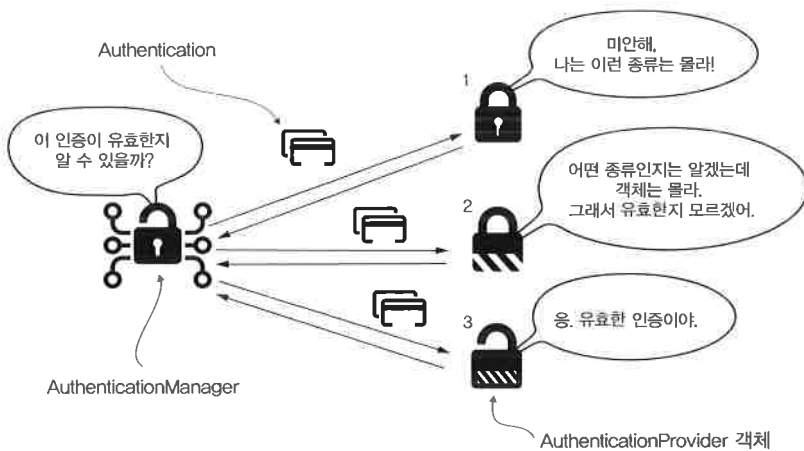


스프링 시큐리티

인 액션

보안 기초부터 OAuth 2까지, 스프링 시큐리티를 활용한
안전한 앱 설계와 구현



스프링 시큐리티 인 액션

보안 기초부터 OAuth 2까지, 스프링 시큐리티를 활용한
안전한 앱 설계와 구현

로렌티우 스피лка 지음
최민석 옮김



머리말

나는 2008년부터 소프트웨어 개발자이자 소프트웨어 개발 강사로 일하고 있다. 두 역할 모두 좋아하지만, 사실 강사/교육자의 역할을 편애하는 편이어서 지식을 공유하고 다른 사람이 기술을 향상하도록 돕는 것이 항상 우선순위였다. 그러나 이 분야에서는 어느 한 역할만 수행할 수는 없다고 생각한다. 모든 소프트웨어 개발자는 어느 정도 강사나 멘토의 역할을 해야 하고, 자신이 가르치는 내용을 실제 시나리오에 적용할 수 있는 확실한 이해 없이는 개발 강사가 될 수 없다.

경험을 통해 보안, 유지 관리, 성능 등과 같은 비기능적 소프트웨어 요구 사항의 중요성을 깨달았는데, 새로운 기술과 프레임워크를 배우는 데 투자한 시간보다 비기능적 측면을 배우는 데 더 많은 시간을 할애했다고 할 수 있을 정도다. 실제로 보통은 비기능적 문제보다 기능적 문제를 발견하고 해결하기가 훨씬 쉽다. 바로 이것이 지저분한 코드, 메모리 관련 문제, 다중 스레드 설계 문제, 그리고 물론 보안 취약성을 해결하기를 두려워하는 개발자들을 많이 만나는 이유일 것이다.

확실히 보안은 소프트웨어의 중요한 비기능적 특징 중 하나다. 스프링 시큐리티는 오늘날 애플리케이션에 보안을 적용하는 데 널리 사용되는 프레임워크 중 하나다. 이는 스프링 생태계인 스프링 프레임워크가 자바와 JVM 환경에서 엔터프라이즈 애플리케이션을 개발하기 위한 기술 선두주자로 인식되기 때문이다.

걱정스러운 점은 스프링 시큐리티를 적절히 이용해 일반적인 취약성으로부터 애플리케이션을 보호하는 방법을 배우는 것이 쉽지 않다는 것이다. 스프링 시큐리티에 관한 모든 세부 정보를 웹에서 찾을 수도 있다. 그러나 프레임워크를 최소한의 노력으로 올바른 순서로 조합하려면 많은 시간과 경험이 필요하다. 더욱이 불완전한 지식으로 솔루션을 구현하면 유지 관리와 개발이 어렵고 심지어 보안 취약성에 노출될 우려가 있다. 내가 건설당한 많은 개발 팀에서 스프링 시큐리티가 잘못 활용되는 사례를 목격했는데 스프링 시큐리티의 사용법에 대한 이해 부족이 주된 원인인 경우가 많았다.

이 때문에 스프링 개발자라면 누구나 스프링 시큐리티를 올바르게 이용하도록 도와주는 책을 쓰기로 했다. 이 책은 스프링 시큐리티에 대한 지식이 전혀 없는 사람도 점진적으로 이해할 수 있게 돕는 자료가 될 것이다. 바라는 것은 독자가 스프링 시큐리티를 배우는 시간을 절약하고 개발하는 앱에서 발생할 수 있는 모든 보안 취약성을 예방하여 큰 가치를 얻을 수 있게 하는 것이다.

책 소개

이 책은 누가 읽어야 하는가?

