스프링(김영한)

Lee Jun Hyeok (wnsx0000@gmail.com)

September 5, 2024

목차

Part I

서론

1. 서론

1.1. 서론

1.1.1. 프로젝트 생성

기존에 스프링만으로 프로젝트를 생성하려면 복잡한 환경설정을 수행해야 했지만, Spring Initializr (start.spring.io)를 사용하면 GUI로 편리하게 프로젝트를 생성하여 다운받을 수 있음.

프로젝트명, 의존성(라이브러리), java 버전 등을 지정하여 프로젝트를 생성함. 이때 java 버전을 잘 맞춰줘야 에러가 발생하지 않음. 웹 개발을 할 것이라면 Spring Web과 Thymeleaf(템플릿 엔진)를 의존성에 추가함.

패키징 방식에는 jar와 war가 있음. jar는 내장 서버(톰캣 등) 사용에 최적화 된 방식이고, war는 기능이 더 추가되어 주로 배포에 최적화 된 방식임. war도 내장 서버를 사용할 수는 있지만 주로 서버가 설치되어 있는 외부 환경에 배포할 때 사용하고, 최근 스프링 부트에서는 주로 jar를 사용함. 추가로, jsp를 사용하려면 war를 선택해야 함.

다운받은 프로젝트 파일을 인텔리제이 등에서 열어 사용할 수 있음.

1.1.2. 라이브러리 개요

지정한 라이브러리들 외에도 기본적으로 설치되는 것들과 의존성에 의해 함께 설치되는 것들이 있음. 예를 들어, 톰캣(서블릿 컨테이너, 웹 서버), SLF4J(로깅), JUnit(테스트) 등이 설치됨.

개괄적으로, 웹 개발을 위해 Spring Web, Thymeleaf를 지정했다면 아래와 같은 라이브러리들이 설치됨.

스프링 부트 라이브러리.
-> spring-boot-starter-web. 톰캣, MVC 등 포함.
-> spring-boot-starter-thymeleaf.
-> spring-boot-starter. 스프링 부트, 스프링 코어, 로깅 등 포함.

테스트 라이브러리.
-> spring-boot-starter-test. 테스트 관련 라이브러리들.

1.1.3. 빌드

프로젝트를 빌드하여 배포할 수 있음.

콘솔에 ./gradlew build를 입력하면 build 디렉토리가 생성되고, 내부에 libs 디렉토리로 들어가면 hello-spring-0.0.1-SNAPSHOT.jar 등의 파일이 있음. 이는 자바 클래스/리소스 파일들을 묶은 압축 파일로, java -jar hello-spring-0.0.1-SNAPSHOT.jar로 실행할 수 있음.

빌드 툴에는 gradle, maven 등이 있지만, 최근에는 대체로 gradle을 사용한다고 함. gradle은 JVM 기반의 빌드 자동화 도구로, 스프링 프로젝트를 생성하면 존재하는 build.gradle 파일은 gradle에 대한 옵션을 지정할 수 있는 스크립트 파일임. 이 파일에서 빌드, 의존성, 리포지토리 등을 지정할 수 있음.

1.1.4. 프로젝트 폴더 구성

스프링 부트의 프로젝트 폴더는 기존의 자바 프로젝트 폴더에서 몇 가지 폴더와 파일이 추가된 구성을 가짐. 특히 gradle 관련 파일과 폴더가 여러 가지 추가되어 있음. src에는 main 폴더와 test 폴더가 있는데, test에는 테스트용 코드와 리소스를 작성함.

main에는 java 폴더와 resources 폴더가 있음. java 폴더에는 java 패키지와 클래스가 위치함. resources에는 웹 리소스가 위치하는데, static 폴더에는 정적 컨텐츠가 존재하고 templates 폴더에는 템플릿 관련 리소스가 존재함.

1.1.5. SpringApplication.run()

스프링 부트 프로젝트를 생성하면 TestSpringApplication 등의 식별자를 가진 클래스가 자동 생성되고, 이 클래스는 SpringApplication.run()를 호출하는 main 메소드를 가지고 있음. 이 SpringApplication.run()를 호출하면 웹 서버가 실행됨.

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

1.1.6. localhost:8080

localhost:8080을 주소로 하여 웹브라우저에서 접속하면 로컬 컴퓨터의 8080 포트로 접속할 수 있음. 이때 당연하게도 웹 서버가 돌아가고 있는 상태여야 함.

8080 포트는 대체로 웹 어플리케이션 개발에서 테스트 용도로 사용하는 포트임. 즉, 로컬 컴퓨터에서 스프링으로 웹 서버를 가동 중이라면 localhost:8080으로 접속하여 테스트할 수 있음.

1.1.7. 표준 문서 확인

스프링, thymeleaf 등 프레임워크나 툴 등은 홈페이지에 가면 공식 문서를 확인할 수 있는데, 문법의 모든 요소들을 외울 수는 없으므로 궁금한 게 생기면 여기서 찾아보면 됨.

1.2. 추가 java 문법들

1.2.1. Optional<T> 클래스

1. Optional<T>
Optional<T> 클래스는 어떤 값이 존재할 수도 있고, 존재하지 않을 수도 있는 상황에 사용하는 컨테이너(wrapper) 클래스임. (Java8에 추가됨.)

값이 존재하지 않는 경우 null을 반환하는 식으로 처리할 수도 있지만, Optional<T> 클래스를 사용하면 더 명시적이고 안전한 처리가 가능함. 또한 Optional<T>의 메소드들을 유용하게 사용할 수 있음. 즉, Optional<T>는 반환 시에 null의 사용(NullPointerException 발생 가능)을 대체하는 문법으로서 등장한 것임.

2. 유의사항
Optional<T>는 반환 타입에 사용되도록 제한적으로 설계되어 있기 때문에 매개변수 등 다른 용도로 사용하면 오류가 발생할 수 있음. 또한 직렬화를 지원하지 않으므로, 필드의 자료형으로 사용하는 것은 지양해야 함.

Optinal<T>를 사용해도 해당 Optinal<T> 객체에 값이 존재하지 않는 경우를 처리해야 하기 때문에, 특정 메소드를 사용해야 함. 이 경우를 처리하지 않는다면 NullPointerException 예외는 피했지만 NoSuchElementException 예외가 발생할 수 있음.

3. 주요 메소드들
Optinal<T>의 주요 메소드들은 아래와 같음.

static <T> Optional<T> empty() : 빈 Optional<T> 객체 반환.
static <T> Optional<T> of(T value) : null이 아닌 값으로 Optinal<T> 객체 생성. null이면 예외 발생.
static <T> Optional<T> ofNullable(T value) : Optinal<T> 객체 생성. null이면 빈 객체가 생성됨.

T get() : 해당 객체에 값이 존재하면 반환하고, 존재하지 않으면 예외 발생.

void ifPresent(Consumer<? super T> consumer) : 해당 객체에 값이 존재하면 작업을 수행하고, 존재하지 않으면 수행하지 않음.

boolean isPresent() : 해당 객체에 값이 존재하는지를 반환.

T orElse(T other) : 해당 객체에 값이 존재하면 값을 반환하고, 존재하지 않으면 other를 반환.
T orElseGet(Supplier<? extends T> other) : 해당 객체에 값이 존재하면 값을 반환하고, 존재하지 않으면 other를 호출하고 그 결과를 반환함.

Optional<T> filter(Predicate<? super T> predicate) : 참이면 해당 객체와 값이 같은 객체를 반환하고, 거짓이면 빈 객체를 반환함.

1.2.2. 함수형 인터페이스

함수형 인터페이스(functional interface)는 오직 1개의 추상 메소드를 가지는 인터페이스로, 함수형 프로그래밍을 위해 java8에 도입된 문법임. 주로 람다 표현식과 함께 사용됨.

오버라이드 시에 @Override 애너테이션을 작성하는 것과 동일하게, 함수형 인터페이스 정의 시에는 @FunctionalInterface 애너테이션을 작성하는 것이 좋음.

자바 표준 라이브러리에는 아래와 같은 함수형 인터페이스들이 포함되어 있음.
Consumer<T> : 하나의 매개변수(T)를 받고, 반환값이 존재하지 않음.
Supplier<T> : 매개변수를 받지 않고, 반환값(T)이 존재함.
Function<T, R> : 하나의 매개변수(T)를 받고, 반환값(R)이 존재함.
Predicate<T> : 하나의 매개변수(T)를 받고, boolean형 데이터를 반환함.

1.2.3. 람다 표현식

람다 표현식(lambda expression)은 익명 함수를 간결하게 표현할 수 있도록 한 문법으로, 함수형 프로그래밍 등을 위해 java8에 도입된 문법임.

람다 표현식은 대체로 함수형 인터페이스 변수에 대입하여 사용함. 사용 시에는 해당 함수형 인터페이스에 존재하는 메소드들로 매개변수를 전달하고 반환값을 얻을 수 있음.

람타 표현식의 매개변수에는 자료형을 작성할 수도 있지만, 함수형 인터페이스에 타입이 명시되어 있다면 생략할 수도 있음.

람다 표현식은 아래의 형태를 가짐.

```
(parameters) -> expression
```

아래는 그 예시임.

```
public class LambdaExample {
    public static void main(String[] args) {
        // Function 함수형 인터페이스를 사용한 람다 표현식
        Function<Integer, Integer> square = x -> x * x;

        System.out.println(square.apply(5)); // 출력: 25
    }
}
```

1.2.4. 스트림

1. 스트림
스트림(stream)은 컬렉션 등의 데이터를 함수형 스타일로 처리할 수 있도록 하는 문법임. 람다 표현식과 같이 Java8에 도입된 문법임.

스트림을 사용하면 데이터 집합의 데이터에 대한 집계, 변환, 필터링 등의 처리를 간단하게 수행할 수 있음.

2. 스트림의 특징
어떤 데이터 소스로 생성한 스트림에 대해 다양한 작업을 수행하더라도, 원본 데이터는 수정되지 않음. 즉, 스트림은 데이터를 읽기만 함. 작업 결과는 배열, 컬렉션 등에 담아서 반환함.

스트림은 재활용이 불가능함. 즉, 최종 연산 이후에는 해당 스트림을 사용할 수 없고, 다시 사용하려고 하면 예외가 발생함. 작업을 수행하려면 원본 데이터로부터 새로운 스트림을 생성해야 함.

중간 연산은 최종 연산이 호출될 때까지 지연되는데, 이를 지연 평가라고 함. 즉, 최종 연산이 수행될 때 중간 연산들이 실제로 수행됨.

3. 스트림 관련 메소드들
모든 내용을 정리할 수는 없어 개괄적으로 정리하였음. 필요한 문법은 그때그때 찾아 쓰자.

스트림 관련 메소드들은 스트림 생성, 중간 연산, 최종 연산으로 나눌 수 있음. 스트림을 생성하고 중간 연산을 통해 가공한 뒤, 최종 연산을 통해 결과물을 반환하는 것. 스트림 생성 메소드와 중간 연산 메소드의 반환 타입은 Stream<T>이고, 최종 연산 메소드의 반환 타입은 배열, 컬렉션 등임.

Collection 클래스에 있는 stream() 메소드로 컬렉션에 대해 스트림을 생성할 수 있음. 또한 배열, 수의 집합 등에 대한 스트림 생성 방법도 존재함.

중간 연산 메소드들로는 filter(), sorted(), distinct() 등이 있고, 최종 연산 메소드들로는 count(), max(), min(), findAny(), toArray(), collect() 등이 있음.

1.2.5. 직렬화

직렬화(Serialization)는 객체의 저장 또는 전송을 위해 객체를 연속된 바이트 형태로 변환하는 것을 말함. 반대로 역직렬화(Deserialization)는 변환되어 있는 바이트 형태의 데이터를 객체로 복원하는 것을 말함.

직렬화/역직렬화는 특정 객체의 값을 저장해둬야 하거나 다른 시스템으로 전송해야 할 때 사용되는 개념임. 물론 객체의 값은 JSON 등으로도 전송이 가능하지만, 컬렉션 등 복잡한 구조를 가지는 객체는 단순 변환이 불가능하기에 별도의 처리가 필요함. 또한 직렬화는 자바에서 제공하는 기능이므로 자바 시스템끼리의 호환성에 있어 유리함.

1.3. 추가 java 라이브러리들

1.3.1. lombok

롬복(lombok) 라이브러리는 생성자, getter/setter, toString(), equals() 등 반복되는 부분에 대한 코드를 자동 생성해 주는 라이브러리임.

클래스에 아래와 같은 애너테이션들을 붙여 생성이 가능함. 실제로 코드가 명시적으로 보이진 않지만 해당 메소드 등을 가져다 쓸 수 있게 됨.

@Getter/@Setter : getter/setter 생성.
@ToString : toString() 메소드(오버라이드) 생성.
@AllArgsConstructor : 모든 필드를 인자로 받는 생성자 생성.
@RequiredArgsConstructor : 모든 final 필드를 인자로 받는 생성자 생성.
@Data : @Getter, @Setter, @ToString, @EqualsAndHashCode, @RequiredArgsConstructor를 자동 적용함.

특히 스프링에서 의존관계를 생성자 주입으로 받으려는 경우, 생성자가 하나인 경우 @Autowired를 생략할 수 있으므로 @RequiredArgsConstructor만 작성하면 별다른 코드를 작성할 필요가 없음.

Spring Initializr(start.spring.io)에서 의존성 추가로도 적용 가능함.

1.4. spring 웹 개발 기초

스프링을 이용해 웹 어플리케이션을 개발하는 데에는 3가지 방법이 있음. 즉, 웹페이지를 구성하는 3가지 방법이 존재하는 것.

1.4.1. 웹 애플리케이션의 구조

스프링 부트를 사용한 일반적인 웹 애플리케이션은 컨트롤러, 서비스, 도메인, 리포지토리, DB로 구성된 구조를 가짐.

컨트롤러 : MVC에서의 그 컨토롤러로, 서비스와 도메인으로부터 얻은 데이터를 view로 전달하는 부분.
서비스 : 서비스의 핵심 비지니스 로직이 구현된 부분.
도메인 : 서비스에서 다루는 도메인(고객, 제품, 주문 등)을 클래스로서 구현한 부분.
리포지토리 : DB에 접근하여 도메인 객체를 저장하고 관리하는 기능을 수행하는 부분.

각 요소는 클래스로 구현되는데, 구현 시에는 주 패키지 내부에 각 요소별로 패키지를 만들고 그 내부에 클래스를 작성할 수도 있고(계층형), 도메인 별로 패키지를 만들어 그 내부에 요소별 패키지를 만들 수도 있음(도메인형).

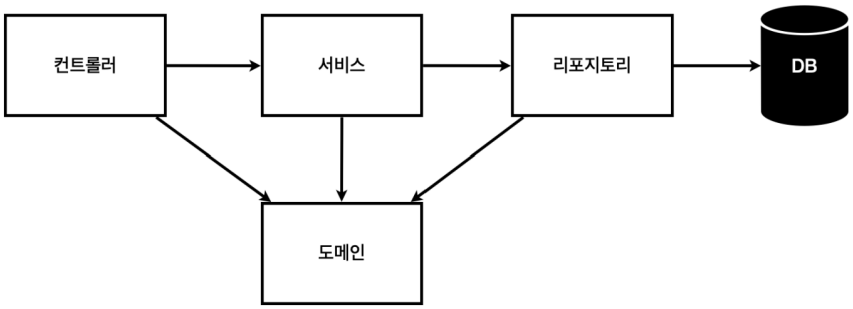
이때 도메인은 특정 문제를 해결하기 위한 비즈니스 로직과 데이터들을 포괄하는 개념으로도 이야기할 수 있으며, 도메인 내의 이용자/상품/주문 등의 고유 객체들을 엔티티(Entity)라고 함.

실제로 하는 작업이 유사할 수는 있어도, 대체로 서비스 부분은 비지니스 의존적으로 구현되고 리포지토리는 DB와의 상호작용을 중심으로 구현된다는 차이가 있음.

각 요소는 의존성에 따른 순서로 구현하는 것이 좋음.

구현이 교체 가능성이 있는 경우 등에서는 해당 요소에 해당하는 부분을 인터페이스로 작성함.

기획 단계에서는 도메인 협력 관계를 작성하게 되고, 이후 실제 개발 시에 구체적인 클래스/객체 관계(다이어그램 등)를 정하게 됨.



1.5. 테스트

1.5.1. JUnit

1. JUnit
JUnit은 테스트를 위한 프레임워크임.

main 메소드나 컨트롤러를 작성하여 테스트를 할 수도 있지만, JUnit은 반복적으로 여러 테스트를 한꺼번에 수행할 수 있음. 이런 테스트를 위한 프레임워크는 프로젝트 규모가 커질수록 그 효용이 증가함.

2. 테스트 코드 작성법
테스트 코드는 src/test/java 하위 패키지에 작성함. 이때 테스트 코드는 말 그대로 테스트만을 위한 것으로, 프로젝트 빌드에 포함되지 않음.

테스트 시에는 테스트할 클래스와 동일한 패키지 구조를 구성하고, 테스트 코드를 작성할 클래스의 클래스명은 테스트를 할 클래스의 클래스명 뒤에 Test를 붙여 지정하는 것이 관례임. 이후 해당 클래스 내에 각 테스트 케이스를 메소드 단위로 작성하는데, 해당 메소드에는 @Test 애너테이션을 붙임.

```
@Test
public void save()
{
    // 테스트 코드
}
```

작성한 테스트 코드는 클래스 단위로 실행할 수도 있고, 패키지 단위로 실행할 수도 있음.

3. Assertions
이때 테스트 성공 여부는 출력 등으로 확인할 수도 있지만, Assertions 메소드들로 판단하는 것이 좋음. assert한 것이 틀렸다면 실행 시에 에러를 발생(테스트 실패)시킴. 맞았다면 정상적으로 수행/종료됨.

JUnit에서 제공하는 Assertions 또는, Assertj 라이브러리를 사용할 수 있음. Junit Assertions는 org.junit.jupiter.api.Assertions를 포함시켜야 하고, Assertj Assertions는 org.assertj.core.api.Assertions를 import해야 함.

JUnit Assertions에는 assert...() 꼴 메소드 등이 있고, Assertj Assertions에는 assertThat().is...() 꼴 메소드 등이 있음.

예를 들어, 아래와 같이 작성할 수 있음. 필요한 메소드들은 사용 시에 찾아보자.

```
import static org.junit.jupiter.api.Assertions.*; // JUnit assertions
import static org.assertj.core.api.Assertions.*; // assertj assertions

...

Member rst = dataBase.findById(member1.getId()).get();
assertThat(rst).isEqualTo(member1);
```

추가로, JUnit의 assertThrows()를 사용하여 특정 예외가 발생하는지를 확인할 수 있음. assertThrows()는 해당 예외 객체를 반환하므로 올바른 예외가 발생한 것인지를 예외 메세지 등으로 검증할 수 있음.

4. LifeCycle 메소드들
@Test 애너테이션이 붙은 각 메소드(테스트케이스)의 LifeCycle에 따라 특정 메소드들이 호출되도록 할 수 있음. 이를테면 아래와 같이 @AfterEach 애너테이션을 붙여 각 테스트케이스의 호출 직후마다 호출되는 LifeCycle 메소드를 정의할 수 있음.

```
@AfterEach
public void afterEach(){
    repository.clearStore();
}
```

당연하게도 모든 메소드에 대한 테스트 코드를 필수적으로 작성해야 하는 것은 아님. 테스트가 충분하다면 굳이 작성할 필요가 없음.

