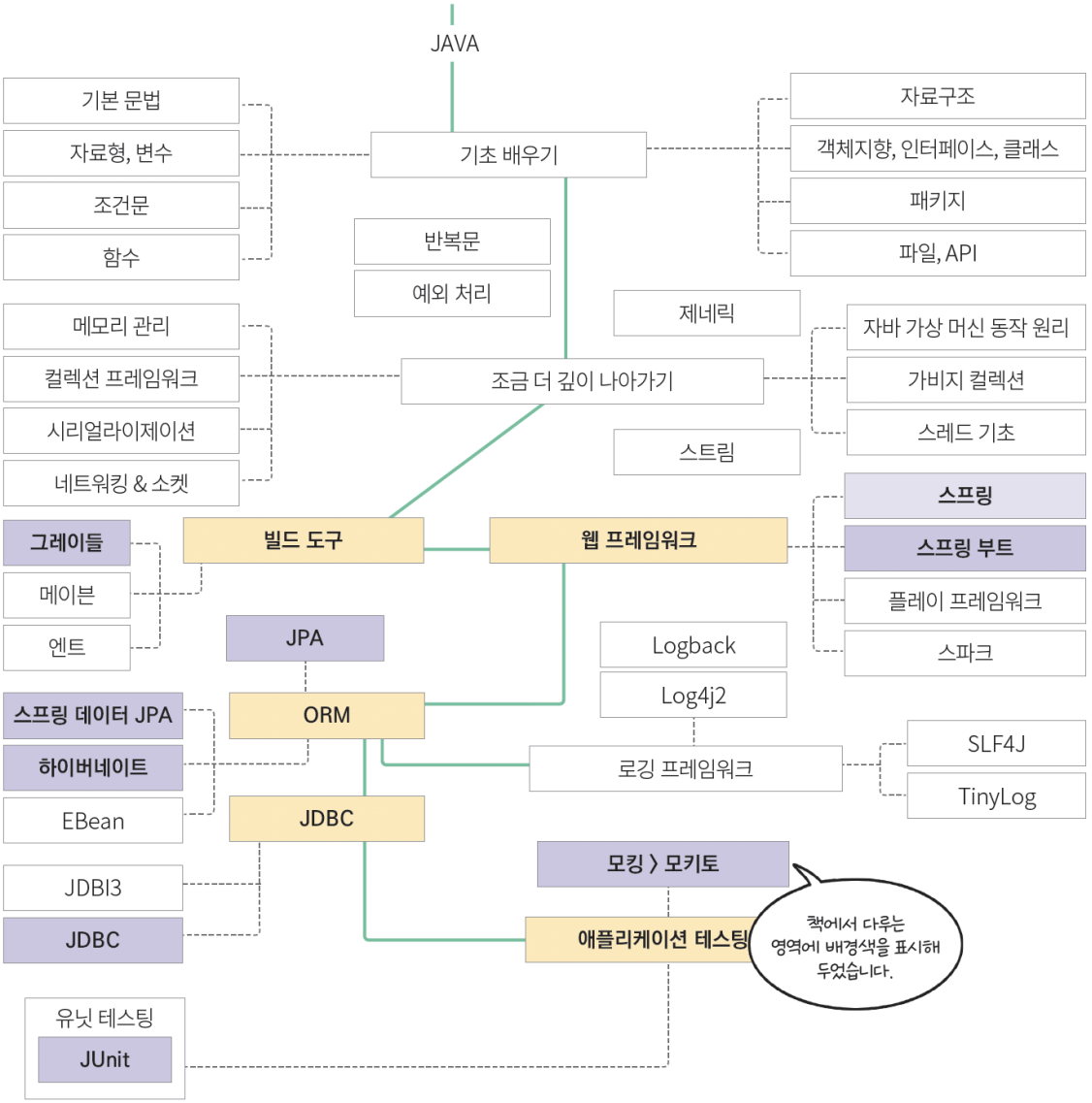
Springboot 노트

<백엔드 로드맵>



<스프링과 스프링부트>

- 스프링 : Java기반의 웹프레임워크. 웹애플리케이션 개발에서 구성요소들의 관리를 돕는다.
- 스프링부트 : 스프링을 더 쉽게 사용할 수 있도록 만들어진 웹프레임워크
  자동설정을 통해 복잡한 스프링의 설정방식을 쉽게 할 수 있도록 함.

내장서버(톰캣)을 제공하여 별도의 서버 설치없이도 빠르게 애플리케이션을 만들수 있음.