이 책은 스프링 프레임워크를 이용해 엔터프라이즈 애플리케이션을 개발하는 개발자를 위해 썼다. 모든 개발자는 개발 프로세스의 초기 단계부터 애플리케이션의 보안 측면을 고려해야 한다. 이 책은 스프링 시큐리티를 이용해 애플리케이션 수준 보안을 구성하는 방법을 다룬다. 스프링 시큐리티를 이용하는 방법과 애플리케이션에 보안 구성을 적절하게 적용하는 방법을 아는 것은 모든 개발자의 필수 요건이다. 이러한 측면은 매우 중요하며 제대로 이해하기 전에는 앱을 구현하는 책임을 맡으면 안 된다.

이 책은 스프링 시큐리티에 관한 배경지식이 없는 개발자가 시작할 수 있게 구성했지만 독자가 다음과 같은 스프링 프레임워크의 기본적인 측면을 다룬 경험이 있다고 가정했다.

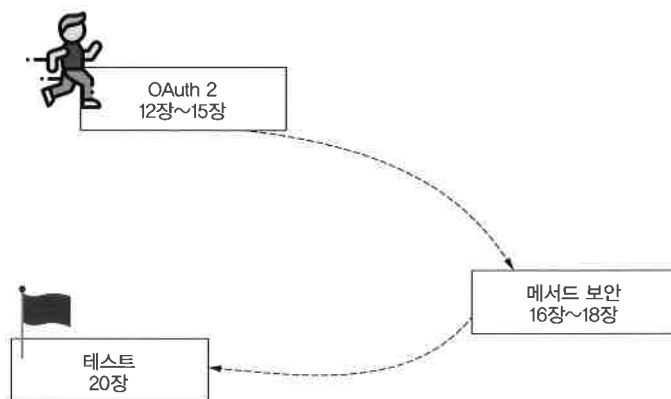
- 스프링 컨텍스트 사용
- REST 엔드포인트 구현
- 데이터 원본 사용

19장에서는 리액티브 앱에 보안 구성을 적용하는 방법을 설명한다. 또한 이 단원에서는 독자가 리액티브 앱의 개념을 이해하고 스프링을 이용해 리액티브 앱을 개발하는 방법을 이해하고 있다고 가정한다. 책 전반에 걸쳐 복습에 활용하기 위한 자료나 논의하는 내용을 제대로 이해하기 위해 알아야 할 주제에 관한 자료를 추천한다.

이 책의 예제는 자바로 작성했다. 스프링 프레임워크를 이용하는 개발자라면 자바도 이해할 수 있을 것이다. 직장에서는 코틀린 등의 다른 언어로 개발할 때도 있지만, 자바도 잘 알고 있을 가능성이 크므로 이 책의 예제를 작성하는 언어로 자바를 선택했다. 이러한 예제는 필요하면 손쉽게 코틀린으로 다시 작성할 수 있다.

- 8장에서는 권한 부여에 관한 논의를 계속하며 특정 HTTP 요청에 권한 부여 제약 조건을 적용하는 방법을 배운다. 이전 단원에서는 상황에 따라 요청을 허용하거나 거부하는 방법만 언급했지만, 이 단원에서는 경로나 HTTP 방식에 따라 특정 요청에 다른 권한 부여 구성을 적용하는 방법을 배운다.
- 9장에서는 필터 체인의 맞춤 구성을 주제로 논의하고 필터 체인이 인증과 권한 부여 구성을 적용하기 위해 HTTP 요청을 가로채는 책임 체인이라는 점을 배운다.
- 10장에서는 사이트 간 요청 위조 방지 기능에 관해 설명하고 스프링 시큐리티로 맞춤 구성하는 방법을 알아본다. 그런 다음 교차 출처 리소스 공유를 설명하고 보다 완전한 CORS 정책을 구성하는 방법과 이를 수행해야 하는 시기에 관해 배운다.
- 11장은 두 번째 실전 예제로 맞춤 구성된 인증과 권한 부여가 구현된 애플리케이션을 작성한다. 이 책에서 이미 배운 내용을 적용하지만, 이외에도 토큰이 무엇이고 권한 부여에서 어떤 역할을 하는지도 배운다.
- 12장에서는 더 복잡한 주제인 OAuth 2를 다루기 시작한다. 이 주제는 12장부터 15장까지의 주된 내용이다. 이 단원에서는 OAuth 2가 무엇인지 배우고 클라이언트가 백엔드 애플리케이션에서 노출한 엔드포인트를 호출하기 위해 액세스 토큰을 얻는 흐름을 설명한다.
- 13장에서는 스프링 시큐리티를 이용해 맞춤형 OAuth 2 권한 부여 서버를 구축하는 방법을 배운다.
- 14장에서는 스프링 시큐리티를 이용해 OAuth 2 시스템에 리소스 서버를 구축하는 방법과 권한 부여 서버에서 발급한 토큰을 리소스 서버가 검증하는 방법을 배운다.
- 15장에서는 시스템이 JSON 웹 토큰을 권한 부여에 이용하는 방법을 설명하면서 OAuth 2 주제를 마무리한다.
- 16장에서는 메서드 수준에 권한 부여 구성을 적용하는 방법을 설명한다.
- 17장은 16장에서 시작한 논의를 계속하고 메서드의 입출력을 나타내는 필터 값에 권한 부여 구성을 적용하는 방법을 배운다.
- 18장에서는 12장에서 17장까지 배운 내용을 적용한 세 번째 실전 예제를 진행하며 부가적으로 OAuth 2 시스템에서 타사 톨 Keycloak을 권한 부여 서버로 이용하는 방법을 배운다.
- 19장에서는 스프링 프레임워크로 개발한 리액티브 애플리케이션에 보안 구성을 적용하는 방법을 배운다.
- 20장에서는 지금까지의 여정을 마무리하며 보안 구성을 위한 통합 테스트를 작성하는 방법을 배운다.

