6부

스프링 시큐리티 테스트

목차

소프트웨어 개발에서 품질 보장에 대한 다양한 방법론이 존재한다. 그중 테스트 코드는 개발 과정에서 발생할 수 있는 오류를 사전에 방지하고, 소프트웨어의 신뢰성을 높여준다. 테스트 코드는 개발자가 의도한 대로 구현 코드가 동작하는지 확인을 돕기 때문에 지속 가능한 소프트웨어 프로젝트를 가능하게 한다. 애자일 소프트웨어 방법론 같은 경우에는 테스트 주도 개발(Test Driven Development, TDD)을 하나의 중요한 관행으로 삼아 구현 코드를 작성하기 전에 테스트 코드를 먼저 작성하기도 한다.

- 개발자는 테스트 코드를 통해 자신의 코드가 의도한 대로 동작하는지 확인할 수 있다. 메서드의 입력과 결과를 테스트 코드를 통해 확인함으로써 작은 단위의 기능들이 제 역할을 수행하는지 판단할 수 있다.
- 테스트 코드는 프로젝트 유지 보수를 쉽게 만든다. 프로덕트를 개발하다 보면 어지러운 코드를 정리하거나 성능을 최적화하기 위한 리팩토링(refactoring)이 자주 일어난다. 리팩토링을 통해 기존 기능이 의도치 않게 망가지는 경우가 발생한다. 테스트 코드는 코드 변경 후에도 모든 기능이 정상적으로 작동하는지 쉽게 확인할 수 있도록 한다.
- 비즈니스 도메인마다 다르겠지만, 시스템을 운영하는 것은 굉장히 힘든 일이다. 금전적인 거래가 이루어지거나 기계들이 동작하는 시스템을 변경한다고 상상해 보라. 작은 코드 변경만으로 사람이 다치거나 큰 손실이 발생할 수 있다. 잘 작성된 테스트 코드들은 내가 변경한 코드가 문제가 없음을 알려주고, 소프트웨어를 실제 환경에 배포하는 데 큰 용기를 준다.
- 테스트 코드는 문서로서 역할을 수행한다. 각 테스트는 구현 코드가 어떻게 동작해야 하는지 설명하는 것이기 때문에, 다른 개발자들이 새로운 기능을 쉽게 추가하거나 기존 기능을 개선하기 위한 컨텍스트를 이해하는 데 도움을 준다.

큰 엔터프라이즈부터 작은 스타트업까지 많은 서비스 조직은 신뢰성 높은 소프트웨어 개발을 위해 테스트를 작성한다. 자동화된 CI/CD 파이프라인(Continuous Integration and Continuous Deployment Pipeline)에서 테스트 코드가 실행된다면 하루에도 빈번한 개발자의 변경 사항들을 효율적으로 확인할 수 있다.

순수한 자바 언어로 작성된 비즈니스 코드를 테스트하는 것은 어렵지 않지만, 프레임워크 기능을 기반으로 동작하는 비즈니스 로직이라면 테스트에서 프레임워크가 개입할 필요가 있다. 스프링 프레임워크는 개발자 경험을 향상시키고 테스트 작성을 원활하게 만들기 위한 도

구들을 제공한다. 프레임워크가 제공하는 테스트 도구들은 테스트 코드의 유지 보수를 쉽게 만들고, 효율적인 전략을 세울 수 있도록 도움을 준다. 이 책의 마지막은 테스트와 이를 지원하는 도구에 관한 이야기를 다룬다.

27

스프링 시큐리티 테스트 지원

앞서 말한 것처럼 순수 자바 언어로 작성된 비즈니스 로직에 대한 테스트 작성은 어렵지 않다. 하지만 프레임워크의 기능에 의존하는 경우는 어떨까? 예를 들어, 다음과 같은 컨트롤러 엔드포인트가 있다고 가정해 보자.

```
@RestController
public class TodoController {

    @GetMapping("/todos/{id}")
    public Todo getTodo(@PathVariable String id) {
        ...
    }
}
```

코드 27.1

이런 기능들을 직접 테스트하는 것은 쉽지 않다. 스프링 프레임워크는 다음처럼 엔드포인트에 대한 HTTP 요청을 처리하기 위해 많은 작업을 대신 수행하기 때문에 테스트 코드에서 TodoController 객체의 getTodo 메서드를 직접 호출하는 것은 사실 크게 의미가 없다.

- HTTP 요청 메서드 중 GET 요청에 대한 식별
- '/todos/{id}' 형태의 경로로 오는 요청을 해당 메서드로 연결
- 매번 바뀌는 id 변수에 매칭되는 값을 추출하여 메서드 인자로 전달

이러한 이유로 스프링 프레임워크는 신뢰성 높은 테스트를 위해 MockMvc 같은 테스트를 지원하는 많은 기능을 제공한다. MockMvc는 웹 계층에서 발생하는 요청과 응답을 개발자가 시뮬레이션할 수 있게 돕는다.

스프링 시큐리티도 다양한 테스트 시나리오를 지원하기 위한 도구들을 제공하지만, 스프링 시큐리티 역시 스프링 프레임워크 기반으로 동작하기 때문에 스프링 프레임워크가 지원하는 테스트 도구들을 확장할 수 있는 기능들을 제공한다.

27장에선 스프링 부트 프레임워크를 활용해 테스트를 수행할 방법에 대해 알아본다. 다음 스프링 시큐리티의 핵심 기능인 인증과 인가(HTTP 기반 혹은 메서드 기반)를 사용하는 비즈니스 기능에 대한 테스트를 지원하는 방법을 살펴본다. 마지막으로 OAuth2 클라이언트 기능을 사용할 때 이를 테스트하는 방법을 정리한다.

- 단위 테스트와 테스트 더블
- 결합 테스트와 스프링 부트 테스트
- 메서드 기반 인가 지원
- 스프링 MVC 테스트 지원 - 기본
- 스프링 MVC 테스트 지원 - OAuth2

이번 장에선 실습 코드를 함께 살펴본다. 예제에선 다음과 같은 의존성을 사용한다.

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-oauth2-client'
    implementation 'org.springframework.boot:spring-boot-starter-security'
    implementation 'org.springframework.boot:spring-boot-starter-web'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
    testImplementation 'org.springframework.security:spring-security-test'
    testRuntimeOnly 'org.junit.platform:junit-platform-launcher'
}
```

코드 27.2

코드 27.2의 의존성들 중 스프링 시큐리티 테스트 지원과 관련된 것은 다음과 같다.

```
testImplementation 'org.springframework.security:spring-security-test'
```

코드 27.3

27.1. 단위 테스트와 테스트 더블

스프링 시큐리티의 테스트 지원 방법에 대해 이야기하기 전에 테스트 코드에 대한 개념을 정리할 필요가 있다.

27.1.1. 단위 테스트

소프트웨어 개발자라면 단위 테스트(unit test)에 관한 이야기를 많이 들었을 것이다. 단위 테스

트는 소프트웨어 개발에서 가장 작은 단위인 개별 모듈, 함수, 메서드, 클래스 등을 독립적으로 검증하는 테스트 방법이다. 아주 작은 단위의 기능이 예상대로 동작하는지 확인하는 테스트 코드이다. 단위 테스트 코드를 작성하는 목적은 다음과 같다.

- 개별 기능의 정확성을 검증한다. 특정 메서드(함수)가 올바르게 작동하는지 확인한다. 이는 개발자가 조기에 코드의 결함을 발견할 수 있도록 도움을 준다.
- 리팩토링의 안정성을 보장한다. 변경한 코드의 문제가 있는지, 기존 기능들이 올바르게 동작하는지 확인하는 데 유용하다. 작은 단위에서 오류 가능 여부를 발견하기 때문에 버그를 추적하고 해결하기가 쉽다.
- 단위 테스트는 실행 속도가 매우 빠르다. 코드를 수정한 직후에 즉시 결과를 확인할 수 있다. 이는 개발자에게 빠른 피드백 루프를 제공하여 개발자들의 개발 속도와 디버깅 속도를 높인다.

실제 소프트웨어는 여러 객체(혹은 기능)들이 서로 협업한다. 단위 테스트의 개념은 작은 단위의 기능을 테스트하는 것인데, 다른 객체와 협업하는 객체의 단위 기능을 어떻게 다른 협업 객체로부터 격리할 수 있을까? 여기서 단위 테스트를 위한 간단한 테크닉이 필요하다.

27.1.2. 테스트 더블

앞서 말한 테크닉을 위해선 테스트 더블이라는 도구가 필요하다. 테스트 더블은 영화 산업에서 위험한 장면을 촬영할 때 배우의 대역을 의미하는 스턴트 더블에서 유래했다. 한국에선 흔히 이를 스턴트맨이라고 부른다. 테스트 더블을 사용하면 어려운 테스트를 쉽게 만들 수 있다. 예를 들어, 다음과 같은 구현 코드가 있다.

- UserService 객체는 JpaUserRepository 객체를 통해 사용자 정보를 조회한다.
- JpaUserRepository 객체는 DB에서 데이터를 조회한다.

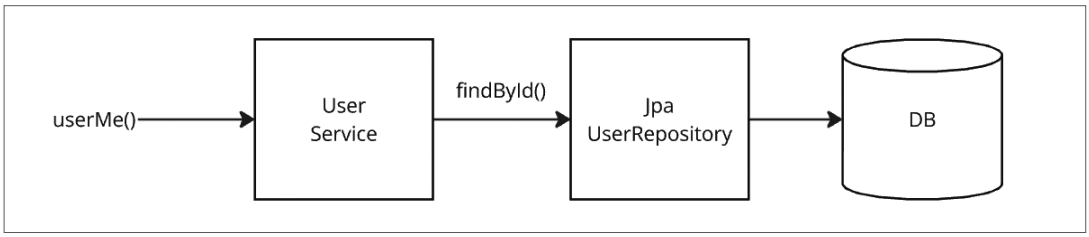

그림 27.1

위 기능은 다음과 같은 관점에서 테스트가 어렵다.

- 데이터베이스를 사용하기 때문에 테스트 결과가 데이터 상황의 영향을 받는다. 테이터의 변경이 발생하면 테스트가 실패한다.

단위 테스트는 어느 상황에서도 성공해야 하지만, findById 메서드가 데이터베이스에 의존하고 있기 때문에 항상 같은 결과를 반환한다는 보장이 없다. 이를 어떻게 해결할 수 있을까? 이런 상황에선 JpaUserRepository 객체를 항상 같은 결과를 반환하는 테스트 더블 객체로 대체한다. 테스트 더블은 항상 고정된 값을 응답하기 때문에 테스트 코드는 항상 성공한다.

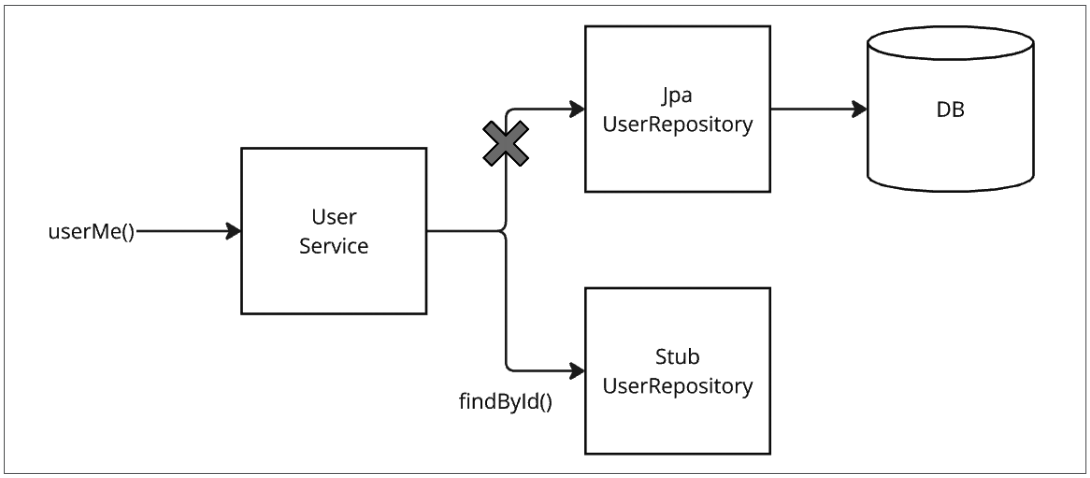

그림 27.2

항상 고정된 결과를 반환하는 응답으로 테스트한다는 사실이 어색할 수 있지만, 중요한 것은 테스트 대상 메서드는 UserService 객체의 userMe 메서드라는 점이다. userMe 메서드는 다음과 같은 기능을 수행한다고 가정한다.

- 엔티티 객체를 모델 객체로 변환
- 사용자 정보가 없는 경우 적절한 예외 처리

단위 테스트는 위 userMe 메서드가 맡은 책임에 대한 기능을 검증하는 것에 초점을 맞춘다. 반환한 데이터에 어떤 값들이 있는지는 중요하지 않다. 위 두 가지 책임을 잘 수행하는지 확인하기 위한 테스트 코드를 작성하고, 테스트 더블의 응답을 각 단위 테스트마다 적절하게 설정한다.

테스트 더블을 효과적으로 사용하기 위해선 UserService 객체는 실행 환경에 따라 적절한 구현체 객체와 협력할 수 있도록 인터페이스 같은 추상화된 인터페이스에 의존해야 한다. 이를 통해 UserService 객체 입장에선 동일한 모습을 한 객체에 일을 맡기는 것이 가능하다. 영화를 보는 관객이 스턴트맨의 액션을 실제 배우의 액션으로 착각하고 보는 것과 동일하다는 생각이 들지 않는가?

참고 **의존성 역전 원칙(Dependency Inversion Principle, DIP)**

객체 지향 프로그래밍에서 SOLID 원칙 중 하나로 구현이 아닌 추상화에 의존하도록 설계하라는 원칙이다. 고수준 모듈(high-level module)과 저수준 모듈(low-level module) 간의 의존성을 뒤집어, 고수준 모듈이 저수준 모듈에 직접 의존하지 않도록 하는 것이다. 이를 통해 시스템의 유연성과 확장성을 향상시킬 수 있다.

- 고수준 모듈인 UserService 객체가 저수준 모듈인 JpaUserRepository 객체에 직접 의존하지 않고, 추상화된 UserRepository 인터페이스에 의존한다.
- 저수준 모듈인 JpaUserRepository, StubUserRepository 객체는 추상화된 UserRepository 인터페이스에 의존한다.
- 고수준 모듈이 저수준 모듈에 직접 의존하는 관계를 뒤집어 고수준, 저수준 모듈 모두 추상화 계층에 의존하도록 설계하는 것을 의존성의 방향을 역전했다고 표현한다.

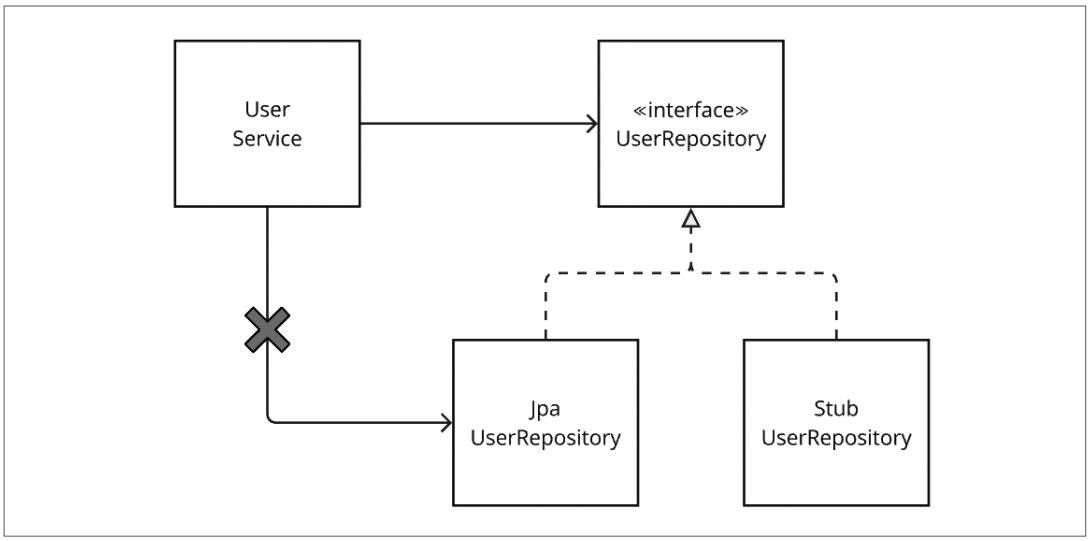

그림 27.3

27.1.3. 테스트 더블의 종류

테스트 더블은 수행하는 역할에 따라 다음과 같이 크게 다섯 종류이다.

- 더미
- 스텁
- 스파이
- 페이크
- 목

간단한 예제 코드를 통해 개념을 짚어본다. 다음과 같은 주문 결제 기능이 있다. OrderService 객체는 추상화된 OrderRepository 인터페이스에 의존하고 있다.