- 차이점: 스프링은 기능이 강력하나 설정이 복잡하고 개발난이도가 높음.
  스프링부트는 스프링의 기능을 단순화하여 빠르고 쉽게 개발할 수 있음.

<클라이언트, 서버, 데이터베이스의 역할>

1. 클라이언트

- GUI(Graphic User Interface)사용자 인터페이스를 제공하고, 사용자의 요청(Request)
  을 처리하여 서버로 전달함.
- 서버에서 받은 응답(Response)을 사용자에게 보여줌
- 예) 웹브라우저, 모바일 앱

2. 서버

- 클라이언트로부터 요청을 받아 처리하고 필요한 데이터를 DB에서 조회하거나 로직(Logic)수행
- 처리된 결과를 클라이언트로 응답
- 예) 웹서버, API 서버

3. 데이터베이스

- 데이터를 저장하고 관리하며, 서버의 요청에 따라 데이터를 검색하거나 수정함
- 예) MySQL, Oracle, MongoDB

** 시나리오 예시 (로그인) **

1. 클라이언트가 사용자에게 로그인화면 제공
2. 사용자가 로그인 정보를 입력
3. 클라이언트는 사용자정보를 암호화해서 서버로 전송
4. 서버는 클라이언트의 로그인요청을 받아서 Service모듈로 전달
5. Service모듈이 로그인 정보의 확인을 위해 데이터베이스에 관련 정보 요청
6. 양쪽 정보를 비교해서 성공하면 로그인 성공 응답을 클라이언트로 전송
7. 클라이언트는 로그인 성공을 사용자에게 알리고 접속화면으로 전환

<웹서버 프로젝트 개발 환경>

1. IDE : 인텔리이제이
2. 웹프레임워크 : 스프링부트3
3. 보안솔루션 : 스프링 시큐리티
4. 데이터베이스 : MySQL 8.4
5. ORM(Object-Relational Mapping) : 하이버네이트, JPA
6. 내장 WAS 사용 : 톰캣

<용어 정리>

1. IP와 Port

- 아이피 : 인터넷에서 컴퓨터관련 기기들이 서로를 식별하고 통신하기 위한 주소
- 포트 : 기기안에서 여러 서비스를 구분하기위한 번호
  예) 이메일서비스는 50번 포트, 카카오톡서비스는 51번 포트...

2. 라이브러리와 프레임워크

- 라이브러리 : 개발에 필요한 여러 유용한 클래스를 모아논 코드 모음
  예) java.sql.*, java.collection.list...
- 프레임워크 : 소프트웨어 개발을 수월하게 하기 위한 개발 환경(!)
  정해진 규칙과 틀안에서 개발을 수행하도록 정해논 환경.

3. 배포(Deployment)

- 개발된 애플리케이션이나 서비스를 사용 가능한 상태로 실제 환경에 릴리즈하는 과정
- 실제 또는 실제에 준하는 사용 환경에 설치하는 것

4. CI/CD

- CI(Continuous Integration) : 지속적 통합
  개발자들이 작성한 코드를 주기적으로 통합하고 자동으로 빌드 및 테스트하는 프로세스
- CD(Continuous Deployment) : 지속적인 배포
  CI가 성공적으로 완료된 후 자동으로 배포하는 프로세스
- 사용도구 : 깃허브, 젠킨스(Jenkins)

5. WS와 WAS

- WS (Web Server) : 기기 또는 시스템간 통신을 담당하며 데이터를 전달하는 역할
- WAS (Web Application Server) : 웹 애플리케이션을 실행하고 로직을 처리하는 서버.
  실제 비즈니스 로직을 처리할 수 있는 기능을 가지고 있음.
- 공통점 : 웹서비스의 요청과 응답을 처리
- 차이점 : WS가 통신시 데이터의 전달에 초점이 맞춰져있다면, WAS는 웹앱을 실행시켜 실제 비즈니스로직을 수행할 수 있는 기능 더 가지고 있는 것
- 유명한 WS : Nginx (엔진엑스)
- 유명한 WAS : Tomcat(톰캣), Jetty(제티)

