1. Repository Pattern — Concept

The **Repository Pattern** is a **design pattern** that separates the **data access logic** (queries, persistence, etc.) from the **business logic** in your application.

Idea:

- Your service layer (business logic) shouldn't care *how* data is stored or retrieved.
- Instead, it interacts with a **Repository**, which provides an **abstraction layer** over the data source.

Analogy:

Think of a Repository as a **middleman** between:

```
Service Layer  <-->  Repository  <-->  Database
```

Benefits:

- Decouples data access logic from business logic.
- Easier to test (you can mock repositories).
- Simplifies maintenance and scaling.
- Improves readability and structure.

Example (Generic Repository Pattern in Java):

```
public interface Repository<T> {
    void save(T entity);
    void delete(T entity);
    T findById(Long id);
    List<T> findAll();
}
```

Then you could have a specific implementation for a `User` entity:

```
public class UserRepositoryImpl implements Repository<User> {
    // Imagine using JDBC or Hibernate directly here
    public void save(User user) { ... }
    public void delete(User user) { ... }
    public User findById(Long id) { ... }
    public List<User> findAll() { ... }
}
```

But writing all this manually for each entity is repetitive and error-prone.

---

2. Spring Data JPA — Simplifies the Repository Pattern

Spring Data JPA automates the Repository Pattern for you.
You don't need to write boilerplate code — Spring generates the implementation automatically at runtime.

Core Interfaces in Spring Data JPA

Spring Data defines several repository interfaces that you can extend:

Interface	Description
`Repository<T, ID>`	Base interface (marker) for all repositories
`CrudRepository<T, ID>`	Provides basic CRUD operations
`PagingAndSortingRepository<T, ID>`	Adds pagination and sorting
`JpaRepository<T, ID>`	Extends all above, adds JPA-specific methods

Example in a Spring Boot Project

Let's say you have a `User` entity:

```
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String username;
    private String email;
}
```

Now create a **Repository** interface:

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

@Repository
public interface UserRepository extends JpaRepository<User, Long> {

    // Spring Data JPA automatically provides implementations for:
    // save(), findById(), findAll(), deleteById(), etc.

    // You can define custom methods using query derivation
    Optional<User> findByUsername(String username);
}
```

Spring automatically generates the implementation at runtime — no need to write SQL or JPQL.

---

3. Query Derivation (Dynamic Finder Methods)

Spring Data JPA can **derive queries** from method names:

Method name	Generated SQL
`findByUsername(String username)`	`SELECT * FROM users WHERE username = ?`
`findByEmailContaining(String email)`	`SELECT * FROM users WHERE email LIKE %?%`
`findByAgeGreaterThan(int age)`	`SELECT * FROM users WHERE age > ?`

You can also write **JPQL** or **native queries** manually using annotations:

```
@Query("SELECT u FROM User u WHERE u.email = :email")
Optional<User> findByEmail(@Param("email") String email);
```

4. How It Fits in Spring Boot Architecture

```
Controller → Service → Repository → Database
```

Example:

```
@Service
public class UserService {
    private final UserRepository userRepository;
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public User createUser(User user) {
        return userRepository.save(user);
    }

    public Optional<User> getUserById(Long id) {
        return userRepository.findById(id);
    }
}
```

5. Summary Table

Aspect	Repository Pattern	Spring Data JPA Repository
Purpose	Abstract data access logic	Implements Repository pattern automatically
Implementation	Manual (custom interfaces + DAO classes)	Auto-generated by Spring
Query creation	Hand-written SQL or JPQL	Derived from method names or annotations
Flexibility	High but verbose	High with minimal code
Testability	High (mock interfaces)	High (mock JpaRepository)

Example in Your Digital Wallet Project

You might have repositories like:

```
public interface WalletRepository extends JpaRepository<Wallet,
Long> {
    Optional<Wallet> findByUserId(Long userId);
}

public interface TransactionRepository extends
JpaRepository<Transaction, Long> {
    List<Transaction> findByWalletId(Long walletId);
}
```

Spring will automatically create their implementations — you just inject and use them in your services.