1.5.2. 테스트케이스 작성 팁

1. 각 테스트케이스의 독립성 유지
각 테스트케이스는 서로 독립적이어야 함. LifeCycle 메소드 등을 사용하여 각 테스트케이스의 시작과 종료 시에 독립성 유지를 위해 필요한 작업을 수행해야 할 수 있음.

예를 들어, 리포지토리에 대한 테스트를 수행한 이후에 데이터셋을 초기화해야 다음 테스트에 영향을 미치지 않음.

2. given, when, then 패턴
given, when, then 패턴은 테스트케이스 작성 패턴으로, 테스트케이스를 더 직관적으로 작성/확인할 수 있도록 함.

given : 테스트 준비 단계.
when : 테스트 수행 단계.
then : 테스트 결과 검증 단계.

예를 들어, 아래와 같이 작성할 수 있음.

```
@Test
public void save()
{
    //given
    Member member1 = new Member();
    member1.setName("Hello");

    //when
    dataBase.save(member1);

    //then
    Member rst = dataBase.findById(member1.getId()).get();
    assertThat(rst).isEqualTo(member1);
}
```

3. IntelliJ 단축키
윈도우에서 Ctrl+Shift+t로 지정한 클래스에 대한 테스트 코드 틀을 자동 생성할 수 있음.

4. 테스트 코드 메소드명
테스트 코드이므로 메소드명을 한글로 알아보기 쉽게 작성하기도 함.

5. 여러 상황에 대한 테스트케이스
정상적인 상황 외에도, 예외 상황에 대한 테스트 케이스를 꼼꼼하게 작성해야 함. 또한 테스트로는 어떤 결과가 맞는지도 확인해야 하고, 틀리는지도 확인해야 함.

1.5.3. 전체 프로젝트 테스트

스프링과 DB 연결 등 모든 것을 포함한 테스트 시에는 @SpringBootTest, @Transactional 등의 애너테이션을 해당 테스트케이스가 존재하는 클래스에 작성함.

@SpringBootTest : 스프링 컨테이너와 테스트를 함께 실행하도록 지정. 기존의 테스트 코드는 스프링 부트를 사용하지 않고 단순히 자바 코드를 돌린 것임.

@Transactional : 테스트 시작 직전에 트랜젝션(DB 수정 단위)을 시작하고, 테스트가 끝나면 반영하는 대신 원래의 상태로 롤백함. 이 경우 @BeforeEach 등의 메소드를 사용할 필요가 없음.

물론 테스트는 컨테이너와 DB를 배제하고 수행할 수 있다면 그렇게 하는 것이 좋음. 컨테이너와 DB가 포함되게 되면 테스트 시간이 굉장히 길어지게 됨.

1.6. 로깅

1.6.1. 로깅

운영 시스템에서 로그를 찍을 때는 System.out.println()가 아니라, 별도의 로깅 라이브러리를 사용하는 것이 좋음. 로그를 사용하면 보기에 편리하고, 상황에 맞춰 출력할 수 있고, 원하는 위치에 출력할 있고, 단순 콘솔 출력보다 성능이 좋음.

여러 로깅 라이브러리가 존재하고, 스프링 부트는 기본적으로 SLF4J, Logback 등을 포함함. SLF4J는 여러 로깅 라이브러리를 위한 인터페이스로서 기능하는 라이브러리이고, Logback은 SLF4J의 구현체로서 사용할 수 있는 라이브러리임.

로그는 TRACE, DEBUG, INFO, WARN, ERROR의 레벨을 가지는데, 특정 레벨 이후부터만 출력되도록 할 수 있음. 개발 환경에서는 DEBUG ERROR, 운영 서버에서는 INFO ERROR를 출력하도록

해야 함. application.properties에서 전체/특정 클래스의 출력 레벨을 지정할 수 있음.

```
# 전체 로그 레벨 설정
logging.level.root=info
# hello.springmvc 패키지와 그 하위 로그 레벨 설정
logging.level.hello.springmvc=debug
```

아래와 같이 작성하여 로그를 찍을 수 있음.

```
@RestController
public class LogTestController {
    private final Logger log = LoggerFactory.getLogger(getClass());

    @RequestMapping("/log-test")
    public String logTest() {
        String name = "Spring";

        log.trace("trace log={}", name);
        log.debug("debug log={}", name);
        log.info(" info log={}", name);
        log.warn(" warn log={}", name);
        log.error("error log={}", name);

        return "ok";
    }
}
```

롬복 라이브러리를 사용 중이라면 아래와 같이 간단히 작성할 수도 있음.

```
@Slf4j
@RestController
public class LogTestController {
    @RequestMapping("/log-test")
    public String logTest() {
        String name = "Spring";

        log.trace("trace log={}", name);
        log.debug("debug log={}", name);
        log.info(" info log={}", name);
        log.warn(" warn log={}", name);
        log.error("error log={}", name);

        return "ok";
    }
}
```

추가로, 로그 메소드 인자에는 log.debug("data=" + data)와 같이 연산이 들어가서는 안 됨. 해당 로그의 출력 여부와는 상관없이 항상 연산이 수행되므로 성능이 크게 저하될 수 있음.

1.7. 스프링 DB 접근

1.7.1. DBMS와 sql

1. DBMS
데이터베이스(DB. Database)는 체계적으로 정리된 데이터의 집합이고, 데이터베이스 관리 시스템(DBMS. Database management system)은 데이터를 저장/검색/관리할 수 있는 시스템임. 대표적인 DBMS로는 MySQL, Oracle, PostgreSQL 등이 있음.

DB 중 데이터를 테이블 형태로 저장하는 것을 관계형 데이터베이스, 테이블 형태가 아닌 다양한 형태로 저장하는 것을 비관계형 데이터베이스라고 함.

기본적으로 DBMS의 연산은 CRUD(Create, Read, Update, Delete)로 구성되어 있음. 이 중에서도 특히 Create와 Read가 주요 연산임.

2. SQL
SQL(Structured Query Language)는 관계형 데이터베이스에서 데이터 관리를 위해 사용하는 표준 언어임.

DB에 대해 알아볼 때 가장 먼저 확인해야 할 것은 CRUD가 어떻게 수행되느냐임. 관계형 데이터베이스의 경우에는 SQL을 어떤 형식으로 사용하는지를 확인할 수 있음. 당연하게도 모든 문법을 암기할 필요는 없고, 그때그때 찾아서 쓰면 될 듯.

1.7.2. JdbcTemplate

1. JdbcTemplate
Jdbc(Java Database Connectivity)는 자바에서 DB와 상호작용하기 위한 표준 API로, 관계형 데이터베이스에 접속하고 SQL 쿼리를 실행한 뒤 그 결과를 사용할 수 있도록 함. JdbcTemplate은 Jdbc의 번거로운 반복 작업들을 줄여주는, 스프링에서 제공하는 Jdbc의 추상화 계층임. 즉, JdbcTemplate은 Jdbc를 더 편리하게 사용할 수 있도록 추상화한 도구임.

현재 실무에서는 Jdbc 대신 JdbcTemplate을 주로 사용함.

2. 환경설정
JdbcTemplate(Jdbc)을 사용하여 H2 데이터베이스를 조작하려면 build.gradle의 dependencies에 아래의 코드를 추가해 줘야 함.

```
implementation 'org.springframework.boot:spring-boot-starter-jdbc'
runtimeOnly 'com.h2database:h2'
```

또한 스프링 부트에 DB 연결 설정을 추가하려면 resources/application.properties에 아래의 코드를 추가해 줘야 함.

```
spring.datasource.url=jdbc:h2:tcp://localhost/~/test
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
```

이때 url에는 DB에서 지정한 Jdbc URL을 지정해야 함. driver class는 해당 DB의 드라이버를 지정해야 함.

3. data source, jdbcTemplate 지정
JdbcTemplate에서는 JdbcTemplate 클래스의 객체를 사용해 작업을 수행할 수 있음. 아래와 같이 필드를 만들고, 스프링에게 의존성을 주입받으면 됨. 컨테이너는 resources/application.properties에 지정한 설정대로 의존성을 주입함. 즉, 사용하려는 DB로의 data source를 JdbcTemplate 객체에 넣어서 작업에 사용하는 것.

```
public class JdbcTemplateMemberReposiory{
    private final JdbcTemplate jdbcTemplate;

    @AutoWired
    public JdbcTemplateMemberReposiory(DataSource dataSource){
        jdbcTemplate = new JdbcTemplate(dataSource);
    }
}
```

4. JdbcTempalte을 통한 CRUD 수행
나름의 문법을 통해 수행이 가능함. 필요하다면 찾아보자.

1.7.3. JPA

1. JPA
JPA(Java Persistence API)는 자바 애플리케이션에서 ORM을 사용할 수 있도록 하는 API로, SQL 쿼리 대신 객체 중심으로 DB와 상호작용할 수 있도록 함.

ORM(Object-Relational Mapping)은 객체 지향 프로그래밍 언어를 사용하여 관계형 데이터베이스의 데이터를 쉽게 다룰 수 있게 하는 기술을 말함.

JPA는 표준 인터페이스이고, 해당 구현은 여러 기술 중에 선택이 가능함.

2. 환경설정
JPA를 사용하려면 build.gradle의 dependencies에 아래의 코드를 추가해 줘야 함. 해당 라이브러리는 Jdbc 또한 포함하고 있음.

```
implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
runtimeOnly 'com.h2database:h2'
```

또한 스프링 부트에 DB 연결 설정과 JPA 관련 설정을 추가하려면 resources/application.properties에 아래의 코드를 추가해 줘야 함.

```
spring.datasource.url=jdbc:h2:tcp://localhost/~/test
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa

spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=none
```

3. 도메인에 애너테이션 지정
JPA를 사용하려면 아래와 같이 도메인 클래스와 그 필드에 @Entitiy, @Id 등을 붙여줘야 함.

```
@Entity
public class Member {
     @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
     private Long id;
     private String name;

     public Long getId() {
        return id;
     }
     ...
 }
```

4. EntitiyManager
JPA에서는 EntitiyManager 클래스의 객체를 사용해 작업을 수행할 수 있음. 아래와 같이 필드를 만들고, 스프링에게 의존성을 주입받으면 됨. 컨테이너는 resources/application.properties에 지정한 설정대로 의존성을 주입함.

```
private final EntitiyManager em;

@AutoWired
public JpaMemberRepository(EntitiyManager em){
    this.em = em;
}
```

5. JPA를 통한 CRUD 수행
나름의 간단한 문법을 통해 수행이 가능함. 필요하다면 찾아보자.

1.7.4. 스프링 데이터 JPA

데이터 JPA는 스프링에서 제공하는 도구로, 작성한 인터페이스 메서드명만으로 기본적인 CRUD에 대한 구현을 자동으로 수행함.

기본적인 CRUD에 대해서는 구현할 필요가 없으므로 굉장히 편리하지만, 특수한 상황에 대한 대체는 불가능하기 때문에 데이터 JPA를 사용하기 이전에 JPA에 대한 지식이 선행되어야 함.

1.8. AOP

1.8.1. AOP

관점 지향 프로그래밍(AOP, Aspect-Oriented Programming)은 공통 관심사를 모듈화하여 핵심 관심사로부터 분리하는 프로그래밍 방식임.

스프링에서는 AOP를 사용하여 가독성과 유지보수를 편리하게 할 수 있고, 중복 코드를 제거할 수 있음.

Part II

HTTP 기본

1. 웹

1.1. 인터넷 네트워크

1.1.1. 인터넷 네트워크

인터넷(Internet) 또는 인터넷 네트워크는 전 세계의 다양한 디바이스와 컴퓨터가 상호 연결되어 있는 네트워크 시스템으로, 여러 프로토콜과 기술을 이용해 데이터를 주고받음.

클라이언트와 서버의 통신은 대체로 일대일로 이루어지지 않고, 그 사이에 인터넷이 존재하게 됨. 데이터가 인터넷의 여러 노드들을 거치며 서로에게 이동하는 것. 클라이언트와 서버는 광범위한 인터넷에서 서로 적절한 곳으로 데이터를 전달하고 전달받기 위해 프로토콜(규약)을 사용함.

1.1.2. IP, TCP

1. IP
IP(Internet Protocol)는 인터넷에서 데이터 전송을 위한 프로토콜임. IP에 의해 네트워크 내의 각 장치들은 IP 주소로 식별되고, 데이터를 패킷 단위로 포장하여 전송함.

패킷(Packet)은 전송할 데이터와 출발지 IP, 목적지 IP 등을 포함하는 전송 단위임. 인터넷의 각 노드들은 패킷을 전달받으면 IP 주소를 확인하고 패킷을 적절한 목적지로 전달함.

IP 프로토콜은 여러 가지 한계를 가짐. 패킷을 받는 대상에 대한 검증 없이 패킷을 보내게 되고, 전송 중간 패킷이 소실되는 경우에 대해 보증할 수 없고, 패킷이 순서대로 전달되지 않을 수 있고, 하나의 IP에 여러 애플리케이션의 패킷이 들어오는 경우를 처리하지 않음. TCP 프로토콜을 사용하면 이런 한계를 극복할 수 있음.

2. TCP
TCP(Trasmission Control Protocol)는 패킷의 전송을 제어하는 프로토콜임.

TCP에 의해 TCP/IP 패킷에는 출발지 포트, 도착지 포트, 순서, 검증 정보 등이 포함됨.

TCP는 3 way hadshake 기법 등을 활용해 클라이언트와 서버가 연결이 된 이후 데이터를 보증하도록 함(연결 지향). 1. 클라이언트는 서버에 SYN(접속 요청) 신호를 보내고, 2. SYN 신호를 받은 서버는 클라이언트에 ACK(요청 수락)+SYN 신호를 보냄. 3. 클라이언트는 이 신호를 받으면 다시 ACK 신호를 서버에 전송함. 이렇게 총 3번의 신호 전송을 통해 클라이언트와 서버가 연결되었음을 확인함.

TCP는 데이터가 목적지에 도달했음을 보증함. 데이터가 목적지에 도달하면 목적지에서는 출발지에 데이터가 도착했음을 알리는 신호를 전송함.

TCP는 패킷의 순서를 보증함. 패킷의 순서를 명시하여 잘못된 순서로 패킷이 도달한 경우 전송을 다시 요청하거나 순서를 맞추도록 함.

3. UDP
UDP(User Datagram Protocol)는 기본 IP 프로토콜에서 포트 등 몇 가지 기능만 추가하는 프로토콜임.

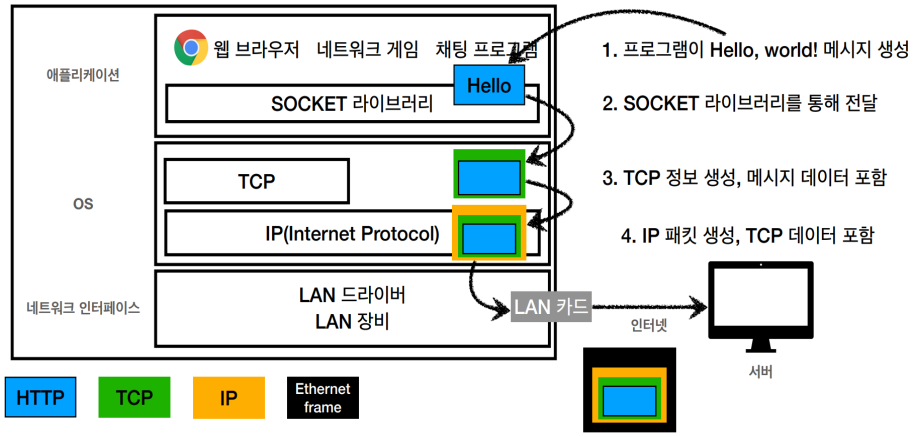
TCP만큼 연결지향적이거나 데이터에 대한 보증을 하지는 못하지만, 단순하고 빠름. 대신 애플리케이션에서 추가적인 작업을 수행해야 함.

1.1.3. TCP/IP 프로토콜 계층

TCP/IP 프로토콜 계층은 인터넷 네트워크 통신을 위한 프로토콜 계층으로, 아래와 같이 4개의 계층으로 이뤄진 모델임.

애플리케이션 계층 : 응용 프로그램과 상호작용하는 계층. (HTTP, HTTPS, DNS 등)
전송 계층 : 데이터 전송에 대한 신뢰성/오류 검증 등을 수행하는 계층. (TCP, UDP)
인터넷 계층 : 데이터 패킷이 인터넷 네트워크에서 전달될 수 있도록 하는 계층. (IP)
네트워크 인터페이스 계층 : 물리적인 네트워크 전송에 대한 계층.

애플리케이션 계층에서 데이터를 하위 계층으로 전달하면 각 계층에 해당하는 처리가 수행되며 결과적으로 전송할 패킷이 생성됨. TCP/IP에 의한 처리가 이루어진 패킷을 TCP/IP 패킷이라고 함.



1.1.4. 포트

포트(Port)는 하나의 IP 주소를 가지는 장치 내에서 각 프로세스별로 패킷을 구분하여 주고받기 위한 논리적인 연결점임.

0부터 65535까지의 숫자가 배정되어 있고, 그 중 0부터 1023까지는 미리 정의된 포트들이므로 사용하지 않는 것이 좋음. (ex. 텔넷은 23, HTTP는 80, HTTPS는 443 등)

1.1.5. DNS

DNS(Domain Name System)는 인터넷에서 도메인 네임을 IP 주소로 변환해주는 시스템임.

IP 주소는 기억하기 불편하고 변경될 수 있으므로 도메인 네임을 대신 사용하고, DNS가 해당 이름을 IP 주소로 변환해주는 식으로 동작함.

1.2. 웹의 동작 원리

1.2.1. URI

1. URI
URI(Uniform Resource Identifier)는 인터넷 리소스을 구분하기 위해 사용되는 통일된 형식의 문자열로, URL과 URN으로 구분됨. URL은 리소스을 위치 기반으로 구분하는 방식이고, URN은 리소스을 이름 기반으로 구분하는 방식임. URN보다는 URL이 주로 사용됨.

