Test Driven Development

TDD

- TDD
  - ✓ 소프트웨어 개발 방법론 중 하나
  - ✓ 기존에 테스트 코드의 작성은 프로덕션 코드가 작성된 이후에 이루어졌지만 TDD를 적용하면 프로덕션 코드 보다 실패하는 테스트 코드를 먼저 작성하고 코드를 테스트를 통과하기 위해 최소한으로 개선한 이후 테스트를 통과한 프로덕션 코드를 리팩토링
  - ✓ TDD 는 테스트 만을 위한 기술이 아니며 오히려 소프트웨어 설계 방법론에 가까움
  - ✓ TDD의 아이러니 중 하나는 테스트 기술이 아니라는 점인데 TDD는 분석 기술이며 설계 기술 - 켄트 벡

TDD

- TDD Cycle
  - ✓ RED → GREEN → REFACTOR
  - ✓ RED: 테스트 코드 작성
    - 동작하는 프로덕션 코드가 없는 상황에서 테스트 코드를 먼저 작성하는 것으로 앞으로 작성될 프로덕션 코드가 어떤 동작을 하면 좋을지 작성
    - 요구 사항을 작성하는 것 과 유사
    - 테스트 코드를 먼저 작성하기 때문에 테스트 코드가 기대하는 로직이 실행되지 않아 테스트는 실패할 것
    - 핵심은 실패하는 테스트 코드를 작성하는 하는 것 즉 RED 를 띄우는 것

```
@Test
void plusTest() {
    // given
    Calculator calculator = new Calculator();

    // when
    int result = calculator.plus(1, 3);

    // then
    assertThat(result).isEqualTo(4);
}
```

TDD

- TDD Cycle
  - ✓ GREEN: 테스트를 통과하는 최소 코드 작성
    - 실패하는 테스트 코드를 통과하도록 만드는 최소한의 코드를 작성
    - 지금 필요한 것은 가장 빠르게 GREEN 을 보는 것
    - 빠르게 구현할 수 있는 깔끔한 코드가 있다면 바로 작성하면 되지만 깔끔하게 코드를 작성하는 일이 오래 (몇 분 정도) 걸릴 것 같다면 일단 테스트 코드를 통과하는 데에만 집중하여 코드를 작성하는데 켄트벡은 이를 죄악이라고 표현하며 GREEN을 보기 위해서는 어떤 죄악을 저질러도 상관없다고 함
    - GREEN 을 보기 위해서는 명백한 실제 구현을 입력하는 방법도 있지만 최대한 빨리 GREEN 을 보기 위해서는 상수를 반환하는 코드를 작성하고 점진적으로 변수로 바꾸어가는 방법도 있음

```
public class Calculator {
    public int plus(int a, int b) {
        return 4;
    }
}
```

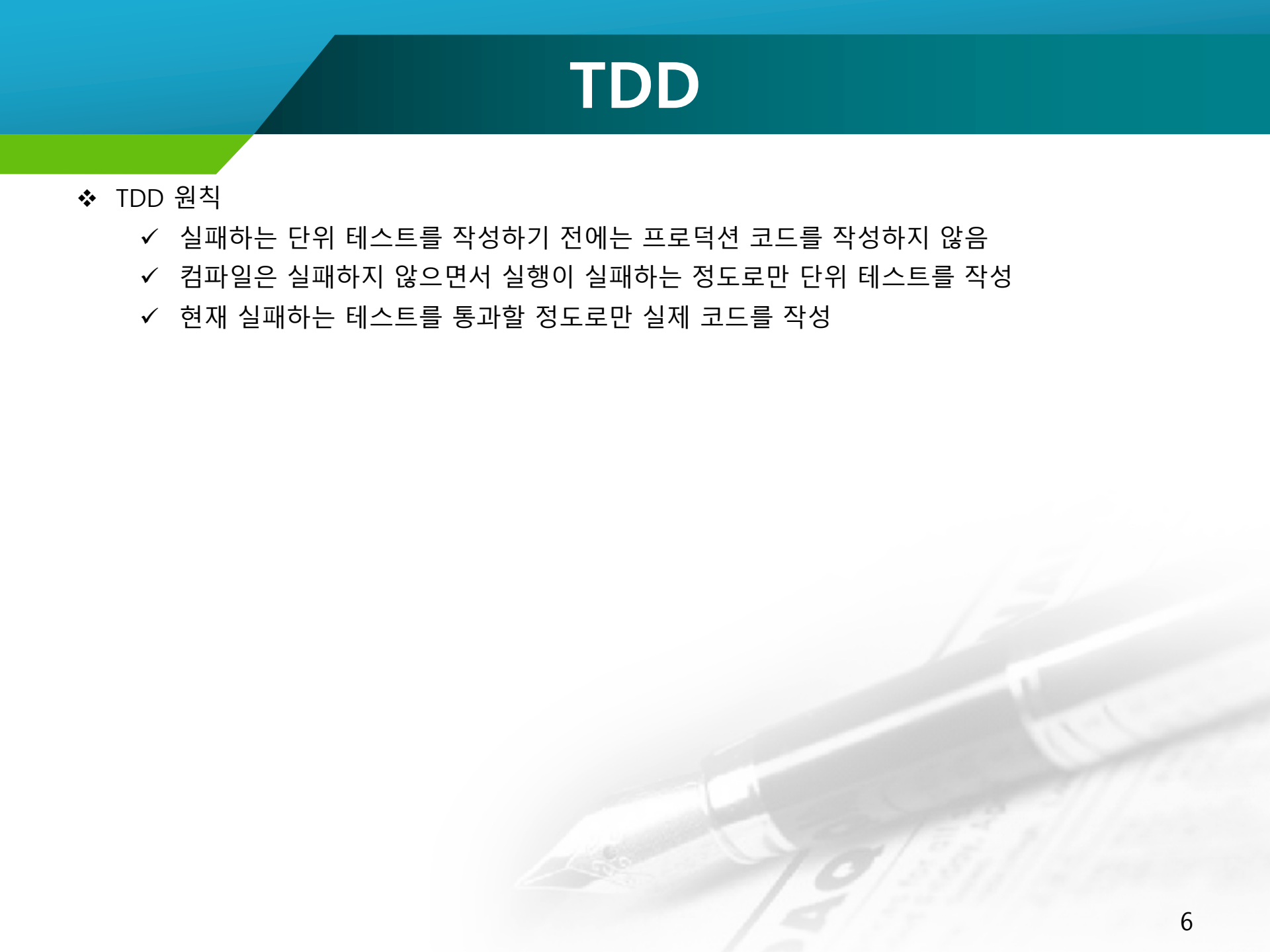
TDD

- TDD Cycle
  - ✓ REFACTOR
    - GREEN 을 통과하기 위해 저지른 죄악(더러운 코드) 을 수습
    - 빠르게 GREEN 을 보기 위해 작성한 더러운 코드를 좋은 코드로 리팩토링

```
public class Calculator {
    public int plus(int a, int b) {
        return a + b;
    }
}
```

TDD

- TDD 원칙
  - ✓ 실패하는 단위 테스트를 작성하기 전에는 프로덕션 코드를 작성하지 않음
  - ✓ 컴파일은 실패하지 않으면서 실행이 실패하는 정도로만 단위 테스트를 작성
  - ✓ 현재 실패하는 테스트를 통과할 정도로만 실제 코드를 작성



TDD

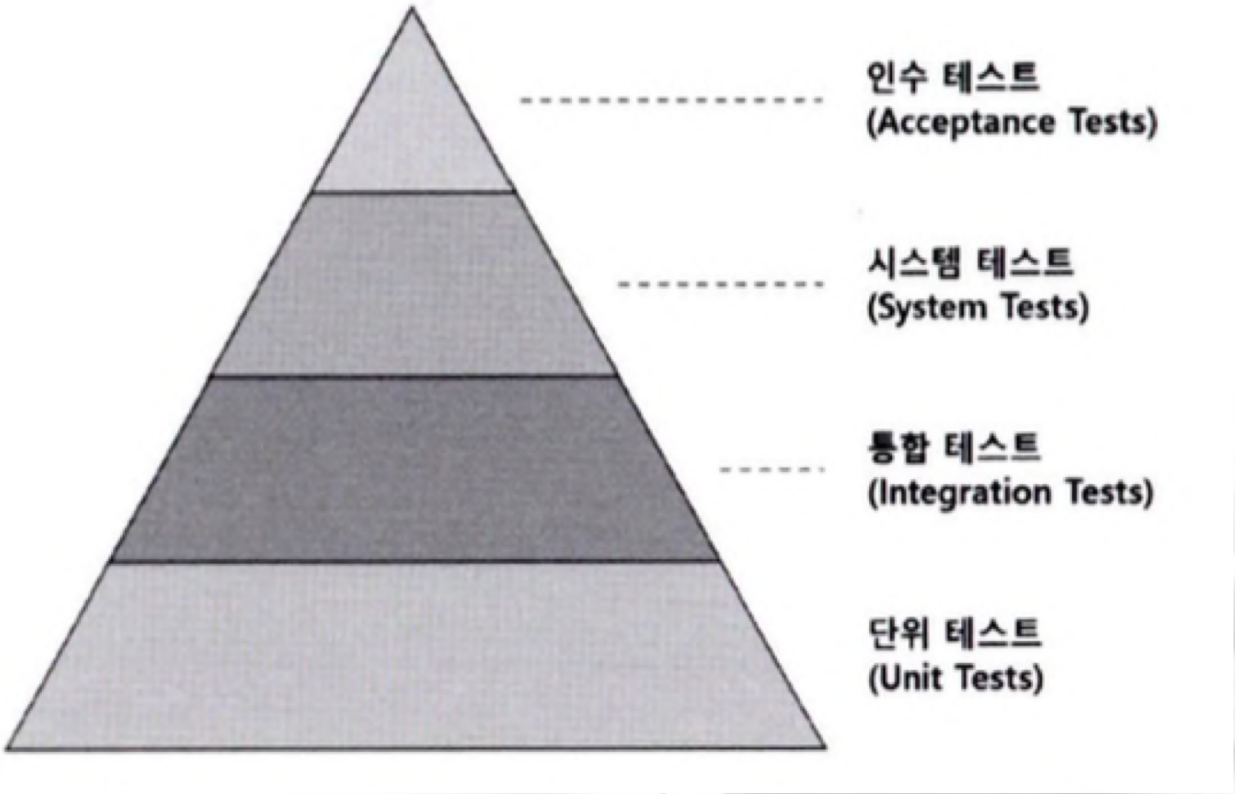
- TDD 필요
  - ✓ 변화에 대한 불안감 해소
    - 프로덕션 코드에 대한 테스트 코드가 이미 작성이 되어 있으므로 변화에 대한 영향력을 빠르게 파악할 수 있음
    - 변경에 의한 문제를 사전에 빠르게 파악할 수 있음
    - 코드가 예상한 대로 동작한다는 자신감을 얻을 수 있음
    - TDD의 핵심은 불안감 해소와 용기
    - 켄트 벡은 TDD를 불안함을 지루함으로 바꾸는 마법의 돌 이라고 표현
  - ✓ 한번에 하나의 일만 집중할 수 있음
    - TDD를 적용하지 않은 상황에서는 코드를 작성하다가 다른 코드에 한눈이 팔리고 다른 모듈을 건드리게 되다 보면 내가 뭘 하고 있었더라 라는 생각이 들기 쉬움
    - TDD를 적용하면 프로덕션 코드를 작성하며 현재 실패한 테스트 케이스를 통과하는 데에만 집중할 수 있음
    - 야크털 깎기(Yak Shaving)를 하지 않을 수 있음
  - ✓ 빠른 피드백
    - TDD 를 적용하면 작성한 테스트 코드로부터 빠른 주기로 피드백 받고 리팩토링 하며 점진적으로 지속적으로 코드를 개선할 수 있기 때문에 버그를 찾는 시점이 빨라질 수 있음
    - 처음부터 완벽한 설계를 하는 것에 집중하기 보다는 설계를 점진적으로 개선해 나가는 데 집중할 수 있음

