RedisTemplate

RedisTemplate is a comprehensive class that provides:

1. **Type Safety**: Through generics `RedisTemplate<K, V>`
2. **Operation Sets**: Specialized methods for each Redis data structure
3. **Connection Handling**: Manages the underlying Redis connections
4. **Serialization**: Converts Java objects to/from Redis storage format
5. **Exception Translation**: Converts Redis-specific exceptions to Spring's consistent data access exceptions

RedisTemplate vs. StringRedisTemplate

`StringRedisTemplate` is a specialized version of `RedisTemplate` that's configured to use String keys and values, with `StringRedisSerializer` for both:

Feature	RedisTemplate	StringRedisTemplate
Default Key Serializer	JdkSerializationRedisSerializer	StringRedisSerializer
Default Value Serializer	JdkSerializationRedisSerializer	StringRedisSerializer
Use Case	Working with Java objects	Working with String values
Interoperability	Less compatible with other Redis clients	Highest compatibility with other Redis clients

When using `StringRedisTemplate`, you need to manually convert objects to/from strings (e.g., using JSON), but it offers better interoperability with other Redis clients and tools.

Operations Sets

RedisTemplate provides specialized operation interfaces for each Redis data structure:

```
// Value operations (String data type)
ValueOperations<K, V> opsForValue = redisTemplate.opsForValue();

// List operations
ListOperations<K, V> opsForList = redisTemplate.opsForList();

// Set operations
SetOperations<K, V> opsForSet = redisTemplate.opsForSet();

// ZSet operations (Sorted Sets)
ZSetOperations<K, V> opsForZSet = redisTemplate.opsForZSet();

// Hash operations
HashOperations<K, HK, HV> opsForHash = redisTemplate.opsForHash();

// Stream operations (Redis Streams - for event streaming)
StreamOperations<K, ?, ?> opsForStream = redisTemplate.opsForStream();

// Geo operations (for geospatial data)
GeoOperations<K, V> opsForGeo = redisTemplate.opsForGeo();

// HyperLogLog operations (for cardinality estimation)
HyperLogLogOperations<K, V> opsForHyperLogLog = redisTemplate.opsForHyperLogLog();
```

Key Operations

In addition to data structure-specific operations, RedisTemplate provides methods for key management:

```
// Check if a key exists
boolean exists = redisTemplate.hasKey("my-key");

// Delete keys
redisTemplate.delete("my-key");
redisTemplate.delete(Arrays.asList("key1", "key2"));

// Set expiration
redisTemplate.expire("my-key", 30, TimeUnit.SECONDS);

// Get expiration
long ttl = redisTemplate.getExpire("my-key");

// Rename a key
redisTemplate.rename("old-key", "new-key");

// Find keys matching a pattern
Set<K> keys = redisTemplate.keys("user:*");
```

```
// Move a key to a different database
redisTemplate.move("my-key", 2);  // Move to database #2
```

Configuring Serializers

RedisTemplate uses serializers to convert between Java objects and binary data. Proper serializer configuration is crucial for compatibility and performance:

```
@Bean
public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory
connectionFactory) {
    RedisTemplate<String, Object> template = new RedisTemplate<>();
    template.setConnectionFactory(connectionFactory);

    // Use Jackson for JSON serialization of values
    Jackson2JsonRedisSerializer<Object> jacksonSerializer =
            new Jackson2JsonRedisSerializer<>(Object.class);

    // Configure ObjectMapper for Jackson serializer
    ObjectMapper om = new ObjectMapper();
    om.setVisibility(PropertyAccessor.ALL, JsonAutoDetect.Visibility.ANY);
    om.activateDefaultTyping(
            om.getPolymorphicTypeValidator(),
            ObjectMapper.DefaultTyping.NON_FINAL,
            JsonTypeInfo.As.PROPERTY);
    jacksonSerializer.setObjectMapper(om);

    // String serializer for keys
    StringRedisSerializer stringSerializer = new StringRedisSerializer();

    // Configure serializers
    template.setKeySerializer(stringSerializer);
    template.setValueSerializer(jacksonSerializer);
    template.setHashKeySerializer(stringSerializer);
    template.setHashValueSerializer(jacksonSerializer);

    template.afterPropertiesSet();
    return template;
}
```

Available serializers include:

- **StringRedisSerializer**: For String values (UTF-8 encoded)
- **GenericToStringSerializer**: Converts objects to strings using a Converter
- **JdkSerializationRedisSerializer**: Uses Java serialization (default)
- **Jackson2JsonRedisSerializer**: Uses Jackson for JSON serialization
- **GenericJackson2JsonRedisSerializer**: Jackson-based with type information
- **OxmSerializer**: XML serialization using Spring OXM

Using RedisCallback and SessionCallback

For advanced operations or when you need direct access to the underlying Redis API, you can use callbacks:

```
// RedisCallback - low-level, gives access to binary Redis commands
Object result = redisTemplate.execute(new RedisCallback<Object>() {
    @Override
    public Object doInRedis(RedisConnection connection) throws DataAccessException
{
        // Use the native Redis connection
        connection.set("key".getBytes(), "value".getBytes());
        return connection.get("key".getBytes());
    }
});

// SessionCallback - for transaction or pipeline support
List<Object> txResults = redisTemplate.execute(new SessionCallback<List<Object>>()
{
    @Override
    public List<Object> execute(RedisOperations operations) throws
DataAccessException {
        operations.multi();  // Start transaction

        operations.opsForValue().set("key1", "value1");
        operations.opsForValue().set("key2", "value2");
        operations.opsForSet().add("set-key", "member1", "member2");

        // Execute transaction and return results
        return operations.exec();
    }
});
```

Caching with Spring Data Redis - Implementing Caching with Redis

Redis is frequently used as a caching solution due to its exceptional performance. Spring Boot simplifies Redis-based caching through its caching abstraction.

Setting Up Redis Caching in Spring Boot

1. **Add dependencies**:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

2. **Enable caching**:

```
@SpringBootApplication
@EnableCaching
public class RedisDemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(RedisDemoApplication.class, args);
    }
}
```

3. **Configure Redis as cache manager**:

```
@Configuration
public class CacheConfig {

    @Bean
    public RedisCacheManager cacheManager(RedisConnectionFactory
connectionFactory) {
        // Default cache configuration
        RedisCacheConfiguration cacheConfig = RedisCacheConfiguration
            .defaultCacheConfig()
            .entryTtl(Duration.ofMinutes(10))  // Set default TTL
            .serializeKeysWith(
                RedisSerializationContext.SerializationPair.fromSerializer(
                    new StringRedisSerializer()))
            .serializeValuesWith(
                RedisSerializationContext.SerializationPair.fromSerializer(
                    new GenericJackson2JsonRedisSerializer()))
            .disableCachingNullValues();

        // Create cache manager
        return RedisCacheManager.builder(connectionFactory)
            .cacheDefaults(cacheConfig)
            // Configure specific caches with different settings
            .withCacheConfiguration("products",
                RedisCacheConfiguration.defaultCacheConfig()
                    .entryTtl(Duration.ofMinutes(5)))
            .withCacheConfiguration("categories",
                RedisCacheConfiguration.defaultCacheConfig()
                    .entryTtl(Duration.ofHours(2)))
            .build();
    }
}
```

Using Spring Cache Annotations

Spring provides annotations for declarative caching:

```java
@Service
public class ProductServiceImpl implements ProductService {

    private final ProductRepository productRepository;

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

    // Cache the result of this method
    @Cacheable(value = "products", key = "#id")
    @Override
    public Product getProductById(String id) {
        // This will only execute if the product is not in cache
        System.out.println("Fetching product from database: " + id);
        return productRepository.findById(id)
                .orElseThrow(() -> new EntityNotFoundException("Product not
found"));
    }

    // Update cache when product is updated
    @CachePut(value = "products", key = "#product.id")
    @Override
    public Product updateProduct(Product product) {
        System.out.println("Updating product in database: " + product.getId());
        return productRepository.save(product);
    }

    // Remove from cache when product is deleted
    @CacheEvict(value = "products", key = "#id")
    @Override
    public void deleteProduct(String id) {
        System.out.println("Deleting product from database: " + id);
        productRepository.deleteById(id);
    }

    // Use condition to determine when to cache
    @Cacheable(value = "products", key = "#category",
               condition = "#category != 'perishable'")
    @Override
    public List<Product> getProductsByCategory(String category) {
        System.out.println("Fetching products by category: " + category);
        return productRepository.findByCategory(category);
    }

    // Clear all entries in a cache
    @CacheEvict(value = "products", allEntries = true)
    @Override
    public void clearProductCache() {
        System.out.println("Clearing product cache");
    }
```

```
    // Multiple cache operations in one method
    @Caching(
        evict = {
            @CacheEvict(value = "products", key = "#id"),
            @CacheEvict(value = "categories", key = "#product.category")
        }
    )
    @Override
    public void updateProductCategory(String id, String newCategory) {
        Product product = productRepository.findById(id).orElseThrow();
        product.setCategory(newCategory);
        productRepository.save(product);
    }
}
```

Implementing Custom Caching Logic

For more complex caching scenarios, you can directly use the CacheManager:

```
@Service
public class AdvancedCacheService {

    private final CacheManager cacheManager;
    private final ProductRepository productRepository;

    @Autowired
    public AdvancedCacheService(
            CacheManager cacheManager,
            ProductRepository productRepository) {
        this.cacheManager = cacheManager;
        this.productRepository = productRepository;
    }

    // Manually add an item to cache
    public void cacheProductManually(Product product) {
        Cache productsCache = cacheManager.getCache("products");
        if (productsCache != null) {
            productsCache.put(product.getId(), product);
        }
    }

    // Manually retrieve an item from cache
    public Product getProductFromCache(String id) {
        Cache productsCache = cacheManager.getCache("products");
        if (productsCache != null) {
            Cache.ValueWrapper wrapper = productsCache.get(id);
            if (wrapper != null) {
                return (Product) wrapper.get();
            }
        }
        // If not in cache, load from repository and cache it
```

```
        Product product = productRepository.findById(id).orElse(null);
        if (product != null) {
            cacheProductManually(product);
        }
        return product;
    }

    // Implement bulk cache operations
    public void cacheBulkProducts(List<Product> products) {
        Cache productsCache = cacheManager.getCache("products");
        if (productsCache != null) {
            products.forEach(product ->
                productsCache.put(product.getId(), product));
        }
    }

    // Implement cache with custom key generation
    public List<Product> getProductsInPriceRange(
            BigDecimal minPrice, BigDecimal maxPrice) {
        // Create a composite key
        String cacheKey = "price_range_" + minPrice + "_" + maxPrice;

        Cache productsCache = cacheManager.getCache("products");
        if (productsCache != null) {
            Cache.ValueWrapper wrapper = productsCache.get(cacheKey);
            if (wrapper != null) {
                return (List<Product>) wrapper.get();
            }
        }

        // Fetch from repository if not in cache
        List<Product> products = productRepository
                .findByPriceBetween(minPrice, maxPrice);

        // Cache the result
        if (productsCache != null) {
            productsCache.put(cacheKey, products);
        }

        return products;
    }
}
```

Cache Synchronization Strategies

When multiple instances of your application are running, keeping caches synchronized becomes important:

1. **Time-To-Live (TTL)**: Set reasonable expiration times for cached items

```
RedisCacheConfiguration.defaultCacheConfig()
    .entryTtl(Duration.ofMinutes(10));
```

2. **Cache eviction on updates**: Ensure cache entries are evicted when the underlying data changes

```
@CacheEvict(value = "products", key = "#product.id")
public Product updateProduct(Product product) {
    // Update in database
    return repository.save(product);
}
```

3. **Publish/Subscribe for cache invalidation**: Use Redis pub/sub to notify all application instances about cache changes

```
@Service
public class CacheInvalidationService {

    private final StringRedisTemplate redisTemplate;
    private final CacheManager cacheManager;

    @Autowired
    public CacheInvalidationService(
            StringRedisTemplate redisTemplate,
            CacheManager cacheManager) {
        this.redisTemplate = redisTemplate;
        this.cacheManager = cacheManager;

        // Subscribe to cache invalidation messages
        redisTemplate.listenToChannel("cache:invalidate", message -> {
            String[] parts = message.getMessage().split(":");
            if (parts.length == 2) {
                String cacheName = parts[0];
                String key = parts[1];

                Cache cache = cacheManager.getCache(cacheName);
                if (cache != null) {
                    cache.evict(key);
                }
            }
        });
    }

    // Publish cache invalidation message
    public void invalidateCache(String cacheName, String key) {
        // Evict from local cache
        Cache cache = cacheManager.getCache(cacheName);
        if (cache != null) {
            cache.evict(key);
        }

        // Publish message to invalidate cache in other instances
        redisTemplate.convertAndSend(
```

```
                "cache:invalidate",
                cacheName + ":" + key
            );
        }
    }
```

Pub/Sub Messaging with Spring Data Redis - Real-time Communication

Redis provides publish/subscribe messaging functionality that enables real-time communication between application components or instances. Spring Data Redis simplifies working with this feature.

Understanding Redis Pub/Sub

Redis Pub/Sub is a messaging pattern where:

- **Publishers** send messages to named channels without knowledge of subscribers
- **Subscribers** express interest in channels and receive messages without knowledge of publishers
- Messages are not persisted and are only delivered to subscribers active at the time of publication

This pattern is ideal for:

- Real-time notifications
- Chat applications
- System events
- Live updates
- Distributed cache invalidation

Implementing Redis Pub/Sub with Spring Data Redis

1. Setting up a Message Listener Container

```
@Configuration
public class RedisPubSubConfig {

    @Bean
    public RedisMessageListenerContainer messageListenerContainer(
            RedisConnectionFactory connectionFactory,
            MessageListenerAdapter orderListenerAdapter) {

        RedisMessageListenerContainer container = new
RedisMessageListenerContainer();
        container.setConnectionFactory(connectionFactory);

        // Register listeners for specific channels
        container.addMessageListener(
            orderListenerAdapter,
            new ChannelTopic("orders:new")
        );

        return container;
```

```
    }

    @Bean
    public MessageListenerAdapter orderListenerAdapter(OrderMessageListener
listener) {
        return new MessageListenerAdapter(listener, "onOrderMessage");
    }

    @Bean
    public OrderMessageListener orderMessageListener() {
        return new OrderMessageListener();
    }
}
```

2. Implementing a Message Listener

```
@Component
public class OrderMessageListener {

    private final ObjectMapper objectMapper;

    @Autowired
    public OrderMessageListener(ObjectMapper objectMapper) {
        this.objectMapper = objectMapper;
    }

    // Method called when a message is received
    public void onOrderMessage(String message) {
        try {
            // Parse the JSON message
            Order order = objectMapper.readValue(message, Order.class);

            // Process the order
            System.out.println("Received new order: " + order.getId());

            // Additional processing logic...

        } catch (Exception e) {
            System.err.println("Error processing order message: " +
e.getMessage());
        }
    }
}
```

3. Publishing Messages

```
@Service
public class OrderPublisherService {
```

```
    private final StringRedisTemplate redisTemplate;
    private final ObjectMapper objectMapper;

    @Autowired
    public OrderPublisherService(
            StringRedisTemplate redisTemplate,
            ObjectMapper objectMapper) {
        this.redisTemplate = redisTemplate;
        this.objectMapper = objectMapper;
    }

    public void publishNewOrder(Order order) {
        try {
            // Convert order to JSON string
            String message = objectMapper.writeValueAsString(order);

            // Publish to the "orders:new" channel
            redisTemplate.convertAndSend("orders:new", message);

            System.out.println("Published new order: " + order.getId());

        } catch (Exception e) {
            System.err.println("Error publishing order: " + e.getMessage());
        }
    }
}
```

Building a Real-time Notification System

Let's create a more comprehensive example - a notification system that delivers real-time updates to users:

```
// Notification model
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Notification {
    private String id;
    private String userId;
    private String type;
    private String message;
    private String data;
    private LocalDateTime timestamp;
    private boolean read;
}
```

Notification Publisher Service

```
@Service
public class NotificationService {

    private final StringRedisTemplate redisTemplate;
    private final ObjectMapper objectMapper;

    @Autowired
    public NotificationService(
            StringRedisTemplate redisTemplate,
            ObjectMapper objectMapper) {
        this.redisTemplate = redisTemplate;
        this.objectMapper = objectMapper;
    }

    // Send notification to a specific user
    public void sendUserNotification(String userId, Notification notification) {
        try {
            notification.setId(UUID.randomUUID().toString());
            notification.setUserId(userId);
            notification.setTimestamp(LocalDateTime.now());
            notification.setRead(false);

            // Serialize notification to JSON
            String message = objectMapper.writeValueAsString(notification);

            // Publish to user-specific channel
            redisTemplate.convertAndSend("user:" + userId + ":notifications",
message);

            // Also save to user's notification list for persistence
            redisTemplate.opsForList().leftPush(
                "user:" + userId + ":notification-list",
                message
            );

            // Trim the list to keep only the 100 most recent notifications
            redisTemplate.opsForList().trim(
                "user:" + userId + ":notification-list",
                0,
                99
            );

        } catch (Exception e) {
            throw new RuntimeException("Failed to send notification", e);
        }
    }

    // Broadcast notification to all users
    public void broadcastNotification(Notification notification) {
        try {
            notification.setId(UUID.randomUUID().toString());
            notification.setTimestamp(LocalDateTime.now());
            notification.setRead(false);
```

```
            // Serialize notification
            String message = objectMapper.writeValueAsString(notification);

            // Publish to broadcast channel
            redisTemplate.convertAndSend("notifications:broadcast", message);

        } catch (Exception e) {
            throw new RuntimeException("Failed to broadcast notification", e);
        }
    }

    // Send notification to users in a specific group
    public void sendGroupNotification(String groupId, Notification notification) {
        try {
            notification.setId(UUID.randomUUID().toString());
            notification.setTimestamp(LocalDateTime.now());
            notification.setRead(false);

            // Serialize notification
            String message = objectMapper.writeValueAsString(notification);

            // Publish to group channel
            redisTemplate.convertAndSend("group:" + groupId + ":notifications",
message);

        } catch (Exception e) {
            throw new RuntimeException("Failed to send group notification", e);
        }
    }
}
```

Notification Listener Service

```
@Service
public class NotificationListenerService {

    private final ObjectMapper objectMapper;
    private final Map<String, List<Consumer<Notification>>> userHandlers = new
ConcurrentHashMap<>();

    @Autowired
    public NotificationListenerService(
            RedisConnectionFactory connectionFactory,
            ObjectMapper objectMapper) {
        this.objectMapper = objectMapper;

        // Create and configure the message listener container
        RedisMessageListenerContainer container = new
RedisMessageListenerContainer();
        container.setConnectionFactory(connectionFactory);
```

```
        // Listen for broadcast notifications
        container.addMessageListener(
            (message, pattern) -> handleBroadcastNotification(new
String(message.getBody())),
            new ChannelTopic("notifications:broadcast")
        );

        // Start the container
        container.start();
    }

    // Subscribe a user to their notifications
    public void subscribeUser(String userId, Consumer<Notification> handler) {
        userHandlers.computeIfAbsent(userId, k -> new CopyOnWriteArrayList<>())
                    .add(handler);

        // Create a dedicated listener for this user
        RedisMessageListenerContainer container = new
RedisMessageListenerContainer();
        container.setConnectionFactory(

ApplicationContextHolder.getContext().getBean(RedisConnectionFactory.class)
        );

        // Add listener for user's notification channel
        container.addMessageListener(
            (message, pattern) -> {
                String notificationJson = new String(message.getBody());
                handleUserNotification(userId, notificationJson);
            },
            new ChannelTopic("user:" + userId + ":notifications")
        );

        container.start();
    }

    private void handleUserNotification(String userId, String notificationJson) {
        try {
            Notification notification = objectMapper.readValue(
                notificationJson,
                Notification.class
            );

            // Notify all handlers for this user
            List<Consumer<Notification>> handlers = userHandlers.get(userId);
            if (handlers != null) {
                for (Consumer<Notification> handler : handlers) {
                    handler.accept(notification);
                }
            }

        } catch (Exception e) {
            System.err.println("Error processing notification: " +
```

```
e.getMessage());
        }
    }

    private void handleBroadcastNotification(String notificationJson) {
        try {
            Notification notification = objectMapper.readValue(
                notificationJson,
                Notification.class
            );

            // Notify all user handlers about broadcast notification
            for (List<Consumer<Notification>> handlers : userHandlers.values()) {
                for (Consumer<Notification> handler : handlers) {
                    handler.accept(notification);
                }
            }

        } catch (Exception e) {
            System.err.println("Error processing broadcast notification: " +
e.getMessage());
        }
    }

    // Unsubscribe a user's handler
    public void unsubscribeUser(String userId, Consumer<Notification> handler) {
        List<Consumer<Notification>> handlers = userHandlers.get(userId);
        if (handlers != null) {
            handlers.remove(handler);
            if (handlers.isEmpty()) {
                userHandlers.remove(userId);
            }
        }
    }
}
```

Implementing a Chat Application

Redis Pub/Sub is particularly well-suited for chat applications. Here's a simplified implementation:

```
// Chat message model
@Data
@NoArgsConstructor
@AllArgsConstructor
public class ChatMessage {
    private String id;
    private String roomId;
    private String sender;
    private String content;
    private LocalDateTime timestamp;
}
```

```
// Chat service
@Service
public class ChatService {

    private final StringRedisTemplate redisTemplate;
    private final ObjectMapper objectMapper;

    @Autowired
    public ChatService(
            StringRedisTemplate redisTemplate,
            ObjectMapper objectMapper) {
        this.redisTemplate = redisTemplate;
        this.objectMapper = objectMapper;
    }

    // Send a message to a chat room
    public void sendMessage(ChatMessage message) {
        try {
            // Set metadata
            message.setId(UUID.randomUUID().toString());
            message.setTimestamp(LocalDateTime.now());

            // Serialize message
            String messageJson = objectMapper.writeValueAsString(message);

            // Publish to room channel
            redisTemplate.convertAndSend("chat:room:" + message.getRoomId(),
messageJson);

            // Store in room history
            redisTemplate.opsForList().leftPush(
                "chat:history:" + message.getRoomId(),
                messageJson
            );

            // Limit history size
            redisTemplate.opsForList().trim(
                "chat:history:" + message.getRoomId(),
                0,
                99  // Keep last 100 messages
            );

        } catch (Exception e) {
            throw new RuntimeException("Failed to send message", e);
        }
    }

    // Get chat history for a room
    public List<ChatMessage> getRoomHistory(String roomId, int limit) {
        List<String> messageJsonList = redisTemplate.opsForList()
                .range("chat:history:" + roomId, 0, limit - 1);

        List<ChatMessage> messages = new ArrayList<>();
```

```
        if (messageJsonList != null) {
            for (String messageJson : messageJsonList) {
                try {
                    messages.add(objectMapper.readValue(messageJson,
ChatMessage.class));
                } catch (Exception e) {
                    // Log error and continue with next message
                    System.err.println("Error parsing message: " +
e.getMessage());
                }
            }
        }

        return messages;
    }

    // Join a chat room
    public void joinRoom(String roomId, String username, Consumer<ChatMessage>
messageHandler) {
        // Create a message listener container
        RedisMessageListenerContainer container = new
RedisMessageListenerContainer();
        container.setConnectionFactory(

ApplicationContextHolder.getContext().getBean(RedisConnectionFactory.class)
        );

        // Add listener for room channel
        container.addMessageListener(
            (message, pattern) -> {
                try {
                    String messageJson = new String(message.getBody());
                    ChatMessage chatMessage = objectMapper.readValue(
                        messageJson,
                        ChatMessage.class
                    );

                    // Invoke the handler
                    messageHandler.accept(chatMessage);

                } catch (Exception e) {
                    System.err.println("Error processing chat message: " +
e.getMessage());
                }
            },
            new ChannelTopic("chat:room:" + roomId)
        );

        container.start();

        // Send a system message about user joining
        ChatMessage joinMessage = new ChatMessage();
        joinMessage.setRoomId(roomId);
```

```
            joinMessage.setSender("SYSTEM");
            joinMessage.setContent(username + " has joined the room");
            sendMessage(joinMessage);
        }
    }
```

Transactions in Redis with Spring Data Redis - Ensuring Data Consistency

Redis provides a transaction mechanism through MULTI/EXEC commands. Spring Data Redis simplifies working with Redis transactions.

Understanding Redis Transactions

Redis transactions allow the execution of a group of commands as a single isolated operation with two important guarantees:

1. **Command isolation**: All commands in a transaction are executed sequentially without other client commands interspersed
2. **Atomic execution**: Either all commands or none are processed

However, Redis transactions differ from traditional RDBMS transactions in important ways:

- There's no rollback capability - if a command fails, other commands are still executed
- Redis does not block other clients' commands during a transaction (optimistic locking is available with WATCH)

Implementing Redis Transactions with Spring Data Redis

Spring Data Redis provides two main approaches for working with transactions:

1. Using the @Transactional Annotation

```
@Service
@Transactional
public class TransactionalOrderService {

    private final StringRedisTemplate redisTemplate;

    @Autowired
    public TransactionalOrderService(StringRedisTemplate redisTemplate) {
        this.redisTemplate = redisTemplate;
    }

    @Transactional
    public void processOrder(Order order) {
        // Update inventory
        String inventoryKey = "inventory:" + order.getProductId();
        Long remainingStock = redisTemplate.opsForValue().decrement(
            inventoryKey,
            order.getQuantity()
```

```
        );

        // If we have stock, proceed with order processing
        if (remainingStock != null && remainingStock >= 0) {
            // Record the order
            String orderKey = "order:" + order.getId();
            redisTemplate.opsForHash().putAll(
                orderKey,
                Map.of(
                    "id", order.getId(),
                    "productId", order.getProductId(),
                    "userId", order.getUserId(),
                    "quantity", String.valueOf(order.getQuantity()),
                    "status", "confirmed",
                    "timestamp", LocalDateTime.now().toString()
                )
            );

            // Add to user's order list
            redisTemplate.opsForList().leftPush(
                "user:" + order.getUserId() + ":orders",
                order.getId()
            );

            // Add to product's order list
            redisTemplate.opsForList().leftPush(
                "product:" + order.getProductId() + ":orders",
                order.getId()
            );
        } else {
            // Not enough stock, restore inventory count
            redisTemplate.opsForValue().increment(
                inventoryKey,
                order.getQuantity()
            );

            throw new IllegalStateException("Insufficient inventory");
        }
    }
}
```

2. Using the SessionCallback Interface

For more control over transactions, particularly when you need to use WATCH for optimistic locking:

```
@Service
public class InventoryService {

    private final StringRedisTemplate redisTemplate;

    @Autowired
```

```
    public InventoryService(StringRedisTemplate redisTemplate) {
        this.redisTemplate = redisTemplate;
    }

    public boolean decrementInventory(String productId, int quantity) {
        String inventoryKey = "inventory:" + productId;

        // Using SessionCallback for transaction with WATCH
        return redisTemplate.execute(new SessionCallback<Boolean>() {
            @Override
            public Boolean execute(RedisOperations operations) throws
DataAccessException {
                // Watch the inventory key for changes
                operations.watch(inventoryKey);

                // Get current inventory
                String currentValue = (String)
operations.opsForValue().get(inventoryKey);
                int currentInventory = currentValue != null ?
                        Integer.parseInt(currentValue) : 0;

                // Check if we have enough stock
                if (currentInventory < quantity) {
                    // Not enough stock, discard transaction
                    operations.unwatch();
                    return false;
                }

                // Start transaction
                operations.multi();

                // Decrement inventory
                operations.opsForValue().set(
                    inventoryKey,
                    String.valueOf(currentInventory - quantity)
                );

                // Execute transaction (returns null if WATCH detected a change)
                return !operations.exec().isEmpty();
            }
        });
    }

    public boolean transferInventory(
            String sourceProductId,
            String targetProductId,
            int quantity) {

        String sourceKey = "inventory:" + sourceProductId;
        String targetKey = "inventory:" + targetProductId;

        return redisTemplate.execute(new SessionCallback<Boolean>() {
            @Override
            public Boolean execute(RedisOperations operations) throws
```

```
DataAccessException {
                try {
                    // Watch both keys
                    operations.watch(sourceKey);
                    operations.watch(targetKey);

                    // Get current values
                    String sourceValue = (String)
operations.opsForValue().get(sourceKey);
                    int sourceInventory = sourceValue != null ?
                            Integer.parseInt(sourceValue) : 0;

                    // Check if source has enough
                    if (sourceInventory < quantity) {
                        operations.unwatch();
                        return false;
                    }

                    // Get target inventory
                    String targetValue = (String)
operations.opsForValue().get(targetKey);
                    int targetInventory = targetValue != null ?
                            Integer.parseInt(targetValue) : 0;

                    // Start transaction
                    operations.multi();

                    // Update inventories
                    operations.opsForValue().set(
                        sourceKey,
                        String.valueOf(sourceInventory - quantity)
                    );
                    operations.opsForValue().set(
                        targetKey,
                        String.valueOf(targetInventory + quantity)
                    );

                    // Execute transaction
                    return !operations.exec().isEmpty();

                } catch (Exception e) {
                    throw new RuntimeException(
                        "Error executing inventory transfer", e
                    );
                }
            }
        });
    }
}
```

Advanced Transaction Patterns

Implementing a Distributed Lock with Redis

Redis can be used to implement distributed locks, ensuring that only one application instance processes a specific resource at a time:

```
@Service
public class RedisLockService {

    private final StringRedisTemplate redisTemplate;
    private final String lockPrefix = "lock:";

    @Autowired
    public RedisLockService(StringRedisTemplate redisTemplate) {
        this.redisTemplate = redisTemplate;
    }

    // Acquire a lock with a timeout
    public boolean acquireLock(String lockName, String clientId, long
timeoutSeconds) {
        String lockKey = lockPrefix + lockName;

        Boolean acquired = redisTemplate.opsForValue().setIfAbsent(
            lockKey,
            clientId,
            Duration.ofSeconds(timeoutSeconds)
        );

        return Boolean.TRUE.equals(acquired);
    }

    // Release a lock (only if we own it)
    public boolean releaseLock(String lockName, String clientId) {
        String lockKey = lockPrefix + lockName;

        // Use a Lua script to ensure atomicity
        String script =
            "if redis.call('get', KEYS[1]) == ARGV[1] then " +
            "  return redis.call('del', KEYS[1]) " +
            "else " +
            "  return 0 " +
            "end";

        Long result = redisTemplate.execute(
            new DefaultRedisScript<>(script, Long.class),
            Collections.singletonList(lockKey),
            clientId
        );

        return Long.valueOf(1).equals(result);
    }

    // Execute code with a lock
```

```
    public <T> T executeWithLock(
            String lockName,
            long timeoutSeconds,
            Supplier<T> supplier) {

        String clientId = UUID.randomUUID().toString();

        try {
            // Try to acquire the lock
            if (!acquireLock(lockName, clientId, timeoutSeconds)) {
                throw new IllegalStateException("Failed to acquire lock: " +
lockName);
            }

            // Execute the supplier function
            return supplier.get();

        } finally {
            // Release the lock
            releaseLock(lockName, clientId);
        }
    }
}
```

Implementing Rate Limiting with Redis

Redis can efficiently implement rate limiting to protect APIs and services:

```
@Service
public class RateLimiterService {

    private final StringRedisTemplate redisTemplate;

    @Autowired
    public RateLimiterService(StringRedisTemplate redisTemplate) {
        this.redisTemplate = redisTemplate;
    }

    // Simple rate limiter using Redis
    public boolean allowRequest(String key, int maxRequests, int windowSeconds) {
        String rateLimiterKey = "rate-limiter:" + key;

        return redisTemplate.execute(new SessionCallback<Boolean>() {
            @Override
            public Boolean execute(RedisOperations operations) throws
DataAccessException {
                // Start transaction
                operations.multi();

                // Increment the counter
                operations.opsForValue().increment(rateLimiterKey, 1);
```

```
                // Set expiry if it doesn't exist
                operations.expire(rateLimiterKey, windowSeconds,
TimeUnit.SECONDS);

                // Execute transaction
                List<Object> results = operations.exec();

                // First result is the current count
                Long currentCount = (Long) results.get(0);

                // Check if we're under the limit
                return currentCount <= maxRequests;
            }
        });
    }

    // Token bucket rate limiter
    public boolean allowRequestTokenBucket(
            String key,
            int capacity,
            int refillRate,
            int refillPeriodSeconds) {

        String bucketKey = "token-bucket:" + key;

        return redisTemplate.execute(new SessionCallback<Boolean>() {
            @Override
            public Boolean execute(RedisOperations operations) throws
DataAccessException {
                // Use Lua script for atomic operation
                String script =
                    "local bucket = redis.call('hgetall', KEYS[1]) " +
                    "local now = tonumber(ARGV[1]) " +
                    "local capacity = tonumber(ARGV[2]) " +
                    "local refillRate = tonumber(ARGV[3]) " +
                    "local refillPeriod = tonumber(ARGV[4]) " +

                    "local tokens = 0 " +
                    "local lastRefill = now " +

                    "if next(bucket) ~= nil then " +
                    "  tokens = tonumber(bucket['tokens']) " +
                    "  lastRefill = tonumber(bucket['lastRefill']) " +
                    "else " +
                    "  tokens = capacity " +
                    "end " +

                    // Calculate token refill
                    "local elapsedTime = now - lastRefill " +
                    "if elapsedTime > 0 then " +
                    "  local newTokens = math.floor(elapsedTime / refillPeriod *
refillRate) " +
                    "  if newTokens > 0 then " +
```

```
                    "    tokens = math.min(capacity, tokens + newTokens) " +
                    "    lastRefill = now " +
                    "  end " +
                    "end " +

                    // Try to consume a token
                    "if tokens >= 1 then " +
                    "  tokens = tokens - 1 " +
                    "  redis.call('hmset', KEYS[1], 'tokens', tokens,
'lastRefill', lastRefill) " +
                    "  redis.call('expire', KEYS[1], refillPeriod * 2) " +
                    "  return 1 " +
                    "else " +
                    "  redis.call('hmset', KEYS[1], 'tokens', tokens,
'lastRefill', lastRefill) " +
                    "  redis.call('expire', KEYS[1], refillPeriod * 2) " +
                    "  return 0 " +
                    "end";

                Long result = (Long) operations.execute(
                    new DefaultRedisScript<>(script, Long.class),
                    Collections.singletonList(bucketKey),
                    String.valueOf(System.currentTimeMillis() / 1000),
                    String.valueOf(capacity),
                    String.valueOf(refillRate),
                    String.valueOf(refillPeriodSeconds)
                );

                return Long.valueOf(1).equals(result);
            }
        });
    }
}
```

The example, provided is a solid foundation for a product management system using Spring Data Redis, but it does need a few more components to be a complete, runnable project.