2. URL
URL(Uniform Resource Locator)은 위치 기반으로 인터넷 리소스을 구분하기 위해 사용되는 통일된 형식의 문자열임. URL은 아래와 같은 형식을 가짐.

```
scheme://[userinfo@]host[:port][/path][?query][#fragment]
```

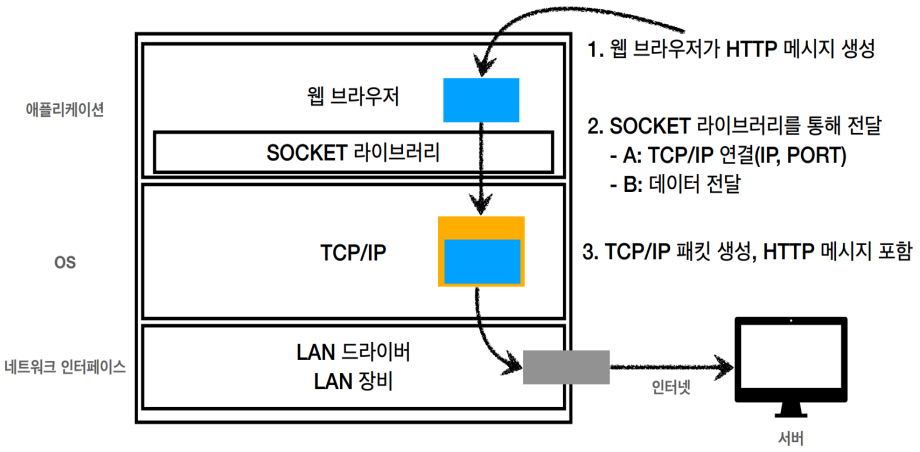
scheme : 주로 프로토콜(HTTP, HTTPs 등)을 명시함. HTTP는 80번 포트가, HTTPs는 443번 포트가 자동 적용되어 뒤에 포트를 생략할 수 있음.
userinfo@ : 사용자 정보를 인증에 사용하도록 지정함. 잘 사용하지 않음.
host : 도메인 네임 또는 IP 주소를 지정함.

port : 접속 포트를 지정함. 일반적으로 생략함.
path : 리소스 경로를 계층적 구조로 명시.
query : 웹 서버에 제공하는 파라미터를 ?로 시작하고 &로 구분하여 key=value 형태로 작성함. 값이 문자열로 사용됨.
fragment : HTML 내부 북마크 등에 사용되는 fragment를 지정함. 서버에 전달되지 않는 부분이고, 잘 사용되지 않음.

1.2.2. 웹의 동작 원리

웹 브라우저에 URL 입력 시에 웹 동작 원리는 아래와 같음.

1. 클라이언트에서 웹 브라우저에 URL을 입력하면 웹 브라우저는 해당 URL을 기반으로 HTTP 메시지를 생성함.
2. TCP/IP 프로토콜에 따라 해당 HTTP 메시지를 데이터로 하는 TCP/IP 패킷이 생성됨.
3. 패킷이 서버로 전송됨.
4. 서버에서 패킷의 데이터를 확인하고 내부 동작을 거친 뒤 응답 내용(HTML 등)을 패킷으로 클라이언트에 전송함.
5. 클라이언트는 패킷을 확인하여 HTML 등을 활용해 웹 브라우저에서 화면을 렌더링함.



2. HTTP

2.1. HTTP

2.1.1. HTTP

1. WWW
WWW(World Wide Web)는 웹 리소스와 멀티미디어 컨테츠 등을 포함하는, 인터넷을 통해 접근이 가능한 정보 리소스의 집합임.

2. HTTP
HTTP(HyperText Transfer Protocol)는 WWW에서 정보를 주고받기 위해 사용하는 프로토콜임. 최초에는 HTML 등을 주고받기 위한 것이었지만, 현재는 일반적인 데이터 교환에 사용되됨. 웹에서는 모든 것이 HTTP를 사용해 서로 통신함.

여러 버전이 존재하는데 현재 사용되는 HTTP의 문법은 HTTP/1.1을 바탕으로 함. 이후에 발표된 HTTP/2, HTTP/3 등도 존재하지만 문법적인 변화보다는 성능적인 변화가 이루어져 왔음.

2.1.2. HTTP의 특징

1. 클라이언트-서버 구조
HTTP는 클라이언트의 요청과 서버의 응답으로 구성되어 있음.

2. 무상태(Stateless)
서버가 클라이언트의 요청에 대한 상태를 보존하지 않음.

HTTP는 기본적으로 무상태이기 때문에 매 요청 시마다 서버군 내의 다른 서버가 응답해 줄 수 있어 서버의 확장에 용이함. 다만 요청이 저장되지 않기 때문에 클라이언트는 매번 모든 데이터를 전송해야 하는 낭비가 발생함.

3. 비연결성(Connectionless)
TCP/IP에 의해 클라이언트와 서버는 연결이 된 후 데이터를 교환하는데, HTTP는 기본적으로 이 연결을 유지하지 않음. 즉, 데이터 교환이 끝나면 연결을 끊음.

큰 규모의 서비스에서도 실제 서버에서 동시해 처리하게 되는 요청은 많지 않으므로 비연결성에 의해 서버 리소스을 아낄 수 있음.

물론 여러 데이터를 주고받아야 하는 경우 계속해서 연결을 끊었다가 다시 연결하면 비효율적이므로 어느 정도 연결을 유지하는 지속 연결(Persistent Connections) 등의 최적화가 이뤄져 있음.

2.1.3. HTTP 메시지 구조

1. 기본 구조
HTTP 메시지는 아래와 같이 시작 라인(Start-line), 헤더(header), 공백 라인(empty line), 메시지 바디(Message Body)의 순서로 구성되어 있음.

```
HTTP/1.1 200 OK                                 -> 시작 라인
Content-Type: text/HTML;charset=UTF-8           -> 헤더
Content-Length: 3423                            -> 헤더

<HTML>                                          -> 메시지 바디
    <body>...</body>
</HTML>
```

2. 시작 라인
시작 라인은 요청 메시지인 경우 request-line으로, 응답 메시지인 경우 status-line으로 작성함. 이때 SP는 공백 문자, CRLF는 엔터를 의미함.

request-line의 경우 아래와 같이 작성함. method에는 HTTP 메서드를, request-target에는 요청 대상(URL에서 /path?query 부분. 이외에 다른 지정 방법들도 존재함)을, HTTP-version에는 HTTP 버전을(HTTP/1.1 등으로) 작성함.

```
request-line = method SP request-target SP HTTP-version CRLF
```

status-line의 경우 아래와 같이 작성함. status-code에는 HTTP 상태 코드를, reason-phrase에는 이해가 편하도록 상태 코드에 대한 짧은 설명을 작성함.

```
status-line = HTTP-verison SP status-code SP reason-phrase CRLF
```

HTTP 상태 코드로는 아래와 같은 것들이 있음.
200 : 성공
400 : 클라이언트의 요청 오류
500 : 서버 내부 오류

3. 헤더
헤더에는 HTTP 전송에 필요한 정보를 작성함. 아래와 같이 header-field와 filed-name, filed-value의 형태로 여러 개의 필드를 작성할 수 있음. 여기서 OWS는 선택적 공백(공백 작성이 가능)를 의미함. 참고로 field-name에는 대소문자에 상관 없이 작성이 가능함.

```
header-field = field-name ":" OWS field-value OWS
```

기본적으로 아래와 같이 요청의 경우는 Host를 지정하고, 응답의 경우는 Content-type과 Content-Length를 지정함.

```
(요청)
GET /search?q=hello&hl=ko HTTP/1.1
Host: www.google.com
-----
(응답)
HTTP/1.1 200 OK
Content-Type: text/HTML;charset=UTF-8
Content-Length: 3423

<HTML>
    <body>...</body>
</HTML>
```

4. 메시지 바디
실제 전송할 데이터를 작성함. HTML, CSS, Json, 이미지, 영상 등 바이트로 표현 가능한 정보를 넣을 수 있음.

2.2. HTTP 메서드

2.2.1. HTTP 메서드

HTTP에서는 작업에 따라 적절한 메서드를 지정해 요청 메시지를 구성해야 함. HTTP 주요 메서드는 아래와 같음.

1. GET
리소스 조회(데이터 획득) 메서드.

이때 서버에 전달할 데이터는 query에 작성함. 메시지 바디에도 작성할 수는 있지만 호화되지 않는 경우가 많음.

아래와 같이 특정 리소스에 대해 요청하면 해당되는 데이터가 응답됨.

```
(요청)
GET /members/100 HTTP/1.1
Host: localhost:8080
-----
(응답)
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 34

{
 "username": "young",
 "age": 20
}
```

2. POST
데이터 전송과 그에 따른 처리 요청 메서드.

POST는 일반적인 데이터 전송으로, 특정 리소스에 POST 요청이 왔을 때의 처리는 정해진 것이 없음. 어떤 처리를 할 지는 서버에서 리소스마다 정의해야 함. 서버에서는 POST에 대해 새 리소스 등록,

단순 데이터 처리, 또는 다른 메서드로 처리하기 어려운 작업 수행 등을 함.

*post*는 뭔갈 보내는 모든 것을 할 수 있지만 의미적으로 명확하다면 적절한 메서드를 사용하는 것이 좋음.

아래와 같이 POST로 리소스 등록을 수행할 수 있음. 등록 시에는 응답 시에 해당 내용을 다시 보내고, 신규 리소스 생성 시에는 주로 201을 보냄.

```
(요청)
POST /members HTTP/1.1
Content-Type: application/json

{
    "username": "young",
    "age": 20
}
-----
(응답)
HTTP/1.1 201 Created
Content-Type: application/json
Content-Length: 34
Location: /members/100

{
    "username": "young",
    "age": 20
}
```

3. PUT

리소스 생성 메서드. 이때 해당 리소스가 이미 존재하면 덮어쓰고, 존재하지 않는 경우에만 생성함.

POST로 리소스를 생성할 때는 클라이언트는 리소스를 특정 위치에 전달만 할 뿐 리소스를 구체적으로 지정하지 않았는데, PUT에서는 클라이언트가 리소스를 구체적으로 지정함. 지정한 것이 존재하면 덮어쓰고, 존재하지 않으면 생성하는 것.

덮어쓸 때는 특정 항목만을 추가하는 것이 아니라, 요청에 포함한 데이터로 리소스 전체를 완전히 대체함.

아래와 같이 리소스를 지정하여 요청함.

```
(요청)
PUT /members/100 HTTP/1.1
Content-Type: application/json

{
    "username": "old",
    "age": 50
}
-----
(응답)
HTTP/1.1 200 OK
Content-Type: application/json

{
    "username": "old",
    "age": 50
}
```

4. PATCH
리소스 부분 변경 메서드.

PATCH는 PUT과 동일하게 작성함. 이때 PUT으로 리소스를 덮어쓰면 리소스가 완전히 대체되는데, PATCH를 사용하면 기존 리소스를 요청에 포함된 데이터로 부분 변경할 수 있음.

5. DELETE
리소스 제거 메서드.

아래와 같이 특정 리소스를 지정하여 삭제할 수 있음.

```
(요청)
DELETE /members/100 HTTP/1.1
Host: localhost:8080
```

정리하면 HTTP를 활용하여 클라이언트에서 서버로 데이터를 전송하는 방법으로는 1. 쿼리 파라미터를 이용하는 방법과 2. 메시지 바디를 이용하는 방법이 있음. 쿼리 파라미터는 GET에서 사용하며 주로 해당 파라미터를 이용하여 검색을 수행함. 메시지 바디는 POST, PUT, PATCH에서 사용하며 주로 리소스에 대한 조작을 수행함.

2.2.2. HTTP 메서드의 속성

HTTP 메서드들은 아래와 같은 특성으로 분류가 가능함.

1. 메시지 바디 존재 여부
필요에 따라 요청과 응답 각각에 메시지 바디는 존재할수도, 존재하지 않을 수도 있음.

기본적으로 GET, DELETE의 요청에는 메시지 바디가 없음. 응답에는 대체로 메시지 바디가 있음.

2. 안전(Safe)
호출해도 리소스를 변경하지 않는 메서드를 안전하다고 함.

GET, PUT, DELETE 등은 안전함.

3. 멱등(Idempotent)
여러 번 연속해서 호출해도 결과가 동일한 메서드를 멱등이라고 함. 이때 외부 요인으로 인해 결과가 달라지는 것은 고려하지 않음.

이를테면 요청했는데 서버에서 응답이 없다면 멱등인 것들은 단순히 여러 번 더 호출할 수 있음.

GET, PUT, DELETE 등은 멱등임.

4. 캐시 가능(Cacheable)
요청에 대한 응답을 캐싱할 수 있는 메서드를 캐시 가능하다고 함.

요청에 대한 응답을 캐싱하여 성능을 향상시킬 수 있음. 특히 GET 등의 경우 URL을 key로 하여 캐싱할 수 있음.

GET 등은 캐시가 가능함.

2.2.3. HTTP를 활용한 데이터 전송 상황

HTTP를 활용하여 클라이언트에서 서버로 데이터를 전송하는 상황에는 아래와 같은 유형이 있음.

1. HTTP api로 전송
HTTP api에 맞춰 HTTP 메시지로 데이터를 전송하는 경우.

단순 데이터 조회에는 GET과 쿼리 파라미터를 사용함. 이미지, 텍스트 등의 정적 데이터를 조회하는 경우에는 단순히 GET으로 조회가 가능하고, 검색, 정렬 등의 작업에서 동적 데이터를 조회하는 경우에는 GET에 쿼리 파라미터를 작성하여 조회할 수 있음.

리소스 조작에는 POST, PUT, PATCH 등과 메시지 바디를 사용함. 이때 다양한 형식으로 데이터를 전송할 수 있지만, 주로 Json을 사용함(Content-Type을 application/json으로 지정).

2. HTML form으로 전송
HTML에서 form 태그를 사용하는 경우. form 태그를 사용하면 HTTP 메시지가 자동으로 생성되고 전송됨. 이때 action으로 지정한 host/path에 method로 지정한 HTTP 메서드로 해당 form의 데이터를 전송함.

순수한 HTML form은 get과 post만 지원하는데, 당연하게도 조회에는 get을 작성하고 리소스 조작의 경우 post를 작성함.

form 태그의 method 속성에 get을 작성한 경우 아래와 같이 쿼리 파라미터로 데이터가 전송됨.

```
GET /save?username=kim&age=20 HTTP/1.1
Host: localhost:8080
```

form 태그의 method 속성에 post를 작성한 경우 아래와 같이 메시지 바디로 데이터가 전송되고, Content-Type이 application/x-www-form-urlencoded로 지정됨. 이때 메시지 바디는 쿼리 파라미터와 동일한 형식으로 작성됨.

```
POST /save HTTP/1.1
Host: localhost:8080
Content-Type: application/x-www-form-urlencoded

username=kim&age=20
```

form 태그에서 이미지, 파일 등 바이너리 데이터를 전송해야 하는 경우 속성으로 enctype="multipart/form-data"를 지정하면 multipart 형식으로 html 메시지가 형성됨. 자세한 내용은 필요하면 찾아보자.

2.3. HTTP api 설계

2.3.1. REST api

1. api
통신에서 api(Application Programming Interface)는 두 소프트웨어 구성 요소가 서로 데이터를 주고받을 수 있도록 정의된 인터페이스임. 서버가 제공하고자 하는 데이터/기능에 대한 api를 만들어 놓으면, 클라이언트는 해당 api로 데이터/기능을 활용할 수 있음.

api는 java api 등 라이브러리로서의 개념으로도 존재하는 용어이지만, 통신에서는 그 의미가 살짝 다름. 물론 프로그래밍을 위한 인터페이스라는 의미에서는 동일한 매커니즘을 가지고 있음.

2. HTTP api
HTTP api는 HTTP를 사용하여 통신하는 api임. 웹에서 이루어지는 대부분의 통신은 HTTP api로 이뤄짐.

3. REST api
REST(Representational State Transfer)는 HTTP 기반 네트워크 통신의 아키텍처 스타일임. 즉, REST는 HTTP를 잘 활용할 수 있도록 하는 원칙 세트이고, REST를 사용하는 api를 REST api라고 함.

REST api는 여러 주요 원칙들을 가지지만(https://restfulapi.net/resource-naming/), 가장 핵심 원칙은 아래와 같이 리소스과 행위를 분리하는 것임.
1. URI는 리소스(리소스)를 나타내야 함.
2. 리소스에 대한 행위는 HTTP 메서드로 나타냄.

엄밀하게는 REST의 주요 원칙들을 모두 지키는 것이 REST api이지만, 실제로 모든 원칙을 준수하기는 어렵기 때문에 대체로 HTTP api와 REST api는 유사한 의미로서 사용됨. 즉, HTTP api에 대한 설계를 한다고 하면 이는 Restful하게 이뤄져야 함. 예를 들어, URL을 지정할 때 각 계층과 대상은 리소스로 구성되도록 하는 것이 일반적임.

4. URI 설계 관련 용어들
Restful한 URI 설계에서는 아래와 같은 용어들이 사용됨.

문서(document) : 레코드/인스턴트 등 하나의 리소스 대상.
컬렉션(Colleciton) : 서버가 관리하는 리소스 디렉토리. 복수 명사로 이름 붙임.

스토어(Store) : 클라이언트가 관리하는 리소스 디렉토리. 복수 명사로 이름 붙임.
컨트롤러(Controller)/컨트롤 URI : 문서/컬렉션/스토어만으로는 처리가 어려운 작업을 수행하는 부분. 동사로 이름 붙임.

컨트롤러는 최소한으로 사용하는 것이 Restful한 설계이지만, 불가피한 경우도 많이 발생함.

2.3.2. HTTP API 설계 예시

Restful한 HTTP API는 아래와 같이 설계할 수 있음.

1. POST 기반 API
리소스 등록을 POST로 수행하도록 설계한 API. 대부분의 경우 POST 기반의 방식을 사용함.

POST로 리소스를 등록할 때는 리소스 URI를 명시하지 않으므로, 클라이언트는 리소스 등록 시에 URI를 고려하지 않음. 서버가 알아서 처리함. 이때의 서버가 관리하는 리소스 디렉토리가 컬렉션임.

예를 들어, 아래와 같이 설계함.
/members에 대해 GET으로 회원 목록 조회, POST로 회원 등록 수행.
/members/id에 대해 GET으로 회원 조회, PATCH/PUT/POST/DELETE로 회원 리소스 조작.

2. PUT 기반 API
리소스 등록을 PUT으로 수행하도록 설계한 API.

PUT으로 리소스를 등록할 때는 리소스 URI를 명시하므로, 클라이언트가 직접 리소스 URI를 고려해야 함. 이때의 클라이언트가 관리하는 리소스 디렉토리가 스토어(Store)임.

예를 들어, 아래와 같이 설계함.
/members에 대해 GET으로 회원 목록 조회, POST로 다중 리소스 등록(별도의 지정한 작업 수행).
/members/id에 대해 GET으로 회원 조회, PUT으로 회원 등록, PUT/PATCH/DELETE로 회원 리소스 조작.

3. HTML 기반 API
순수한 HTML의 경우 form에서 get과 post만을 지원함. 단순히 조회에는 GET을, 리소스 조작에는 POST를 쓰도록 설계하면 됨.

2.4. HTTP 상태 코드

2.4.1. HTTP 상태 코드

응답 메시지에 작성되는 HTTP 상태 코드는 아래와 같음.

1. 100번대 (Informational)
: 요청이 수신되어 처리중인 경우.

거의 사용되지 않음.

2. 200번대 (Successful)
: 요청이 정상적으로 처리된 경우.

200 OK : 요청 성공.
201 Created : 요청 성공, 새로운 리소스 생성됨.
202 Accepted : 요청 성공, 아직 처리는 되지 않음.
204 No Content : 요청 성공, 응답으로 보낼 데이터가 존재하지 않음.

주로 200만 사용하거나, 200과 201만 사용함.

3. 300번대 (Redirection)
: 리다이렉션 수행을 해야 하는 경우. 응답 메시지에 Location 헤더가 존재하면, 응답을 받은 순간 해당 위치로 이동(리다이렉트)하여 요청을 다시 보냄.

리다이렉션에는 3가지 종류가 있음.