6. Annotation(어노테이션) @

- 스프링부트에서 어노테이션은 매우 특별하고 많은 기능을 가짐.
- 개발자가 복잡한 설정을 직접 처리하지 않도록 도와줌

a. 애플리케이션 설정 및 초기화

b. 컴포넌트 등록 및 관리
   컴포넌트란, 자바 클래스를 말하는 용어

c. 의존성 주입
   필요한 컴포넌트 객체를 자동으로 찾아서 참조할 수 있도록 도와주는 것

d. 웹 요청 처리
   웹 요청을 처리할 메서드나 컨트롤러를 지정함

e. 데이터베이스 작업
   데이터베이스와 관련된 클래스나 메서드를 쉽게 관리해줌

f. 실행 흐름 관리
   특정작업이 시작하거나 종료될때 필요한 로직을 실행시켜 줌

** 요약하면,
클래스나 메서드위에 선언한 어노테이션의 종류에 따라서 "이건 이렇게 해줘"라는 개발자의 요청을 스프링부트가 알아서 처리할 수 있게 도와주는 역할을 수행
각 역할별로 다양한 어노테이션이 존재함

<스프링부트 프로젝트 기본 설명>

1. 웹페이지(start.spring.io)에서 설정하여 파일형태로 다운로드

- Project : Maven
- Language : Java
- 버전 : 3.4.0
- Metadata : Group(com.dw), Artifact(프로젝트명) Packaging(jar), Java(21)
- 의존성 : Web, JPA, MySQL

2. 폴더별 설명

- src/main/java/com/dw/프로젝트명 : 코드영역. java파일들이 위치함
- src/main/resources/application.properties : 환경설정파일
- pom.xml : 의존성 정의, 빌드환경 세팅

3. application.properties

- 데이터베이스 연결 정보를 기록
  spring.datasource.url=jdbc:mysql://localhost:3306/testdb
  spring.datasource.username=root
  spring.datasource.password=root
- 이외 각종 스프링관련 세팅을 설정하는 파일임

4. pom.xml

- Maven에서 사용되는 프로젝트 객체 모델 파일로서 프로젝트의 구성정보와 의존성을 정의
- 여러 프로젝트 정보를 정의하고 외부 라이브러리 의존성을 관리
- 빌드 설정을 정의

