**1. Overview and Setup**

**Explanation:**  
You start by initializing a Spring Boot project with the necessary dependencies. This sets up the foundation for your application, providing tools for data persistence (JPA), in-memory database (H2), web APIs, and code simplification (Lombok).

**Code: application.properties**

spring.datasource.url=jdbc:h2:mem:testdb  
spring.datasource.driverClassName=org.h2.Driver  
spring.datasource.username=sa  
spring.datasource.password=password  
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

*This configures an in-memory H2 database for development and testing.*

**2. Creating Entities**

**Explanation:**  
Entities represent your data model. Each entity maps to a database table. Relationships (like one-to-many) are established via annotations.

**Code: Department.java**

@Entity  
public class Department {  
 @Id @GeneratedValue  
 private Long id;  
 private String name;  
  
 @OneToMany(mappedBy = "department", cascade = CascadeType.ALL)  
 private List<Employee> employees = new ArrayList<>();  
}

**Code: Employee.java**

@Entity  
public class Employee {  
 @Id @GeneratedValue  
 private Long id;  
 private String name;  
 private String email;  
  
 @ManyToOne  
 @JoinColumn(name = "department\_id")  
 private Department department;  
}

*This sets up a one-to-many relationship: a department has many employees, each employee belongs to one department.*

**3. Creating Repositories**

**Explanation:**  
Repositories provide CRUD operations without boilerplate code. Extending JpaRepository gives you ready-to-use methods.

**Code: EmployeeRepository.java**

public interface EmployeeRepository extends JpaRepository<Employee, Long> {  
 List<Employee> findByDepartmentName(String name); // Derived query method  
}

**Code: DepartmentRepository.java**

public interface DepartmentRepository extends JpaRepository<Department, Long> {}

*You can define custom queries by method names or using annotations.*

**4. Implementing CRUD Operations**

**Explanation:**  
Controllers handle HTTP requests and use repositories/services to perform CRUD operations.

**Code: EmployeeController.java**

@RestController  
@RequestMapping("/employees")  
public class EmployeeController {  
 @Autowired  
 private EmployeeRepository employeeRepository;  
  
 @PostMapping  
 public Employee create(@RequestBody Employee emp) {  
 return employeeRepository.save(emp);  
 }  
  
 @GetMapping("/{id}")  
 public Employee get(@PathVariable Long id) {  
 return employeeRepository.findById(id).orElseThrow();  
 }  
  
 @PutMapping("/{id}")  
 public Employee update(@PathVariable Long id, @RequestBody Employee emp) {  
 emp.setId(id);  
 return employeeRepository.save(emp);  
 }  
  
 @DeleteMapping("/{id}")  
 public void delete(@PathVariable Long id) {  
 employeeRepository.deleteById(id);  
 }  
}

*This exposes RESTful endpoints for create, read, update, and delete operations.*

**5. Defining Query Methods**

**Explanation:**  
Spring Data JPA allows you to define queries by method names or use the @Query annotation for custom SQL/JPQL.

**Code: Custom Query Example**

public interface EmployeeRepository extends JpaRepository<Employee, Long> {  
 List<Employee> findByNameContaining(String keyword); // Derived query  
  
 @Query("SELECT e FROM Employee e WHERE e.email = ?1")  
 Employee findByEmail(String email); // JPQL custom query  
}

*This enables flexible data retrieval beyond basic CRUD.*

**6. Pagination and Sorting**

**Explanation:**  
Pagination and sorting help manage large datasets efficiently and improve API usability.

**Code: Pagination & Sorting Endpoint**

@GetMapping  
public Page<Employee> list(  
 @RequestParam(defaultValue = "0") int page,  
 @RequestParam(defaultValue = "10") int size,  
 @RequestParam(defaultValue = "id") String sortBy  
) {  
 Pageable pageable = PageRequest.of(page, size, Sort.by(sortBy));  
 return employeeRepository.findAll(pageable);  
}

*Clients can request specific pages and sort results by any field.*

**7. Enabling Entity Auditing**

**Explanation:**  
Auditing tracks who created or modified records and when. Requires enabling JPA auditing in your configuration.

**Code: Auditing Fields in Entity**

@EntityListeners(AuditingEntityListener.class)  
@Entity  
public class Employee {  
 @Id @GeneratedValue  
 private Long id;  
 private String name;  
 private String email;  
  
 @CreatedDate  
 private LocalDateTime createdDate;  
  
 @LastModifiedDate  
 private LocalDateTime modifiedDate;  
}

**Code: Enable Auditing**

@SpringBootApplication  
@EnableJpaAuditing  
public class EmployeeManagementSystemApplication {}

*This automatically fills in auditing fields on entity changes.*

**8. Creating Projections**

**Explanation:**  
Projections fetch only specific fields, improving performance and security.

**Code: Interface-Based Projection**

public interface EmployeeNameEmail {  
 String getName();  
 String getEmail();  
}  
  
public interface EmployeeRepository extends JpaRepository<Employee, Long> {  
 List<EmployeeNameEmail> findByDepartmentId(Long deptId);  
}

*Only name and email are fetched for each employee, not the entire entity.*

**9. Customizing Data Source Configuration**

**Explanation:**  
Externalizing configuration makes it easy to switch databases or manage multiple data sources.

**Code: application.properties (for multiple profiles)**

# application-dev.properties  
spring.datasource.url=jdbc:h2:mem:devdb  
  
# application-prod.properties  
spring.datasource.url=jdbc:mysql://localhost/proddb

*Spring Boot selects the correct configuration based on the active profile.*

**10. Hibernate-Specific Features**

**Explanation:**  
Hibernate offers advanced features like batch processing and custom mappings for performance tuning.

**Code: Batch Processing Config**

spring.jpa.properties.hibernate.jdbc.batch\_size=30

**Code: Custom Mapping Example**

@Entity  
@org.hibernate.annotations.DynamicUpdate  
public class Employee {  
 // Only changed columns are included in SQL UPDATE statements  
}

*Batch size improves bulk operation performance; DynamicUpdate optimizes update queries.*

**Summary:**  
Each step builds on the previous, from project setup to advanced data management, ensuring your Employee Management System is robust, efficient, and maintainable. The provided code snippets and explanations help you understand both the "how" and "why" behind each implementation.

⁂