Advanced Spring Data JPA — Reference Implementation

This document contains a **reference codebase** for the advanced assignment (entities, repositories, DTOs, specifications, auditing, soft deletes, transactions, controllers). Copy the files into a Spring Boot project (Maven) and run with an H2 in-memory DB.

Project structure (recommended)

```
advanced-spring-data-jpa/
├─ src/main/java/com/example/demo/
│   ├─ DemoApplication.java
│   ├─ config/
│   │   └─ JpaConfig.java
│   ├─ controller/
│   │   ├─ CustomerController.java
│   │   └─ OrderController.java
│   ├─ dto/
│   │   ├─ OrderSummaryDto.java
│   │   └─ OrderItemProjection.java
│   ├─ entity/
│   │   ├─ Customer.java
│   │   ├─ Order.java
│   │   ├─ OrderItem.java
│   │   └─ OrderStatus.java
│   ├─ repository/
│   │   ├─ CustomerRepository.java
│   │   ├─ CustomerRepositoryCustom.java
│   │   ├─ CustomerRepositoryImpl.java
│   │   ├─ OrderRepository.java
│   │   └─ OrderSpecifications.java
│   └─ service/
│       ├─ CustomerService.java
│       └─ OrderService.java
└─ src/main/resources/
    └─ application.properties
```

Note: paste each file into your IDE under the paths above. Use Java 17+ and Spring Boot 3.x.

`pom.xml` (minimal essential deps)

```
<project ...>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>advanced-spring-data-jpa</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>
  <properties>
```

```
    <java.version>17</java.version>
    <spring.boot.version>3.2.0</spring.boot.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <optional>true</optional>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>
```

application.properties

```
spring.datasource.url=jdbc:h2:mem:demo;DB_CLOSE_DELAY=-
1;DB_CLOSE_ON_EXIT=FALSE
spring.datasource.username=sa
spring.datasource.password=
spring.datasource.driver-class-name=org.h2.Driver
spring.jpa.hibernate.ddl-auto=update
spring.h2.console.enabled=true
logging.level.org.hibernate.SQL=DEBUG
```

Main application & config

DemoApplication.java

```
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.data.jpa.repository.config.EnableJpaAuditing;

@SpringBootApplication
@EnableJpaAuditing
public class DemoApplication {
```

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

`JpaConfig.java` (enable auditing is above; no extra config required for basic setup)

```
package com.example.demo.config;

import org.springframework.context.annotation.Configuration;

@Configuration
public class JpaConfig {
    // placeholder for additional JPA configuration like custom converters
}
```

Entities

Uses Lombok for brevity. If you don't want Lombok, expand getters/setters/constructors.

`Customer.java`

```
package com.example.demo.entity;

import jakarta.persistence.*;
import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Size;
import lombok.*;
import org.springframework.data.annotation.CreatedDate;
import org.springframework.data.annotation.LastModifiedDate;

import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;
import java.util.UUID;

@Entity
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
@Table(name = "customers", uniqueConstraints =
@UniqueConstraint(columnNames = "email"))
public class Customer {

    @Id
    @Column(columnDefinition = "BINARY(16)")
    private UUID id;

    @NotBlank
    @Size(min = 3)
    private String name;

    @Email
```

```
    @NotBlank
    private String email;

    @CreatedDate
    private LocalDateTime registeredDate;

    private boolean deleted = false; // soft delete flag

    // example extra field used for transaction demo
    private java.math.BigDecimal creditLimit = java.math.BigDecimal.ZERO;

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

    @PrePersist
    public void prePersist() {
        if (id == null) id = UUID.randomUUID();
        if (registeredDate == null) registeredDate = LocalDateTime.now();
    }
}
```

Order.java

```
package com.example.demo.entity;

import jakarta.persistence.*;
import lombok.*;
import org.springframework.data.annotation.CreatedDate;

import java.math.BigDecimal;
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;

@Entity
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
@Table(name = "orders")
public class Order {

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

    private LocalDateTime orderDate;

    @Enumerated(EnumType.STRING)
    private OrderStatus status;

    private BigDecimal totalAmount;

    @ManyToOne
    @JoinColumn(name = "customer_id")
    private Customer customer;

    @OneToMany(mappedBy = "order", cascade = CascadeType.ALL, orphanRemoval
= true)
    private List<OrderItem> items = new ArrayList<>();
```

```
    @PrePersist
    public void prePersist() {
        if (orderDate == null) orderDate = LocalDateTime.now();
        if (status == null) status = OrderStatus.PENDING;
        if (totalAmount == null) totalAmount = BigDecimal.ZERO;
    }
}
```

