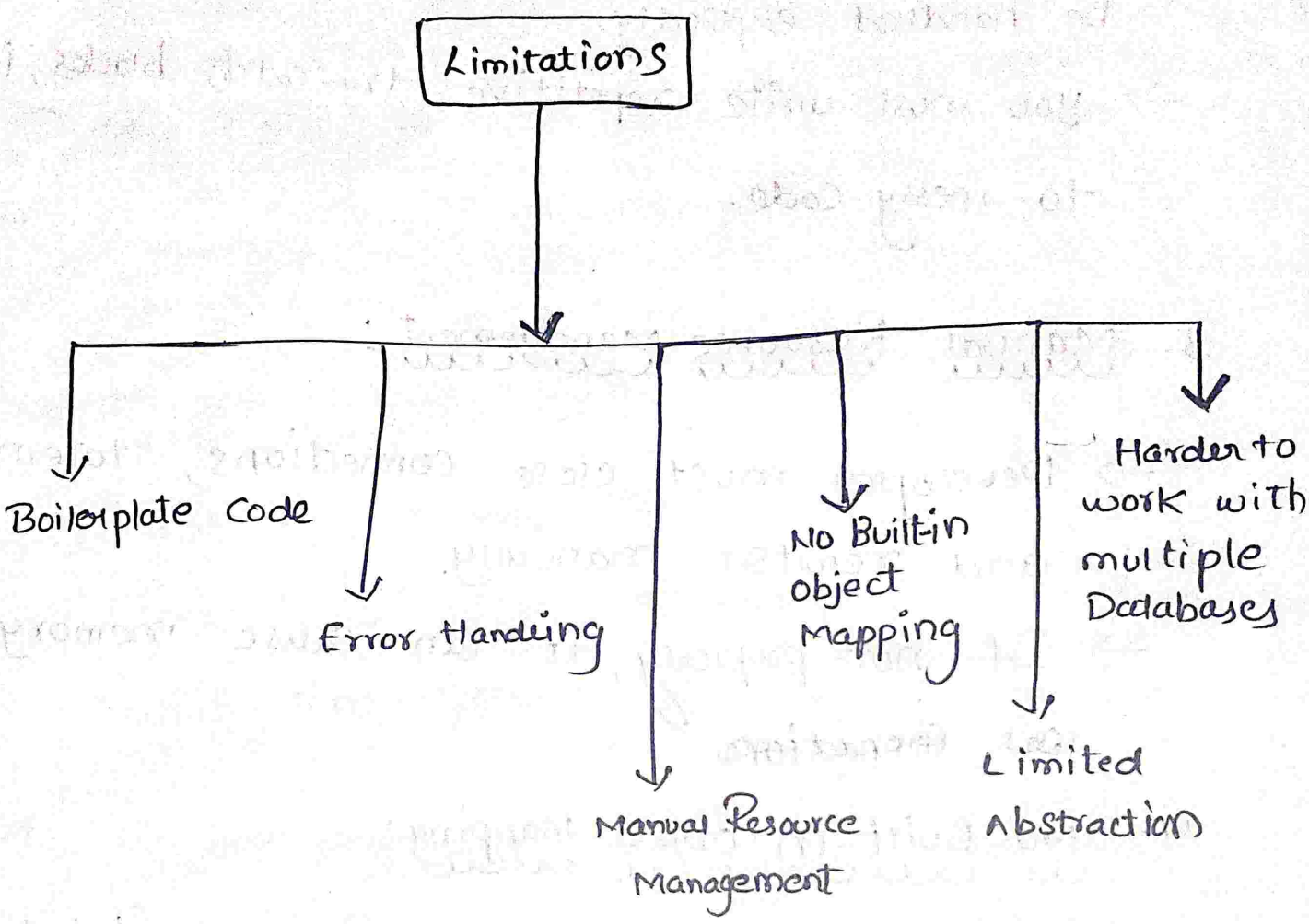
# UNIT-3

## * Limitations of JDBC API (Spring Boot):

→ The Limitations of JDBC (Java DataBase connectivity) in SpringBoot are




### 1. Boilerplate Code:

→ JDBC require alot of manual work:

- ↳ Opening Connection
- ↳ Creating Statements
- ↳ Handling Exceptions
- ↳ Closing Connection etc

→ This makes the code lengthy and Hard to maintain.

2. Error Handling is complex:

→ JDBC throws checked SQL Exception which must be handled explicitly.

→ you must write repetitive try-catch blocks, leading to messy code.

3. Manual Resource Management:

→ Developer must close connections, statements and resultset manually.

→ If not properly, it can cause memory leaks. (Or) ~~Connections~~

4. NO Built-in object Mapping:

→ JDBC returns data as Result set (tabular format), and you have to manually map this to Java objects.

→ No Support for ORM (Object Relational Mapping).

5. Limited Abstraction:

→ JDBC is a low level API. It doesn't provide abstraction for transactions, connection pooling (or) object mapping.