Full Project:- Product Management System

```
// File:
src/main/java/com/example/productmanagement/ProductManagementApplication.java
package com.example.productmanagement;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class ProductManagementApplication {

    public static void main(String[] args) {
```

```
        SpringApplication.run(ProductManagementApplication.class, args);
    }
}

// File: src/main/java/com/example/productmanagement/model/Product.java
package com.example.productmanagement.model;

import org.springframework.data.annotation.Id;
import org.springframework.data.redis.core.RedisHash;
import org.springframework.data.redis.core.index.Indexed;

import java.io.Serializable;
import java.math.BigDecimal;
import java.time.LocalDateTime;

@RedisHash("product")  // Defines the key prefix in Redis
public class Product implements Serializable {

    @Id  // Marks the field as the identifier
    private String id;

    @Indexed  // Creates a secondary index for faster lookups
    private String sku;

    private String name;
    private String description;
    private BigDecimal price;
    private Integer stockQuantity;
    private String category;
    private LocalDateTime createdAt;
    private LocalDateTime updatedAt;

    // Default constructor required for Redis serialization
    public Product() {
        this.createdAt = LocalDateTime.now();
        this.updatedAt = LocalDateTime.now();
    }

    // Constructor with essential fields
    public Product(String sku, String name, BigDecimal price, Integer
stockQuantity, String category) {
        this();
        this.sku = sku;
        this.name = name;
        this.price = price;
        this.stockQuantity = stockQuantity;
        this.category = category;
    }

    // Getters and setters
    public String getId() {
        return id;
    }
```

```
    public void setId(String id) {
        this.id = id;
    }

    public String getSku() {
        return sku;
    }

    public void setSku(String sku) {
        this.sku = sku;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public BigDecimal getPrice() {
        return price;
    }

    public void setPrice(BigDecimal price) {
        this.price = price;
    }

    public Integer getStockQuantity() {
        return stockQuantity;
    }

    public void setStockQuantity(Integer stockQuantity) {
        this.stockQuantity = stockQuantity;
    }

    public String getCategory() {
        return category;
    }

    public void setCategory(String category) {
        this.category = category;
    }

    public LocalDateTime getCreatedAt() {
        return createdAt;
```

```
    }

    public void setCreatedAt(LocalDateTime createdAt) {
        this.createdAt = createdAt;
    }

    public LocalDateTime getUpdatedAt() {
        return updatedAt;
    }

    public void setUpdatedAt(LocalDateTime updatedAt) {
        this.updatedAt = updatedAt;
    }

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

// File:
src/main/java/com/example/productmanagement/repository/ProductRepository.java
package com.example.productmanagement.repository;

import com.example.productmanagement.model.Product;
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;

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

@Repository
public interface ProductRepository extends CrudRepository<Product, String> {

    // Spring Data automatically implements this method
    // It will use the secondary index created by @Indexed annotation
    Optional<Product> findBySku(String sku);

    // Find products by category
    List<Product> findByCategory(String category);

    // Find products with stock below threshold
    List<Product> findByStockQuantityLessThan(Integer threshold);
}

// File: src/main/java/com/example/productmanagement/service/ProductService.java
package com.example.productmanagement.service;
```

```
import com.example.productmanagement.model.Product;
import com.example.productmanagement.repository.ProductRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

@Service
public class ProductService {

    private final ProductRepository productRepository;

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

    public Product createProduct(Product product) {
        // Save the product to Redis
        return productRepository.save(product);
    }

    public Optional<Product> getProductById(String id) {
        // Retrieve a product by its ID
        return productRepository.findById(id);
    }

    public Optional<Product> getProductBySku(String sku) {
        // Retrieve a product by its SKU
        return productRepository.findBySku(sku);
    }

    public List<Product> getAllProducts() {
        // Get all products from Redis
        List<Product> products = new ArrayList<>();
        productRepository.findAll().forEach(products::add);
        return products;
    }

    public List<Product> getProductsByCategory(String category) {
        // Get products filtered by category
        return productRepository.findByCategory(category);
    }

    public List<Product> getLowStockProducts(int threshold) {
        // Get products with stock below the specified threshold
        return productRepository.findByStockQuantityLessThan(threshold);
    }

    public Product updateProduct(String id, Product productDetails) {
```

```
        // Update an existing product
        return productRepository.findById(id)
                .map(existingProduct -> {
                    if (productDetails.getName() != null) {
                        existingProduct.setName(productDetails.getName());
                    }
                    if (productDetails.getDescription() != null) {

existingProduct.setDescription(productDetails.getDescription());
                    }
                    if (productDetails.getPrice() != null) {
                        existingProduct.setPrice(productDetails.getPrice());
                    }
                    if (productDetails.getStockQuantity() != null) {

existingProduct.setStockQuantity(productDetails.getStockQuantity());
                    }
                    if (productDetails.getCategory() != null) {
                        existingProduct.setCategory(productDetails.getCategory());
                    }

                    existingProduct.setUpdatedAt(LocalDateTime.now());
                    return productRepository.save(existingProduct);
                })
                .orElseThrow(() -> new RuntimeException("Product not found with
id: " + id));
    }

    public boolean deleteProduct(String id) {
        // Delete a product by its ID
        if (productRepository.existsById(id)) {
            productRepository.deleteById(id);
            return true;
        }
        return false;
    }
}

// File:
src/main/java/com/example/productmanagement/controller/ProductController.java
package com.example.productmanagement.controller;

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

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

```
    private final ProductService productService;

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

    @PostMapping
    public ResponseEntity<Product> createProduct(@RequestBody Product product) {
        Product createdProduct = productService.createProduct(product);
        return new ResponseEntity<>(createdProduct, HttpStatus.CREATED);
    }

    @GetMapping("/{id}")
    public ResponseEntity<Product> getProductById(@PathVariable String id) {
        return productService.getProductById(id)
                .map(product -> new ResponseEntity<>(product, HttpStatus.OK))
                .orElse(new ResponseEntity<>(HttpStatus.NOT_FOUND));
    }

    @GetMapping("/sku/{sku}")
    public ResponseEntity<Product> getProductBySku(@PathVariable String sku) {
        return productService.getProductBySku(sku)
                .map(product -> new ResponseEntity<>(product, HttpStatus.OK))
                .orElse(new ResponseEntity<>(HttpStatus.NOT_FOUND));
    }

    @GetMapping
    public ResponseEntity<List<Product>> getAllProducts() {
        List<Product> products = productService.getAllProducts();
        return new ResponseEntity<>(products, HttpStatus.OK);
    }

    @GetMapping("/category/{category}")
    public ResponseEntity<List<Product>> getProductsByCategory(@PathVariable
String category) {
        List<Product> products = productService.getProductsByCategory(category);
        return new ResponseEntity<>(products, HttpStatus.OK);
    }

    @GetMapping("/low-stock/{threshold}")
    public ResponseEntity<List<Product>> getLowStockProducts(@PathVariable int
threshold) {
        List<Product> products = productService.getLowStockProducts(threshold);
        return new ResponseEntity<>(products, HttpStatus.OK);
    }

    @PutMapping("/{id}")
    public ResponseEntity<Product> updateProduct(@PathVariable String id,
@RequestBody Product productDetails) {
        try {
            Product updatedProduct = productService.updateProduct(id,
productDetails);
```

```
            return new ResponseEntity<>(updatedProduct, HttpStatus.OK);
        } catch (RuntimeException e) {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteProduct(@PathVariable String id) {
        boolean deleted = productService.deleteProduct(id);
        return deleted ? new ResponseEntity<>(HttpStatus.NO_CONTENT) : new
ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
}

// File: src/main/java/com/example/productmanagement/config/RedisConfig.java
package com.example.productmanagement.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.data.redis.connection.lettuce.LettuceConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;
import
org.springframework.data.redis.repository.configuration.EnableRedisRepositories;
import
org.springframework.data.redis.serializer.GenericJackson2JsonRedisSerializer;
import org.springframework.data.redis.serializer.StringRedisSerializer;

@Configuration
@EnableRedisRepositories(basePackages =
"com.example.productmanagement.repository")
public class RedisConfig {

    @Bean
    public RedisConnectionFactory redisConnectionFactory() {
        // Using Lettuce, a modern Redis client
        return new LettuceConnectionFactory("localhost", 6379);
    }

    @Bean
    public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory
connectionFactory) {
        // Configure Redis template with appropriate serializers
        RedisTemplate<String, Object> template = new RedisTemplate<>();
        template.setConnectionFactory(connectionFactory);

        // Use String serializer for keys
        template.setKeySerializer(new StringRedisSerializer());

        // Use JSON serializer for values
        template.setValueSerializer(new GenericJackson2JsonRedisSerializer());

        // Use the same serializers for hash operations
        template.setHashKeySerializer(new StringRedisSerializer());
```

```
        template.setHashValueSerializer(new GenericJackson2JsonRedisSerializer());

        template.afterPropertiesSet();
        return template;
    }
}

// File:
src/main/java/com/example/productmanagement/exception/ProductNotFoundException.jav
a
package com.example.productmanagement.exception;

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

@ResponseStatus(HttpStatus.NOT_FOUND)
public class ProductNotFoundException extends RuntimeException {

    public ProductNotFoundException(String id) {
        super("Product not found with id: " + id);
    }

    public ProductNotFoundException(String message, Throwable cause) {
        super(message, cause);
    }
}

// File:
src/main/java/com/example/productmanagement/exception/GlobalExceptionHandler.java
package com.example.productmanagement.exception;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.context.request.WebRequest;

import java.time.LocalDateTime;
import java.util.LinkedHashMap;
import java.util.Map;

@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ProductNotFoundException.class)
    public ResponseEntity<Object> handleProductNotFoundException(
            ProductNotFoundException ex, WebRequest request) {

        Map<String, Object> body = new LinkedHashMap<>();
        body.put("timestamp", LocalDateTime.now());
        body.put("message", ex.getMessage());

        return new ResponseEntity<>(body, HttpStatus.NOT_FOUND);
    }
```

```
    @ExceptionHandler(Exception.class)
    public ResponseEntity<Object> handleGlobalException(
            Exception ex, WebRequest request) {

        Map<String, Object> body = new LinkedHashMap<>();
        body.put("timestamp", LocalDateTime.now());
        body.put("message", "An unexpected error occurred");

        return new ResponseEntity<>(body, HttpStatus.INTERNAL_SERVER_ERROR);
    }
}

// File: src/main/java/com/example/productmanagement/util/DataLoader.java
package com.example.productmanagement.util;

import com.example.productmanagement.model.Product;
import com.example.productmanagement.repository.ProductRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

import java.math.BigDecimal;

@Component
public class DataLoader implements CommandLineRunner {

    private final ProductRepository productRepository;

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

    @Override
    public void run(String... args) {
        // Load sample data if no products exist
        if (productRepository.count() == 0) {
            loadSampleProducts();
        }
    }

    private void loadSampleProducts() {
        // Create sample product data
        Product laptop = new Product(
                "TECH1001",
                "Premium Laptop",
                new BigDecimal("1299.99"),
                50,
                "Electronics"
        );
        laptop.setDescription("High-performance laptop with 16GB RAM and 512GB
SSD");
```

```
        Product smartphone = new Product(
                "TECH1002",
                "Smartphone Pro",
                new BigDecimal("899.99"),
                100,
                "Electronics"
        );
        smartphone.setDescription("Latest smartphone with advanced camera
system");

        Product headphones = new Product(
                "TECH1003",
                "Wireless Headphones",
                new BigDecimal("199.99"),
                200,
                "Electronics"
        );
        headphones.setDescription("Noise-cancelling wireless headphones with 30-
hour battery life");

        Product desk = new Product(
                "FURN2001",
                "Standing Desk",
                new BigDecimal("349.99"),
                30,
                "Furniture"
        );
        desk.setDescription("Adjustable height standing desk for home or office");

        Product chair = new Product(
                "FURN2002",
                "Ergonomic Chair",
                new BigDecimal("249.99"),
                45,
                "Furniture"
        );
        chair.setDescription("Comfortable office chair with lumbar support");

        // Save sample products to Redis
        productRepository.save(laptop);
        productRepository.save(smartphone);
        productRepository.save(headphones);
        productRepository.save(desk);
        productRepository.save(chair);
    }
}

// File: src/main/resources/application.properties
# Redis Configuration
spring.redis.host=localhost
spring.redis.port=6379
spring.redis.timeout=2000

# Server Configuration
```

```
server.port=8080

# Logging Configuration
logging.level.org.springframework.data.redis=DEBUG
logging.level.com.example.productmanagement=INFO

# File:
src/test/java/com/example/productmanagement/ProductManagementApplicationTests.java
package com.example.productmanagement;

import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
class ProductManagementApplicationTests {

    @Test
    void contextLoads() {
    }
}

// File:
src/test/java/com/example/productmanagement/service/ProductServiceTest.java
package com.example.productmanagement.service;

import com.example.productmanagement.model.Product;
import com.example.productmanagement.repository.ProductRepository;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;

import java.math.BigDecimal;
import java.util.Arrays;
import java.util.List;
import java.util.Optional;

import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.ArgumentMatchers.any;
import static org.mockito.Mockito.*;

class ProductServiceTest {

    @Mock
    private ProductRepository productRepository;

    @InjectMocks
    private ProductService productService;

    private Product testProduct;

    @BeforeEach
    void setUp() {
```

```
        MockitoAnnotations.openMocks(this);

        testProduct = new Product(
            "TEST001",
            "Test Product",
            new BigDecimal("19.99"),
            10,
            "Test"
        );
        testProduct.setId("1");
    }

    @Test
    void createProduct() {
        when(productRepository.save(any(Product.class))).thenReturn(testProduct);

        Product created = productService.createProduct(testProduct);

        assertNotNull(created);
        assertEquals("1", created.getId());
        assertEquals("TEST001", created.getSku());
        verify(productRepository, times(1)).save(any(Product.class));
    }

    @Test
    void getProductById() {

when(productRepository.findById("1")).thenReturn(Optional.of(testProduct));

        Optional<Product> found = productService.getProductById("1");

        assertTrue(found.isPresent());
        assertEquals("TEST001", found.get().getSku());
        verify(productRepository, times(1)).findById("1");
    }

    @Test
    void getProductBySku() {

when(productRepository.findBySku("TEST001")).thenReturn(Optional.of(testProduct));

        Optional<Product> found = productService.getProductBySku("TEST001");

        assertTrue(found.isPresent());
        assertEquals("1", found.get().getId());
        verify(productRepository, times(1)).findBySku("TEST001");
    }

    @Test
    void getAllProducts() {
        when(productRepository.findAll()).thenReturn(Arrays.asList(testProduct));

        List<Product> products = productService.getAllProducts();
```

```
        assertNotNull(products);
        assertEquals(1, products.size());
        verify(productRepository, times(1)).findAll();
    }
}

// File: pom.xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.2.4</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.example</groupId>
    <artifactId>product-management</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>product-management</name>
    <description>Product Management System with Spring Data Redis</description>

    <properties>
        <java.version>17</java.version>
    </properties>

    <dependencies>
        <!-- Spring Boot Starters -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-redis</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-validation</artifactId>
        </dependency>

        <!-- Redis Client -->
        <dependency>
            <groupId>io.lettuce</groupId>
            <artifactId>lettuce-core</artifactId>
        </dependency>

        <!-- Developer Tools -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
```

```
            <scope>runtime</scope>
            <optional>true</optional>
        </dependency>

        <!-- Testing -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>

        <!-- Documentation -->
        <dependency>
            <groupId>org.springdoc</groupId>
            <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
            <version>2.3.0</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

// File: README.md

```
# Product Management System with Spring Data Redis

A comprehensive product management system demonstrating the use of Redis as a key-value store with Spring Data Redis.

## Features

- Complete CRUD operations for products
- Secondary indexes for efficient lookups by SKU and category
- RESTful API with proper HTTP status codes
- Sample data loader for demonstration
- Exception handling
- Unit tests

## Getting Started

1. Ensure Redis is running on your local machine (default port 6379)

2. Clone the repository
```

git clone https://github.com/yourusername/product-management.git cd product-management

```
3. Build and run the application
```

mvn spring-boot:run

```
4. The application will be available at `http://localhost:8080`
- Swagger UI documentation available at `http://localhost:8080/swagger-ui.html`

## API Endpoints

- Create a product: `POST /api/products`
- Get all products: `GET /api/products`
- Get product by ID: `GET /api/products/{id}`
- Get product by SKU: `GET /api/products/sku/{sku}`
- Get products by category: `GET /api/products/category/{category}`
- Get low stock products: `GET /api/products/low-stock/{threshold}`
- Update a product: `PUT /api/products/{id}`
- Delete a product: `DELETE /api/products/{id}`

## How It Works

This application demonstrates Spring Data Redis for key-value storage:

1. Product data is stored in Redis as hashes
2. Spring Data Redis creates appropriate indices for efficient lookups
3. The Repository pattern abstracts away Redis-specific operations
4. Spring Boot auto-configures the Redis connection

## Redis Data Structure

Products are stored in Redis with the following structure:

- Key: `product:{id}`
- Value: A hash containing all product properties
- Secondary indices for SKU and category lookups

## Testing

Run the tests with:

mvn test
```

1. The Main Application Class

I've added the `ProductManagementApplication` class with the `@SpringBootApplication` annotation. This serves as the entry point for the application with the `main()` method that launches the Spring context.

2. Exception Handling

I've created a proper exception handling mechanism with:

- A custom `ProductNotFoundException` class
- A `GlobalExceptionHandler` with `@ControllerAdvice` to centralize error handling
- Clean response formatting for API errors

3. Sample Data Loader

I've added a `DataLoader` class that implements `CommandLineRunner` to load sample product data when the application starts. This gives users something to explore right away.

4. Enhanced Configuration

The `application.properties` file now includes:

- More comprehensive Redis configuration options
- Logging settings to help with troubleshooting
- Server configuration

5. Unit Tests

I've added test classes:

- A basic application context test
- A more comprehensive `ProductServiceTest` with JUnit and Mockito

6. Project Build File

I've included a complete `pom.xml` with:

- All necessary dependencies
- Spring Boot parent definition
- Java version specification
- Build plugins
- Documentation tools (SpringDoc OpenAPI)

7. Documentation

Added a `README.md` file that explains:

- How to install and run the application
- Available API endpoints
- How the Redis integration works
- Testing instructions

Chapter 6: Advanced Data Mapping and Relationships in Spring Data JPA

Advanced Entity Mappings

@MappedSuperclass for Inheritance and Code Reuse

The `@MappedSuperclass` annotation allows you to define a class with common attributes that can be inherited by multiple entity classes, without becoming an entity itself. This promotes code reuse and establishes a clean inheritance hierarchy.

When you use `@MappedSuperclass`, the annotated class doesn't generate a table in the database. Instead, its fields are included in the tables of the classes that extend it.

Let's look at a common example—a base class with audit fields:

```
@MappedSuperclass
public abstract class BaseEntity {

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

    @Column(name = "created_date", nullable = false, updatable = false)
    @CreatedDate
    private LocalDateTime createdDate;

    @Column(name = "last_modified_date")
    @LastModifiedDate
    private LocalDateTime lastModifiedDate;

    @Column(name = "created_by", length = 50)
    @CreatedBy
    private String createdBy;

    @Column(name = "last_modified_by", length = 50)
    @LastModifiedBy
    private String lastModifiedBy;

    // Getters and setters
}
```

Now, any entity can extend this base class to inherit these common fields:

```
@Entity
@Table(name = "products")
public class Product extends BaseEntity {
```

```
    @Column(name = "name", nullable = false)
    private String name;

    @Column(name = "price")
    private BigDecimal price;

    // Product-specific fields, getters, and setters
}
```

When you use `@MappedSuperclass`, the `Product` table will include columns for all the fields defined in `BaseEntity` plus its own fields. This approach lets you maintain a clean codebase with common attributes defined in one place.

@Inheritance and Inheritance Strategies

The `@Inheritance` annotation lets you map an inheritance hierarchy of classes to database tables. JPA supports three inheritance strategies:

1. SINGLE_TABLE Strategy

With `SINGLE_TABLE`, all classes in the hierarchy are mapped to a single table. A discriminator column is used to identify which concrete class each row represents.

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "payment_type", discriminatorType =
DiscriminatorType.STRING)
public abstract class Payment {

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

    private BigDecimal amount;

    // Getters and setters
}

@Entity
@DiscriminatorValue("CREDIT_CARD")
public class CreditCardPayment extends Payment {

    private String cardNumber;
    private String cardType;
    private String expiryMonth;
    private String expiryYear;

    // Getters and setters
}

@Entity
```

```
@DiscriminatorValue("BANK_TRANSFER")
public class BankTransferPayment extends Payment {

    private String bankName;
    private String accountNumber;
    private String routingNumber;

    // Getters and setters
}
```

Advantages of SINGLE_TABLE:

- Best performance (no joins needed)
- Simple queries
- Polymorphic relationships are easy to handle

Disadvantages:

- Many nullable columns (since not all subclasses will use all columns)
- Table can become wide and unwieldy with many subclasses

2. JOINED Strategy

With `JOINED`, each class in the hierarchy gets its own table. The parent class table contains common fields, and child class tables contain only their specific fields plus a foreign key to the parent table.

```
@Entity
@Table(name = "vehicles")
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Vehicle {

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

    private String manufacturer;
    private String model;
    private Integer year;

    // Getters and setters
}

@Entity
@Table(name = "cars")
public class Car extends Vehicle {

    private Integer numberOfDoors;
    private String fuelType;
    private Double engineSize;

    // Getters and setters
```

```
}

@Entity
@Table(name = "motorcycles")
public class Motorcycle extends Vehicle {

    private String engineType;
    private Double engineCapacity;

    // Getters and setters
}
```

Advantages of JOINED:

- Each table has only the columns it needs (no nulls)
- Database normalization principles are respected
- Suitable for complex inheritance hierarchies

Disadvantages:

- Requires joins for most queries, which impacts performance
- More complex queries for polymorphic operations

3. TABLE_PER_CLASS Strategy

With `TABLE_PER_CLASS`, each concrete class gets its own complete table with all fields from the hierarchy. No table is created for abstract classes.

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Person {

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

    private String name;
    private LocalDate dateOfBirth;

    // Getters and setters
}

@Entity
@Table(name = "employees")
public class Employee extends Person {

    private String employeeId;
    private String department;
    private BigDecimal salary;

    // Getters and setters
```

```
}

@Entity
@Table(name = "customers")
public class Customer extends Person {

    private String customerNumber;
    private Integer loyaltyPoints;

    // Getters and setters
}
```

Advantages of TABLE_PER_CLASS:

- Each table is complete and independent (no joins needed for specific class queries)
- No nullable columns

Disadvantages:

- Duplicate columns across tables
- Complex polymorphic queries (requires UNION operations)
- Not as widely supported by all JPA implementations

@SecondaryTable for Mapping to Multiple Tables

`@SecondaryTable` allows you to map a single entity to multiple database tables. This can be useful when dealing with legacy databases or when you want to organize data logically without changing your object model.

```
@Entity
@Table(name = "users")
@SecondaryTable(name = "user_details",
                pkJoinColumns = @PrimaryKeyJoinColumn(name = "user_id"))
@SecondaryTable(name = "user_preferences",
                pkJoinColumns = @PrimaryKeyJoinColumn(name = "user_id"))
public class User {

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

    @Column(name = "username")
    private String username;

    @Column(name = "email")
    private String email;

    // Fields in the secondary table "user_details"
    @Column(name = "first_name", table = "user_details")
    private String firstName;
```

```
    @Column(name = "last_name", table = "user_details")
    private String lastName;

    @Column(name = "phone_number", table = "user_details")
    private String phoneNumber;

    // Fields in the secondary table "user_preferences"
    @Column(name = "theme", table = "user_preferences")
    private String theme;

    @Column(name = "notifications_enabled", table = "user_preferences")
    private Boolean notificationsEnabled;

    // Getters and setters
}
```

In this example, a single `User` entity is mapped to three tables:

- `users` (primary table) for core user information
- `user_details` for personal information
- `user_preferences` for user settings

When you perform CRUD operations on a `User` entity, Hibernate will automatically handle the operations across all three tables. This provides a clean object model while accommodating complex database structures.

Composite Keys and Embeddable IDs

Sometimes entities require composite primary keys (keys made up of multiple columns). JPA offers two approaches for handling composite keys:

1. @EmbeddedId Approach

With `@EmbeddedId`, you create a separate class for the composite key and embed it in the entity.

```
// Composite key class
@Embeddable
public class OrderItemId implements Serializable {

    @Column(name = "order_id")
    private Long orderId;

    @Column(name = "product_id")
    private Long productId;

    // Equals, hashCode, and constructors are required
    @Override
    public boolean equals(Object o) {
        // Implementation
    }

    @Override
```

```
    public int hashCode() {
        // Implementation
    }

    // Getters and setters
}

// Entity using the composite key
@Entity
@Table(name = "order_items")
public class OrderItem {

    @EmbeddedId
    private OrderItemId id;

    private Integer quantity;

    private BigDecimal price;

    // Many-to-One associations that map to parts of the composite key
    @ManyToOne
    @MapsId("orderId") // Maps to the orderId field in OrderItemId
    private Order order;

    @ManyToOne
    @MapsId("productId") // Maps to the productId field in OrderItemId
    private Product product;

    // Getters and setters
}
```

The `@MapsId` annotation is particularly important here as it tells JPA that a field in the embedded ID corresponds to the primary key of the associated entity. This creates a clean mapping between the composite key and the related entities.

2. @IdClass Approach

With `@IdClass`, you still create a separate class for the composite key, but the fields are repeated in both the ID class and the entity.

```
// Composite key class
public class BookEditionId implements Serializable {

    private String isbn;
    private Integer edition;

    // Equals, hashCode, and constructors are required
    @Override
    public boolean equals(Object o) {
        // Implementation
    }
```

```
    @Override
    public int hashCode() {
        // Implementation
    }

    // Getters and setters
}

// Entity using the composite key
@Entity
@Table(name = "book_editions")
@IdClass(BookEditionId.class)
public class BookEdition {

    @Id
    @Column(name = "isbn")
    private String isbn;

    @Id
    @Column(name = "edition")
    private Integer edition;

    @Column(name = "publication_year")
    private Integer publicationYear;

    @Column(name = "pages")
    private Integer pages;

    // Getters and setters
}
```

The key difference in the `@IdClass` approach is that you repeat the key fields in both the entity and the ID class. The field names must match between the two classes.

Choosing Between @EmbeddedId and @IdClass

Both approaches have their advantages:

- `@EmbeddedId` provides a cleaner encapsulation as the key is a single field in the entity
- `@IdClass` can be more straightforward and simpler to understand initially
- `@EmbeddedId` works better with `@MapsId` for managing foreign key relationships
- `@IdClass` may be easier to use with JPQL queries

In practice, `@EmbeddedId` is often preferred for complex scenarios, particularly when the composite key participates in relationships.

Custom Data Types and Converters

JPA provides built-in mapping for standard Java types, but sometimes you need to map custom Java types to database columns. The `AttributeConverter` interface allows you to create custom conversions between

Java types and database representations.

Creating Custom Converters with AttributeConverter

Here's an example of a converter that maps a custom `MonetaryAmount` class to a `DECIMAL` column:

```
// Custom data type
public class MonetaryAmount {

    private final BigDecimal amount;
    private final Currency currency;

    public MonetaryAmount(BigDecimal amount, Currency currency) {
        this.amount = amount;
        this.currency = currency;
    }

    // Getters and convenience methods
    public BigDecimal getAmount() {
        return amount;
    }

    public Currency getCurrency() {
        return currency;
    }

    public String getFormatted() {
        return currency.getSymbol() + " " + amount.toString();
    }
}

// Converter
@Converter
public class MonetaryAmountConverter implements AttributeConverter<MonetaryAmount,
String> {

    @Override
    public String convertToDatabaseColumn(MonetaryAmount monetaryAmount) {
        if (monetaryAmount == null) {
            return null;
        }
        return monetaryAmount.getAmount() + ":" +
monetaryAmount.getCurrency().getCurrencyCode();
    }

    @Override
    public MonetaryAmount convertToEntityAttribute(String dbData) {
        if (dbData == null || dbData.isEmpty()) {
            return null;
        }

        String[] parts = dbData.split(":");
```

```
            BigDecimal amount = new BigDecimal(parts[0]);
            Currency currency = Currency.getInstance(parts[1]);

            return new MonetaryAmount(amount, currency);
        }
    }

    // Using the converter in an entity
    @Entity
    @Table(name = "invoices")
    public class Invoice {

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

        @Column(name = "invoice_number")
        private String invoiceNumber;

        @Convert(converter = MonetaryAmountConverter.class)
        @Column(name = "amount")
        private MonetaryAmount amount;

        // Other fields, getters, and setters
    }
```

In this example, the `MonetaryAmount` custom type is converted to a string representation in the database using the format "amount:currencyCode". The converter handles both the conversion to the database column type and back to the entity field type.

Auto-applying Converters

You can make a converter apply automatically to all fields of a specific type by setting `autoApply = true`:

```
    @Converter(autoApply = true)
    public class MonetaryAmountConverter implements AttributeConverter<MonetaryAmount,
    String> {
        // Implementation remains the same
    }
```

With `autoApply = true`, you don't need to add the `@Convert` annotation to each field of the specified type—it's automatically applied.

Built-in Converters for Enums

For enum types, JPA provides built-in conversion, but you can customize it:

```
    public enum PaymentStatus {
        PENDING, PROCESSING, COMPLETED, FAILED, REFUNDED
```

```
}

@Entity
@Table(name = "payments")
public class Payment {

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

    // Store enum as a string
    @Enumerated(EnumType.STRING)
    @Column(name = "status")
    private PaymentStatus status;

    // Store enum as an ordinal (integer)
    @Enumerated(EnumType.ORDINAL)
    @Column(name = "type_ordinal")
    private PaymentType type;

    // Other fields, getters and setters
}
```

Using `EnumType.STRING` is generally safer than `EnumType.ORDINAL` as it's resilient to enum reordering, though it uses more storage space.

Entity Listeners and Callbacks

JPA provides a mechanism to intercept entity lifecycle events through callbacks and listeners. This allows you to execute code at specific points in an entity's lifecycle.