TDD

- TDD 필요
  - ✓ 테스트 코드 자체가 문서가 될 수 있음
    - TDD 를 적용하며 작성한 테스트 코드는 다른 관점에서 보면 프로덕션 코드에 대한 요구사항 이기 때문에 테스트 코드 자체가 문서가 될 수 있음을 알려줌
  - ✓ 테스트를 나중에 작성하는 것은 귀찮음
    - 프로덕션 코드를 작성한 상태에서 테스트 케이스를 작성하는 것은 귀찮기 때문에 프로덕션 코드를 작성한 다음 테스트 케이스를 작성하게 되면 프로덕션 코드에 맞춰 통과하는 테스트 케이스를 만들게 될 수 있는데 이렇게 되면 테스트 코드의 의미가 무색해질수도 있음
  - ✓ 코드 퀄리티
    - TDD를 기반으로 테스트 코드를 작성하면 의존성이 높은 테스트 코드를 작성하기 어려우므로 모듈 간 결합도를 낮추고 응집도를 높일 수 있음

TDD

❖ 종류


인수 테스트
(Acceptance Tests)
시스템 테스트
(System Tests)
통합 테스트
(Integration Tests)
단위 테스트
(Unit Tests)

TDD

- ❖ 종류
  - ✓ Unit Test
    - 테스트 대상의 범위를 기준으로 가장 작은 단위의 테스트 방식
    - 일반적으로 메서드 단위로 테스트를 수행하게 되며 메서드 호출을 통해 의도한 결과가 나오는지 확인하는 수준으로 테스트를 진행
    - 테스트 비용이 적게 들기 때문에 테스트 피드백을 빠르게 받을 수 있음
  - ✓ Integration Test
    - 통합 테스트는 모듈을 통합하는 과정에서의 호환성 등을 포함해 애플리케이션이 정상적으로 동작하는지 확인하기 위해 수행하는 테스트
    - 단위 테스트는 모듈을 독립적으로 테스트하는 반면 통합 테스트는 여러 모듈을 함께 테스트해서 정상적인 로직 수행이 가능한지를 확인
    - 단위 테스트는 일반적으로 특정 모듈에 대한 테스트만 진행하기 때문에 데이터베이스나 네트워크 같은 외부 요인들을 제외하고 진행하는 데 비해 통합 테스트는 외부 요인들을 포함하고 테스트를 진행하므로 애플리케이션이 온전히 동작하는지를 테스트
    - 테스트를 수행할 때마다 모든 컴포넌트가 동작해야 하기 때문에 테스트 비용이 커지는 단점이 있음

TDD

- ❖ 종류
  - ✓ Regression Test
    - 회귀 테스트란 시스템을 운영, 유지 보수하면서 발생한 버그들을 테스트 케이스로 만들어서 이전에 발생한 버그들이 다시 발생하지 않도록 테스트하는 것
    - 버그가 발생한 입력 조건 값을 사용하여 정상 동작하는 테스트 케이스를 작성

TDD

- ❖ 종류
  - ✓ System Test
    - 시스템 테스트는 정보시스템이 완전히 통합되어 구축된 상태에서 정보 시스템의 기능을 총체적으로 테스트하는 것
    - 통합된 각 모듈들이 원래 계획했던 대로 작동하는지 시스템의 실제 동작과 원래 의도했던 요구 사항과는 차이가 없는지 등을 판단
    - 수행 시간, 파일 저장 및 처리 능력, 최대 부하, 복구 및 재시동 능력, 수작업 절차 등을 점검
    - 시스템 테스트는 시스템의 내부적인 구현 방식이나 설계에 대한 지식에 관계없이 테스트를 수행하는 블랙박스 테스트의 일종으로 분류
    - 시스템 테스트는 모든 통합 테스트를 통과한 통합된 소프트웨어 컴포넌트들 그리고 필요한 하드웨어들과 통합된 소프트웨어 시스템 전체를 대상으로 수행
    - 통합 테스트는 서로 통합된 두 소프트웨어 혹은 하드웨어 단위 사이에 서로 잘 맞지 않는 부분이 있는지를 파악하기 위해서 수행하고 시스템 테스트는 통합된 컴포넌트들 사이에 발생하는 결함과 통합된 전체 시스템내에서 발생하는 결함 모두를 발견하기 위해서 수행

TDD

- ❖ 종류
  - ✓ Acceptance Test
    - 프로젝트에 참여하는 사람들(기획자, 클라이언트 대표, 개발자 등)이 토의해서 시나리오를 만들고 개발자는 이에 의거해서 코드를 작성
    - 개발자가 직접 시나리오를 제작할 수도 있지만 다른 의사 소통 집단으로부터 시나리오를 받아(인수) 개발한다는 의미
    - 인수 테스트는 애자일 개발 방법론에서 파생했는데 익스트림 프로그래밍(XP)에서 사용하는 용어로 시나리오가 정상적으로 동작하는지를 테스트하기 때문에 통합 테스트 와는 분류가 다름
    - 시나리오에서 요구하는 것은 누가, 어떤 목적으로, 무엇을 하는가 인데 개발을 하다 보면 이런 기능은 API를 통해 드러나며 인수 테스트는 주로 이 API를 확인하는 방식으로 이루어짐
    - 인수 테스트는 소프트웨어 인수를 목적으로 하는 테스트이기 때문에 소프트웨어를 인수하기 전에 명세한 요구사항(인수 조건)대로 잘 작동하는지 검증이 필요
    - 소프트웨어를 인수할 때, 소프트웨어 내부 구조나 구현 방법을 고려하기보다는 실제 사용자 관점에서 테스트하는 경우가 많기 때문에 인수 테스트는 소프트웨어 내부 코드에 관심을 가지지 않는 블랙박스 테스트
    - Java에서는 RestAssured, MockMvc 같은 도구를 활용하여 인수 테스트를 작성할 수 있음

TDD

- ❖ 테스트 코드 작성 방법
  - ✓ Given-When-Then 패턴
    - 단계를 설정해서 각 단계의 목적에 맞게 코드를 작성
    - 단계
      - ◆ Given: 테스트를 수행하기 전에 테스트에 필요한 환경을 설정하는 단계로 테스트에 필요한 변수를 정의하거나 Mock 객체를 통해 특정 상황에 대한 행동을 정의
      - ◆ When: 테스트의 목적을 보여주는 단계로 실제 테스트 코드가 포함되며 테스트를 통한 결과를 가져오게 됨
      - ◆ Then: 테스트의 결과를 검증하는 단계로 When 단계에서 나온 결과를 검증하는 작업을 수행하며 결과가 아니더라도 테스트를 통해 나온 결과에서 검증해야 하는 부분이 있다면 이 단계에 포함
    - Given-When-Then 패턴은 테스트 주도 개발에서 파생된 BDD(Behavior-Driven-Devlopment - 행위 주도 개발)를 통해 탄생한 테스트 접근 방식
    - 단위 테스트보다는 비교적 많은 환경을 포함해서 테스트하는 인수 테스트에서 사용하는 것에 적합하다고 알려져 있음

TDD

- ❖ 테스트 코드 작성 방법
  - ✓ Given-When-Then 패턴
    - ● 테스트 메서드

```
@Test
public void testHashSetContainsNonDuplicatedValue() {

        // Given
        Integer value = Integer.valueOf(1);
        Set<Integer> set = new HashSet<>();

        // When
        set.add(value);
        set.add(value);
        set.add(value);

        // Then
        Assertions.assertEquals(1, set.size());
        Assertions.assertTrue(set.contains(value));
}
```

TDD

- ❖ 테스트 코드 작성 방법
  - ✓ 좋은 테스트를 위한 5가지 속성 - FIRST
    - 빠르게(Fast)
      - ◆ 테스트 케이스는 빠르게(fast) 동작해야 함
      - ◆ 실행 시간이 오래 걸리는 테스트 케이스는 성공 여부를 빠르게 확인할 수 없어 개발 시간에 영향을 미침
      - ◆ 테스트 과정을 자동화한 경우에는 모든 테스트 케이스를 실행하여 테스트 결과를 확인하고 버그를 수정하는 과정을 반복해야 하는데 테스트 케이스가 느리다면 이 과정 자체가 느려지고 개발 속도에 영향을 미침
      - ◆ 패키지 빌드 과정에 테스트 단계를 포함한다면 패키지 빌드 시간은 테스트 케이스 실행 시간과 비례하게 됨
    - 고립된(Isolated), 독립적(Elated)
      - ◆ 하나의 테스트 코드는 목적으로 여기는 하나의 대상에 대해서만 수행돼야 하는데 하나의 테스트가 다른 테스트 코드와 상호 작용하거나 관리할 수 없는 외부 소스를 사용하게 되면 외부 요인으로 인해 테스트가 수행되지 않을 수 있음
      - ◆ 테스트 케이스들을 실행하는 순서에 따라 결과가 달라질 수 있으며 테스트 케이스를 수정하면 다른 테스트 케이스의 결과 값에 영향을 줄 수 있기 때문
      - ◆ 테스트 케이스를 작성할 때는 전역 변수를 선언하여 여러 테스트 케이스가 하나의 데이터를 수정 및 참조하는 경우에 유의
      - ◆ 상태(status) 값을 갖는 데이터를 테스트할 때도 주의

TDD

- ❖ 테스트 코드 작성 방법
  - ✓ 좋은 테스트를 위한 5가지 속성 - FIRST
    - 반복 가능한(Repeatable): 테스트는 어떤 환경에서도 반복 가능하도록 작성해야 하는데 이 의미는 앞의 Isolated 규칙과 비슷한 의미를 갖고 있는데 테스트는 개발 환경의 변화나 네트워크의 연결 여부와 상관없이 수행돼야 함
    - 자가 검증(Self-Validating)
      - ◆ 테스트는 그 자체만으로도 테스트의 검증이 완료돼야 하는데 테스트가 성공했는지 실패했는지 확인할 수 있는 코드를 함께 작성해서 결과 와 기대치를 비교하는 작업을 코드가 아니라 개발자가 직접하고 있다면 좋지 못한 테스트 코드
      - ◆ 사람이 테스트 결과 값을 확인하고 기대하는 값과 결과 값을 비교해야 한다면 테스트 과정을 자동화
    - 적시에(Timely): 테스트 코드는 테스트하려는 애플리케이션 코드를 구현하기 전에 완성돼야 하는데 너무 늦게 작성된 테스트 코드는 정상적인 역할을 수행하기 어려울 수 있으며 테스트 코드로 인해 발견된 문제를 해결하기 위해 소모되는 개발 비용도 커지기 쉬운데 다만 이 개념은 테스트 주도 개발의 원칙을 따르는 테스트 작성 규칙으로 테스트 주도 개발을 기반으로 애플리케이션을 개발하는 것이 아니라면 이 규칙은 제외하고 진행하기도 함