1. 영구 리다이렉션 : 리소스의 URI가 영구적으로 이동됨. 검색 엔진 등에 반영되어 이후에도 바뀐 URI

를 사용함.
301 Moved Permanently : 리다이렉트 시에 (대체로) 요청 메서드가 GET으로 변하고 본문이 제거됨.
308 Permanent Redirect : 리다이렉트 시에 메서드와 본문 내용을 유지함.

2. 일시 리다이렉션 : 리소스의 URI가 일시적으로 이동됨.
302 Found : 리다이렉트 시에 (대체로) 요청 메서드가 GET으로 변하고 본문이 제거됨.
307 Temporary Redirect : 리다이렉트 시에 메서드와 본문 내용을 유지함.
303 See Other : 리다이렉트 시에 반드시 요청 메서드가 GET으로 변하고 본문이 제거됨.

3. 특수 리다이렉션 : 특수한 목적의 리다이렉션.
304 Not Modified : 리소스가 수정되지 않았음을 알림. 클라이언트는 해당 메시지를 받으면 기존의 캐시를 사용함.

4. 400번대 (Client Error)
: 클라이언트의 오류로 요청 수행이 불가능한 경우. 동일한 조건으로 재요청하면 실패함.

400 Bad Request : 클라이언트의 요청이 잘못됨.
401 Unauthorized : 인증(로그인)되지 않음.
403 Forbidden : 인증(로그인)은 되었지만 인가(접근 권한)되지 않음.
404 Not Found : 요청한 리소스가 존재하지 않음.

5. 500번대 (Server Error)
: 서버의 오류로 요청 수행이 불가능한 경우. 서비스 로직 등의 문제가 아니라 예외 발생, DB 연결 오류 등 중대한 서버 오류를 나타냄.

500 Internal Sever Error : 서버 내부 문제 발생. 대부분의 경우에 해당함.

HTTP 상태 코드는 5가지로 분류되므로, 클라이언트가 인식할 수 없는 상태 코드를 서버가 전송하더라도 클라이언트는 맨 앞자리 숫자로 어떤 상태인지를 알 수 있음.

2.4.2. PRG

PRG(Post-Redirect-Get) 패턴은 웹 애플리케이션에서 form 제출 후 리다이렉션을 하여 새로고침 등에 의한 중복 제출을 방지하는 설계 패턴임.

웹브라우저에서 새로고침을 하면 직전에 요청된 http 메시지를 다시 보냄. 문제는 단순히 POST로 데이터를 전송하고 200 OK로 응답받는 경우, 새로고침을 하면 직전의 POST가 한 번 더 전송되는 것임. 이때 POST에 대한 응답에 GET으로 리다이렉션하는 상태 코드를 넣으면 새로고침하더라도 GET이 수행되도록(리소스 변경 없음) 할 수 있음.

즉, 아래와 같이 http 메시지를 작성하면 됨.

```
(요청)
POST /order HTTP/1.1
Host: localhost:8080

itemId=mouse&count=1
-----
(응답)
HTTP/1.1 302 Found
Location: /order-result/19

-----
(요청)
GET /order-result/19 HTTP/1.1          // 바로 다음 요청을 보내게 됨.
Host: localhost:8080

-----
(응답)
HTTP/1.1 200 OK
```

```
<html>
    <body>주문완료.</body>
</html>
```

3. HTTP 헤더

3.1. HTTP 헤더 분류

3.1.1. HTTP 헤더 분류

HTTP 헤더는 HTTP 전송에 필요한 부가정보를 작성하는 부분으로, HTTP/1.1에서는 아래와 같이 4 가지로 분류할 수 있음.

1. General 헤더 : HTTP 메시지 전체에 적용되는 정보를 작성하는 헤더.
2. Request 헤더 : 요청 정보를 작성하는 헤더.
3. Response 헤더 : 응답 정보를 작성하는 헤더.
4. Entitiy 헤더 : 엔티티 바디에 대한 정보를 작성하는 헤더.

엔티티는 엔티티 본문과 그에 대한 메타데이터(엔티티 헤더)로 구성되는 데이터를 의미함. HTTP 메시지에서 엔티티 본문은 메시지 바디에 작성하고, 엔티티 헤더를 헤더에 작성함.

3.1.2. 표현

표현(Representation)은 표현 데이터와 그에 대한 메타데이터(표현 헤더)로 구성되는 데이터를 의미함. HTTP 메시지에서 표현 데이터는 메시지 바디에 작성하고, 표현 헤더를 헤더에 작성함.

HTTP/1.1에서는 엔티티라는 개념을 사용했지만, HTTP/2, HTTP/3에서는 표현이라는 개념으로 이를 대체하였음. 실제 데이터에 대한 표현으로서의 데이터를 전송하는 것이기 때문에 표현이라는 워딩을 사용하는 것.

3.2. 기본 HTTP 헤더

3.2.1. 표현 헤더

표현 데이터에 대한 표현 헤더로는 아래와 같은 것들이 있음.

Content-Type : 표현 데이터 미디어 타입(ex. application/json, image/png).
Content-Encoding : 표현 데이터의 인코딩 방식.
Content-Language : 표현 데이터에 사용된 자연 언어.
Content-Length : 표현 데이터의 길이. 바이트 단위로 작성.

3.2.2. 협상

협상 헤더는 클라이언트가 요청 메시지에 작성하여, 선호하는 표현을 요청하는 헤더임.

Accept : 선호 미디어 타입.
Accept-Encoding : 선호 인코딩 방식.
Accept-Language : 선호 자연 언어.

이때 선호하는 대상을 여러 개 작성할 수도 있고, 각 대상 별 우선순위를 지정해(숫자, 구체성 등으로) 상위 순위의 대상이 없을 때 하위 대상을 전달받을 수도 있음.

3.2.3. 전송 방식

아래와 같이 전송 방식 별 헤더 지정이 가능함.

1. 단순 전송 : 별다른 지정 없이 전송 가능.
2. 압축 전송 : Content-Encoding으로 인코딩(압축) 방식을 지정하여 전송 가능.
3. 분할 전송 : Transfer-Encoding으로 분할 전송 지정 가능.
4. 범위 전송 : Content-range로 특정 범위만 지정하여 전송 가능.

3.2.4. 정보 교환

아래의 헤더들로 클라이언트와 서버 사이에 정보 교환을 할 수 있음.

1. 일반 정보
From : 유저 에이전트(클라이언트 애플리케이션)의 이메일 정보. 요청 시 사용.
Referer : 현재 요청된 페이지의 이전 페이지의 주소. 요청 시 사용.
User-Agent : 유저 에이전트 정보. 요청 시 사용.
Sever : origin 서버의 소프트웨어 정보. 응답 시 사용.
Date : 메시지 발생 시간. 응답 시 사용.

2. 주요 정보
Host : 요청할 호스트의 도메인 네임. 요청에서 필수적으로 사용. 하나의 IP를 가지는 서버가 여러 개의 도메인 네임을 가지고 있을 수 있음.
Location : 응답 상태 코드가 300번대이면 해당 위치로 리다이렉트함. 201이면 요청에 의해 생성된 리소스의 URI임.
Allow : 허용 가능한 HTTP 메서드. 응답 상태 코드가 405 Method Not Allowed인 경우 사용 가능한 메서드를 헤더로 지정해 보내줘야 함.
Retry-After : 유저 에이전트가 다음 요청까지 기다려야 하는 시간. 응답 상태 코드가 503인 경우 얼마나 기다려야 하는지 알려줄 수 있음.

3.2.5. 인증

인증과 관련된 헤더는 아래와 같음.

Authorization : 클라이언트 인증 정보. 서버로 인증 정보 전달 시 사용.
WWW-Authenticate : 인증 방법 전달. 응답 상태 코드 401 Unauthorized가 발생한 경우 인증 방법을 제공하는 식으로 사용함.

3.2.6. 쿠키

쿠키(Cookie)는 클라이언트-서버 사이의 상태나 사용자 정보를 저장하기 위해 클라이언트 쪽에 저장되는 데이터임.

HTTP는 기본적으로 무상태(Stateless)이기 때문에 필요한 모든 데이터를 매번 서버에 전달해야 함. 하지만 이를 HTTP 작성 시마다 전부 작성하는 것은 불편하므로, 클라이언트 쪽에서 쿠키 저장소에 쿠키(데이터)를 저장해두고 필요할 때마다 브라우저가 꺼내서 사용하도록 한 것임.

쿠키 관련 헤더는 아래와 같음.

Set-Cookie : 응답 시에 서버에서 클라이언트로 쿠키를 전송함. 나름의 형식에 따라 만료일, 도메인, 보안 설정 등을 지정할 수 있음.
Cookie : 요청 시에 클라이언트에서 보관하던 쿠키를 전송함.

3.3. 캐시 관련 헤더

3.3.1. 캐시

HTTP 캐시는 HTTP를 사용하여 데이터를 요청할 때, 웹브라우저나 프록시 서버가 웹 리소스를 저장해 두고 필요할 때 이를 재사용하는 것임. 웹 리소스를 캐싱하면 불필요한 요청과 데이터 이동을 줄일

수 있음.

캐싱된 데이터는 브라우저 캐시에 저장되고, 나름의 유효 기간을 가지고 있어 해당 기간 안에서는 클라이언트가 데이터를 마음대로 꺼내서 사용함. 유효 기간이 끝나면 클라이언트는 해당 데이터에 대한 요청을 다시 보내며, 서버는 데이터를 재전송하거나 유효 기간을 갱신함. 유효 기간 갱신을 할 경우 데이터 전체를 전송할 필요가 없으므로 효율적임.

캐시 데이터와 서버 데이터가 같은지 검증하는 데이터를 작성하는 헤더를 검증 헤더라고 하고, 캐시 데이터에 대한 정보를 작성하여 조건부 요청을 수행하도록 하는 헤더를 조건부 요청 헤더라고 함.

캐시 검증 헤더는 2가지로 나누어 볼 수 있음.

1. 시점 기준 캐싱
서버는 데이터 요청에 따른 응답 시에 Last-Modified 헤더로 마지막 수정 시점을 함께 전송함. 클라이언트는 해당 데이터와 전달받은 마지막 수정 시점을 브라우저 캐시에 저장함.

캐시의 유효 기간이 지난 경우, 클라이언트는 아래와 같이 If-Modified-Since 헤더로 마지막 수정 시점을 포함시켜 데이터 요청을 함.

```
GET /star.jpg
If-Modified-Since: 2020년 11월 10일 10:00:00
```

서버는 전달받은 마지막 수정 시점과 실제 마지막 수정 시점이 다르면 200 OK와 함께 데이터를 다시 전송하고, 같으면 메시지 바디를 비우고 304 Not Modified로 응답함.

```
(시점이 같은 경우)
HTTP/1.1 304 Not Modified
Content-Type: image/jpeg
Cache-Control: max-age=60
Last-Modified: 2020년 11월 10일 10:00:00
Content-Length: 34012
```

2. ETag 기준 캐싱
시점 대신 버전별 문자열 등을 사용하여 수정 여부를 나타낼 수 있는데, 이를 통해 직접 로직을 만들어 캐시를 관리할 수 있음.

이 경우 Last-Modified와 If-modified-since 대신 Etag와 If-None-Match를 사용하면 됨.

3.3.2. 프록시 캐시

1. 프록시 서버
서버에는 origin 서버와 프록시(Proxy) 서버가 있음. origin 서버는 실제 HTTP 요청에 대한 응답을 하는 서버이고, 프록시 서버는 origin 서버와 클라이언트 사이의 중계 역할을 하는 서버임.

2. 프록시 캐시
모든 데이터를 클라이언트에서 캐시하는 대신 프록시 서버에 데이터를 캐시하고, 필요 시 해당 데이터를 사용하기도 함. 이때의 프록시 서버의 캐시를 프록시 캐시라고 함.

클라이언트의 캐시를 private 캐시, 프록시 서버의 캐시를 public 캐시라고도 함. 개인정보 등 민감한 데이터는 public 캐시에 저장되어서는 안 됨.

3.3.3. Cache-Control

Cache-Control 헤더는 캐싱 동작을 제어하는 헤더임. 여러 헤더가 존재하지만 이는 호환성을 위한 것이고 Cache-Control로 대부분 제어가 가능함. 이 헤더에는 아래와 같은 것들을 지정할 수 있음.

max-age=(시간) : 캐시 유효 시간 지정.

no-cache : 캐시 적용, 항상 origin 서버에 검증하고 데이터 사용.
no-store : 캐시하지 않음.
must-revalidate : 캐시 적용, 반드시 origin 서버에 검증하고 데이터 사용. (검증 실패 시 504 발생)
public : 응답이 public 캐시에 저장될 수 있음.
private : 응답이 private 캐시에만 저장되어야 함.
s-maxage=(시간) : 프록시 캐시의 유효 시간 지정.

특히 캐시 무효화 관련해서 주의해야 하는데, no-cache로 지정한 경우 origin 서버에 접근하지 못하면 브라우저 등에서 프록시 서버의 데이터를 사용해 버릴 수도 있음. 이때는 must-revalidate를 지정하면 접근 실패 시 504 코드를 발생시킴.

Part III

스프링

1. 스프링 기본

1.1. 스프링

1.1.1. 스프링

스프링(Spring)은 엔터프라이즈 애플리케이션 등의 개발에 사용되는 오픈 소스 자바 프레임워크로, 좋은 객체 지향 프로그래밍을 할 수 있도록 지원함.

현재의 스프링에는 스프링 프레임워크, 스프링 부트가 필수적으로 포함되고, 추가적으로 스프링 데이터/세션/시큐리티/배치/클라우드 등이 사용됨. 이런 스프링의 요소들을 통틀어 스프링 생태계라고도 함. 스프링 프레임워크는 컨테이너, AOP, MVC, JDBC, 테스트 등을 포함하는 핵심 요소임. 스프링 부트는 톰캣, starter 등을 포함하여 스프링을 편리하게 사용할 수 있도록 하는 요소임.

스프링은 기존에 엔터프라이즈 애플리케이션 개발에서 주로 사용되던 EJB의 반발로서 등장했음.

1.1.2. 좋은 객체 지향 프로그래밍

객체지향 프로그래밍은 프로그램을 명령어의 목록으로 보는 대신, 객체들의 집합과 그 객체들 사이의 상호작용으로 보는 것을 말함. 결과적으로 객체지향 프로그래밍은 프로그램을 유연하고 교체 용이하게 함.

좋은 객체 지향 프로그래밍은 다형성을 많이 활용(역할과 구현의 구분)하고, SOLID를 잘 준수하는 프로그래밍을 말함.

기존 자바 문법만으로는 SOLID의 OCP, DIP를 완전히 지킬 수는 없는데, 스프링은 이런 객체지향적 요구사항들을 충족하여 프로그래밍을 할 수 있도록 돕는 프레임워크임.

물론 현실적으로 모든 기능을 분리하는 것은 좋은 방법이 아닐 수도 있음. 변경 가능성이 없는 부분은 단순 구현으로 작성하는 것이 효율적일 수 있음.

1.1.3. SOLID

SOLID는 객체지향 설계의 5가지 원칙을 말함.

1. SRP(Single Responsibility principle, 단일 책임 원칙)
한 클래스는 하나의 책임만을 가져야 함. 즉, 변경이 존재할 때 파급효과가 적어야 함.

2. OCP(Open/Closed principle, 개방-폐쇄 원칙)
소프트웨어 요소는 확장에는 열려 있고 변경에는 닫혀 있어야 함. 즉, 기능의 확장은 용이하게 하되,

구체적인 코드의 변경은 지양하도록 설계해야 함. (다형성 활용)

물론 단순 자바 문법만으로 OCP를 완전히 지킬 수는 없는데(코드의 변경이 발생할 수밖에 없음), 이에 따라 스프링에서는 컨테이너가 사용됨.

3. LSP(Liskov Substitution principle, 리스코프 치환 원칙)
객체는 원래의 기능을 유지하며 하위 타입의 객체로 대체될 수 있어야 함. 즉, 기능적인 보장이 이뤄지며 하위 타입 객체로 대체될 수 있어야 함.

참고로 리스코프는 LSP를 제안한 사람 이름임.

4. ISP(Interface Segregation principle, 인터페이스 분리 원칙)
범용 인터페이스 하나를 사용하는 것보다 각 요소를 위한 인터페이스로 분리하는 것이 좋음. 인터페이스를 적절히 분리해야 유지보수와 대체가 용이해짐.

5. DIP(Dependency Inversion principle, 의존성 역전 원칙)
구체화에 의존하는 대신 추상화에 의존해야 함. 즉, 어떤 기능을 사용할 때 구체적인 클래스를 사용하는 대신 인터페이스로 해당 구현을 받아 사용하는 것이 좋음.

단순 자바 문법만을 사용한다면 구현 클래스를 직접 작성해야 함(의존하게 됨). 스프링에서는 컨테이너를 사용하여 이 문제를 해결함.

만약 자바 코드만으로 다형성을 활용하고 SOLID를 지키려면 아래와 같이 하면 됨.

1. 의존성에 따라 클래스 생성자에서 객체를 주입받도록 함.
2. 사용 시에 객체를 직접 생성하는 대신 객체를 생성해 주는 별도의 클래스(companion class)를 사용함. 참고로 AppConfig 내부의 객체 생성 메소드들도 알아보기 쉽게 최대한 기능별로 분리하여 작성하는 것이 좋음.

결과적으로, 구현체의 변경이 발생한 경우에도 AppConfig 클래스에서 해당 클래스명만 수정하면 됨.

1.1.4. 스프링의 주요 특징들

1. IoC(Inverse of Control, 제어의 역전)
: 객체지향 프로그림에서 객체의 생성 및 관리를 개발자가 아닌 외부에서 하도록 하는 것. 즉, 제어가 개발자로부터 외부(별도의 클래스 또는 컨테이너 등)로 넘어가는 것을 말함.

2. DI(Dependency Injection, 의존성 주입)
: 런타임에 외부에서 해당 객체가 필요로 하는 객체를 생성해 주입하는 것.

스프링에서 의존성(Dependency)이란 한 객체가 다른 객체에 의존하거나 해당 객체의 기능을 사용해야 하는 관계를 말함.

의존관계는 정적인(컴파일 타임, 클래스) 의존관계와 동적인(런타임, 객체) 의존관계로 구분하여 볼 수 있음. DI를 사용하면 다형성에 의해 정적인 의존관계를 수정하지 않고도 동적인 의존관계를 수정할 수 있음.

참고로, IoC는 프레임워크와 라이브러리의 구분에서도 유의미함. 프레임워크와 라이브러리는 모두 재사용 가능한 코드를 제공하지만, 프레임워크는 IoC가 적용된 것이고 라이브러리는 IoC가 적용되지 않은 것임. 프레임워크에서는 프로그램의 기본 구조가 정해져 있고, 개발자가 코드를 작성하면 프레임워크에서 그 코드를 기반으로 제어를 조작함. 반면에 라이브러리에서는 개발자가 필요할 때 라이브러리를 호출하고, 프로그램의 흐름을 제어함.

1.2. 빈과 컨테이너

1.2.1. 빈과 컨테이너

1. 스프링 컨테이너
스프링 컨테이너(Spring Container)는 1. 빈(객체)의 생성/설정/생명주기 등을 관리하고 2. 의존성을 주입하는 스프링 프레임워크의 핵심 요소임. 스프링 컨테이너, DI 컨테이너, IoC 컨테이너 등으로 부름.