OrderItem.java

```
package com.example.demo.entity;

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

import java.math.BigDecimal;

@Entity
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
@Table(name = "order_items")
public class OrderItem {

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

    private String productName;
    private int quantity;
    private BigDecimal price;

    @ManyToOne
    @JoinColumn(name = "order_id")
    private Order order;
}
```

OrderStatus.java

```
package com.example.demo.entity;

public enum OrderStatus {
    PENDING, SHIPPED, DELIVERED, CANCELLED
}
```

DTOs & Projections

`OrderSummaryDto.java` (class-based DTO via JPQL constructor expression)

```
package com.example.demo.dto;

import java.math.BigDecimal;

public class OrderSummaryDto {
```

```
    private Long orderId;
    private String customerName;
    private BigDecimal totalAmount;

    public OrderSummaryDto(Long orderId, String customerName, BigDecimal
totalAmount) {
        this.orderId = orderId;
        this.customerName = customerName;
        this.totalAmount = totalAmount;
    }

    // getters
    public Long getOrderId() { return orderId; }
    public String getCustomerName() { return customerName; }
    public BigDecimal getTotalAmount() { return totalAmount; }
}
```

`OrderItemProjection.java` (interface-based projection)

```
package com.example.demo.dto;

public interface OrderItemProjection {
    String getProductName();
    int getQuantity();
}
```

Repositories

`CustomerRepository.java`

```
package com.example.demo.repository;

import com.example.demo.entity.Customer;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.stereotype.Repository;

import java.util.Optional;
import java.util.UUID;

@Repository
public interface CustomerRepository extends JpaRepository<Customer, UUID>,
CustomerRepositoryCustom {

    // exclude soft-deleted customers by default
    @Query("select c from Customer c where c.deleted = false and c.id =
:id")
    Optional<Customer> findByIdAndNotDeleted(UUID id);

    @Query("select c from Customer c where c.deleted = false")
    java.util.List<Customer> findAllNotDeleted();

    boolean existsByEmail(String email);
}
```

`CustomerRepositoryCustom.java` & `CustomerRepositoryImpl.java` (soft delete custom method)

```
package com.example.demo.repository;

import java.util.UUID;

public interface CustomerRepositoryCustom {
    void softDeleteCustomer(UUID id);
}
package com.example.demo.repository;

import com.example.demo.entity.Customer;
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import java.util.UUID;

@Repository
public class CustomerRepositoryImpl implements CustomerRepositoryCustom {

    @PersistenceContext
    private EntityManager em;

    @Override
    @Transactional
    public void softDeleteCustomer(UUID id) {
        Customer c = em.find(Customer.class, id);
        if (c != null) {
            c.setDeleted(true);
            em.merge(c);
        }
    }
}
```

OrderRepository.java

```
package com.example.demo.repository;

import com.example.demo.dto.OrderItemProjection;
import com.example.demo.dto.OrderSummaryDto;
import com.example.demo.entity.Order;
import com.example.demo.entity.OrderStatus;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.jpa.repository.JpaSpecificationExecutor;
import org.springframework.stereotype.Repository;

import java.util.List;

@Repository
public interface OrderRepository extends JpaRepository<Order, Long>,
JpaSpecificationExecutor<Order> {

    // 1. class-based DTO projection via JPQL
    @Query("select new com.example.demo.dto.OrderSummaryDto(o.id,
o.customer.name, o.totalAmount) from Order o")
    List<OrderSummaryDto> findOrderSummaries();

    // 2. interface projection for order items
```

```
    @Query("select i.productName as productName, i.quantity as quantity
from OrderItem i where i.order.id = :orderId")
    List<OrderItemProjection> findItemsByOrderId(Long orderId);

    // 3. JPQL sorted
    @Query("select o from Order o order by o.totalAmount desc")
    List<Order> findAllOrderByTotalAmountDesc();

    // 4. native query example: highest total per order (returns order id)
    @Query(value = "SELECT * FROM orders o WHERE o.total_amount = (SELECT
MAX(total_amount) FROM orders)", nativeQuery = true)
    Order findOrderWithHighestTotalNative();

    Page<Order> findAll(Pageable pageable);
}
```

Specifications (dynamic filtering)