TDD

- 테스트 코드 작성
  - ✓ MSA 환경에서 테스트 케이스는 더욱 유용하게 사용될 수 있음
  - ✓ MSA는 여러 작은 컴포넌트가 서로 연동하여 서비스를 제공하는 특징이 있기 때문에 서비스는 분산된 작은 기능의 조합
  - ✓ 작은 기능 중 하나에만 버그가 있더라도 서비스는 동작하지 않기 때문에 각 기능은 각각의 컴포넌트에서 제공
  - ✓ 서비스를 테스트하려면 관련된 모든 컴포넌트를 배포하고 QA 과정에서 테스트하므로 테스트 및 배포 과정은 빠르게 실행하고 바로 결과를 얻기가 힘듬
  - ✓ 컴포넌트들은 각 기능들에 테스트 케이스를 작성하고 자동화된 테스트 프로세스를 구축하여 신뢰성 있는 애플리케이션을 개발
  - ✓ CI/CD 즉 지속적으로 애플리케이션을 배포하기 위해서도 테스트 과정이 반드시 필요한데 CI/CD 과정에서 자동화된 테스트 과정도 필요하지만 높은 테스트 커버리지 또한 필요
  - ✓ 테스트 커버리지는 소프트웨어 기능을 얼마나 충분히 테스트하는지 나타낸 것이고 실제로 많이 사용하는 기능 혹은 클래스는 다양한 형태의 테스트를 추가 및 확장하는 것이 중요
  - ✓ 이제는 테스트 케이스도 애플리케이션을 개발하는 주요 기능이라고 볼 수 있음

Spring Boot Test

- Spring Boot 테스트 설정
  - ✓ 별도의 starter를 제공: spring-boot-starter-test
  - ✓ 제공되는 라이브러리
    - Junit
      - 테스트 케이스를 작성하고 실행할 수 있는 기능들을 제공
      - 테스트 케이스를 정의 할 수 있는 Annotation 과 실행한 테스트 결과 값을 예상 값과 비교 및 검증할 수 있는 클래스들을 제공
    - Hamcrest
      - 테스트 케이스의 결과 값을 검증하는 클래스와 메서드를 제공
      - Junit에서도 검증 메서드들을 제공하지만 Hamcrest에서 제공하는 메서드는 서술적으로 작성할 수 있어 가독성을 높임
    - Mockito
      - 목 객체는 개발자가 입력 값 에 따라 출력 값을 프로그래밍한 일종의 가짜 객체
      - 테스트 대상 클래스가 의존하는 객체를 목 객체로 바꿀 수 있는 기능과 목 객체를 만들 수 있는 기능을 제공

Spring Boot Test

- Spring Boot 테스트 설정
  - ✓ 제공되는 라이브러리
    - spring-test: 스프링 프레임워크의 기능을 통합 테스트할 수 있는 기능을 제공하는데 스프링 웹 MVC 같은 기능을 테스트
    - spring-boot-test: 스프링 부트 프레임워크의 기능을 통합 테스트할 수 있는 기능을 제공
    - spring-boot-test-autoconfiguration: 스프링 부트 프레임워크를 테스트하려고 자동 설정 기능을 제공

Spring Boot Test

- JUnit
  - ✓ JUnit은 자바 언어에서 사용되는 대표적인 테스트 프레임워크로서 단위 테스트 뿐 만 아니라 통합 테스트를 할 수 있는 기능도 제공
  - ✓ JUnit의 가장 큰 특징은 Annotation 기반의 테스트 방식을 지원한다는 것으로 JUnit을 사용하면 몇 개의 Annotation 만으로 간편하게 테스트 코드를 작성할 수 있음
  - ✓ JUnit을 활용하면 단정문(assert)을 통해 테스트 케이스의 기댓값이 정상적으로 도출됐는지 검토할 수 있다는 장점이 있음
  - ✓ @Test: 테스트 코드를 포함한 메서드를 정의할 때 사용하는 Annotation
    - 메서드의 접근 제어자는 Public
    - 메서드의 리턴 타입은 void
    - 메서드의 이름은 test로 시작하는 것이 일반적
  - ✓ JUnit Life Cycle
    - @BeforeAll: 테스트를 시작하기 전에 호출돠는 메서드를 정의
    - @BeforeEach: 각 테스트 메서드가 실행되기 전에 동작하는 메서드를 정의
    - @AfterAll: 테스트를 종료하면서 호출되는 메서드를 정의
    - @AfterEach: 각 테스트 메서드가 종료되면서 호출되는 메서드를 정의

Spring Boot Test

- JUnit
  - ✓ JUnit을 이용한 테스트 클래스 생성 및 수행

```
public class MiscTest {

    @BeforeAll
    public static void setup() {
        System.out.println("before all tests in the current test class");
    }

    @AfterAll
    public static void destory() {
        System.out.println("after all tests in the current test class");
    }

    @BeforeEach
    public void init() {
        System.out.println("before each @Test");
    }
```

Spring Boot Test

- JUnit
  - ✓ JUnit을 이용한 테스트 클래스 생성 및 수행

```
@Test
public void testHashSetContainsNonDuplicatedValue() {
    //Given
    Integer value = Integer.valueOf(1);
    Set<Integer> set = new HashSet<>();

    // When
    set.add(value);
    set.add(value);
    set.add(value);

    // Then
    Assertions.assertEquals(1, set.size());
    Assertions.assertTrue(set.contains(value));
}

@Test
public void testDummy() {
    Assertions.assertTrue(Boolean.TRUE);
}
```

Spring Boot Test

- JUnit
  - ✓ JUnit을 이용한 테스트 클래스 생성 및 수행

```
    @AfterEach
    public void cleanup() {
        System.out.println("after each @Test");
    }
}
```

Spring Boot Test

- JUnit
  - ✓ JUnit을 이용한 테스트 클래스 생성 및 수행
    - 일반적으로 테스트 메서드는 given-when-then 패턴을 사용하여 세 파트로 나누어 개발
    - given 파트에서는 테스트 케이스를 준비하는 과정을 포함하기 때문에 HashSet 객체를 초기화 하고 add() 메서드의 인자로 사용할 value 객체를 초기화
    - when 파트는 테스트를 실행하는 과정이므로 같은 인자 값 value를 add() 메서드에 넣어서 여러 번 호출
    - then 파트에서는 테스트를 검증하는 과정을 포함하는데 예제의 테스트 메서드처럼 반드시 세 파트 (Given. When. Then)로 나누어 테스트 케이스를 작성할 필요는 없지만 일관성있는 테스트 코드가 유지 보수에 유리하므로 많은 개발자가 이 방법을 사용
    - 테스트 클래스를 작성할 때 테스트 케이스마다 준비 과정이 반복된다면 라이프 사이클 Annotation을 고려해 볼 만한데 보통 테스트 대상 범위가 너무 커서 테스트 대상 범위를 줄이기 위해 mocking하는 작업이나 테스트 데이터를 초기화하고 종료하는 작업이 필요할 때 많이 사용

Spring Boot Test

- JUnit
  - ✓ JUnit을 이용한 테스트 클래스 생성 및 수행
    - 실행 순서
      - @BeforeAll setup()
      - @BeforeEach init()
      - @Test testHashSetContainsNonDuplicatedValue()
      - @AfterEach cleanup()
      - @BeforeEach init()
      - @Test testDummy()
      - @AfterEach cleanup()
      - @AfterAll destroy()

Spring Boot Test

- ❖ JUnit
  - ✓ JUnit을 이용한 테스트 클래스 생성 및 수행
    - ● Assertions 클래스
      - ◆ static 검증 메서드를 제공
      - ◆ 검증 메서드 이름은 assert 머리말로 시작
      - ◆ 실패하는 경우 AssertionFailedError 예외가 발생하며 Junit 프레임워크는 이 정보를 바탕으로 각 테스트 케이스의 성공/실패 여부를 종합
      - ◆ 검증 메서드의 인자 이름이 expect이면 예상 값을 의미하며 actual은 테스트 대상의 실제 값을 의미하는데 예를 들어 assertEquals() 메서드는 expect와 actual 인자를 받는데 MiscTest.java 예제에서 set.getSize()의 예상 값은 1이므로 예상 값 1을 expect 인자의 인수로 입력하고 set.getSize()의 리턴 값은 실제 응답 값이므로 actual 인자의 인수로 사용하면 됨
      - ◆ 예상 값과 실제 값을 둘 다 인자로 받는 메서드는 순서가 바뀌어도 테스트를 진행할 수 있지만 예상 값 과 실제 값이 달라 에러가 발생하면 정확한 테스트 결과 메시지를 받을 수 없음

Spring Boot Test

- ❖ JUnit
  - ✓ JUnit을 이용한 테스트 클래스 생성 및 수행
    - ● Assertions 클래스
      - ◆ Assertions 클래스에서 제공하는 검증 메서드
        - ▪ assertNull(Object actual): 실제 값이 Null인지 검증
        - ▪ assertNotNull(Object actual): 실제 값이 Not null인지 검증
        - ▪ assertTrue(boolean condition): 조건이 참인지 검증
        - ▪ assertFalse(boolean condition): 조건이 거짓인지 검증
        - ▪ assertEquals(Object expect, Object actual): 예상 값과 실제 값이 같은지 비교하는데 두 값을 비교할 때는 equals() 메서드를 사용
        - ▪ assertNotEquals(Object expect. Object actual): 예상 값과 실제 값이 다른지 비교하는데 assertEquals와 마찬가지로 equals() 메서드를 기반으로 비교
        - ▪ assertSame(Object expect. Object actual): assertEquals처럼 두 인자 값을 비교하되 == 연산자를 사용
        - ▪ assertNotSame(Object expect, Object actual): assertNotEquals처럼 두 인자가 다른지 비교하되 == 연산자를 사용
      - ◆ 검증 메서드는 인자 타입에 따라 수많은 메서드가 오버로딩되어 있기 때문에 int, long, byte 같은 원시 타입이나 배열같은 데이터도 타입 변환없이 바로 검증 가능

Spring Boot Test

- ❖ JUnit
  - ✓ JUnit을 이용한 테스트 클래스 생성 및 수행
    - ● Assertions 클래스
      - ◆ 테스트 대상 메서드가 예외를 던지는 상황도 검증할 수 있는데 이때 사용하는 검증 메서드는 Assertions 클래스의 assertThrow()
      - ◆ assertThrow() 메서드도 여러 인자를 받도록 오버로딩되어 있으며 가장 많이 사용하는 타입은 assertThrow(Class<T> expectedType, Executable executable)로 첫 번째 인자는 테스트 대상 메서드가 던질 것으로 예상하는 예외 클래스 타입 이며 두 번째 인자는 Junit 프레임워크의 Executable 인터페이스
      - ◆ Executable 인터페이스는 함수형 인터페이스로 void execute() throws Throwable 추상 메서드를 가지고 있으며 람다식으로 테스트 대상 클래스의 메서드를 구현하면 되는데 람다식으로 작성된 테스트 대상 메서드를 입력하면 assertThrow 메서드가 첫 번째 인자와 비교하여 예외 발생 여부를 검증할 수 있음