이 책은 첫 단원부터 마지막 단원까지 순서대로 공부하기 쉽게 설계했으며 대체적으로 특정 단원의 내용을 이해하려면 이전에 논의한 주제를 이해하고 있어야 한다. 예를 들어 2장에서 논의한 스프링 시큐리티 기본 아키텍처의 개요를 이해하지 못한 상태로 사



기본 사항에 이미 익숙하고 특정 주제(예: OAuth 2)에만 관심이 있다면 해당 주제를 설명하는 단원으로 건너뛸 수 있다.

OAuth 2에 관심이 없다면 11장을 읽고 나서 OAuth 2 부분을 완전히 건너뛰고 16장과 17장을 진행해도 된다. OAuth 2 부분은 12장부터 15장까지 순서대로 읽어야 한다. 또한 OAuth 2 부분과 16장, 17장을 모두 마무리한 후 이 책의 마지막 실전 예제인 18장을 진행하는 것도 이치에 맞는다.

19장은 리액티브 앱 보안에 관한 내용이므로 리액티브 앱에 관심이 있는 독자만 읽으면 된다.

마지막 단원인 20장은 보안 구성을 위한 단위 테스트를 정의하는 방법을 다룬다. 이 책 전체에서 내용을 설명하기 위해 예제를 이용했는데 예제를 원활하게 진행하려면 이전의 모든 단원에서 배운 개념을 이해해야 한다. 한편 20장은 여러 절로 분할했는데, 각 절은 이 책에서 논의한 핵심 개념과 관계가 있다. 즉, 통합 테스트를 작성하는 방법은 배우고 싶지만 리액티브 앱에는 관심이 없다면 20장을 진행하면서 리액티브 앱과 관련된 절은 건너뛰어도 괜찮다.

저자 소개

로렌티우 스피лка(LAURENȚIU SPILCĂ)는 엔다바의 헌신적인 리더이자 트레이너이며 북유럽 국가의 금융 시장을 위한 프로젝트 개발을 이끌고 있다. 9년간의 실무 경험을 가지고 있으며 이전에는 전 세계에 설치된 가장 큰 규모의 ERP(전사적 자원 관리) 솔루션 중 하나를 구축하는 프로젝트에 소프트웨어 개발자로 참여했다.

그는 고품질의 소프트웨어를 제공하는 것도 중요하지만 지식을 공유하고 다른 사람들이 기술을 향상하도록 돕는 것도 중요하다고 믿으며 이런 생각으로 자바 기술에 관한 교육 과정을 설계하고 강의하며 미국과 유럽에서 프레젠테이션과 워크숍을 제공하고 있다. 그의 강연은 복스드 데이즈(Voxxed Days), 테크플로우(TechFlow), 부쿠레슈티 테크놀로지 위크(Bucharest Technology Week), 자바스콧(JavaSkop), 오라클 코드 익스플로어(Oracle Code Explore), 오라일리 소프트웨어 아키텍처(O'Reilly Software Architecture), 오라클 코드 원(Oracle Code One) 등에서 접할 수 있다.

표지 설명

《스프링 시큐리티 인 액션》의 표지로 사용된 그림에는 “무르시아에서 온 남자”라는 제목이 붙어 있다. 이 삽화는 자크 그라세 드 생소비르(1757~1810)가 1788년 프랑스에서 출판한 다른 나라의 의상(Costumes de Differents Pays)이라는 제목의 여러 나라의 드레스 의상 컬렉션에서 가져온 것이다. 각 일러스트는 정교하게 손으로 그리고 채색한 것으로 컬렉션의 다채로운 그림을 보고 있으면 불과 200여 년 전만 해도 세계의 도시와 지역이 문화적으로 얼마나 다채로운 모습을 보여주었는지 상기시켜준다. 당시 사람들은 서로와 단절된 채 다양한 언어와 방언으로 의사소통 했지만, 거리나 시골에서 이들의 옷차림을 보면 어디에 살고 어떤 일을 하는 사람인지 쉽게 알 수 있었다.