스프링 컨테이너는 빈(객체)를 관리해야 함으로써 개발자의 부담을 줄여주고 SOLID를 지키며 다형성을 활용할 수 있도록 함.

2. 스프링 빈
스프링 빈(Spring Bean)은 스프링에서 스프링 컨테이너가 관리하는 객체임. 빈은 컨테이너에 의해 주입되고, 공용으로 사용될 수 있음.

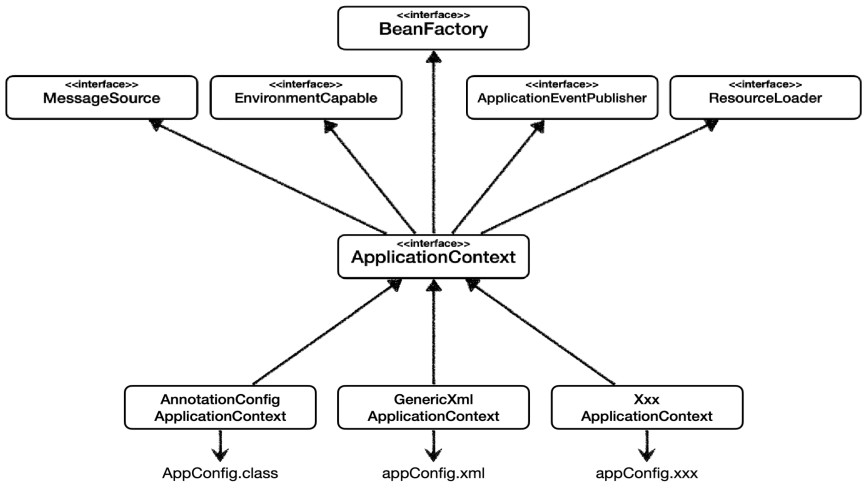
어떤 클래스를 컨테이너에 등록하면 컨테이너는 빈을 생성/관리하게 되는데, 이때 컴포넌트 스캔, configuration 클래스 등으로 클래스의 지정이 가능함.

1.2.2. ApplicationContext

1. BeanFactory
스프링 컨테이너의 최상위 클래스로, 스프링 빈의 관리와 조회를 담당하는 인터페이스임.

2. ApplicationContext
스프링 컨테이너 클래스로, BeanFactory와 그 외 부가 기능들에 대한 인터페이스들을 상속받는 인터페이스임. 스프링 컨테이너라고 하면 ApplicationContext로 가리키는 객체를 의미함.

ApplicationContext의 구현체는 AnnotationConfigApplicationContext 클래스 등이 있음.



1.2.3. 스프링 컨테이너 생성

ApplicationContext 인터페이스로 스프링 컨테이너를 생성할 수 있음. 컨테이너 생성 방식에 따라 다양한 구현체가 존재하지만(XML 기반 등), 여기서는 애너테이션 기반의 설정 클래스를 사용하는 방법을 정리함. XML 기반으로 생성하는 것도 유사한 방식으로 하면 됨.

1. configuration 클래스 작성
아래와 같이 클래스에 @Configuration 애너테이션을, 객체를 반환하는 메소드에 @Bean 애너테이션을 작성하여 스프링 컨테이너를 위한 configuration 클래스를 만들 수 있음.

이때 @Bean으로 지정한 메소드가 반환하는 객체가 스프링 빈으로 등록됨.

```
@Configuration
public class AppConfig{
    @Bean
    public MemberService memberService()
    {
        return (new MemberService(memberRepository()));
    }

    @Bean
    public MemberRepository memberRepository()
    {
        return (new MemberRepository());
    }
}
```

2. 스프링 컨테이너 생성
아래와 같이 작성해 애너테이션 기반의 configuration 클래스(여기서는 AppConfig)로 스프링 컨테이너를 생성할 수 있음. 또는 특정 클래스.class를 작성하여 해당 클래스의 객체만 빈으로 등록할 수도 있음.

```
ApplicationContext aC =
                new AnnotationConfigApplicationContext(AppConfig.class)
```

3. 스프링 컨테이너 생성 및 구성 과정
위와 같이 configuration 클래스로 컨테이너를 생성했다면 아래와 같은 과정이 수행됨.

1) AppConfig 클래스를 기반으로 스프링 컨테이너를 생성함.

2) AppConfig 클래스에서 @Bean 애너테이션이 붙은 메소드들을 실행해 반환받은 객체를 스프링 빈 저장소에 저장함. 빈은 빈 이름과 빈 객체로 구성되는데, 빈 이름은 해당 메소드명으로 지정되고 빈 객체는 반환받은 객체로 지정됨. 참고로 애너테이션에서 빈 이름을 직접 지정할수도 있음.

3) configuration 정보로 빈 사이의 의존관계를 설정(주입)함.

1.2.4. 스프링 빈 조회

컨테이너에 대해 아래의 메소드 등을 사용하여 스프링 빈을 조회할 수 있음. 컨테이너 객체를 가지고 있는 변수를 ac라고 가정함.

getBean(빈 이름, 타입.class) : 빈 이름과 타입으로 빈을 찾아 반환함. 빈 이름 또는 타입 둘 중 하나만 작성해도 됨. 타입 지정 시 해당 타입으로 가리킬 수 있는 빈이 조회됨. 이때 해당되는 빈이 없거나 2개 이상인 경우 예외가 발생함. 이 경우 getBeansOfType()을 사용해야 함.

getBeansOfType(타입.class) : 해당 타입으로 가리킬 수 있는 빈 전부를 Map<String,T>으로 반환함. 해당 Map은 빈 이름을 key로, 빈 객체를 value로 하는 자료구조임.

getBeanDefinitionNames() : 해당 컨테이너가 가진 모든 빈의 빈 이름을 문자열 배열로 반환함.

getBeanDefinition(빈 이름) : 해당 빈 이름의 빈에 대한 메타데이터를 객체로 반환함. BeanDefinition 타입으로 해당 객체를 가리킬 수 있음. (AnnotationConfigApplicationContext에 있는 메소드임)

타입으로 빈을 조회하는 경우 해당 타입으로 가리킬 수 있는 객체가 조회되므로, 상위타입으로 조회한다면 해당 타입과 그 하위타입이 모두 조회됨. 특히 Object로 조회하면 모든 빈이 조회됨.

1.2.5. BeanDefinition

BeanDefinition은 빈의 메타데이터를 저장하는데 사용되는 인터페이스임. BeanDefiniton의 구현체는 빈의 클래스명, 싱글톤, 생성시기 등 메타데이터에 대한 정보를 가지고 있음.

ApplicationContext 인터페이스에 대한 구현체는 내부적으로 BeanDefiniton의 구현체를 생성함. 이때 BeanDefintion은 빈 당 하나 생성되며, 스프링 컨테이너는 BeanDefinition으로 해당 객체를 읽어 빈을 생성함. 즉, 애너테이션 기반이든 xml 기반이든 사용자 정의 기반이든 BeanDefinition으로 가리킬 수 있으면 빈이 생성됨.

1.3. 싱글톤 컨테이너

1.3.1. 싱글톤 패턴

싱글톤 패턴(Singleton Pattern)은 어떤 클래스의 객체가 단 하나만 생성되고 공유되도록 하는 소프트웨어 디자인 패턴임.

싱글톤 패턴을 구현하는 다양한 방법이 있지만, 아래와 같이 단순하게 구현할 수 있음. 이 경우 SingletonSerivce의 객체는 단 하나만 생성될 수 있고, 해당 객체에 접근하려면 getInstance() 메소드를 사용해야 함.

```
public class SingletonService {
private static final SingletonService instance = new SingletonService();

    public static SingletonService getInstance() {
        return instance;
    }
    private SingletonService() {   // private 생성자로 생성 막음
    }
    ...
}
```

객체를 계속해서 생성하는 것이 비효율적인 경우 싱글톤 패턴을 사용하여 개선할 수 있지만, 위와 같이 구현할 경우 매번 코드를 작성해야 하는 번거로움이 있고 SOLID를 위반하게 됨. 이외의 여러 단점들이 존재하여 안티패턴으로 취급되기도 함.

1.3.2. 싱글톤 컨테이너

스프링에서는 컨테이너가 기존의 문제점들을 해결하면서 빈을 싱글톤으로 관리함. 즉, 스프링 컨테이너는 싱글콘 컨테이너로서 동작하고, 이런 싱글톤 방식의 관리 기능을 싱글톤 레지스트리라고 함.

스프링 컨테이너는 기본적으로 싱글톤을 적용하지만, 싱글톤을 적용하지 않도록 지정할 수도 있음.

1.3.3. 싱글톤 방식의 주의점

단순 자바로 싱글톤 패턴을 사용하든 스프링 컨테이너에 의한 싱글톤 패턴을 사용하든 싱글톤 방식에서 주의할 점은, **싱글톤이 적용되는 클래스에 클라이언트에 의존적인 공유 필드가 존재해서는 안 된다는 것임**. 즉, 해당 객체를 이용하는 각 클라이언트가 서로 영향을 주고받지 않도록 해야 함.

1.3.4. @Configuration

@Configuration은 특정 클래스를 스프링 설정 클래스로 지정하는 애너테이션임.

@Configuration은 여러 기능과 관련이 있고, 특히 @Bean 메소드들에 대해서 싱글톤을 보장해주는 역할을 함. 즉, 아래와 같이 @Bean 메소드끼리 서로 호출하는 경우에도 싱글톤이 보장됨. @Configuration 없이 @Bean만 작성하여 빈으로 등록하는 것도 가능하지만, 이 경우에는 내부에서 다른 @Bean 메소드를 호출하는 경우 결과적으로 여러 개의 객체가 생성될 수 있어 싱글톤이 보장되지 않음.

```
@Configuration
public class AppConfig {
    @Bean
    public static MemberService memberService()
    {
        return (new MemberServiceImpl(memberRepository()));
    }

    @Bean
    public static MemberRepository memberRepository()
    {
        return (new MemoryMemberRepository());
    }
}
```

@Configuration을 작성하면 내부적으로 해당 클래스의 하위 타입 클래스를 하나 만들고, CGLIB이라는 라이브러리로 기존의 메소드를 오버라이드한 바이트 코드를 삽입함. 해당 바이트 코드에 의해, 스프링 컨테이너에 빈이 이미 존재하면 해당 객체를 반환하고 존재하지 않는 경우에만 빈을 새로 생성하게 됨.

1.4. 컴포넌트 스캔

1.4.1. 컴포넌트 스캔

@Configuration 클래스에 @Bean 메소드로 빈을 등록할 수 있지만, 컴포넌트 스캔과 자동 의존관계 주입을 사용해 빈을 자동으로 등록하고 주입할 수도 있음.

1. @ComponentScan
아래와 같이 @Configuration 클래스에 @ComponentScan을 붙여 컴포넌트 스캔을 하도록 지정할 수 있음.

```
@Configuration
@ComponentScan
public class AutoAppConfig {
}
```

컴포넌트 스캔을 하는 경우에도 컨테이너는 기존의 방법과 동일하게 생성하면 됨.

2. @Component
@ComponentScan을 작성하여 컴포넌트 스캔을 하도록 지정했다면 아래와 같이 @Component가 붙은 클래스의 객체를 생성해 빈으로 등록함. 이때 빈 이름은 해당 클래스명에서 첫 문자를 소문자로 한 것으로 함. 물론 이름을 직접 지정할 수도 있음.

```
// memoryMemberRepository를 이름으로 객체가 빈으로 등록됨
@Component
public class MemoryMemberRepository {
    ...
}
```

기본적은 편리하고 직관적인 자동 빈 등록(컴포넌트 스캔)을 사용하되, 애플리케이션에 광범위한 영향을 미치고 그 수가 적은 기술 지원 로직(AOP 등)은 수동 빈 등록(@Bean)으로 명시하는 것이 좋음.

또한 빈을 자동 등록하고 Map<K,V>, List<E>로 여러 개의 빈을 주입받다면 어떤 빈들이 주입되는지 파악하기 불편하기 때문에, 이 경우에도 수동 빈 등록(@Bean)으로 작성하는 것이 좋음.

1.4.2. 자동 의존관계 주입

컴포넌트 스캔으로 등록된 빈은 컨테이너에 의해 다른 클래스에 자동으로 주입될 수 있는데, 자동 의존관계 주입에는 아래와 같이 여러 가지 방법이 존재함.

이때 주입은 빈에 대해서만 이뤄질 수 있음. @Component 등의 애너테이션이 붙지 않은 임의의 클래스에 @Autowired를 작성하면 주입되지 않음.

1. 생성자 주입
생성자 주입의 경우 아래와 같이 @Autowired를 생성자에 작성함.

```
@Component
public class MemberController {
    private final MemberService memberService;

    @Autowired
    public MemberController(MemberService memberService) {
        this.memberService = memverService;
    }
}
```

해당 클래스가 빈으로 등록될 때 생성자가 호출되는데, 컨테이너는 스프링 빈 저장소에서 매개변수로 작성한 타입과 호환되는 빈을 찾아 주입함. 생성자는 한 번만 호출되므로 ***필수적이고 수정이 없는 객체(빈)를 주입받을 때 주로 사용하는 방법임****.*

컴포넌트 스캔이 적용되는 클래스에 대해 ***생성자가 하나만 존재하는 경우 @Autowired를 생략해도 생성자로 의존성을 주입받을 수 있음****.*

2. setter 주입
setter 주입의 경우 아래와 같이 @Autowired를 setter에 작성함.

```
@Component
public class MemberController {
    private MemberService memberService;

    @Autowired
    public setMemberService(MemberService memberService) {
        this.memberService = memverService;
    }
}
```

컨테이너는 스프링 빈 저장소에서 매개변수로 작성한 타입과 호환되는 빈을 찾아 주입함. 이때 setter 주입이라고 setter 호출 시에만 주입되는 것은 아님. 컨테이너에 의한 최초 의존관계 주입 시에 자동으로 주입되고, 추후에 setter로 바꿀 수도 있는 것. setter를 이용한 값의 수정이 가능하므로 ***선택적이거나 변경이 가능한 객체(빈)를 주입받을 때 주로 사용하는 방법임****.*

3. 필드 주입
필드 주입의 경우 아래와 같이 @Autowired를 필드에 작성함.

```
@Component
public class MemberController {
    @Autowired
    private MemberService memberService;
}
```

필드 주입에서는 스프링을 사용하지 않는다면 해당 필드에 접근할 수 없어 테스트에 불리하므로 사용하지 않는 것이 좋음. 접근한다고 setter를 작성할 것이라면 애초에 setter 주입을 하면 됨.

테스트 코드 등에서는 사용하기도 함.

4. 일반 메서드 주입
setter 주입과 동일한 원리로 주입이 가능함. setter는 하나의 필드에 대해서만 작성하므로 일반 메소드 주입을 사용하면 여러 필드에 대해 한 번에 처리할 수 있음.

대부분의 경우 생성자/setter 주입으로 잘 사용되지 않는 방법임.

컨테이너는 1. 컨테이너 생성 2. 빈 등록 3. 의존관계 주입의 lifecycle을 가짐. 객체 생성 시에 생성자는 불가피하게 호출되므로 생성자 주입은 빈 등록 시에 수행되고, setter 주입은 이후 의존관계 주입 시점에 수행됨. 그래서 생성자 주입에서는 의존관계를 저장하는 변수에 final을 붙일 수 있지만, 다른 방법들에서는 붙이지 못함.

기본적으로 생성자 주입을 사용하고, 의존성에 변화가 불가피한 경우에만 setter 주입을 사용하는 것이 좋음. 또는 생성자 주입과 setter 주입을 함께 사용할 수도 있음. 생성자 주입과 함께 final을 지정하여 사용하면 많은 경우를 컴파일 에러로 바로잡을 수 있고, 순수 자바 코드만으로 테스트를 수행할 수 있음.

참고로 @Configuration에서 @Bean으로 지정해 등록된 빈도 @Component로 지정한 클래스에 주입될 수 있음.

1.4.3. 컴포넌트 탐색 위치

@ComponentScan은 기본적으로 해당 @Configuration 클래스가 위치한 패키지에 대해 컴포넌트 스캔을 수행함.

@ComponentScan에서 특정 패키지에 대해 컴포넌트 탐색을 하도록 지정할 수도 있지만, @Configuration 클래스(설정 정보 클래스)를 프로젝트의 최상단에 위치시키는 것이 관례임.

참고로 스프링 부트에서 기본 생성하는 스프링 부트 시작 클래스에 붙어 있는 @SpringBootApplication에는 @Configuration, @ComponentScan 등이 들어 있음.

1.4.4. 기본 스캔 대상

컴포넌트 스캔 시에는 @Component 외에도 아래의 애너테이션이 붙은 클래스들 또한 스캔되는데, 이는 이 애너테이션들이 내부적으로 @Component를 포함하기 때문임.

@Controller : 해당 클래스를 MVC 컨트롤러로서 인식함.
@Service : 해당 클래스가 비지니스 로직으로 포함함을 나타냄.
@Repository : 해당 클래스가 데이터 접근 계층임을 나타내고, 적절한 예외 처리 등을 수행함.
@Configuration : 해당 클래스에서 정의하는 내용을 설정 정보로 반영하고, 싱글톤 처리 등을 수행함.

별도의 애너테이션을 개발자가 직접 정의하고, @ComponentScan에서 includeFilters, excludeFilters 옵션으로 필터를 지정하면 특정 클래스를 스캔에 포함/배제하도록 지정할 수 있음.

참고로 애너테이션이 내부적으로 다른 애너테이션을 포함하는 것은 자바 문법이 아닌 스프링에서 자체적으로 지원하는 문법임.

1.4.5. 중복 등록 시 처리

자동으로 등록한 빈들끼리 이름이 중복되는 경우 예외가 발생하고, 자동으로 등록한 빈과 수동으로 등록한 빈이 이름이 중복되는 경우 수동으로 등록한 것이 우선되어 적용됨.

기본적으로 스프링에서는 이런 식으로 적용되지만, 빈 이름이 중복되는 경우는 대체로 의도되지 않은 경우이기 때문에 스프링 부트는 이에 대해 에러를 발생시킴. 즉, 결과적으로 빈 이름이 겹치는 경우가 없도록 코드를 짜야 함.

1.4.6. 빈이 존재하지 않는 경우의 처리

@Autowired에 의한 의존관계 주입 시에 빈이 존재하지 않는 경우는 아래의 방법들로 처리가 가능함.

1. @Autowired(required = false)
@Autowired 애너테이션에 required를 false로 지정하면 매개변수에 해당되는 빈이 존재하지 않는 경우 메소드가 호출되지 않음.

@Autowired의 required는 true가 default값임.

2. @Nullable
매개변수에 @Nullable 애너테이션을 지정하면 그 매개변수에 해당되는 빈이 존재하지 않는 경우 빈 대신 null이 주입됨.

3. Optional<T>
매개변수의 자료형을 Optional<T>로 지정하면 T에 해당되는 빈이 존재하지 않는 경우 빈 대신 비어 있는 Optional<T> 객체가 주입됨.

1.4.7. 빈이 여러 개 존재하는 경우의 처리 - 하나만 주입받는 방법

@Autowired에 의한 의존관계 주입 시에는 작성한 타입에 의해 빈이 결정되는데, 이때 여러 개의 빈이 존재하는 경우(해당 타입으로 가리킬 수 있는 객체가 여러 개인 경우)는 아래의 방법으로 빈을 특정하여 주입받을 수 있음.