Entity Callbacks

Entity callbacks are methods within the entity class itself that are called when specific lifecycle events occur. They're annotated with the appropriate callback annotation:

```
@Entity
@Table(name = "products")
public class Product {

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

    private String name;
    private BigDecimal price;
    private Integer stock;

    @Column(name = "last_updated")
    private LocalDateTime lastUpdated;
```

```
    @Column(name = "created_at")
    private LocalDateTime createdAt;

    // Called before the entity is persisted (inserted)
    @PrePersist
    public void prePersist() {
        createdAt = LocalDateTime.now();
        lastUpdated = LocalDateTime.now();
    }

    // Called before the entity is updated
    @PreUpdate
    public void preUpdate() {
        lastUpdated = LocalDateTime.now();
    }

    // Called after the entity is loaded from the database
    @PostLoad
    public void postLoad() {
        // You could initialize transient fields or perform validations here
    }

    // Other callback annotations: @PostPersist, @PostUpdate, @PreRemove,
@PostRemove

    // Getters and setters
}
```

Available callback annotations include:

- `@PrePersist`: Before the entity is persisted (inserted)
- `@PostPersist`: After the entity is persisted
- `@PreUpdate`: Before the entity is updated
- `@PostUpdate`: After the entity is updated
- `@PreRemove`: Before the entity is deleted
- `@PostRemove`: After the entity is deleted
- `@PostLoad`: After the entity is loaded from the database

Entity Listeners

Entity listeners allow you to extract callback logic into separate classes. This promotes code reuse across multiple entities and follows the single responsibility principle.

```
// Entity listener class
public class AuditListener {

    @PrePersist
    public void setCreatedOn(Object entity) {
        if (entity instanceof Auditable) {
            Auditable auditable = (Auditable) entity;
```

```
            auditable.setCreatedAt(LocalDateTime.now());
            auditable.setCreatedBy(getCurrentUser());
        }
    }

    @PreUpdate
    public void setUpdatedOn(Object entity) {
        if (entity instanceof Auditable) {
            Auditable auditable = (Auditable) entity;
            auditable.setUpdatedAt(LocalDateTime.now());
            auditable.setUpdatedBy(getCurrentUser());
        }
    }

    private String getCurrentUser() {
        // Logic to get the current user from the security context
        return SecurityContextHolder.getContext().getAuthentication().getName();
    }
}

// Interface for auditable entities
public interface Auditable {
    void setCreatedAt(LocalDateTime dateTime);
    void setCreatedBy(String user);
    void setUpdatedAt(LocalDateTime dateTime);
    void setUpdatedBy(String user);
}

// Entity using the listener
@Entity
@Table(name = "employees")
@EntityListeners(AuditListener.class)
public class Employee implements Auditable {

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

    private String name;
    private String position;

    @Column(name = "created_at")
    private LocalDateTime createdAt;

    @Column(name = "created_by")
    private String createdBy;

    @Column(name = "updated_at")
    private LocalDateTime updatedAt;

    @Column(name = "updated_by")
    private String updatedBy;

    // Implementation of Auditable interface methods
```

```
    @Override
    public void setCreatedAt(LocalDateTime dateTime) {
        this.createdAt = dateTime;
    }

    @Override
    public void setCreatedBy(String user) {
        this.createdBy = user;
    }

    @Override
    public void setUpdatedAt(LocalDateTime dateTime) {
        this.updatedAt = dateTime;
    }

    @Override
    public void setUpdatedBy(String user) {
        this.updatedBy = user;
    }

    // Other getters and setters
}
```

You can apply multiple listeners to a single entity, and listeners can be used across many entities. This makes listeners a powerful tool for implementing cross-cutting concerns like auditing, validation, or caching.

Global Entity Listeners

For listeners that should apply to all entities, you can configure them in the `orm.xml` file:

```
<entity-mappings>
    <persistence-unit-metadata>
        <persistence-unit-defaults>
            <entity-listeners>
                <entity-listener
class="com.example.listeners.GlobalAuditListener">
                    <pre-persist method-name="trackCreation"/>
                    <pre-update method-name="trackUpdate"/>
                </entity-listener>
            </entity-listeners>
        </persistence-unit-defaults>
    </persistence-unit-metadata>
</entity-mappings>
```

Bi-directional and Uni-directional Relationships

In JPA, you can model relationships between entities in two ways: uni-directional (navigable in only one direction) or bi-directional (navigable in both directions).

Uni-directional Relationships

In a uni-directional relationship, only one entity knows about the other.

```
// Uni-directional One-to-Many
@Entity
public class Department {

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

    private String name;

    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
    @JoinColumn(name = "department_id") // This column is in the employee table
    private List<Employee> employees = new ArrayList<>();

    // Getters and setters
}

@Entity
public class Employee {

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

    private String name;

    // No reference back to Department

    // Getters and setters
}
```

In this example, `Department` knows about its `Employee` entities, but `Employee` does not know about its `Department`. You can navigate from `Department` to `Employee`, but not the other way around.

Bi-directional Relationships

In a bi-directional relationship, both entities know about each other.

```
// Bi-directional One-to-Many / Many-to-One
@Entity
public class Department {

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

    private String name;
```

```
    @OneToMany(mappedBy = "department", cascade = CascadeType.ALL, orphanRemoval =
true)
    private List<Employee> employees = new ArrayList<>();

    // Helper methods for managing the relationship
    public void addEmployee(Employee employee) {
        employees.add(employee);
        employee.setDepartment(this);
    }

    public void removeEmployee(Employee employee) {
        employees.remove(employee);
        employee.setDepartment(null);
    }

    // Getters and setters
}

@Entity
public class Employee {

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

    private String name;

    @ManyToOne
    @JoinColumn(name = "department_id")
    private Department department;

    // Getters and setters
}
```

In this bi-directional relationship:

- `Department` has a collection of `Employee` entities
- `Employee` has a reference to its `Department`
- The `mappedBy` attribute specifies that `Department` doesn't own the relationship—`Employee` does
- Helper methods in `Department` ensure both sides of the relationship stay in sync

Choosing Between Uni-directional and Bi-directional

When deciding which type of relationship to use, consider these factors:

Choose Uni-directional when:

- You only need to navigate the relationship in one direction
- You want to simplify your object model
- The relationship isn't a core part of your domain model
- You want to reduce the coupling between entities

Choose Bi-directional when:

- You need to navigate the relationship in both directions frequently
- The relationship is a fundamental part of your domain model
- You want to enforce relationship integrity on both sides
- You need to optimize queries that access the relationship from either side

Relationship Ownership

In JPA, one side of a relationship must be the "owner" side that controls database updates. For One-to-Many/Many-to-One relationships, the Many side is usually the owner (it has the foreign key). For Many-to-Many relationships, you must choose one side as the owner.

```
// Many-to-Many bi-directional relationship
@Entity
public class Student {

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

    private String name;

    @ManyToMany
    @JoinTable(
        name = "student_course",
        joinColumns = @JoinColumn(name = "student_id"),
        inverseJoinColumns = @JoinColumn(name = "course_id")
    )
    private Set<Course> courses = new HashSet<>();

    // Helper methods
    public void addCourse(Course course) {
        courses.add(course);
        course.getStudents().add(this);
    }

    public void removeCourse(Course course) {
        courses.remove(course);
        course.getStudents().remove(this);
    }

    // Getters and setters
}

@Entity
public class Course {

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

```
    private String name;

    @ManyToMany(mappedBy = "courses")
    private Set<Student> students = new HashSet<>();

    // Getters and setters
}
```

In this example, `Student` is the owner of the relationship (it has the `@JoinTable` annotation), while `Course` uses `mappedBy` to indicate it's not the owner.

Performance Considerations for Complex Mappings

Complex entity mappings and relationships can significantly impact application performance. Here are key strategies to optimize performance:

1. Fetch Type Optimization

JPA provides two fetch types: EAGER (fetch immediately) and LAZY (fetch on demand).

```
@Entity
public class Blog {

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

    private String title;

    // LAZY fetching - comments are loaded only when accessed
    @OneToMany(mappedBy = "blog", fetch = FetchType.LAZY)
    private List<Comment> comments = new ArrayList<>();

    // EAGER fetching - author is always loaded with the blog
    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "author_id")
    private Author author;

    // Getters and setters
}
```

Best practices for fetch types:

- Use LAZY for collections (`@OneToMany`, `@ManyToMany`) to avoid loading large amounts of data unnecessarily
- Consider EAGER for simple `@ManyToOne` or `@OneToOne` relationships that are frequently accessed
- The default fetch type is EAGER for `@ManyToOne` and `@OneToOne`, and LAZY for `@OneToMany` and `@ManyToMany`

2. N+1 Query Problem and Solutions

The N+1 query problem occurs when you load a collection of entities and then access a lazy-loaded relationship for each entity, causing N additional queries.

Consider this code that would cause N+1 queries:

```
List<Department> departments = entityManager.createQuery("SELECT d FROM Department
d", Department.class).getResultList();

// This will cause N additional queries, one for each department
for (Department department : departments) {
    System.out.println("Department: " + department.getName());
    System.out.println("Number of employees: " +
department.getEmployees().size());
}
```

Solutions to the N+1 problem:

a) Using JOIN FETCH in JPQL:

```
List<Department> departments = entityManager.createQuery(
    "SELECT d FROM Department d JOIN FETCH d.employees",
Department.class).getResultList();

// Now no additional queries are needed
for (Department department : departments) {
    System.out.println("Department: " + department.getName());
    System.out.println("Number of employees: " +
department.getEmployees().size());
}
```

b) Using Entity Graphs:

```
EntityGraph<Department> entityGraph =
entityManager.createEntityGraph(Department.class);
entityGraph.addSubgraph("employees");

Map<String, Object> properties = new HashMap<>();
properties.put("javax.persistence.fetchgraph", entityGraph);

Department department = entityManager.find(Department.class, departmentId,
properties);
// department.getEmployees() is now already loaded
```

c) Using Batch Fetching:

```
@Entity
public class Department {

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

    private String name;

    @OneToMany(mappedBy = "department")
    @BatchSize(size = 20)
    private List<Employee> employees = new ArrayList<>();

    // Getters and setters
}
```

With `@BatchSize`, Hibernate will fetch up to 20 collections at once using an IN clause, reducing the number of queries.

3. Caching Strategies

Hibernate's cache can significantly improve performance for complex entity relationships.

```
@Entity
@Cacheable
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class Country {

    @Id
    private String code;

    private String name;

    @OneToMany(mappedBy = "country")
    @Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
    private Set<City> cities = new HashSet<>();

    // Getters and setters
}
```

Cache concurrency strategies:

- READ_ONLY: For reference data that never changes
- NONSTRICT_READ_WRITE: For data that rarely changes
- READ_WRITE: For data that may be updated but where strict transaction isolation isn't required
- TRANSACTIONAL: For data requiring full transaction isolation

4. Selective Loading with DTOs

For complex object graphs, consider using DTOs to load only the data you need:

```
// DTO for customer summary
public class CustomerSummaryDTO {
    private Long id;
    private String name;
    private String email;
    private Integer orderCount;

    // Constructor, getters, setters
    public CustomerSummaryDTO(Long id, String name, String email, Integer
orderCount) {
        this.id = id;
        this.name = name;
        this.email = email;
        this.orderCount = orderCount;
    }
}

// JPQL query using constructor expression
List<CustomerSummaryDTO> customerSummaries = entityManager.createQuery(
    "SELECT NEW com.example.dto.CustomerSummaryDTO(c.id, c.name, c.email,
SIZE(c.orders)) " +
    "FROM Customer c",
    CustomerSummaryDTO.class).getResultList();
```

This approach avoids loading the entire Customer entity with all its associations when you only need specific fields.

5. Using Projections in Spring Data JPA

Spring Data JPA offers projections as an elegant way to selectively load entity data:

```
// Interface-based projection
public interface CustomerProjection {
    Long getId();
    String getName();
    String getEmail();

    @Value("#{target.orders.size()}")
    Integer getOrderCount();
}

// Repository method using projection
public interface CustomerRepository extends JpaRepository<Customer, Long> {
    List<CustomerProjection> findByCountry(String country);
}
```

6. Optimizing Inheritance Mappings

Each inheritance strategy has different performance characteristics:

- **SINGLE_TABLE**: Fastest for most queries (no joins needed) but can waste space with nullable columns
- **JOINED**: More normalized but requires joins for most operations, which can hurt performance
- **TABLE_PER_CLASS**: Avoids joins for single-class queries but polymorphic queries can be expensive

Choose the strategy that aligns with your most common query patterns.

Practical Examples and Code Demonstrations

Let's put these concepts together with a comprehensive example of a library management system.

```
// Base audit entity using
@MappedSuperclass
@MappedSuperclass
@EntityListeners(AuditingEntityListener.class)
public abstract class AuditableEntity {

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

    @CreatedDate
    @Column(name = "created_date", nullable = false, updatable = false)
    private LocalDateTime createdDate;

    @LastModifiedDate
    @Column(name = "last_modified_date")
    private LocalDateTime lastModifiedDate;

    @CreatedBy
    @Column(name = "created_by", length = 50)
    private String createdBy;

    @LastModifiedBy
    @Column(name = "last_modified_by", length = 50)
    private String lastModifiedBy;

    // Getters and setters
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public LocalDateTime getCreatedDate() {
        return createdDate;
    }

    public void setCreatedDate(LocalDateTime createdDate) {
```

```
        this.createdDate = createdDate;
    }

    public LocalDateTime getLastModifiedDate() {
        return lastModifiedDate;
    }

    public void setLastModifiedDate(LocalDateTime lastModifiedDate) {
        this.lastModifiedDate = lastModifiedDate;
    }

    public String getCreatedBy() {
        return createdBy;
    }

    public void setCreatedBy(String createdBy) {
        this.createdBy = createdBy;
    }

    public String getLastModifiedBy() {
        return lastModifiedBy;
    }

    public void setLastModifiedBy(String lastModifiedBy) {
        this.lastModifiedBy = lastModifiedBy;
    }
}

// Custom Attribute Converter for ISBN
@Converter(autoApply = true)
public class ISBNConverter implements AttributeConverter<ISBN, String> {

    @Override
    public String convertToDatabaseColumn(ISBN isbn) {
        return isbn != null ? isbn.toString() : null;
    }

    @Override
    public ISBN convertToEntityAttribute(String dbData) {
        try {
            return dbData != null ? new ISBN(dbData) : null;
        } catch (IllegalArgumentException e) {
            // Log the error
            return null;
        }
    }
}

// Custom ISBN value object
public class ISBN {
    private final String value;

    public ISBN(String isbn) {
        if (!isValid(isbn)) {
```

```
            throw new IllegalArgumentException("Invalid ISBN format: " + isbn);
        }
        this.value = normalizeISBN(isbn);
    }

    private boolean isValid(String isbn) {
        // ISBN validation logic
        if (isbn == null || isbn.isEmpty()) {
            return false;
        }

        String normalized = normalizeISBN(isbn);

        // Check if it's ISBN-10 or ISBN-13
        return normalized.length() == 10 || normalized.length() == 13;
    }

    private String normalizeISBN(String isbn) {
        // Remove hyphens and spaces
        return isbn.replaceAll("[-\\s]", "");
    }

    @Override
    public String toString() {
        return value;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        ISBN isbn = (ISBN) o;
        return Objects.equals(value, isbn.value);
    }

    @Override
    public int hashCode() {
        return Objects.hash(value);
    }
}

// Person class hierarchy with JOINED inheritance strategy
@Entity
@Table(name = "people")
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Person extends AuditableEntity {

    @Column(name = "first_name", nullable = false)
    private String firstName;

    @Column(name = "last_name", nullable = false)
    private String lastName;

    @Column(name = "email", unique = true)
```

```
    private String email;

    @Column(name = "phone")
    private String phone;

    // Getters and setters
    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getPhone() {
        return phone;
    }

    public void setPhone(String phone) {
        this.phone = phone;
    }

    // Convenience method
    public String getFullName() {
        return firstName + " " + lastName;
    }
}

@Entity
@Table(name = "patrons")
public class Patron extends Person {

    @Column(name = "library_card_number", unique = true)
    private String libraryCardNumber;

    @Column(name = "membership_date")
    private LocalDate membershipDate;
```

```
    @Column(name = "status")
    @Enumerated(EnumType.STRING)
    private PatronStatus status = PatronStatus.ACTIVE;

    @OneToMany(mappedBy = "patron", cascade = CascadeType.ALL, orphanRemoval =
true)
    private List<Loan> loans = new ArrayList<>();

    // Pre-persist callback
    @PrePersist
    public void prePersist() {
        if (membershipDate == null) {
            membershipDate = LocalDate.now();
        }
        if (libraryCardNumber == null) {
            // Generate a library card number
            libraryCardNumber = "LIB" + System.currentTimeMillis();
        }
    }

    // Helper methods for managing the relationship
    public void addLoan(Loan loan) {
        loans.add(loan);
        loan.setPatron(this);
    }

    public void removeLoan(Loan loan) {
        loans.remove(loan);
        loan.setPatron(null);
    }

    // Getters and setters
    public String getLibraryCardNumber() {
        return libraryCardNumber;
    }

    public void setLibraryCardNumber(String libraryCardNumber) {
        this.libraryCardNumber = libraryCardNumber;
    }

    public LocalDate getMembershipDate() {
        return membershipDate;
    }

    public void setMembershipDate(LocalDate membershipDate) {
        this.membershipDate = membershipDate;
    }

    public PatronStatus getStatus() {
        return status;
    }

    public void setStatus(PatronStatus status) {
        this.status = status;
```

```
    }

    public List<Loan> getLoans() {
        return loans;
    }

    public void setLoans(List<Loan> loans) {
        this.loans = loans;
    }
}

@Entity
@Table(name = "librarians")
public class Librarian extends Person {

    @Column(name = "employee_id", unique = true)
    private String employeeId;

    @Column(name = "hire_date")
    private LocalDate hireDate;

    @Column(name = "position")
    private String position;

    // Getters and setters
    public String getEmployeeId() {
        return employeeId;
    }

    public void setEmployeeId(String employeeId) {
        this.employeeId = employeeId;
    }

    public LocalDate getHireDate() {
        return hireDate;
    }

    public void setHireDate(LocalDate hireDate) {
        this.hireDate = hireDate;
    }

    public String getPosition() {
        return position;
    }

    public void setPosition(String position) {
        this.position = position;
    }
}

// Book item with @SecondaryTable for additional details
@Entity
@Table(name = "books")
@SecondaryTable(
```

```
        name = "book_details",
        pkJoinColumns = @PrimaryKeyJoinColumn(name = "book_id")
    )
    public class Book extends AuditableEntity {

        @Column(name = "title", nullable = false)
        private String title;

        @Column(name = "isbn", unique = true)
        private ISBN isbn;

        @ManyToOne
        @JoinColumn(name = "publisher_id")
        private Publisher publisher;

        @Column(name = "publication_year")
        private Integer publicationYear;

        @ManyToMany
        @JoinTable(
            name = "book_author",
            joinColumns = @JoinColumn(name = "book_id"),
            inverseJoinColumns = @JoinColumn(name = "author_id")
        )
        private Set<Author> authors = new HashSet<>();

        // Fields in the secondary table
        @Column(name = "description", table = "book_details")
        private String description;

        @Column(name = "language", table = "book_details")
        private String language;

        @Column(name = "page_count", table = "book_details")
        private Integer pageCount;

        @Enumerated(EnumType.STRING)
        @Column(name = "book_format", table = "book_details")
        private BookFormat format;

        @OneToMany(mappedBy = "book", cascade = CascadeType.ALL, orphanRemoval = true)
        private List<BookItem> items = new ArrayList<>();

        // Helper methods for managing relationships
        public void addAuthor(Author author) {
            authors.add(author);
            author.getBooks().add(this);
        }

        public void removeAuthor(Author author) {
            authors.remove(author);
            author.getBooks().remove(this);
        }
```

```
    public void addItem(BookItem item) {
        items.add(item);
        item.setBook(this);
    }

    public void removeItem(BookItem item) {
        items.remove(item);
        item.setBook(null);
    }

    // Getters and setters
    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public ISBN getIsbn() {
        return isbn;
    }

    public void setIsbn(ISBN isbn) {
        this.isbn = isbn;
    }

    public Publisher getPublisher() {
        return publisher;
    }

    public void setPublisher(Publisher publisher) {
        this.publisher = publisher;
    }

    public Integer getPublicationYear() {
        return publicationYear;
    }

    public void setPublicationYear(Integer publicationYear) {
        this.publicationYear = publicationYear;
    }

    public Set<Author> getAuthors() {
        return authors;
    }

    public void setAuthors(Set<Author> authors) {
        this.authors = authors;
    }

    public String getDescription() {
        return description;
    }
```

```
    public void setDescription(String description) {
        this.description = description;
    }

    public String getLanguage() {
        return language;
    }

    public void setLanguage(String language) {
        this.language = language;
    }

    public Integer getPageCount() {
        return pageCount;
    }

    public void setPageCount(Integer pageCount) {
        this.pageCount = pageCount;
    }

    public BookFormat getFormat() {
        return format;
    }

    public void setFormat(BookFormat format) {
        this.format = format;
    }

    public List<BookItem> getItems() {
        return items;
    }

    public void setItems(List<BookItem> items) {
        this.items = items;
    }
}

@Entity
@Table(name = "book_items")
public class BookItem extends AuditableEntity {

    @Column(name = "barcode", unique = true)
    private String barcode;

    @ManyToOne
    @JoinColumn(name = "book_id", nullable = false)
    private Book book;

    @Enumerated(EnumType.STRING)
    @Column(name = "status")
    private BookStatus status = BookStatus.AVAILABLE;

    @OneToMany(mappedBy = "bookItem", cascade = CascadeType.ALL)
```

```
    private List<Loan> loans = new ArrayList<>();

    @Column(name = "acquisition_date")
    private LocalDate acquisitionDate;

    @PrePersist
    public void prePersist() {
        if (barcode == null) {
            // Generate a barcode
            barcode = "ITEM" + System.currentTimeMillis();
        }
        if (acquisitionDate == null) {
            acquisitionDate = LocalDate.now();
        }
    }

    // Helper methods for managing relationships
    public void addLoan(Loan loan) {
        loans.add(loan);
        loan.setBookItem(this);
    }

    // Getters and setters
    public String getBarcode() {
        return barcode;
    }

    public void setBarcode(String barcode) {
        this.barcode = barcode;
    }

    public Book getBook() {
        return book;
    }

    public void setBook(Book book) {
        this.book = book;
    }

    public BookStatus getStatus() {
        return status;
    }

    public void setStatus(BookStatus status) {
        this.status = status;
    }

    public List<Loan> getLoans() {
        return loans;
    }

    public void setLoans(List<Loan> loans) {
        this.loans = loans;
    }
```

```
    public LocalDate getAcquisitionDate() {
        return acquisitionDate;
    }

    public void setAcquisitionDate(LocalDate acquisitionDate) {
        this.acquisitionDate = acquisitionDate;
    }
}

// Loan entity with composite key using @EmbeddedId
@Embeddable
public class LoanId implements Serializable {

    @Column(name = "patron_id")
    private Long patronId;

    @Column(name = "book_item_id")
    private Long bookItemId;

    @Column(name = "loan_date")
    private LocalDate loanDate;

    // Default constructor required by JPA
    public LoanId() {
    }

    public LoanId(Long patronId, Long bookItemId, LocalDate loanDate) {
        this.patronId = patronId;
        this.bookItemId = bookItemId;
        this.loanDate = loanDate;
    }

    // Equals and hashCode methods
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        LoanId loanId = (LoanId) o;
        return Objects.equals(patronId, loanId.patronId) &&
               Objects.equals(bookItemId, loanId.bookItemId) &&
               Objects.equals(loanDate, loanId.loanDate);
    }

    @Override
    public int hashCode() {
        return Objects.hash(patronId, bookItemId, loanDate);
    }

    // Getters and setters
    public Long getPatronId() {
        return patronId;
    }
```

```
    public void setPatronId(Long patronId) {
        this.patronId = patronId;
    }

    public Long getBookItemId() {
        return bookItemId;
    }

    public void setBookItemId(Long bookItemId) {
        this.bookItemId = bookItemId;
    }

    public LocalDate getLoanDate() {
        return loanDate;
    }

    public void setLoanDate(LocalDate loanDate) {
        this.loanDate = loanDate;
    }
}

@Entity
@Table(name = "loans")
@EntityListeners(LoanEntityListener.class)
public class Loan {

    @EmbeddedId
    private LoanId id;

    @ManyToOne
    @MapsId("patronId")
    @JoinColumn(name = "patron_id")
    private Patron patron;

    @ManyToOne
    @MapsId("bookItemId")
    @JoinColumn(name = "book_item_id")
    private BookItem bookItem;

    @Column(name = "due_date", nullable = false)
    private LocalDate dueDate;

    @Column(name = "return_date")
    private LocalDate returnDate;

    @Column(name = "fine_amount")
    private BigDecimal fineAmount;

    @PrePersist
    public void prePersist() {
        if (id == null) {
            id = new LoanId(
                patron.getId(),
                bookItem.getId(),
```

```
                LocalDate.now()
            );
        }
    }

    // Calculate fine if returned late
    public BigDecimal calculateFine() {
        if (returnDate == null || !returnDate.isAfter(dueDate)) {
            return BigDecimal.ZERO;
        }

        long daysLate = ChronoUnit.DAYS.between(dueDate, returnDate);
        // Assuming $0.50 per day late
        return new BigDecimal("0.50").multiply(new BigDecimal(daysLate));
    }

    // Getters and setters
    public LoanId getId() {
        return id;
    }

    public void setId(LoanId id) {
        this.id = id;
    }

    public Patron getPatron() {
        return patron;
    }

    public void setPatron(Patron patron) {
        this.patron = patron;
    }

    public BookItem getBookItem() {
        return bookItem;
    }

    public void setBookItem(BookItem bookItem) {
        this.bookItem = bookItem;
    }

    public LocalDate getDueDate() {
        return dueDate;
    }

    public void setDueDate(LocalDate dueDate) {
        this.dueDate = dueDate;
    }

    public LocalDate getReturnDate() {
        return returnDate;
    }

    public void setReturnDate(LocalDate returnDate) {
```

```
        this.returnDate = returnDate;
    }

    public BigDecimal getFineAmount() {
        return fineAmount;
    }

    public void setFineAmount(BigDecimal fineAmount) {
        this.fineAmount = fineAmount;
    }
}

// Entity listener for loan status updates
public class LoanEntityListener {

    @PrePersist
    public void prePersist(Loan loan) {
        // Set the book item status to LOANED when a loan is created
        loan.getBookItem().setStatus(BookStatus.LOANED);
    }

    @PostUpdate
    public void postUpdate(Loan loan) {
        // When a book is returned, update its status
        if (loan.getReturnDate() != null) {
            loan.getBookItem().setStatus(BookStatus.AVAILABLE);

            // Calculate and set fine if returned late
            if (loan.getReturnDate().isAfter(loan.getDueDate())) {
                loan.setFineAmount(loan.calculateFine());
            }
        }
    }
}

@Entity
@Table(name = "authors")
public class Author extends AuditableEntity {

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

    @Column(name = "biography", length = 2000)
    private String biography;

    @ManyToMany(mappedBy = "authors")
    private Set<Book> books = new HashSet<>();

    // Getters and setters
    public String getName() {
        return name;
    }

    public void setName(String name) {
```

```
        this.name = name;
    }

    public String getBiography() {
        return biography;
    }

    public void setBiography(String biography) {
        this.biography = biography;
    }

    public Set<Book> getBooks() {
        return books;
    }

    public void setBooks(Set<Book> books) {
        this.books = books;
    }
}

@Entity
@Table(name = "publishers")
public class Publisher extends AuditableEntity {

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

    @Column(name = "address")
    private String address;

    @Column(name = "website")
    private String website;

    @OneToMany(mappedBy = "publisher")
    private Set<Book> publishedBooks = new HashSet<>();

    // Getters and setters
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    public String getWebsite() {
```

```
        return website;
    }

    public void setWebsite(String website) {
        this.website = website;
    }

    public Set<Book> getPublishedBooks() {
        return publishedBooks;
    }

    public void setPublishedBooks(Set<Book> publishedBooks) {
        this.publishedBooks = publishedBooks;
    }
}

// Enums for the domain model
public enum PatronStatus {
    ACTIVE, SUSPENDED, EXPIRED
}

public enum BookStatus {
    AVAILABLE, LOANED, RESERVED, LOST, DAMAGED
}

public enum BookFormat {
    HARDCOVER, PAPERBACK, AUDIO_BOOK, E_BOOK, LARGE_PRINT
}

// Service class demonstrating optimized loading
@Service
@Transactional
public class LibraryService {

    @PersistenceContext
    private EntityManager entityManager;

    @Autowired
    private BookRepository bookRepository;

    @Autowired
    private PatronRepository patronRepository;

    // Avoiding N+1 problem with JOIN FETCH
    public List<Book> findAllBooksWithAuthors() {
        return entityManager.createQuery(
            "SELECT DISTINCT b FROM Book b LEFT JOIN FETCH b.authors", Book.class)
            .getResultList();
    }

    // Using entity graph for optimized loading
    public Book findBookWithAllDetails(Long bookId) {
        EntityGraph<Book> entityGraph =
entityManager.createEntityGraph(Book.class);
```

```
        entityGraph.addAttributeNodes("authors", "publisher", "items");

        Map<String, Object> properties = new HashMap<>();
        properties.put("javax.persistence.fetchgraph", entityGraph);

        return entityManager.find(Book.class, bookId, properties);
    }

    // Using projection to avoid loading complete entities
    public List<BookSummaryDTO> getBookSummaries() {
        return entityManager.createQuery(
            "SELECT NEW com.example.dto.BookSummaryDTO(b.id, b.title,
b.isbn.toString(), " +
            "b.publicationYear, p.name, SIZE(b.items)) " +
            "FROM Book b JOIN b.publisher p",
            BookSummaryDTO.class)
            .getResultList();
    }

    // Using batch size for optimization
    @Transactional(readOnly = true)
    public List<Patron> findPatronsWithActiveLoans() {
        // Patrons.loans has @BatchSize(size=10) annotation
        List<Patron> patrons = patronRepository.findByStatus(PatronStatus.ACTIVE);

        // This would trigger batch loading rather than N+1 queries
        for (Patron patron : patrons) {
            if (!patron.getLoans().isEmpty()) {
                // Process active loans
            }
        }

        return patrons;
    }

    // Issuing a loan with transactional business logic
    @Transactional
    public Loan issueLoan(Long patronId, Long bookItemId, int loanDurationDays) {
        Patron patron = patronRepository.findById(patronId)
            .orElseThrow(() -> new IllegalArgumentException("Patron not found"));

        BookItem bookItem = entityManager.find(BookItem.class, bookItemId);
        if (bookItem == null) {
            throw new IllegalArgumentException("Book item not found");
        }

        if (bookItem.getStatus() != BookStatus.AVAILABLE) {
            throw new IllegalStateException("Book is not available for loan");
        }

        // Create the loan
        Loan loan = new Loan();
        loan.setPatron(patron);
        loan.setBookItem(bookItem);
```

```
        // Set due date (current date + loan duration)
        LocalDate dueDate = LocalDate.now().plusDays(loanDurationDays);
        loan.setDueDate(dueDate);

        // The relationship management helper methods will handle bidirectional
associations
        patron.addLoan(loan);
        bookItem.addLoan(loan);

        // BookItem status will be updated by the LoanEntityListener

        entityManager.persist(loan);

        return loan;
    }

    // Returning a book
    @Transactional
    public void returnBook(Long patronId, Long bookItemId) {
        LoanId loanId = new LoanId();
        loanId.setPatronId(patronId);
        loanId.setBookItemId(bookItemId);
        // We need the loan date, which we can get by finding the active loan

        String query = "SELECT l FROM Loan l WHERE l.patron.id = :patronId AND " +
                       "l.bookItem.id = :bookItemId AND l.returnDate IS NULL";

        Loan loan = entityManager.createQuery(query, Loan.class)
            .setParameter("patronId", patronId)
            .setParameter("bookItemId", bookItemId)
            .getSingleResult();

        // Set return date
        loan.setReturnDate(LocalDate.now());

        // Calculate fine if returned late
        if (LocalDate.now().isAfter(loan.getDueDate())) {
            loan.setFineAmount(loan.calculateFine());
        }

        // The LoanEntityListener will handle updating the BookItem status
        entityManager.merge(loan);
    }
}

// DTO for optimized loading
public class BookSummaryDTO {
    private Long id;
    private String title;
    private String isbn;
    private Integer publicationYear;
    private String publisherName;
    private Integer copiesCount;
```

```
    public BookSummaryDTO(Long id, String title, String isbn,
                          Integer publicationYear, String publisherName,
                          Integer copiesCount) {
        this.id = id;
        this.title = title;
        this.isbn = isbn;
        this.publicationYear = publicationYear;
        this.publisherName = publisherName;
        this.copiesCount = copiesCount;
    }

    // Getters only - no setters needed for DTOs
    public Long getId() {
        return id;
    }

    public String getTitle() {
        return title;
    }

    public String getIsbn() {
        return isbn;
    }

    public Integer getPublicationYear() {
        return publicationYear;
    }

    public String getPublisherName() {
        return publisherName;
    }

    public Integer getCopiesCount() {
        return copiesCount;
    }
}

// Spring Data JPA repositories
public interface BookRepository extends JpaRepository<Book, Long> {

    // Using a custom entity graph
    @EntityGraph(attributePaths = {"authors", "publisher"})
    List<Book> findByPublicationYearAfter(Integer year);

    // Using a projection interface
    interface BookTitleProjection {
        Long getId();
        String getTitle();

        @Value("#{target.isbn.toString()}")
        String getIsbn();

        Integer getPublicationYear();
```

```
    }

    List<BookTitleProjection> findByTitleContaining(String title);

    // Using specifications for dynamic queries
    @EntityGraph(attributePaths = {"items"})
    List<Book> findAll(Specification<Book> specification);
}

public interface PatronRepository extends JpaRepository<Patron, Long> {

    List<Patron> findByStatus(PatronStatus status);

    // Special query for active borrowers
    @Query("SELECT DISTINCT p FROM Patron p JOIN FETCH p.loans l WHERE
l.returnDate IS NULL")
    List<Patron> findAllWithActiveLoans();
}
```

Chapter Summary and Key Takeaways

In this chapter, we've explored advanced data mapping techniques and relationship management in Spring Data JPA. Here are the key concepts we've covered:

Advanced Entity Mappings

1. **@MappedSuperclass** provides a powerful way to extract common fields and behavior into a base class without creating a table for it, promoting code reuse and cleaner entities.
2. **Inheritance Strategies** with `@Inheritance`:
    - SINGLE_TABLE: Best for performance but can waste space
    - JOINED: Best for normalization but requires joins
    - TABLE_PER_CLASS: Avoids joins for single-class queries
3. **@SecondaryTable** allows mapping a single entity to multiple tables, which is valuable for working with legacy databases or logical data separation.

Composite Keys and Embeddable IDs

1. **@EmbeddedId** creates a clean, encapsulated approach to composite keys by using a separate class for the key.
2. **@IdClass** provides an alternative approach where key fields are duplicated in both the entity and a separate ID class.
3. **@MapsId** creates a clean way to map foreign key relationships that are part of a composite key.

Custom Data Types and Converters

1. **AttributeConverter** enables mapping between custom Java types and database column types, enhancing type safety and encapsulation.
2. **autoApply** automatically applies a converter to all fields of a specific type.
3. **Built-in Enum Conversions** with `@Enumerated` allow storing enums as either strings or ordinals.

Entity Listeners and Callbacks

1. **Entity Callbacks** (`@PrePersist`, `@PostLoad`, etc.) allow entities to respond to lifecycle events.
2. **Entity Listeners** extract callback logic into separate classes for reuse across entities.
3. **Global Entity Listeners** can be configured to apply to all entities in the persistence unit.

Bi-directional and Uni-directional Relationships

1. **Choosing Relationship Direction** depends on navigation needs, with bi-directional relationships offering more flexibility but requiring more management.
2. **Relationship Ownership** is critical, with one side designated as the owner that controls database updates.
3. **Helper Methods should be used in bi-directional relationships to maintain consistency and avoid common pitfalls. Specifically:

Synchronization: Helper methods ensure that both sides of the relationship are updated simultaneously. For example, when adding a child to a parent, the parent should be added to the child's collection as well.

Encapsulation: They encapsulate the logic for managing the relationship, preventing direct manipulation of the collections or references, which can lead to inconsistencies.

Clarity and Readability: Helper methods improve code clarity by providing a well-defined interface for managing relationships, making the code easier to understand and maintain.

Preventing Infinite Loops: In bi-directional relationships, naive implementations of toString() or equals() methods can lead to infinite loops due to recursive calls. Helper methods can assist to prevent this issue by only updating the owning side of the relationship.

Reducing Errors: They help reduce errors that can occur when manually managing relationships, such as forgetting to update both sides of the relationship or accidentally creating duplicate entries.

Example:

```
If a Parent has a list of Child objects, a addChild(Child child) method in the
Parent class would not only add the Child to the Parent's list, but also set the
Child's parent reference to the Parent instance.
Likewise a removeChild(Child child) method should remove the child from the
parent's list, and set the childs parent to null.
```

````
# Chapter 7: Data Validation and Transactions in Spring Boot Data Applications

## Introduction

In modern data-centric applications, ensuring data integrity and consistency is
paramount. Spring Boot provides robust mechanisms for data validation and
transaction management that help maintain data quality while simplifying
development. This chapter explores these essential capabilities in depth, showing
you how to implement proper validation and transaction strategies in your Spring
Boot data applications.