OrderSpecifications.java

```
package com.example.demo.repository;

import com.example.demo.entity.Order;
import com.example.demo.entity.OrderStatus;
import org.springframework.data.jpa.domain.Specification;

import java.math.BigDecimal;
import java.time.LocalDateTime;

public class OrderSpecifications {

    public static Specification<Order> hasStatus(OrderStatus status) {
        return (root, query, cb) -> status == null ? null :
cb.equal(root.get("status"), status);
    }

    public static Specification<Order>
totalAmountGreaterThanOrEqual(BigDecimal min) {
        return (root, query, cb) -> min == null ? null :
cb.greaterThanOrEqualTo(root.get("totalAmount"), min);
    }

    public static Specification<Order>
totalAmountLessThanOrEqual(BigDecimal max) {
        return (root, query, cb) -> max == null ? null :
cb.lessThanOrEqualTo(root.get("totalAmount"), max);
    }

    public static Specification<Order> orderDateBetween(LocalDateTime
start, LocalDateTime end) {
        return (root, query, cb) -> {
            if (start == null && end == null) return null;
            if (start != null && end != null) return
cb.between(root.get("orderDate"), start, end);
            if (start != null) return
cb.greaterThanOrEqualTo(root.get("orderDate"), start);
            return cb.lessThanOrEqualTo(root.get("orderDate"), end);
        };
```

```
    }

    public static Specification<Order> customerNameLike(String namePart) {
        return (root, query, cb) -> {
            if (namePart == null || namePart.isBlank()) return null;
            return cb.like(cb.lower(root.get("customer").get("name")), "%"
+ namePart.toLowerCase() + "%");
        };
    }
}
```

Services (transaction demo & business logic)

`CustomerService.java`

```
package com.example.demo.service;

import com.example.demo.entity.Customer;
import com.example.demo.repository.CustomerRepository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.UUID;

@Service
public class CustomerService {

    private final CustomerRepository customerRepository;

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

    public Customer save(Customer c) {
        return customerRepository.save(c);
    }

    public void softDelete(UUID id) {
        customerRepository.softDeleteCustomer(id);
    }
}
```

`OrderService.java` **(transactional composite operation)**

```
package com.example.demo.service;

import com.example.demo.entity.Customer;
import com.example.demo.entity.Order;
import com.example.demo.repository.CustomerRepository;
import com.example.demo.repository.OrderRepository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.math.BigDecimal;
import java.util.UUID;

@Service
public class OrderService {
```

```
    private final OrderRepository orderRepository;
    private final CustomerRepository customerRepository;

    public OrderService(OrderRepository orderRepository, CustomerRepository
customerRepository) {
        this.orderRepository = orderRepository;
        this.customerRepository = customerRepository;
    }

    // transactional operation: deduct credit and create order
    @Transactional
    public Order createOrderAndDeduct(UUID customerId, Order order,
BigDecimal deductAmount, boolean throwAfterSave) {
        Customer c = customerRepository.findByIdAndNotDeleted(customerId)
                .orElseThrow(() -> new IllegalArgumentException("Customer
not found"));

        // deduct credit limit
        c.setCreditLimit(c.getCreditLimit().subtract(deductAmount));
        customerRepository.save(c);

        // set relation and save order
        order.setCustomer(c);
        Order saved = orderRepository.save(order);

        // optionally trigger rollback
        if (throwAfterSave) {
            throw new RuntimeException("Simulated failure after saving
order — should rollback everything");
        }

        return saved;
    }
}
```

Controllers (sample endpoints)

CustomerController.java

```
package com.example.demo.controller;

import com.example.demo.entity.Customer;
import com.example.demo.service.CustomerService;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;
import java.util.UUID;

@RestController
@RequestMapping("/api/customers")
public class CustomerController {

    private final CustomerService customerService;

    public CustomerController(CustomerService cs) { this.customerService =
cs; }
```

```
    @PostMapping
    public ResponseEntity<Customer> createCustomer(@RequestBody Customer c)
{
        return ResponseEntity.ok(customerService.save(c));
    }

    @GetMapping
    public ResponseEntity<List<Customer>> list() {
        // use repository findAllNotDeleted in real app via service
        throw new UnsupportedOperationException("Implement listing via
repository/service");
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> softDelete(@PathVariable UUID id) {
        customerService.softDelete(id);
        return ResponseEntity.noContent().build();
    }
}
```

Note: controller listing method is left intentionally minimal — adapt it to your app.