```
package com.example.testdouble.service;

import com.example.testdouble.domain.Order;
import com.example.testdouble.domain.User;
import com.example.testdouble.repository.OrderRepository;

import java.util.UUID;

public class OrderService {
    private final OrderRepository orderRepository;

    public OrderService(OrderRepository orderRepository) {
        this.orderRepository = orderRepository;
    }

    public UUID placeOrder(User user, Order order) {
        if (!user.isAdmin()) {
            throw new RuntimeException("only admins can place orders");
        }
        return orderRepository.placeOrder(order);
    }

    public Order findById(UUID id) {
        var order = orderRepository.findById(id);
        if (order == null) {
            throw new RuntimeException("Order not found");
        }
        return order;
    }
}
```

코드 27.4

처음 살펴볼 테스트 더블은 더미이다. 더미는 아무런 동작을 하지 않고, 컴파일 에러 등의 문제를 해결하기 위해 자리를 채우는 용도로 사용된다. 메서드 시그니처를 맞추기 위해 의미 없는 객체를 전달하는 것이다. 아래 단위 테스트는 관리자가 아닌 경우 예외가 발생하는지 확인한다. 테스트에서 검증하는 기능에 OrderRepository 인스턴스나 Order 객체는 필요하지 않지만, 없는 경우 컴파일 에러가 발생하기 때문에 인자로 전달한다.

```
class DummyOrderRepository implements OrderRepository {

    @Override
    public UUID placeOrder(Order order) {
        return null;
    }

    @Override
    public Order findById(UUID id) {
        return null;
    }
}

public class DummyCaseTest {

    @Test
    void givenNotAdmin_whenPlaceOrder_thenThrowException() {
        OrderService sut = new OrderService(new DummyOrderRepository());

        assertThrows(RuntimeException.class, () -> {
            sut.placeOrder(
                    new User("junhyunny", "ROLE_USER"),
                    new Order()
            );
        });
    }
}
```

코드 27.5

스텁은 고정된 결과를 반환하도록 미리 설정된 객체다. 스텁을 활용하면 개발자는 테스트 코드에서 원하는 조건이나 결과를 제어할 수 있다.

```
class StubOrderRepository implements OrderRepository {

    UUID returnOrderId;

    public void setReturnOrderId(UUID returnOrderId) {
        this.returnOrderId = returnOrderId;
    }

    @Override
    public UUID placeOrder(Order order) {
        return returnOrderId;
    }

    @Override
    public Order findById(UUID id) {
        return null;
    }
}

public class StubCaseTest {

    @Test
    void givenAdmin_whenPlaceOrder_thenReturnOderId() {
        StubOrderRepository stub = new StubOrderRepository();
        stub.setReturnOrderId(UUID.fromString("a1b3360b-687e-4491-a6ca-
f8f2d1474b6b"));
        OrderService sut = new OrderService(stub);

        var result = sut.placeOrder(
                new User("junhyunny", "ROLE_ADMIN"),
                new Order()
        );
```

```
        assertEquals(
                UUID.fromString("a1b3360b-687e-4491-a6ca-f8f2d1474b6b"),
                result
        );
    }
}
```

코드 27.6

스파이는 특정 메서드의 동작 여부를 감시할 수 있는 객체이다. 단위 기능 내부에서 특정 메서드가 어떤 인수로 호출되었는지 확인할 수 있다. 단위 기능의 일부 동작을 검증할 때 사용한다.

```
class SpyOrderRepository implements OrderRepository {

    int placeOrderCalledTimes;
    Order argumentPlaceOrder;

    @Override
    public UUID placeOrder(Order order) {
        placeOrderCalledTimes++;
        argumentPlaceOrder = order;
        return UUID.randomUUID();
    }

    @Override
    public Order findById(UUID id) {
        return null;
    }
}

public class SpyCaseTest {

    @Test
    void givenAdmin_whenPlaceOrder_thenCallPlaceOrderOfRepository() {
        SpyOrderRepository spy = new SpyOrderRepository();
```

```
        OrderService sut = new OrderService(spy);

        sut.placeOrder(
                new User("junhyunny", "ROLE_ADMIN"),
                new Order(1000)
        );

        assertEquals(
                1,
                spy.placeOrderCalledTimes
        );
        assertEquals(
                new Order(1000),
                spy.argumentPlaceOrder
        );
    }
}
```

코드 27.7

목은 스파이와 스텁의 역할을 동시에 수행하는 객체를 의미한다. 스파이처럼 특정 행위를 검증하고, 스텁처럼 준비된 응답을 반환하는 스펙트럼이 넓은 테스트 더블이다. 아래 예제 코드는 Mockito 라이브러리를 사용해 목 객체를 만들었다.

```
public class MockCaseTest {

    @Test
    void givenAdmin_whenPlaceOrder_thenReturnOderId() {
        OrderRepository mock = Mockito.mock(OrderRepository.class);
        Mockito.when(
                mock.placeOrder(any())
        ).thenReturn(
                UUID.fromString("a1b3360b-687e-4491-a6ca-f8f2d1474b6b")
        );
        OrderService sut = new OrderService(mock);
```

```
        var result = sut.placeOrder(
                new User("junhyunny", "ROLE_ADMIN"),
                new Order()
        );

        assertEquals(
                UUID.fromString("a1b3360b-687e-4491-a6ca-f8f2d1474b6b"),
                result
        );
    }

    @Test
    void givenAdmin_whenPlaceOrder_thenCallPlaceOrderOfRepository() {
        OrderRepository mock = Mockito.mock(OrderRepository.class);
        OrderService sut = new OrderService(mock);

        sut.placeOrder(
                new User("junhyunny", "ROLE_ADMIN"),
                new Order(1000)
        );

        verify(mock, times(1))
                .placeOrder(new Order(1000));
    }
}
```

코드 27.8

페이크는 실제 구현과 유사하게 동작하는 테스트 더블이다. 하지만 실제 환경에서 사용할 수 없다. 아래 예시는 데이터베이스와 유사하게 동작하도록 HashMap 객체로 구현한 페이크이다.