## Data Validation with Bean Validation API in Spring Boot Data

The Bean Validation API (JSR-303/JSR-349, and more recently JSR-380) provides a
standard approach to validating data in Java applications. Spring Boot seamlessly
integrates this API, making it straightforward to validate your data models.

### Adding Validation Annotations to JPA Entities

Let's start with a simple example of a JPA entity with validation constraints:

```java
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.validation.constraints.*;
import java.time.LocalDate;

@Entity
@Table(name = "employees")
public class Employee {

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

    @NotBlank(message = "Name is required")
    @Size(min = 2, max = 100, message = "Name must be between 2 and 100
characters")
    private String name;

    @Email(message = "Email should be valid")
    @NotNull(message = "Email cannot be null")
    private String email;

    @Min(value = 18, message = "Age should not be less than 18")
    @Max(value = 100, message = "Age should not be greater than 100")
    private int age;

    @Past(message = "Hire date must be in the past")
    private LocalDate hireDate;

    @Pattern(regexp = "^\\d{10}$", message = "Phone number must be 10 digits")
    private String phoneNumber;
````
````

```
    // Getters and setters
}
```

Similarly, you can apply these annotations to MongoDB documents:

```
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;
import javax.validation.constraints.*;
import java.time.LocalDate;

@Document(collection = "employees")
public class Employee {

    @Id
    private String id;

    @NotBlank(message = "Name is required")
    @Size(min = 2, max = 100, message = "Name must be between 2 and 100
characters")
    private String name;

    // Other fields with validation annotations (same as JPA example)
}
```

Common Validation Annotations

Annotation	Description	Example Usage
`@NotNull`	Value cannot be null	`@NotNull(message = "Field cannot be null")`
`@NotBlank`	String value cannot be null or whitespace	`@NotBlank(message = "Field cannot be empty")`
`@NotEmpty`	Collection, map, array, or string cannot be null or empty	`@NotEmpty(message = "List cannot be empty")`
`@Size`	Element size must be between min and max	`@Size(min = 2, max = 30)`
`@Min`	Value must be at least specified value	`@Min(value = 18)`
`@Max`	Value must be at most specified value	`@Max(value = 100)`
`@Email`	Value must be valid email	`@Email(message = "Invalid email")`
`@Pattern`	Value must match regular expression	`@Pattern(regexp = "^\\d{10}$")`
`@Past`	Date must be in the past	`@Past(message = "Date must be in the past")`

Annotation	Description	Example Usage
`@Future`	Date must be in the future	`@Future(message = "Date must be in future")`

Triggering Validation in Spring Boot Data

Spring Boot automatically triggers validation at various points:

1. **Repository Method Execution**: When you save or update an entity using repository methods
2. **REST API Validation**: When you use the `@Valid` annotation in your controller methods
3. **Service Layer Validation**: When you use the `@Validated` annotation on your service class

Here's how validation works in a controller:

```
import org.springframework.http.ResponseEntity;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.*;
import javax.validation.Valid;

@RestController
@RequestMapping("/api/employees")
public class EmployeeController {

    private final EmployeeRepository employeeRepository;

    public EmployeeController(EmployeeRepository employeeRepository) {
        this.employeeRepository = employeeRepository;
    }

    @PostMapping
    public ResponseEntity<Employee> createEmployee(@Valid @RequestBody Employee
employee) {
        // The @Valid annotation triggers validation
        // If validation fails, Spring automatically throws a
MethodArgumentNotValidException
        Employee savedEmployee = employeeRepository.save(employee);
        return ResponseEntity.ok(savedEmployee);
    }
}
```

Handling Validation Errors

When validation fails, Spring throws validation exceptions. You can handle these exceptions globally using `@ControllerAdvice`:

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.validation.FieldError;
```

```
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;

import java.util.HashMap;
import java.util.Map;

@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<Map<String, String>>
handleValidationExceptions(MethodArgumentNotValidException ex) {
        Map<String, String> errors = new HashMap<>();

        ex.getBindingResult().getAllErrors().forEach(error -> {
            String fieldName = ((FieldError) error).getField();
            String errorMessage = error.getDefaultMessage();
            errors.put(fieldName, errorMessage);
        });

        return new ResponseEntity<>(errors, HttpStatus.BAD_REQUEST);
    }
}
```

This handler returns a map of field names and their corresponding error messages when validation fails, making it easy to display validation errors to users.

Custom Validation in Spring Boot Data

While the standard validation annotations cover many common use cases, you often need custom validation logic for business-specific rules.

Creating a Custom Validation Annotation

Let's create a custom validation annotation to check if an employee's salary is within the range for their department:

1. First, create the annotation:

```
import javax.validation.Constraint;
import javax.validation.Payload;
import java.lang.annotation.*;

@Documented
@Constraint(validatedBy = SalaryConstraintValidator.class)
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface ValidSalaryRange {

    String message() default "Salary is not within valid range for this
```

```
department";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};
}
```

2. Then, implement the validator:

```
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

public class SalaryConstraintValidator implements
ConstraintValidator<ValidSalaryRange, Employee> {

    @Override
    public void initialize(ValidSalaryRange constraintAnnotation) {
    }

    @Override
    public boolean isValid(Employee employee, ConstraintValidatorContext context)
{
        if (employee == null || employee.getDepartment() == null) {
            return true; // Let @NotNull handle this case
        }

        // Custom validation logic based on department and salary
        switch (employee.getDepartment()) {
            case "IT":
                return employee.getSalary() >= 50000 && employee.getSalary() <=
150000;
            case "HR":
                return employee.getSalary() >= 40000 && employee.getSalary() <=
100000;
            case "SALES":
                return employee.getSalary() >= 35000 && employee.getSalary() <=
120000;
            default:
                return employee.getSalary() >= 30000 && employee.getSalary() <=
90000;
        }
    }
}
```

3. Apply the annotation to your entity:

```
@Entity
@ValidSalaryRange
public class Employee {
```

```
    // Fields, getters, and setters

    private String department;
    private double salary;

    // Remaining code from earlier examples
}
```

Creating a More Complex Custom Validator with Dependencies

For more complex validation scenarios, you might need to inject services or repositories into your validator:

```
import org.springframework.beans.factory.annotation.Autowired;
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

public class UniqueEmailValidator implements ConstraintValidator<UniqueEmail,
String> {

    private final EmployeeRepository employeeRepository;

    @Autowired
    public UniqueEmailValidator(EmployeeRepository employeeRepository) {
        this.employeeRepository = employeeRepository;
    }

    @Override
    public boolean isValid(String email, ConstraintValidatorContext context) {
        if (email == null) {
            return true; // Let @NotNull handle this
        }

        // Check if email already exists in the database
        return !employeeRepository.existsByEmail(email);
    }
}
```

And the corresponding annotation:

```
@Documented
@Constraint(validatedBy = UniqueEmailValidator.class)
@Target({ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface UniqueEmail {

    String message() default "Email already exists";

    Class<?>[] groups() default {};
```

```
    Class<? extends Payload>[] payload() default {};
}
```

Apply it to the email field:

```
@Email(message = "Email should be valid")
@NotNull(message = "Email cannot be null")
@UniqueEmail
private String email;
```

Transaction Management in Spring Boot Data

Transactions ensure that a series of data operations either all succeed or all fail, maintaining data consistency. Spring Boot provides flexible approaches to transaction management.

Declarative Transaction Management

The most common way to manage transactions in Spring Boot is declaratively using the `@Transactional` annotation:

```
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
public class EmployeeService {

    private final EmployeeRepository employeeRepository;
    private final DepartmentRepository departmentRepository;

    public EmployeeService(EmployeeRepository employeeRepository,
                           DepartmentRepository departmentRepository) {
        this.employeeRepository = employeeRepository;
        this.departmentRepository = departmentRepository;
    }

    @Transactional
    public Employee hireEmployee(Employee employee, String departmentName) {
        // Both operations will be in the same transaction
        Department department = departmentRepository.findByName(departmentName)
            .orElseThrow(() -> new RuntimeException("Department not found"));

        employee.setDepartment(department);
        department.incrementEmployeeCount();

        departmentRepository.save(department);
        return employeeRepository.save(employee);
    }
}
```

Transaction Attributes

The `@Transactional` annotation supports several attributes that control transaction behavior:

Propagation

Propagation defines how transactions relate to each other when methods are called within transactions:

```
@Transactional(propagation = Propagation.REQUIRED)
```

Propagation Level	Description
`REQUIRED` (default)	Uses existing transaction if available, creates new one if not
`SUPPORTS`	Uses existing transaction if available, runs non-transactionally if not
`MANDATORY`	Uses existing transaction, throws exception if no transaction exists
`REQUIRES_NEW`	Creates new transaction, suspends current transaction if one exists
`NOT_SUPPORTED`	Executes non-transactionally, suspends current transaction if one exists
`NEVER`	Executes non-transactionally, throws exception if transaction exists
`NESTED`	Executes within a nested transaction if transaction exists, otherwise like REQUIRED

Isolation

Isolation determines how changes made by one transaction are visible to other transactions:

```
@Transactional(isolation = Isolation.READ_COMMITTED)
```

Isolation Level	Description	Dirty Reads	Non-repeatable Reads	Phantom Reads
`DEFAULT`	Database default isolation	Depends on DB	Depends on DB	Depends on DB
`READ_UNCOMMITTED`	Lowest isolation, can read uncommitted data	Possible	Possible	Possible
`READ_COMMITTED`	Can only read committed data	Prevented	Possible	Possible
`REPEATABLE_READ`	Same reads return same data within transaction	Prevented	Prevented	Possible
`SERIALIZABLE`	Complete isolation	Prevented	Prevented	Prevented

Read-Only

Marking a transaction as read-only allows optimizations and prevents data modification:

```
@Transactional(readOnly = true)
public List<Employee> getAllEmployees() {
    return employeeRepository.findAll();
}
```

Timeout

Set a timeout for long-running transactions:

```
@Transactional(timeout = 30) // 30 seconds
```

Rollback Rules

Control when transactions are rolled back:

```
@Transactional(rollbackFor = {SQLException.class},
               noRollbackFor = {DataAccessException.class})
```

Programmatic Transaction Management

For more fine-grained control, you can manage transactions programmatically using `TransactionTemplate`:

```
import org.springframework.stereotype.Service;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.support.TransactionTemplate;

@Service
public class EmployeeService {

    private final EmployeeRepository employeeRepository;
    private final TransactionTemplate transactionTemplate;

    public EmployeeService(EmployeeRepository employeeRepository,
                           PlatformTransactionManager transactionManager) {
        this.employeeRepository = employeeRepository;
        this.transactionTemplate = new TransactionTemplate(transactionManager);
        // Configure the transaction template if needed

this.transactionTemplate.setIsolationLevel(TransactionDefinition.ISOLATION_READ_CO
MMITTED);
        this.transactionTemplate.setTimeout(30); // 30 seconds
    }
```

```
    public Employee saveEmployeeWithAudit(Employee employee, AuditLog auditLog) {
        return transactionTemplate.execute(status -> {
            try {
                Employee savedEmployee = employeeRepository.save(employee);
                auditLog.setEntityId(savedEmployee.getId());
                auditLogRepository.save(auditLog);
                return savedEmployee;
            } catch (Exception e) {
                // Decide programmatically whether to roll back
                status.setRollbackOnly();
                throw e;
            }
        });
    }

    // For read-only operations, you can use a specific configuration
    public List<Employee> getEmployeesByDepartment(String department) {
        TransactionTemplate readOnlyTemplate = new
TransactionTemplate(transactionManager);
        readOnlyTemplate.setReadOnly(true);

        return readOnlyTemplate.execute(status ->
            employeeRepository.findByDepartment(department));
    }
}
```

Transaction Management Best Practices

1. **Keep transactions short and focused**: Long-running transactions can lead to resource contention and locking issues.
2. **Set appropriate isolation levels**: Choose the lowest isolation level that meets your consistency requirements.
3. **Use read-only transactions for queries**: This allows the database to optimize performance.
4. **Be careful with exception handling**: Understand which exceptions trigger rollbacks and which don't.
5. **Be mindful of transaction boundaries**: Ensure that all related operations are included within the same transaction scope.
6. **Consider using distributed transactions for multi-database operations**: For complex scenarios involving multiple databases, consider JTA (Java Transaction API).
7. **Avoid calling @Transactional methods from within the same class**: Due to Spring's proxy-based approach, internal method calls do not go through the proxy, so transaction attributes might not be applied.

```
// PROBLEMATIC: Internal calls don't go through the proxy
@Service
public class EmployeeService {
```

```
    @Transactional(readOnly = true)
    public Employee getEmployee(Long id) {
        return employeeRepository.findById(id)
            .orElseThrow(() -> new RuntimeException("Employee not found"));
    }

    @Transactional
    public void updateEmployee(Long id, String newName) {
        // THIS CALL WON'T BE READ-ONLY because it's an internal call
        Employee employee = getEmployee(id);
        employee.setName(newName);
        employeeRepository.save(employee);
    }
}
```

A better approach:

```
@Service
public class EmployeeService {

    private final EmployeeRepository employeeRepository;
    // Use constructor injection

    // No self-invocation, each method has clear transaction boundaries
    @Transactional(readOnly = true)
    public Employee getEmployee(Long id) {
        return employeeRepository.findById(id)
            .orElseThrow(() -> new RuntimeException("Employee not found"));
    }

    @Transactional
    public void updateEmployee(Long id, String newName) {
        Employee employee = employeeRepository.findById(id)
            .orElseThrow(() -> new RuntimeException("Employee not found"));
        employee.setName(newName);
        employeeRepository.save(employee);
    }
}
```

Handling Exceptions in Data Operations

Spring provides a consistent approach to database exception handling through its `DataAccessException` hierarchy.

DataAccessException Hierarchy

Spring translates native database exceptions (like `SQLException`) into its own hierarchy of unchecked exceptions:

```
DataAccessException
├── CleanupFailureDataAccessException
├── ConcurrencyFailureException
│   ├── OptimisticLockingFailureException
│   ├── PessimisticLockingFailureException
├── DataAccessResourceFailureException
├── DataIntegrityViolationException
│   ├── DuplicateKeyException
├── DataRetrievalFailureException
│   ├── EmptyResultDataAccessException
├── InvalidDataAccessApiUsageException
├── InvalidDataAccessResourceUsageException
│   ├── BadSqlGrammarException
└── UncategorizedDataAccessException
```

This hierarchy makes it easier to handle database exceptions in a database-agnostic way.

Exception Translation

Spring automatically translates native database exceptions to its own hierarchy. For example:

```
@Service
public class EmployeeService {

    private final EmployeeRepository employeeRepository;

    // Constructor injection

    @Transactional
    public Employee createEmployee(Employee employee) {
        try {
            return employeeRepository.save(employee);
        } catch (DataIntegrityViolationException e) {
            // Handle constraint violations (e.g., unique email constraint)
            if (e.getCause() instanceof
org.hibernate.exception.ConstraintViolationException) {
                throw new BusinessException("Employee with this email already
exists", e);
            }
            throw e;
        }
    }
}
```

Exception Handling Best Practices

1. **Create a custom exception hierarchy**: Define your own business exception classes extending from Spring's exceptions.

```
public class BusinessException extends RuntimeException {
    public BusinessException(String message) {
        super(message);
    }

    public BusinessException(String message, Throwable cause) {
        super(message, cause);
    }
}

public class EntityNotFoundException extends BusinessException {
    public EntityNotFoundException(String entityType, Object id) {
        super(entityType + " not found with id: " + id);
    }
}
```

2. **Use @ControllerAdvice for global exception handling**:

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(EntityNotFoundException.class)
    public ResponseEntity<ErrorResponse>
handleEntityNotFound(EntityNotFoundException ex) {
        ErrorResponse error = new ErrorResponse("NOT_FOUND", ex.getMessage());
        return new ResponseEntity<>(error, HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(DataIntegrityViolationException.class)
    public ResponseEntity<ErrorResponse>
handleDataIntegrity(DataIntegrityViolationException ex) {
        ErrorResponse error = new ErrorResponse("DATA_INTEGRITY_ERROR",
            "Data integrity violation: " +
ex.getMostSpecificCause().getMessage());
        return new ResponseEntity<>(error, HttpStatus.CONFLICT);
    }

    // Other exception handlers
}
```

3. **Implement consistent error responses**:

```
public class ErrorResponse {
    private String errorCode;
    private String message;
    private LocalDateTime timestamp;

    public ErrorResponse(String errorCode, String message) {
        this.errorCode = errorCode;
```

```
        this.message = message;
        this.timestamp = LocalDateTime.now();
    }

    // Getters
}
```

Data Integrity and Constraints

Data integrity involves ensuring data accuracy and consistency throughout its lifecycle. This can be enforced at multiple levels.

Database-Level Constraints

Database constraints are rules enforced by the database:

```
@Entity
@Table(name = "employees",
       uniqueConstraints = @UniqueConstraint(columnNames = {"email"}))
public class Employee {

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

    @Column(nullable = false, length = 100)
    private String name;

    @Column(nullable = false, unique = true)
    private String email;

    // Other fields
}
```

Common database constraints:

- Primary Key constraints
- Foreign Key constraints
- Unique constraints
- Check constraints
- Not Null constraints

Application-Level Validation

Application-level validation complements database constraints:

1. **Entity Validation**: Using Bean Validation API as discussed earlier
2. **Service-Layer Validation**: Complex business rule validation in service methods
3. **Custom Validators**: For cross-field validation and business rules

Example of service-layer validation:

```
@Service
public class EmployeeService {

    // Constructor injection

    @Transactional
    public Employee promoteEmployee(Long id, String newPosition, double
salaryIncrease) {
        Employee employee = employeeRepository.findById(id)
            .orElseThrow(() -> new EntityNotFoundException("Employee", id));

        // Business rule validation
        if (employee.getHireDate().plusYears(1).isAfter(LocalDate.now())) {
            throw new BusinessException("Employee must be with the company for at
least 1 year before promotion");
        }

        if (salaryIncrease > employee.getSalary() * 0.2) {
            throw new BusinessException("Salary increase cannot exceed 20% of
current salary");
        }

        employee.setPosition(newPosition);
        employee.setSalary(employee.getSalary() + salaryIncrease);

        return employeeRepository.save(employee);
    }
}
```

Balancing Database and Application Constraints

A good strategy is to implement constraints at both levels:

1. **Critical constraints in the database**: Uniqueness, foreign keys, not null
2. **Complex business rules in the application**: Multi-field validations, conditional validations
3. **Simple validations at both levels**: For defense in depth

Practical Examples

Let's look at a comprehensive example that ties together validation, transactions, and error handling.

Complete Employee Management Service

```
package com.example.employeemanagement.service;

import com.example.employeemanagement.entity.Department;
import com.example.employeemanagement.entity.Employee;
import com.example.employeemanagement.exception.BusinessException;
```

```
import com.example.employeemanagement.exception.EntityNotFoundException;
import com.example.employeemanagement.repository.DepartmentRepository;
import com.example.employeemanagement.repository.EmployeeRepository;
import org.springframework.dao.DataIntegrityViolationException;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Isolation;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.validation.annotation.Validated;

import javax.validation.Valid;
import java.time.LocalDate;
import java.util.List;

@Service
@Validated
public class EmployeeService {

    private final EmployeeRepository employeeRepository;
    private final DepartmentRepository departmentRepository;

    public EmployeeService(EmployeeRepository employeeRepository,
                           DepartmentRepository departmentRepository) {
        this.employeeRepository = employeeRepository;
        this.departmentRepository = departmentRepository;
    }

    /**
     * Retrieve all employees with pagination
     */
    @Transactional(readOnly = true)
    public List<Employee> getAllEmployees(int page, int size) {
        return employeeRepository.findAll(PageRequest.of(page,
size)).getContent();
    }

    /**
     * Find an employee by ID
     */
    @Transactional(readOnly = true)
    public Employee getEmployeeById(Long id) {
        return employeeRepository.findById(id)
            .orElseThrow(() -> new EntityNotFoundException("Employee", id));
    }

    /**
     * Create a new employee with validation
     */
    @Transactional(propagation = Propagation.REQUIRED,
                   isolation = Isolation.READ_COMMITTED,
                   rollbackFor = Exception.class)
    public Employee createEmployee(@Valid Employee employee, String
departmentName) {
        try {
```

```
            // Validate department exists
            Department department =
departmentRepository.findByName(departmentName)
                .orElseThrow(() -> new EntityNotFoundException("Department",
departmentName));

            // Business validation
            if (employee.getSalary() < department.getMinSalary()) {
                throw new BusinessException("Salary cannot be less than department
minimum: "
                    + department.getMinSalary());
            }

            // Associate employee with department
            employee.setDepartment(department);
            employee.setHireDate(LocalDate.now());

            // Update department employee count
            department.setEmployeeCount(department.getEmployeeCount() + 1);
            departmentRepository.save(department);

            // Save employee
            return employeeRepository.save(employee);
        } catch (DataIntegrityViolationException e) {
            // Check for specific constraint violations
            if (e.getMessage().contains("email_unique")) {
                throw new BusinessException("An employee with this email already
exists", e);
            }
            throw e;
        }
    }

    /**
     * Update an existing employee
     */
    @Transactional
    public Employee updateEmployee(Long id, @Valid Employee updatedEmployee) {
        Employee existingEmployee = getEmployeeById(id);

        // Update only non-null fields
        if (updatedEmployee.getName() != null) {
            existingEmployee.setName(updatedEmployee.getName());
        }

        if (updatedEmployee.getEmail() != null) {
            existingEmployee.setEmail(updatedEmployee.getEmail());
        }

        if (updatedEmployee.getAge() > 0) {
            existingEmployee.setAge(updatedEmployee.getAge());
        }

        if (updatedEmployee.getPhoneNumber() != null) {
```

```
            existingEmployee.setPhoneNumber(updatedEmployee.getPhoneNumber());
        }

        if (updatedEmployee.getSalary() > 0) {
            existingEmployee.setSalary(updatedEmployee.getSalary());
        }

        return employeeRepository.save(existingEmployee);
    }

    /**
     * Transfer employee between departments
     */
    @Transactional
    public Employee transferEmployee(Long employeeId, String newDepartmentName) {
        Employee employee = getEmployeeById(employeeId);
        Department oldDepartment = employee.getDepartment();
        Department newDepartment =
departmentRepository.findByName(newDepartmentName)
            .orElseThrow(() -> new EntityNotFoundException("Department",
newDepartmentName));

        // Validation
        if (employee.getSalary() < newDepartment.getMinSalary()) {
            throw new BusinessException("Employee salary is below the minimum for
the new department");
        }

        // Adjust employee counts for both departments
        oldDepartment.setEmployeeCount(oldDepartment.getEmployeeCount() - 1);
        newDepartment.setEmployeeCount(newDepartment.getEmployeeCount() + 1);

        // Update employee department
        employee.setDepartment(newDepartment);

        // Save all changes
        departmentRepository.save(oldDepartment);
        departmentRepository.save(newDepartment);
        return employeeRepository.save(employee);
    }

    /**
     * Delete an employee
     */
    @Transactional
    public void deleteEmployee(Long id) {
        Employee employee = getEmployeeById(id);

        // Update department count
        Department department = employee.getDepartment();
        department.setEmployeeCount(department.getEmployeeCount() - 1);
        departmentRepository.save(department);

        // Delete employee
```

```
        employeeRepository.delete(employee);
    }

    /**
     * Apply salary increase to all employees in a department
     */
    @Transactional
    public void applyDepartmentSalaryIncrease(String departmentName, double
percentageIncrease) {
        if (percentageIncrease <= 0 || percentageIncrease > 25) {
            throw new BusinessException("Percentage increase must be between 0%
and 25%");
        }

        Department department = departmentRepository.findByName(departmentName)
            .orElseThrow(() -> new EntityNotFoundException("Department",
departmentName));

        List<Employee> employees =
employeeRepository.findByDepartment(department);

        for (Employee employee : employees) {
            double newSalary = employee.getSalary() * (1 + percentageIncrease /
100);
            employee.setSalary(newSalary);
            employeeRepository.save(employee);
        }
    }
}
```

Database Schema Design with Constraints

```
-- Department Table
CREATE TABLE departments (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL UNIQUE,
    min_salary DECIMAL(10, 2) NOT NULL,
    employee_count INT NOT NULL DEFAULT 0,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);

-- Employee Table
CREATE TABLE employees (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(100) NOT NULL UNIQUE,
    age INT NOT NULL CHECK (age >= 18 AND age <= 100),
    hire_date DATE NOT NULL,
    phone_number VARCHAR(20),
```

```
    salary DECIMAL(10, 2) NOT NULL,
    department_id BIGINT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    FOREIGN KEY (department_id) REFERENCES departments(id),
    CONSTRAINT salary_check CHECK (salary > 0)
);

-- Audit Log Table
CREATE TABLE audit_logs (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    action VARCHAR(50) NOT NULL,
    entity_type VARCHAR(50) NOT NULL,
    entity_id BIGINT NOT NULL,
    user_id VARCHAR(100) NOT NULL,
    details TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Global Exception Handler

Handling Exceptions in Data Operations (continued)

```
@ControllerAdvice
public class GlobalExceptionHandler {

    private static final Logger logger =
LoggerFactory.getLogger(GlobalExceptionHandler.class);

    @ExceptionHandler(EntityNotFoundException.class)
    public ResponseEntity<ErrorResponse>
handleEntityNotFound(EntityNotFoundException ex) {
        logger.error("Entity not found: {}", ex.getMessage());
        ErrorResponse error = new ErrorResponse("NOT_FOUND", ex.getMessage());
        return new ResponseEntity<>(error, HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(BusinessException.class)
    public ResponseEntity<ErrorResponse> handleBusinessException(BusinessException
ex) {
        logger.error("Business rule violation: {}", ex.getMessage());
        ErrorResponse error = new ErrorResponse("BUSINESS_RULE_VIOLATION",
ex.getMessage());
        return new ResponseEntity<>(error, HttpStatus.BAD_REQUEST);
    }

    @ExceptionHandler(DataIntegrityViolationException.class)
    public ResponseEntity<ErrorResponse>
handleDataIntegrityViolation(DataIntegrityViolationException ex) {
        logger.error("Data integrity violation: {}", ex.getMessage());
```

```
        String message = "Data integrity violation";
        if (ex.getCause() instanceof
org.hibernate.exception.ConstraintViolationException) {
            message = "A constraint was violated: " +
ex.getRootCause().getMessage();
        }

        ErrorResponse error = new ErrorResponse("DATA_INTEGRITY_ERROR", message);
        return new ResponseEntity<>(error, HttpStatus.CONFLICT);
    }

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<ValidationErrorResponse>
handleValidationExceptions(MethodArgumentNotValidException ex) {
        logger.error("Validation error: {}", ex.getMessage());

        ValidationErrorResponse response = new ValidationErrorResponse();
        response.setErrorCode("VALIDATION_ERROR");
        response.setMessage("Validation failed for request");

        ex.getBindingResult().getAllErrors().forEach(error -> {
            String fieldName = ((FieldError) error).getField();
            String errorMessage = error.getDefaultMessage();
            response.addFieldError(fieldName, errorMessage);
        });

        return new ResponseEntity<>(response, HttpStatus.BAD_REQUEST);
    }

    @ExceptionHandler(TransactionSystemException.class)
    public ResponseEntity<ErrorResponse>
handleTransactionSystemException(TransactionSystemException ex) {
        logger.error("Transaction failed: {}", ex.getMessage());

        // Check if there's a validation error within the transaction exception
        if (ex.getRootCause() instanceof ConstraintViolationException) {
            ConstraintViolationException validationException =
(ConstraintViolationException) ex.getRootCause();
            ValidationErrorResponse response = new ValidationErrorResponse();
            response.setErrorCode("VALIDATION_ERROR");
            response.setMessage("Validation failed during transaction");

            validationException.getConstraintViolations().forEach(violation -> {
                String propertyPath = violation.getPropertyPath().toString();
                String message = violation.getMessage();
                response.addFieldError(propertyPath, message);
            });

            return new ResponseEntity<>(response, HttpStatus.BAD_REQUEST);
        }

        ErrorResponse error = new ErrorResponse("TRANSACTION_ERROR", "Transaction
failed: " + ex.getMostSpecificCause().getMessage());
        return new ResponseEntity<>(error, HttpStatus.INTERNAL_SERVER_ERROR);
```

```
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorResponse> handleGenericException(Exception ex) {
        logger.error("Unhandled exception occurred", ex);
        ErrorResponse error = new ErrorResponse("SERVER_ERROR", "An unexpected
error occurred");
        return new ResponseEntity<>(error, HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

REST Controller with Validation