이후로 의상의 양식이 달라졌고 당시 풍부했던 지역별 다양성도 사라졌다. 이제는 마을이나 지역은 고사하고 다른 대륙에 사는 사람을 구분하기도 어려워졌다. 어쩌면 우리는 문화적 다양성을 더 다양한 개인의 삶, 그리고 다채롭고 빠르게 변하는 기술적 삶과 맞바꿨는지 모른다.

그리고 컴퓨터 책을 서로 구분하기도 어려운 지금, 매닝은 이렇게 보기 드문 자크 그라세 드 생소비르의 그림을 발굴해 2세기 전 지역의 풍부한 삶의 다양성을 보여주는 책 표지로 사용함으로써 컴퓨터 산업의 창의성과 진취성을 표현하고자 한다.

02부

구현

2 장 \ 안녕! 스프링 시큐리티	35
2.1 첫 번째 프로젝트 시작	36
2.2 기본 구성이란?	41
2.3 기본 구성 재정의	46
2.3.1 UserDetailsService 구성 요소 재정의	46
2.3.2 엔드포인트 권한 부여 구성 재정의	51
2.3.3 다른 방법으로 구성 설정	53
2.3.4 AuthenticationProvider 구현 재정의	57
2.3.5 프로젝트에 여러 구성 클래스 이용	61
요약	63
3 장 \ 사용자 관리	66
3.1 스프링 시큐리티의 인증 구현	67
3.2 사용자 기술하기	70
3.2.1 UserDetails 계약의 정의 이해하기	70
3.2.2 GrantedAuthority 계약 살펴보기	72
3.2.3 최소한의 UserDetails 구현 작성	73
3.2.4 빌더를 이용해 UserDetails 형식의 인스턴스 만들기	76
3.2.5 사용자와 연관된 여러 책임 결합	77
3.3 스프링 시큐리티가 사용자를 관리하는 방법 지정	81
3.3.1 UserDetailsService 계약의 이해	82
3.3.2 UserDetailsService 계약 구현	83
3.3.3 UserDetailsManager 계약 구현	86
요약	95

6 장 \ 실전: 작고 안전한 웹 애플리케이션	149
6.1 프로젝트 요구 사항과 설정	149
6.2 사용자 관리 구현	156
6.3 맞춤형 인증 논리 구현	161
6.4 주 페이지 구현	165
6.5 애플리케이션 실행 및 테스트	168
요약	169
 7 장 \ 권한 부여 구성: 액세스 제한	172
7.1 권한과 역할에 따라 접근 제한	172
7.1.1 사용자 권한을 기준으로 모든 엔드포인트에 접근 제한	173
7.1.2 사용자 역할을 기준으로 모든 엔드포인트에 대한 접근을 제한	183
7.1.3 모든 엔드포인트에 대한 접근 제한	188
요약	190
 8 장 \ 권한 부여 구성: 제한 적용	191
8.1 선택기 메서드로 엔드포인트 선택	192
8.2 MVC 선택기로 권한을 부여할 요청 선택	199
8.3 앤트 선택기로 권한을 부여할 요청 선택	208
8.4 정규식 선택기로 권한을 부여할 요청 선택	213
요약	219

11.4.3 AuthenticationProvider 인터페이스 구현	307
11.4.4 필터 구현	309
11.4.5 보안 구성 작성	316
11.4.6 전체 시스템 테스트	317
요약	319
12 장 \ OAuth 2가 작동하는 방법	320
12.1 OAuth 2 프레임워크	321
12.2 OAuth 2 인증 아키텍처의 구성 요소	323
12.3 OAuth 2를 구현하는 방법 선택	324
12.3.1 승인 코드 그랜트 유형의 구현	325
12.3.2 암호 그랜트 유형 구현	330
12.3.3 클라이언트 자격 증명 그랜트 유형 구현	333
12.3.4 갱신 토큰으로 새 액세스 토큰 얻기	334
12.4 OAuth 2의 허점	336
12.5 간단한 SSO(Single Sign-On) 애플리케이션 구현	337
12.5.1 권한 부여 서버 관리	338
12.5.2 구현 시작	341
12.5.3 ClientRegistration 구현	343
12.5.4 ClientRegistrationRepository 구현	346
12.5.5 스프링 부트 구성의 순수한 마법	349
12.5.6 인증된 사용자의 세부 정보 얻기	350
12.5.7 애플리케이션 테스트	351
요약	354