```
class FakeOrderRepository implements OrderRepository {
    HashMap<UUID, Order> orders = new HashMap<>();

    @Override
    public UUID placeOrder(Order order) {
```

```
        return null;
    }

    @Override
    public Order findById(UUID id) {
        return orders.get(id);
    }
}

public class FakeCaseTest {

    @Test
    void givenOrderIsExisted_whenPlaceOrder_thenReturnOrder() {
        var orderId = UUID.fromString("a1b3360b-687e-4491-a6ca-f8f2d1474b6b");
        FakeOrderRepository fake = new FakeOrderRepository();
        fake.orders.put(
                orderId,
                new Order(orderId, 1000)
        );
        OrderService sut = new OrderService(fake);

        var result = sut.findById(orderId);

        assertEquals(
                new Order(orderId, 1000),
                result
        );
    }

    @Test
    void givenOrderIsNotExisted_whenPlaceOrder_thenThrowException() {
        var orderId = UUID.fromString("a1b3360b-687e-4491-a6ca-f8f2d1474b6b");
        FakeOrderRepository fake = new FakeOrderRepository();
        OrderService sut = new OrderService(fake);
```

```
        assertThrows(RuntimeException.class, () -> {
            sut.findById(orderId);
        });
    }
}
```

코드 27.9

이 책에서 자주 사용하는 H2 인-메모리 데이터베이스도 대표적인 페이크이다.

27.2. 결합 테스트와 스프링 부트 테스트

27.2.1. 결합 테스트

앞서 말했듯 소프트웨어는 여러 객체들의 협업을 통해 동작한다. 단위 테스트를 통해 작은 단위의 기능을 검증하는 것만으로 소프트웨어에 문제가 없다는 것을 보증할 순 없다. 모듈 간의 인터페이스나 데이터 교환 과정에서 예상치 못한 문제가 발생할 수 있다. 이런 문제를 해결하기 위해 결합 테스트를 수행한다.

결합 테스트는 소프트웨어 개발 과정에서 개별적으로 테스트된 모듈(단위 테스트로 검증된 모듈)들을 서로 결합하여 상호작용이 정상적으로 이루어지는지 확인하는 테스트다. 다음과 같은 목적을 갖는다.

- 모듈 간 상호작용을 확인한다. 단위 테스트에서 각 모듈이 개별적으로 정상적으로 작동하더라도, 모듈 간 상호작용에 문제가 있을 수 있다. 결합 테스트는 이 상호작용이 제대로 이루어지는지 확인한다.
- 데이터 흐름을 검증한다. 하나의 모듈에서 다른 모듈로 데이터를 전달할 때, 데이터가 올바르게 전달되고 사용되는지 검증한다. 이는 데이터 형식의 일관성, 데이터 변환, 데이터 유효성 검증 등이 포함될 수 있다.
- 외부 시스템과의 통합을 검증한다. 결합 테스트는 데이터베이스, API, 메시징 시스템 등과 같은 외부 시스템과의 통합을 테스트하는 데에도 사용된다. 이는 외부 시스템과 예상대로 상호작용하고, 통신이 잘 이루어지는지 확인하는 데 도움을 준다.
- 의존성 및 구성 요소 연결을 확인한다. 모듈들 간의 의존성 설정이 올바르게 구성되었는지, 필요한 구성 요소가 제대로 연결되었는지 확인한다.

참고 **테스트의 종류**

실제 구성 요소를 통합하는 정도에 따라 테스트를 구분한다. 일반적으로 일부는 실제 모듈, 컨트롤하기 어려운 외부 종속성에 테스트 더블을 사용한다면 결합 테스트로 구분하는 것으로 보인다. 인프라, 애플리케이션, 캐시, 데이터 스토리지 등 시스템을 구성하는 모든 컴포넌트를 연결한 채로 테스트하는 것은 E2E(End-To-End) 테스트라고 한다.

27.2.2. 스프링 부트 테스트

스프링 프레임워크와 그 생태계는 개발자들의 쉬운 개발을 위해 많은 것들을 해준다.

- HTTP 요청/응답을 위한 엔드포인트 생성
- HTTP 통신을 위한 클라이언트 기능 제공
- 캐시, 데이터베이스과 같은 데이터 저장소 연결 및 통신
- 사용자 인증 및 권한 부여
- 객체 관리 및 의존성 연결

이 책에서 다룬 기능들을 기준으로 스프링 부트 프레임워크의 컴포넌트를 크게 나눠보면 다음과 같은 모습을 갖는다.

- HTTP 요청은 필터 체인, 디스패처 서블릿, 핸들러 맵핑, 핸들러 어댑터를 지나 컨트롤러 레이어까지 전달된다.
- 리포지토리 레이어는 JPA, 하이버네이트를 통해 데이터베이스와 통신한다.
- 시큐리티 필터 체인은 필터 체인 중간 프록시 필터에 의해 연결된다.

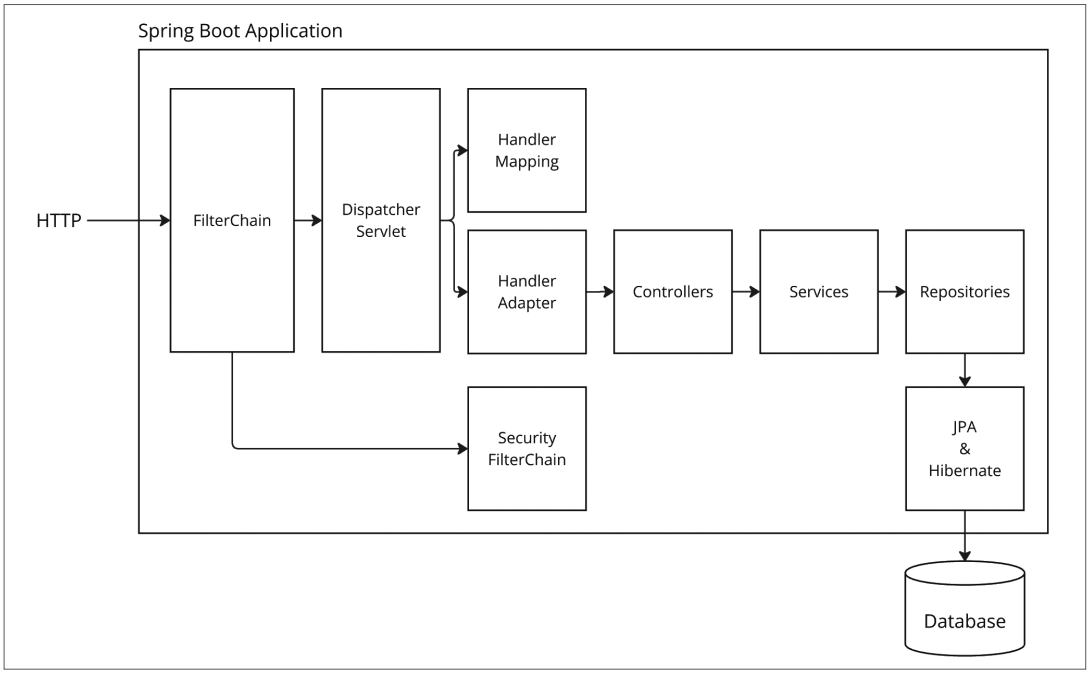

그림 27.4

프레임워크는 애플리케이션이 제공하는 서비스를 다양한 기능으로 지원한다. 그래서 애플리케이션이 정상적으로 동작하는지 확인하기 위해선 프레임워크의 도움이 필요하다. 스프링 부트 프레임워크는 프레임워크가 제공하는 기능을 함께 결합하여 테스트할 수 있도록 몇 가지 애너테이션을 제공한다.

- @WebMvcTest
- @DataJpaTest
- @SpringBootTest

아래 예제 코드처럼 테스트 클래스 위에 추가한다.

```
@SpringBootTest
class DemoApplicationTests {
    ...
}
```

코드 27.10

각 애너테이션은 사용할 때 테스트 컨텍스트에 준비되는 컴포넌트 범위가 다르다.

- @WebMvcTest 애너테이션은 필터 체인, 디스패처 서블릿, 핸들러 맵핑, 핸들러 어댑터, 컨트롤러까지 테스트 컨텍스트에 준비한다. 여기에 시큐리티 필터 체인은 포함되지 않는다. 시큐리티 필터 체인이 추가되지 않으면 HTTP 기반 인증, 인가 작업의 적용 여부를 알 수 없다.
- @DataJpaTest 애너테이션은 데이터베이스와 함께 결합 테스트를 하기 위해 JPA, Hibernate 관련된 의존성을 테스트 컨텍스트에 준비한다. 데이터베이스와 관련된 테스트를 준비하기 때문에 테스트 컨텍스트에 시큐리티 필터 체인이 포함되지 않는다.
- @SpringBootTest 애너테이션은 실제 애플리케이션이 동작하는 것과 동일한 환경을 구성한다. 테스트 컨텍스트에 시큐리티 필터 체인이 포함된다.

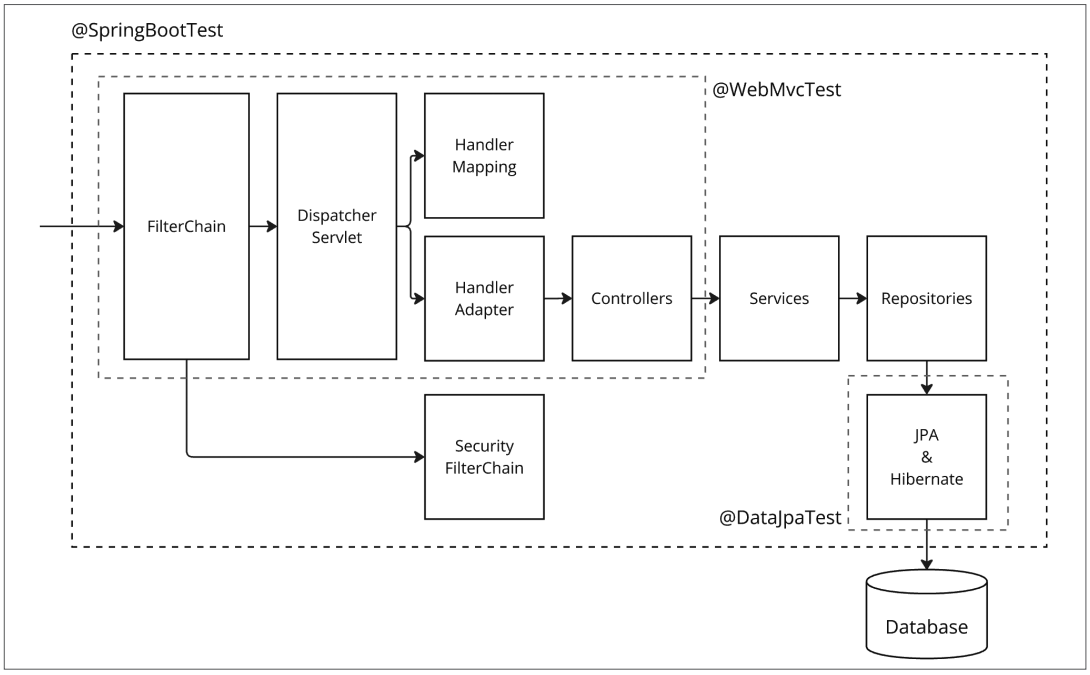

그림 27.5

스프링 시큐리티의 인증, HTTP 기반 인가 처리는 시큐리티 필터 체인에 포함된다. 테스트 환경에서 이를 함께 검증하기 위해선 @WebMvcTest 애너테이션에 별도 작업을 통해 시큐리티 필터 체인을 추가하거나 @SpringBootTest 애너테이션을 사용하면 된다.

메서드 기반 인가 처리가 포함된 기능을 테스트하기 위해선 프레임워크의 AOP 기능이 필요하다. AOP 기능을 위한 프록시는 bean 객체를 사용해야 활성화하기 때문에 @SpringBootTest 애너테이션을 사용해 테스트 컨텍스트를 준비한다.

27.3. 메서드 기반 인가 지원

다음과 같은 기능을 테스트하고 싶다.

[1] 인증된 사용자는 다른 사용자 정보를 조회할 수 있다.

[2] 인증된 사용자 정보를 확인하기 위한 로그를 출력한다.

[3] 사용자 정보를 반환한다.