```
@RestController
@RequestMapping("/api/employees")
public class EmployeeController {

    private final EmployeeService employeeService;

    public EmployeeController(EmployeeService employeeService) {
        this.employeeService = employeeService;
    }

    @GetMapping
    public ResponseEntity<List<Employee>> getAllEmployees(
            @RequestParam(defaultValue = "0") int page,
            @RequestParam(defaultValue = "10") int size) {

        return ResponseEntity.ok(employeeService.getAllEmployees(page, size));
    }

    @GetMapping("/{id}")
    public ResponseEntity<Employee> getEmployeeById(@PathVariable Long id) {
        return ResponseEntity.ok(employeeService.getEmployeeById(id));
    }

    @PostMapping
    public ResponseEntity<Employee> createEmployee(
            @Valid @RequestBody Employee employee,
            @RequestParam String departmentName) {

        Employee createdEmployee = employeeService.createEmployee(employee,
departmentName);
        URI location = ServletUriComponentsBuilder
            .fromCurrentRequest()
            .path("/{id}")
            .buildAndExpand(createdEmployee.getId())
            .toUri();

        return ResponseEntity.created(location).body(createdEmployee);
    }
```

```
    @PutMapping("/{id}")
    public ResponseEntity<Employee> updateEmployee(
            @PathVariable Long id,
            @Valid @RequestBody Employee employee) {

        return ResponseEntity.ok(employeeService.updateEmployee(id, employee));
    }

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

    @PostMapping("/{id}/transfer")
    public ResponseEntity<Employee> transferEmployee(
            @PathVariable Long id,
            @RequestParam String departmentName) {

        Employee updatedEmployee = employeeService.transferEmployee(id,
departmentName);
        return ResponseEntity.ok(updatedEmployee);
    }

    @PatchMapping("/departments/{departmentName}/salary-increase")
    public ResponseEntity<Void> applyDepartmentSalaryIncrease(
            @PathVariable String departmentName,
            @RequestParam double percentageIncrease) {

        employeeService.applyDepartmentSalaryIncrease(departmentName,
percentageIncrease);
        return ResponseEntity.noContent().build();
    }
}
```

Combining Database and Application-level Validation

For complete data integrity, it's important to define validation at both the database and application levels. For the Employee entity, here's how we might combine both approaches:

```
@Entity
@Table(name = "employees",
       uniqueConstraints = @UniqueConstraint(name = "email_unique", columnNames =
{"email"}))
@ValidSalaryRange  // Custom class-level validation
public class Employee {

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

```
    @Column(nullable = false, length = 100)
    @NotBlank(message = "Name is required")
    @Size(min = 2, max = 100, message = "Name must be between 2 and 100
characters")
    private String name;

    @Column(nullable = false, unique = true)
    @NotBlank(message = "Email is required")
    @Email(message = "Email should be valid")
    @UniqueEmail  // Custom validation
    private String email;

    @Column(nullable = false)
    @Min(value = 18, message = "Age should not be less than 18")
    @Max(value = 100, message = "Age should not be greater than 100")
    private int age;

    @Column(nullable = false)
    @NotNull(message = "Hire date is required")
    @Past(message = "Hire date must be in the past")
    private LocalDate hireDate;

    @Column(length = 20)
    @Pattern(regexp = "^\\d{10}$", message = "Phone number must be 10 digits")
    private String phoneNumber;

    @Column(nullable = false, precision = 10, scale = 2)
    @Positive(message = "Salary must be positive")
    private BigDecimal salary;

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "department_id", nullable = false)
    @NotNull(message = "Department is required")
    private Department department;

    // Getters and setters
}
```

This approach provides multiple layers of protection:

1. **Database constraints**: NOT NULL, UNIQUE, CHECK, and Foreign Key constraints
2. **Bean Validation**: Standard annotations like @NotBlank, @Size, @Min, @Max
3. **Custom validation**: Cross-field validation with @ValidSalaryRange and unique constraints with @UniqueEmail
4. **Service-layer validation**: Additional business logic in service methods

Audit Logging for Data Changes

To maintain a complete record of changes, you can implement an audit logging system using Spring's AOP capabilities:

```java
@Aspect
@Component
public class DataAuditAspect {

    private final AuditLogRepository auditLogRepository;
    private final UserContextHolder userContextHolder;

    public DataAuditAspect(AuditLogRepository auditLogRepository,
                           UserContextHolder userContextHolder) {
        this.auditLogRepository = auditLogRepository;
        this.userContextHolder = userContextHolder;
    }

    @AfterReturning(
        pointcut = "execution(* com.example.*.repository.*.save(..)) || " +
                   "execution(* com.example.*.repository.*.delete(..))",
        returning = "result"
    )
    public void auditEntityChanges(JoinPoint joinPoint, Object result) {
        String methodName = joinPoint.getSignature().getName();
        String action = methodName.startsWith("delete") ? "DELETE" : "SAVE";

        Object entity = result;
        if (entity == null && joinPoint.getArgs().length > 0) {
            entity = joinPoint.getArgs()[0];
        }

        if (entity != null) {
            String entityType = entity.getClass().getSimpleName();
            Long entityId = getEntityId(entity);
            String userId = userContextHolder.getCurrentUserId();
            String details = generateAuditDetails(action, entity);

            AuditLog auditLog = new AuditLog();
            auditLog.setAction(action);
            auditLog.setEntityType(entityType);
            auditLog.setEntityId(entityId);
            auditLog.setUserId(userId);
            auditLog.setDetails(details);

            auditLogRepository.save(auditLog);
        }
    }

    private Long getEntityId(Object entity) {
        try {
            Method getIdMethod = entity.getClass().getMethod("getId");
            return (Long) getIdMethod.invoke(entity);
        } catch (Exception e) {
            return null;
        }
    }
```

```
    private String generateAuditDetails(String action, Object entity) {
        // Generate a JSON representation of the entity
        try {
            ObjectMapper mapper = new ObjectMapper();
            return mapper.writeValueAsString(entity);
        } catch (Exception e) {
            return "Failed to serialize entity details";
        }
    }
}
```

Transaction Management with Spring Boot Data for MongoDB

While most transaction concepts apply similarly to MongoDB (since version 4.0+), there are some differences in configuration:

```
@Configuration
@EnableMongoRepositories
public class MongoTransactionConfig {

    @Bean
    MongoTransactionManager transactionManager(MongoDatabaseFactory dbFactory) {
        return new MongoTransactionManager(dbFactory);
    }
}
```

Using transactions with MongoDB in a service:

```
@Service
public class ProductService {

    private final ProductRepository productRepository;
    private final InventoryRepository inventoryRepository;

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

    @Transactional
    public Product createProductWithInventory(Product product, Inventory
inventory) {
        // MongoDB will perform both operations in a transaction
        Product savedProduct = productRepository.save(product);

        inventory.setProductId(savedProduct.getId());
        inventoryRepository.save(inventory);
```

```
            return savedProduct;
        }
    }
```

Key Takeaways

Data Validation

1. **Multi-layered validation** provides robust data integrity:
    - Database constraints for fundamental rules
    - Bean Validation for standard field validations
    - Custom validators for complex business rules
    - Service-layer validation for contextual rules
2. **Bean Validation (JSR-380)** provides a powerful, declarative approach to validation with a rich set of annotations.
3. **Custom validators** enable complex validation scenarios, including:
    - Cross-field validation
    - Business rule validation
    - Integration with repositories for database-dependent validation
4. **Validation should be triggered** at appropriate points:
    - Controller layer for API input validation
    - Service layer for business logic validation
    - Repository layer for persistence validation

Transaction Management

1. **Declarative transactions** using `@Transactional` provide a clean, simple approach for most scenarios.
2. **Transaction attributes** allow fine-tuning:
    - Propagation for controlling transaction boundaries
    - Isolation for managing concurrency
    - Timeout for limiting long-running operations
    - Read-only for optimizing read operations
    - Rollback rules for explicit exception handling
3. **Programmatic transactions** offer maximum control for complex scenarios.
4. **Best practices** include:
    - Keeping transactions short and focused
    - Using appropriate isolation levels
    - Being mindful of transaction boundaries
    - Careful exception handling

Exception Handling

1. **Spring's DataAccessException hierarchy** provides a consistent approach to handling database-specific exceptions.
2. **Structured exception handling** with global exception handlers creates a consistent API experience.
3. **Custom exception types** improve code readability and error reporting.

Data Integrity

1. **Combining constraints at multiple levels** provides defense in depth.
2. **Audit logging** creates accountability and trackability for data changes.
3. **Consistent error responses** improve API usability and debuggability.

By implementing these patterns and practices, your Spring Boot Data applications will have robust data validation, consistent transaction management, and graceful error handling, resulting in more reliable and maintainable systems.

In a real-world application, these concepts should be applied together, creating multiple layers of protection for your data, while still maintaining clean code and separation of concerns. The examples provided in this chapter demonstrate how these techniques can be integrated into a cohesive, well-designed Spring Boot application.

Chapter 8: Performance Tuning and Optimization for Spring Boot Data Applications

8.1 Performance Bottlenecks in Data-Centric Applications - Identifying Performance Issues

Data-centric applications often face unique performance challenges that can significantly impact user experience. In Spring Boot applications, these issues typically manifest in several common patterns.

Common Performance Bottlenecks

Slow Queries

The most frequent performance issue in data-centric applications is slow database queries. These queries consume excessive resources and increase response times. They typically result from missing indexes, inefficient query structures, or retrieving more data than necessary.

Consider this simple query:

```
@Query("SELECT u FROM User u JOIN u.orders o WHERE o.status = 'COMPLETED'")
List<User> findUsersWithCompletedOrders();
```

Without proper indexing on the `status` field and a thoughtful approach to data fetching, this query could become progressively slower as your data grows.

The N+1 Problem

This insidious issue occurs when an application executes one query to retrieve a collection of parent entities, followed by N additional queries to fetch related child entities—one query per parent.

For instance, with this code:

```
List<Department> departments = departmentRepository.findAll();
for (Department department : departments) {
    // This triggers a separate query for each department
    List<Employee> employees = department.getEmployees();
    // Process employees
}
```

If you have 100 departments, this code executes 101 database queries—1 to fetch departments and 100 to fetch their employees. This pattern creates significant database load and network overhead.

Database Connection Issues

Database connection pools that are misconfigured can lead to connection starvation, where your application must wait for connections to become available. Alternatively, having too many connections can overwhelm your database server.

Caching Inefficiencies

Improper caching strategies—either caching too little (missing opportunities to avoid database calls) or caching too much (consuming excessive memory)—can diminish application performance.

Identifying Performance Issues

Recognizing these bottlenecks requires a systematic approach to monitoring and diagnostics.

Profiling Tools

Spring Boot offers integration with various profiling tools that help pinpoint performance issues:

- **Spring Boot Actuator**: Provides endpoints for monitoring applications, including database metrics.
- **Micrometer**: Spring Boot's metrics collection system that can track database operations.
- **VisualVM**: Attaches to your running JVM to provide CPU and memory profiles.

Monitoring Database Activity

Database-specific tools can reveal underperforming queries:

- **Slow Query Log**: Most databases maintain logs of queries that exceed execution time thresholds.
- **Hibernate Statistics**: Enable statistics collection to track the performance of Hibernate operations:

```
@Bean
public HibernatePropertiesCustomizer hibernateStatisticsCustomizer() {
    return hibernateProperties ->
        hibernateProperties.put("hibernate.generate_statistics", "true");
}
```

Query Execution Plans

Analyzing how your database executes queries can reveal optimization opportunities. Most databases provide tools to see execution plans, such as PostgreSQL's EXPLAIN or MongoDB's explain() method.

By collecting and analyzing these metrics, you can identify the specific areas of your data access layer that require optimization, allowing for targeted performance improvements rather than premature or unnecessary optimizations.

8.2 Query Optimization in Spring Data JPA and MongoDB

8.2.1 Spring Data JPA Query Optimization

JPA query optimization focuses on reducing database load and enhancing retrieval efficiency. Let's explore key techniques to achieve this.

Indexing

Indexes dramatically improve query performance by creating data structures that allow the database to locate rows without scanning entire tables. In Spring Data JPA applications, you can create indexes through entity annotations:

```
@Entity
@Table(name = "products", indexes = {
    @Index(name = "idx_product_name", columnList = "name"),
    @Index(name = "idx_product_category_price", columnList = "category, price")
})
public class Product {
    @Id
    private Long id;
    private String name;
    private String category;
    private BigDecimal price;
    // Other fields, getters, setters
}
```

While indexes speed up queries, they slow down write operations and consume storage space. Index judiciously, focusing on columns that appear in WHERE clauses, JOIN conditions, and ORDER BY clauses.

Fetch Joins

Fetch joins allow you to retrieve related entities in a single query, effectively addressing the N+1 problem:

```
@Query("SELECT o FROM Order o JOIN FETCH o.items WHERE o.status = :status")
List<Order> findOrdersWithItemsByStatus(@Param("status") OrderStatus status);
```

This query retrieves orders and their associated items in one database round trip, significantly reducing database load.

Entity Graphs

Entity graphs provide a more flexible way to specify which associations to fetch:

```
@EntityGraph(attributePaths = {"items", "customer"})
List<Order> findByStatus(OrderStatus status);
```

This approach instructs JPA to eagerly fetch both items and customer associations when retrieving orders.

Query Hints

JPA query hints can influence how the persistence provider executes queries:

```
@QueryHints(value = {
    @QueryHint(name = "org.hibernate.cacheable", value = "true"),
    @QueryHint(name = "org.hibernate.cacheRegion", value = "order-region")
})
List<Order> findByCustomerId(Long customerId);
```

These hints enable query caching and specify the cache region, improving performance for frequently executed queries.

JPQL/SQL Optimization

Sometimes, optimizing the query itself yields significant performance gains:

```
// Inefficient: Fetches all orders, then filters in memory
List<Order> findAll().stream()
    .filter(o -> o.getTotal().compareTo(new BigDecimal("1000")) > 0)
    .collect(Collectors.toList());

// Efficient: Filters at the database level
@Query("SELECT o FROM Order o WHERE o.total > 1000")
List<Order> findOrdersWithHighValue();
```

The second approach leverages the database's query optimizer and indexes, reducing the amount of data transferred between the database and application.

8.2.2 Spring Data MongoDB Query Optimization

MongoDB performance optimization has its own unique aspects compared to relational databases.

MongoDB Indexing

Indexes are equally crucial in MongoDB:

```
@Document(collection = "products")
@CompoundIndex(name = "category_price_idx", def = "{'category': 1, 'price': -1}")
public class Product {
    @Id
    private String id;
    @Indexed(name = "name_idx")
    private String name;
    private String category;
    private BigDecimal price;
    // Other fields, getters, setters
}
```

This creates both a single-field index on `name` and a compound index on `category` (ascending) and `price` (descending).

Query Optimization

MongoDB queries benefit from proper use of projection to limit returned fields:

```
@Query(value = "{'category': ?0}", fields = "{'name': 1, 'price': 1}")
List<Product> findByCategoryProjection(String category);
```

This query only returns the `name` and `price` fields (plus the `_id` field by default), reducing network traffic and memory usage.

Aggregation Pipeline Optimization

For complex data transformations, optimize your aggregation pipelines by placing filtering stages early:

```
@Aggregation(pipeline = {
    "{ $match: { 'status': 'ACTIVE' } }",
    "{ $lookup: { from: 'reviews', localField: '_id', foreignField: 'productId',
as: 'reviews' } }",
    "{ $project: { 'name': 1, 'avgRating': { $avg: '$reviews.rating' } } }"
})
List<ProductRatingDTO> findAverageRatingForActiveProducts();
```

By filtering with `$match` before the `$lookup` stage, we reduce the number of documents processed in subsequent stages.

Explain Plans

MongoDB's explain() method reveals how queries execute:

```
@Autowired
private MongoTemplate mongoTemplate;

public void analyzeQuery() {
    Query query = new Query(Criteria.where("category").is("electronics"));
    Document explainPlan = mongoTemplate.execute(Product.class, collection -> {
        return collection.find(query.getQueryObject()).explain();
    });
    // Analyze explain plan
}
```

The explain plan helps you determine if your query uses indexes efficiently or requires a collection scan.

8.2.3 Using Explain Plans for Query Analysis

Both relational databases and MongoDB provide tools to analyze query execution. Let's look at how to use them effectively.

JPQL/SQL Explain Plans

For JPA applications with relational databases, you can obtain explain plans:

```
@Autowired
private EntityManager entityManager;

public void analyzeJpaQuery() {
    // First, get the SQL that Hibernate will generate
    Session session = entityManager.unwrap(Session.class);
    String sql = session.createQuery("FROM Product p WHERE p.category =
:category")
        .setParameter("category", "electronics")
        .unwrap(org.hibernate.query.Query.class)
        .getQueryString();

    // Then, use native query to get explain plan
    Query explainQuery = entityManager.createNativeQuery("EXPLAIN " + sql);
    List<Object[]> explainRows = explainQuery.getResultList();
    // Analyze explain plan
}
```

For Spring Data JPA repositories, you can enable SQL logging to see the generated SQL and then manually run EXPLAIN:

```
# In application.properties
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

MongoDB Explain Plans

MongoDB's explain plan offers insights into query execution:

```
Query query = new Query(Criteria.where("price").gt(100));
ExplainVerbosity verbosity = ExplainVerbosity.EXECUTION_STATS;

Document explainOutput = mongoTemplate.execute(Product.class, collection -> {
    return collection.find(query.getQueryObject()).modifiers(
        new Document("$explain", true)
    ).first();
});

System.out.println("Execution plan: " + explainOutput.toJson());
```

The explain output shows whether MongoDB used an index or performed a collection scan, how many documents were examined versus returned, and execution time metrics.

By regularly analyzing explain plans, you can identify and address inefficient queries before they impact your application's performance in production environments.

8.3 Caching Strategies in Spring Boot Data

Caching plays a crucial role in optimizing data-centric applications by reducing database load and improving response times. Spring Boot offers several caching strategies that can be applied at different levels of your application.

8.3.1 JPA Second-Level Cache

The JPA second-level cache stores entity data across multiple sessions, reducing database queries for frequently accessed entities.

Configuring Hibernate Second-Level Cache with EhCache

First, add the required dependencies:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-ehcache</artifactId>
```

```
</dependency>
<dependency>
    <groupId>net.sf.ehcache</groupId>
    <artifactId>ehcache</artifactId>
</dependency>
```

Then configure Hibernate caching in application.properties:

```
spring.jpa.properties.hibernate.cache.use_second_level_cache=true
spring.jpa.properties.hibernate.cache.region.factory_class=org.hibernate.cache.ehc
ache.EhCacheRegionFactory
spring.jpa.properties.javax.persistence.sharedCache.mode=ENABLE_SELECTIVE
spring.jpa.properties.hibernate.cache.use_query_cache=true
```

Finally, mark entities as cacheable:

```
@Entity
@Cacheable
@org.hibernate.annotations.Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class Product {
    // Entity fields and methods
}
```

The second-level cache significantly reduces database queries for read-heavy applications that frequently access the same entities.

8.3.2 Redis Caching with Spring Data Redis

Redis provides a distributed caching solution that works well in clustered environments.

Setting up Redis Cache

Add the required dependencies:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
```

Configure Redis connection in application.properties:

```
spring.redis.host=localhost
spring.redis.port=6379
spring.cache.type=redis
spring.cache.redis.time-to-live=600000
spring.cache.redis.cache-null-values=false
```

Then create a Redis cache configuration class:

```
@Configuration
@EnableCaching
public class RedisCacheConfig {

    @Bean
    public RedisCacheManager cacheManager(RedisConnectionFactory
connectionFactory) {
        RedisCacheConfiguration config =
RedisCacheConfiguration.defaultCacheConfig()
            .entryTtl(Duration.ofMinutes(10))
            .serializeKeysWith(
                RedisSerializationContext.SerializationPair.fromSerializer(new
StringRedisSerializer()))
            .serializeValuesWith(
                RedisSerializationContext.SerializationPair.fromSerializer(
                    new GenericJackson2JsonRedisSerializer()));

        return RedisCacheManager.builder(connectionFactory)
            .cacheDefaults(config)
            .withCacheConfiguration("products",

RedisCacheConfiguration.defaultCacheConfig().entryTtl(Duration.ofMinutes(5)))
            .withCacheConfiguration("categories",

RedisCacheConfiguration.defaultCacheConfig().entryTtl(Duration.ofMinutes(30)))
            .build();
    }
}
```

This configuration creates a Redis cache manager with different TTL values for different cache regions.

8.3.3 Application-Level Caching with Spring's Caching Abstraction

Spring's caching abstraction provides a higher-level approach that can work with various cache implementations.

Setting up Spring Cache

First, enable caching in your application:

```
@SpringBootApplication
@EnableCaching
public class EcommerceApplication {
    public static void main(String[] args) {
        SpringApplication.run(EcommerceApplication.class, args);
    }
}
```

Then use caching annotations in your service classes:

```
@Service
public class ProductService {

    private final ProductRepository productRepository;

    // Constructor injection

    @Cacheable(value = "products", key = "#id")
    public Product findById(Long id) {
        // This method will be invoked only if the product isn't in the cache
        return productRepository.findById(id)
            .orElseThrow(() -> new ProductNotFoundException(id));
    }

    @CacheEvict(value = "products", key = "#product.id")
    public Product update(Product product) {
        // Update product and remove from cache
        return productRepository.save(product);
    }

    @CachePut(value = "products", key = "#result.id")
    public Product create(Product product) {
        // Create product and add to cache
        return productRepository.save(product);
    }

    @CacheEvict(value = "products", allEntries = true)
    public void refreshAllProducts() {
        // Method that requires clearing the entire products cache
    }
}
```

Spring's caching annotations provide a declarative approach to caching:

- `@Cacheable`: Caches method results based on parameters
- `@CacheEvict`: Removes entries from the cache
- `@CachePut`: Updates the cache without affecting method execution
- `@Caching`: Groups multiple caching operations

8.3.4 Choosing the Right Caching Strategy

Each caching approach has ideal use cases:

Caching Strategy	Best For	Considerations
JPA Second-Level Cache	Entity-centric applications with frequent entity lookups	Limited to JPA entities; works within a single JVM
Redis Cache	Distributed applications, microservices	Requires Redis server; excellent for distributed environments
Spring Caching Abstraction	Service-level caching, method results	Flexible, works with various cache providers

When deciding on a caching strategy, consider:

1. **Data volatility**: Frequently changing data might not benefit from caching
2. **Access patterns**: Cache data that's read frequently but updated rarely
3. **Consistency requirements**: Some applications can tolerate stale data, others cannot
4. **Scalability needs**: Multi-node applications typically require distributed caching

Remember that caching introduces complexity around cache invalidation and potential data consistency issues. Always carefully consider your caching strategy's impact on data consistency and application behavior.

8.4 Batch Operations and Bulk Data Processing

Processing large volumes of data efficiently requires special techniques to minimize database overhead and optimize resource usage. Spring Data provides several approaches for batch processing that significantly improve performance for bulk operations.

8.4.1 JPA Batch Operations

Batch Inserts and Updates

For bulk insert operations, configure batch processing in your application.properties:

```
spring.jpa.properties.hibernate.jdbc.batch_size=50
spring.jpa.properties.hibernate.order_inserts=true
spring.jpa.properties.hibernate.order_updates=true
spring.jpa.properties.hibernate.batch_versioned_data=true
```

These settings instruct Hibernate to:

- Group insert/update operations into batches of 50
- Order operations by entity type to maximize batching efficiency
- Apply batching even to versioned entities

Then use the EntityManager directly for maximum control over batching:

```
@Service
@Transactional
public class ProductBatchService {

    @PersistenceContext
    private EntityManager entityManager;

    public void batchInsertProducts(List<Product> products) {
        int batchSize = 50;

        for (int i = 0; i < products.size(); i++) {
            entityManager.persist(products.get(i));

            // Flush and clear the persistence context periodically
            if (i > 0 && i % batchSize == 0) {
                entityManager.flush();
                entityManager.clear();
            }
        }

        // Final flush for any remaining entities
        entityManager.flush();
    }
}
```

This approach prevents memory issues when processing large datasets by periodically clearing the persistence context.

Using JdbcTemplate for Maximum Performance

For extreme performance requirements, consider bypassing JPA and using Spring's JdbcTemplate:

```
@Service
public class ProductBulkImportService {

    private final JdbcTemplate jdbcTemplate;

    // Constructor injection

    public void bulkImportProducts(List<Product> products) {
        String sql = "INSERT INTO products (id, name, category, price) VALUES (?,
?, ?, ?)";

        jdbcTemplate.batchUpdate(sql, new BatchPreparedStatementSetter() {
            @Override
            public void setValues(PreparedStatement ps, int i) throws SQLException
{
                Product product = products.get(i);
                ps.setLong(1, product.getId());
                ps.setString(2, product.getName());
```

```
                ps.setString(3, product.getCategory());
                ps.setBigDecimal(4, product.getPrice());
            }

            @Override
            public int getBatchSize() {
                return products.size();
            }
        });
    }
}
```

JdbcTemplate offers better performance for bulk operations by eliminating the overhead of entity mapping and persistence context management.

8.4.2 MongoDB Bulk Operations

MongoDB provides native support for bulk operations through its BulkOperations API:

```
@Service
public class ProductBulkMongoService {

    private final MongoTemplate mongoTemplate;

    // Constructor injection

    public void bulkUpsertProducts(List<Product> products) {
        BulkOperations bulkOps =
mongoTemplate.bulkOps(BulkOperations.BulkMode.UNORDERED, Product.class);

        for (Product product : products) {
            Query query = new Query(Criteria.where("_id").is(product.getId()));
            Update update = new Update()
                .set("name", product.getName())
                .set("category", product.getCategory())
                .set("price", product.getPrice());

            bulkOps.upsert(query, update);
        }

        BulkWriteResult result = bulkOps.execute();
        log.info("Bulk operation completed: {} inserts, {} updates",
            result.getInsertedCount(), result.getModifiedCount());
    }
}
```

The `UNORDERED` mode allows MongoDB to optimize the execution order of operations for better performance. For even better performance with inserts, use the `insertAll` method:

```
public void bulkInsertProducts(List<Product> products) {
    mongoTemplate.insertAll(products);
}
```

8.4.3 Spring Batch for Complex Processing

For complex batch processing scenarios, Spring Batch provides a comprehensive framework:

```
@Configuration
@EnableBatchProcessing
public class ProductImportBatchConfig {

    @Autowired
    private JobBuilderFactory jobBuilderFactory;

    @Autowired
    private StepBuilderFactory stepBuilderFactory;

    @Autowired
    private ProductRepository productRepository;

    @Bean
    public Job importProductsJob(Step importProductsStep) {
        return jobBuilderFactory.get("importProductsJob")
            .incrementer(new RunIdIncrementer())
            .flow(importProductsStep)
            .end()
            .build();
    }

    @Bean
    public Step importProductsStep(ProductItemReader reader, ProductItemProcessor
processor) {
        return stepBuilderFactory.get("importProductsStep")
            .<ProductDTO, Product>chunk(100)
            .reader(reader)
            .processor(processor)
            .writer(products -> productRepository.saveAll(products))
            .build();
    }

    // Additional beans for reader and processor
}
```

Spring Batch handles the complex aspects of batch processing:

- Reading from various data sources (CSV, XML, databases)
- Processing items in chunks to control memory usage
- Transaction management and error handling

- Job monitoring and restart capabilities

8.4.4 Performance Considerations for Bulk Operations

When implementing bulk operations, consider these performance factors:

1. **Batch size**: Larger batches reduce overhead but increase memory usage. Find the optimal batch size through testing.
2. **Transaction management**: Use a single transaction for related operations, but beware of transaction timeouts for very large batches.
3. **Indexes and constraints**: Consider temporarily disabling non-essential indexes and constraints during massive bulk operations.
4. **Audit trail**: Disable entity listeners or audit logging during bulk operations if possible.
5. **Monitoring**: Implement progress tracking for long-running batch operations to provide visibility.

By applying these batch processing techniques, you can significantly improve performance for data-intensive operations, reducing execution time and resource consumption.

8.5 Asynchronous Data Operations

Traditional synchronous data operations can block request threads, reducing application throughput. Spring provides several ways to implement asynchronous data processing, allowing your application to handle more concurrent requests.

8.5.1 Using Spring's @Async Annotation

The `@Async` annotation enables methods to be executed in a separate thread pool:

```
@Configuration
@EnableAsync
public class AsyncConfig {

    @Bean(name = "dataTaskExecutor")
    public Executor dataTaskExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(5);
        executor.setMaxPoolSize(10);
        executor.setQueueCapacity(25);
        executor.setThreadNamePrefix("data-async-");
        executor.initialize();
        return executor;
    }
}
```

Then use the `@Async` annotation in your service methods:

```
@Service
public class ReportService {

    private final OrderRepository orderRepository;
    private final EmailService emailService;

    // Constructor injection

    @Async("dataTaskExecutor")
    public CompletableFuture<Report> generateLargeReport(ReportCriteria criteria)
{
        log.info("Generating report on thread: {}",
Thread.currentThread().getName());

        // Potentially long-running database operations
        List<Order> orders = orderRepository.findByCriteria(criteria);
        Report report = processOrders(orders);

        return CompletableFuture.completedFuture(report);
    }

    @Async("dataTaskExecutor")
    public void processOrdersAndNotify(List<Order> orders, String
notificationEmail) {
        for (Order order : orders) {
            processOrder(order);
        }

        emailService.sendNotification(notificationEmail, "Orders processed
successfully");
    }
}
```

The `@Async` annotation offloads operations to a dedicated thread pool, freeing up request-handling threads. When returning values, use `CompletableFuture<T>` to enable the caller to retrieve the result when processing completes.

8.5.2 Working with CompletableFuture

`CompletableFuture` provides a rich API for asynchronous processing:

```
@Service
public class ProductAnalysisService {

    private final ProductRepository productRepository;
    private final ReviewRepository reviewRepository;
    private final InventoryService inventoryService;

    // Constructor injection
```

```
    public CompletableFuture<ProductAnalysis> analyzeProduct(Long productId) {
        CompletableFuture<Product> productFuture =
            CompletableFuture.supplyAsync(() ->
productRepository.findById(productId).orElseThrow());

        CompletableFuture<List<Review>> reviewsFuture =
            CompletableFuture.supplyAsync(() ->
reviewRepository.findByProductId(productId));

        CompletableFuture<InventoryStatus> inventoryFuture =
            CompletableFuture.supplyAsync(() ->
inventoryService.getStatus(productId));

        return CompletableFuture.allOf(productFuture, reviewsFuture,
inventoryFuture)
            .thenApply(v -> {
                Product product = productFuture.join();
                List<Review> reviews = reviewsFuture.join();
                InventoryStatus inventory = inventoryFuture.join();

                return new ProductAnalysis(product, reviews, inventory);
            });
    }
}
```

This code fetches product information, reviews, and inventory status concurrently, then combines the results when all operations complete. The `CompletableFuture.allOf` method waits for all futures to complete, while `thenApply` transforms the results.

8.5.3 Spring Data's Asynchronous Repository Methods

Spring Data JPA supports asynchronous query methods that return `CompletableFuture`:

```
public interface OrderRepository extends JpaRepository<Order, Long> {

    @Async
    CompletableFuture<List<Order>> findByCustomerId(Long customerId);

    @Async
    CompletableFuture<List<Order>> findByStatusAndCreatedDateBetween(
        OrderStatus status, LocalDateTime start, LocalDateTime end);
}
```

Spring Data MongoDB offers similar capabilities:

```
public interface ProductRepository extends MongoRepository<Product, String> {

    @Async
    CompletableFuture<List<Product>> findByCategoryAndPriceGreaterThan(String
```

```
category, BigDecimal price);

    @Async
    CompletableFuture<Long> countByCategory(String category);
}
```

These methods automatically execute in a separate thread, making them ideal for potentially long-running queries.

8.5.4 Reactive Programming with Spring WebFlux and R2DBC

For applications with extreme scalability requirements, consider reactive programming with Spring WebFlux and R2DBC:

```
// Entity
@Table("products")
public class Product {
    @Id
    private Long id;
    private String name;
    private String category;
    private BigDecimal price;
    // Getters and setters
}

// Repository
public interface ProductRepository extends ReactiveCrudRepository<Product, Long> {
    Flux<Product> findByCategory(String category);
    Mono<Long> countByPriceGreaterThan(BigDecimal price);
}

// Service
@Service
public class ProductService {
    private final ProductRepository productRepository;