15.3	JWT에 맞춤형 세부 정보 추가	428
15.3.1	토큰에 맞춤형 세부 정보를 추가하도록 권한 부여 서버 구성	429
15.3.2	JWT의 맞춤형 세부 정보를 읽을 수 있게 리소스 서버 구성	432
	요약	435
16 장	전역 메서드 보안: 사전 및 사후 권한 부여	436
16.1	전역 메서드 보안 활성화	437
16.1.1	호출 권한 부여의 이해	438
16.1.2	프로젝트에서 전역 메서드 보안 활성화	441
16.2	권한과 역할에 사전 권한 부여 적용	442
16.3	사후 권한 부여 적용	448
16.4	메서드의 사용 권한 구현	453
	요약	466
17 장	전역 메서드 보안: 사전 및 사후 필터링	467
17.1	메서드 권한 부여를 위한 사전 필터링 적용	468
17.2	메서드 권한 부여를 위한 사후 필터링 적용	478
17.3	스프링 데이터 리포지토리에 필터링 이용	481
	요약	488
18 장	실전: OAuth 2 애플리케이션	490
18.1	애플리케이션 시나리오	491
18.2	Keycloak을 권한 부여 서버로 구성	492
18.2.1	시스템에 클라이언트 등록	497
18.2.2	클라이언트 범위 지정	498
18.2.3	사용자 추가 및 액세스 토큰 얻기	501
18.2.4	사용자 역할 정의	505

부록 A \ 스프링 부트 프로젝트 만들기	583
A.1 start.spring.io로 프로젝트 만들기	584
A.2 STS(스프링 툴 스위트)로 프로젝트 만들기	585

에 원치 않게 참여하는 등 다른 시스템 오류도 발생할 수 있다. 비기능적 요구 사항의 숨겨진 측면(누락되거나 숨겨진 것을 찾기 어려움) 때문에 이러한 문제는 더 위험하다.



그림 1.1 기능적 요구 사항만 생각하는 사용자. 비기능적 특징인 성능에 신경을 쓰는 사용자는 종종 있지만, 아쉽게도 보안을 중시하는 사용자는 상당히 드물다. 비기능적 요구 사항은 기능적 요구 사항에 비해 잘 드러나지 않는다.

소프트웨어 시스템을 다룰 때는 여러 비기능적 측면을 고려해야 한다. 이러한 비기능적 측면도 모두 중요하며 소프트웨어 개발 프로세스에서 책임감 있게 다뤄야 한다. 이 책에서는 이러한 비기능적 측면 중 하나인 보안에 초점을 맞추고 스프링 시큐리티를 이용해 단계적으로 애플리케이션을 보호하는 방법을 배운다.

그러나 시작하기 전에 이 단원은 독자의 숙련도에 따라 다소 부담스럽게 느껴질 수 있다는 점을 밝혀 둔다. 하지만 아직 모든 내용이 명확하게 이해되지 않아도 걱정할 필요는 없다. 이 단원은 보안 개념의 큰 그림을 보여주기 위한 것이다. 이 책 전체에서 실용적인 예를 다루면서 필요에 따라 이 단원에서 다룬 내용을 다시 참조할 것이며 필요할 때는 더 자세하게 설명한다. 그리고 여기저기에서 특정 주제를 더 자세하게 알아볼 수 있는 다른 자료(책, 기사, 설명서)를 소개한다.

션, 빈, 그리고 일반적으로 스프링 방식의 구성 스타일을 능숙하게 이용해 애플리케이션 수준의 보안을 정의하는 것이다. 스프링 애플리케이션에서 보호해야 하는 동작은 메서드로 정의된다.

애플리케이션 수준 보안을 논의할 때는 집의 출입을 통제하는 방법을 비교할 수 있다. 현관 깔개 밑에 열쇠를 숨겨두는가? 현관문 열쇠가 있기는 한가? 애플리케이션에도 같은 개념이 적용되며 이 기능을 개발하는 데 스프링 시큐리티가 도움이 된다. 스프링 시큐리티는 시스템을 묘사하는 정확한 이미지를 구축하는 데 다양한 선택지를 제공하는 퍼즐이다. 집을 아주 취약한 상태로 방치할 수도 있고 아무도 들어오지 못하게도 할 수도 있다.

보안을 구성하는 방법은 열쇠를 깔개 밑에 숨기는 것처럼 간단할 수도 있고 다양한 정보 시스템, 비디오 카메라, 자물쇠를 설치하는 것처럼 더 복잡할 수도 있다. 여러분의 애플리케이션에도 같은 옵션이 있으며 현실처럼 복잡도를 높일수록 비용도 증가한다. 애플리케이션에서 이 비용은 보안이 유지 관리와 성능에 미치는 영향을 의미한다.

그러면 스프링 애플리케이션에 스프링 시큐리티를 어떻게 이용하면 좋을까? 일반적으로 애플리케이션 수준에서 가장 흔한 보안의 활용 사례는 누가 작업을 수행할 수 있는지, 특정 데이터를 이용할 수 있는지를 결정하는 것이다. 구성을 기반으로 요청을 가로채고 권한을 가진 사용자만 보호된 리소스에 접근할 수 있도록 스프링 시큐리티 구성 요소를 작성한다. 개발자는 원하는 것을 정확하게 수행하도록 구성 요소를 구성한다. 정보 시스템을 설치한다면 문에만 설치하고 끝낼 것이 아니라 창문에도 설치하는 것이 여러분의 역할이다. 창문에 설치하는 것을 깜박했다면 도둑이 창문으로 침입할 때 정보 시스템이 작동하지 않은 것은 정보 시스템의 문제가 아니라 여러분의 잘못이다.