→ You need to configure all these features Separately (or) use libraries.

6. Harder to work with Multiple Databases:

→ SQL Syntax may vary between databases.

→ JDBC code often becomes tightly coupled with the Specific database being used.

7. Slower Development

→ Due to all the above limitations writing and maintaining JDBC code takes more time and effort.

Spring Boot alternatives:

→ TO overcome JDBC limitations, Spring Boot provides:

(1) Spring JDBC template :- Reduces boiler plate code.

(2) Spring Data JPA / Hibernate - provides ORM and powerful repository Support

(3) Spring Transaction Management - abstracts transaction handling.

# * Pagination and Sorting in Spring Boot:

## Pagination:

→ Dividing large amount of data into smaller "pages".

Reduces memory usage and improves performance.

## Sorting:

→ Ordering the results based on one (or) more columns

Example:

byname, bydate

## Spring Data JPA Support:

→ Spring data JPA provides built in classes for this:

→ Pageable - for Pagination

→ Sort - for sorting

## Example Setup

1. Entity class

```
@Entity
public class Product
{
    @Id
    @GeneratedValue(Strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private Double price;
        //Getters and Setters
}
```

2. Repository Interface

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

}
```

3. Controller – pagination & Sorting :

```
@RestController
@RequestMapping("/products")
Public class ProductController
{
private ProductRepository productRepository

  // pagination + Sorting

    @GetMapping

Public Page<product> getProducts(
@RequestParam(defaultValue = "0") int page,
@RequestParam(defaultValue = "5") int size,
@RequestParam(defaultValue = "id") String sortBy )
    {

Pageable pageable = PageRequest.of(page, size,
                    Sort.by(SortBy));

return ProductRepository.findAll(pageable);

}
}
```

Represents the content of a single page.

To request paged data

To sort by a field

## Sample URL Requests:

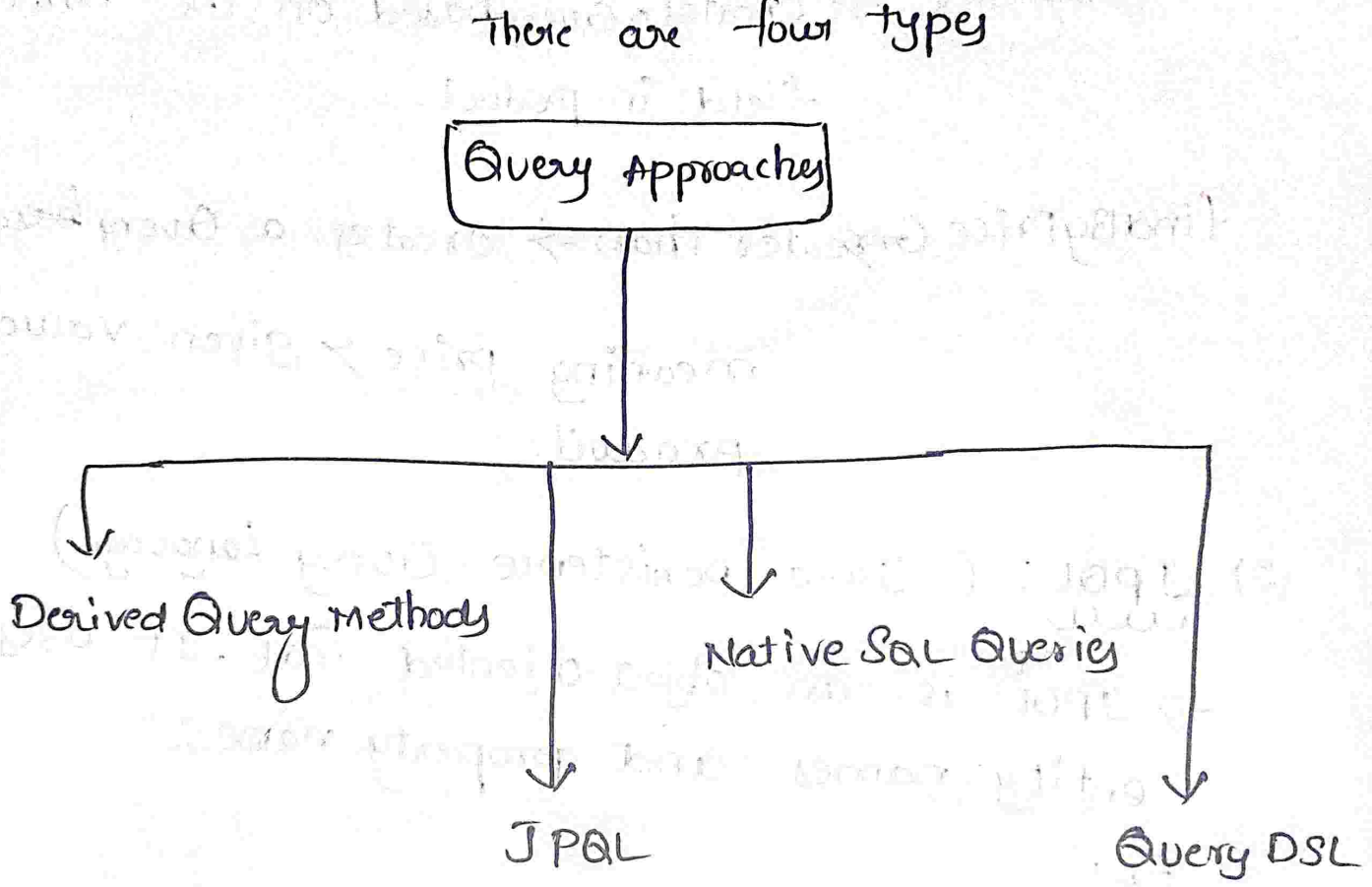
1. /products → Default : page 0, size 5, sorted by id
2. /products?page = 1 & size = 10 → page 1, 10 items per page.
3. /products ? page = 0 & size = 5 & sortBy = name → page 0, sorted by name

page object contain

↳ getContent() → list of data on the current page
↳ getTotalPages() → Total number of pages
↳ getTotalElements() → Total number of records
↳ getNumber() → current page number

* Query approaches in Spring Boot:

→ In Spring Boot (using Spring Data JPA), there are several main Query approaches for retrieving data from the database.




(1) Derived Query Methods (Queries based on method names)

→ Spring data JPA automatically generates SQL Queries based on the method name.

Example:

- List<product> findByName(String name);
- List<product> findByPriceGreaterThan(Double price);

findByName → creates a Query based on the 'name' field in product.

findByPriceGreaterThan → creates a Query ~~based~~ meaning price > given value in product.

(2) JPQL: (Java persistence Query Language)

→ JPQL is an object oriented SQL. It uses entity names and property names.

Ex:

```
@Query("SELECT p From product p where P.price
        > :price);
List<Product>
getProductCostlierThan(@Param("price") Double price);
```

@Query → used to provide a custom Query to Spring Data.

Price → This is a named parameter

Product → Refers to Entity.

## Native SQL Queries:

→ you can directly write normal SQL using native Query = true.

Ex:

```
@Query( value = "SELECT * FROM product WHERE price=?1",
        nativeQuery= true)
List<product> getProductNative (Double price);
```

native Query → true

- native Query = true → indicates that this is a manually written SQL Query.
- ?1 → Refers to the first parameter.

## (4) Query DSL (Dynamic Queries)

→ when dynamic conditions are needed, Specification is used.

Ex:

```
interface ProductRepository extends JpaRepository
<product, Long>, JpaSpecificationExecutor<product>{}
```

→ JpaSpecificationExecutor<T> → used to filter based on Specification.

# *Spring Data JPA with Spring Boot:

→ JPA (Java persistence API) is a Specification used for ORM (Object Relational Mapping).

→ Spring Data JPA is a module in Spring Boot that makes database operations very simple by reducing boilerplate code.

→ Hibernate is the default JPA implementation.

## features:

→ Automatic CRUD operations.

→ Less code, fast development

→ Derived Query methods

→ Supports JPQL

→ Pagination & Sorting built-in

→ works Smoothly with Spring Boot auto configuration

## Important Annotations

→ @ Entity – Marks class as database table.

→ @ Id – primary key

→ @ GeneratedValue – AutoID generation

→ @ Table / Column – custom table/column mapping

→ @ Repository – Repository @ layer class.

## Basic Setup (Pom.xml)

→ Spring-boot-starter-data-jpa

→ mySQL – Connector-j

## application.properties

→ Spring.jpa.hibernate.ddl-auto = update

Spring.jpa.show-sql = true

## Entity Example:

```
@Entity
Public Class Student
{ @Id @GeneratedValue
  private int id;
  private String name;
}
```

# Repository Example

```
public interface StudentRepo extends
                    JpaRepository<Student, Integer>
{
}
```

## Common CRUD Operations:

→ Save (Entity) - insert/update

→ findAll() - All records

→ findById (id) - Single record

→ deleteById (id) - Delete record

## Derived Query Methods

→ findByName (String Name);

→ findByCity (String city);

## JPQL Example:

```
@Query("Select s from Student s where s.city=?1")
List<Student> getByCity (String city);
```

# * Spring Data JPA Configuration:

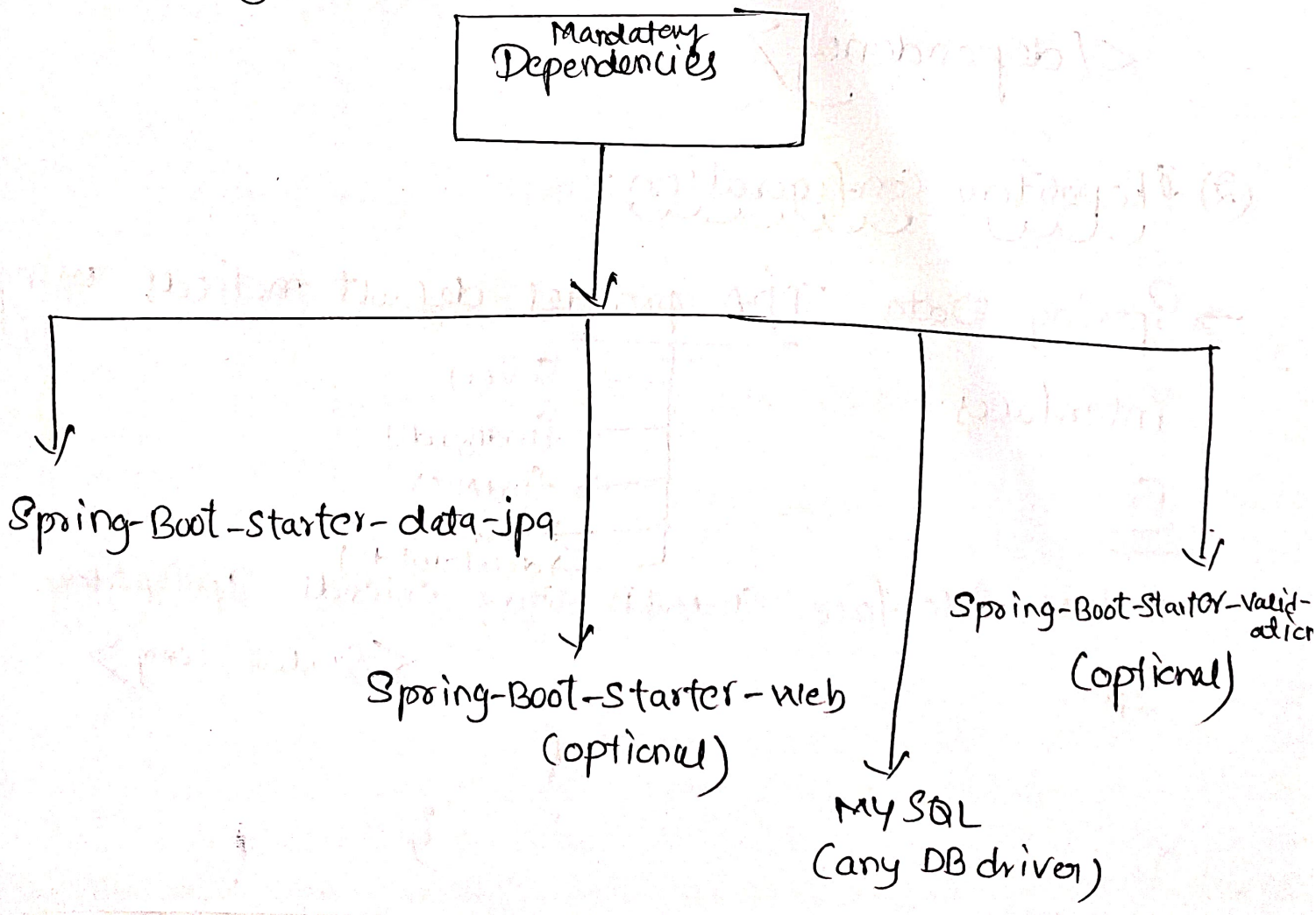
→ Spring Data JPA is a part of Spring framework that Simplifies database operations.

→ It helps developer perform CRUD operations without writing boilerplate JDBC code.

## (1) Required Dependencies:

→ In Spring Boot, add the following dependencies in pom.xml:

Mandatory Dependencies:



Example (pom.xml)

xml

```
<dependency>
<groupId> org.Springframework.boot </groupId>
<artifact Id> Spring-boot-Starter-data-Jpa </artifact Id>
</dependency>

<dependency>
<groupId> mySQL </groupId>
<artifact Id> mySQL-connector-j </artifact Id>
</dependency>
```

(2) IRepository configuration:

→ Spring Data JPA provides default methods using interfaces.

- save()
- findById()
- findAll()
- count()
- deleteById()

Ex

```
public interface StudentRepository extends JpaRepository
                                        <Student, Long>
{
}
```

## (3) Entity class Configuration:

Use @Entity to map class to database table.

fx:

```
@Entity
public class Student
    {
        @Id
        @GeneratedValue(Strategy = GenerationType.IDENTITY)

        private Long id;

        private String name;
        private String email;
    }
```

@Entity – Marks class as table

@Id – primary key

@GeneratedValue – Auto generated ID

@Column – Customize column

@Table – Custom table name

# * Spring Declarative Transaction Management:

## Transadion

→ A Transadion is a group of operations that should be executed as a single unit.

Either all succeed (or) all fail.

### Acid properties

→ A – Atomicity: All steps acts as one

→ C – consistency: Data should remain valid.

→ I – Isolation: one transaction does not affect another.

→ D – Durability: Data is saved even after system failure.

## Declarative Transaction Management:

→ Declarative = using annotations, not manual code.

### we tey Spring:

→ Manage transactions for me using @Transactional

No need to write SQL BEGIN, COMMIT, ROLLBACK.

Spring Boot automatically configures:

→ Transaction Manager

Ex: Jpa TransactionManager

→ platform TransactionManager bean

So, you need to use @Transactional

@Transactional Annotation:

→ Service methods

→ class level

→ Sometimes repository custom methods.

Example