Spring Boot Test

- JUnit
  - ✓ JUnit을 이용한 테스트 클래스 생성 및 수행
    - Assertions 클래스
      - assertThrow() 검증 메서드를 사용 - 테스트 대상인 IOUtils.copy() 메서드가 입력 조건인 input, output 인자를 받으면 IOException 예외를 던진다는 것을 검증하고 있음

```
Assertions.assertThrows(IOException.class,
    () -> IOUtils.copy(input, output));
```

Spring Boot Test

- ❖ @SpringBootTest
  - ✓ 내장 컨테이너를 이용해서 테스트 - @SpringBootTest를 사용한 스프링 부트 테스트
    - 스프링 빈 클래스와 일반 자바 클래스를 모두 Junit의 기능을 사용하여 테스트 케이스를 작성하지만 테스트 대상 클래스를 생성하는 방법이 다름
    - 일반 자바 클래스를 테스트하려면 new 키워드를 사용하여 객체를 생성하고 생성된 객체의 메서드를 테스트
    - 스프링 빈 클래스도 POJO 객체이므로 new 키워드를 사용하여 객체를 생성해서 테스트해도 되지만 이는 스프링 빈이 다른 스프링 빈에 의존하지 않을 때만 가능
    - 테스트 대상 클래스가 다른 여러 스프링 빈 에 의존한다면 이런 방법을 사용할 수 없는데 ApplicationContext가 테스트 대상 클래스가 의존하는 적절한 스프링 빈들을 생성하고 스프링 빈을 주입해야 테스트할 수 있으며 스프링 프레임워크의 기능을 사용한다면 스프링 프레임워크의 기능과 함께 테스트해야 하는데 이를 위해 스프링 부트 프레임워크는 @SpringBootTest Annotation을 제공
    - SpringBootTest Annotation을 테스트 클래스에 선언하면 @SpringBootApplication Annotation이 적용된 클래스를 찾음
    - @SpringBootTest에 설정된 속성과 함께 애플리케이션에 선언된 스프링 빈들도 스캔하고 생성하며 스프링 프레임워크의 기능도 활성화되므로 테스트할 수 있는 환경이 조성됨

Spring Boot Test

- ❖ @SpringBootTest
  - ✓ 내장 컨테이너를 이용해서 테스트 - @SpringBootTest를 사용한 스프링 부트 테스트
    - ● 요청 DTO 클래스 생성 – dto.HotelRequest

```
@Getter
@ToString
public class HotelRequest {
    private String hotelName;

    public HotelRequest() {
    }

    public HotelRequest(String hotelName) {
        this.hotelName = hotelName;
    }
}
```

Spring Boot Test

- ❖ @SpringBootTest
  - ✓ 내장 컨테이너를 이용해서 테스트 - @SpringBootTest를 사용한 스프링 부트 테스트
    - ● 응답 DTO 클래스 생성 – dto.HotelResponse

```
@ToString
@Getter
public class HotelResponse {

    private Long hotelId;
    private String hotelName;
    private String address;
    private String phoneNumber;

    public HotelResponse(Long hotelId, String hotelName, String address, String
     phoneNumber) {
        this.hotelId = hotelId;
        this.hotelName = hotelName;
        this.address = address;
        this.phoneNumber = phoneNumber;
    }
}
```

Spring Boot Test

- ❖ @SpringBootTest
  - ✓ 내장 컨테이너를 이용해서 테스트 - @SpringBootTest를 사용한 스프링 부트 테스트
    - ● Service 인터페이스 생성 – service.DisplayService

```
public interface DisplayService {

    List<HotelResponse> getHotelsByName(HotelRequest hotelRequest);
}
```

Spring Boot Test

- ❖ @SpringBootTest
  - ✓ 내장 컨테이너를 이용해서 테스트 - @SpringBootTest를 사용한 스프링 부트 테스트
    - Service 클래스 생성 – service.DisplayServiceImpl

```
@Slf4j
@Service
public class DisplayServiceImpl implements DisplayService {
    @Override
    public List<HotelResponse> getHotelsByName(HotelRequest hotelRequest) {

        try {
            TimeUnit.SECONDS.sleep(5);
        } catch (InterruptedException e) {
            log.error("error", e);
        }

        return List.of(
                new HotelResponse(1000L,
                        "Ragged Point Inn",
                        "18091 CA-1, San Simeon, CA 93452",
                        "+18885846374"
                )
        );
    }
}
```

Spring Boot Test

- ❖ @SpringBootTest
  - ✓ 내장 컨테이너를 이용해서 테스트 - @SpringBootTest를 사용한 스프링 부트 테스트
    - ● 테스트 클래스 생성 – service.DisplayServiceTest

```
@SpringBootTest
class DisplayServiceTest {

    @Autowired
    private   DisplayService displayService;
    @Autowired
    private   ApplicationContext applicationContext;

    @Test
    public void testReturnOneHotelWhenRequestIsHotelName() {

        // Given
        HotelRequest hotelRequest = new HotelRequest("Line hotel");
        // When
        List<HotelResponse> hotelResponses =
    displayService.getHotelsByName(hotelRequest);
        // Then
        Assertions.assertNotNull(hotelResponses);
        Assertions.assertEquals(1, hotelResponses.size());
    }
```

Spring Boot Test

- ❖ @SpringBootTest
  - ✓ 내장 컨테이너를 이용해서 테스트 - @SpringBootTest를 사용한 스프링 부트 테스트
    - ● 테스트 클래스 생성 – service.DisplayServiceTest

```
@Test
public void testApplicationContext() {
    DisplayService displayService =
 applicationContext.getBean(DisplayService.class);

    Assertions.assertNotNull(displayService);
    Assertions.assertTrue(DisplayService.class.isInstance(displayService));
}

}
```

Spring Boot Test

- @SpringBootTest
  - ✓ @SpringBootTest 구조

```
package org.springframework.boot.test.context;

@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@BootstrapWith(SpringBootTestContextBootstrapper.class)
@ExtendWith({SpringExtension.class})
public @interface SpringBootTest {
    @AliasFor("properties")
    String[] value() default {};

    @AliasFor("value")
    String[] properties() default {};

    Class<?>[] classes() default {};

    String!] args() default {};

    SpringBootTest.WebEnvironment webEnvironment() default
    SpringBootTest.WebEnvironment.MOCK;
}
```

Spring Boot Test

- ❖ @SpringBootTest
  - ✓ @SpringBootTest 구조
    - value나 properties 속성은 프로퍼티 값을 설정하는 데 사용되는데 value, properties 속성은 서로 호환되므로 어떤 속성 이름을 사용해도 상관없는데 속성 값에 사용된 프로퍼티는 해당 테스트 케이스가 실행될 때 사용되는데 테스트에 필요한 실행 환경을 설정할 때 사용
    - 테스트 환경을 구축하는 데 사용되는 ApplicationContext 객체가 로딩 할 자바 설정 클래스 들을 설정할 수 있는데 자바 설정 클래스에 정의된 스프링 빈들이 추가로 로딩되면 스프링 빈 설정에 따라 코드베이스의 스프링 빈을 다시 재정의할 수 있는데 이 속성을 설정하지 않으면 ApplicationContext는 코드베이스에 있는 모든 스프링 빈 객체를 로딩
    - 스프링 부트 애플리케이션을 실행하는 SpringApplication 클래스의 run() 메서드에 인자를 설정할 때 사용
    - 스프링 부트의 웹 테스트 환경을 설정할 수 있는데 기본값은 WebEnvironment.MOCK으로 MOCK 으로 설정된 테스트 케이스는 실제 서블릿 컨테이너를 사용하지 않고 테스트를 실행

Spring Boot Test

- @SpringBootTest
  - ✓ @SpringBootTest Annotation의 WebEnvironment 속성에 설정할 수 있는 값
    - WebEnvironment.MOCK: 서블릿 컨테이너를 실행하지 않고 서블릿을 목으로 만들어 테스트를 실행하는데 스프링 테스트 모듈에서 제공하는 MockMvc 객체를 사용하여 스프링 MVC 기능을 테스트 가능
    - WebEnvironment.RANDOM_PORT: 테스트를 진행할 때 서블릿 컨테이너를 실행하지만 서블릿 컨테이너의 포트를 랜덤 값으로 설정
    - WebEnvironment.DEFINED.PORT: RANDOM_PORT와 마찬가지로 서블릿 컨테이너를 실행하는데 application.properties에 정의된 포트를 사용하여 서블릿 컨테이너를 실행
    - WebEnvironment.NONE: 서블릿 환경을 구성하지 않고 테스트를 실행

Spring Boot Test

- ❖ @SpringBootTest
  - ✓ @SpringBootTest Annotation의 properties 설정 – 해당 테스트 케이스 클래스에만 유효
    - ● @SpringBootTest(properties={"search. host=127.0.0.1", "search. port=19999"}
      - ◆ search.host와 search.port 두 개의 키-밸류를 설정
      - ◆ application,properties에 같은 키 이름의 값이 있다면 그 값을 @SpringBootTest Annotation에 설정된 값으로 대체
    - ● @SpringBootTest(properties={"spring.config.location=classpath:application-test.properties"})
      - ◆ 테스트 케이스에서만 사용할 application-test.properties를 새로 생성하고 테스트에 필요한 여러 프로퍼티 키-밸류 값들을 설정
      - ◆ 여러 설정 값을 테스트 케이스끼리 공용으로 사용할 때 유용

Spring Boot Test

- @TestConfiguration
  - ✓ 테스트 케이스를 작성할 때는 테스트 대상 클래스의 기능을 테스트할 수 있도록 테스트 범위를 좁히는 것이 중요
  - ✓ 대상 클래스의 기능이 다른 클래스에 의존할 때는 의존하는 클래스 설정에 따라 대상 클래스의 기능을 정확하게 테스트할 수 없는 경우가 발생
  - ✓ 테스트 대상 클래스 내부에 데이터베이스에 쿼리하는 기능을 포함하거나 원격 서버의 REST-API를 호출하는 기능을 포함하는 경우 테스트 환경은 Dev 또는 Production 환경과 분리되어 있어 테스트를 위한 데이터베이스나 서버를 구축해야 하는데 서버를 구축해도 같은 테스트를 여러 번 실행하면 데이터베이스의 데이터나 서버의 데이터가 훼손되어 매번 같은 결과 값을 확인할 수 없음
  - ✓ 테스트 환경을 구축하지 못한다면 테스트 케이스를 실행할 때마다 ConnectionException 같은 예외가 발생하여 테스트를 정상적으로 실행할 수 없으므로 테스트를 할 때 테스트 대상 클래스의 기능만 독립적으로 테스트할 수 있으면 되는데 가상의 실행 환경을 만들면 가능
  - ✓ 스프링 프레임워크에서 제공하는 @Testconfiguration Annotation을 사용하면 테스트 환경을 쉽게 구축 할 수 있음
  - ✓ @TestConfiguration은 @Configuration Annotation과 비슷한 기능을 제공하며 자바 설정 클래스를 정의하는 목적으로 사용
  - ✓ @TestConfiguration Annotation이 사용된 클래스에는 @Bean으로 정의한 스프링 빈을 하나 이상 포함할 수 있음