스프링 시큐리티 구성 요소의 다른 책임은 시스템의 다른 부분 간의 데이터 전송 및 저장과 관련이 있다. 구성 요소는 이 다른 부분에 대한 호출을 가로채서 데이터에 작업을 수행한다. 예를 들어 데이터가 저장될 때 암호화나 해싱 알고리즘을 적용할 수 있으며 데이터 인코딩으로 이용 권리가 있는 주체만 데이터에 접근하게 할 수 있다. 개발자는 스프링 애플리케이션에서 필요할 때마다 작업의 이 부분을 수행하도록 구성 요소를 추가하고 구성해야 한다. 스프링 시큐리티는 프레임워크를 위해 구현해야 하는 것을 알려주는 계약을 제공하고, 개발자는 애플리케이션 설계에 따라 구현을 작성한다. 데이터 전송에도 같은 개념이 적용된다.

실제 구현에서는 통신하는 두 구성 요소가 서로를 신뢰하지 않는 경우가 있다. 어떤 구성 요소가 특정 메시지를 보냈는지 어떻게 확신할 수 있을까? 전화 통화를 하며 상대방에게 자신의 개인 정보를 알려줘야 하는 경우를 생각해보자. 상대방이 이러한 정보를 받아도 되는 사람인지 어떻게 알 수 있을까? 이

그러나 아파치 시로는 애플리케이션의 요구를 충족하는 데 너무 '경량급'일 수 있다. 스프링 시큐리티는 단순히 하나의 툴이 아니라 모든 툴을 준비한 세트로서 광범위한 가능성을 제공하며 스프링 애플리케이션을 위해 특별히 설계됐다. 또한 많은 활동적인 개발자 커뮤니티에서 도움을 받을 수 있고 계속해서 향상되고 있다.

1.2 소프트웨어 보안이란?

현재의 소프트웨어 시스템은 특히 현재 GDPR(General Data Protection Regulations; 일반 데이터 보호 규정) 요구 사항을 고려할 때 상당 부분이 민감한 정보일 수 있는 대량의 데이터를 관리한다. 사용자가 개인적이라고 생각하는 모든 정보는 소프트웨어 애플리케이션에서 민감한 정보가 된다. 민감한 정보에는 전화번호, 이메일 주소 또는 식별 번호와 같은 무해한 정보도 있지만, 유출됐을 때 위험성이 높은 신용 카드 정보 등의 정보는 더 중요하게 고려해야 한다. 애플리케이션은 이러한 정보에 접근, 변경 또는 가로챌 기회가 없게 해야 하며 의도된 사용자 이외의 대상은 어떤 식으로든 데이터와 상호 작용할 수 없게 해야 한다. 이것이 광범위하게 표현한 보안의 의미다.

 **참고** GDPR은 2018년 도입된 후 전 세계적으로 상당히 큰 반향을 일으켰다. GDPR은 일반적으로 데이터 보호와 관련해 사람들에게 개인 데이터에 대한 더 많은 제어 권한을 부여하는 일련의 유럽 법률을 나타내며 유럽 사용자를 보유한 시스템의 소유자에게 적용된다. 이러한 애플리케이션의 소유자가 적용 규정을 준수하지 않으면 상당한 처벌을 받을 수 있다.

보안은 계층별로 적용해야 하며 각 계층에 다른 접근 방식이 필요하다. 이러한 각 계층은 성을 보호하는 일과 비교할 수 있다(그림 1.2). 해커는 앱이 보호하는 리소스를 획득하기 위해 여러 장애물을 통과해야 한다. 각 계층을 더 잘 보호할수록 악의적인 대상이 데이터에 접근하거나 무단 작업을 수행할 가능성이 낮아진다.

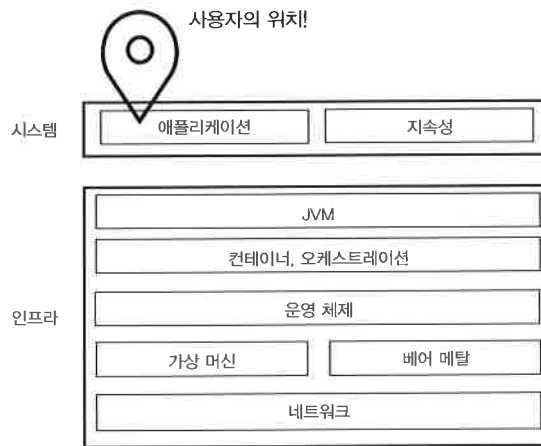


그림 1.3 보안은 각 계층에 적용되고 각 계층은 그 아래의 계층에 의존한다. 이 책에서는 최상층에서 애플리케이션 수준 보안을 구현하는 프레임 워크인 스프링 시큐리티를 다룬다.