```
@Transactional
public void registerUser(User user)
    {
    UserRepository.save(user);
    accountRepository.createAccount(user.getId());
    }
```

→ If any step fails → Spring rolls back the entire Transaction.

# * update operations in Spring Data JPA:

→ update means modifying an existing row (record) in the database table.

Ex

changing name, email, salary etc.

→ Spring Data JPA supports update in 3 main ways:

(1) update using Entity save()

(2) update using @Query (JPQL / Native SQL)

(3) update using Dirty checking (automatic update)

## (1) update using save() method:

→ If an entity already exists (same ID), save() Performs update.

Ex:

```
Student s = studentRepository.findById(1).get();
s.setName("paramesh");
studentRepository.save(s);
```

→ Find Student with ID 1
→ Changes the Name
→ save() updates the record.

## (2) Updating using JPQL @Query (Custom update)

→ you dont want to fetch the object first.

→ you want faster update

**Need**

@ Modifying

@ Transactional.

**Ex**

```
@ Modifying
@ Transactional
@ Query ("update Student s set s.name = :name
                where s.id = :id")
int updateStudentName (int id, String name);
```

**Usage**

```
StudentRepository.updateStudentName (1, "vaibhav");
```

→ Fast because it does not load the entity

→ Direct update Query

) Dirty checking (Automatic update)

→ Spring data JPA automatically detects changes when inside a transaction.

Ex

```
@Transactional
public void updateMark()
{
Student s = Student.RepositoryofindBYId(1).get();

s.setMarks(90);
}
```

→ No Save() required