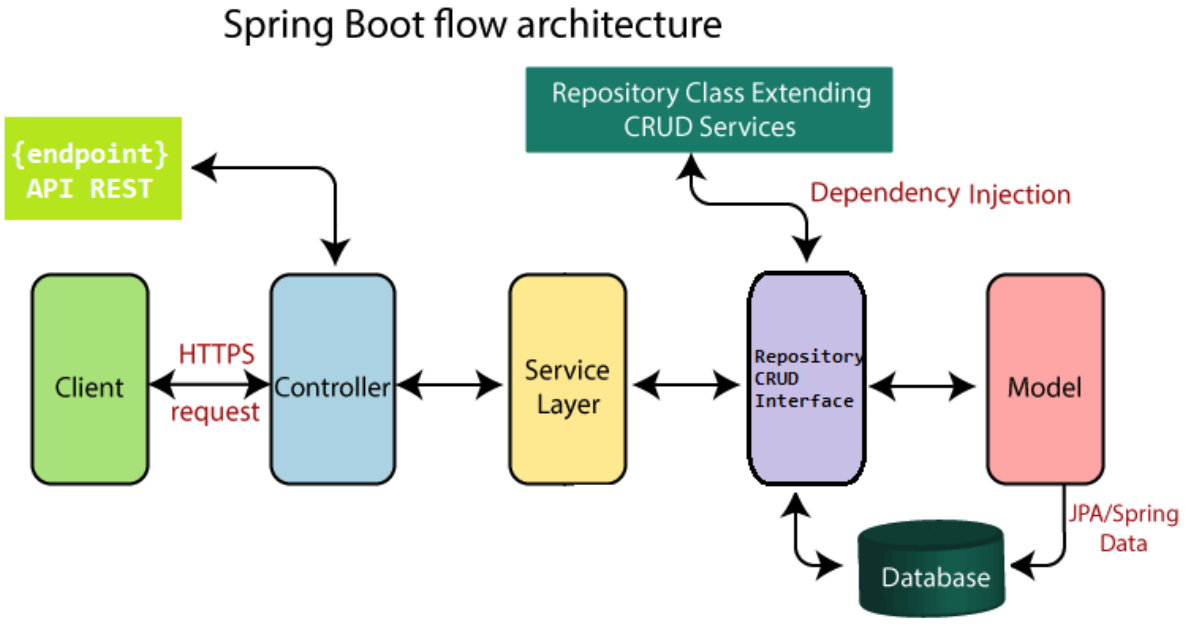
<스프링부트의 계층형 구조 형태>

: 아래와 같이 계층형 구조를 나누는 것은 필수는 아니지만 권장됨

1. Model (또는 Entity)
   데이터베이스 테이블과 매핑되는 Entity클래스가 위치함
2. Controller

- 클라이언트의 요청을 받고 적당한 서비스를 호출하여 결과를 반환하는 역할
- 주로 REST API의 엔드포인트를 정의하는 곳

3. Service
   비즈니스 로직을 처리하는 서비스 클래스가 위치함
4. Repository (또는 DAO)
   데이터베이스와 상호작용하는 클래스가 위치함. DB의 CRUD 작업을 처리.
5. DTO
   Data Transfer object. 전달할 목적으로 정의된 데이터 클래스가 위치함
6. Exception
   예외처리 클래스가 위치함



<스프링부트의 컨셉>

1. Bean

- Spring Container 또는 Application Context에 의해 생성, 관리되는 객체를 의미함.

- 이 객체들은 스프링 컨테이너에 의해 생성되고 필요한 곳에 주입되어 사용됨 (new를 이용하여 인스턴스화 할 필요 없음)
- @Component : Bean으로 등록할 클래스를 표시. 많은 어노테이션들의 상위어노테이션이기 때문에 빈으로 만들기 위해 반드시 @Component를 사용할 필요는 없음

  @Service, @Controller, @Repository들은 모두 @Component의 하위 어노테이션들임
- @ComponentScan : 모든 @Component가 붙은 클래스를 찾아 자동으로 빈으로 등록하는 역할

2. IoC (=Inversion of Control) 제어의 역전

- 특정 클래스가 내부에 외부 클래스를 명시적으로 사용할 경우, 의존성 관계가 발생함.
- 그때 객체의 제어(=생성)을 그 객체를 사용하는 클래스가 가지지 않고 스프링 컨테이너가 가지는 개념
- 자바 : 객체생성의 주체는 클래스
- 스프링/스프링부트 : 객체생성의 주체는 스프링 컨테이너

3. DI (=Dependency Injection) 의존성 주입

객체가 직접 필요로 하는 의존 객체를 직접 생성(=new)하지 않고, 외부에서 의존 객체를 주입받는 것(=@Autowired)을 말함.

@Autowired를 선언하는 위치가 3곳

- 필드주입 : 필드에 직접 선언하는 방식. 간편하나 테스트시 불리함
- 생성자주입 : 생성자에 선언하는 방식. (권장)
- 세터주입 : setter 메서드에 선언하는 방식.

4. AOP (=Aspect Oriented Programming) 관점 지향 프로그래밍

프로그램을 설계할 때 각 기능을 모듈화하고 관점에 따라 나누어 프로그래밍하는 프로그래밍 패러다임임. OOP(Object-Oriented Programming)를 보완하여 코드의 재사용성과 모듈성을 향상시키는 데 도움을 줄 수 있음.

특히 기능을 핵심관점, 부가관점, 공통관점등으로 나누고 부가, 공통 관심 사항을 애플리케이션의 핵심 비즈니스 로직과 분리함. 예를 들어, 로깅, 트랜잭션 관리, 보안, 성능 측정 등의

여러 곳에서 반복적으로 사용되는 코드들을 AOP를 사용하여 한 곳에서 관리할 수 있도록 함.

5. PSA (=Portable Service Abstraction) 이식가능한 서비스 추상화

다양한 환경에서 동일한 코드를 실행할 수 있도록 해주는 기능. 이식 가능한 추상화는 특정 환경에 종속되지 않고 동일한 인터페이스를 통해 다양한 환경에서 작동할 수 있도록 설계됨. 이를 통해 개발자는 코드를 작성할 때 특정 환경에 의존하지 않고 환경에 관계없이 일관된 방식으로 작업할 수 있음.

예를 들어, 스프링 프레임워크를 사용하여 JPA를 구성하고 애플리케이션을 개발할 때, 애플리케이션 코드는 JPA의 Entity 클래스와 Repository 인터페이스를 사용하여 데이터베이스와 상호 작용함. 이때 개발자는 실제 데이터베이스에 대한 세부 사항(예: SQL 문법, 데이터베이스 제품의 특정 기능)을 신경 쓰지 않고도 애플리케이션을 개발할 수 있음. 만약 데이터베이스를 변경해야 하는 경우, 코드를 수정하지 않고도 JPA 설정만 변경하여 새로운 데이터베이스에 대해 동일한 애플리케이션을 실행할 수 있음.