```
package com.example.demo.service;

import com.example.demo.domain.User;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.stereotype.Service;

@Service
public class DefaultUserService implements UserService {

    /* [1] 메소드 기반 인가 처리 - 인증된 사용자만 접근 가능 */
    @PreAuthorize("isAuthenticated()")
    @Override
    public User getUser(String username) {
        var authentication = SecurityContextHolder
                .getContext()
                .getAuthentication();
        /* [2] 로그 출력 */
        System.out.println(authentication);
        /* [3] 사용자 정보 반환 */
        return new User(username, "HR");
    }
}
```

코드 27.11

테스트를 위해선 메서드 기반 인가 처리 활성화가 필요하다. 애플리케이션 메인 메서드가 위치한 클래스 위에 @EnableMethodSecurity 애너테이션을 설정한다.

```
@EnableMethodSecurity
@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }

}
```

코드 27.12

27.3.1. @WithMockUser 애너테이션

위 메서드는 인증된 사용자만 호출할 수 있다. 준비 없이 호출하는 경우 예외가 발생한다.

```
@SpringBootTest
class WithMockUserServiceTest {

    @Autowired
    UserService sut;

    ...

    @Test
    void givenNotAuthenticated_whenGetUser_thenThrowException() {
        assertThrows(
                AuthenticationCredentialsNotFoundException.class,
                () -> sut.getUser("junhyunny")
        );
    }
}
```

코드 27.13

인증된 사용자를 SecurityContextHolder 클래스에 직접 준비해도 되지만, @WithMockUser 애너테이션을 사용하면 더 쉽게 테스트 컨텍스트를 준비할 수 있다.

• 예외가 발생하지 않고 의도한 User 객체를 반환한다.

```
@SpringBootTest
class WithMockUserServiceTest {

    @Autowired
    UserService sut;

    ...

    @WithMockUser
    @Test
    void givenAuthenticated_whenGetUser_thenReturnUser() {
        assertEquals(
                new User("junhyunny", "HR"),
                sut.getUser("junhyunny")
        );
    }
}
```

코드 27.14

다음과 같은 실행 로그를 볼 수 있다. @WithMockUser 애너테이션을 통해 준비된 사용자 정보를 확인할 수 있다.

- Authentication 인스턴스 타입은 UsernamePasswordAuthenticationToken 클래스이다.
- Principal 인스턴스 타입은 org.springframework.security.core.userdetails.User 클래스이다.
- 사용자 이름은 'user'이다.
- 로그에 출력되진 않지만, 비밀번호는 'password'이다.
- 역할은 'ROLE_USER'이다.

```
UsernamePasswordAuthenticationToken [Principal=org.springframework.security.
core.userdetails.User [Username=user, Password=[PROTECTED], Enabled=true,
AccountNonExpired=true, CredentialsNonExpired=true, AccountNonLocked=true, Granted
Authorities=[ROLE_USER]], Credentials=[PROTECTED], Authenticated=true, Details=null,
Granted Authorities=[ROLE_USER]]
```

사용자 이름이나 역할을 변경할 수 있다. 테스트 대상 메서드를 다음과 같이 변경해 본다.

- 'ADMIN' 역할을 가진 사용자만 접근할 수 있도록 변경한다.

```
@Service
public class DefaultUserService implements UserService {

    /* ADMIN 사용자만 접근 가능 */
    @PreAuthorize("hasRole('ADMIN')")
    @Override
    public User getUser(String username) {
        var authentication = SecurityContextHolder
                .getContext()
                .getAuthentication();
        System.out.println(authentication);
        return new User(username, "HR");
    }
}
```

코드 27.15

테스트 사용자를 준비할 때 이름과 역할을 변경한다.

- 사용자 이름은 'admin', 역할은 'ADMIN'이다.

```
@SpringBootTest
class WithMockUserServiceTest {

    @Autowired
    UserService sut;

    ...

    /* 인증된 목 사용자 준비 */
    @WithMockUser(username = "admin", roles = {"ADMIN"})
    @Test
    void givenAdmin_whenGetUser_thenReturnUser() {
```

```
        assertEquals(
                new User("junhyunny", "HR"),
                sut.getUser("junhyunny")
        );
    }
}
```

코드 27.16

매번 메서드마다 지정하는 것이 번거로울 수 있다. @WithMockUser 애너테이션은 클래스 레벨에 지정하는 것도 가능하다.

```
@WithMockUser(username = "admin", roles = {"ADMIN"})
@SpringBootTest
class WithMockUserServiceTest {
    ...
}
```

코드 27.17

27.3.2. @WithAnonymousUser 애너테이션

@WithAnonymousUser 애너테이션을 사용하면 시큐리티 컨텍스트에 익명 사용자를 준비할 수 있다.

[1] 시큐리티 컨텍스트에 익명 사용자를 준비한다.

[2] 시큐리티 컨텍스트에 익명 사용자가 존재하기 때문에 AuthorizationDeniedException 예외가 발생한다.

```
@SpringBootTest
class WithAnonymousUserTest {

    @Autowired
    UserService sut;

    /* [1] 익명 목 사용자 준비 */
    @WithAnonymousUser
    @Test
```

```
    void givenAnonymous_whenGetUser_thenThrowException() {
        /* [2] 인가 거부 예외 발생 */
        assertThrows(
                AuthorizationDeniedException.class,
                () -> sut.getUser("junhyunny")
        );
    }
}
```

코드 27.18

@WithAnonymousUser 애너테이션을 사용하면 시큐리티 컨텍스트에 다음과 같은 정보가 담긴다.

- Authentication 인스턴스 타입은 AnonymousAuthenticationToken 클래스이다.
- Principal 인스턴스는 "anonymous" 문자열이다.
- 역할은 'ROLE_ANONYMOUS'이다.

27.3.3. @WithUserDetails 애너테이션

인증 과정에 UserDetailsService 인스턴스를 사용하고, 이와 함께 결합 테스트를 수행하고 싶다면 @WithUserDetails 애너테이션이 도움이 된다. @WithUserDetails 애너테이션을 사용하면 UserDetailsService 인스턴스의 loadUserByUsername 메서드를 호출할 때 인자로 전달되는 사용자 이름을 제어할 수 있다.

예를 들어, 다음과 같은 UserDetailsService 인스턴스가 있다고 가정한다.

```
package com.example.demo.service;

import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

@Service
public class DefaultUserDetailsService implements UserDetailsService {
```

```
    @Override
    public UserDetails loadUserByUsername(
            String username
    ) throws UsernameNotFoundException {
        return User.withDefaultPasswordEncoder()
                .username(username)
                .password("12345")
                .roles("ADMIN")
                .authorities("READ::ALL", "WRITE::ALL")
                .build();
    }
}
```

코드 27.19

다음과 같이 테스트 코드에서 @WithUserDetails 애너테이션을 사용한다.

```
@SpringBootTest
class WithUserDetailsTest {

    @Autowired
    UserService sut;

    ...

    @Test
    @WithUserDetails
    void whenGetUser_thenReturnUser() {
        assertEquals(
                new User("junhyunny", "HR"),
                sut.getUser("junhyunny")
        );
    }
}
```

코드 27.20

로그를 확인해 보면 DefaultUserDetailsService 객체에서 반환한 객체가 인증 주체로 담겨 있는 것을 확인할 수 있다.

```
UsernamePasswordAuthenticationToken [Principal=org.springframework.security.
core.userdetails.User [Username=user, Password=[PROTECTED], Enabled=true,
AccountNonExpired=true, CredentialsNonExpired=true, AccountNonLocked=true, Granted
Authorities=[READ::ALL, WRITE::ALL]], Credentials=[PROTECTED], Authenticated=true,
Details=null, Granted Authorities=[READ::ALL, WRITE::ALL]]
```

메서드 인자로 전달되는 사용자 이름을 변경하거나 bean 객체를 직접 지정하는 것도 가능하다.

```
@SpringBootTest
class WithUserDetailsTest {

    @Autowired
    UserService sut;

    ...

    @Test
    @WithUserDetails(
        value = "admin",
        userDetailsServiceBeanName = "defaultUserDetailsService"
    )
    void givenAdmin_whenGetUser_thenReturnUser() {
        assertEquals(
                new User("junhyunny", "HR"),
                sut.getUser("junhyunny")
        );
    }
}
```

코드 27.21

로그를 확인하면 사용자 이름이 'admin'으로 변경된 것을 확인할 수 있다.

```
UsernamePasswordAuthenticationToken [Principal=org.springframework.security.
core.userdetails.User [Username=admin, Password=[PROTECTED], Enabled=true,
AccountNonExpired=true, CredentialsNonExpired=true, AccountNonLocked=true, Granted
Authorities=[READ::ALL, WRITE::ALL]], Credentials=[PROTECTED], Authenticated=true,
Details=null, Granted Authorities=[READ::ALL, WRITE::ALL]]
```

@WithMockUser 애너테이션과 달리 @WithUserDetails 애너테이션은 UserDetailsService 인스턴스를 통해 사용자를 실제로 조회하기 때문에 데이터 소스에 사용자 데이터가 준비되어 있어야 한다.

27.3.4. @WithSecurityContext 애너테이션

@WithSecurityContext 애너테이션을 사용하면 자유도가 더 높아진다. 시큐리티 컨텍스트에 저장되는 Authentication 인스턴스나 Principal 인스턴스를 원하는 대로 변경할 수 있다. @WithSecurityContext 애너테이션을 사용해 RememberMeAuthenticationToken 객체가 담긴 시큐리티 컨텍스트를 준비하는 커스텀 애너테이션을 만들자.

[1] 커스텀 시큐리티 컨텍스트를 생성하는 팩토리 클래스를 지정한다.

[2] 기본 사용자 이름은 'user'로 지정한다.

[3] 기본 사용자 역할을 'USER'로 지정한다.

```
package com.example.demo.meta;

import com.example.demo.factory.WithRememberMeUserFactory;
import org.springframework.security.test.context.support.WithSecurityContext;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)
/* [1] 커스텀 시큐리티 컨텍스트를 생성하는 팩토리 클래스 지정 */
@WithSecurityContext(factory = WithRememberMeUserFactory.class)
public @interface WithRememberMeUser {
```

```
    /* [2] 사용자 이름 지정 */
    String username() default "user";

    /* [3] 사용자 역할 지정 */
    String[] roles() default {"USER"};
}
```

코드 27.22

WithRememberMeUserFactory 클래스는 WithSecurityContextFactory 인터페이스를 구현한다. WithRememberMeUserFactory 객체의 createSecurityContext 메서드는 테스트에 필요한 시큐리티 컨텍스트를 생성 후 반환한다.

[1] 코드 27.22에서 만든 @WithRememberMeUser 애너테이션 인스턴스를 주입받는다.

[2] 빈 SecurityContext 인스턴스를 생성한다.

[3] 인증 주체 UserDetails 객체를 만든다. 애너테이션에 지정된 사용자 이름과 역할을 사용한다.

[4] RememberMeAuthenticationToken 객체를 만든다.

[5] SecurityContext 인스턴스에 RememberMeAuthenticationToken 객체를 저장한다.

[6] SecurityContext 인스턴스를 반환한다.

```
package com.example.demo.factory;

import com.example.demo.meta.WithRememberMeUser;
import org.springframework.security.authentication.RememberMeAuthenticationToken;
import org.springframework.security.authentication.UsernamePasswordAuthenticationT
oken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.context.SecurityContext;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.test.context.support.
WithSecurityContextFactory;

public class WithRememberMeUserFactory
```

```
        implements WithSecurityContextFactory<WithRememberMeUser> {

    @Override
    /* [1] 애너테이션 객체를 주입 */
    public SecurityContext createSecurityContext(WithRememberMeUser annotation) {
        /* [2] 빈 컨텍스트 준비 */
        SecurityContext context = SecurityContextHolder.createEmptyContext();
        /* [3] 사용자 정보 준비 */
        UserDetails principal = User.withDefaultPasswordEncoder()
                .username(annotation.username())
                .password("12345")
                .roles(annotation.roles())
                .build();
        /* [4] 인증 토큰 생성 */
        Authentication authentication = new RememberMeAuthenticationToken(
                "custom-remember-key",
                principal,
                principal.getAuthorities()
        );
        /* [5] 인증 토큰 컨텍스트에 추가 */
        context.setAuthentication(authentication);
        /* [6] 컨텍스트 반환 */
        return context;
    }
}
```

코드 27.23

이제 위에서 만든 커스텀 @WithRememberMeUser 애너테이션을 사용해 본다. 다른 애너테이션들과 동일하게 메서드 위에 지정하면 된다.