OrderController.java

```
package com.example.demo.controller;

import com.example.demo.dto.OrderItemProjection;
import com.example.demo.dto.OrderSummaryDto;
import com.example.demo.entity.Order;
import com.example.demo.entity.OrderStatus;
import com.example.demo.repository.OrderRepository;
import com.example.demo.repository.OrderSpecifications;
import com.example.demo.service.OrderService;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort;
import org.springframework.data.jpa.domain.Specification;
import org.springframework.format.annotation.DateTimeFormat;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.math.BigDecimal;
import java.time.LocalDateTime;
import java.util.List;
import java.util.UUID;

@RestController
@RequestMapping("/api/orders")
public class OrderController {

    private final OrderRepository orderRepository;
    private final OrderService orderService;

    public OrderController(OrderRepository orderRepository, OrderService
orderService) {
        this.orderRepository = orderRepository;
        this.orderService = orderService;
    }
```

```
    @GetMapping("/summaries")
    public ResponseEntity<List<OrderSummaryDto>> summaries() {
        return ResponseEntity.ok(orderRepository.findOrderSummaries());
    }

    @GetMapping("/{id}/items")
    public ResponseEntity<List<OrderItemProjection>> items(@PathVariable
Long id) {
        return ResponseEntity.ok(orderRepository.findItemsByOrderId(id));
    }

    @GetMapping("/search")
    public ResponseEntity<Page<Order>> search(
            @RequestParam(defaultValue = "0") int page,
            @RequestParam(defaultValue = "10") int size,
            @RequestParam(defaultValue = "orderDate") String sortField,
            @RequestParam(defaultValue = "desc") String sortDir,
            @RequestParam(required = false) OrderStatus status,
            @RequestParam(required = false) BigDecimal minAmount,
            @RequestParam(required = false) BigDecimal maxAmount,
            @RequestParam(required = false) String customerNameContains,
            @RequestParam(required = false) @DateTimeFormat(iso =
DateTimeFormat.ISO.DATE_TIME) LocalDateTime startDate,
            @RequestParam(required = false) @DateTimeFormat(iso =
DateTimeFormat.ISO.DATE_TIME) LocalDateTime endDate
    ) {
        Sort sort = Sort.by(Sort.Direction.fromString(sortDir), sortField);
        Pageable pageable = PageRequest.of(page, size, sort);

        Specification<Order> spec = Specification.where(
                OrderSpecifications.hasStatus(status)
        ).and(OrderSpecifications.totalAmountGreaterThanOrEqual(minAmount))

.and(OrderSpecifications.totalAmountLessThanOrEqual(maxAmount))

.and(OrderSpecifications.customerNameLike(customerNameContains))
                .and(OrderSpecifications.orderDateBetween(startDate,
endDate));

        Page<Order> result = orderRepository.findAll(spec, pageable);
        return ResponseEntity.ok(result);
    }

    // demo transactional endpoint
    @PostMapping("/create/{customerId}")
    public ResponseEntity<Order> createAndDeduct(@PathVariable UUID
customerId,
                                                 @RequestBody Order order,
                                                 @RequestParam BigDecimal
deduct,
                                                 @RequestParam(defaultValue
= "false") boolean failAfterSave) {
        Order saved = orderService.createOrderAndDeduct(customerId, order,
deduct, failAfterSave);
        return ResponseEntity.ok(saved);
    }
}
```

Testing tips

1. Start the app and open `http://localhost:8080/h2-console` (JDBC URL: `jdbc:h2:mem:demo`).
2. Use Postman to create customers and orders. Example create Customer JSON:

```
{
  "name": "Alice Example",
  "email": "alice@example.com",
  "creditLimit": 1000
}
```

3. Create an order payload that includes items:

```
{
  "items": [
    {"productName": "Widget A", "quantity": 2, "price": 10.5},
    {"productName": "Widget B", "quantity": 1, "price": 5.0}
  ],
  "totalAmount": 26.0
}
```

4. Call `/api/orders/create/{customerId}?deduct=100&failAfterSave=false` to create the order and deduct 100. Set `failAfterSave=true` to simulate a failure and verify rollback.
5. Use `/api/orders/search` with params `page`, `size`, `sortField`, `sortDir`, `status`, `minAmount`, `maxAmount`, `customerNameContains`, `startDate`, `endDate`.

Extra credit examples you can add

- Add `@EntityGraph(attributePaths = {"items"})` on repository methods to avoid N+1 when loading orders with items.
- Add a native query to compute total revenue per customer: `SELECT c.id, c.name, SUM(o.total_amount) FROM customers c JOIN orders o ON c.id = o.customer_id GROUP BY c.id, c.name` and map to a projection.
- Create a read-only JPA `@Entity` mapped to a DB view.