1. @Autowired 필드명 매칭
타입 매칭을 시도했는데 빈이 여러 개 존재하는 경우, @Autowired는 의존관계를 주입하는 대상 클래스의 변수명(필드명 또는 매개변수명)과 특정 빈의 빈 이름이 같다면 해당 빈을 주입함.

변수명을 직접 지정해야 하기 때문에 번거롭고 다형성에 위배될 수 있음.

2. @Qualifier
아래와 같이 빈으로 등록할 클래스와, 해당 클래스의 객체를 주입받을 클래스의 변수명(필드명 또는 매개변수명) 앞에 @Qualifier 애너테이션을 작성하여 빈을 특정할 수 있음.

```
@Component
@Qualifier("mainMemberRepository")
public class MemmoryMemberRepository implements MemberRepository {
    ...
}

----------

public class MemberServiceImple implements MemberService {
    MemberRepository repository;

    @Autowired
    MemberServiceImple(@Qualifier("mainMemberRepository")
                                        MemberRepository repository) {
        this.repository = repository;
    }
}
```

@Qualifier는 컴포넌트 스캔에서도 사용이 가능하고, 직접 @Bean으로 수동 빈 등록 시에도 작성이 가능함. 이 경우 단순히 @Bean과 함께 해당 클래스에 작성하면 됨.

추가로 @Qualifier에 문자열을 매번 지정하여 사용하면 오타 등이 컴파일 타임에 체크가 안 됨. 이 경우 디버깅이 까다로워지므로 아래와 같이 애너테이션을 직접 정의해서 사용하는 것이 좋음. (이런 식의 정의는 스프링에서 제공하는 기능임)

```
@Target({ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER,
                            ElementType.TYPE, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Qualifier("mainDiscountPolicy")
public @interface MainDiscountPolicy {
}
```

3. @Primary
빈으로 등록할 클래스에 @Primary 애너테이션을 작성하면 여러 빈이 존재하는 경우에 해당 빈이 우선 사용됨.

참고로 @Qualifier와 @Primary가 충돌하는 경우에는 @Qualifier가 우선됨. 스프링에서는 구체적이고 좁은 의미의 문법이 대체로 우선권을 가짐.

자주 사용하는 빈을 @Primary로 지정해 두고, 가끔 사용하는 빈을 @Qualifier로 지정하는 것이 자연스러운 코드 관리에 좋음.

1.4.8. 빈이 여러 개 존재하는 경우의 처리 - 모든 빈을 가져오는 방법

@Autowired에 의한 의존관계 주입 시에는 작성한 타입에 의해 빈이 결정되는데, 이때 여러 개의 빈이 존재하는 경우(해당 타입으로 가리킬 수 있는 객체가 여러 개인 경우) Map<K,V>, List<E>로 해당되는 모든 빈을 주입받을 수 있음.

아래와 같이 단순히 기존 매개변수/필드 위치에 Map<String,타입>, List<타입>을 작성하면 해당되는 모든 빈이 주입됨. 이때 Map<K,V>의 key에는 빈 이름이 문자열로 저장됨.

```
@Component
public class MemberServiceImple {
    private final Map<String, DiscountPolicy> policyMap;

    @Autowired
    public MemberServiceImple(Map<String, DiscountPolicy> policyMap) {
        this.policyMap = policyMap;
    }
}
```

1.5. 빈 생명주기와 콜백

1.5.1. 빈 생명주기와 콜백

스프링 빈은 아래와 같은 생명주기를 가짐.

스프링 컨테이너 생성 - 스프링 빈 생성 - 의존관계 주입 - 초기화 콜백
- 빈 사용 - 소멸 전 콜백 - 빈 소멸 or 스프링 컨테이너 종료

이때 콜백(Callback)이란 스프링 빈의 생명주기 관리를 위해 사용되는 메소드로, 빈 초기화/소멸 시에 호출되어 필요한 작업들을 수행함.

빈의 의존관계 주입이 끝난 후 빈 초기화 작업(네트워크 연결 등)을 수행하거나, 빈의 사용이 끝나고 빈 소멸 전에 종료 작업을 수행해야 할 수 있는데, 문제는 기본적으로 개발자가 의존관계 주입이 끝나는 시점을 알 수가 없음. 이 때문에 콜백 메소드가 지원되는 것임.

참고로 초기화 작업이 규모가 작고 단순한 경우가 아니라면, 객체 생성과 초기화는 분리하는 것이 유지보수 등에 있어서 유리함. 그렇기에 초기화와 종료 작업에는 콜백 함수를 사용하는 것이 좋음.

1.5.2. 콜백 메소드 지정

스프링에서 콜백 메소드는 아래의 방법들로 지정할 수 있음.

1. @Bean에서 initMethod, destoryMethod로 지정
아래와 같이 @Bean에서 initMethod, destoryMethod에 해당 빈에 존재하는 특정 메소드명을 작성하면 해당 메소드들을 초기화/종료 콜백 메소드로 지정할 수 있음.

```
@Configuration
public class Config {
    @Bean(initMethod = "init", destoryMethod = "close")
    public static MemberService memberService()
    {
        return (new MemberServiceImple(memberRepository()));
    }
    ...
}
```

참고로 이때 destroyMethod를 지정하지 않아도 close, shutdown이라는 이름의 메소드가 존재하면 해당 메소드를 종료 콜백 메소드로 자동 등록함.

2. @PostConstruct, @PreDestroy로 지정
아래와 같이 빈의 클래스에서 특정 메소드에 @PostConstruct, @PreDestroy 애너테이션을 붙여 해당 메소드들을 초기화/종료 콜백 메소드로 지정할 수 있음.

```
@Component
public class NetworkClient {
    @PostConstruct
    public void init() {
        ...
    }

    @PreDestroy
    public void close() {
        ...
    }
}
```

기본적으로 @PostConstruct, @PreDestroy로 지정하는 방법을 사용하되, 외부 라이브러리에 지정해야 하는 경우에는 사용이 불가능해 @Bean에서 initMethod, destoryMethod로 지정하는 방법을 사용해야 함.

1.6. 빈 스코프

1.6.1. 빈 스코프

빈 스코프(Scope)는 빈의 라이프사이클과 존재 범위에 대한 개념임.

1. 스프링에서 지원하는 스코프들
스프링은 아래와 같은 스코프들을 지원함.

싱글톤 스코프 : 컨테이너의 시작과 종료까지 빈이 유지/관리되는 기본 스코프.
프로토타입 스코프 : 컨테이너가 생성/DI/초기화까지만 관여하고 이후에는 관여하지 않는 좁은 스코프.
웹 스코프 : request, session, application 등 웹 관련 스코프들.

2. 스코프 지정 방법
자동 등록이든 수동 등록이든 아래와 같이 @Scope 애너테이션으로 스코프를 지정할 수 있음.

```
@Scope("prototype")
@Component
public class MemoryMemberRepository {
    ...
}
```

앞에서 설명한 싱글톤 또한 하나의 스코프로 생각할 수 있음.

1.6.2. 프로토타입 스코프

프로토타입(Prototype) 컨테이너가 생성/DI/초기화까지만 관여하고 이후에는 관여하지 않는 스코프로, 빈 요청 시마다 새로운 인스턴스(빈)가 생성/DI/초기화하여 제공됨.

컨테이너가 생성/DI/초기화까지만 관여하므로 이후 해당 빈을 받은 클라이언트가 빈을 관리해야 하고, 당연하게도 종료 콜백은 수행되지 않으므로 클라이언트가 콜백 메소드를 직접 호출해야 함.

싱글톤은 컨테이너 생성 시에 생성/DI/초기화되는데에 반해 프로토타입은 요청 시에 생성/DI/초기화된다는 점에 유의하자.

1.6.3. provider

1. 싱글톤과 프로토타입 혼용 시 주의점
대부분의 빈은 싱글톤이므로 프로토타입 빈을 사용하는 경우 다른 싱글톤 빈과 혼용(주입)하게 되는 경우가 많음. 이때 주의할 점은, 싱글톤 빈이 프로토타입 빈을 주입받는 경우 주입받은 프로토타입 빈은 계속 유지되므로 싱글톤처럼 동작한다는 것임. (주입된 빈이 싱글톤 빈과 함께 공유되어 계속 사용되므로.) 프로토타입을 사용한다면 그 특성을 살려야 하므로(그렇지 않다면 싱글톤을 사용하면 됨), 이 경우 provider로 해결이 가능함.

2. ObjectProvider
스프링에서 제공하는 provider로, 지정한 타입의 빈을 컨테이너에서 찾아 제공함. 즉, DL(Dependency Lookup, 의존관계 탐색)을 수행함.

아래와 같이 실제 객체 대신 provider 객체를 주입받으면, provider의 메소드를 통해 원래의 필요한 객체를 새로 제공받을 수 있음.

```
@Component
@RequiredArgsConstructor
public class TestClass {
    private final ObjectProvider<PrototypeBean> provider; // provider 주입됨

    public void logic() {
        PrototypeBean pb = provider.getObject(); // 빈 획득
    }
}
```

provider는 프로토타입-싱글톤의 혼용 시 외에도, DL이 필요한 경우라면 아무데서나 사용할 수 있음.

ObjectProvider는 스프링 의존적이라는 한계가 존재하는데, 해당 기능이 스프링 독립적이어야 한다면 자바에서 표준으로 제공하는 JSR-330 provider를 사용할 수 있음. 필요하다면 찾아보자.

1.6.4. 웹 스코프

웹 스코프는 웹 환경에서 동작하는 스코프임.

웹 스코프는 프로토타입 스코프와는 달리, 생성부터 종료까지 컨테이너에 의해 관리되어 종료 콜백이 수행될 수 있음.

1. 웹 스코프들
웹 스코프들에는 아래와 같은 것들이 있음.

requset : HTTP 요청 하나가 들어오고 나갈 때까지 유지되는 스코프로, 각 요청 별 빈이 생성됨.
session : HTTP 세션과 동일한 생명주기를 가지는 스코프.
application : 서블릿 컨텍스트와 동일한 생명주기를 가지는 스코프.
websocket : 웹 소켓과 동일한 생명주기를 가지는 스코프.

2. 웹 스코프의 사용
프로토타입 스코프와 동일하게 @Scope 애너테이션으로 스코프를 지정할 수 있음. 이때 웹 스코프를 사용하기 위해서는 build.gradle에 웹 라이브러리(스프링 부트 스타터에서도 지정 가능)를 추가해야 함.

1.6.5. request 빈의 의존관계 주입

1. request 빈의 의존관계 주입
컨트롤러, 서비스 빈은 웹 스코프 빈을 주입받아야 할 수 있는데, request 빈 등의 경우 웹 요청이 들어와야 생성되므로 컨테이너의 DI 시에 빈이 존재하지 않음. 그래서 단순히 request 빈을 주입받도록 코드를 작성하면 에러가 발생함.

provider 또는 프록시를 사용하여 빈 생성을 지연시키면 이를 해결할 수 있음.

2. provider를 이용한 해결법
DI 시에 request 빈 대신 이 빈을 제공하는 provider를 주입받고, 실제 빈이 필요할 때 provider로부터 빈을 제공받는 방법.

아래와 같이 코드를 작성할 수 있음.

```
@Controller
@RequiredArgsConstructor
public class TestService {
    private final ObjectProvider<RequestBean> provider;

    public void logic() {
        RequstBean rb = provider.getObject();
        ...
    }
}
```

3. 프록시를 이용한 해결법
@Scope에 프록시를 사용하도록 지정하면 해당 클래스에 대한 프록시 빈을 대신 주입함. 이 프록시 빈은 싱글톤처럼 동작하므로 request 빈을 단순 주입받는 것처럼 코드를 작성할 수 있음.

@Scope에 아래와 같이 지정하기만 하면 됨.

```
@Component
@Scope(value = "request", proxyMode = "ScopedProxyMode.TARGET_CLASS")
public class RequestBean {
    ...
}
```

@Configuration에서와 동일하게 내부적으로 해당 클래스의 하위 타입 클래스를 하나 만들고, CGLIB

이라는 라이브러리로 기존의 메소드를 오버라이드한 바이트 코드를 삽입함. 이렇게 바이트 코드를 삽입한 하위 타입 객체를 프록시(대리인) 객체라고 함. 프록시 객체를 request 객체인 것처럼 사용하면 프록시 객체는 내부적으로 실제 request 객체를 찾아서 작업을 수행함. 다형성에 의해 개발자는 request 객체인 것처럼 생각하여 작업을 수행할 수 있음.

2. 스프링 mvc

2.1. 웹 애플리케이션

2.1.1. 웹 시스템 구성

1. 웹 서버
웹 서버(Web Server) HTTP 기반으로 통신하며 정적 리소스(HTML/CSS/JS, img 등)를 제공을 수행하는 서버 프로그램임.

대표적인 웹 서버로는 APACHE 등이 있음.

2. WAS
WAS(Web Application Server)는 HTTP 기반으로 통신하며 정적/동적 리소스의 제공과 애플리케이션 로직을 수행하는 서버 프로그램임.

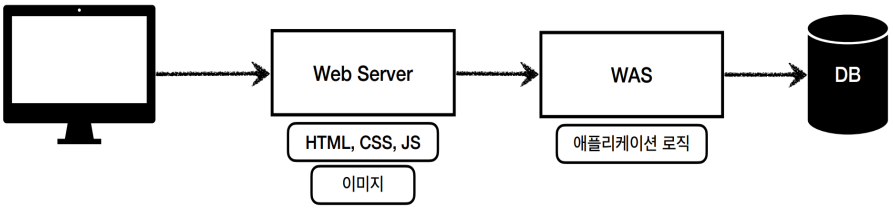
자바에서는 주로 서블릿 컨테이너로서 기능할 수 있으면 WAS라고 함.

대표적인 WAS로는 톰캣 등이 있음.

3. 웹 시스템 구성
WAS가 정적 리소스 제공과 애플리케이션 로직을 모두 처리할 수 있기 때문에 웹 시스템은 WAS와 DB만으로 구성이 가능함. 하지만 정적 리소스는 웹 서버가 제공하고 WAS는 주요 활동인 애플리케이션 로직에만 특화시키는 것이 안정성/확장성에 있어서 유리함.

특히 애플리케이션 특성에 따라 웹 서버, WAS 한쪽만 증설할 수도 있음.

물론 웹 서버와 WAS가 흑과 백처럼 완전히 구분되는 개념은 아님.



2.1.2. 웹 데이터 전송의 유형

요청에서의 데이터 전송은 1. 쿼리 파라미터를 이용한 GET으로 전송 2. 쿼리 파라미터 형태의 POST (HTML)로 전송 3. HTTP API로 전송으로 분류할 수 있음.

응답에서의 데이터 전송은 1. 정적 리소스(HTML, CSS, img 등) 전송 2. 동적으로 HTML 생성 후 전송(뷰 템플릿) 3. HTTP API로 전송으로 분류할 수 있음. 이때 데이터의 형태는 주로 1. 텍스트 2. HTML 3. Json임.

서버에서 데이터 전송 시에는 이 유형을 모두 고려해야 함.

HTTP API를 사용한 전송은 다양한 시스템에서 사용되는데, 주로 전송 대상이 서버인 경우가 많음.

2.1.3. 웹 렌더링

웹 렌더링 방식에는 SSR, CSR 등이 있음.

SSR(Server-Side Rendering)은 서버에서 HTML을 구성해서 웹 브라우저에 전달하는 방식임. 주로 정적인 화면에 사용되며, JSP, Thymeleaf등을 사용함(백엔드에서 주로 개발).

CSR(Client-Side Rendering)은 서버에서 전송한 HTML 기본 구조 등을 바탕으로 웹 브라우저에서 Javascript등을 사용해 동적으로 생성/적용하는 방식임. 주로 동적인 화면에 사용되며, React, Vue.js 등을 사용함(프론트엔드에서 주로 개발).

2.2. 서블릿

2.2.1. 서블릿

1. 서블릿
서블릿(Servlet)은 HTTP 요청을 수신하고 비즈니스 로직 수행 이후 HTTP 응답을 생성하여 반환하는 객체임.

실제 HTTP 통신을 위해서는 TCP/IP 연결 대기, HTTP 메시지 파싱, 저장, HTTP 메시지 생성, 응답 전달 등의 작업을 수행해야 함. 이 반복적인 과정을 개발자가 전부 관리하는 것은 번거롭기 때문에 서블릿이 이를 수행해 주는 것임.

2. 서블릿 컨테이너
서블릿 컨테이너는 서블릿을 생성, 호출, 관리하는 WAS임.

서블릿 컨테이너는 서블릿 객체에 대한 생성, 호출, 종료 등 생명주기를 관리함. 이때 서블릿 컨테이너는 서블릿 객체를 싱글톤으로 관리하고, 이에 따라 서블릿 객체는 서블릿 컨테이너 종료 시에 함께 종료됨.

대표적인 서블릿 컨테이너로는 톰캣이 있음. 스프링 부트는 톰캣을 기본으로 내장하여 컨테이너로 사용함.

2.2.2. 서블릿과 멀티스레드

1. 멀티스레스 지원
서블릿 컨테이너는 동시 요청을 처리하기 위해 멀티스레드를 지원함. 요청에 스레드를 할당하여 서블릿 객체를 사용할 수 있도록 하는 것.

물론 서블릿 컨테이너가 알아서 처리해주기 때문에, 개발자는 멀티스레드 관리를 크게 신경쓸 필요가 없음.

2. 스레드 풀
스레드 풀(Thread Pool)은 효율적인 스레드 관리를 위해 사용하는 디자인 패턴임. 미리 스레드를 지정한 만큼 생성해 두고, 할당/반납하는 식으로 설계하는 것.

모든 요청에 대해 스레드를 계속해서 생성하면 매 생성과 종료에 따른 비효율이 발생하고, 많은 스레드에 의한 컨텍스트 스위칭 비용이 증가하고, 하드웨어 자원이 전부 소진될 위험성이 생김. 이에 따라 스레드를 미리 생성해 두고, 할당 가능한 개수를 정해 놓는 것.

스레드 풀 사용 시에는 최대 스레드 수를 튜닝하여 성능을 최적화하는 것이 중요함. 서버 장애 발생 시 클라우드 기반이라면 서버 개수를 우선 늘리고 이후 튜닝하고, 그렇지 않다면 튜닝하는 식으로 해결함.

2.2.3. 서블릿 클래스

서블릿 클래스를 직접 작성하여 서블릿 컨테이너가 해당 클래스로 서블릿을 생성하도록 할 수 있음.

아래와 같이 @SpringBootApplication 클래스에 @ServletComponentScan 애너테이션을 작성하여 서블릿 클래스를 스캔하도록 할 수 있음. 이때 해당 클래스가 속한 패키지와 그 하위 패키지를 스캔함.