더 명확한 이해를 위해 현실적인 사례로 그림 1.4와 같은 시스템을 구축했다고 생각해보자. 이 상황은 마이크로서비스 아키텍처를 이용해 설계된 시스템에서 특히 클라우드의 여러 가용 영역에 배포하는 경우 흔히 볼 수 있다.

이러한 마이크로서비스 아키텍처에서는 다양한 취약성이 생길 수 있으므로 주의해야 한다. 앞서 언급했듯이 보안은 우리가 여러 계층에서 설계해야 하는 공통 관심사다. 한 계층의 보안 문제를 해결할 때는 되도록 위 계층이 존재하지 않는다고 가정하는 것이 바람직하다. 그림 1.2에 나온 성의 비유를 생각해보자. 병사 30명과 ‘(보안) 계층’을 관리하는 책임자라면 최선을 다해 병사를 훈련시킬 것이다. 병사를 만나기 전에 불타는 다리를 건너야 한다고 해서 방심하지 않을 것이다.

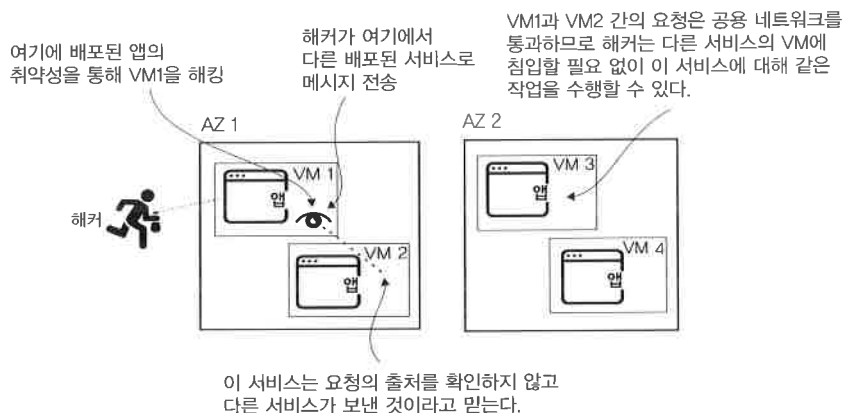


그림 1.4 악성 사용자가 애플리케이션 보안이 적용되지 않은 VM(가상 머신)에 침입하는 데 성공하면 시스템에 속한 다른 애플리케이션도 통제할 수 있다. 두 개의 서로 다른 AZ(가용 영역) 간에 통신이 이루어지면 악성 사용자는 더 쉽게 메시지를 가로챌 수 있으며 이 취약성을 악용해 데이터를 훔치거나 사용자를 가장할 수 있다.

누가 접근을 요청하는지 알아야 한다. 즉, 인증을 해야 한다. 또한 사용자가 시스템의 해당 부분을 이용하도록 허용하려면 사용자에게 어떤 이용 권리가 있는지 알아야 한다. 시스템이 복잡해지면 특정한 인증과 권한 부여의 구현이 필요한 다양한 상황이 발생한다.

예를 들어 사용자를 대신해 시스템의 특정 구성 요소에 데이터의 하위 집합이나 작업에 대한 권한을 부여하려면 어떻게 해야 할까? 예를 들어 프린터에 사용자의 문서를 읽을 수 있는 접근 권한이 필요할 수 있다. 간단하게 사용자의 자격 증명을 프린터에 제공해야 할까? 하지만 이렇게 하면 프린터에 필요 이상의 권한이 부여되고 사용자의 자격 증명도 노출되는 문제가 있다. 사용자를 가장하지 않고 해결하는 올바른 방법이 있을까? 이는 핵심적인 질문이고 애플리케이션을 개발하면서 자주 접할 질문이다. 이 책에서 스프링 시큐리티를 사용한 애플리케이션을 작성하며 그 대답을 찾을 수 있을 것이다.

인증과 권한 부여는 시스템을 위해 선택한 아키텍처에 따라 구성 요소만이 아닌 전체 시스템 수준에 이용될 수 있다. 또한 이 책의 뒷부분에서 더 자세히 살펴보겠지만, 스프링 시큐리티를 이용할 때는 같은 구성 요소의 다른 계층에도 권한 부여를 적용하는 것이 좋을 수 있다. 16장에서 이를 위한 전역 메서드 보안을 살펴본다. 사전 정의된 역할과 권한의 집합이 있으면 이 설계가 더욱 복잡해진다.

데이터 저장소에 관해서도 주의할 필요가 있다. 저장 데이터(Data at Rest)는 애플리케이션의 책임을 가중한다. 모든 데이터를 읽을 수 있는 형식으로 저장하지 말고 개인 키로 암호화한 데이터나 해시된 데이터로 저장해야 한다. 자격 증명 및 개인 키와 같은 비밀도 저장 데이터로 간주할 수 있는데, 일반적으로 이러한 데이터는 비밀 볼트에 조심스럽게 저장해야 한다.