Spring Boot Test

- ❖ @TestConfiguration
  - ✓ @TestConfiguration 과 @Configuration의 차이점
    - @TestConfiguration Annotation은 테스트 환경에서만 유효한 스프링 빈을 추가로 정의할 수 있으며 실행 환경에 중복된 스프링 빈이 있다면 @TestConfiguration의 스프링 빈으로 재정의(overwrite)
    - @Configuration은 @SpringBootConfiguration이 자동으로 스캔한 후 로딩되지만 @Testconfiguration은 그렇지 않기 때문에 테스트 클래스에서 Annotation을 사용하여 명시적으로 로딩해야 함
    - @Configuration이 정의된 자바 설정 클래스는 src > main 디렉터리에 저장하고 @TestConfiguration이 정의된 테스트 자바 설정 클래스는 src > test 디렉터리에 저장
  - ✓ 이 기능을 사용하면 기존 애플리케이션에서 이미 정의한 스프링 빈을 테스트에 적합하게 커스터마이징할 수 있음
  - ✓ 테스트 실행 환경에서는 직접 데이터베이스에 연결하지 않고 애플리케이션 내부의 메모리 데이터베이스에 접근하도록 변경할 수 있고 다른 서버의 REST-API를 바로 호출하지 않고 내부의 목(mock) 서버를 연결하도록 커스터마이징 할 수 있음
  - ✓ 커스터마이징된 스프링 빈은 ApplicationContext에 로딩되고 이를 의존하는 다른 스프링 빈에 주입되는데 ApplicationContext 덕분에 애플리케이션의 코드를 수정하지 않고 테스트 환경에 적합하게 커스터마이징된 스프링 빈을 쉽게 주입할 수 있음

Spring Boot Test

- ❖ @TestConfiguration
  - ✓ 테스트 환경 설정
    - ● @Configuration은 @SpringBootConfiguration이 자동으로 스캔한 후 로딩되지만 @Testconfiguration은 그렇지 않기 때문에 테스트 클래스에서 Annotation을 사용하여 명시적으로 로딩해야 함
    - ● @Configuration이 정의된 자바 설정 클래스는 src > main 디렉터리에 저장하고 @TestConfiguration이 정의된 테스트 자바 설정 클래스는 src > test 디렉터리에 저장

Spring Boot Test

- ❖ @TestConfiguration
  - ✓ 테스트 환경 설정
    - ● Entity 클래스 생성 – domain.HotelRoomEntity

```
@Getter
@ToString
public class HotelRoomEntity {

    private Long id;
    private String code;
    private Integer floor;
    private Integer bedCount;
    private Integer bathCount;

    public HotelRoomEntity(Long id, String code, Integer floor, Integer bedCount,
     Integer bathCount) {
        this.id = id;
        this.code = code;
        this.floor = floor;
        this.bedCount = bedCount;
        this.bathCount = bathCount;
    }
}
```

Spring Boot Test

- ❖ @TestConfiguration
  - ✓ 테스트 환경 설정
    - ● DTO 클래스 생성 – dto.HotelRoomResponse

```
@Getter
public class HotelRoomResponse {

    private Long hotelRoomId;
    private String code;
    private Integer floor;
    private Integer bedCount;
    private Integer bathCount;

    private HotelRoomResponse(Long hotelRoomId, String code, Integer floor,
     Integer bedCount, Integer bathCount) {
        this.hotelRoomId = hotelRoomId;
        this.code = code;
        this.floor = floor;
        this.bedCount = bedCount;
        this.bathCount = bathCount;
    }
```

Spring Boot Test

- ❖ @TestConfiguration
  - ✓ 테스트 환경 설정
    - ● DTO 클래스 생성 – dto.HotelRoomResponse

```
        public static HotelRoomResponse from(HotelRoomEntity hotelRoomEntity) {
            return new HotelRoomResponse(hotelRoomEntity.getId(),
                    hotelRoomEntity.getCode(),
                    hotelRoomEntity.getFloor(),
                    hotelRoomEntity.getBedCount(),
                    hotelRoomEntity.getBathCount());
        }
    }
```

Spring Boot Test

- ❖ @TestConfiguration
  - ✓ 테스트 환경 설정
    - ● Repository 클래스 생성 – persistence.HotelRoomRepository

```
@Repository
public class HotelRoomRepository {

    public HotelRoomEntity findById(Long id) {
        return new HotelRoomEntity(id, "EAST-1902", 19, 2, 1);
    }
}
```

Spring Boot Test

- ❖ @TestConfiguration
  - ✓ 테스트 환경 설정
    - Service 클래스 생성 – service.HotelRoomDisplayService

```
@Slf4j
@Service
public class HotelRoomDisplayService {

    private final HotelRoomRepository hotelRoomRepository;
    private final ThreadPoolTaskExecutor threadPoolTaskExecutor;

    public HotelRoomDisplayService(HotelRoomRepository hotelRoomRepository,
                                   ThreadPoolTaskExecutor threadPoolTaskExecutor) {
        this.hotelRoomRepository = hotelRoomRepository;
        this.threadPoolTaskExecutor = threadPoolTaskExecutor;
    }

    public HotelRoomResponse getHotelRoomById(Long id) {
        HotelRoomEntity hotelRoomEntity = hotelRoomRepository.findById(id);
        threadPoolTaskExecutor.execute(() -> log.warn("entity :{}",
     hotelRoomEntity.toString()));
        return HotelRoomResponse.from(hotelRoomEntity);
    }
}
```

Spring Boot Test

- ❖ @TestConfiguration
  - ✓ 테스트 환경 설정
    - ● Config 클래스 생성 – config.ThreadPoolConfig

```
@Configuration
public class ThreadPoolConfig {

    @Bean(destroyMethod = "shutdown")
    public ThreadPoolTaskExecutor threadPoolTaskExecutor() {
        ThreadPoolTaskExecutor taskExecutor = new ThreadPoolTaskExecutor();
        taskExecutor.setMaxPoolSize(10);
        taskExecutor.setThreadNamePrefix("AsyncExecutor-");
        taskExecutor.afterPropertiesSet();
        return taskExecutor;
    }
}
```

Spring Boot Test

- ❖ @TestConfiguration
  - ✓ 테스트 환경 설정
    - ● Config 클래스 생성 – config.TestConfig

```
@TestConfiguration
public class TestConfig {

    @Bean(destroyMethod = "shutdown")
    public ThreadPoolTaskExecutor threadPoolTaskExecutor() {
        ThreadPoolTaskExecutor taskExecutor = new ThreadPoolTaskExecutor();
        taskExecutor.setMaxPoolSize(1);
        taskExecutor.setThreadNamePrefix("TestExecutor-");
        taskExecutor.afterPropertiesSet();
        return taskExecutor;
    }

}
```

Spring Boot Test

- ❖ @TestConfiguration
  - ✓ 테스트 환경 설정
    - application-test.properties 파일을 생성하고 작성

      spring.main.allow-bean-definition-overriding=true
      - ◆ @TestConfiguration을 사용하여 스프링 빈을 재정의하려면 스프링 부트 프레임워크 에 추가 설정이 필요
      - ◆ 스프링 부트 프레임워크의 기본값은 스프링 빈 재정의를 허용하지 않기 때문에 설정을 추가
      - ◆ 재정의 설정을 하지 않으면 BeanDefinitionOverrideException이 발생하여 테스트 코드가 정상적으로 실행되지 않음

Spring Boot Test

- ❖ @TestConfiguration
  - ✓ 테스트 환경 설정
    - ● 테스트 클래스를 생성하고 실행

```
@SpringBootTest
@ContextConfiguration(classes = TestConfig.class)
@TestPropertySource(locations = "classpath:application-test.properties")
public class HotelRoomDisplayServiceTest01 {

    @Autowired
    private HotelRoomDisplayService hotelRoomDisplayService;

    @Test
    public void testTestConfiguration() {
        HotelRoomResponse hotelRoomResponse =
     hotelRoomDisplayService.getHotelRoomById(1L);

        Assertions.assertNotNull(hotelRoomResponse);
        Assertions.assertEquals(1L, hotelRoomResponse.getHotelRoomId());
    }
}
```

Spring Boot Test

- MockBean
  - ✓ spring-boot-starter-test 스타터는 spring-boot-test 모듈을 포함하고 있는데 spring-boot-test 모듈은 Mockito 라이브러리와 @MockBean Annotation을 같이 제공
  - ✓ Mockito는 일종의 모의 객체라고 하는 목(mock) 객체를 쉽게 만들고 목 객체의 행동과 결과를 검증할 수 있는 기능을 제공하기 때문에 JUnit과 함께 많이 사용되는 라이브러리이며 테스트 환경 설정부터 테스트 검증 까지 테스트 전체 과정 모두를 처리할 수 있는 기능을 제공
  - ✓ Mockito 라이브러리에서 제공하는 @Mock Annotation도 있는데 @Mock과 @MockBean Annotation을 비교하면 @Mock Annotation은 Junit과 Mockito를 사용하여 일반 자바 객체를 테스트하는 데 사용하는 반면 스프링 프레임워크의 ApplicationContext를 사용하여 주입된 스프링 빈을 목 객체로 만들려면 @MockBean Annotation을 사용

Spring Boot Test

- MockBean
  - ✓ Mock 객체를 이용한 테스트
    - ● 테스트 클래스 생성

```
import static org.mockito.ArgumentMatchers.any;
import static org.mockito.BDDMockito.given;

@SpringBootTest
public class HotelRoomDisplayServiceTest02 {
    @Autowired
    private HotelRoomDisplayService hotelRoomDisplayService;
    @MockBean
    private HotelRoomRepository hotelRoomRepository;
    @Test
    public void testMockBean() {
        given(this.hotelRoomRepository.findById(any()))
                .willReturn(new HotelRoomEntity(10L, "test", 1, 1, 1));

        HotelRoomResponse hotelRoomResponse =
     hotelRoomDisplayService.getHotelRoomById(1L);

        Assertions.assertNotNull(hotelRoomResponse);
        Assertions.assertEquals(10L, hotelRoomResponse.getHotelRoomId());
    }
}
```

Spring Boot Test

- MockBean
  - ✓ Mock 객체를 이용한 테스트
    - 테스트 클래스 생성
      - HotelRoomRepository에 @MockBean Annotation을 사용하여 목 객체를 생성
      - Annotation을 선언하면 테스트 프레임워크가 만든 HotelRoomRepository 목 객체가 Applicationcontext에 포함되는데 해당 목 객체는 ApplicationContext의 HotelRoomDisplayService에 주입
      - 목 객체의 행동을 설정할 수 있는데 hotelRoomRepository의 findByld 메서드가 호출되면 호텔 객실 아이디가 10L인 HotelRoomEntity 객체를 리턴
      - findByldO 메서드에 어떤 인자를 의미하는 any()를 설정했으므로 인수 값에 상관없이 willReturnO 메서드의 인자로 설정된 HotelRoomEntity 객체를 무조건 리턴
      - given()은 BDDMockito 클래스의 static 메서드이며 any()는 ArgumentMatchers의 static 메서드
      - 코드 가독성을 높이고자 static import를 사용하여 테스트 케이스를 작성하기도 함
      - 테스트 케이스에서 설정된 HotelRoomRepository 목 객체는 항상 hotelRoomId가 10L인 HotelRoomEntity를 리턴하므로 HotelRoomResponse 객체의 hotelRoomId가 10L인지 검증