```
@SpringBootTest
class WithRememberMeTest {

    @Autowired
    UserService sut;
```

```
    @WithRememberMeUser
    @Test
    void whenGetUser_thenReturnUser() {
        assertEquals(
                new User("junhyunny", "HR"),
                sut.getUser("junhyunny")
        );
    }
}
```

코드 27.24

테스트 실행 로그를 확인하면 WithRememberMeUserFactory 객체에서 만든 RememberMeAuthenticationToken 객체가 Authentication 인스턴스로 담겨 있는 것을 볼 수 있다. @WithRememberMeUser 애너테이션에 디폴트값으로 지정된 사용자 이름과 역할이 사용된다.

```
RememberMeAuthenticationToken [Principal=org.springframework.security.core.userdetails.User [Username=user, Password=[PROTECTED], Enabled=true, AccountNonExpired=true, CredentialsNonExpired=true, AccountNonLocked=true, Granted Authorities=[ROLE_USER]], Credentials=[PROTECTED], Authenticated=true, Details=null, Granted Authorities=[ROLE_USER]]
```

@WithRememberMeUser 애너테이션의 속성을 변경할 수 있다.

```
@SpringBootTest
class WithRememberMeTest {

    @Autowired
    UserService sut;

    ...

    @WithRememberMeUser(username = "admin", roles = {"ADMIN"})
    @Test
    void givenAdmin_whenGetUser_thenReturnUser() {
```

```
        assertEquals(
                new User("junhyunny", "HR"),
                sut.getUser("junhyunny")
        );
    }
}
```

코드 27.25

테스트 로그를 확인하면 위에서 변경한 사용자 이름과 역할이 사용된 것을 알 수 있다.

```
RememberMeAuthenticationToken [Principal=org.springframework.security.core.userdetails.
User [Username=admin, Password=[PROTECTED], Enabled=true, AccountNonExpired=true,
CredentialsNonExpired=true, AccountNonLocked=true, Granted Authorities=[ROLE_ADMIN]],
Credentials=[PROTECTED], Authenticated=true, Details=null, Granted Authorities=[ROLE_
ADMIN]]
```

27.3.5. 메타 애너테이션

테스트마다 동일한 사용자를 사용한다면 특정 속성값을 반복적으로 설정하기보단 메타 애너테이션을 만드는 것도 좋은 방법이다. 애플리케이션에서 관리자 사용자에 대한 테스트가 많다면 다음과 같은 애너테이션을 만들어 재사용할 수 있다.

```
package com.example.demo.meta;

import org.springframework.security.test.context.support.WithMockUser;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)
@WithMockUser(value = "admin", roles = "ADMIN")
public @interface WithMockAdmin {
}
```

코드 27.26

27.4. 스프링 MVC 테스트 지원 – 기본

스프링 프레임워크는 개발자가 손쉽게 HTTP 요청, 응답을 위한 엔드포인트를 만들 수 있게 도와준다. 구현한 HTTP 엔드포인트를 테스트할 수 있도록 MockMvc라는 기능을 지원한다. MockMvc 객체는 애플리케이션 웹 계층을 테스트하기 위해 HTTP 요청과 응답에 대한 프레임워크 기능을 시뮬레이션한다. MockMvc 객체를 사용하면 실제 서버를 구동하지 않고도 스프링 MVC 컨트롤러와 관련된 기능을 테스트할 수 있다.

스프링 시큐리티가 적용된 상태로 웹 계층 테스트 코드를 작성하기 위해선 시큐리티 필터 체인이 서블릿 필터 체인에 포함되어야 한다. 요청을 시큐리티 필터 체인으로 연결하기 위해 FilterChainProxy 인스턴스를 필터로써 서블릿 필터 체인에 추가해야 한다.

스프링 부트 프레임워크를 사용하면 가장 쉬운 방법은 @SpringBootTest 애너테이션과 @AutoConfigureMockMvc 애너테이션을 함께 사용하는 것이다. @SpringBootTest 애너테이션만 사용하면 MockMvc 객체 생성을 개발자가 직접 해야 하지만, @AutoConfigureMockMvc 애너테이션을 함께 사용하면 프레임워크가 준비해 주어 쉽게 주입받을 수 있다.

```
@SpringBootTest
@AutoConfigureMockMvc
public class ControllerIntegrationTest {

    @Autowired
    private MockMvc mockMvc;

    ...
}
```

코드 27.27

스프링 시큐리티는 시큐리티 필터 체인에서 인증이나 인가를 제공하기 때문에 HTTP 요청을 처리하는 데 협력하는 컴포넌트들을 테스트 더블로 대체하는 것이 어렵다. 그렇기에 HTTP 요청을 시뮬레이션할 때 필요한 정보를 미리 준비하는 기능을 SecurityMockMvcRequestPostProcessors 클래스를 통해 제공한다. 이번 절에선 HTTP 요청에 필요한 정보를 준비하는 방법에 대해 알아본다.

테스트를 위해 필요한 시큐리티 필터 체인은 다음과 같다.

[1] '/api/admin' 엔드포인트는 'ADMIN' 역할을 갖는 사용자만 접근할 수 있다.

[2] 기타 엔드포인트는 전부 인증된 사용자만 접근할 수 있다.

```
package com.example.demo.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.
EnableWebSecurity;
import org.springframework.security.web.SecurityFilterChain;

@EnableWebSecurity
@Configuration
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(
            HttpSecurity http
    ) throws Exception {
        http.authorizeHttpRequests(
                configurer -> configure
                        /* [1] ADMIN 사용자만 접근할 수 있는 인가 규칙 지정 */
                        .requestMatchers("/api/admin").hasRole("ADMIN")
                        /* [2] 인증된 사용자만 접근할 수 있는 인가 규칙 지정 */
                        .anyRequest().authenticated()
        );
        http.formLogin(
                Customizer.withDefaults()
        );
        http.httpBasic(
                Customizer.withDefaults()
```

```
        );
        return http.build();
    }
}
```

코드 27.28

엔드포인트가 정의된 컨트롤러는 다음과 같다. 모든 엔드포인트는 Authentication 인스턴스 정보를 로그로 출력한다.

[1] GET 요청을 처리하는 '/api/user' 엔드포인트

[2] POST 요청을 처리하는 '/api/user' 엔드포인트

[3] GET 요청을 처리하는 '/api/admin' 엔드포인트

```
package com.example.demo.controller;

import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api")
public class EndpointController {

    /* [1] 사용자 리소스에 대한 GET 요청 처리 */
    @GetMapping("/user")
    public String user() {
        var authentication = SecurityContextHolder
                .getContext()
                .getAuthentication();
        System.out.println(authentication);
        return "ok";
    }

    /* [2] 사용자 리소스에 대한 POST 요청 처리 */
```

```
    @PostMapping("/user")
    public String postUser() {
        var authentication = SecurityContextHolder
                .getContext()
                .getAuthentication();
        System.out.println(authentication);
        return "ok";
    }

    /* [3] 관리자 리소스에 대한 GET 요청 처리 */
    @GetMapping("/admin")
    public String admin() {
        var authentication = SecurityContextHolder
                .getContext()
                .getAuthentication();
        System.out.println(authentication);
        return "ok";
    }
}
```

코드 27.29

27.4.1. 테스트 사용자 지정하기

HTTP 요청을 보낼 때 사용자 정보가 필요한 경우가 있다. 인가 규칙이 적용되어 있거나 엔드포인트 메서드에서 AuthenticatedPrincipal 애너테이션을 사용해 인증 주체를 주입받는 경우를 예로 들 수 있다. 인증된 사용자가 시큐리티 컨텍스트에 준비되어 있지 않다면 '/api/user' 엔드포인트로 보내는 GET 요청은 403 Forbidden 응답을 받는다.

```
package com.example.demo.controller;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.
AutoConfigureMockMvc;
```

```
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.web.servlet.MockMvc;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.
get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.
status;

@SpringBootTest
@AutoConfigureMockMvc
public class UserRequestPostProcessorTest {

    @Autowired
    MockMvc mockMvc;

    @Test
    void whenGetUser_thenIsUnauthorized() throws Exception {
        mockMvc.perform(
                get("/api/user")
        ).andExpect(
                status().isUnauthorized()
        );
    }
}
```

코드 27.30

@WithMockUser 애너테이션을 사용하는 것도 한 가지 옵션이다. @WithMockUser 애너테이션을 사용해서 필요한 사용자 정보를 시큐리티 컨텍스트에 준비한다.

```
@SpringBootTest
@AutoConfigureMockMvc
public class UserRequestPostProcessorTest {

    @Autowired
    MockMvc mockMvc;
```

```
    ...

    @Test
    @WithMockUser
    void givenWithMockUser_whenGetUser_thenIsOk() throws Exception {
        mockMvc.perform(
                get("/api/user")
        ).andExpect(
                status().isOk()
        );
    }
}
```

코드 27.31

@WithMockUser 애너테이션 외에도 MockMvcRequestBuilders 클래스의 with 메서드를 사용하는 방법이 있다. RequestPostProcessor 구현체 인스턴스를 with 메서드의 인자로 전달한다. 스프링 시큐리티는 SecurityMockMvcRequestPostProcessors 클래스를 통해 다양한 방식으로 인증된 사용자를 준비할 수 있게 한다. 아래 코드는 HTTP 요청을 위해 사용자 이름이 'user', 비밀번호가 'password', 역할이 'ROLE_USER'인 사용자를 준비하는 예제이다.

```
package com.example.demo.controller;

...

import static org.springframework.security.test.web.servlet.request.SecurityMockMv
cRequestPostProcessors.user;

@SpringBootTest
@AutoConfigureMockMvc
public class UserRequestPostProcessorTest {

    @Autowired
    MockMvc mockMvc;
```

```
    ...

    @Test
    void givenUser_whenGetUser_thenIsOk() throws Exception {
        mockMvc.perform(
                get("/api/user")
                        .with(user("user"))
        ).andExpect(
                status().isOk()
        );
    }
}
```

코드 27.32

요청 로그를 살펴보면 시큐리티 컨텍스트에 보관된 Authentication 인스턴스는 Username PasswordAuthenticationToken 객체인 것을 확인할 수 있다.

```
UsernamePasswordAuthenticationToken [Principal=org.springframework.security.
core.userdetails.User [Username=user, Password=[PROTECTED], Enabled=true,
AccountNonExpired=true, CredentialsNonExpired=true, AccountNonLocked=true, Granted
Authorities=[ROLE_USER]], Credentials=[PROTECTED], Authenticated=true, Details=null,
Granted Authorities=[ROLE_USER]]
```

위 요청은 쉽게 커스터마이징할 수 있다. 예를 들어, 사용자 이름을 'admin', 비밀번호는 'pass', 역할은 'ROLE_ADMIN'으로 지정한다.

```
@SpringBootTest
@AutoConfigureMockMvc
public class UserRequestPostProcessorTest {

    @Autowired
    MockMvc mockMvc;

    ...