<Rest API (=RESTful API)>

: Representational State Transfer Application Programming Interface
: 클라이언트와 서버간의 통신을 설계하는데 사용하는 소프트웨어
아키텍처 스타일
: 주로 HTTP를 기반으로하며, 웹서비스에서 데이터를 요청하고 응답하기 위해 널리 사용됨
: REST의 주요 개념

1. 리소스(자원) - 모든 데이터는 URL을 통해 고유의 자원으로 표현됨
   예)
   www.mypage.com/user/1234
2. HTTP 메서드를 사용

- GET(조회), POST(생성), PUT(수정), DELETE(삭제)

3. Stateless (무상태성)

- 각 요청은 독립적으로 처리되며, 서버는 클라이언트의 이전 요청 상태를 저장하지 않음. 그러므로 클라이언트의 모든 요청은 매번 전부 포함되어야 한다.

4. 표현(응답)

- 자원은 여러 형식으로 반환하나 주로 JSON 과 XML이 사용됨

* GET메서드에서 매개변수를 사용하는 방법

(1) Query Parameters (쿼리 문자열)
예)
http://localhost:8080/employee?id=E01

```
// Query Parameters (쿼리 문자열)
@GetMapping("/employee")
public Employee getEmployeeById(@RequestParam String id) {
    return employeeService.getEmployeeById(id);
}
```

(2) Path Parameters (경로 매개변수)
예)
http://localhost:8080/employee/E01

```
// Path Parameters (경로 매개변수)
@GetMapping("/employee/{id}")
public Employee getEmployeeById_2(@PathVariable String id) {
    return employeeService.getEmployeeById(id);
}
```

* POST/PUT 메서드에서 데이터를 전달하는 방법

POST/PUT 메서드는 주로 전송하는 데이터의 크기가 크거나 객체 형태를 가지는 경우가 많기때문에 HTTP 요청 본문에 데이터를 포함하여 전달함. 주로 JSON(JavaScript Object Notation)형태로 전송

⇒ 브라우저의 주소창을 통해서는 불가능하며 Javascript를 사용하거나 “포스트맨”등의 어플을 이용함

```
POST /api/resource
Content-Type: application/json

{
  "name": "John",
  "age": 30,
  "address": {
```

```
    "city": "Daejeon",
    "country": "South Korea"
  }
}
```

- **장점**:
  - 구조화된 데이터 전달에 최적.
  - 대량의 데이터 및 중첩된 객체를 포함한 복잡한 구조 지원.
  - URL에 데이터가 노출되지 않아 보안에 유리.
- **단점**:
  - 데이터 처리가 추가적으로 필요할 수 있음 (e.g., JSON 파싱).
  - Query Parameter처럼 URL에서 바로 확인하기 어렵기 때문에 디버깅이 다소 번거로울 수 있음.

⇒ Query Parameter와 Path Variable은 간단한 데이터 전달에는 편리하지만, 보안 및 복잡한 데이터 구조를 다룰 때는 부적합합니다. 민감한 데이터를 다루거나 구조화된 데이터를 전달할 경우 **Body**를 사용하는 것이 안전하고 효율적입니다.

<HTTP (HyperText Transfer Protocol)>

웹브라우저와 서버간의 데이터를 주고받기 위한 인터넷 프로토콜 (프로토콜(Protocol)이란 규약, 약속의 의미임). 웹에서 컨텐츠(텍스트, 이미지, 사운드, 비디오등)를 전송하기 위해 가장 널리 사용되는 프로토콜이며 기본적으로 요청-응답으로 이루어져 있음

1. 클라이언트-서버 구조이며 클라이언트는 요청하고 서버는 응답을 담당
2. HTTP는 기본적으로 상태를 저장하지 않음 (Stateless)
3. 모든 메시지는 헤더와 본문으로 구성
4. 요청은 아래와 같은 메서드로 구분되어 있음
   : GET, POST, PUT, DELETE, HEAD, OPTIONS, PATCH
5. 상태코드 - 성공,실패를 원인과 함께 미리 정의해논 코드로 표시
   : 예) 200 OK
   201 Created
   400 Bad Request
   404 Not Found
   500 Internal Server Error

6. HTTPS

- HTTP : 데이터를 일반 텍스트로 전송. 보안에 취약. 중간에 가로채기 하거나 데이터 유출이 가능
- HTTPS : HTTP에 SSL/TLS 암호화를 추가한 프로토콜. 데이터를 암호화하기때문에 안전한 통신을 보장함

<JDBC Template>

Spring Framework에서 제공하는 데이터베이스 연동을 간편하게 처리하기 위한 유틸리티 클래스.