Spring Boot Test

- MockBean
  - ✓ Mock 객체를 이용한 테스트
    - 테스트 클래스 생성
      - BDDMockito.given(T methodCall): stub을 만드는 메서드이며 given()의 인자에 stub로 만들 대상 메서드를 입력하면 되는데 stub는 프로그래밍 할 수 있으며 개발자가 원하는 결과를 응답하는 메서드를 의미하는데 다음 영역에서는 인자로 입력한 hotelRepository.findById() 메서드를 given() 메서드를 사용하여 stub로 만들었고 HotelRoomDisplayService 객체 내부에서 hotelRepository.findById() 메서드를 호출하면 프로그래밍 된 결과가 응답됨
      - ArgumentMatchers.any(): hotelRepository.findById() 메서드는 Long id 인자를 받는데 HotelRoomDisplayService 객체가 hotelRepository.findById() 메서드를 호출 할 때 인자에 다양한 값이 전달될 수 있는데 이때 인자 값에 대한 조건을 설정할 수 있고 ArgumentMatchers의 static 메서드를 적절히 사용하면 되는데 예제에서는 어떤 모든 타입의 값을 의미하는 ArgumentMatchers.any() 메서드를 설정했으므로 어떤 인자 값을 사용 하더라도 given()으로 만들어진 stub이 동작
      - BDDMockito.willReturn(T value): 앞서 생성한 조건에 맞는 stub가 호출되면 willReturn() 메서드의 인자로 입력된 값을 응답하므로 설정된 내용을 해석하면 어떤 인자 값을 사용하더라도 hotelRepository.findById() 메서드를 호출하면 willReturn() 메서드의 인자인 HotelRoomEntity 객체를 응답한다는 의미

Spring Boot Test

- MockBean
  - ✓ Mock 객체를 이용한 테스트
    - 테스트 클래스 생성
      - Mockito 라이브러리에서는 BDDMockito.java와 같이 stub를 만들 수 있는 Mockito.java 클래스를 제공하는데 BDDMockito 클래스는 Mockito 클래스를 상속받기 때문에 두 클래스에서 제공하는 메서드는 거의 동일
      - BDDMockito의 given()과 같은 기능으로는 Mockito의 when() 메서드가 있지만 두 클래스의 차이점은 행위 주도 개발(Behavior Driven Development - BDD) 기반의 메서드 이름을 제공하는 것
      - 행위 주도 개발론은 테스트 케이스를 작성할 때 기대하는 행동과 결과를 처리하는 과정을 자연어에 가깝게 프로그래밍하여 가독성과 유지 보수성을 높이는 것이 목적

Spring Boot Test

- MockBean
  - ✓ Mock 객체를 이용한 테스트
    - 테스트 클래스 생성
      - ArgumentMatchers는 stub의 인자 조건을 설정할 수 있는 기능을 제공
      - ArgumentMatchers의 메서드
        - public static <T> T any(Class<T> type): null을 제외한 type 객체와 일치하는 경우를 의미
        - public static <T> T isA(Class<T> type): 클래스 type 인자를 구현하는 경우를 의미
        - public static boolean anyBoolean(): 불리언 타입인 어떤 값이라도 일치하는 경우를 의미
        - public static byte anyByte(): byte 혹은 Byte 타입인 어떤 값이라도 일치하는 경우를 의미
        - public static byte anyInt(): int 혹은 Integer 타입인 어떤 값이라도 일치하는 경우를 의미하는데 이외에도 이와 비슷한 anyLong(), anyFloat(), anyDouble(), anyShort(), anyString() 등이 있음다.
        - public static <T> List<T> anyList(): List 타입인 어떤 값이라도 일치하는 경우를 의미하는데 이외에도 anySet(), anyMap()과 같은 자료 구조들과 매칭할 수 있는 메서드가 있음
        - public static int eq(int value): value 값과 일치하는 경우를 의미하는데 이외에도 여러 인자 타입을 입력받을 수 있는 eq() 메서드를 오버로딩

Spring Boot Test

- MockBean
  - ✓ Mock 객체를 이용한 테스트
    - ● 테스트 클래스 생성
      - ◆ ArgumentMatchers의 조건과 given()으로 생성된 stub의 응답 값을 설정하는 메서드
        - public static BDDStubber willReturn(Object toBeReturned): 조건이 충족되면 미리 정해진 값을 응답하는 방법으로 고정된 값을 응답
        - public static BDDStubber willReturn(Object toBeReturned, Object... toBeReturnedNext): 조건이 충족되면 미리 정해진 값을 응답하고, 계속해서 stub 메서드를 호출하면 toBeReturnNext 인자들이 순서대로 응답
        - public static BDDStubber willDoNothing(): 아무것도 응답하지 않음
        - public static BDDStubber willThrow(Class<? extends Throwable> toBeThrown): 조건이 충족되면 toBeThrown 예외를 응답
        - public static BDDStubber will(Answer<?> answer): 인자 Answer 인터페이스를 사용하여 개발자가 원하는 응답을 프로그래밍할 수 있는데 willReturn()은 고정된 값을 리턴하지만 will()은 상황에 따라 동적으로 응답하도록 프로그래밍 가능
        - public static BDDStubber willAnswer(Answer<?> answer): will()과 같은 역할

Spring Boot Test

- ❖ MockBean
  - ✓ Mock 객체를 이용한 테스트
    - ● 테스트 클래스 생성
      - ◆ willAnswer() static 메서드는 Answer 인터페이스를 인자로 받는데 Answer 인터페이스는 목 객체의 메서드가 리턴하는 값을 동적으로 프로그래밍할 수 있는 기능을 제공하는데 추상 메서드 answer()에서 응답을 프로그래밍할 수 있으며 answer()의 인자 InvocationOnMock 객체는 목 객체가 응답하는 메서드의 정보를 포함하고 있어서 이 둘을 사용하여 answer() 메서드를 구현하면 상황에 따라 동적으로 응답하는 목 객체를 만들 수 있음

Spring Boot Test

- MockBean
  - ✓ Mock 객체를 이용한 테스트
    - Answer 사용
      - ◆ Answer 인터페이스

```
public interface Answer<T> {
        T answer(InvocationOnMock invocation) throws Throwable;
}
```

      - ◆ Answer 인터페이스 사용법

```
given(this.hotelRoomRepository.findById(any())).willAnswer(new
    Answer<HotelRoomEntity>() {
        @Override
        public HotelRoomEntity answer(InvocationOnMock invocation)
    throws Throwable{
        Long id = invocation.getArgument(0);
        if (id != null && id > 10)
            return new HotelRoomEntity(id, "CODE", 10L, 2, 2);
        else
            return new HotelRoomEntity(10L, "test", 1, 1, 1 );
        }
    });
```

Spring Boot Test

- ❖ MockBean
  - ✓ Mock 객체를 이용한 테스트
    - ● Answer 사용
      - ◆ stub 메서드가 응답하는 클래스의 타입을 제네릭으로 입력한 후 hotelRoomRepository의 findById() 메서드가 응답하는 클래스 타입은 HotelRoomEntity 클래스
      - ◆ InvocationOnMock은 목 메서드의 인자, 메서드 이름, 목 객체 정보를 포함하며 이 정보들을 참조할 수 있는 메서드를 제공
      - ◆ InvocationOnMock의 getArgument() 메서드를 사용하면 런타임 시점에서 목 메서드의 인자 정보를 획득할 수 있음
      - ◆ getArgument()는 0 부터 시작하는 인덱스 기반의 인자를 받으므로 getArgument()는 findById() 메서드의 첫 번째 인자를 리턴
      - ◆ 상황에 따라 동적으로 응답하는 코드를 작성할 수 있는데 id 값이 10을 초과하면 id 값을 그대로 사용해서 HotelRoomEntity 객체를 만들어 리턴하고 아니면 hotelRoomId 값이 10L로 고정 된 HotelRoomEntity 객체를 만들어 리턴
      - ◆ 모든 테스트 메서드마다 같은 목 객체를 설정하는 것은 매우 비효율적
      - ◆ @BeforeAll이나 @BeforeEach Annotation을 사용하여 목 객체를 설정하는 부분을 따로 정의하면 보다 간략하고 생산성 있는 테스트 케이스를 작성할 수 있음

Spring Boot Test

- Test Slice Annotation
  - ✓ @SpringBootTest Annotation으로 테스트를 실행하면 Applicationcontext를 이용하여 스프링 빈을 스캔하고 의존성 주입
  - ✓ 애플리케이션의 기능이 많다면 스캔해야 할 대상이 많아지고 그만큼 많은 객체를 생성해야 하기 때문에 테스트 시간이 오래 걸릴 수밖에 없는데 이는 F.IR.S.T 원칙의 Fast 항목에 위배됨
  - ✓ 테스트를 진행하여 결과를 확인하는 피드백 시간이 늘어나면 애플리케이션을 디버깅할 수 있는 시간이 줄어들고 배포 전략에 따라 테스트를 통과해야 패키징을 한다고 가정하면 테스트 시간이 오래 걸리면 빠른 배포를 할 수 없음
  - ✓ 스프링 부트 프레임워크는 테스트 시간을 줄이려고 테스트 슬라이스 개념을 제공하는데 기능 별로 잘라서 테스트 대상을 좁히는 방법을 의미
  - ✓ 테스트 대상이 좁혀지면 ApplicationContext가 스캔해야 하는 스프링 빈의 개수도 줄어들고 기능도 초기화하지 않아도 됨
  - ✓ @Controller의 웹 MVC 기능만 테스트한다고 가정하면 개발자가 의도한 대로 HTTP 파라미터가 변경되는지 REST-API 응답 코드가 적절한지 테스트하려면 웹 MVC 기능만 테스트하면 되므로 ApplicationContext가 모든 스프링 빈을 스캔하지 않고 기능에 필요한 스프링 빈만 스캔

Spring Boot Test

- Test Slice Annotation
  - ✓ 테스트 슬라이스 기능과 관련하여 스프링 부트 프레임워크가 제공하는 Annotation
    - @WebMvcTest
      - 스프링 MVC 프레임워크의 기능을 테스트
      - @Controller, @ControllerAdvice를 스캔하고 Converter, Filter, WebMvcConfigurer 같은 MVC 기능도 제공
      - @Service, @Component, @Repository로 정의된 스프링 빈들은 스캔하지 않음
    - @DataJpaTest
      - 데이터 영속성 프레임워크인 JPA 기능을 테스트
      - @Repository Annotation을 대상으로 스캔
      - JPA 프레임워크에서 사용할 수 있는 EntityManager, TestEntityManager, DataSource 같은 기능을 제공
      - @Service, @Component, @Controller로 정의된 스프링 빈들은 스캔하지 않음
    - @JsonTest
      - Json 직렬화(serialization), 역직렬화(deserialization)를 테스트
      - @JsonComponent, ObjectMapper 같은 기능을 테스트할 수 있고 @Service, @Component, @Controller, @Repository로 정의된 스프링 빈들은 스캔하지 않음