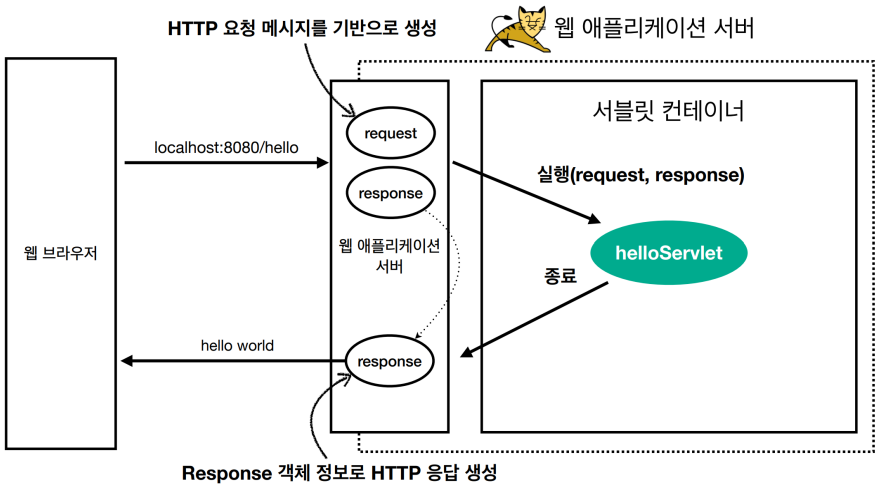
```
@ServletComponentScan //서블릿 자동 등록
@SpringBootApplication
public class ServletApplication {
    public static void main(String[] args) {
        SpringApplication.run(ServletApplication.class, args);
    }
}
```

아래와 같이 서블릿 클래스를 작성하여 서블릿 컨테이너가 해당 클래스의 객체를 사용하도록 할 수 있음. 지정한 URL이 호출되면 해당 서블릿 객체의 service() 메서드가 호출됨.

```
@WebServlet(name = "helloServlet", urlPatterns = "/hello")
public class HelloServlet extends HttpServlet {
    @Override
    protected void service(HttpServletRequest request,
                                        HttpServletResponse response) {
        // 애플리케이션 로직
    }
}
```

HTTP 요청이 들어오면 WAS는 요청에 대한 Request, Response 객체를 만들어서 서블릿 객체를 호출함. 개발자는 Request를 사용해 비지니스 로직을 수행하도록 하고, 응답 정보를 Response에 입력함. WAS는 Response의 내용으로 HTTP 응답을 생성함.

스프링 프레임워크와 서블릿은 관계 없는 개념이지만, 스프링 부트에서 톰캣을 내장하고 있으므로 편리하게 사용이 가능함.



2.2.4. HttpServletRequest

WAS는 HTTP 요청을 파싱하여 HttpServletRequest 객체에 담아 반환함. 이 객체는 메서드로 정보를 제공하는 등 여러 기능을 포함하고 있음.

1. HTTP 파싱하여 반환받기

HttpServletRequest의 메서드를 사용해 시작 라인, 헤더, 부가 정보 등을 각각 반환받을 수 있음.

GET을 사용해 쿼리 파라미터로 전송하든, POST를 사용해 메시지 바디에서 쿼리 파라미터 형식으로 전송하든 getParameter(), getParameterValues()(값이 여러 개인 경우) 등으로 값을 반환받을 수 있음.

Json으로 데이터를 전달받은 경우, 해당 값을 객체로 받아 사용하려면(기본적으로는 문자열로 반환받

게 됨) 별도의 라이브러리가 필요함. 스프링 mvc에서 기본으로 제공하는 ObjectMapper를 사용할 수 있음.

2. 기타 기능
HttpServletRequest는 요청 메시지 파싱 이외에도 여러 기능들이 포함되어 있음. 특히 데이터 저장을 위한 별도의 공간을 가지고 있고, 이를 settAttribute(), getAttribute()로 데이터를 관리할 수 있음. mvc 패턴에서 model로 사용 가능함.

2.2.5. HttpServletResponse

전달받은 HttpServletResponse 객체에 응답에 대해 지정하면 WAS는 이 정보로 응답 메시지를 구성해 클라이언트에 반환함.

메서드로 응답 코드, 헤더, 메시지 바디 등을 지정할 수 있음. 특히 메시지 바디의 경우 getWriter() 메서드로 PrintWriter 객체를 반환받을 수 있는데, 이 PrintWriter 객체의 메서드를 사용해 정보를 입력할 수 있음.

2.3. 스프링 mvc 구조

2.3.1. 템플릿 엔진과 mvc 패턴

1. 템플릿 엔진
템플릿 엔진은 웹페이지를 동적으로 구성하는 도구임.

템플릿 엔진을 사용하면 매번 HttpServletResponse에 문자열을 직접 작성해 넣는 대신, HTML 파일에 별도의 코드만 일부 작성하여 HTTP로 전송할 수 있음. 템플릿 엔진을 사용하면 주로 mvc 패턴을 함께 적용함.

스프링 mvc에서는 템플릿 엔진으로 JSP, thymeleaf 등을 사용할 수 있는데, thymeleaf가 권고됨.

2. mvc 패턴
mvc(Model, View, Controller) 패턴은 controller에서 내부적인 작업을 수행하고 그 결과를 model에 담아서 view에 전달하면 view에서 웹페이지를 출력하는 디자인 패턴임.

mvc 패턴을 사용하여 비즈니스 로직과 뷰 렌더링을 분리하면 각각 기능에 특화되도록 할 수 있고, 유지보수가 용이해짐.

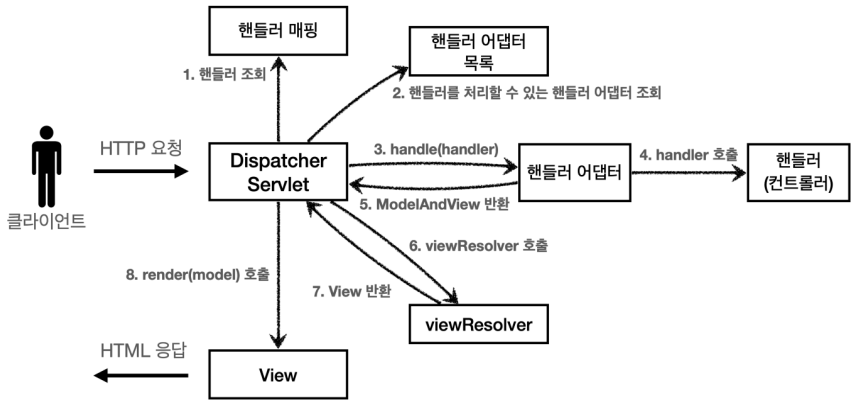
스프링 부트를 사용하는 웹 개발에서 controller 부분은 java, model은 java 객체, view 부분은 html 등으로 구현됨.

2.3.2. 스프링 mvc 구조

스프링 mvc 구조는 아래의 그림과 같음. 각 부분은 인터페이스로 정의되어 있어 확장에 용이함.

작업 순서에 따라 나열하면 아래와 같음.

0. HTTP 요청 발생
1. url에 따른 핸들러 조회
2. 핸들러에 따른 핸들러 어댑터 조회
3. 핸들러 어댑터 실행
4. 핸들러 실행
5. 핸들러 어댑터에서 실행 결과를 ModelAndView로 반환
6. 적절한 view resolver 실행
7. view resolver에서 반환한 view 객체로로 view 렌더링 수행
8. HTTP 응답 전송



핸들러는 컨트롤러의 상위 개념임. 컨트롤러는 특정 애너테이션을 사용하거나 특정 인터페이스를 구현하는 구체적인 클래스 등을 의미하지만, 핸들러는 요청을 처리할 수 있는 모든 객체를 포괄하는 개념임.

2.3.3. dispatcher servlet

1. front controller
front controller는 모든 요청을 중앙에서 처리하는 컨트롤러로, 하나의 컨트롤러만을 서블릿으로 등록해 두고 해당 컨트롤러에서 url에 대응되는 각 컨트롤러를 호출하여 작업을 수행하도록 하는 디자인 패턴에 사용됨.

url별로 각각 클래스를 만들고 @WebServlet과 HttpsServlet를 작성하여 서블릿으로 사용하는 것은 번거롭고 비효율적이기 때문에 front controller를 사용하는 것.

2. dispatcher servlet
dispathcer servlet은 front controller로서 동작하는 서블릿으로, 스프링 mvc의 핵심 코드들을 포함하고 있음.

dispatcher servlet 또한 서블릿이므로 서블릿 클래스의 코드 구성을 가짐. DispatcharServlet 클래스는 HttpServlet을 상위 클래스로 가지고 있고, DispatcharServlet의 부모 클래스에 service()가 오버라이드 되어 있음. 또한 모든 경로(urlpatterns="/")에 대해 매핑이 되어있음.*

오버라이드된 service()를 호출하면 내부적으로 여러 메소드를 거쳐 doDispatch()를 호출하는데, 이 메소드가 여러 메소드들을 다시 호출하며 핵심 기능을 함.

2.3.4. 핸들러 어댑터

1. 핸들러 어댑터
핸들러 어댑터는 핸들러를 형식에 관계없이 사용할 수 있도록 어댑터로서 기능하는 객체임.

핸들러 어댑터 인터페이스는 아래와 같이 supports() 메소드와 handle() 메소드 등으로 구성됨. supports()는 해당 핸들러 어댑터가 매개변수로 작성한 핸들러에 호환되는지를 반환하고, handle()은 적절한 컨트롤러를 호출하고 그 결과를 ModelAndView 형으로 반환함.

```
public interface HandlerAdapter {
     boolean supports(Object handler);
     ModelAndView handle(HttpServletRequest request,
            HttpServletResponse response, Object handler) throws Exception;
}
```

핸들러 어댑터 목록에서 supports()를 사용해 적절한 어댑터를 찾아 가져오고, dispatcher servlet에서 해당 핸들러의 handle()을 호출하면 handle 내부에서 핸들러를 호출하는 식으로 동작함.

2. ModelAndView 클래스
ModelAndView 클래스는 핸들러에서 작업 수행 후 응답 생성을 위해 어댑터에서 사용되는 클래스임.

ModelAndView는 내부적으로 view에 전달할 데이터를 map의 형태로, 해당되는 view의 이름을 문자열의 형태로 가지고 있음. 핸들러 어댑터는 핸들러가 전달한 데이터로 ModelAndView를 구성하여 dispatcher servlet에 전달함.

3. 핸들러로의 데이터 전달/반환
핸들러로의 데이터 전달/반환 방법은 핸들러와 그에 따른 어댑터에 따라 정의됨. 다만 비교적 편리한 방법으로는, 아래와 같이 전달/반환할 수 있음.

전달 시에는, 각 핸들러에 dispatcher servlet으로부터 매번 HttpServletRequest와 HttpServletResponse를 전달하는 것은 서블릿 종속적인 코드를 작성하게 되고 해당 정보가 필요 없는 경우가 있을 수 있으므로, 데이터만을 map 등에 담아서 전달할 수 있음.

반환 시에는, 핸들러에서 ModelAndView를 만들어 반환하도록 할 수도 있지만, 핸들러에서는 view 이름만 반환하도록 하고, 매개변수로 별도의 map을 전달하여 데이터는 해당 map에 담도록 할 수 있음. 이렇게 하면 프레임워크는 복잡해지지만 개발자 입장에서는 훨씬 단순하게 코드를 작성할 수 있게 됨.

4. 핸들러 매핑/핸들러 어댑터 실제 구현 방식
어떤 핸들러(컨트롤러)가 사용되려면 1. 핸들러 매핑에서 핸들러를 찾을 수 있어야 하고 2. 핸들러 어댑터 목록에서 해당 핸들러를 처리할 수 있는 어댑터를 찾을 수 있어야 함.

스프링은 대부분의 경우에 대한 핸들러 매핑과 핸들러 어댑터를 처리 방식을 이미 가지고 있음.

HandlerMapping는 핸들러 매핑을 수행에 대한 인터페이스로, 아래와 같은 구현체들(우선순위대로 작성함)을 가짐. 우선순위에 따라 이 구현체들로 요청에 대응되는 핸들러를 찾음.
RequestMappingHandlerMapping : 애너테이션 기반 매핑 수행.
BeanNameUrlHandlerMapping : 스프링 빈 이름 기반(요청 url과 일치하는지 검사) 매핑 수행.

HandlerAdapter는 핸들러 어댑터 조회에 대한 인터페이스로, 아래와 같은 구현체들(우선순위대로 작성함)을 가짐. 우선순위에 따라 이 구현체들로 핸들러에 대응되는 어댑터를 찾음.
RequestMappingHandlerAdapter : 애노테이션 기반의 컨트롤러인 @RequestMapping을 사용하는 핸들러 처리.
HttpRequestHandlerAdapter : HttpRequestHandler 인터페이스를 사용하는 핸들러 처리.
SimpleControllerHandlerAdapter : Controller 인터페이스를 사용하는 처리.

현재는 대부분의 경우 애너테이션 기반으로 개발함.

2.3.5. view/view resolver

1. view
View에서는 템플릿 엔진으로 HTML을 구성해 응답으로 전송함. view를 분리하면 각 핸들러에서 view에 대한 작업을 수행하는 대신, 하나의 공통된 객체가 view를 처리하도록 할 수 있음.

View는 view에 대한 인터페이스로 여러 구현체를 가지는데, view resolver에 의해 반환된 view가 사용됨.

2. view resolver
view resolver는 논리 이름을 물리 이름으로 변환하고, 렌더링을 수행하는 view 객체를 반환함. viewResolver를 사용하면 view의 물리 이름(전체 경로) 대신 논리 이름만을 전달하도록 할 수 있음.

ViewResolver는 view resolver에 대한 인터페이스로, 아래와 같은 구현체들(우선순위대로 작성함)을 가짐. 우선순위에 따라 이 구현체들로 적절한 view resolver를 찾아 수행함.
BeanNameViewResolver : 빈 이름으로 뷰를 찾아서(논리 이름과 일치하는 빈 선택) 반환함.
InternalResourceViewResolver : JSP 등을 처리할 수 있는 뷰를 반환함.

스프링 부트는 InternalResourceViewResolver를 자동으로 등록하는데, application.properties에 아래와 같이 앞뒤로 작성될 문자열을 지정할 수 있음.

```
spring.mvc.view.prefix=/WEB-INF/views/
spring.mvc.view.suffix=.jsp
```

2.4. 스프링 mvc 사용법

2.4.1. 애너테이션 기반 컨트롤러

1. 애너테이션 기반 컨트롤러
현재의 스프링은 애너테이션 기반 컨트롤러를 지원하고, 실무에서는 대부분 이 방식으로 사용함.

스프링에서는 RequestMappingHandlerMapping, RequestMappingHandlerAdapter에 의해 애너테이션 기반 컨트롤러가 매핑되고, 해당 컨트롤러를 처리하는 어댑터가 사용됨. 즉, 애너테이션을 적절히 지정하기만 하면 컨트롤러로 사용이 가능함.

2. 코드 작성 방식
아래와 같이 @RequestMapping 등의 애너테이션을 사용하여 컨트롤러로 등록할 수 있음.

```
@Controller
public class ExampleController {
    @RequestMapping("/members/new-form")
    public ModelAndView save(HttpServletRequest request,
                                        HttpServletResponse response) {
        ...
        return (new ModelAndView("save"));
    }
}
```

@RequestMapping은 메소드에 사용이 가능함. 지정한 경로가 매핑 정보로 사용되어, 해당 경로로 접근하면 매핑된 컨트롤러(메소드)가 호출됨.

RequestMappingHandlerMapping은 스프링 빈 중에서 @Controller가 클래스에 사용된 것들을 컨트롤러로 인식하여 매핑함. 즉, @Controller를 붙이면 해당 클래스의 객체가 컨포넌트 스캔에 의해 빈으로 등록되고, RequestMappingHandlerMapping에 의해 컨트롤러로 인식됨.

매개변수로 request, response 등을 작성하여 파라미터를 받고, 결과를 전달할 수 있음.

이때 컨트롤러가 애너테이션 기반으로 동작하므로, 메소드명은 임의로 지정할 수 있음.

3. 더 실용적인 방식
아래와 같이 더 편리하게 애너테이션 기반 컨트롤러 코드를 작성할 수 있음. 실제로는 이렇게 사용함.

```
@Controller
@RequestMapping("/members")
public class ExampleController {
    @GetMapping("/new-form")
    public String newForm() {
        return "new-form";
    }

    @PostMapping("/save")
    public String save(
        @RequestParam("username") String username,
        @RequestParam("age") int age,
        Model model) {
        ...
        model.addAttribute("member", member);
        return "save";
    }

    @GetMapping
    public String members(Model model) {
        ...
        model.addAttribute("member", member);
        return "members";
    }
}
```

하나의 클래스 내에서 여러 메소드에 @RequestMapping 애너테이션을 사용하여 하나의 클래스 내에서 여러 메소드를 컨트롤러로 지정할 수 있음.

@RequestMapping은 클래스에도 사용이 가능함. 클래스에 사용한 @RequestMapping에 문자열을 지정하면 내부의 모든 @RequestMapping 메소드의 경로에 대해서, 해당 문자열이 앞에 공통적으로 적용됨. 이때 @RequestMapping만을 작성하면 클래스의 문자열만이 사용됨.

ModelAndView 객체를 반환하는 대신 view 논리 이름만을 문자열로 반환해도 스프링이 적절하게 인식함.

매개변수로 request, response를 받는 대신 @RequestParam으로 키를 지정해 파라미더를 받고, 작업 결과를 model에 담아서 전달할 수 있음.

자세한 설명은 아래에서 함.

2.4.2. 요청 매핑

요청에 따른 컨트롤러의 매핑은 기본적으로 @RequestMapping 애너테이션을 사용함. 추가로 매핑에 대한 옵션을 지정할 수도 있음.

1. HTTP 메서드 지정
컨트롤러가 특정 HTTP 메서드로 온 요청만을 처리하도록 하는 것이 좋음.

메소드의 @RequestMapping에 옵션을 추가하여 처리할 수도 있고, @RequestMapping 대신 @GetMapping, @PostMapping 등을 사용하여 처리할 수도 있음.

@GetMapping은 GET으로 온 요청만 처리하고, @PostMapping은 POST로 온 요청만 처리하는 식으로 동작함. @GetMapping, @PostMapping은 내부적으로 @RequestMapping을 포함하고 있음.

2. 경로 변수 사용
아래와 같이 작성하여 URL 경로로 작성된 값을 변수에 받을 수 있고, 다중으로도 작성이 가능함.

```
@GetMapping("/mapping/users/{userId}/orders/{orderId}")
public String mappingPath(@PathVariable("userId") String userId,
                                    @PathVariable(orderId) Long orderId) {
    log.info("mappingPath userId={}, orderId={}", userId, orderId);
    return "ok";
}
```

이때 변수명과 {} 내부의 파라미터명이 같으면 애너테이션에서 문자열을 지정하지 않아도 인식되지만, 항상 문자열을 명확하게 명시해주는 것이 좋음.

3. 매핑 조건 지정
@RequestMapping에 옵션을 작성하여 특정 파라미터, HTTP 헤더, 미디어 타입 등에 따라 매핑 여부가 결정되도록 할 수 있음.

참고로 미디어 타입도 헤더로 지정되지만 별도의 작업 수행을 위해 따로 지정하도록 되어 있다고 함.

/hello와 /hello/는 서로 다른 URL 요청으로, 구분되어 매핑됨.