    // Constructor injection

    public Flux<ProductDTO> findProductsByCategory(String category) {
        return productRepository.findByCategory(category)
            .map(this::convertToDto)
            .doOnNext(dto -> log.info("Processed product: {}", dto.getId()));
    }

    public Mono<ProductSummary> getProductSummary() {
        Mono<Long> totalCount = productRepository.count();
        Mono<Long> premiumCount = productRepository.countByPriceGreaterThan(new
BigDecimal("100"));

        return Mono.zip(totalCount, premiumCount)
            .map(tuple -> new ProductSummary(tuple.getT1(), tuple.getT2()));
```

```
    }
}
```

Reactive programming uses a non-blocking approach that's especially beneficial for I/O-bound operations. It allows applications to handle more concurrent requests with fewer threads, potentially improving scalability.

8.5.5 Best Practices for Asynchronous Data Operations

When implementing asynchronous data processing, consider these guidelines:

1. **Thread pool sizing**: Configure thread pools based on the nature of your operations. CPU-intensive tasks benefit from a thread pool sized to match the number of CPU cores, while I/O-bound tasks can use larger pools.

2. **Error handling**: Always implement proper error handling in asynchronous code:

```
@Async
public CompletableFuture<Report> generateReport(ReportCriteria criteria) {
    try {
        // Report generation logic
        return CompletableFuture.completedFuture(report);
    } catch (Exception e) {
        log.error("Error generating report", e);
        CompletableFuture<Report> future = new CompletableFuture<>();
        future.completeExceptionally(e);
        return future;
    }
}
```

3. **Transaction management**: Be cautious with transactions in asynchronous methods, as transaction context doesn't automatically propagate to async threads.

4. **Timeout handling**: Implement timeouts to prevent resource exhaustion from long-running operations:

```
CompletableFuture<List<Product>> future =
productService.findProductsByCategory("electronics");
List<Product> products = future.orTimeout(5, TimeUnit.SECONDS).join();
```

5. **Progress tracking**: For long-running operations, consider implementing a way to track progress:

```
@Async
public CompletableFuture<ImportResult> importProducts(List<ProductDTO> products) {
    ImportProgress progress = new ImportProgress(0, products.size());
    progressRepository.save(progress);

    for (int i = 0; i < products.size(); i++) {
        // Process product
```

```
            progress.setProcessed(i + 1);
            if (i % 100 == 0) {
                progressRepository.save(progress);
            }
        }

        ImportResult result = new ImportResult(/* result details */);
        return CompletableFuture.completedFuture(result);
    }
```

Asynchronous data operations can significantly improve application responsiveness and throughput when applied appropriately. By understanding the different asynchronous processing models in Spring, you can choose the approach that best fits your specific requirements.

8.6 Connection Pooling Optimization

Database connection pools manage connections between your application and the database, significantly impacting performance. Spring Boot uses HikariCP by default, a high-performance JDBC connection pool.

8.6.1 Understanding Connection Pool Parameters

Let's examine the key configuration parameters for HikariCP:

```
# Basic connection pool configuration
spring.datasource.hikari.maximum-pool-size=10
spring.datasource.hikari.minimum-idle=5
spring.datasource.hikari.idle-timeout=600000
spring.datasource.hikari.max-lifetime=1800000
spring.datasource.hikari.connection-timeout=30000
```

Understanding these parameters is crucial for optimal tuning:

maximum-pool-size: Sets the maximum number of connections in the pool. The ideal value depends on:

- Database server capacity
- Application instance count
- Application thread count

A common formula is: `connections = (core_count * 2) + effective_spindle_count`

However, this needs adjustment based on your specific workload. Start with 10 connections per instance and monitor pool usage.

minimum-idle: Controls the minimum number of idle connections maintained in the pool. Setting this lower than maximum-pool-size allows the pool to shrink during periods of low activity, reducing resource consumption. However, maintaining some idle connections helps handle sudden traffic increases without connection establishment delays.

idle-timeout: Determines how long (in milliseconds) a connection can remain idle before being removed from the pool. The default is 10 minutes (600,000ms), which works well for most applications. Very short timeouts can cause excessive connection cycling, while very long timeouts might retain unnecessary connections.

max-lifetime: Specifies the maximum lifetime of a connection in the pool. This helps prevent issues with stale connections that might become invalid due to database timeout policies. Setting this slightly below the database's connection timeout (typically 30 minutes to 2 hours) ensures connections are refreshed before the database terminates them.

connection-timeout: Controls how long the pool will wait for a connection to become available before throwing an exception. The default of 30 seconds (30,000ms) works for most applications, but high-traffic applications might benefit from shorter timeouts to fail fast rather than waiting for unavailable connections.

8.6.2 Advanced HikariCP Configuration

For high-performance applications, consider these additional configuration options:

```
# Advanced pool settings
spring.datasource.hikari.auto-commit=true
spring.datasource.hikari.validation-timeout=5000
spring.datasource.hikari.leak-detection-threshold=60000
spring.datasource.hikari.register-mbeans=true
```

auto-commit: Controls the default auto-commit behavior for connections. Setting this to match your application's needs can reduce unnecessary transaction management overhead.

validation-timeout: Sets the maximum time to wait for connection validation queries to complete. This should be less than connection-timeout and typically represents how quickly your database can respond to simple queries.

leak-detection-threshold: Enables connection leak detection when a connection is out of the pool for longer than this value. This helps identify code that fails to properly close connections, potentially causing connection pool exhaustion.

register-mbeans: When set to true, registers management beans with the JMX server, enabling monitoring through JMX management tools.

8.6.3 Configuration via Java Code

For more programmatic control, configure the connection pool in Java:

```
@Configuration
public class DataSourceConfig {

    @Bean
    @ConfigurationProperties("spring.datasource.hikari")
    public HikariConfig hikariConfig() {
        HikariConfig config = new HikariConfig();
```

```
        // Base settings
        config.setMaximumPoolSize(20);
        config.setMinimumIdle(5);
        config.setIdleTimeout(600000);
        config.setMaxLifetime(1800000);
        config.setConnectionTimeout(30000);

        // Performance optimizations
        config.setAutoCommit(true);
        config.addDataSourceProperty("cachePrepStmts", "true");
        config.addDataSourceProperty("prepStmtCacheSize", "250");
        config.addDataSourceProperty("prepStmtCacheSqlLimit", "2048");
        config.addDataSourceProperty("useServerPrepStmts", "true");

        // Monitoring
        config.setMetricsTrackerFactory(new PrometheusMetricsTrackerFactory());

        return config;
    }

    @Bean
    public DataSource dataSource(HikariConfig hikariConfig) {
        return new HikariDataSource(hikariConfig);
    }
}
```

This approach allows you to configure database-specific optimizations and integrate custom monitoring solutions.

8.6.4 Sizing Connection Pools Appropriately

Properly sizing connection pools is critical for performance. Too few connections can cause thread contention and request queuing, while too many can overwhelm the database server.

Consider these factors when sizing your connection pool:

1. **Application thread count**: The connection pool should accommodate the maximum number of concurrent database operations. This typically correlates with the maximum thread count in your web server (e.g., Tomcat's max threads).

2. **Database capacity**: Your database server has limits on concurrent connections. Ensure your combined connection pools across all application instances don't exceed the database's capacity.

3. **Connection utilization**: Monitor the actual connection usage in your application. If connections remain mostly idle, reduce the pool size. If connections are frequently all in use, consider increasing the pool size.

4. **Transaction duration**: Applications with long-running transactions need more connections to handle the same request rate compared to applications with short transactions.

A starting point formula for pool sizing:

```
connections_per_instance = (core_count * 2) + effective_spindle_count
```

For modern systems with SSDs, a simplified approach is:

```
connections_per_instance = core_count * 2
```

For a typical 4-core server, this suggests an 8-connection pool. However, always validate through performance testing and monitoring.

8.6.5 Monitoring Connection Pool Performance

Spring Boot Actuator provides connection pool metrics that help identify sizing issues:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

Configure Actuator to expose connection pool metrics:

```
management.endpoints.web.exposure.include=health,info,metrics,prometheus
management.metrics.export.prometheus.enabled=true
```

Key metrics to monitor:

- **hikaricp.connections.active**: Currently active connections (in use)
- **hikaricp.connections.idle**: Currently idle connections
- **hikaricp.connections.pending**: Threads waiting for connections
- **hikaricp.connections.timeout**: Connection timeouts
- **hikaricp.connections.max**: Maximum number of connections
- **hikaricp.connections.min**: Minimum number of connections
- **hikaricp.connections.usage**: Connection usage time

These metrics provide insights for connection pool tuning:

- **High number of pending connections**: Indicates insufficient pool size
- **High number of timeout errors**: Suggests connection starvation
- **Consistently high active/max ratio**: Indicates the pool may be undersized
- **Consistently low active/max ratio**: Suggests the pool may be oversized

By continuously monitoring these metrics, you can adjust connection pool parameters to match your application's actual needs, ensuring optimal performance and resource utilization.

8.7 Monitoring Data Access Performance

Effective monitoring is essential for identifying and resolving performance issues in data-centric applications. Spring Boot provides robust tools for monitoring every aspect of data access performance.

8.7.1 Spring Boot Actuator for Data Metrics

Spring Boot Actuator exposes comprehensive metrics about your application's data access layer.

First, configure Actuator with appropriate endpoints:

```
# Actuator configuration
management.endpoints.web.exposure.include=health,info,metrics,prometheus
management.endpoint.health.show-details=always
management.endpoint.metrics.enabled=true
management.metrics.export.prometheus.enabled=true
```

This configuration enables the metrics endpoint and Prometheus export, facilitating integration with monitoring systems.

8.7.2 Database-Specific Metrics

Spring Boot provides specialized metrics for different database technologies:

JDBC Metrics

For JDBC-based data access, including JPA:

```
# Enable JDBC metrics
spring.datasource.hikari.register-mbeans=true
```

This exposes metrics including:

- **jdbc.connections.active**: Currently active connections
- **jdbc.connections.max**: Maximum number of connections
- **jdbc.connections.min**: Minimum number of connections
- **jdbc.statements**: Statement execution statistics
- **hikaricp.connections.usage**: Connection usage time distribution

MongoDB Metrics

For MongoDB applications:

```
# Enable MongoDB metrics
management.metrics.mongo.command.enabled=true
management.metrics.mongo.connectionpool.enabled=true
```

These settings expose metrics such as:

- **mongodb.driver.commands**: Command execution time and counts
- **mongodb.driver.connections**: Connection pool statistics
- **mongodb.driver.server**: Server latency

8.7.3 Hibernate-Specific Monitoring

For JPA applications using Hibernate, enable statistics collection:

```
# Hibernate statistics
spring.jpa.properties.hibernate.generate_statistics=true
spring.jpa.properties.hibernate.session.events.log.LOG_QUERIES_SLOWER_THAN_MS=25
```

This configuration enables Hibernate's built-in statistics and logs queries taking longer than 25ms. To programmatically access these statistics:

```
@Service
public class HibernateMonitoringService {

    @PersistenceUnit
    private EntityManagerFactory entityManagerFactory;

    public HibernateStatistics getStatistics() {
        SessionFactory sessionFactory =
entityManagerFactory.unwrap(SessionFactory.class);
        Statistics statistics = sessionFactory.getStatistics();

        return new HibernateStatistics(
            statistics.getQueryExecutionCount(),
            statistics.getQueryExecutionMaxTime(),
            statistics.getQueryExecutionMaxTimeQueryString(),
            statistics.getEntityFetchCount(),
            statistics.getSecondLevelCacheHitCount(),
            statistics.getSecondLevelCacheMissCount()
        );
    }
}
```

This service extracts valuable information about query performance, entity fetching, and cache effectiveness.

8.7.4 SQL Logging and Analysis

For deeper insight into SQL queries, configure SQL logging:

```
# SQL logging
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
```

For production environments, selective SQL logging is more appropriate:

```
@Aspect
@Component
public class SlowQueryLogger {

    private static final Logger log = LoggerFactory.getLogger("slow-query-log");
    private final int thresholdMs = 100; // Log queries slower than 100ms

    @Around("execution(* org.springframework.jdbc.core.JdbcTemplate.*(..))")
    public Object logSlowQueries(ProceedingJoinPoint joinPoint) throws Throwable {
        Object[] args = joinPoint.getArgs();
        String query = args.length > 0 && args[0] instanceof String ? (String)
args[0] : "";

        long start = System.currentTimeMillis();
        Object result = joinPoint.proceed();
        long executionTime = System.currentTimeMillis() - start;

        if (executionTime > thresholdMs) {
            log.warn("Slow query detected ({}ms): {}", executionTime, query);
        }

        return result;
    }
}
```

This aspect logs only queries that exceed a performance threshold, avoiding excessive logging while capturing problematic queries.

8.7.5 Integration with Monitoring Systems

For production environments, integrate with comprehensive monitoring systems:

Prometheus and Grafana

Create a Prometheus configuration to scrape Spring Boot metrics:

```
# prometheus.yml
scrape_configs:
```

```
  - job_name: 'spring-boot-app'
    metrics_path: '/actuator/prometheus'
    static_configs:
      - targets: ['app-host:8080']
```

Then configure Grafana dashboards to visualize:

- Connection pool utilization
- Query execution times
- Cache hit/miss ratios
- Transaction statistics

Application Performance Monitoring (APM) Tools

APM tools like New Relic, Dynatrace, or Elastic APM provide deeper insights into data access performance:

```
<!-- Example for Elastic APM integration -->
<dependency>
    <groupId>co.elastic.apm</groupId>
    <artifactId>elastic-apm-agent</artifactId>
    <version>1.34.1</version>
</dependency>
```

APM tools automatically trace database operations, showing:

- Individual query performance
- Transaction traces including database operations
- Connection pool utilization
- Database server health

8.7.6 Custom Monitoring Solutions

For specialized requirements, implement custom monitoring:

```
@Component
public class DatabaseMetricsCollector {

    private final JdbcTemplate jdbcTemplate;
    private final MeterRegistry meterRegistry;

    // Constructor injection

    @Scheduled(fixedRate = 60000) // Every minute
    public void collectDatabaseMetrics() {
        // Example: Monitor table sizes
        try {
            jdbcTemplate.query(
                "SELECT table_name, pg_total_relation_size(table_name) AS size
```

```
FROM information_schema.tables WHERE table_schema = 'public'",
                rs -> {
                    String tableName = rs.getString("table_name");
                    long sizeBytes = rs.getLong("size");
                    meterRegistry.gauge("database.table.size", Tags.of("table",
tableName), sizeBytes);
                }
            );
        } catch (Exception e) {
            log.error("Error collecting table size metrics", e);
        }

        // Example: Monitor index usage
        try {
            jdbcTemplate.query(
                "SELECT indexrelname, idx_scan, idx_tup_read FROM
pg_stat_user_indexes",
                rs -> {
                    String indexName = rs.getString("indexrelname");
                    long scans = rs.getLong("idx_scan");
                    meterRegistry.gauge("database.index.scans", Tags.of("index",
indexName), scans);
                }
            );
        } catch (Exception e) {
            log.error("Error collecting index usage metrics", e);
        }
    }
}
```

This example periodically collects database-specific metrics like table sizes and index usage statistics, exposing them through Spring's metrics system.

By implementing comprehensive monitoring for your data access layer, you gain visibility into performance bottlenecks and can make informed decisions about optimizations. Regular analysis of these metrics helps identify trends and potential issues before they impact users.

8.8 Performance Testing Data Access Layers

Performance testing plays a critical role in ensuring your data access layer can handle expected loads efficiently. A systematic approach to performance testing helps identify bottlenecks and validate optimizations.

8.8.1 Building a Testing Strategy

An effective performance testing strategy for data access layers includes:

1. **Unit Testing**: Test individual repository methods with performance assertions
2. **Integration Testing**: Test the interaction between service and repository layers
3. **Load Testing**: Simulate realistic user loads to identify bottlenecks
4. **Endurance Testing**: Run extended tests to identify memory leaks or degradation

5. **Profiling**: Analyze CPU, memory, and database usage during tests

8.8.2 Unit Testing for Performance

Unit tests with performance assertions help catch regressions early:

```
@SpringBootTest
@TestPropertySource(properties = {
    "spring.jpa.properties.hibernate.generate_statistics=true"
})
public class ProductRepositoryPerformanceTest {

    @Autowired
    private ProductRepository productRepository;

    @Autowired
    private EntityManagerFactory entityManagerFactory;

    @Test
    public void testFindByCategory_Performance() {
        // Prepare test data
        // ...

        // Reset statistics
        Statistics stats =
entityManagerFactory.unwrap(SessionFactory.class).getStatistics();
        stats.clear();

        // Execute the query
        Stopwatch stopwatch = Stopwatch.createStarted();
        List<Product> products = productRepository.findByCategory("electronics");
        stopwatch.stop();

        // Assert on execution time
        assertThat(stopwatch.elapsed(TimeUnit.MILLISECONDS)).isLessThan(50);

        // Assert on query count - check for N+1 problems
        assertThat(stats.getQueryExecutionCount()).isEqualTo(1);

        // Assert on entity load count
        assertThat(stats.getEntityLoadCount()).isEqualTo(products.size());
    }
}
```

This test verifies that the query executes within an acceptable time frame and doesn't generate excessive database queries, which could indicate an N+1 problem.

8.8.3 Integration Testing with Performance Metrics

Integration tests evaluate the entire data access flow:

```
@SpringBootTest
@TestPropertySource(properties = {
    "spring.jpa.properties.hibernate.generate_statistics=true"
})
public class ProductServicePerformanceTest {

    @Autowired
    private ProductService productService;

    @Autowired
    private EntityManagerFactory entityManagerFactory;

    @Test
    public void testProductSearch_Performance() {
        // Reset statistics
        Statistics stats =
entityManagerFactory.unwrap(SessionFactory.class).getStatistics();
        stats.clear();

        // Execute the service method
        Stopwatch stopwatch = Stopwatch.createStarted();
        Page<ProductDTO> results = productService.searchProducts(
            ProductSearchCriteria.builder()
                .category("electronics")
                .priceMin(new BigDecimal("100"))
                .priceMax(new BigDecimal("500"))
                .page(0)
                .size(20)
                .build()
        );
        stopwatch.stop();

        // Assert on execution time
        assertThat(stopwatch.elapsed(TimeUnit.MILLISECONDS)).isLessThan(100);

        // Assert on query count
        assertThat(stats.getQueryExecutionCount()).isLessThan(3);

        // Assert on cache utilization if applicable
        assertThat(stats.getSecondLevelCacheHitCount() +
stats.getQueryCacheHitCount())
            .isGreaterThan(0);
    }
}
```

This test evaluates the entire service method, ensuring it meets performance requirements and effectively utilizes caching.

8.8.4 Load Testing with JMeter

JMeter provides a powerful framework for load testing data access operations:

1. **Create a Test Plan**: Define the load test structure
2. **Add Thread Groups**: Specify the number of concurrent users
3. **Configure HTTP Requests**: Set up endpoints that exercise your data access layer
4. **Add Listeners**: Capture response times and throughput
5. **Configure Assertions**: Define acceptable performance thresholds

A typical JMeter test plan for a Spring Boot application looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<jmeterTestPlan version="1.2" properties="5.0">
  <hashTree>
    <TestPlan guiclass="TestPlanGui" testclass="TestPlan" testname="Spring Boot
Data Load Test">
      <elementProp name="TestPlan.user_defined_variables" elementType="Arguments">
        <collectionProp name="Arguments.arguments"/>
      </elementProp>
    </TestPlan>
    <hashTree>
      <ThreadGroup guiclass="ThreadGroupGui" testclass="ThreadGroup"
testname="Product API Users">
        <intProp name="ThreadGroup.num_threads">100</intProp>
        <intProp name="ThreadGroup.ramp_time">30</intProp>
        <boolProp name="ThreadGroup.scheduler">true</boolProp>
        <stringProp name="ThreadGroup.duration">300</stringProp>
        <stringProp name="ThreadGroup.delay">0</stringProp>
      </ThreadGroup>
      <hashTree>
        <HTTPSamplerProxy guiclass="HttpTestSampleGui"
testclass="HTTPSamplerProxy" testname="Get Products">
          <stringProp name="HTTPSampler.path">/api/products</stringProp>
          <stringProp name="HTTPSampler.method">GET</stringProp>
          <boolProp name="HTTPSampler.follow_redirects">true</boolProp>
          <stringProp name="HTTPSampler.protocol">http</stringProp>
          <stringProp name="HTTPSampler.domain">localhost</stringProp>
          <stringProp name="HTTPSampler.port">8080</stringProp>
        </HTTPSamplerProxy>
        <hashTree>
          <ResponseAssertion guiclass="AssertionGui"
testclass="ResponseAssertion">
            <collectionProp name="Asserion.test_strings">
              <stringProp name="49586">200</stringProp>
            </collectionProp>
            <stringProp
name="Assertion.test_field">Assertion.response_code</stringProp>
            <boolProp name="Assertion.assume_success">false</boolProp>
            <intProp name="Assertion.test_type">8</intProp>
          </ResponseAssertion>
          <hashTree/>
          <DurationAssertion guiclass="DurationAssertionGui"
testclass="DurationAssertion">
            <stringProp name="DurationAssertion.duration">500</stringProp>
          </DurationAssertion>
```

```
          <hashTree/>
        </hashTree>
        <!-- Additional HTTP requests for other endpoints -->
      </hashTree>
      <ResultCollector guiclass="SummaryReport" testclass="ResultCollector"
testname="Summary Report"/>
      <hashTree/>
      <ResultCollector guiclass="GraphVisualizer" testclass="ResultCollector"
testname="Graph Results"/>
      <hashTree/>
    </hashTree>
  </hashTree>
</jmeterTestPlan>
```

This test simulates 100 concurrent users accessing the product API over a 5-minute period, with assertions checking for successful responses within 500ms.

8.8.5 Profiling During Performance Tests

Profiling tools provide insights into system behavior during performance tests:

JVM Profiling with VisualVM

1. Launch VisualVM and connect to your application JVM
2. Run your performance tests
3. Analyze CPU, memory, and thread profiles
4. Identify hotspots in your data access code

Look for patterns such as:

- Methods with high CPU usage
- Excessive object creation
- Thread contention on database connections
- Memory leaks from improperly closed resources

Database Profiling

Most databases offer profiling tools to monitor query performance:

- **PostgreSQL**: pg_stat_statements extension
- **MySQL**: Performance Schema
- **MongoDB**: Database Profiler

Configure these tools during performance tests to identify slow queries:

```
-- PostgreSQL example: Enable query statistics
CREATE EXTENSION IF NOT EXISTS pg_stat_statements;

-- After test, analyze queries
SELECT query, calls, total_time, mean_time, rows
```

```
FROM pg_stat_statements
ORDER BY mean_time DESC
LIMIT 20;
```

8.8.6 Automated Performance Testing

Incorporate performance tests into your CI/CD pipeline for continuous performance monitoring:

```
# Jenkins pipeline example
pipeline {
    agent any
    stages {
        // Other stages...
        stage('Performance Test') {
            steps {
                sh 'jmeter -n -t tests/jmeter/product-api-test.jmx -l results.jtl'
                perfReport 'results.jtl'
            }
            post {
                always {
                    archiveArtifacts artifacts: 'results.jtl', fingerprint: true
                }
                unstable {
                    echo 'Performance tests failed performance thresholds'
                }
            }
        }
    }
}
```

This pipeline executes JMeter tests and reports results, marking the build as unstable if performance thresholds aren't met.

8.8.7 Realistic Test Data

Performance tests require realistic data volumes and distributions:

```
@Profile("perf-test")
@Component
public class RealisticTestDataGenerator implements ApplicationRunner {

    private final ProductRepository productRepository;
    private final CategoryRepository categoryRepository;
    private final ReviewRepository reviewRepository;
    private final Random random = new Random();

    // Constructor injection

    @Override
```

```
    public void run(ApplicationArguments args) throws Exception {
        long existingProducts = productRepository.count();
        if (existingProducts > 0) {
            log.info("Test data already exists, skipping generation");
            return;
        }

        log.info("Generating realistic test data...");

        // Generate categories
        List<Category> categories = generateCategories(20);

        // Generate products (100,000)
        List<Product> products = new ArrayList<>();
        for (int i = 0; i < 100_000; i++) {
            Product product = new Product();
            product.setName("Product " + i);

product.setCategory(categories.get(random.nextInt(categories.size())));
            product.setPrice(new BigDecimal(10 + random.nextInt(990)));
            product.setActive(random.nextDouble() > 0.1); // 90% active

            products.add(product);

            if (i % 5000 == 0) {
                productRepository.saveAll(products);
                products.clear();
                log.info("Generated {} products", i);
            }
        }

        // Generate reviews (500,000)
        // ...

        log.info("Test data generation complete");
    }

    private List<Category> generateCategories(int count) {
        List<Category> categories = new ArrayList<>();
        for (int i = 0; i < count; i++) {
            Category category = new Category();
            category.setName("Category " + i);
            categories.add(category);
        }
        return categoryRepository.saveAll(categories);
    }
}
```

This generator creates a realistic dataset with proper relationships and data distributions, allowing performance tests to accurately reflect production conditions.

By implementing a comprehensive performance testing strategy, you can identify and address data access performance issues before they impact production systems. Regular performance testing also helps validate optimizations and ensure your application meets its performance requirements as data volumes grow.

8.9 Best Practices for Performance Tuning Data-Centric Spring Boot Applications

After exploring specific optimization techniques, let's consolidate key best practices for building high-performance data-centric Spring Boot applications.

8.9.1 Data Access Layer Design

Start with a Well-Designed Data Model

- Normalize data appropriately, but don't over-normalize
- Design entities with performance in mind (consider fetch type, cascade operations)
- Use appropriate data types (e.g., use UUID instead of String for IDs if needed)
- Plan index strategy early based on anticipated query patterns

Repository Design

- Create focused repositories with specific query methods
- Avoid generic repositories that expose too many operations
- Consider using the Command Query Responsibility Segregation (CQRS) pattern for complex applications
- Create separate read and write repositories for tables with dramatically different access patterns

```
public interface ProductReadRepository extends Repository<Product, Long> {
    Optional<Product> findById(Long id);
    List<Product> findByCategory(String category);
    Page<Product> findAll(Specification<Product> spec, Pageable pageable);
}

public interface ProductWriteRepository extends Repository<Product, Long> {
    <S extends Product> S save(S entity);
    <S extends Product> List<S> saveAll(Iterable<S> entities);
    void deleteById(Long id);
}
```

This separation allows different optimization strategies for read and write operations.

8.9.2 Query Optimization Best Practices

Write Efficient Queries

- Use specific queries instead of finding by example or criteria when possible
- Return only the data you need (DTOs, projections, custom result sets)
- Implement pagination for large result sets
- Use query hints strategically for cache control or optimizer hints

Leverage Native SQL When Appropriate

- Use native queries for complex operations that don't translate well to JPQL
- Consider stored procedures for database-intensive operations

```
@Query(value = "SELECT p.id, p.name, AVG(r.rating) as avg_rating " +
               "FROM products p " +
               "JOIN reviews r ON p.id = r.product_id " +
               "WHERE p.category = :category " +
               "GROUP BY p.id, p.name " +
               "HAVING AVG(r.rating) >= :minRating " +
               "ORDER BY avg_rating DESC " +
               "LIMIT :limit",
       nativeQuery = true)
List<ProductRatingProjection> findTopRatedProductsInCategory(
    @Param("category") String category,
    @Param("minRating") double minRating,
    @Param("limit") int limit);
```

8.9.3 Caching Strategy

Implement a Multi-Level Caching Strategy

- Use the JPA second-level cache for frequently accessed, rarely changing entities
- Implement application-level caching for service method results
- Consider distributed caching (Redis, Hazelcast) for clustered environments
- Cache query results, not just entities

Cache Management

- Implement proper cache eviction strategies
- Set appropriate TTL for cached items based on data volatility
- Monitor cache hit rates and adjust caching strategy accordingly
- Use cache conditions to prevent caching inappropriate data

```
@Cacheable(
    value = "productSummaries",
    key = "#category + '-' + #page + '-' + #size",
    condition = "#page < 10 && #category != 'clearance'"
)
public Page<ProductSummary> getProductSummaries(String category, int page, int
size) {
    // Method implementation
}
```

8.9.4 Transaction Management

Optimize Transaction Boundaries

- Keep transactions as short as possible
- Avoid executing non-database operations within transactions
- Use read-only transactions for queries that don't modify data
- Consider transaction isolation levels based on your application's needs

```
@Service
public class OrderService {

    private final OrderRepository orderRepository;
    private final PaymentService paymentService;
    private final NotificationService notificationService;

    // Constructor injection

    @Transactional
    public Order createOrder(OrderRequest request) {
        // Database operations only within transaction
        Order order = new Order();
        order.setCustomer(request.getCustomerId());
        order.setItems(mapItems(request.getItems()));
        order.setStatus(OrderStatus.CREATED);

        return orderRepository.save(order);
    }

    // Non-transactional method for operations after the transaction
    public void processCreatedOrder(Order order) {
        // Non-database operations outside transaction
        paymentService.processPayment(order);
        notificationService.sendOrderConfirmation(order);
    }

    @Transactional(readOnly = true)
    public Order getOrderDetails(Long orderId) {
        return orderRepository.findByIdWithDetails(orderId)
            .orElseThrow(() -> new OrderNotFoundException(orderId));
    }
}
```

8.9.5 Batch Processing and Pagination

Implement Efficient Batch Processing

- Process large datasets in chunks
- Use JPA batch insert/update settings
- Consider using JdbcTemplate for very large operations
- Implement pagination for large result sets

```
@Service
public class DataImportService {

    private final ProductRepository productRepository;
    private final JdbcTemplate jdbcTemplate;

    // Constructor injection

    public void importProducts(InputStream dataStream) {
        try (CSVReader reader = new CSVReader(new InputStreamReader(dataStream)))
{
            List<Product> batch = new ArrayList<>(1000);
            String[] line;
            int count = 0;

            while ((line = reader.readNext()) != null) {
                Product product = mapToProduct(line);
                batch.add(product);
                count++;

                if (batch.size() >= 1000) {
                    productRepository.saveAll(batch);
                    batch.clear();
                    log.info("Imported {} products", count);
                }
            }

            if (!batch.isEmpty()) {
                productRepository.saveAll(batch);
            }

            log.info("Import completed, total products: {}", count);
        } catch (IOException e) {
            throw new ImportException("Failed to import products", e);
        }
    }

    private Product mapToProduct(String[] data) {
        // Mapping logic
    }
}
```

Implement Pagination for APIs

- Always paginate list endpoints
- Use appropriate page sizes (typically 20-50 items)
- Support sorting parameters
- Consider using keyset pagination for high-performance scenarios

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

    private final ProductService productService;

    // Constructor injection

    @GetMapping
    public Page<ProductDTO> getProducts(
            @RequestParam(defaultValue = "0") int page,
            @RequestParam(defaultValue = "20") int size,
            @RequestParam(defaultValue = "id") String sortBy,
            @RequestParam(defaultValue = "asc") String direction) {

        Sort sort = direction.equalsIgnoreCase("asc") ?
            Sort.by(sortBy).ascending() :
            Sort.by(sortBy).descending();

        Pageable pageable = PageRequest.of(page, size, sort);
        return productService.findProducts(pageable);
    }
}
```

8.9.6 Connection and Resource Management

Optimize Database Connection Usage

Connection management is critical for application performance. Here's how to optimize connection usage:

- Configure connection pool size based on your application's concurrent database operation requirements
- Monitor connection usage and adjust pool settings based on actual usage patterns
- Set appropriate timeout values for idle connections to release resources when not needed
- Implement connection validation to detect and replace stale connections

```
@Configuration
public class DataSourceConfiguration {

    @Bean
    @ConfigurationProperties("spring.datasource.hikari")
    public HikariConfig hikariConfig() {
        HikariConfig config = new HikariConfig();

        // Set reasonable defaults based on core count
        int cpuCores = Runtime.getRuntime().availableProcessors();
        config.setMaximumPoolSize(cpuCores * 4);  // Rule of thumb starting point
        config.setMinimumIdle(cpuCores);

        // Set appropriate timeouts
```

```
        config.setIdleTimeout(300000);  // 5 minutes
        config.setMaxLifetime(1800000); // 30 minutes

        // Enable leak detection
        config.setLeakDetectionThreshold(60000); // 1 minute

        return config;
    }

    @Bean
    public DataSource dataSource(HikariConfig hikariConfig) {
        return new HikariDataSource(hikariConfig);
    }
}
```

Resource Management Best Practices

- Always close resources explicitly in non-Spring-managed code
- Use try-with-resources for streams, result sets, and other closeable resources
- Release large objects when no longer needed to assist garbage collection
- Clear collections when they're no longer needed

```
@Service
public class ReportService {

    private final JdbcTemplate jdbcTemplate;