    @Test
```

```
    void givenAdmin_whenGetAdmin_thenIsOk() throws Exception {
        mockMvc.perform(
                get("/api/admin")
                        .with(
                                user("admin")
                                        .password("pass")
                                        .roles("ADMIN")
                        )
        ).andExpect(
                status().isOk()
        );
    }
}
```

코드 27.33

준비된 UserDetails 인스턴스가 있다면 이를 직접 사용하는 것도 좋은 방법이다. UserDetails 인스턴스는 UsernamePasswordAuthenticationToken 객체의 인증 주체로 동작한다.

```
@SpringBootTest
@AutoConfigureMockMvc
public class UserRequestPostProcessorTest {

    @Autowired
    MockMvc mockMvc;

    UserDetails userDetails = User.withDefaultPasswordEncoder()
            .username("user")
            .password("password")
            .roles("USER")
            .build();

    ...

    @Test
    void givenUserDetails_whenGetUser_thenIsOk() throws Exception {
```

```
        mockMvc.perform(
                get("/api/user")
                        .with(user(userDetails))
        ).andExpect(
                status().isOk()
        );
    }
}
```

코드 27.34

anonymous 메서드를 사용하면 인증이 되지 않은 익명 사용자도 지정할 수 있다.

```
package com.example.demo.controller;

...

import static org.springframework.security.test.web.servlet.request.SecurityMockMv
cRequestPostProcessors.anonymous;

@SpringBootTest
@AutoConfigureMockMvc
public class UserRequestPostProcessorTest {

    @Autowired
    MockMvc mockMvc;

    ...

    @Test
    void givenAnonymous_whenGetUser_thenIsUnauthorized() throws Exception {
        mockMvc.perform(
                get("/api/user")
                        .with(anonymous())
        ).andExpect(
                status().isUnauthorized()
```

```
        );
    }
}
```

코드 27.35

authentication 메서드와 securityContext 메서드를 사용하면 Authenticiation 인스턴스나 시큐리티 컨텍스트를 직접 지정하는 것도 가능하다.

```
package com.example.demo.controller;

...

import static org.springframework.security.test.web.servlet.request.SecurityMockMv
cRequestPostProcessors.authentication;
import static org.springframework.security.test.web.servlet.request.SecurityMockMv
cRequestPostProcessors.securityContext;

@SpringBootTest
@AutoConfigureMockMvc
public class UserRequestPostProcessorTest {

    @Autowired
    MockMvc mockMvc;

    UserDetails userDetails = User.withDefaultPasswordEncoder()
            .username("user")
            .password("password")
            .roles("USER")
            .build();

    Authentication rememberMeAuthentication = new RememberMeAuthenticationToken(
            "remember-me-key",
            userDetails,
            userDetails.getAuthorities()
    );
```

```
    SecurityContext securityContext = new SecurityContextImpl(rememberMeAuthentica
tion);

    ...

    @Test
    void givenAuthentication_whenGetUser_thenIsOk() throws Exception {
        mockMvc.perform(
                get("/api/user")
                        .with(authentication(rememberMeAuthentication))
        ).andExpect(
                status().isOk()
        );
    }

    @Test
    void givenSecurityContext_whenGetUser_thenIsOk() throws Exception {
        mockMvc.perform(
                get("/api/user")
                        .with(securityContext(securityContext))
        ).andExpect(
                status().isOk()
        );
    }
}
```

코드 27.36

자주 사용되는 사용자라면 메서드로 정의하는 것이 편리하다. 예를 들어, 다음과 같이 사용자 이름이 'junhyunny', 역할은 'ADMIN'인 사용자를 반환하는 정적 메서드를 만든다. 정적 메서드의 이름은 사용자를 식별할 수 있도록 사용자 이름으로 지정한다.

```
package com.example.demo.controller;

...
```

```
import org.springframework.test.web.servlet.request.RequestPostProcessor;

class CustomRequestProcessor {

    public static RequestPostProcessor junhyunny() {
        return user("junhyunny").roles("ADMIN");
    }
}

@SpringBootTest
@AutoConfigureMockMvc
public class UserRequestPostProcessorTest {

    @Autowired
    MockMvc mockMvc;

    ...

    @Test
    void givenJunhyunny_whenGetAdmin_thenIsOk() throws Exception {
        mockMvc.perform(
                get("/api/admin")
                        .with(CustomRequestProcessor.junhyunny())
        ).andExpect(
                status().isOk()
        );
    }
}
```

코드 27.37

27.4.2. CSRF 보호 지원

시큐리티 필터 체인은 안전하지 않은 HTTP 요청을 보낼 때 CSRF 보호가 적용된다. 그런 경우 CSRF 토큰이 요청 헤더나 파라미터에 담겨 있어야 한다. 요청에 CSRF 토큰 정보가 없다면 요청이 거부된다.

```
package com.example.demo.controller;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.
AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.web.servlet.MockMvc;

import static org.springframework.security.test.web.servlet.request.SecurityMockMv
cRequestPostProcessors.user;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.
post;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.
status;



@SpringBootTest
@AutoConfigureMockMvc
public class CsrfRequestPostProcessorTest {

    @Autowired
    MockMvc mockMvc;

    @Test
    void givenWithoutCsrfToken_whenPostUser_thenIsForbidden() throws Exception {
        mockMvc.perform(
                post("/api/user")
```

```
                        .with(user("user"))
        ).andExpect(
                status().isForbidden()
        );
    }
}
```

코드 27.38

유효한 CSRF 토큰을 요청 파라미터로 추가하려면 SecurityMockMvcRequestPostProcessors 클래스의 csrf 메서드를 사용한다.

```
package com.example.demo.controller;

...

import static org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestPostProcessors.csrf;

@SpringBootTest
@AutoConfigureMockMvc
public class CsrfRequestPostProcessorTest {

    @Autowired
    MockMvc mockMvc;

    ...

    @Test
    @WithMockUser
    void givenWithCsrfToken_whenPostUser_thenIsOk() throws Exception {
        mockMvc.perform(
                post("/api/user")
                        .with(user("user"))
                        .with(csrf())
```

```
        ).andExpect(
                status().isOk()
        );
    }
}
```

코드 27.39

CSRF 토큰을 헤더에 포함하고 싶다면 다음과 같이 설정한다.

```
@SpringBootTest
@AutoConfigureMockMvc
public class CsrfRequestPostProcessorTest {

    @Autowired
    MockMvc mockMvc;

    ...

    @Test
    void givenWithCsrfTokenInHeader_whenPostUser_thenIsOk() throws Exception {
        mockMvc.perform(
                post("/api/user")
                        .with(user("user"))
                        .with(csrf().asHeader())
        ).andExpect(
                status().isOk()
        );
    }
}
```

코드 27.40

유효하지 않은 CSRF 토큰을 지정할 수도 있다.

```
@SpringBootTest
@AutoConfigureMockMvc
public class CsrfRequestPostProcessorTest {
```

```
    @Autowired
    MockMvc mockMvc;

    ...

    @Test
    void givenWithInvalidCsrfToken_whenPostUser_thenIsForbidden() throws Exception
{
        mockMvc.perform(
                post("/api/user")
                        .with(user("user"))
                        .with(csrf().useInvalidToken())
        ).andExpect(
                status().isForbidden()
        );
    }
}
```

코드 27.41

27.4.3. 폼 로그인 지원

폼 로그인은 커스터마이징이 가능하다. 다음과 같은 경우를 예로 들 수 있다.

- 사용자 이름, 비밀번호 파라미터 이름을 변경한다.
- 로그인 처리 URL을 변경한다.
- 로그인 성공 후 리다이렉트 URL을 변경한다.
- 로그인 실패 후 리다이렉트 URL을 변경한다.
- 로그인 성공, 실패 후처리를 위한 핸들러는 지정한다.

스프링 시큐리티는 변경한 폼 로그인 설정을 확인할 방법을 제공한다. 테스트를 위해 다음과 같이 항상 고정된 사용자 정보를 반환하는 UserDetailsService 인스턴스가 있다.

- 애플리케이션 프로파일이 'form-login'인 경우에만 해당 bean 객체가 애플리케이션 컨텍스트에 등록된다.

```
package com.example.demo.controller;

...

import org.springframework.context.annotation.Profile;

/* 프로파일을 통한 선택적 bean 객체 등록 */
@Profile("form-login")
@Service
class StubFormLoginUserService implements UserDetailsService {
    @Override
    public UserDetails loadUserByUsername(
            String username
    ) throws UsernameNotFoundException {
        return User.withDefaultPasswordEncoder()
                .username("junhyunny")
                .password("12345")
                .roles("USER")
                .build();
    }
}
```

코드 27.42

테스트 코드에서 폼 로그인을 수행한다.

[1] 테스트 환경의 활성화된 프로파일은 'form-login'이다.

[2] 폼 로그인을 시도한다.

[3] 사용자 이름은 'junhyunny', 비밀번호는 '12345'이다.

[4] 로그인 성공 시 지정된 경로로 리다이렉트하는지 확인한다.

```
package com.example.demo.controller;

...

import static org.springframework.security.test.web.servlet.request.
```

```
SecurityMockMvcRequestBuilders.formLogin;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.
redirectedUrl;

/* [1] 테스트용 프로파일 지정 */
@SpringBootTest(properties = {"spring.profiles.active=form-login"})
@AutoConfigureMockMvc
public class FormRequestPostProcessorTest {

    @Autowired
    MockMvc mockMvc;

    @Test
    void whenFormLogin_thenRedirectIndex() throws Exception {
        mockMvc.perform(
                /* [2] 폼 로그인 시도 */
                formLogin()
                        /* [3] 폼 로그인 사용자 정보 */
                        .user("junhyunny")
                        .password("12345")
        ).andExpect(
                /* [4] 결과 확인 */
                redirectedUrl("/")
        );
    }
}
```

코드 27.43

로그인 수행 경로를 변경하였다면 폼 로그인 시뮬레이션에서도 경로를 지정할 수 있다.

```
@SpringBootTest(properties = {"spring.profiles.active=form-login"})
@AutoConfigureMockMvc
public class FormRequestPostProcessorTest {

    @Autowired
```

```
    MockMvc mockMvc;

    ...

    @Test
    void givenCustomLoginPath_whenFormLogin_thenRedirectIndex() throws Exception {
        mockMvc.perform(
                formLogin("/login")
                        .user("junhyunny")
                        .password("12345")
        ).andExpect(
                redirectedUrl("/")
        );
    }
}
```

코드 27.44

로그인 수행 시 전달하는 사용자 이름, 비밀번호가 매칭된 파라미터 이름을 커스터마이징하여 사용하고 있다면, 다음과 같이 파라미터 이름을 지정해 시뮬레이션할 수 있다.

```
@SpringBootTest(properties = {"spring.profiles.active=form-login"})
@AutoConfigureMockMvc
public class FormRequestPostProcessorTest {

    @Autowired
    MockMvc mockMvc;

    ...

    @Test
    void givenCustomParameterName_whenFormLogin_thenRedirectIndex() throws
Exception {
        mockMvc.perform(
                formLogin("/login")
                        .user("username","junhyunny")
```

```
                        .password("password", "12345")
        ).andExpect(
                redirectedUrl("/")
        );
    }
}
```

코드 27.45

폼 로그인은 기본적으로 로그인이 실패했을 때 사용자를 다시 로그인 화면으로 리다이렉트 시키는 엔트리포인트(entrypoint) 전략을 사용한다. 만약, 엔트리포인트를 커스터마이징했다면 이를 확인하는 것이 가능하다. 다음은 폼 로그인에 대한 기본 엔트리포인트 전략을 사용할 때 로그인 실패 결과를 검증하는 코드다.

```
@SpringBootTest(properties = {"spring.profiles.active=form-login"})
@AutoConfigureMockMvc
public class FormRequestPostProcessorTest {

    @Autowired
    MockMvc mockMvc;

    ...