데이터베이스 작업 시 반복되는 코드를 줄이고, 효율적으로 SQL 쿼리를 실행할 수 있도록 설계됨

JdbcTemplate의 주요 특징

- **단순화된 데이터베이스 작업**:
  - JDBC를 사용할 때 발생하는 Connection, Statement, ResultSet 관리 및 예외 처리를 자동화.
- **SQL 중심의 코드 작성**:
  - SQL 쿼리를 문자열로 작성하며, ORM (예: JPA, Hibernate) 없이도 데이터베이스 작업을 수행 가능.
- **예외 변환**:
  - Spring의 DataAccessException 계층으로 JDBC 예외를 추상화하여 처리.
- **효율적인 작업 지원**:
  - PreparedStatement를 사용한 파라미터 바인딩 지원.
  - RowMapper를 사용하여 조회한 데이터를 객체로 변환하는 과정을 효율적으로 처리할 수 있음. 이를 통해 코드 중복을 줄이고, 객체 매핑이 필요한 대부분의 상황에서 가독성과 유지보수성을 높일 수 있음.

RowMapper란?

`RowMapper`는 인터페이스로, 데이터베이스에서 조회된 각 행(`ResultSet`)을 도메인 객체로 매핑하는 데 사용됨.

1) 익명 클래스 사용

```
private final RowMapper<Department> departmentRowMapper = n
ew RowMapper<Department>() {
        @Override
        public Department mapRow(ResultSet rs, int rowNum)
throws SQLException {
            Department department = new Department();
            department.setDepartmentId(rs.getString("부서번
호"));
            department.setDepartmentName(rs.getString("부서
명"));
            return department;
        }
    };
```

2) 람다 표현식 사용

```
private final RowMapper<MileGrade> mileGradeRowMapper = (r
s, rowNum) -> {
        MileGrade mileGrade = new MileGrade();
        mileGrade.setGrade(rs.getString("등급명"));
        mileGrade.setLowerMileage(rs.getInt("하한마일리지"));
        mileGrade.setUpperMileage(rs.getInt("상한마일리지"));
        return mileGrade;
    };
```

JdbcTemplate 주요 메서드

JdbcTemplate은 다양한 메서드를 제공하여 데이터 조회, 삽입, 수정, 삭제를 간단히 처리할 수 있습니다.

1) 데이터 조회 (SELECT)

- `query`: 다중 행 결과를 처리. List<T> 반환.
- `queryForObject`: 단일 행 결과를 반환. 단일 클래스를 반환.
- 예:

```
public List<Customer> getAllCustomers() {
        String query = "select * from 고객";
        return jdbcTemplate.query(query, customerRowMapp
er);
}

public Product getProductById(int productNumber) {
        String query = "select * from 제품 where 제품번호
=?";
        return jdbcTemplate.queryForObject(query, produc
tRowMapper,productNumber);
}
```

2) 데이터 삽입, 수정, 삭제

- `update`: INSERT, UPDATE, DELETE와 같은 변경 쿼리를 실행.
- 예:

```
public Product saveProduct(Product product) {
        String query = "insert into 제품(제품번호,제품명,포장
단위,단가,재고)" +
                " values(?,?,?,?,?)";
        jdbcTemplate.update(query,
                product.getProductId(),
                product.getProductName(),
                product.getPackageUnit(),
                product.getUnitPrice(),
                product.getStock());
        return product;
}
```

JdbcTemplate 사용 시 장단점

장점:

- JDBC 코드에서 반복적인 자원 관리 (Connection, Statement 등) 작업을 제거하여 코드 간소화.

- SQL 중심의 데이터베이스 접근 방식을 선호하는 경우 적합.
- Spring의 예외 추상화를 통해 다양한 데이터베이스에 대한 이식성을 높임.
- 트랜잭션 관리와 통합 가능 (Spring Transaction).

단점:

- 직접 SQL을 작성해야 하므로, 복잡한 쿼리 작업에서는 코드가 길어질 수 있음.
- 객체-관계 매핑(ORM) 지원이 부족해 데이터베이스와 객체 간 변환을 수동으로 처리해야 함.

<스프링부트 의존성주입을 통한 Repository 의 확장>

스프링 부트의 의존성 주입(Dependency Injection)을 활용하면 애플리케이션에서 **Repository 계층의 확장성**을 쉽게 구현할 수 있음. 이를 통해 동일한 인터페이스를 기반으로 **다양한 구현체**를 선택적으로 사용할 수 있게 되어 유지보수성과 테스트 용이성이 크게 향상됨.