    // Constructor injection

    public byte[] generateLargeReport() {
        List<ReportRow> reportData = new ArrayList<>();

        try (Connection conn = jdbcTemplate.getDataSource().getConnection();
             PreparedStatement stmt = conn.prepareStatement("SELECT * FROM
report_data");
             ResultSet rs = stmt.executeQuery()) {

            while (rs.next()) {
                reportData.add(mapResultSetToReportRow(rs));
            }

            // Generate report from data
            byte[] reportBytes = createPdfReport(reportData);

            // Clear large collection when no longer needed
            reportData.clear();

            return reportBytes;
        } catch (SQLException | IOException e) {
            throw new ReportGenerationException("Failed to generate report", e);
        }
```

```
    }

    private ReportRow mapResultSetToReportRow(ResultSet rs) throws SQLException {
        // Mapping logic
    }

    private byte[] createPdfReport(List<ReportRow> data) throws IOException {
        // PDF generation logic
    }
}
```

8.9.7 Data Access Layer Testing

Implement Comprehensive Testing

- Write unit tests for repository methods
- Create integration tests for repository interaction with the database
- Test edge cases like empty results and large result sets
- Include performance assertions in tests

```
@DataJpaTest
@TestPropertySource(properties = {
    "spring.jpa.hibernate.ddl-auto=create-drop",
    "spring.datasource.url=jdbc:h2:mem:testdb"
})
public class ProductRepositoryTest {

    @Autowired
    private ProductRepository productRepository;

    @Autowired
    private TestEntityManager entityManager;

    @Test
    public void testFindByCategoryPerformance() {
        // Create test data
        for (int i = 0; i < 100; i++) {
            Product product = new Product();
            product.setName("Product " + i);
            product.setCategory(i % 5 == 0 ? "electronics" : "clothing");
            product.setPrice(new BigDecimal("99.99"));
            entityManager.persist(product);
        }
        entityManager.flush();

        // Test performance
        long startTime = System.nanoTime();
        List<Product> products = productRepository.findByCategory("electronics");
        long endTime = System.nanoTime();

        assertThat(products).hasSize(20);
```

```
        // Assert query executes within reasonable time
        long executionTimeMs = TimeUnit.NANOSECONDS.toMillis(endTime - startTime);
        assertThat(executionTimeMs).isLessThan(100);
    }

    @Test
    public void testFindByNameWithMultipleResults() {
        // Test with multiple matching results
    }

    @Test
    public void testFindByNameWithNoResults() {
        // Test with no matching results
    }
}
```

Test with Realistic Data Volume

- Create test fixtures that simulate production data volumes
- Test with different data distributions
- Verify performance with varying result sizes

8.9.8 Monitoring and Observability

Implement Comprehensive Monitoring

- Track key metrics for database operations
- Monitor connection pool usage
- Log slow queries
- Implement health checks for database connections

```
@Configuration
public class DataAccessMonitoringConfig {

    @Bean
    public FilterRegistrationBean<SlowQueryFilter> slowQueryFilter() {
        FilterRegistrationBean<SlowQueryFilter> registrationBean = new
FilterRegistrationBean<>();

        SlowQueryFilter slowQueryFilter = new SlowQueryFilter();
        registrationBean.setFilter(slowQueryFilter);
        registrationBean.addUrlPatterns("/api/*");

        return registrationBean;
    }

    @Bean
    public HealthIndicator databaseHealthIndicator(DataSource dataSource) {
        return new DataSourceHealthIndicator(dataSource, "SELECT 1");
    }
```

```
}

@Component
public class SlowQueryFilter extends OncePerRequestFilter {

    private static final Logger log =
LoggerFactory.getLogger(SlowQueryFilter.class);

    @Override
    protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response,
                                    FilterChain filterChain) throws
ServletException, IOException {
        long startTime = System.currentTimeMillis();

        try {
            filterChain.doFilter(request, response);
        } finally {
            long executionTime = System.currentTimeMillis() - startTime;

            if (executionTime > 1000) { // Log requests taking more than 1 second
                log.warn("Slow request detected: {} {} - {}ms",
                         request.getMethod(), request.getRequestURI(),
executionTime);
            }
        }
    }
}
```

Centralized Logging and Analysis

- Implement structured logging for database operations
- Integrate with log aggregation systems (ELK, Graylog, Splunk)
- Create dashboards for key database performance metrics
- Set up alerts for abnormal patterns

8.9.9 Performance Optimization Checklist

Before deploying your data-centric Spring Boot application to production, verify these optimization points:

1. **Database Schema**
    - Appropriate normalization level
    - Proper indexes for common queries
    - Optimized data types
    - Foreign key constraints
2. **Entity Mapping**
    - Lazy loading for non-essential relationships
    - Fetch joins for commonly accessed relationships
    - Appropriate cascade types

- Optimized entity hierarchies

3. **Repository Layer**
    - Specific query methods for common operations
    - Projections or DTOs for complex queries
    - Pagination for list endpoints
    - Optimized JPQL or native queries for complex operations
4. **Caching Strategy**
    - Second-level cache for appropriate entities
    - Query cache for frequent queries
    - Application-level caching for computed results
    - Proper cache eviction strategies
5. **Connection Pool Configuration**
    - Properly sized connection pool
    - Appropriate timeouts
    - Connection validation enabled
    - Leak detection configured
6. **Transaction Management**
    - Appropriate transaction boundaries
    - Read-only transactions for queries
    - Proper isolation levels
    - Transaction timeout settings
7. **Monitoring and Alerts**
    - Connection pool metrics
    - Query performance metrics
    - Slow query logging
    - Database health checks
8. **Performance Testing**
    - Load testing for common scenarios
    - Endurance testing for memory leaks
    - Integration with CI/CD pipeline
    - Alerting for performance regressions

8.10 Scaling Data Access for High-Volume Applications

As your application grows, you'll need strategies to scale your data access layer to handle increased loads.

8.10.1 Vertical Scaling Strategies

Database Server Optimization

- Allocate appropriate hardware resources (CPU, RAM, disk I/O)
- Optimize database configuration for your workload
- Use SSDs for improved I/O performance
- Implement proper database maintenance (vacuum, reindex)

Application Server Optimization

- Increase connection pool size proportionally to available hardware
- Adjust JVM settings for optimal performance
- Scale vertically by adding more resources to your application servers

```
@Configuration
public class DataSourceScalingConfig {

    @Value("${server.tomcat.threads.max:200}")
    private int maxThreads;

    @Bean
    @ConfigurationProperties("spring.datasource.hikari")
    public HikariConfig hikariConfig() {
        HikariConfig config = new HikariConfig();

        // Scale connection pool with available threads
        // A good rule of thumb is 10-20% of max threads
        int poolSize = Math.max(10, maxThreads / 5);
        config.setMaximumPoolSize(poolSize);

        return config;
    }
}
```

8.10.2 Horizontal Scaling Strategies

Read Replicas

- Direct read operations to database replicas
- Keep write operations on the primary database
- Implement read/write splitting in your data access layer

```
@Configuration
public class ReadWriteDataSourceConfig {

    @Bean
    @Primary
    @ConfigurationProperties("spring.datasource.write")
    public DataSource writeDataSource() {
        return DataSourceBuilder.create().build();
    }
```

```
    @Bean
    @ConfigurationProperties("spring.datasource.read")
    public DataSource readDataSource() {
        return DataSourceBuilder.create().build();
    }

    @Bean
    public DataSource routingDataSource(
            @Qualifier("writeDataSource") DataSource writeDataSource,
            @Qualifier("readDataSource") DataSource readDataSource) {

        ReplicaAwareRoutingDataSource routingDataSource = new
ReplicaAwareRoutingDataSource();

        Map<Object, Object> dataSources = new HashMap<>();
        dataSources.put("write", writeDataSource);
        dataSources.put("read", readDataSource);

        routingDataSource.setTargetDataSources(dataSources);
        routingDataSource.setDefaultTargetDataSource(writeDataSource);

        return routingDataSource;
    }

    public class ReplicaAwareRoutingDataSource extends AbstractRoutingDataSource {
        @Override
        protected Object determineCurrentLookupKey() {
            return
TransactionSynchronizationManager.isCurrentTransactionReadOnly() ?
                "read" : "write";
        }
    }
}
```

Database Sharding

- Partition data across multiple database instances
- Implement a sharding strategy based on your data access patterns
- Use consistent hashing for even distribution

```
@Configuration
public class ShardingDataSourceConfig {

    @Bean
    public ShardingDataSource shardingDataSource() {
        Map<String, DataSource> dataSourceMap = createDataSourceMap();

        ShardingRuleConfiguration shardingRuleConfig = new
ShardingRuleConfiguration();

        // Configure table sharding rules
```

```
shardingRuleConfig.getTableRuleConfigs().add(getOrderTableRuleConfiguration());

        // Configure default database sharding strategy
        shardingRuleConfig.setDefaultDatabaseShardingStrategyConfig(
            new StandardShardingStrategyConfiguration("customer_id",
                new CustomerIdDatabaseShardingAlgorithm()));

        return new ShardingDataSource(dataSourceMap, shardingRuleConfig);
    }

    private TableRuleConfiguration getOrderTableRuleConfiguration() {
        TableRuleConfiguration orderTableRuleConfig = new
TableRuleConfiguration("orders");

        // Configure order_id sharding column
        orderTableRuleConfig.setTableShardingStrategyConfig(
            new StandardShardingStrategyConfiguration("order_id",
                new OrderIdTableShardingAlgorithm()));

        return orderTableRuleConfig;
    }

    private Map<String, DataSource> createDataSourceMap() {
        // Create map of shard data sources
    }

    public class CustomerIdDatabaseShardingAlgorithm implements
PreciseShardingAlgorithm<Long> {
        @Override
        public String doSharding(Collection<String> databaseNames,
PreciseShardingValue<Long> shardingValue) {
            // Sharding algorithm implementation
        }
    }
}
```

8.10.3 Caching at Scale

Distributed Caching

- Implement Redis or Hazelcast for distributed caching
- Configure cache regions for different entity types
- Set appropriate TTL based on data volatility
- Implement cache invalidation strategies

```
@Configuration
@EnableCaching
public class DistributedCacheConfig {

    @Bean
```

```
    public RedisCacheManager cacheManager(RedisConnectionFactory
connectionFactory) {
        // Set default cache configuration
        RedisCacheConfiguration defaultConfig =
RedisCacheConfiguration.defaultCacheConfig()
            .entryTtl(Duration.ofMinutes(10))
            .serializeKeysWith(RedisSerializationContext.SerializationPair
                .fromSerializer(new StringRedisSerializer()))
            .serializeValuesWith(RedisSerializationContext.SerializationPair
                .fromSerializer(new GenericJackson2JsonRedisSerializer()));

        // Create custom configurations for specific caches
        Map<String, RedisCacheConfiguration> cacheConfigurations = new HashMap<>
();

        // Products cache with longer TTL
        cacheConfigurations.put("products",
defaultConfig.entryTtl(Duration.ofHours(1)));

        // Customer cache with shorter TTL due to frequent updates
        cacheConfigurations.put("customers",
defaultConfig.entryTtl(Duration.ofMinutes(5)));

        return RedisCacheManager.builder(connectionFactory)
            .cacheDefaults(defaultConfig)
            .withInitialCacheConfigurations(cacheConfigurations)
            .build();
    }
}
```

Cache Warming and Preloading

- Implement cache warming for frequently accessed data
- Preload caches after deployment or restart
- Schedule periodic cache refreshes for time-sensitive data

```
@Component
public class CacheWarmer implements ApplicationListener<ApplicationReadyEvent> {

    private final ProductService productService;
    private final CategoryService categoryService;

    // Constructor injection

    @Override
    public void onApplicationEvent(ApplicationReadyEvent event) {
        log.info("Warming caches after application startup");

        // Preload categories (typically small dataset)
        categoryService.getAllCategories();
```

```
        // Preload featured products
        productService.getFeaturedProducts();

        log.info("Cache warming completed");
    }

    @Scheduled(cron = "0 0 */4 * * *") // Every 4 hours
    public void refreshCaches() {
        log.info("Refreshing caches");

        // Clear and reload frequently changing caches
        categoryService.refreshCategories();
        productService.refreshFeaturedProducts();

        log.info("Cache refresh completed");
    }
}
```

8.10.4 Asynchronous Processing

Background Processing for Data-Intensive Operations

- Move data-intensive operations to background threads
- Use Spring's @Async for simple async operations
- Implement message queues for more complex workflows

```
@Service
public class ReportService {

    private final ReportRepository reportRepository;
    private final NotificationService notificationService;

    // Constructor injection

    public Long scheduleReport(ReportRequest request) {
        // Create report record in pending state
        Report report = new Report();
        report.setType(request.getType());
        report.setParameters(request.getParameters());
        report.setStatus(ReportStatus.PENDING);
        report.setRequestedBy(request.getUserId());
        report.setRequestedAt(LocalDateTime.now());

        Report savedReport = reportRepository.save(report);

        // Trigger async generation
        generateReportAsync(savedReport.getId());

        return savedReport.getId();
    }
```

```
    @Async("reportTaskExecutor")
    public void generateReportAsync(Long reportId) {
        try {
            Report report = reportRepository.findById(reportId)
                .orElseThrow(() -> new ReportNotFoundException(reportId));

            // Update status to processing
            report.setStatus(ReportStatus.PROCESSING);
            reportRepository.save(report);

            // Generate report (potentially long-running operation)
            byte[] reportData = generateReportData(report);

            // Update with result
            report.setStatus(ReportStatus.COMPLETED);
            report.setResultData(reportData);
            report.setCompletedAt(LocalDateTime.now());
            reportRepository.save(report);

            // Notify user
            notificationService.sendReportCompletionNotification(report);
        } catch (Exception e) {
            log.error("Error generating report {}", reportId, e);

            Report report = reportRepository.findById(reportId).orElse(null);
            if (report != null) {
                report.setStatus(ReportStatus.FAILED);
                report.setErrorMessage(e.getMessage());
                reportRepository.save(report);

                notificationService.sendReportFailureNotification(report);
            }
        }
    }

    private byte[] generateReportData(Report report) {
        // Report generation logic
    }
}

@Configuration
public class AsyncConfig {

    @Bean(name = "reportTaskExecutor")
    public TaskExecutor reportTaskExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(5);
        executor.setMaxPoolSize(10);
        executor.setQueueCapacity(25);
        executor.setThreadNamePrefix("report-");
        return executor;
    }
}
```

Message Queue Integration

- Use message queues (RabbitMQ, Kafka) for inter-service communication
- Implement event-driven architecture for data processing
- Decouple data producers from consumers

```
@Configuration
public class RabbitMQConfig {

    @Bean
    public Queue ordersQueue() {
        return new Queue("orders.processing");
    }

    @Bean
    public DirectExchange ordersExchange() {
        return new DirectExchange("orders.exchange");
    }

    @Bean
    public Binding ordersBinding(Queue ordersQueue, DirectExchange ordersExchange)
{
        return BindingBuilder.bind(ordersQueue)
            .to(ordersExchange)
            .with("orders.new");
    }
}

@Service
public class OrderService {

    private final OrderRepository orderRepository;
    private final RabbitTemplate rabbitTemplate;

    // Constructor injection

    @Transactional
    public Order createOrder(OrderRequest request) {
        // Database operations
        Order order = new Order();
        // Set order properties

        Order savedOrder = orderRepository.save(order);

        // Publish event for async processing
        rabbitTemplate.convertAndSend(
            "orders.exchange",
            "orders.new",
            new OrderCreatedEvent(savedOrder.getId())
        );
```

```
        return savedOrder;
    }
}

@Component
public class OrderProcessor {

    private final OrderRepository orderRepository;
    private final InventoryService inventoryService;
    private final PaymentService paymentService;

    // Constructor injection

    @RabbitListener(queues = "orders.processing")
    public void processOrder(OrderCreatedEvent event) {
        try {
            Order order = orderRepository.findById(event.getOrderId())
                .orElseThrow(() -> new
OrderNotFoundException(event.getOrderId()));

            // Process inventory
            inventoryService.reserveInventory(order);

            // Process payment
            PaymentResult result = paymentService.processPayment(order);

            // Update order status based on payment result
            if (result.isSuccessful()) {
                order.setStatus(OrderStatus.PAYMENT_COMPLETED);
            } else {
                order.setStatus(OrderStatus.PAYMENT_FAILED);
                order.setStatusMessage(result.getErrorMessage());

                // Release inventory
                inventoryService.releaseInventory(order);
            }

            orderRepository.save(order);
        } catch (Exception e) {
            log.error("Error processing order {}", event.getOrderId(), e);
            // Handle exception, possibly retry or move to DLQ
        }
    }
}
```

8.10.5 Microservices Data Patterns

Database per Service

- Implement separate databases for different microservices
- Ensure data independence between services
- Use service APIs for cross-service data access

Event Sourcing

- Store state changes as a sequence of events
- Reconstruct current state by replaying events
- Enable complete audit history and time-travel queries

```
@Entity
@Table(name = "events")
public class DomainEvent {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String aggregateType;

    @Column(nullable = false)
    private String aggregateId;

    @Column(nullable = false)
    private String eventType;

    @Column(nullable = false)
    @Lob
    private String eventData;

    @Column(nullable = false)
    private LocalDateTime timestamp;

    // Getters and setters
}

@Service
public class EventSourceService {

    private final EventRepository eventRepository;
    private final ObjectMapper objectMapper;

    // Constructor injection

    @Transactional
    public void saveEvent(String aggregateType, String aggregateId,
                          String eventType, Object eventData) throws
JsonProcessingException {
        DomainEvent event = new DomainEvent();
        event.setAggregateType(aggregateType);
        event.setAggregateId(aggregateId);
        event.setEventType(eventType);
        event.setEventData(objectMapper.writeValueAsString(eventData));
        event.setTimestamp(LocalDateTime.now());

        eventRepository.save(event);
```

```
    }

    public <T> T reconstructAggregate(String aggregateType, String aggregateId,
Class<T> type) {
        List<DomainEvent> events =
eventRepository.findByAggregateTypeAndAggregateIdOrderByTimestamp(
            aggregateType, aggregateId);

        if (events.isEmpty()) {
            return null;
        }

        try {
            // Create empty aggregate instance
            T aggregate = type.getDeclaredConstructor().newInstance();

            // Apply each event to rebuild state
            for (DomainEvent event : events) {
                applyEvent(aggregate, event);
            }

            return aggregate;
        } catch (Exception e) {
            throw new AggregateReconstructionException(
                "Failed to reconstruct aggregate " + aggregateType + ":" +
aggregateId, e);
        }
    }

    private <T> void applyEvent(T aggregate, DomainEvent event) throws IOException
{
        try {
            String methodName = "apply" +
StringUtils.capitalize(event.getEventType());
            Method method = aggregate.getClass().getDeclaredMethod(
                methodName,
objectMapper.readTree(event.getEventData()).getClass());

            Object eventObject = objectMapper.readValue(event.getEventData(),
                method.getParameterTypes()[0]);

            method.invoke(aggregate, eventObject);
        } catch (Exception e) {
            throw new EventApplicationException(
                "Failed to apply event " + event.getEventType() + " to aggregate",
e);
        }
    }
}
```

8.10.6 Monitoring and Operations at Scale

Distributed Tracing

- Implement distributed tracing for request flows across services
- Track database operations in the context of user requests
- Identify bottlenecks in multi-service architectures

```
@Configuration
public class TracingConfig {

    @Bean
    public OpenTelemetry openTelemetry() {
        return OpenTelemetrySdk.builder()
            .setTracerProvider(
                SdkTracerProvider.builder()
                    .addSpanProcessor(BatchSpanProcessor.builder(
                        JaegerGrpcSpanExporter.builder()
                            .setEndpoint("http://jaeger:14250")
                            .build())
                        .build())
                    .build())
            .setPropagators(ContextPropagators.create(
                W3CTraceContextPropagator.getInstance()))
            .build();
    }

    @Bean
    public Filter tracingFilter(OpenTelemetry openTelemetry) {
        return new TracingFilter(openTelemetry);
    }
}

public class TracingFilter extends OncePerRequestFilter {

    private final OpenTelemetry openTelemetry;
    private final Tracer tracer;

    public TracingFilter(OpenTelemetry openTelemetry) {
        this.openTelemetry = openTelemetry;
        this.tracer = openTelemetry.getTracer("app.request");
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response,
                                    FilterChain filterChain) throws
ServletException, IOException {
        String uri = request.getRequestURI();

        SpanBuilder spanBuilder = tracer.spanBuilder("HTTP " + request.getMethod()
+ " " + uri)
            .setSpanKind(SpanKind.SERVER);
```

```
        // Extract context from request headers
        TextMapPropagator propagator =
openTelemetry.getPropagators().getTextMapPropagator();
        Context extractedContext = propagator.extract(Context.current(), request,
            new HttpServletRequestGetter());
        spanBuilder.setParent(extractedContext);

        // Create span for this request
        try (Scope scope = spanBuilder.startSpan().makeCurrent()) {
            Span span = Span.current();

            // Add request details as span attributes
            span.setAttribute("http.method", request.getMethod());
            span.setAttribute("http.url", request.getRequestURL().toString());
            span.setAttribute("http.host", request.getServerName());

            try {
                // Execute the rest of the filter chain
                filterChain.doFilter(request, response);

                // Add response details
                span.setAttribute("http.status_code", response.getStatus());
            } catch (Exception e) {
                span.recordException(e);
                span.setStatus(StatusCode.ERROR, e.getMessage());
                throw e;
            }
        }
    }

    private static class HttpServletRequestGetter implements
TextMapGetter<HttpServletRequest> {
        @Override
        public Iterable<String> keys(HttpServletRequest request) {
            return Collections.list(request.getHeaderNames());
        }

        @Override
        public String get(HttpServletRequest request, String key) {
            return request.getHeader(key);
        }
    }
}
```

Centralized Logging and Metrics

- Implement structured logging for data operations
- Correlate logs with request traces
- Create dashboards for key performance indicators
- Set up alerts for database performance issues

By following these best practices and scaling strategies, you can build data-centric Spring Boot applications that perform well even under high loads. Proper design, thoughtful optimization, and continuous monitoring enable your application to scale efficiently as your user base and data volume grow.

8.11 Summary

In this chapter, we explored comprehensive strategies for optimizing data access in Spring Boot applications. We covered fundamental areas including entity design, query optimization, and transaction management, as well as advanced topics like connection pooling, caching strategies, and scaling approaches.

Key takeaways include:

1. **Start with good design**: Properly designed entities, thoughtful relationship mapping, and strategic indexing form the foundation of performant data access.
2. **Optimize queries**: Use appropriate fetch strategies, projections, and pagination to reduce database load and improve response times.
3. **Implement effective caching**: Multi-level caching strategies can dramatically improve performance for read-heavy applications.
4. **Manage resources carefully**: Properly configured connection pools and transaction boundaries ensure efficient resource utilization.
5. **Monitor continuously**: Comprehensive monitoring of database operations helps identify bottlenecks and validate optimizations.
6. **Scale thoughtfully**: Choose appropriate scaling strategies based on your application's specific data access patterns.
7. **Test performance rigorously**: Regular performance testing helps catch issues early and ensures your application meets its performance requirements.

Chapter 11: Deployment and Monitoring of Data-Centric Spring Boot Applications

Introduction

Moving data-centric Spring Boot applications from development to production represents a critical transition that demands careful planning and execution. Unlike stateless applications, data-centric applications carry additional responsibilities around data integrity, persistence, security, and performance at scale. This chapter explores the complete lifecycle of deploying and monitoring such applications, with particular attention to the unique challenges posed by database interactions in production environments.

Deployment Strategies for Data-Centric Applications

Cloud Deployment

Cloud platforms provide robust infrastructure for data-centric applications, offering managed services that can significantly reduce operational overhead.

AWS Deployment Options

Amazon Web Services offers several deployment models well-suited for Spring Boot applications:

- **Elastic Beanstalk**: Provides a platform-as-a-service (PaaS) environment where you can deploy your Spring Boot JAR with minimal configuration. Beanstalk automatically handles capacity provisioning, load balancing, and application health monitoring.
- **ECS (Elastic Container Service)**: Ideal for containerized Spring Boot applications, ECS allows you to run Docker containers without managing the underlying infrastructure. This approach provides consistency between development and production environments.
- **Lambda with Spring Cloud Function**: For event-driven microservices that access data, AWS Lambda with Spring Cloud Function offers a serverless deployment model, though with considerations around cold starts and connection management.

Azure Deployment Options

Microsoft Azure provides similar capabilities with some unique advantages:

- **Azure Spring Apps**: A managed service specifically designed for Spring Boot applications, offering built-in support for configuration servers, service discovery, and application lifecycle management.
- **Azure App Service**: PaaS offering supporting Java applications with easy integration to other Azure services.
- **Azure Kubernetes Service (AKS)**: Managed Kubernetes service for container orchestration, providing advanced scheduling and management for containerized Spring Boot applications.

Google Cloud Platform (GCP) Options

GCP completes the major cloud provider landscape with:

- **Google App Engine**: Platform-as-a-service with automatic scaling and load balancing capabilities.
- **Google Kubernetes Engine (GKE)**: Managed Kubernetes service for containerized applications.
- **Cloud Run**: Serverless platform specifically designed for containerized applications, offering quick deployment and automatic scaling to zero when not in use.

On-Premise Deployment

Despite the cloud's advantages, many organizations maintain on-premise deployments due to regulatory requirements, existing infrastructure investments, or specific performance needs.

Traditional Server Deployment

For traditional server deployments, Spring Boot applications can be deployed as:

- **Standalone JARs with systemd**: Using Linux's systemd to manage application lifecycle, ensuring automatic restarts and proper logging.
- **Application Servers**: Deploying to Tomcat, Jetty, or other J2EE application servers, which can provide additional management capabilities for enterprises with existing Java infrastructure.

Private Cloud and Virtualization

On-premise private clouds built using virtualization technologies offer many cloud-like advantages:

- **OpenStack**: Open-source cloud computing platform that supports deploying Spring Boot applications in a private cloud environment.
- **VMware vSphere**: Enterprise virtualization platform that can host Spring Boot applications with advanced management features.

Hybrid Deployment Models

Many organizations adopt hybrid approaches where:

- Application components may run in cloud environments
- Databases remain on-premise for security or compliance reasons
- Data synchronization strategies become critical in such architectures

This hybrid model requires careful consideration of network latency, security, and data consistency challenges.

Packaging Data-Centric Spring Boot Applications

Executable JARs

Spring Boot's capability to create self-contained executable JARs simplifies deployment significantly. For data-centric applications, this approach has several advantages:

Creating Executable JARs

Using Maven, the Spring Boot Maven plugin creates an executable JAR with all dependencies included:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <executable>true</executable>
            </configuration>
        </plugin>
    </plugins>
</build>
```

This configuration creates a JAR that can be executed directly with `java -jar application.jar`.

Profile-Specific Properties for Different Environments

For data-centric applications, managing environment-specific database configurations is crucial:

```
src/main/resources/
  ├── application.properties      # Common properties
  ├── application-dev.properties  # Development database settings
  ├── application-test.properties # Test database configuration
  └── application-prod.properties # Production database settings
```

When launching the application, specify the active profile:

```
java -jar -Dspring.profiles.active=prod application.jar
```

Fat JAR Considerations for Data-Centric Applications

While fat JARs are convenient, they present some considerations for data-centric applications:

- Larger deployment artifacts can slow CI/CD pipelines
- Database drivers and connection pool libraries increase JAR size
- JVM memory requirements increase with JAR size

Containerization with Docker

Containerization has become a standard approach for deploying Spring Boot applications, offering consistency across environments and easy scaling.

Creating Efficient Docker Images

For data-centric applications, Docker image efficiency is especially important:

```
# Multi-stage build to reduce image size
FROM maven:3.8.6-openjdk-17 AS build
WORKDIR /app
COPY pom.xml .
# Download dependencies separately for better caching
RUN mvn dependency:go-offline

COPY src/ /app/src/
RUN mvn package -DskipTests

# Runtime image
FROM eclipse-temurin:17-jre
WORKDIR /app
# Copy only the built JAR from the build stage
COPY --from=build /app/target/*.jar app.jar

# Externalize database configuration
```

```
ENV SPRING_PROFILES_ACTIVE=prod
ENV SPRING_DATASOURCE_URL=jdbc:postgresql://db:5432/myapp
ENV SPRING_DATASOURCE_USERNAME=username
ENV SPRING_DATASOURCE_PASSWORD=password

ENTRYPOINT ["java", "-jar", "/app/app.jar"]
```

Docker Compose for Local Development

Docker Compose simplifies local development with databases:

```
version: '3.8'
services:
  app:
    build: .
    ports:
      - "8080:8080"
    environment:
      - SPRING_PROFILES_ACTIVE=dev
      - SPRING_DATASOURCE_URL=jdbc:postgresql://db:5432/myapp
    depends_on:
      - db

  db:
    image: postgres:14
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
      - POSTGRES_DB=myapp
    volumes:
      - postgres-data:/var/lib/postgresql/data
    ports:
      - "5432:5432"

volumes:
  postgres-data:
```

Container-Specific Optimizations for Java Applications

Data-centric applications in containers benefit from specific optimizations:

- Setting appropriate JVM memory limits for containerized environments
- Using container-aware connection pools that respect container constraints
- Implementing health checks that verify database connectivity

Database Deployment and Management in Production

Cloud Databases

Cloud database services offer managed solutions that reduce operational overhead and provide built-in high availability, backup, and scaling capabilities.

AWS RDS (Relational Database Service)

AWS RDS supports multiple database engines compatible with Spring Boot:

- PostgreSQL, MySQL, MariaDB for open-source options
- Oracle and SQL Server for enterprise deployments
- Aurora for cloud-native high-performance needs

Configuration in Spring Boot:

```
# AWS RDS PostgreSQL example
spring.datasource.url=jdbc:postgresql://myinstance.123456789012.us-east-
1.rds.amazonaws.com:5432/mydb
spring.datasource.username=${RDS_USERNAME}
spring.datasource.password=${RDS_PASSWORD}
spring.datasource.hikari.maximum-pool-size=10
```

Azure SQL Database

Azure's managed SQL offering provides:

- DTU-based or vCore-based purchasing models
- Point-in-time restore capabilities
- Geo-replication options

Configuration:

```
# Azure SQL Database with Microsoft JDBC driver
spring.datasource.url=jdbc:sqlserver://myserver.database.windows.net:1433;database
=mydb;encrypt=true;trustServerCertificate=false;hostNameInCertificate=*.database.w
indows.net;loginTimeout=30;
spring.datasource.username=${AZURE_SQL_USERNAME}
spring.datasource.password=${AZURE_SQL_PASSWORD}
spring.datasource.driver-class-name=com.microsoft.sqlserver.jdbc.SQLServerDriver
```

Google Cloud SQL

Google Cloud SQL provides managed MySQL, PostgreSQL, and SQL Server databases with:

- Automatic backups and point-in-time recovery
- High availability configuration
- Integration with Google's security features

Configuration:

```
# Google Cloud SQL PostgreSQL example
spring.datasource.url=jdbc:postgresql://google/mydb?cloudSqlInstance=myproject:us-central1:myinstance&socketFactory=com.google.cloud.sql.postgres.SocketFactory
spring.datasource.username=${CLOUD_SQL_USERNAME}
spring.datasource.password=${CLOUD_SQL_PASSWORD}
spring.datasource.driver-class-name=org.postgresql.Driver
```

Managed Database Services

Beyond the major cloud providers, specialized database services offer unique capabilities.

MongoDB Atlas

For Spring Boot applications using MongoDB:

```
# MongoDB Atlas configuration
spring.data.mongodb.uri=mongodb+srv://${MONGO_USERNAME}:${MONGO_PASSWORD}@cluster0.mongodb.net/mydb?retryWrites=true&w=majority
```

Amazon Aurora Serverless

Serverless relational database option with automatic scaling to zero:

```
# Amazon Aurora Serverless configuration
spring.datasource.url=jdbc:mysql://${AURORA_ENDPOINT}/mydb
spring.datasource.username=${AURORA_USERNAME}
spring.datasource.password=${AURORA_PASSWORD}
# Configure minimum connections for intermittent usage patterns
spring.datasource.hikari.minimum-idle=1
spring.datasource.hikari.maximum-pool-size=10
```

Managed NoSQL Services

For applications using non-relational databases:

- **Amazon DynamoDB** with Spring Data DynamoDB
- **Azure Cosmos DB** with Spring Data Cosmos
- **Firebase/Firestore** for real-time data applications

Self-Managed Databases in Cloud Infrastructure

For teams requiring more control, self-managed databases on cloud infrastructure provide a middle ground:

- Database installed on EC2, Azure VM, or Google Compute Engine
- Responsibility for patching, backups, and scaling
- Greater control over configuration and tuning

This approach requires careful planning for high availability, backup strategies, and performance optimization.