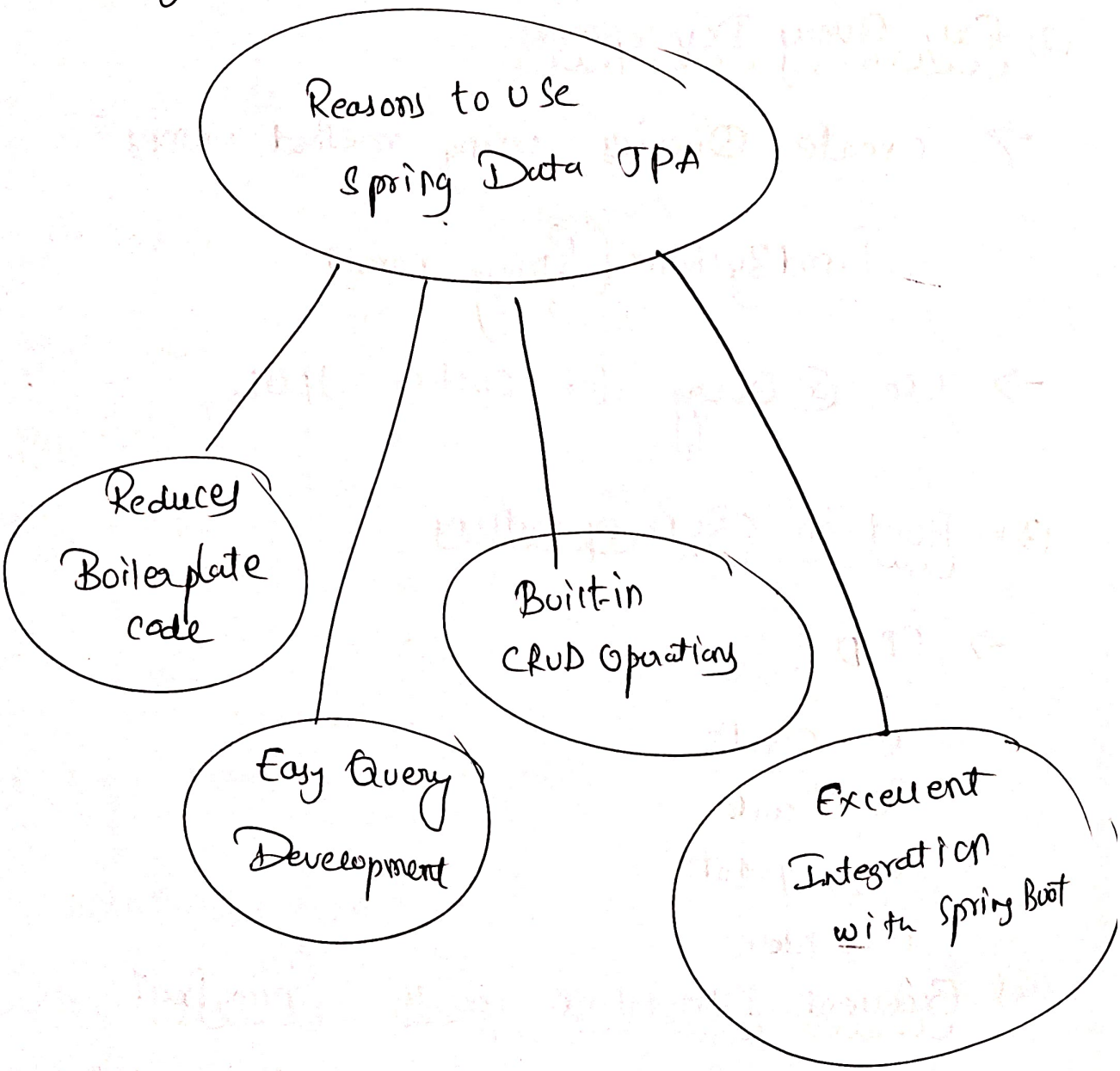
→ JPA sees the change & updates automatically.

UNit-3

# * Why Spring Data JPA?

→ Spring data JPA is apart of the Spring framework used to simplify database operations.

→ It works on top of JPA and removes the need for writing boilerplate SQL and JDBC code.



(1) Reduce Boilerplate code:

→ No Need to write JDBC code, Result set handling (or) SQL Queries.

→ CRUD methods like Save(), findAll(), delete() are automatically provided.

(2) Easy Query Development

→ Create Queries using method names:

findByName(String name)

→ use @Query for custom JPQL

(3) Built-in CRUD operations

→ CRUD

C - Create
R - Reade
U - update
D - Delete

(4) Excellent Integration with Spring Boot:

→ Auto Configuration for database drivers & Entity manager.

→ works Smoothly with Hibernate.

# Clean Architecture

Controller
↓
Service
↓
Repository

→ Improves maintainability & Readability

## Use SpringData JPA:

→ you want Quicle development.
→ you need automatic CRUD.
→ you want cleaner code.
→ you need Scalable Enterprise applications

Spring Data JPA Simplifies database access by removing boilerplate code, providing automatic CRUD operations, Supporting Easy Query creation and integrating well with Spring Boot.

# * Named Query and Query in Spring data JPA:

## Named Query:

→ A Named Query is a predefined Query written in Entity class (or) XML and assigned a fixed name.

→ Spring Data JPA will look for this Query by that name

## where to define:

→ inside entity class using @Named Query

→ inside orm.xml

### Example