Repository 확장 구현의 기본 구조

스프링 부트에서 의존성 주입을 활용하면 인터페이스와 구현체를 분리하여 확장 가능한 구조를 설계할 수 있음. 이를 통해 유지보수성과 확장성을 극대화.

1) 공통 인터페이스 정의

공통된 동작을 명세하는 인터페이스를 먼저 정의함

```
public interface CustomerRepository {
    List<Customer> getAllCustomers();
}
```

2) 구현체 정의

```
@Repository
public class CustomerJdbcRepository implements CustomerRepo
sitory {
```

```
@Repository
public class CustomerTemplateRepository implements Customer
```

```
Repository {
```

3) Repository 구현체 주입을 통한 확장성

```
@Service
public class CustomerService {
    @Autowired
    @Qualifier("customerTemplateRepository") //@Qualifier
("customerJdbcRepository")
    CustomerRepository customerRepository;
```

장점

- **유연한 구조**:
  - 동일한 인터페이스를 구현한 여러 Repository 클래스 중 하나를 선택적으로 주입 가능.
  - 새로운 데이터 접근 방식이 필요할 경우, 기존 코드를 수정하지 않고도 새로운 구현체를 추가할 수 있음.
- **테스트 용이성**:
  - 테스트 환경에서 **가짜(Mock) Repository**를 구현하여 주입 가능.
- **환경별 구성을 통한 유지보수성**:
  - 프로덕션 환경에서는 `JdbcTemplateUserRepository`, 개발 환경에서는 `JdbcUserRepository` 등을 사용 가능.
  - 환경별 분리로 코드 수정 없이 다른 구현체로 전환 가능.

< 예외 처리 >

자바의 예외 처리는 프로그램 실행 중 발생하는 오류를 처리하여 프로그램이 갑작스럽게 종료되지 않고 안정적으로 실행되도록 돕는 기법임. 이를 통해 예외 상황을 관리하고 프로그램의 품질을 높일 수 있음.

예외 처리 방식의 특징

자바에서는 예외 처리를 두 가지 방식으로 처리:

- **Checked Exception**: 컴파일 시점에 반드시 처리해야 하는 예외. `IOException`, `SQLException` 등이 이에 해당하며, 명시적으로 예외 처리를 하지 않으면 컴파일 오류가 발생.
- **Unchecked Exception**: 실행 시점에 발생하는 예외로, `RuntimeException`을 상속받은 예외. `NullPointerException`, `ArrayIndexOutOfBoundsException` 등이 이에 해당함. 컴파일러가 처리 강제를 하지 않음.

예외처리의 두가지 방법

1) `try-catch` 문

- 예외를 직접 처리할 때 사용.
- 예외가 발생할 가능성이 있는 코드를 `try` 블록 안에 작성하고, 발생한 예외를 `catch` 블록에서 처리.
- 프로그램의 흐름을 예외 상황에서도 유지할 수 있음

```
public class TryCatchExample {
    public static void main(String[] args) {
        try {
            int result = 10 / 0; // 예외 발생
        } catch (ArithmeticException e) {
            System.out.println("예외 처리: 0으로 나눌 수 없습니다."
        }
        System.out.println("프로그램 계속 실행");
    }
}
```

2) `throws` 문

- 메서드 선언부에서 사용하여, 해당 메서드가 특정 예외를 던질 수 있음을 명시함.
- 예외를 직접 처리하지 않고 호출한 메서드로 예외를 위임.
- “예외전가” 라고 부르며 모든 메서드가 예외를 전가(`throws`)만 하고 직접 처리(`try-catch`)하지 않는 경우, 예외는 계속 상위 호출 메서드로 전달됨. 결국, 예외가 `main()` 메서드까지 전달되고, `main()`에서도 예외를 처리하지 않으면 JVM(Java Virtual Machine)이 예외를 처리하게 됨.
- 즉, 예외를 직접 처리하지 않게되므로 예외처리를 수행하는 의미가 없음.

```
public class ThrowsExample {
    public static void main(String[] args) {
        try {
            readFile();
        } catch (FileNotFoundException e) {
            System.out.println("파일을 찾을 수 없습니다.");
        }
    }

    public static void readFile() throws FileNotFoundExcept
ion {
        Scanner scanner = new Scanner(new File("nonexistent
file.txt"));
    }
}
```

예외 전가의 설계적 관점

1) 예외 전가를 통해 코드 중복 최소화

- 각 메서드에서 별도로 예외를 처리하면 코드 중복이 발생할 가능성이 높아짐.
- 예외 전가는 예외를 호출한 상위 메서드나 전역 핸들러로 전달함으로써, **하나의 일관된 예외 처리 로직**을 구현할 수 있도록 도움.