Spring Boot Test

- Test Slice Annotation
  - ✓ 테스트 슬라이스 기능과 관련하여 스프링 부트 프레임워크가 제공하는 Annotation
    - @RestClientTest
      - HTTP 클라이언트의 동작을 테스트할 수 있는 기능을 제공
      - MockRestServiceServer 와 Jackson 자동 설정 기능을 제공
      - @Service, @Component, @Controller, @Repository로 정의된 스프링 빈들은 스캔하지 않음
    - @DataMongoTest
      - MongoDB를 테스트하기 위해 MongoTemplate, CrudRepository 같은 기능을 테스트
      - @Service, @Component. @Controller로 정의된 스프링 빈들은 스캔하지 않음

Spring Boot Test

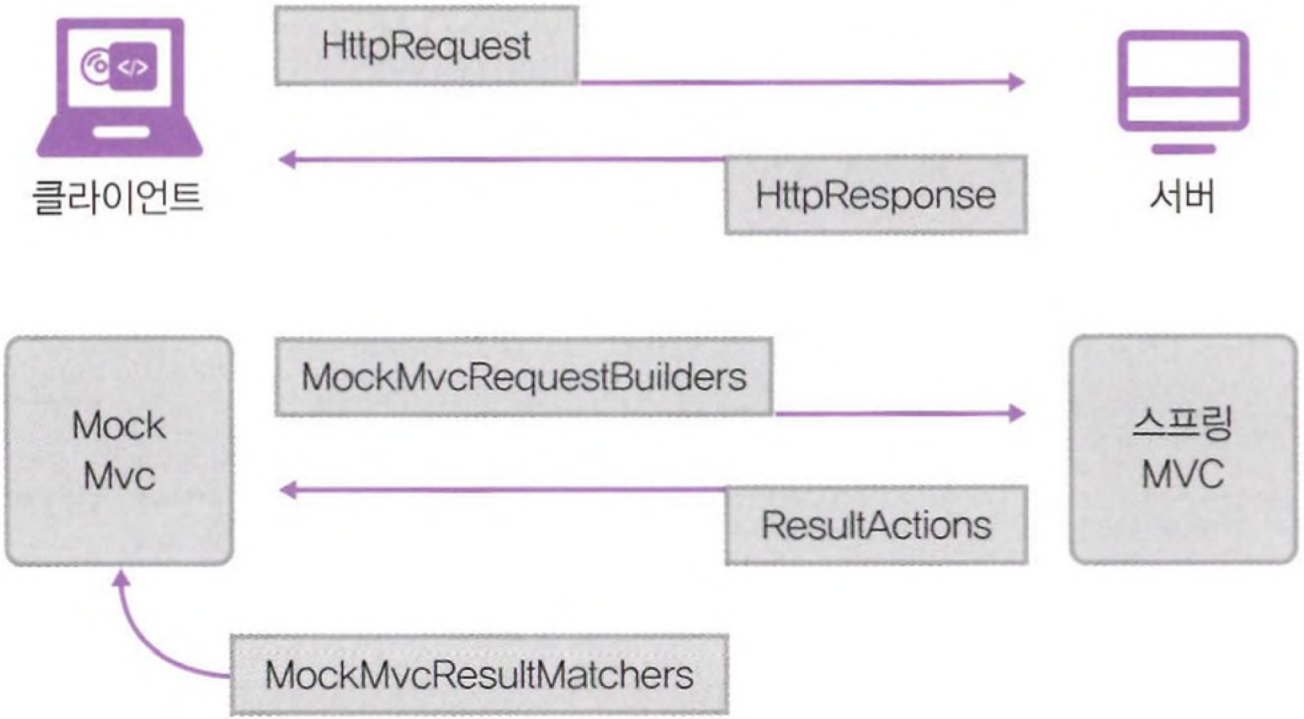
- Spring Boot Web MVC Test
  - ✓ HTTP 프로토콜을 사용하여 클라이언트가 요청을 보내고 서버가 어떤 응답을 하는지 테스트하는 방식이 아니라 @WebMvcTest Annotqtion을 사용
  - ✓ 서버와 클라이언트 사이의 네트워크 구간은 문제없다고 생각하고 서버 애플리케이션의 요청, 응답 기능이 정상적으로 동작하는지 확인
  - ✓ 스프링 프레임워크에서 서버 기능을 제 공하는 모듈이 WebMVC라서 이런 웹 테스트를 WebMVC 테스트라고도 함
  - ✓ REST-API 요청 조건에 따라 컨트롤러 클래스가 어떤 형식의 JSON 응답 메시지를 리턴하는지 테스트
  - ✓ 응답 메시지의 Content-type 헤더 값을 검증하거나 JSON 메시지 값을 검증하는 테스트 케이스를 작성할 수 있음

Spring Boot Test

- Spring Boot Web MVC Test
  - ✓ MockMvc
    - spring-test 모듈은 WebMVC를 테스트할 수 있는 org.springframework.test.web.servlet.MockMvc 클래스를 제공하는데 HTTP 클라이언트처럼 서버의 API에 요청을 하고 그 응답을 받아 올 수 있 는 기능을 제공
    - 실제로 HTTP 프로토콜을 사용하여 서버의 API를 호출하지 않고 요청을 전송하고 응답을 받아 올 수 있는 기능을 목 객체로 제공
    - MockMvc를 사용하여 테스트 케이스를 작성할 때 주로 함께 사용하는 클래스는 MockMvcRequestBuilders, MockMvcResultMatchers, ResultActions
    - MockMvcRequestBuilders는 MockMvc를 사용하여 HTTP 요청을 전달할 때 HTTP 요청을 목 객체로 만들 수 있는 기능을 제공하기 때문에 요청 메시지의 HTTP 헤더 나 파라미터, HTTP 바디를 설정할 수 있는 메서드를 제공
    - MockMvcResultMatchers는 HTTP 응답 메시지를 검증할 수 있는 기능들을 제공하고 ResultActions는 서버에서 처리한 HTTP 응답 메시지의 각 속성에 접근할 수 있는 메서드를 제공

Spring Boot Test

- ❖ Spring Boot Web MVC Test
  - ✓ MockMvc
    - ● MockMvc와 관련 클래스들의 역할



Spring Boot Test

- ❖ Spring Boot Web MVC Test
  - ✓ MockMvc
    - ● perform 메서드

```
public ResultActions perform(RequestBuilder requestBuilder) throws Exception {
        // 생략
}
```

      - ◆ perform() 메서드는 인자로 RequestBuilder 인터페이스를 받음
      - ◆ 테스트 HTTP 요청 객체 즉 MockHttpServletRequest 객체를 만드는 기능을 제공
      - ◆ GET, POST, PUT. DELETE. HTTP 메서드, 메시지 바디, 헤더, URI, 파라미터 등을 설정할 수 있음

Spring Boot Test

- Spring Boot Web MVC Test
  - ✓ MockMvc
    - perform 메서드

```
mockMvc.perform(post("/hotels/{id}", 11)
        .contentType(MediaType.APPLICATION_JSON)
        .accept(MediaType.APPLICATION_JSON)
        .content(jsonString)
        .header("x-channel-name", "web")
);
mockMvc.perfom(get ("/hotels/{id}", 11)
        .queryParam("type,', "5star")
);
mockMvc.perform(put("/hotels/{id}", 11));
mockMvc.perform(delete("/hotels/{id}", 11));
```

Spring Boot Test

- ❖ Spring Boot Web MVC Test
  - ✓ MockMvc
    - perform 메서드
      - ◆ http POST 메서드를 사용하여 URI '/hotels/11'로 요청하는데 이때 {id}는 post() 메서드의 두 번째 인자인 11로 교체
      - ◆ 요청 메시지의 Content-type 헤더에 'application/json' 헤더 값을 설정
      - ◆ 요청 메시지의 Accept 헤더에 'application/json' 헤더 값을 설정
      - ◆ 요청 메시지의 바디에 문자열 값을 설정하는데 content() 메서드 외에 오버로딩된 content(byte[ ] content) 메서드도 제공하기 때문에 파일을 업로드하는 테스트도 가능
      - ◆ 요청 메시지의 헤더 영역에 헤더와 헤더 값을 설정할 수 있는데 x-channel-name 헤더에 web 값을 설정
      - ◆ HTTP GET 메서드를 사용하여 URI '/hotels/11'로 요청하는데 post() 메서드와 마찬가지로 두 번째 인자 11이 {id} 와 교체
      - ◆ URI에 쿼리 파라미터를 설정하므로 URI와 합쳐 GET /hoteLs/11?type=5star
      - ◆ http PUT 메서드를 사용하여 URI '/hotels/11'로 요청
      - ◆ HTTP DELETE 메서드를 사용하여 URI '/hotels/11'로 요청

Spring Boot Test

- Spring Boot Web MVC Test
  - ✓ MockMvc
    - MockMvcRequestBuilders에서도 queryParam() 과 비슷한 param() 메서드를 제공하는데 param() 메서드는 application/x-www-form-urlencoded 콘텐츠 타입의 요청 메시지 바디에 키-밸류 데이터를 설정하는데 사용
    - MockMvc의 perform() 메서드는 실행된 결과를 포함하고 있는 ResultActions를 리턴
    - ResultActions 클래스는 응답 데이터를 검증할 수 있는 andExpect() 메서드를 제공하는데 andExpect() ResultMatcher 객체를 인자로 받으며 메서드 체인 방식으로 게속 연결하여 사용할 수 있음
    - MockMvcResultMatchers에서 제공하는 static 메서드들은 ResultMatcher 객체를 리턴하며 ResultActions의 andExpect() 메서드를 사용하여 테스트 결과를 검증하는데 HTTP 응답 메시지를 검증해야 하므로 HTTP 응답 메시지의 상태 코드. 메시지 바디. 헤더 등을 검증할 수 있음

Spring Boot Test

- Spring Boot Web MVC Test
  - ✓ 테스트
    - Controller 클래스 생성

```
@Slf4j
@RestController
@RequiredArgsConstructor
public class HotelController {

    private final DisplayService displayService;

    @ResponseBody
    @PostMapping( "/hotels/fetch-by-name")
    public ResponseEntity<List<HotelResponse>>
     getHotelByName(@RequestBody HotelRequest hotelRequest) {
        List<HotelResponse> hotelResponses =
     displayService.getHotelsByName(hotelRequest);
        return ResponseEntity.ok(hotelResponses);
    }
}
```

Spring Boot Test

- Spring Boot Web MVC Test
  - ✓ 테스트
    - JsonUtil 클래스 생성

```
public class JsonUtil {

    public static final ObjectMapper objectMapper = new ObjectMapper();

}
```

Spring Boot Test

- ❖ Spring Boot Web MVC Test
  - ✓ 테스트
    - ● Test 클래스 생성

```
import static
    org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static
    org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)
@AutoConfigureMockMvc
public class ApiControllerTest01 {

    @Autowired
    private MockMvc mockMvc;
```