```
@Entity
@NamedQuery(
        name = "Student.findByName",
Query = "SELECT s FROM Student s where
                s.name = :name")
public class Student
{
@Id
private Long id;
private String name; }
```

## Usage in Repository:

```
public interface StudentRepository extends JpaRepository<Student,
                                                    Long>
{
    List<Student> findByName(String name);
}
```

→ Here, Spring automatically uses the named Query Student.findByName.

### Advantages of Named Query:

→ Faster performance
→ Easy to reuse
→ Good for complex Queries
→ Validated at Startup

@Query:

→ @Query is used to write custom SQL directly inside Repository interface.

Ex

```
public interface StudentRepository extends JpaRepository
                                        <Student, Long>
{
    @Query("SELECT s FROM Student s WHERE s.name=:name")
    List<Student> searchByName(String name);
}
```

Native SQL:

```
@Query(value = "SELECT * FROM students WHERE name
                = :name", nativeQuery = true)
List<Student> searchNative(String name);
```

→ @Query: write Query directly in repository, easy, flexible.

# * Why Spring Transaction:

## - Transaction:

→ A Transaction is a group of operations that must be treated as a single unit of work.

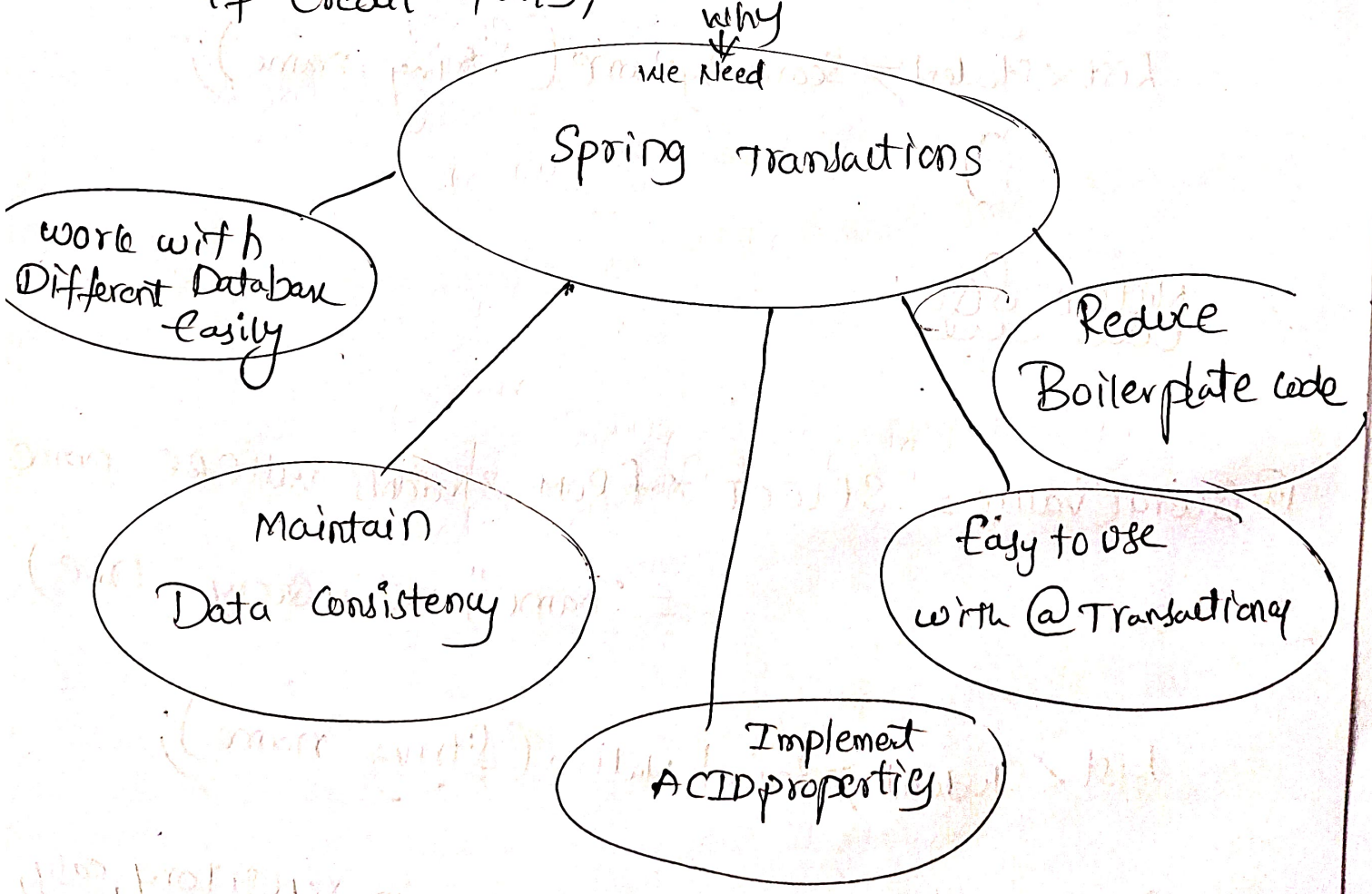
Example

- Transferring money from Account A → Account B

(1) Debit A

(2) Credit B

if credit fails, debit must rollback.



(1) To Maintain Data consistency:

→ If one operation fails, Spring rollbacks automatically.

↳ No partial updates

↳ prevents data corruption.

(2) To implement ACID properties:

Spring helps achieve:

Atomicity — All (or) None

Consistency — Safe & valid data

Isolation — one transaction won't affect another

Durability — once committed, data is saved permanently.

(3) Easy to use with @Transactional