    @Test
    void givenWrongUser_whenFormLogin_thenRedirectLoginPage() throws Exception {
        mockMvc.perform(
                formLogin()
                        .user("user")
                        .password("12345")
        ).andExpect(
                redirectedUrl("/login?error")
        );
    }

    @Test
    void givenWrongPassword_whenFormLogin_thenRedirectLoginPage() throws Exception
```

```
{
        mockMvc.perform(
                formLogin()
                        .user("junhyunny")
                        .password("wrong-password")
        ).andExpect(
                redirectedUrl("/login?error")
        );
    }
}
```

코드 27.46

27.4.4. HTTP 기본 인증 지원

7.6.1. SecurityConfig 클래스에서 다뤘듯이 시큐리티 필터 체인은 HTTP 기본 인증과 이를 커스터마이징하는 기능들을 제공한다. 예를 들어 다음과 같은 것들을 설정할 수 있다.

- 렐름 정보
- 인증 실패에 대한 인증 엔트리포인트

HTTP 기본 인증을 커스텀하게 변경했다면 테스트 코드에서 이를 검증하는 방법을 지원한다. 폼 로그인 방식과 동일하게 특정 프로파일에서만 컨텍스트에 등록되는 UserDetailsService 인스턴스를 준비한다.

- 'http-basic' 프로파일인 경우에만 해당 bean 객체가 애플리케이션 컨텍스트에 등록된다.

```
/* 프로파일을 위한 선택적 bean 객체 등록 */
@Profile("http-basic")
@Service
class StubHttpBasicUserService implements UserDetailsService {
    @Override
    public UserDetails loadUserByUsername(
            String username
    ) throws UsernameNotFoundException {
        return User.withDefaultPasswordEncoder()
                .username("junhyunny")
```

```
                .password("12345")
                .roles("USER")
                .build();
    }
}
```

코드 27.47

HTTP 기본 인증을 위해 httpBasic 메서드를 사용하면 해당 요청을 시뮬레이션할 때 사용자 이름과 비밀번호를 함께 Base64 인코딩하고, 해당 값을 헤더에 함께 담아 기본 인증을 시도한다.

```
package com.example.demo.controller;

...

import static org.springframework.security.test.web.servlet.request.SecurityMockMv
cRequestPostProcessors.httpBasic;

@SpringBootTest(properties = {"spring.profiles.active=http-basic"})
@AutoConfigureMockMvc
public class HttpBasicRequestPostProcessorTest {

    @Autowired
    MockMvc mockMvc;

    @Test
    void whenHttpBasic_thenIsOk() throws Exception {
        mockMvc.perform(
                get("/api/user")
                        .with(
                                httpBasic(
                                        "junhyunny",
                                        "12345"
                                )
                        )
```

```
        ).andExpect(
                status().isOk()
        );
    }
}
```

코드 27.48

렐름 정보를 바꿨거나 엔트리포인트 기능을 변경했다면 해당 내용을 테스트하는 것도 가능하다.

```
package com.example.demo.controller;

...

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.
header;

@SpringBootTest(properties = {"spring.profiles.active=http-basic"})
@AutoConfigureMockMvc
public class HttpBasicRequestPostProcessorTest {

    @Autowired
    MockMvc mockMvc;

    ...

    @Test
    void givenWrongPassword_whenHttpBasic_thenIsUnauthorized() throws Exception {
        mockMvc.perform(
                get("/api/user")
                        .with(
                                httpBasic(
                                        "junhyunny",
                                        "wrong-password"
                                )
```

```
                        )
        ).andExpect(
                status().isUnauthorized()
        ).andExpect(
                header().string(
                        "WWW-Authenticate",
                        "Basic realm=\"Realm\""
                )
        );
    }
}
```

코드 27.49

27.4.5. 로그아웃 지원

로그아웃 기능도 커스터마이징할 수 있다.

- 로그아웃 URL 변경
- 로그아웃 성공 URL 변경
- 로그아웃 성공 핸들러
- 지정한 쿠키 제거

스프링 시큐리티는 커스터마이징한 로그아웃 기능에 대한 테스트를 logout 메서드를 통해 지원한다. logout 메서드를 사용해 로그아웃을 수행하면 유효한 CSRF 토큰과 함께 지정한 로그아웃 경로로 POST 요청을 제출한다.

```
package com.example.demo.controller;

...

import static org.springframework.security.test.web.servlet.request.
SecurityMockMvcRequestBuilders.logout;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.
redirectedUrl;
```

```
@SpringBootTest
@AutoConfigureMockMvc
public class LogoutRequestTest {

    @Autowired
    MockMvc mockMvc;

    @Test
    void whenLogout_thenRedirectLoginPage() throws Exception {
        mockMvc.perform(
                logout("/logout")
        ).andExpect(
                redirectedUrl("/login?logout")
        );
    }
}
```

코드 27.50

27.5. 스프링 MVC 테스트 지원 – OAuth2

스프링 프레임워크는 엔드포인트에서 인증 주체 정보를 주입받을 수 있다. 다음처럼 JDK에 기본으로 포함된 Principal 인터페이스를 사용한다면 스프링 시큐리티의 OAuth2 지원이 필요 없다. @WithMockUser 애너테이션만으로 충분히 필요한 사용자를 준비할 수 있다.

```
package com.example.demo.controller;

...

import java.security.Principal;

@RestController
@RequestMapping("/api")
public class EndpointController {
```

```
    @GetMapping("/user")
    public String user(
            Principal principal
    ) {
        System.out.println(principal);
        return "ok";
    }
}
```

코드 27.51

하지만 스프링 시큐리티 OAuth2 클라이언트 의존성에 포함된 인터페이스를 직접 사용해 인증 주체를 주입받는다면 해당 타입의 사용자 정보를 테스트 코드에서 준비해야 한다.

```
package com.example.demo.controller;

...

import org.springframework.security.oauth2.core.oidc.user.OidcUser;

@RestController
@RequestMapping("/api")
public class EndpointController {

    @GetMapping("/user")
    public String user(
            @AuthenticationPrincipal OidcUser oidcUser
    ) {
        System.out.println(oidcUser);
        return "ok";
    }
}
```

코드 27.52

인증 주체뿐만 아니라 액세스 토큰과 클라이언트 정보를 사용하기 위해 OAuth2Authorized Client 객체를 주입받는다면 이를 위한 정보를 테스트 코드에서 준비해야 한다. 이번 절은

스프링 시큐리티가 OAuth2 기능과 결합 테스트를 위해 지원하는 내용들을 다룬다.

준비된 시큐리티 필터 체인은 다음과 같다.

```
package com.example.demo.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.Customizer;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.
EnableWebSecurity;
import org.springframework.security.web.SecurityFilterChain;

@EnableWebSecurity
@Configuration
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(
            HttpSecurity http
    ) throws Exception {
        http.authorizeHttpRequests(
                configurer -> configurer
                        .anyRequest()
                        .authenticated()
        );
        http.oauth2Login(
                Customizer.withDefaults()
        );
        return http.build();
    }
}
```

코드 27.53

27.5.1. OIDC 로그인 지원

OIDC 로그인 지원 기능을 확인하기 위해 다음과 같은 엔드포인트를 만든다.

```
package com.example.demo.controller;

...

import org.springframework.security.core.annotation.AuthenticationPrincipal;
import org.springframework.security.oauth2.core.oidc.user.OidcUser;

@RestController
@RequestMapping("/api")
public class EndpointController {

    @GetMapping("/user/oidc")
    public String oidcUser(
            @AuthenticationPrincipal OidcUser oidcUser
    ) {
        System.out.printf("TokenValue - %s\n", oidcUser.getIdToken().
getTokenValue());
        System.out.printf("UserInfo - %s\n", oidcUser.getUserInfo());
        System.out.printf("Claims - %s\n", oidcUser.getClaims());
        System.out.printf("Authorities - %s\n", oidcUser.getAuthorities());
        return "ok";
    }

    @GetMapping("/user/oauth2")
    public String oauth2User(
            @AuthenticationPrincipal OAuth2User oauth2User
    ) {
        System.out.println(oauth2User);
        return "ok";
    }
```

```
    @GetMapping("/user/client")
    public String oauth2Client(
            @RegisteredOAuth2AuthorizedClient("google")
            OAuth2AuthorizedClient authorizedClient
    ) {
        System.out.println(authorizedClient);
        return "ok";
    }
}
```

코드 27.54

먼저 OIDC 로그인 지원을 살펴본다. oidcLogin 메서드를 사용하면 OidcUser 인스턴스를 시큐리티 컨텍스트에 준비할 수 있다.

```
package com.example.demo;

...

import static org.springframework.security.test.web.servlet.request.SecurityMockMv
cRequestPostProcessors.oidcLogin;

@SpringBootTest
@AutoConfigureMockMvc
public class OidcPostRequestProcessorTest {

    @Autowired
    MockMvc mockMvc;

    @Test
    void whenGetUser_thenIsOk() throws Exception {
        mockMvc.perform(
                get("/api/user/oidc")
                        .with(oidcLogin())
```

```
        ).andExpect(
                status().isOk()
        );
    }
}
```

코드 27.55

해당 엔드포인트에서 출력하는 로그를 살펴보면 어떤 기본값들이 설정되었는지 확인할 수 있다.

- 사용자 정보는 널값이다.
- 클레임의 서브젝트 값은 'user'이다.
- 지정된 권한은 'OIDC_USER', 'SCOPE_read'이다.

```
TokenValue - id-token
UserInfo - null
Claims - {sub=user}
Authorities - [OIDC_USER, SCOPE_read]
```

테스트를 위해 인가 정보를 변경하고 싶다면 authorities 메서드를 사용한다.

```
package com.example.demo;

...

import org.springframework.security.core.authority.SimpleGrantedAuthority;

@SpringBootTest
@AutoConfigureMockMvc
public class OidcPostRequestProcessorTest {

    @Autowired
    MockMvc mockMvc;

    ...
```

```
    @Test
    void givenCustomAuthorities_whenGetUser_thenIsOk() throws Exception {
        var scopeUserRead = new SimpleGrantedAuthority("SCOPE_user:read");
        mockMvc.perform(
                get("/api/user/oidc")
                        .with(oidcLogin().authorities(scopeUserRead))
        ).andExpect(
                status().isOk()
        );
    }
}
```

코드 27.56

로그를 통해 지정한 권한 정보가 설정된 것을 확인할 수 있다.

```
TokenValue - id-token
UserInfo - null
Claims - {sub=user}
Authorities - [SCOPE_user:read]
```

idToken 메서드를 사용하면 클레임 정보를 커스터마이징할 수 있다.

```
@SpringBootTest
@AutoConfigureMockMvc
public class OidcPostRequestProcessorTest {

    @Autowired
    MockMvc mockMvc;

    ...

    @Test
    void givenCustomClaims_whenGetUser_thenIsOk() throws Exception {
        mockMvc.perform(
                get("/api/user/oidc")
```

```
                        .with(oidcLogin().idToken(
                                token -> token.claim("user_id", "junhyunny")
                        ))
        ).andExpect(
                status().isOk()
        );
    }
}
```

코드 27.57

로그를 통해 클레임 정보가 변경된 것을 확인할 수 있다.

```
TokenValue - id-token
UserInfo - null
Claims - {sub=user, user_id=junhyunny}
Authorities - [OIDC_USER, SCOPE_read]
```

oidcUser 메서드를 사용하면 OidcUser 인스턴스를 생성해 직접 설정하는 것도 가능하다.

```
package com.example.demo;

...

import org.springframework.security.core.authority.AuthorityUtils;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.oauth2.core.oidc.OidcIdToken;
import org.springframework.security.oauth2.core.oidc.OidcUserInfo;
import org.springframework.security.oauth2.core.oidc.user.DefaultOidcUser;
import org.springframework.security.oauth2.core.oidc.user.OidcUser;

@SpringBootTest
@AutoConfigureMockMvc
public class OidcPostRequestProcessorTest {

    OidcUser oidcUser = new DefaultOidcUser(
```

```
            AuthorityUtils.createAuthorityList("SCOPE_user:read"),
            OidcIdToken
                    .withTokenValue("custom-id-token")
                    .claim("user_name", "junhyunny")
                    .build(),
            OidcUserInfo.builder()
                    .email("junhyunny@gmail.com")
                    .build(),
            "user_name"
    );

    @Autowired
    MockMvc mockMvc;

    ...