2) 책임의 분리

- 예외 전가는 비즈니스 로직과 예외 처리를 분리.
- 메서드 자체는 비즈니스 로직에만 집중하고, 예외 처리는 상위 계층 또는 전역 계층에서 책임질 수 있음.

3) 예외 처리의 중앙 집중화

- 예외 전가를 통해 최종적으로 **전역 핸들러(@RestControllerAdvice)**와 같은 중앙 집중형 예외 처리 구조를 구축할 수 있음.
- 이를 통해 예외 응답 형식(예: JSON), HTTP 상태 코드, 로깅 등을 통합적으로 관리할 수 있음.

< 스프링 부트에서 예외를 처리하는 방법 >

1) 전통적인 `try-catch`를 사용한 예외 처리

- 각 컨트롤러나 서비스 레벨에서 `try-catch`를 사용하여 직접 예외를 처리.
- 간단한 예외 처리에는 적합하지만, 코드 중복과 가독성 저하 문제가 발생할 수 있음.

2) `@ExceptionHandler`를 사용한 예외 처리

- 특정 컨트롤러 클래스에서 발생하는 예외를 처리.
- 컨트롤러 내에서 메서드에 `@ExceptionHandler`를 선언하여 예외를 처리

3) `@RestControllerAdvice`를 사용한 전역 예외 처리

- 애플리케이션 전역에서 예외를 처리합니다.
- 모든 `@RestController`에서 발생하는 예외를 일괄적으로 처리할 수 있어, 중복 코드를 줄이고 일관된 응답 구조를 유지할 수 있음.
- <중요> 서비스와 리포지토리에서 발생하는 예외는 모두 컨트롤러에서 처리되므로 일반적인 스프링부트의 프로세스에서는 `@RestControllerAdvice`를 사용한 전역에서 처리가 가능함. 하지만 좀 더 복잡한 구조를 가지게 되는 경우, 예를 들어 테스트와 같은 독립적인 모듈이 포함된 경우에는 다른 방식의 전역 예외처리가 필요할 수 있음.
- 독립적인 모듈의 예:

  테스트 모듈: 독립적으로 실행되는 단위 테스트, 통합 테스트, E2E 테스트.

  비동기 작업: 이벤트 큐 또는 스케줄러를 통해 비동기로 실행되는 작업.

  서브 모듈: 다른 팀이나 서브 프로젝트에서 개발된 독립적인 기능 모듈.

  서드파티 통합: 외부 API와 통신하는 서브시스템.

`@RestControllerAdvice`의 역할

`@RestControllerAdvice`는 컨트롤러 전역에서 예외를 처리하기 위한 어노테이션.

`@ControllerAdvice`와 달리 REST 컨트롤러에 특화되어 JSON 응답을 기본으로 처리함.

- `@RestControllerAdvice`는 `@ControllerAdvice`와 `@ResponseBody`가 결합된 형태임.
- 애플리케이션의 전역적인 예외 처리 핸들러로 동작함.

```
@Order(Ordered.HIGHEST_PRECEDENCE)
@RestControllerAdvice
public class CustomExceptionHandler {
```

```
    @ExceptionHandler(InvalidRequestException.class)
    public ResponseEntity<String> handleInvalidRequestExcep
tion(
            InvalidRequestException e) {
        return new ResponseEntity<>(
                e.getMessage(), HttpStatus.BAD_REQUEST);
    }

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<String> handleResourceNotFoundExc
eption(
            ResourceNotFoundException e) {
        return new ResponseEntity<>(
                e.getMessage(), HttpStatus.NOT_FOUND);
    }
}
```

`@RestControllerAdvice`의 주요 장점

1) **전역 예외 처리**

- 애플리케이션 전반에서 발생하는 예외를 중앙에서 관리할 수 있음.
- 특정 컨트롤러에 종속되지 않으므로 코드 중복을 줄일 수 있음.

2) **일관된 API 응답**

- <중요> 모든 예외 처리에 대해 일관된 JSON 응답을 반환할 수 있음!!!
- 클라이언트와 서버 간의 규격을 유지할 수 있음.

3) **모듈화된 예외 처리**

- 특정 패키지나 컨트롤러를 대상으로 예외 처리 핸들러를 설정할 수 있음.