```
@Transactional
public void transfer()
{
    //multiple DataBase Operations
}
```

→ Spring automatically controls commit, Rollback & connection Handling.

(4) Reduces Boiler plate code:

→ without Spring, developers manually write:

- ↳ begin transaction
- ↳ commit
- ↳ rollback
- ↳ try-catch-finally

→ Spring removes all this, you only focus on bussiness logic

(5) Work with Different DataBase Easily:

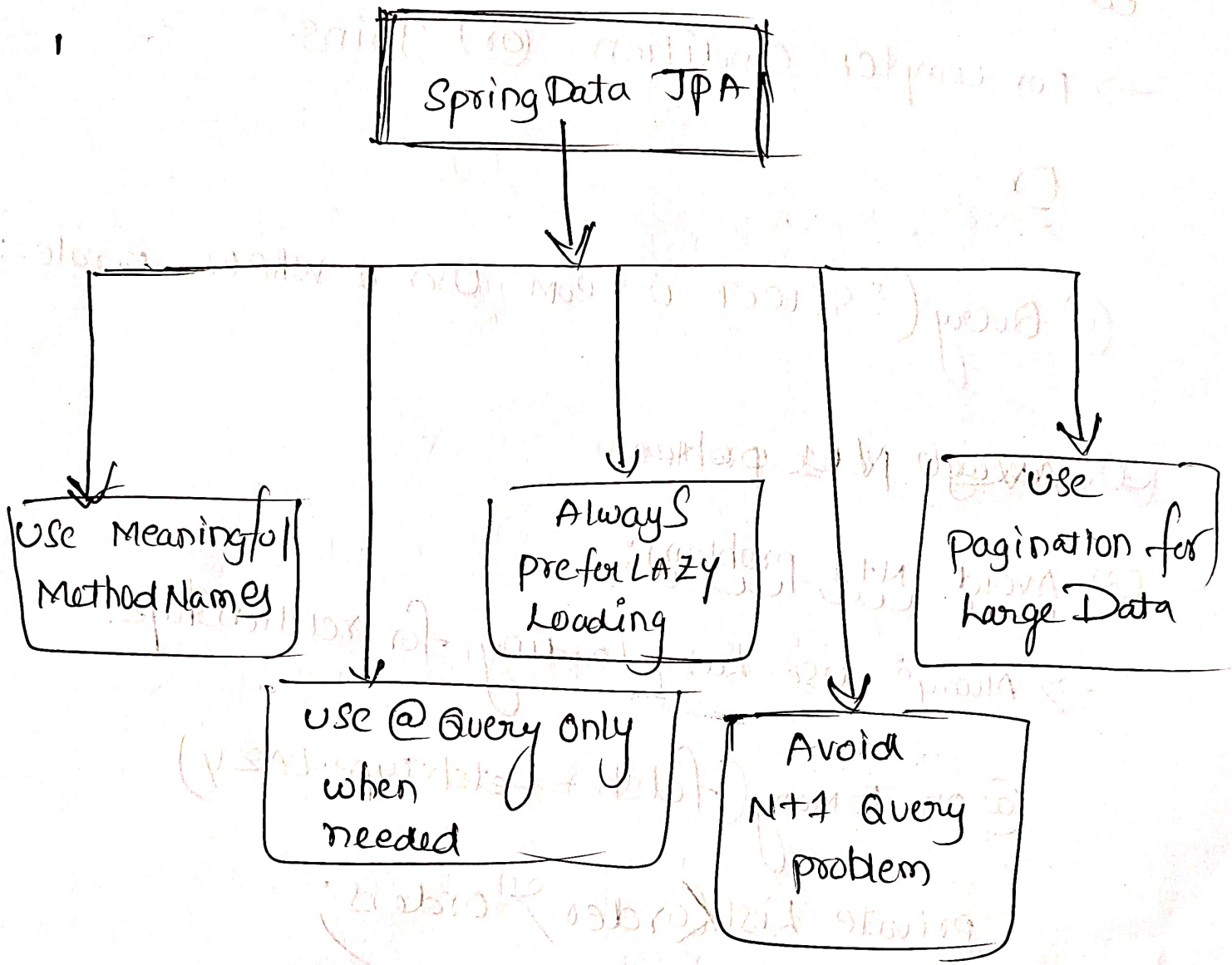
→ Spring Transaction layer is independent of DataBase.

Same code coorks with:

- → MySQL
- → oracle
- → H2

→ Spring Transaction = Safe, consistent & easy database operations

# * Spring Data JPA : (Best practices)




(1) Use Meaningful Method Names:

→ Let Spring create Queries automatically.

Ex

(1) findByEmail()

(2) findByNameAndStatus()

(2) Use @Query only when Needed:

→ For complex condition (or) Joins.

Ex

```
@Query("SELECT u FROM User u WHERE u.role=:role")
```

~~(3) Always N+1 problem~~

(3) Avoid N+1 problem:

→ Always use Lazy loading for relationships:

```
@OneToMany(fetch = FetchType.LAZY)
private List<Order> orders;
```

→ To overcome Lazy Loading issues, use JOIN-FETCH, EntityGraph.

(4) Use pagination for Large Data:

→ Never fetch thousands of rows at once.

Use

```
Page<User> findAll(Pageable pageable);
```

(5) Always prefer LAZY loading.

→ Use:

@OneToMany(fetch = FetchType.LAZY)

→ Reduces unwanted Queries.

Advantages

→ Spring Data JPA Best practices =

(1) Clean code

(2) Better Performance

(3) Fewer bugs

(4) Easy maintenance.

* Custom Repository Implementation:

→ Spring Data JPA gives us built in CRUD Methods, but Sometimes we need:

① Comple Queries

② Dynamic operations

③ custom bussines logic

→ In Such Cases, we create our own custom repository with custom methods.

Use a custom repository when:

→ @Query (or) method is not enough

→ Need to use Entity Manager directly

→ Need dynamic Queries

→ Need batch operations (or) manual transactions

Steps to create custom Repository

Create an Interface : (custom)

```
public interface StudentCustomRepository
{
List<Student> findByNameCustom(String name);
}
```

Add custom Interface to main Repository :

```
public interface StudentRepository extends
                    JpaRepository<Student, Long>,
                    StudentCustomRepository
{

}
```

Now StudentRepository Contains

→ Default JPA methods

→ Custom methods.

Example usage in Service:

```
@Service
public class StudentService
{
  @Autowired
  private StudentRepository studentRepository;

public List<Student> getStudentByName(String name)
    {
return StudentRepository.findByNameCustom(name);
    }
}
```

Advantages

→ Reusable custom logic

→ cleaner Code

→ Support complex Queries