Spring Boot Test

- ❖ Spring Boot Web MVC Test
  - ✓ 테스트
    - Test 클래스 생성

```
@Test
public void testGetHotelById() throws Exception {
    HotelRequest hotelRequest = new HotelRequest("Ragged Point Inn");
    String jsonRequest =
JsonUtil.objectMapper.writeValueAsString(hotelRequest);
    mockMvc.perform(post("/hotels/fetch-by-name")
            .content(jsonRequest)
            .contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(content().contentType(MediaType.APPLICATION_JSON))
            .andExpect(content().string("[{₩"hotelId₩":1000,₩"hotelName₩":₩"Rag
ged Point Inn₩",₩"address₩":₩"18091 CA-1, San Simeon, CA
93452₩",₩"phoneNumber₩":₩"+18885846374₩"}]"))
            .andExpect(content().json("[{₩"hotelId₩":1000,₩"hotelName₩":₩"Ragg
ed Point Inn₩",₩"address₩":₩"18091 CA-1, San Simeon, CA
93452₩",₩"phoneNumber₩":₩"+18885846374₩"}]"))
            .andExpect(jsonPath("$[0].hotelId", Matchers.is(1000)))
            .andExpect(jsonPath("$[0].hotelName", Matchers.is("Ragged Point
Inn")))
            .andDo(MockMvcResultHandlers.print(System.out));
    }
}
```

Spring Boot Test

- ❖ Spring Boot Web MVC Test
  - ✓ 테스트
    - ● Test 클래스
      - ◆ 테스트 케이스의 가독성을 높이려고 MockMvcRequestBuilders와 MockMvcResultMatchers의 메서드를 import
      - ◆ MockMvcBuilders 클래스를 사용하여 MockMvc 객체를 생성할 수 있지만 스프링 부트에서 제공하는 @AutoConfigureMockMvc를 정의하면 MockMvc 객체를 스프링 빈으로 주입받을 수 있어서 @SpringBootTest 애너테이션을 사용하여 스프링 MVC 기능을 테스트할 때 같이 사용
      - ◆ WebEnvironment.MOCK 설정을 사용하여 @SpringBootTest 애너테이션을 정의하는데 실제 서블릿 컨테이너를 실행하지 않고 목 서블릿 컨테이너를 사용하여 테스트
      - ◆ @AutoConfigureMockMvc로 생성된 MockMvc 스프링 빈을 주입
      - ◆ 테스트 대상 REST-API는 POST 메서드를 사용하며 URI는 /hotels/fetch-by-name 이므로 요청 메시지를 작성하려고 MockMvc의 content() 메서드를 사용하는데 content() 메서드는 요청 메시지 바디에 JSON 메시지인 jsonRequest를 추가하고 contentType() 메서드를 사용하여 콘텐츠 타입을 JSON으로 설정해서 생성된 RequestBuilder와 함께 mockMvc.perform()을 실행하여 테스트를 진행

Spring Boot Test

- ❖ Spring Boot Web MVC Test
  - ✓ 테스트
    - ● Test 클래스
      - ◆ MockMvcResultMatchers의 status()는 응답 메시지의 상태 코드를 검증하기 위해 이를 검증할 수 있는 StatusResultMatchers를 리턴하므로 StatusResultMatchers의 isOk() 메서드를 사용하며 응답 메지지가 200 OK 인지 검증
      - ◆ MockMvcResultMatchers의 content() 메서드는 ContentResultMatchers를 리턴하고 리턴된 ContentResultMatchers를 사용하면 응답 메시지를 검증할 수 있는데 contentType() 메서드는 콘텐츠 타입을 검증하며 여기서는 JSON 타입인지 검증
      - ◆ MockMvcResultMatchers의 jsonPath() 메서드는 JSON 메시지의 속성을 검증할 수 있는 JsonPathResultMatchers를 리턴하는데 첫 번째 인자는 JSON path 표현식이며 두 번째 인자는 기대하는 값을 검증할 수 있는 Matcher 객체로 JSON 배열의 첫 번째 엘리먼트에서는 hotelName이 검증 대상

Spring Boot Test

- ❖ Spring Boot Web MVC Test
  - ✓ 테스트
    - ● Test 클래스
      - ◆ @SpringBootTest 애너테이션을 사용하여 테스트했는데 코드베이스의 모든 스프링 빈이 Applicationcontext에 로딩되고 의존성 주입되고 HotelController 와 HotelDisplayService 모두 스프링 빈으로 로딩되고 HotelDisplayService가 HotelController에 주입됨
      - ◆ MockMvc를 사용하여 REST-API를 테스트하면 HotelDisplayService의 실제 코드가 동작
      - ◆ REST-API의 응답 메시지를 테스트할 때 가장 많이 사용하는 것은 jsonPath() 메서드인데 JSON path 표현식을 사용하면 .JSON 응답 메시지의 특정 속성 값을 쉽게 검증할 수 있는데 사용된 표현식 $[0].hotelName의 각 요소가 의미하는 것은 다음과 같음
        - ▪ $: 조회할 최상위 JSON 엘리먼트로 JSON path 표현식을 사용할 때 모든 표현식은 $로 시작해야 함
        - ▪ [(index)]: JSON 배열의 인덱스 값과 같이 사용하며 해당 위치에 있는JSON 엘리먼트를 참조할 수 있음
        - ▪ .<name>: 엘리먼트의 자식을 참조

Spring Boot Test

- ❖ Spring Boot Web MVC Test
  - ✓ 테스트
    - ● Test 클래스 작성

```
import static org.mockito.ArgumentMatchers.any;
import static org.mockito.BDDMockito.given;
import static
    org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static
    org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@WebMvcTest(controllers = HotelController.class)
public class ApiControllerTest02 {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private DisplayService displayService;
```

Spring Boot Test

- Spring Boot Web MVC Test
  - ✓ 테스트
    - Test 클래스 작성

```
@BeforeEach
public void init() {
    given(displayService.getHotelsByName(any()))
            .willAnswer(new Answer<List<HotelResponse>>() {
                @Override
                public List<HotelResponse> answer(InvocationOnMock invocation)
throws Throwable {
                    HotelRequest hotelRequest = invocation.getArgument(0);
                    return List.of(new HotelResponse(1L,
hotelRequest.getHotelName(), "unknown", "213-820-3xxx"));
                }
            });
}
```

Spring Boot Test

- ❖ Spring Boot Web MVC Test
  - ✓ 테스트
    - ● Test 클래스 작성

```
@Test
public void testGetHotelById() throws Exception {
    HotelRequest hotelRequest = new HotelRequest("Line Hotel");
    String jsonRequest =
JsonUtil.objectMapper.writeValueAsString(hotelRequest);

    mockMvc.perform(post("/hotels/fetch-by-name")
            .content(jsonRequest)
            .contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(content().contentType(MediaType.APPLICATION_JSON))
            .andExpect(jsonPath("$[0].hotelId", Matchers.is(1)))
            .andExpect(jsonPath("$[0].hotelName", Matchers.is("Line Hotel")))
            .andDo(MockMvcResultHandlers.print(System.out));
    }
}
```

Spring Boot Test

- ❖ Spring Boot Web MVC Test
  - ✓ 테스트
    - ● Test 클래스 작성
      - ◆ @WebMvcTest 애너테이션을 사용하면 @Controller, @ControllerAdvice, @RestController 애너테이션이 정의된 스프링 빈만 로딩
      - ◆ HotelController가 의존하는 HotelDisplayService 클래스는 @Service 애너테이션으로 정의되어 있으므로 @Service 애너테이션으로 정의된 HotelDisplayService 클래스는 스캔 대상이 될 수 없으므로 테스트를 실행하면 정상적으로 테스트가 실행되지 않음
      - ◆ 주입할 HotelDisplayService 스프링 빈이 없으므로 NoSuchBeanDefinitionException 예외가 발생하므로 ApiControllerTest02 예제처럼 스프링 스캔에 포함되지 않는 것들에는 목 객체를 별도로 생성해야 하며 이를 위해 @MockBean 애너테이션을 사용
      - ◆ @WebMvcTest 애너테이션의 가장 큰 장점은 빠르게 테스트할 수 있다는 것
      - ◆ 스캔 대상 이 줄어들어 빠르게 테스트하는 테스트 슬라이스 방식을 갖고 있으므로 추가적인 개발이 필요할 수 있지만 스프링 MVC 기능만 테스트한다면 @SpringBootTest 처럼 모든 스프링 빈을 스캔하고 생성할 필요 없음
      - ◆ @SpringBootTest 애너테이션과 다른 점은 @WebMvcTest는 @AutoConfigureMockMvc 애너테이션 없이도 MockMvc 객체를 생성

Spring Boot Test

- JPA 테스트
  - ✓ 스프링 프레임워크에서 제공하는 @Repository 애너테이션은 데이터베이스에 데이터를 생성, 삭제, 수정, 조회하는 기능을 제공하는 클래스를 스프링 빈으로 정의할 때 사용
  - ✓ @WebMvcTest 애너테이션이 WebMvc 영역을 구분하여 빠르게 테스트하는 것처럼 데이터를 처리하는 영역을 구분하여 빠르게 테스트할 수 있는 방법이 있는데 @DataJpaTest 나 @DataMongoTest 애너테이션
  - ✓ @DataJpaTest 애너테이션은 @Repository 만 스캔하므로 @Service, @Component. @Controller 애너테이션은 스캔하지 않음
  - ✓ @DataJpaTest를 사용하지 않고 @SpringBootTest 애너테이션을 사용하여 테스트 케이스를 작성한다면 테스트 케이스마다 @Transactional 애너테이션을 정의하는 것이 바람직한데 @DataJpaTest는 소스 내부에 @Transactional 애너테이션을 포함하고 있어 별도로 정의하지 않아도 되는데 테스트 케이스에 @Transactional 애너테이션을 정의하면 테스트 종료 후 롤백되므로 테스트 도중에 발생한 데이터 생성, 수정, 삭제 등 변경된 데이터들이 롤백되어 다시 초기화 상태가 되지지만 SpringBootTest의 environment 속성을 WebEnvironment.RANDOM_PORT나 DEFINED_PORT로 설정하면 롤백되지 않는데 이 설정 때문에 테스트 케이스를 실행하면 별도의 서블릿 컨테이너가 실행됨
  - ✓ 테스트 케이스를 실행하는 스레드와 테스트 케이스가 호출한 서블릿 컨테이너의 스레드가 서로 달라 서블릿 컨테이너의 트랜잭션을 테스트 케이스에서 롤백할 수 없기 때문

Spring Boot Test

- JPA 테스트
  - ✓ 현업에서 테스트 케이스를 유지 보수하는 일은 매우 어렵기 때문에 테스트 케이스를 애플리케이션의 일부라고 생각하지 않는 사람이 많으며 시간에 쫓기면 가장 먼저 포기하는 것이 테스트 케이스이기 때문이지만 테스트 케이스는 지속 가능한 애플리케이션을 만들 수 있는 가장 중요한 자산이자 최후의 보루
  - ✓ 버그가 발생하거나 기능을 추가할 때마다 테스트 케이스를 작성한다면 애플리케이션을 서비스한 기간만큼 테스트 케이스도 많아질 것이고 많아진 테스트 케이스는 애플리케이션의 대부분 기능을 테스트할 것이고 버그에 대응한 테스트가 많아지면 별도의 노력없이 자동으로 회귀 테스트가 됨
  - ✓ 이 정도로 구축되면 여러분의 애플리케이션은 언제든 리팩터링할 수 있으며. 같은 에러가 다시 발생하지 않는 신뢰성 있는 서비스가 될 것
  - ✓ 직접 REST-API를 호출 해서 기능을 테스트하기보다 테스트 케이스를 사용하여 여러분이 개발한 기능을 테스트하는 습관을 갖는 것이 바람직