    @Test
    void givenCustomOidcUser_whenGetUser_thenIsOk() throws Exception {
        mockMvc.perform(
                get("/api/user/oidc")
                        .with(oidcLogin().oidcUser(oidcUser))
        ).andExpect(
                status().isOk()
        );
    }
}
```

코드 27.58

로그를 통해 설정된 정보를 확인할 수 있다. OidcUserInfo 객체에 지정한 이메일 정보가 클레임에 추가된 것을 확인할 수 있다.

```
TokenValue - custom-id-token
UserInfo - org.springframework.security.oauth2.core.oidc.OidcUserInfo@2d7701fc
Claims - {user_name=junhyunny, email=junhyunny@gmail.com}
Authorities - [SCOPE_user:read]
```

27.5.2. OAuth2 로그인 지원

OAuth2 로그인 지원 기능을 확인하기 위해 다음과 같은 엔드포인트를 준비한다.

```
package com.example.demo.controller;

...

import org.springframework.security.core.annotation.AuthenticationPrincipal;
import org.springframework.security.oauth2.core.user.OAuth2User;

@RestController
@RequestMapping("/api")
public class EndpointController {

    ...

    @GetMapping("/user/oauth2")
    public String oauth2User(
            @AuthenticationPrincipal OAuth2User oauth2User
    ) {
        System.out.println(oauth2User);
        return "ok";
    }
}
```

코드 27.59

OIDC 로그인 테스트와 거의 유사한 방법으로 모의 인가 흐름을 만든다. oauth2Login RequestPostProcessor 메서드를 사용하면 스프링 시큐리티가 제공하는 기본 OAuth2User 인스턴스를 시큐리티 컨텍스트에 추가할 수 있다.

```
package com.example.demo;

...
```

```
import static org.springframework.security.test.web.servlet.request.SecurityMockMv
cRequestPostProcessors.oauth2Login;

@SpringBootTest
@AutoConfigureMockMvc
public class OAuth2RequestPostProcessorTest {

    @Autowired
    MockMvc mockMvc;

    @Test
    void whenGetUser_thenIsOk() throws Exception {
        mockMvc.perform(
                get("/api/user/oauth2")
                        .with(oauth2Login())
        ).andExpect(
                status().isOk()
        );
    }
}
```

코드 27.60

로그를 통해 스프링 시큐리티가 기본으로 제공하는 값들을 확인할 수 있다.

- 사용자 속성에 저장된 서브젝트 값은 'user'이다.
- 지정된 권한은 'OAUTH2_USER', 'SCOPE_read'이다.

```
Name: [user], Granted Authorities: [[OAUTH2_USER, SCOPE_read]], User Attributes:
[{sub=user}]
```

OIDC 로그인 테스트 지원과 마찬가지로 authorities 메서드를 사용하면 권한 정보를 변경할 수 있다.

```
package com.example.demo;
```

```
...

import org.springframework.security.core.authority.SimpleGrantedAuthority;

@SpringBootTest
@AutoConfigureMockMvc
public class OAuth2RequestPostProcessorTest {

    @Autowired
    MockMvc mockMvc;

    ...

    @Test
    void givenCustomAuthorities_whenGetUser_thenIsOk() throws Exception {
        var scopeUserRead = new SimpleGrantedAuthority("SCOPE_user:read");
        mockMvc.perform(
                get("/api/user/oauth2")
                        .with(oauth2Login().authorities(scopeUserRead))
        ).andExpect(
                status().isOk()
        );
    }
}
```

코드 27.61

로그를 통해 변경된 권한 정보를 확인할 수 있다.

```
Name: [user], Granted Authorities: [[SCOPE_user:read]], User Attributes: [{sub=user}]
```

사용자 속성을 추가하고 싶다면 attribute 메서드를 사용한다.

```
@SpringBootTest
@AutoConfigureMockMvc
public class OAuth2RequestPostProcessorTest {
```

```
    @Autowired
    MockMvc mockMvc;

    ...

    @Test
    void givenCustomAttributes_whenGetUser_thenIsOk() throws Exception {
        mockMvc.perform(
                get("/api/user/oauth2")
                        .with(oauth2Login().attributes(
                                attrs -> attrs.put("user_id", "junhyunny")
                        ))
        ).andExpect(
                status().isOk()
        );
    }
}
```

코드 27.62

테스트에서 추가한 사용자 속성이 로그에 반영된 것을 확인할 수 있다.

```
Name: [user], Granted Authorities: [[OAUTH2_USER, SCOPE_read]], User Attributes:
[{sub=user, user_id=junhyunny}]
```

OAuth2User 인스턴스를 직접 만들어 지정할 수도 있다. oauth2User 메서드를 사용한다.

```
package com.example.demo;

...

import org.springframework.security.core.authority.AuthorityUtils;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.oauth2.core.user.DefaultOAuth2User;
import org.springframework.security.oauth2.core.user.OAuth2User;

@SpringBootTest
```

```
@AutoConfigureMockMvc
public class OAuth2RequestPostProcessorTest {

    OAuth2User oauth2User = new DefaultOAuth2User(
            AuthorityUtils.createAuthorityList("SCOPE_user:read"),
            Collections.singletonMap("user_name", "junhyunny"),
            "user_name"
    );

    @Autowired
    MockMvc mockMvc;

    ...

    @Test
    void givenCustomOAuth2User_whenGetUser_thenIsOk() throws Exception {
        mockMvc.perform(
                get("/api/user/oauth2")
                        .with(oauth2Login().oauth2User(oauth2User))
        ).andExpect(
                status().isOk()
        );
    }
}
```

코드 27.63

27.5.3. OAuth2 클라이언트 지원

OAuth2 클라이언트 지원 기능을 확인하기 위해 다음과 같은 엔드포인트를 만든다.

```
package com.example.demo.controller;

...
```

```
import org.springframework.security.oauth2.client.OAuth2AuthorizedClient;
import org.springframework.security.oauth2.client.annotation.RegisteredOAuth2Autho
rizedClient;

@RestController
@RequestMapping("/api")
public class EndpointController {

    ...

    @GetMapping("/user/client")
    public String oauth2Client(
            @RegisteredOAuth2AuthorizedClient("google")
            OAuth2AuthorizedClient authorizedClient
    ) {
        System.out.printf("ClientId - %s\n", authorizedClient.
getClientRegistration().getClientId());
        System.out.printf("ClientSecret - %s\n", authorizedClient.
getClientRegistration().getClientSecret());
        System.out.printf("Principal - %s\n", authorizedClient.
getPrincipalName());
        System.out.printf("AccessToken - %s\n", authorizedClient.getAccessToken().
getTokenValue());
        System.out.printf("Scopes - %s\n", authorizedClient.getAccessToken().
getScopes());
        return "ok";
    }
}
```

코드 27.64

oauth2Client 메서드로 생성한 RequestPostProcessor인스턴스를 사용하면 테스트에 필요한 OAuth2AuthorizedClient 객체를 모의 OAuth2AuthorizedClientRepository 인스턴스에 추가할 수 있다.

[1] oauth2Client 기능은 애플리케이션에 인증된 사용자를 저장하지 않는다. 엔드포인트가 인증된 사용자만 접근 가능하다면 인증된 사용자 정보를 테스트 컨텍스트에 준비한다.

[2] 기본 registrationId 값은 'test'이기 때문에 엔드포인트에서 사용 중인 'google'로 지정한다.

```
package com.example.demo;

...

import static org.springframework.security.test.web.servlet.request.SecurityMockMv
cRequestPostProcessors.oauth2Client;
import static org.springframework.security.test.web.servlet.request.SecurityMockMv
cRequestPostProcessors.oauth2Login;

@SpringBootTest
@AutoConfigureMockMvc
public class OAuth2ClientRequestPostProcessorTest {

    @Autowired
    MockMvc mockMvc;

    @Test
    void whenGetUser_thenIsOk() throws Exception {
        mockMvc.perform(
                get("/api/user/client")
                        /* [1] oauth2 로그인 유저 준비 */
                        .with(oauth2Login())
                        /* [2] oauth2 인가 정보 준비 */
                        .with(oauth2Client("google"))
        ).andExpect(
                status().isOk()
        );
    }
}
```

코드 27.65

로그를 통해 oauth2Client 메서드가 준비하는 기본값을 확인할 수 있다.

- 클라이언트 아이디는 'test-client'이다.
- 클라이언트 시크릿은 'test-secret'이다.
- 인증 주체는 'user'이다.
- 액세스 토큰은 'access-token'이다.
- scope는 'read'이다.

```
ClientId - test-client
ClientSecret - test-secret
Principal - user
AccessToken - access-token
Scopes - [read]
```

accessToken 메서드를 통해 액세스 토큰에 대한 정보를 변경할 수 있다.

```
package com.example.demo;

...

import org.springframework.security.oauth2.core.OAuth2AccessToken;
import java.util.Collections;
import static org.springframework.security.oauth2.core.OAuth2AccessToken.
TokenType.BEARER;

@SpringBootTest
@AutoConfigureMockMvc
public class OAuth2ClientRequestPostProcessorTest {

    @Autowired
    MockMvc mockMvc;

    ...

    @Test
```

```
    void givenCustomAccessToken_whenGetUser_thenIsOk() throws Exception {
        mockMvc.perform(
                get("/api/user/client")
                        .with(oauth2Login())
                        .with(oauth2Client("google")
                                .accessToken(
                                        new OAuth2AccessToken(
                                                BEARER,
                                                "custom-token",
                                                null,
                                                null,
                                                Collections.singleton("message:read")
                                        )
                                )
                        )
        ).andExpect(
                status().isOk()
        );
    }
}
```

코드 27.66

로그를 통해 변경된 액세스 토큰 값을 확인할 수 있다.

- 토큰과 scope 정보가 변경된다.

```
ClientId - test-client
ClientSecret - test-secret
Principal - user
AccessToken - custom-token
Scopes - [message:read]
```

인증 주체 이름이나 클라이언트 등록 정보를 변경하는 것도 가능하다. 스프링 컨텍스트에 준비된 ClientRegistrationRepository 인스턴스를 사용한다면 application YAML 파일에 등록된 클라이언트 정보를 테스트 환경에서 사용할 수 있다.

```
package com.example.demo;

...

import org.springframework.security.oauth2.client.registration.
ClientRegistrationRepository;

@SpringBootTest
@AutoConfigureMockMvc
public class OAuth2ClientRequestPostProcessorTest {

    @Autowired
    ClientRegistrationRepository clientRegistrationRepository;

    @Autowired
    MockMvc mockMvc;

    ...

    @Test
    void givenCustomClientRegistration_whenGetUser_thenIsOk() throws Exception {
        mockMvc.perform(
                get("/api/user/client")
                        .with(oauth2Login())
                        .with(oauth2Client()
                                .principalName("junhyunny")
                                .clientRegistration(
                                        clientRegistrationRepository
                                            .findByRegistrationId("google")
                                )
                        )
        ).andExpect(
                status().isOk()
```

```
        );
    }
}
```

코드 27.67

로그를 통해 변경된 값을 확인할 수 있다.

- application YAML 파일에 등록된 클라이언트 아이디와 시크릿을 사용한다.
- 인증 주체 이름이 'junhyunny'로 변경된다.

```
ClientId - DUMMY_CLIENT_ID
ClientSecret - DUMMY_CLIENT_SECRET
Principal - junhyunny
AccessToken - access-token
Scopes - [read]
```