@RequestMapping의 매핑 경로로는 배열을 입력하여 다중 설정이 가능함. (ex. "/hello-basic", "/hello-go")

2.4.3. 요청 처리

이전에 다뤘던 것처럼 요청은 1. 쿼리 파라미터를 이용한 GET으로 전송 2. 쿼리 파라미터 형태의 POST (HTML)로 전송 3. HTTP API로 전송으로 분류할 수 있음. 이때 1번과 2번은 공통적으로 처리됨.

1. 헤더 조회
아래와 같이 컨트롤러 메서드의 파라미터를 적절히 작성하고 해당 변수를 활용하여 HTTP 헤더 정보를 조회할 수 있음.

```
@RequestMapping("/headers")
public String headers(HttpServletRequest request,
    HttpServletResponse response,
    HttpMethod httpMethod,
    Locale locale,
    @RequestHeader MultiValueMap<String, String> headerMap,
    @RequestHeader("host") String host,
    @CookieValue(value = "myCookie", required = false) String cookie
    ) {
    ...
}
```

request의 메소드를 사용하거나, @RequestHeader를 사용하는 등으로 특정 헤더를 조회할 수 있음.

MultiValueMap은 map과 유사하지만 하나의 key에 대한 값이 여러 개일 수 있는 자료구조임.

2. 쿼리 파라미터 조회 - 단순 조회
request.getParameter()로 GET/POST의 쿼리 파라미터 형식 데이터를 조회할 수 있고, 이를 요청 파라미터(Requset Parameter) 조회라고 함.

아래와 같이 @RequestParam을 사용하면 요청 파라미터 조회보다 편리하게 조회가 가능함.

```
@RequestMapping("/request-param-v2")
public String requestParamV2(
        @RequestParam("username") String memberName,
        @RequestParam("age") int memberAge) {
    ...
}
```

@RequestParam에 required 옵션을 작성하여 파라미터의 필수 여부를 지정할 수 있음. 해당 파라미터가 전달되지 않으면 400 에러를 발생시키는 것. 참고로 만약 파라미터로 value를 작성하지 않고 key만 작성한 경우 빈 문자가 전달되어 에러가 발생하지 않음.

@RequestParam에 defaultValue 옵션을 작성하여 파라미터의 default값을 지정할 수 있음. 빈 문자가 전달된 경우(key는 있는데 value가 없는 경우)에도 default값이 적용됨.

해당 값이 없으면 변수에는 null이 전달됨. 만약 변수의 자료형이 primitive인 경우 null을 저장할 수 없으므로 500 에러가 발생하므로, default값을 지정하거나 wrapper class로 바꿔야 함.

이때 변수명과 {} 내부의 파라미터명이 같으면 애너테이션에서 문자열을 지정하지 않아도 인식되고, primitive 자료형, wrapper 자료형 등의 경우 @RequestParam도 생략이 가능함. 하지만 항상 문자열을 명확하게 명시해주는 것이 좋음.

3. 쿼리 파라미터 조회 - @ModelAttribute
@ModelAttribute로 요청 파라미터 값들로 구성된 객체를 자동 생성받을 수 있음.

@ModelAttribute는 아래의 순서와 같이 동작함.
1. 해당 참조 변수에 해당하는 객체를 생성함.
2. 객체의 프로퍼티 중 요청 파라미터와 일치하는 것을 찾고, 해당 프로퍼티의 setter를 호출해서 (프로퍼티 접근법) 요청 파라미터의 값을 저장(바인딩)함.

아래와 같이 작성할 수 있음.

```
@RequestMapping("/request-param-v2")
public String requestParamV2(@ModelAttribute TestData testData) {
    ...
}
```

@ModelAttribute도 생략이 가능하지만 작성하는 것이 좋음.

아래와 같이 @ModelAttribute에 문자열을 지정하면 파라미터로 받은 해당 객체는 지정한 문자열을 이름으로 model에 자동으로 등록됨. 문자열을 지정하지 않으면 클래스명에서 첫 문자를 소문자로 바꾼 것을 이름으로 사용함.

```
@RequestMapping("/request-param-v2")
public String requestParamV2(@ModelAttribute("data") TestData testData) {
    ...
}
```

4. 요청 메시지 조회 - 텍스트
request를 받아서 조회하거나, InputStream을 받아서 조회하거나, HttpEntitiy<>/RequestEntity<>를 받아서 조회하는 등의 방법들이 있지만, 아래와 같이 @RequestBody 애너테이션으로 조회하는 방법이 가장 간단함.

아래와 같이 작성하면 messageBody에 메시지 바디에 담긴 데이터가 문자열로 저장됨.

```
@ResponseBody
@PostMapping("/request-body-string-v4")
public String requestBodyStringV4(@RequestBody String messageBody) {
    ...
}
```

5. 요청 메시지 조회 - json
json 요청 메시지를 텍스트로 받고 objectMapper 라이브러리를 사용해 객체로 변환할 수도 있지만, 아래와 같이 단순히 @RequestBody 애너테이션을 사용하면 텍스트 데이터가 해당 타입으로 자동 변환됨.

```
@ResponseBody
@PostMapping("/request-body-json")
public String requestBodyStringV4(@RequestBody TestData testData) {
    ...
}
```

참고로 HttpEntitiy<>로 받아도 자동 변환이 수행됨.

쿼리 파라미터 조회와 요청 메시지(메시지 바디) 조회는 서로 완전히 다른 매커니즘으로 동작함.

json과 객체, 객체와 json 등 사이의 자동 변환은 HTTP 메시지 컨버터라는 요소가 수행함.

2.4.4. 응답 전송

이전에 다뤘던 것처럼 응답은 1. 정적 리소스(HTML, CSS, img 등) 전송 2. 동적으로 HTML 생성 후 전송(뷰 템플릿) 3. HTTP API로 전송으로 분류할 수 있음.

1. 정적 리소스
정적 리소스는 html, CSS, Javascript, img 등의 리소스를 말함.

src/main/resources/static 등 스프링 부트가 정적 리소스로 인식할 수 있는 위치에 리소스를 넣어두고 사용함.

정적 리소스에 대해서는 별도의 컨트롤러가 필요하지 않고 단순 경로로 접근할 수 있음.

특정 url로 웹브라우저에서 접속을 시도하면 우선 대응되는 매핑(컨트롤러)이 존재하는지가 검사되고, 존재하지 않는다면 정적 컨텐츠가 사용됨.

2. 뷰 템플릿
뷰 템플릿은 템플릿 엔진이 처리할 템플릿 파일 또는 템플릿 엔진 자체를 의미함. 템플릿 엔진에 의해 HTML을 동적으로 생성하여 전송할 수 있음.

뷰 템플릿은 src/main/resources/templates에 넣어두고 사용함.

아래와 같이 컨트롤러에서 ModelAndView 객체를 반환하거나, 문자열을 반환해서(view resolver 동작) 뷰 템플릿을 지정할 수 있음.

전달할 데이터는 ModelAndView 또는 Model에 담으면 됨. model의 경우 model.addAttribute() 등을 사용할 수 있는데, 이때 데이터의 이름(문자열)과 데이터를 지정해야 함. 뷰 템플릿에서 해당 이름으로 데이터를 사용함.

```
@RequestMapping("/response-view-v1")
public ModelAndView responseViewV1() {
ModelAndView mav = new ModelAndView("response/hello")
            .addObject("data", "hello!");
    return mav;
}
@RequestMapping("/response-view-v2")
public String responseViewV2(Model model) {
    model.addAttribute("data", "hello!!");
    return "response/hello";
}
```

3. HTTP API

HTTP API를 제공하는 경우 메시지 바디에 데이터 넣어 전달함. 아래와 같이 response에 데이터를 넣거나, ResponseEntitiy<>(HttpEntity<>의 하위 타입) 객체에 데이터(문자열, 객체 등)를 넣어 반환하거나, @ResponseBody를 사용하거나, 객체를 반환하여 뷰 템플릿을 거치지 않고 데이터를 전달할 수 있음.

```
@Controller
public class ResponseBodyController {
    @ResponseBody
    @GetMapping("/response-body-string-v3")
    public String responseBodyV3() {
        return "ok";
    }
    @GetMapping("/response-body-json-v1")
    public ResponseEntity<HelloData> responseBodyJsonV1() {
        HelloData helloData = new HelloData();
        helloData.setUsername("userA");
        helloData.setAge(20);
        return new ResponseEntity<>(helloData, HttpStatus.OK);
    }
    @ResponseStatus(HttpStatus.OK)
    @ResponseBody
    @GetMapping("/response-body-json-v2")
    public HelloData responseBodyJsonV2() {
        HelloData helloData = new HelloData();
        helloData.setUsername("userA");
        helloData.setAge(20);
        return helloData;
    }
}
```

ResponseEntity<(객체)>나 객체를 반환하는 경우 HTTP 메시지 컨버터에 의해 해당 객체가 json으로 자동 변환됨.

4. @ResponseBody

@ResponseBody 애너테이션을 컨트롤러(메서드)에 작성하면 뷰 템플릿을 사용하는 대신 HTTP 메시지 컨버터가 동작하여 해당 반환값을 메시지 바디에 직접 입력함.

@ResponseBody를 클래스에 작성하면 모든 메서드에 적용되는데, 대신 @RestController를 작성할 수 있음. @RestController는 @ResponseBody와 @Controller를 합친 기능을 제공함.

객체를 반환하는 경우 상태 코드를 동적으로 지정하기 어렵기 때문에, 동적인 지정이 필요하다면 ResponseEntity<> 객체를 반환하는 방식을 사용하는 것이 좋음.

2.4.5. thymeleaf 기본 사용법

thymeleaf는 스프링에서 주로 사용하는 템플릿 엔진임.

thymeleaf를 사용하려면 html 태그에서 <html xmlns:th="http://www.thymeleaf.org">를 지정해야 함.

1. 기본 문법 형식
thymeleaf의 문법은 기본적으로 th:href, th:action와 같이 특정 html 속성 앞에 th:를 붙이는 방식임. th:가 붙은 속성은 서버에서 렌더링되어 기존의 속성을 대체하고, 정적 리소스로서 해당 html 파일에 접근하는 등 서버 렌더링이 수행되지 않은 경우 기존의 속성이 사용됨.

기존의 순수한 html을 유지하면서 뷰 템플릿으로도 사용할 수 있도록 하는 thymeleaf의 특징을 natural templates라고 함.

예를 들어, 아래와 같이 작성하여 사용함.

```
<tr th:each="item : ${items}">
    <td><a href="item.html" th:href="@{/basic/items/{itemId}
    (itemId=${item.id})}" th:text="${item.id}">회원id</a></td>

    <td><a href="item.html" th:href="@{|/basic/items/${item.id}|}"
    th:text="${item.itemName}">상품명</a></td>

    <td th:text="${item.price}">10000</td>

    <td th:text="${item.quantity}">10</td>
</tr>
```

2. 변수 표현식
thymeleaf에서 변수를 사용할 때는 À¸Ě의 꼴로 작성함. model에 포함되어 있는 데이터를 데이터의 이름으로 사용할 수 있음.

변수 표현식은 주로 속성값으로 사용함. 즉, 속성="..." 안에 작성하여 사용함.

3. 링크 표현식
thymeleaf에서 링크를 사용할 때는 @{링크}의 꼴로 작성함. 아래와 같이 괄호를 사용해 링크 표현식 내부에 변수를 넣을 수도 있음.

```
<td><a th:href="@{/basic/items/{itemId}(itemId=${item.id})}"</td>
```

4. 리터럴 대체
아래와 같이 |...|를 작성하여 리터럴, 변수, 공백 문자를 혼용하여 사용할 수 있음.

```
<p th:text="|Hello, ${name}!|"></p>
```

5. 컬렉션 처리
아래와 같이 컬렉션 데이터에 대해 th:each를 사용하면 for each문처럼 각 요소를 반복 처리할 수 있음. 해당 블럭 내부에서 지정한 이름으로 각 요소를 사용할 수 있는 것.

```
<ul>
    <li th:each="product : ${products}">
        <span th:text="${product.name}">Product Name</span>
    </li>
</ul>
```

6. 조건
아래와 같이 th:if를 사용하여 해당 조건이 참인 경우에만 지정된 태그가 렌더링되도록 할 수 있음. 해당 조건이 거짓이면 태그가 아예 사라짐.

```
    <h2 th:if="${param.status}" th:text="'저장 완료!'"></h2>
```

7. param 변수
쿼리 파라미터는 빈번하게 사용되기 때문에, 컨트롤러에서 model에 쿼리 파라미터를 담지 않아도, 쿼리 파라미터는 param이라는 변수에 자동으로 담겨 뷰 템플릿에서 사용이 가능함.

즉, model에 담지 않아도 아래와 같이 사용이 가능함.

```
    <h2 th:if="${param.status}" th:text="'저장 완료!'"></h2>
```

2.4.6. 리다이렉트

응답 메시지에 상태 코드와 Location 헤더를 직접 지정하여 리다이렉션할 수도 있지만, 스프링이 제공하는 별도의 문법을 사용하면 더 편리하게 리다이렉션할 수 있음.

1. 문자열에서 리다이렉트 지정
아래와 같이 컨트롤러가 문자열 앞에 redirect:를 붙여서 url을 작성하면 해당 url로 리다이렉트하도록 지정할 수 있음. 주의할 점은 view 이름이 아니라 url로 리다이렉트 하는 것임.

```
    @PostMapping("/{itemId}/edit")
    public String edit(@PathVariable Long itemId) {
        ...
        return "redirect:/basic/items";
    }
```

2. RedirectAttributes
아래와 같이 컨트롤러의 파라미터로 RedirectAttributes 객체를 받아서 리다이렉트 속성을 지정할 수 있음.

속성은 속성 이름(문자열)과 값을 작성하여 지정함. view 논리 이름에 속성 이름을 작성하여 동적으로 이름을 구성할 수 있음. 이름 구성에 사용되지 않은 속성은 리다이렉트(GET) 시에 쿼리 파라미터로 사용됨.

```
    @PostMapping("/add")
    public String addItemV6(Item item, RedirectAttributes redirectAttributes) {
        Item savedItem = itemRepository.save(item);
        redirectAttributes.addAttribute("itemId", savedItem.getId());
        redirectAttributes.addAttribute("status", true);
        return "redirect:/basic/items/{itemId}";  //status 속성은 ?status=true로 전
달됨
    }
```

2.5. HTTP 메시지 컨버터

2.5.1. HTTP 메시지 컨버터

HTTP 메시지 컨버터는 메시지 바디에 데이터를 직접 입력하는 경우에 대해 다른 데이터 타입으로 자동 변환해주는 요소임. HTTP 메시지 컨버터를 활용하면 request/response 등에 직접 데이터를 입력하는

코드를 작성할 필요가 없음.

1. HttpMessageConverter<T> 인터페이스
HTTP 메시지 컨버터는 아래와 같이 HttpMessageConverter<T>라는 인터페이스를 가지고 있음.

```
public interface HttpMessageConverter<T> {
    boolean canRead(Class<?> clazz, @Nullable MediaType mediaType);
    boolean canWrite(Class<?> clazz, @Nullable MediaType mediaType);
    List<MediaType> getSupportedMediaTypes();

    T read(Class<? extends T> clazz, HttpInputMessage inputMessage)
                        throws IOException, HttpMessageNotReadableException;
    void write(T t, @Nullable MediaType contentType,
                                        HttpOutputMessage outputMessage)
                        throws IOException, HttpMessageNotWritableException;
}
```

canRead()/canWrite()로 현재의 요청/응답에 사용될 수 있는지를 판단하고, read()/write()로 요청/응답을 읽거나 작성함. 핸들러 어댑터와 유사한 방식으로 구성되어 있음.

2. HttpMessageConverter<T>의 구현체
HttpMessageConverter<T>에 대한 구현체로는 우선순위에 따라 아래와 같은 것들이 있음.

ByteArrayHttpMessageConverter : byte[] 타입을 처리하고, 모든 미디어 타입에 호환됨. 응답 시에는 application/octet-stream을 미디어 타입으로 작성함.

StringHttpMessageConverter : String 타입을 처리하고, 모든 미디어 타입에 호환됨. 응답 시에는 text/plain을 미디어 타입으로 작성함.

MappingJackson2HttpMessageConverter : 객체/HashMap을 처리하고, application/json 등에 호환됨. 응답 시에는 application/json 등을 미디어 타입으로 작성함.

3. 동작 과정
스프링 mvc는 요청의 경우 @RequestBody, HttpEntity(RequestEntity)에, 응답의 경우 @ResponseBody, HttpEntity(ResponseEntity)에 HTTP 메시지 컨버터를 적용함. 즉, request/response 등을 사용하여 직접 요청/응답을 처리하지 않는 경우에 대해서 동작함.

요청 시에 HTTP 메시지 컨버터는 아래의 과정을 거쳐 동작함.
1. 컨트롤러에서 @RequestBody 또는 HttpEntity를 사용함.
2. 우선순위에 따라 구현체의 canRead()를 호출함. canRead()에서는 해당 구현체에 대해 클래스 타입이 맞는지 검사하고, 요청의 Content-Type이 호환되는지 검사함.
3. canRead()가 참이면 read()를 호출함.

응답 시에 HTTP 메시지 컨버터는 아래의 과정을 거쳐 동작함.
1. 컨트롤러에서 @ResponseBody 또는 HttpEntity를 사용함.
2. 우선순위에 따라 구현체의 canwrite()을 호출함. canwrite()에서는 해당 구현체에 대해 클래스 타입이 맞는지 검사하고, 요청의 Accept이 호환되는지 검사함.
3. canwrite()가 참이면 write()을 호출함.

2.5.2. HTTP 메시지 컨버터의 사용 위치

HTTP 메시지 컨버터는 애너테이션 기반 컨트롤러에 @RequestBody/@ResponseBody 또는 HttpEntity를 사용한 경우에 동작함. 즉, 애너테이션 기반인 RequestMappingHandlerAdapter에 의해 사용됨.

1. RequestMappingHandlerAdapter 동작 방식
RequestMappingHandlerAdapter는 내부적으로 argument resolver를 호출하여 매개변수에 들어갈 적절한 인자 값을 반환받고, 이 값으로 컨트롤러를 호출함. 해당 컨트롤러의 반환값은 return value handler에 의해 변환되어 RequestMappingHandlerAdapter에 전달됨.

argument resolver와 return value handler가 특정 상황에 내부적으로 HTTP 메시지 컨버터를 사용하여 데이터를 변환함.

2. argument resolver
argument resolver는 적절한 파라미터 값(인자에 들어갈 값)을 생성해서 반환하는 요소로, HandlerMethodArgumentResolver라는 이름의 인터페이스로 정의되어 있음. 스프링은 여러 argument resolver를 기본으로 제공함.

@RequestBody 파라미터를 처리하는 argument resolver와, HttpEntity 파라미터를 처리하는 argument resolver에서 HTTP 메시지 컨버터를 사용함.

3. return value handler
return value handler는 컨트롤러의 반환값을 적절히 변환하여 반환하는 요소로, HandlerMethodReturnValueHandler라는 이름의 인터페이스로 정의되어 있음. 스프링은 여러 return value handler를 기본으로 제공함.

@ResponseBody 반환값을 처리하는 return value handler와, HttpEntity 반환값을 처리하는 return value handler에서 HTTP 메시지 컨버터를 사용함.

인터페이스로 정의되어 있으므로 필요한 경우 구현체는 직접 작성할 수도 있음.