Configuration Management for Data Sources in Production

Externalized Configuration

Spring Boot's externalized configuration capabilities are particularly valuable for data-centric applications, where database connection details may differ across environments.

Environment Variables

Environment variables provide a secure and platform-independent way to configure database connections:

```
# In production environment
export SPRING_DATASOURCE_URL=jdbc:postgresql://production-
db.example.com:5432/myapp
export SPRING_DATASOURCE_USERNAME=app_user
export SPRING_DATASOURCE_PASSWORD=complex_password
```

In the application:

```
# application.properties
spring.datasource.url=${SPRING_DATASOURCE_URL}
spring.datasource.username=${SPRING_DATASOURCE_USERNAME}
spring.datasource.password=${SPRING_DATASOURCE_PASSWORD}
```

External Property Files

For more complex configurations, external property files offer flexibility:

```
# Launch application with external config
java -jar app.jar --spring.config.location=file:/etc/myapp/application.properties
```

Spring Cloud Config Server

For distributed applications, Spring Cloud Config Server centralizes configuration:

```
# bootstrap.properties in application
spring.application.name=data-service
spring.cloud.config.uri=http://config-server:8888
```

The config server stores environment-specific database configurations and can be integrated with Git for version control.

Secrets Management

Database credentials require special handling as sensitive information.

Kubernetes Secrets

In Kubernetes environments:

```
# Secret definition
apiVersion: v1
kind: Secret
metadata:
  name: database-credentials
type: Opaque
data:
  username: YXBwX3VzZXI=  # Base64 encoded
  password: Y29tcGxleF9wYXNzd29yZA==  # Base64 encoded
```

Mounting as environment variables:

```
# Deployment using the secret
env:
  - name: SPRING_DATASOURCE_USERNAME
    valueFrom:
      secretKeyRef:
        name: database-credentials
        key: username
  - name: SPRING_DATASOURCE_PASSWORD
    valueFrom:
      secretKeyRef:
        name: database-credentials
        key: password
```

Cloud Provider Secret Management Services

Cloud providers offer dedicated secret management:

- **AWS Secrets Manager**: Integration via AWS SDK or JDBC wrapper
- **Azure Key Vault**: With Spring Cloud Azure Key Vault integration
- **Google Secret Manager**: Using Spring Cloud GCP integration

Example with AWS Secrets Manager:

```
@Configuration
public class DataSourceConfig {

    @Value("${aws.secretsmanager.secretName}")
    private String secretName;

    @Bean
```

```
    public DataSource dataSource() {
        AWSSecretsManagerSecret secret =
amazonSecretsManagerClient.getSecret(secretName);

        return DataSourceBuilder.create()
            .url(secret.getProperty("url"))
            .username(secret.getProperty("username"))
            .password(secret.getProperty("password"))
            .build();
    }
}
```

HashiCorp Vault

For organizations with multi-cloud or hybrid deployments, HashiCorp Vault provides consistent secrets management:

```
# With Spring Cloud Vault
spring.cloud.vault.uri=https://vault.example.com:8200
spring.cloud.vault.token=${VAULT_TOKEN}
spring.cloud.vault.kv.enabled=true
```

Using Vault's dynamic database credentials generation:

```
spring.cloud.vault.database.enabled=true
spring.cloud.vault.database.role=readonly
spring.cloud.vault.database.backend=postgresql
```

Monitoring Data Access Performance in Production

Metrics with Spring Boot Actuator

Spring Boot Actuator provides built-in capabilities for exposing and monitoring metrics relevant to data access.

Enabling Database Metrics

To enable database metrics in a Spring Boot application:

```
# Enable all Actuator endpoints
management.endpoints.web.exposure.include=*

# Database-specific metrics
management.metrics.enable.jdbc=true
management.metrics.enable.hibernate=true
```

Key Metrics for Data-Centric Applications

Critical metrics to monitor include:

- **Connection pool metrics**: Active connections, idle connections, max connections, wait time
- **Query execution metrics**: Query execution time, query count
- **Transaction metrics**: Transaction count, rollback count
- **Cache metrics**: Cache hit ratio, cache size

Exposing Metrics to Monitoring Systems

Spring Boot supports various formats for metrics:

```
# Prometheus endpoint for metrics scraping
management.endpoints.web.exposure.include=prometheus
management.metrics.export.prometheus.enabled=true
```

Custom Data Access Metrics

For application-specific metrics:

```
@Repository
public class CustomerRepository {

    private final MeterRegistry meterRegistry;
    private final JdbcTemplate jdbcTemplate;

    @Autowired
    public CustomerRepository(MeterRegistry meterRegistry, JdbcTemplate
jdbcTemplate) {
        this.meterRegistry = meterRegistry;
        this.jdbcTemplate = jdbcTemplate;
    }

    public List<Customer> findByRegion(String region) {
        Timer.Sample sample = Timer.start(meterRegistry);
        try {
            List<Customer> customers = jdbcTemplate.query(
                "SELECT * FROM customers WHERE region = ?",
                new Object[]{region},
                new CustomerRowMapper()
            );
            return customers;
        } finally {
            sample.stop(meterRegistry.timer("repository.customer.findByRegion",
                "region", region));
        }
    }
}
```

Logging for Data Access Operations

Logging provides valuable insights into database operations and potential issues.

Configuring SQL Logging

Enabling SQL logging in development and selectively in production:

```
# Show SQL statements in logs
logging.level.org.hibernate.SQL=DEBUG

# Show SQL parameter values
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE

# Log connection acquisition/release for connection pool troubleshooting
logging.level.com.zaxxer.hikari=DEBUG
```

Structured Logging for Database Operations

Using JSON logging format for better integration with log analysis tools:

```
<dependency>
    <groupId>net.logstash.logback</groupId>
    <artifactId>logstash-logback-encoder</artifactId>
    <version>7.3</version>
</dependency>
```

With Logback configuration:

```
<appender name="JSON" class="ch.qos.logback.core.ConsoleAppender">
    <encoder class="net.logstash.logback.encoder.LogstashEncoder"/>
</appender>
```

AOP for Comprehensive Database Operation Logging

Aspect-Oriented Programming can provide detailed logging without modifying repository code:

```
@Aspect
@Component
public class RepositoryMonitoringAspect {

    private static final Logger log =
LoggerFactory.getLogger(RepositoryMonitoringAspect.class);

    @Around("execution(* org.example.repository.*Repository.*(..))")
    public Object logRepositoryAccess(ProceedingJoinPoint joinPoint) throws
```

```
Throwable {
        String methodName = joinPoint.getSignature().getName();
        String className = joinPoint.getTarget().getClass().getSimpleName();

        long startTime = System.currentTimeMillis();
        try {
            Object result = joinPoint.proceed();
            long executionTime = System.currentTimeMillis() - startTime;

            log.info("Repository method {}.{} executed in {} ms",
                    className, methodName, executionTime);

            return result;
        } catch (Exception e) {
            log.error("Error executing {}.{}: {}",
                    className, methodName, e.getMessage(), e);
            throw e;
        }
    }
}
```

Application Performance Monitoring (APM) Tools

APM tools provide deeper insights into application performance, including detailed database operation analysis.

Spring Boot Integration with APM Tools

Popular APM options for Spring Boot applications:

- **New Relic**: Offers Java agent with database visibility
- **Dynatrace**: Provides AI-powered analysis of database performance
- **Elastic APM**: Open-source option with strong database monitoring
- **DataDog APM**: Features database monitoring with query-level visibility

Configuration with Spring Boot (New Relic example):

```
# Add Java agent to application startup
java -javaagent:/path/to/newrelic.jar -jar app.jar
```

Features to Look for in APM for Data-Centric Applications

Key APM capabilities for data-centric applications:

- SQL query performance analysis
- Slow query detection
- Database error tracking
- Connection pool monitoring
- Query plan visualization

- Database load correlation with application throughput

Distributed Tracing for Database Operations

For microservices architectures, distributed tracing helps correlate database operations across services:

```
# Enable Micrometer Tracing with Zipkin
management.tracing.sampling.probability=1.0
management.zipkin.tracing.endpoint=http://zipkin:9411/api/v2/spans
```

Database Connection Pooling in Production

Tuning Connection Pools for Production Load

Connection pooling is critical for data-centric applications, balancing resource efficiency with performance.

HikariCP Configuration for Production

HikariCP is Spring Boot's default connection pool. Key configuration properties:

```
# Basic pool sizing
spring.datasource.hikari.minimum-idle=10
spring.datasource.hikari.maximum-pool-size=50

# Connection lifetime management
spring.datasource.hikari.max-lifetime=1800000  # 30 minutes
spring.datasource.hikari.idle-timeout=600000   # 10 minutes

# Connection testing
spring.datasource.hikari.connection-test-query=SELECT 1
spring.datasource.hikari.validation-timeout=5000

# Performance optimization
spring.datasource.hikari.connection-timeout=30000
```

Connection Pool Sizing Strategies

Determining optimal connection pool size:

```
Connections = ((Threads * QueriesPerRequest * QueryExecutionTime) /
AcceptableLatency) + Safety Buffer
```

Where:

- Threads: Maximum number of concurrent threads in your application server
- QueriesPerRequest: Average number of queries per request
- QueryExecutionTime: Average query execution time in milliseconds

- AcceptableLatency: Maximum acceptable latency in milliseconds
- Safety Buffer: Additional connections to handle spikes

Monitoring Connection Pool Health

Critical metrics to watch:

- Connection acquisition time
- Connection usage time
- Connection wait count (threads waiting for connections)
- Connection timeout count
- Total vs. active connections

Example Micrometer metrics for HikariCP:

```
hikaricp.connections.active
hikaricp.connections.idle
hikaricp.connections.pending
hikaricp.connections.max
hikaricp.connections.timeout
```

Dynamic Pool Adjustment

For applications with variable load patterns:

```
@Component
public class DynamicConnectionPoolConfig {

    private final HikariDataSource dataSource;

    @Autowired
    public DynamicConnectionPoolConfig(DataSource dataSource) {
        this.dataSource = (HikariDataSource) dataSource;
    }

    @Scheduled(fixedRate = 60000)  // Check every minute
    public void adjustPoolSize() {
        int currentLoad = getCurrentSystemLoad();  // Method to get current system
load

        if (currentLoad > 80) {  // High load
            int newPoolSize = Math.min(dataSource.getMaximumPoolSize() + 5, 100);
            dataSource.setMaximumPoolSize(newPoolSize);
            log.info("Increased connection pool maximum to {}", newPoolSize);
        } else if (currentLoad < 30) {  // Low load
            int newPoolSize = Math.max(dataSource.getMaximumPoolSize() - 5, 10);
            dataSource.setMaximumPoolSize(newPoolSize);
            log.info("Decreased connection pool maximum to {}", newPoolSize);
        }
```

```
    }
}
```

Backup and Recovery Strategies for Data

Data Backup and Disaster Recovery Planning

Ensuring data safety is a critical aspect of data-centric application deployment.

Backup Strategies for Different Database Types

Common backup approaches by database type:

- **Relational Databases (MySQL, PostgreSQL, Oracle)**:
    - Full backups (periodic complete database dumps)
    - Incremental backups (changes since last backup)
    - Transaction log backups (continuous recording of transactions)
- **NoSQL Databases (MongoDB, Cassandra)**:
    - Replica set snapshots
    - Collection-level exports
    - Multi-region replication
- **NewSQL Databases (CockroachDB, Google Spanner)**:
    - Built-in replication across nodes
    - Geographic backup strategies
    - Consistent snapshot backups

Automating Backup Procedures

Spring Boot applications can integrate backup processes:

```
@Component
public class DatabaseBackupScheduler {

    @Scheduled(cron = "0 0 2 * * *")  // Daily at 2 AM
    public void performDatabaseBackup() {
        ProcessBuilder processBuilder = new ProcessBuilder();

        // For PostgreSQL example
        processBuilder.command(
            "pg_dump",
            "-h", "db.example.com",
            "-U", "backup_user",
            "-d", "myapp",
            "-f", "/backups/myapp-" + LocalDate.now() + ".sql"
        );
```

```
            try {
                Process process = processBuilder.start();
                int exitCode = process.waitFor();

                if (exitCode == 0) {
                    log.info("Database backup completed successfully");
                } else {
                    log.error("Database backup failed with exit code: {}", exitCode);
                }
            } catch (Exception e) {
                log.error("Error performing database backup", e);
            }
        }
    }
```

Testing Recovery Procedures

Regular recovery testing is essential:

- Scheduled recovery drills using backup data
- Automated restore verification jobs
- Documentation of recovery time objectives (RTO) and procedures

Data Replication for High Availability

Data replication provides both backup and high availability:

- **Primary-Replica (Master-Slave) Setups**: Data written to primary, automatically replicated to one or more replicas
- **Multi-Master Replication**: Multiple writable nodes with conflict resolution
- **Geo-Replication**: Data replicated across geographic regions

Spring Boot configuration for read-replica usage:

```
@Configuration
public class DataSourceConfig {

    @Bean
    @Primary
    @ConfigurationProperties("spring.datasource.primary")
    public DataSource primaryDataSource() {
        return DataSourceBuilder.create().build();
    }

    @Bean(name = "replicaDataSource")
    @ConfigurationProperties("spring.datasource.replica")
    public DataSource replicaDataSource() {
        return DataSourceBuilder.create().build();
    }

    @Bean
```

```
    public AbstractRoutingDataSource routingDataSource(
            @Qualifier("primaryDataSource") DataSource primaryDataSource,
            @Qualifier("replicaDataSource") DataSource replicaDataSource) {

        Map<Object, Object> targetDataSources = new HashMap<>();
        targetDataSources.put(DataSourceType.PRIMARY, primaryDataSource);
        targetDataSources.put(DataSourceType.REPLICA, replicaDataSource);

        AbstractRoutingDataSource routingDataSource = new
ReadWriteRoutingDataSource();
        routingDataSource.setTargetDataSources(targetDataSources);
        routingDataSource.setDefaultTargetDataSource(primaryDataSource);

        return routingDataSource;
    }
}
```

Scaling Data-Centric Spring Boot Applications

Scaling Data Access Layers and Databases

As applications grow, scaling data access becomes a key challenge.

Horizontal vs. Vertical Scaling Strategies

Both approaches have merits for data-centric applications:

- **Vertical Scaling (Scaling Up)**:
    - Adding more resources (CPU, memory, disk) to existing servers
    - Simpler to implement but has physical limitations
    - Often requires downtime for upgrades
- **Horizontal Scaling (Scaling Out)**:
    - Adding more servers to distribute load
    - More complex to implement but offers better fault tolerance
    - Usually allows for scaling without downtime

Read-Write Splitting

Separating read and write operations improves scalability:

```
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface ReadOnly {
}

@Aspect
@Component
public class DataSourceRoutingAspect {
```

```
    @Pointcut("@annotation(org.example.ReadOnly)")
    public void readOnlyOperation() {}

    @Before("readOnlyOperation()")
    public void setReadOnlyDataSource() {
        DataSourceContextHolder.setDataSourceType(DataSourceType.REPLICA);
    }

    @After("readOnlyOperation()")
    public void clearDataSourceType() {
        DataSourceContextHolder.clear();
    }
}
```

Usage in service layer:

```
@Service
public class CustomerService {

    private final CustomerRepository repository;

    @Autowired
    public CustomerService(CustomerRepository repository) {
        this.repository = repository;
    }

    // Uses primary (write) data source
    public Customer createCustomer(Customer customer) {
        return repository.save(customer);
    }

    // Uses replica (read) data source
    @ReadOnly
    public List<Customer> findAllCustomers() {
        return repository.findAll();
    }
}
```

Database Sharding Strategies

Sharding distributes data across multiple database instances:

- **Horizontal Sharding**: Splits rows across servers based on a shard key
- **Vertical Sharding**: Splits columns/tables across servers based on usage patterns
- **Functional Sharding**: Splits by function or bounded context

Spring Boot implementation with custom routing:

```
@Configuration
public class ShardedDataSourceConfig {

    @Bean
    public DataSource shardedDataSource() {
        ShardingDataSource dataSource = new ShardingDataSource();

        Map<String, DataSource> shards = new HashMap<>();
        shards.put("shard1",
createDataSource("jdbc:postgresql://shard1:5432/myapp"));
        shards.put("shard2",
createDataSource("jdbc:postgresql://shard2:5432/myapp"));
        shards.put("shard3",
createDataSource("jdbc:postgresql://shard3:5432/myapp"));

        dataSource.setShards(shards);
        return dataSource;
    }

    private DataSource createDataSource(String url) {
        HikariDataSource ds = new HikariDataSource();
        ds.setJdbcUrl(url);
        ds.setUsername("app_user");
        ds.setPassword("password");
        return ds;
    }
}

class ShardingDataSource extends AbstractRoutingDataSource {
    @Override
    protected Object determineCurrentLookupKey() {
        return ShardingContextHolder.getCurrentShard();
    }
}
```

Caching Strategies

Caching reduces database load and improves performance:

```
@Configuration
@EnableCaching
public class CachingConfig {

    @Bean
    public CacheManager cacheManager() {
        SimpleCacheManager cacheManager = new SimpleCacheManager();
        cacheManager.setCaches(Arrays.asList(
            new ConcurrentMapCache("customers"),
            new ConcurrentMapCache("products"),
            new ConcurrentMapCache("orders")
        ));
```

```
        return cacheManager;
    }
}

@Service
public class ProductService {

    private final ProductRepository repository;

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

    @Cacheable(value = "products", key = "#id")
    public Product findProduct(Long id) {
        return repository.findById(id)
            .orElseThrow(() -> new ProductNotFoundException(id));
    }

    @CacheEvict(value = "products", key = "#product.id")
    public Product updateProduct(Product product) {
        return repository.save(product);
    }

    @CacheEvict(value = "products", allEntries = true)
    @Scheduled(fixedRate = 86400000)  // Daily refresh
    public void refreshProductCache() {
        // Clear cache to prevent stale data
    }
}
```

Distributed Caching with Redis or Hazelcast

For multi-instance deployments, distributed caching is essential:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

Configuration:

```
# Redis configuration
spring.redis.host=redis.example.com
spring.redis.port=6379
spring.cache.type=redis
spring.cache.redis.time-to-live=3600000
```

Best Practices for Deployment and Monitoring Data-Centric Applications

Automation, Infrastructure as Code, Continuous Monitoring

Bringing together all aspects of production deployment requires a cohesive approach.

Automation with CI/CD Pipelines

Continuous Integration/Continuous Deployment automates the release process:

```
# GitLab CI/CD example
stages:
  - build
  - test
  - database-migration
  - deploy
  - post-deploy-tests

build:
  stage: build
  script:
    - ./mvnw package -DskipTests
  artifacts:
    paths:
      - target/*.jar

test:
  stage: test
  script:
    - ./mvnw test

database-migration:
  stage: database-migration
  script:
    - ./mvnw flyway:migrate -Dflyway.url=$DB_URL -Dflyway.user=$DB_USER -
Dflyway.password=$DB_PASSWORD
  environment:
    name: production

deploy:
  stage: deploy
  script:
    - aws elasticbeanstalk update-environment --application-name myapp --
environment-name production --version-label $CI_COMMIT_SHA
  environment:
    name: production

post-deploy-tests:
  stage: post-deploy-tests
  script:
    - ./mvnw verify -Dspring.profiles.active=integration-test
    - curl -f https://myapp-production.example.com/actuator/health
```

```
    environment:
      name: production
```

Infrastructure as Code (IaC)

IaC ensures consistent, reproducible infrastructure deployment:

Terraform for Cloud Infrastructure

```
# AWS RDS deployment example
resource "aws_db_instance" "production" {
  allocated_storage    = 100
  storage_type         = "gp2"
  engine               = "postgres"
  engine_version       = "14"
  instance_class       = "db.m5.large"
  name                 = "myapp"
  username             = var.db_username
  password             = var.db_password
  parameter_group_name = aws_db_parameter_group.postgres14.name
  backup_retention_period = 7
  backup_window        = "03:00-04:00"
  maintenance_window   = "mon:04:00-mon:05:00"
  multi_az             = true
  skip_final_snapshot  = false
  final_snapshot_identifier = "myapp-final-snapshot"

  tags = {
    Environment = "Production"
    Application = "MyApp"
  }
}

# Parameter group with database tuning
resource "aws_db_parameter_group" "postgres14" {
  name   = "myapp-postgres14"
  family = "postgres14"

  parameter {
    name  = "shared_buffers"
    value = "4GB"
  }

  parameter {
    name  = "work_mem"
    value = "64MB"
  }

  parameter {
    name  = "effective_cache_size"
    value = "12GB"
```

```
    }
}
```

Kubernetes Deployment with Helm

For containerized applications, Helm charts provide template-driven deployment:

```
# values.yaml for Helm chart
replicaCount: 3

image:
  repository: myapp
  tag: latest
  pullPolicy: Always

database:
  url: jdbc:postgresql://myapp-postgres:5432/myapp

resources:
  requests:
    cpu: 500m
    memory: 1Gi
  limits:
    cpu: 1000m
    memory: 2Gi

autoscaling:
  enabled: true
  minReplicas: 3
  maxReplicas: 10
  targetCPUUtilizationPercentage: 70

connectionPool:
  minIdle: 5
  maxPoolSize: 20
```

With corresponding deployment templates that use these values for consistent application deployment.

Continuous Monitoring and Alerting

Proactive monitoring prevents problems before they impact users:

Prometheus and Grafana for Metrics Visualization

Configuring Prometheus for Spring Boot monitoring:

```
# prometheus.yml
scrape_configs:
  - job_name: 'spring-boot-app'
    metrics_path: '/actuator/prometheus'
```

```
    scrape_interval: 15s
    static_configs:
      - targets: ['app1:8080', 'app2:8080', 'app3:8080']
```

Key Grafana dashboards for data-centric applications should include:

- Database connection pool utilization
- Query performance by endpoint
- Transaction success/failure rates
- Database errors by type
- Cache hit/miss ratio
- Database lag for replicated databases

Alerting Rules for Database Issues

Prometheus alerting rules for common database problems:

```
groups:
- name: database-alerts
  rules:
  - alert: HighConnectionPoolUtilization
    expr: hikaricp_connections_usage_ratio > 0.8
    for: 5m
    labels:
      severity: warning
    annotations:
      summary: "High connection pool utilization"
      description: "Connection pool utilization is over 80% for 5 minutes."

  - alert: SlowDatabaseQueries
    expr: hikaricp_connections_creation_seconds_max > 1
    for: 5m
    labels:
      severity: warning
    annotations:
      summary: "Slow database connection creation"
      description: "Database connections are taking more than 1 second to create."

  - alert: HighDatabaseErrorRate
    expr: rate(spring_datasource_query_errors_total[5m]) > 0.01
    for: 3m
    labels:
      severity: critical
    annotations:
      summary: "High database error rate"
      description: "Database query error rate is above 1% for the last 3 minutes."
```

Log Aggregation and Analysis

Centralizing logs for efficient troubleshooting:

- **ELK Stack** (Elasticsearch, Logstash, Kibana) for log collection and analysis
- **Fluentd** or **Filebeat** for log collection from containers
- **Graylog** as an alternative for enterprise log management

Spring Boot application configuration for centralized logging:

```
<appender name="LOGSTASH"
class="net.logstash.logback.appender.LogstashTcpSocketAppender">
    <destination>logstash:5000</destination>
    <encoder class="net.logstash.logback.encoder.LogstashEncoder">
        <includeMdc>true</includeMdc>
        <customFields>{"application":"myapp","environment":"production"}
</customFields>
    </encoder>
</appender>
```

Database Change Management in Production

Managing database schema changes safely in production environments:

Schema Migration with Flyway

Flyway provides version-controlled database migrations:

```
@Configuration
public class FlywayConfiguration {

    @Bean
    @Profile("production")
    public FlywayMigrationStrategy flywayMigrationStrategy() {
        return flyway -> {
            // Validate migration scripts before applying
            flyway.validate();

            // Apply migrations
            flyway.migrate();
        };
    }
}
```

Migration scripts follow a versioned naming convention:

```
-- V1__Initial_schema.sql
CREATE TABLE customers (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
```

```
);

-- V2__Add_customer_status.sql
ALTER TABLE customers
ADD COLUMN status VARCHAR(20) NOT NULL DEFAULT 'ACTIVE';
```

Zero-Downtime Database Migrations

For high-availability environments, schema changes without downtime:

1. **Backwards-Compatible Changes**: First deploy application code that works with both old and new schema
2. **Incremental Migration**: Perform schema changes that maintain compatibility
3. **Final Code Deployment**: Deploy application code that fully utilizes the new schema

Example of a backwards-compatible migration approach:

```
-- Step 1: Add new column (doesn't break existing code)
ALTER TABLE customers ADD COLUMN phone_number VARCHAR(20);

-- Step 2: Create trigger to maintain both old and new fields during transition
CREATE TRIGGER sync_contact_info
AFTER UPDATE ON customers
FOR EACH ROW
BEGIN
    IF NEW.contact_info IS NOT NULL AND NEW.phone_number IS NULL THEN
        SET NEW.phone_number = JSON_EXTRACT(NEW.contact_info, '$.phone');
    END IF;

    IF NEW.phone_number IS NOT NULL THEN
        SET NEW.contact_info = JSON_SET(
            COALESCE(NEW.contact_info, '{}'),
            '$.phone',
            NEW.phone_number
        );
    END IF;
END;

-- Step 3 (after code transition): Remove trigger and old column
DROP TRIGGER sync_contact_info;
-- (Only after all application instances use the new column)
ALTER TABLE customers DROP COLUMN contact_info;
```

Database Rollback Strategies

Having a rollback plan for database changes:

- Point-in-time recovery from backups
- Maintaining rollback scripts for each migration
- Blue-green deployment with database cloning

Scaling Data-Centric Spring Boot Applications (continued)

Advanced Scaling Techniques

Event-Driven Architecture for Scaling

Using events to decouple operations and improve scalability:

```
@Service
public class OrderService {

    private final OrderRepository orderRepository;
    private final ApplicationEventPublisher eventPublisher;

    @Autowired
    public OrderService(OrderRepository orderRepository, ApplicationEventPublisher
eventPublisher) {
        this.orderRepository = orderRepository;
        this.eventPublisher = eventPublisher;
    }

    @Transactional
    public Order createOrder(Order order) {
        Order savedOrder = orderRepository.save(order);

        // Publish event instead of calling other services directly
        eventPublisher.publishEvent(new OrderCreatedEvent(savedOrder));

        return savedOrder;
    }
}

@Component
public class InventoryListener {

    private final InventoryService inventoryService;

    @Autowired
    public InventoryListener(InventoryService inventoryService) {
        this.inventoryService = inventoryService;
    }

    @EventListener
    @Async
    public void handleOrderCreatedEvent(OrderCreatedEvent event) {
        // Process inventory changes asynchronously
        inventoryService.reserveInventory(event.getOrder());
    }
}
```

This pattern can be extended with message brokers like RabbitMQ or Kafka for inter-service communication:

```
# Spring Cloud Stream with Kafka
spring.cloud.stream.bindings.orderEvents-out-0.destination=order-events
spring.cloud.stream.bindings.orderEvents-out-0.content-type=application/json
spring.cloud.stream.kafka.binder.brokers=kafka:9092
```

Microservices Data Patterns

When splitting monolithic databases for microservices:

- **Database per Service**: Each service has its own database
- **Shared Database**: Multiple services share a database with schema separation
- **Event Sourcing**: Store state changes as events in an event store
- **CQRS (Command Query Responsibility Segregation)**: Separate read and write models

Implementation example of CQRS:

```
// Command side (write model)
@Service
public class ProductCommandService {

    private final ProductRepository repository;
    private final EventPublisher eventPublisher;

    @Transactional
    public void createProduct(CreateProductCommand command) {
        Product product = new Product(command.getName(), command.getPrice());
        repository.save(product);
        eventPublisher.publish(new ProductCreatedEvent(product.getId(),
product.getName(), product.getPrice()));
    }
}

// Query side (read model)
@Service
public class ProductQueryService {

    private final ProductReadRepository readRepository;

    public ProductDTO getProduct(Long id) {
        return readRepository.findById(id)
            .map(this::mapToDTO)
            .orElseThrow(() -> new ProductNotFoundException(id));
    }

    public List<ProductDTO> searchProducts(String keyword) {
        return readRepository.findByNameContaining(keyword)
            .stream()
            .map(this::mapToDTO)
            .collect(Collectors.toList());
```

```
    }
}
```

Polyglot Persistence

Using different database technologies for different data needs:

```
@Configuration
public class DatabaseConfig {

    @Bean
    @Primary
    @ConfigurationProperties("spring.datasource.relational")
    public DataSource relationaldataSource() {
        return DataSourceBuilder.create().build();
    }

    @Bean
    @ConfigurationProperties("spring.data.mongodb")
    public MongoClient mongoClient() {
        return MongoClients.create();
    }

    @Bean
    @ConfigurationProperties("spring.redis")
    public RedisConnectionFactory redisConnectionFactory() {
        return new LettuceConnectionFactory();
    }
}

@Service
public class ProductService {

    private final ProductRepository sqlRepository;
    private final ProductDetailRepository mongoRepository;
    private final RedisTemplate<String, Product> redisTemplate;

    public Product getProduct(Long id) {
        // Try cache first
        Product cached = redisTemplate.opsForValue().get("product:" + id);
        if (cached != null) {
            return cached;
        }

        // Get base data from SQL
        Product product = sqlRepository.findById(id)
            .orElseThrow(() -> new ProductNotFoundException(id));

        // Enrich with document data from MongoDB
        ProductDetail detail = mongoRepository.findByProductId(id);
        if (detail != null) {
            product.setAttributes(detail.getAttributes());
```

```
            product.setDescription(detail.getDescription());
        }

        // Cache for future requests
        redisTemplate.opsForValue().set("product:" + id, product, 30,
TimeUnit.MINUTES);

        return product;
    }
}
```

Chapter Summary and Key Takeaways

Deploying and monitoring data-centric Spring Boot applications requires careful consideration of numerous factors to ensure reliability, performance, and scalability in production environments.

Key Deployment Considerations

1. **Choose the Right Deployment Strategy**: Select from cloud deployment (AWS, Azure, GCP) or on-premise options based on organizational requirements, considering factors like compliance, existing infrastructure, and operational expertise.

2. **Package Applications Appropriately**: Utilize Spring Boot's executable JARs for simplicity or containerization with Docker for consistency across environments, ensuring proper externalization of database configurations.

3. **Leverage Managed Database Services**: Where possible, use cloud provider managed database services (AWS RDS, Azure SQL, Google Cloud SQL) to reduce operational overhead and benefit from built-in high availability, backup, and scaling features.

4. **Implement Robust Configuration Management**: Externalize configuration using environment variables, property files, or configuration servers, with special attention to secure management of database credentials through dedicated secrets management solutions.

Critical Monitoring and Performance Practices

1. **Comprehensive Metrics Collection**: Implement detailed monitoring of database operations, connection pools, query performance, and transaction metrics using Spring Boot Actuator and integration with monitoring platforms.

2. **Structured Logging for Database Operations**: Enable appropriate SQL logging and implement structured logging practices to facilitate troubleshooting of database-related issues.

3. **Connection Pool Optimization**: Carefully tune connection pool parameters based on application load characteristics, monitoring connection usage patterns to balance resource utilization with performance.

4. **Robust Backup and Recovery Strategies**: Implement and regularly test backup procedures, considering both point-in-time recovery needs and disaster recovery scenarios.

Scaling for Growth

1. **Implement Appropriate Scaling Strategies**: Choose horizontal or vertical scaling approaches based on application architecture, considering read-write splitting, caching, and database sharding for high-scale scenarios.

2. **Adopt Event-Driven Patterns**: Use event-driven architecture to decouple operations and improve scalability, particularly for cross-service data operations.

3. **Consider Polyglot Persistence**: Evaluate using different database technologies for different data needs, matching storage technology to access patterns.

4. **Implement Progressive Database Changes**: Use database migration tools like Flyway to manage schema evolution with version control, planning for zero-downtime migrations in production.

Best Practices for Operational Excellence

1. **Automate Deployment and Infrastructure**: Implement CI/CD pipelines for application deployment and use Infrastructure as Code for consistent, reproducible infrastructure provisioning.

2. **Implement Proactive Monitoring and Alerting**: Set up comprehensive monitoring dashboards and alerting rules to detect and respond to issues before they impact users.

3. **Regular Performance Testing**: Conduct database performance testing under realistic load conditions, identifying and addressing bottlenecks before they impact production.

4. **Documentation and Runbooks**: Maintain detailed documentation of database architecture, scaling decisions, backup procedures, and recovery runbooks to support operational teams.