참고 데이터는 저장 데이터(Data at Rest)와 전송 중 데이터(Data in Transit)로 구분한다. 이 맥락에서 저장 데이터는 컴퓨터 스토리지에 있는 데이터, 즉 지속된 데이터를 말한다. 전송 중 데이터는 한 위치에서 다른 위치로 교환 중인 모든 데이터를 말한다. 데이터의 유형에 따라 다른 보안 조치를 적용해야 한다.

실행 중인 애플리케이션은 내부 메모리도 관리해야 한다. 이상하게 들릴 수 있지만, 애플리케이션의 힙에 저장된 데이터도 취약성의 원인일 수 있다. 클래스 디자인에 따라 앱이 자격 증명이나 개인 키 등의 민감한 데이터를 장시간 보관할 때가 있는데 힙 덤프 이용 권리가 있는 누군가가 이 사실을 알고 악용할 수 있다.

이 간략한 설명으로 이 주제의 복잡성과 애플리케이션 보안의 의미를 어느 정도 이해했기를 바란다. 소프트웨어 보안은 아주 복잡한 주제다. 이 분야의 전문가가 되려면 솔루션을 이해하는 것은 물론 시스템

- 조직의 내부 데이터를 관리해야 하는 백오피스 애플리케이션에서 어떤 이유로 일부 정보가 누출됐다.
- 차량 공유 애플리케이션의 사용자가 자신의 계정에서 본인이 이용하지 않은 차량 이용에 대해 금액이 인출되는 것을 발견했다.
- 모바일 은행 애플리케이션을 업데이트하자 다른 사용자의 거래 내역이 표시되기 시작했다.

첫 번째 사례에서는 소프트웨어를 이용하는 조직은 물론 그 직원들이 영향을 받을 수 있다. 때에 따라 특정 회사는 사태의 책임을 지고 상당한 비용 손실을 볼 수 있다. 이 상황에 사용자는 애플리케이션을 변경할 수 없지만 조직은 소프트웨어 공급업체를 변경하도록 결정할 수 있다.

두 번째 사례에서 사용자가 서비스 공급업체를 변경할 것이고 애플리케이션을 개발한 회사의 이미지가 상당히 손상될 가능성이 크다. 이 사례에서는 회사의 이미지 손상이 비용 손실보다 더 큰 문제다. 잘못 청구된 요금을 고객에게 반환해도 애플리케이션의 사용자는 일정 수준 감소할 것이다. 이로써 수익성이 악화되고 최악의 경우 사업이 실패할 수도 있다. 세 번째 사례의 경우 이 은행은 법적 책임을 져야 하는 것은 물론 신뢰성 측면에서 심각한 타격을 받는다.

대부분은 해커가 시스템의 취약성을 악용할 수 있게 방지하는 것보다 미리 보안에 투자하는 것이 훨씬 현명한 선택이다. 이 모든 사례에서 아주 작은 약점만으로도 이러한 결과가 발생할 수 있다. 첫 번째 사례는 인증 취약성이나 CSRF(사이트 간 요청 위조)를 악용한 공격일 수 있고 두 번째와 세 번째 사례는 메서드 접근 통제가 부족해서 생긴 상황일 수 있다. 그리고 모든 사례에서 여러 취약성의 조합이 원인일 수 있다.

한 걸음 더 나아가 방어 관련 시스템의 보안에 대해서도 논의할 수 있다. 이러한 시스템에서 보안을 소홀히 한 대가는 금전적 피해에 그치지 않고 인명 피해로 이어질 수 있다. 환자를 돌보는 의료 시스템이 공격받는다면 어떤 피해가 발생할지 상상할 수 있겠는가? 원자력 발전소를 통제하는 시스템이 공격받는다면 어떨까? 애플리케이션 보안에 조기에 투자하고 보안 전문가가 보안 메커니즘을 개발하고 테스트할 수 있도록 충분한 시간을 할당하면 위험을 줄일 수 있다.

참고 앞의 실패 사례에서 얻을 수 있는 교훈은, 취약성을 예방하는 것이 공격받은 후 대처하는 것보다 비용이 적게 든다는 것이다.

이 책의 나머지 부분에서는 스프링 시큐리티를 적용해 앞서 살펴본 상황을 미리 방지하는 방법을 배운다. 보안의 중요성에 관해서는 아무리 강조해도 부족함이 없다. 시스템의 보안을 타협해야 하는 상황이라면 위험을 올바르게 평가하도록 노력해보자.

권한 부여(Authorization)는 인증된 호출자가 특정 기능과 데이터에 대한 이용 권리가 있는지 확인하는 프로세스다. 모바일 은행 애플리케이션을 예로 들면 대부분의 인증된 사용자는 자금을 이체할 수 있지만 자신의 계좌에서만 가능하다.

인증 취약성이 있다는 것은 사용자가 악의를 가지고 다른 사람의 기능이나 데이터에 접근할 수 있다는 의미다. 스프링 시큐리티와 같은 프레임워크로 이러한 취약성이 발생할 우려를 줄일 수 있지만, 올바르게 이용하지 않으면 여전히 위험성은 있다. 예를 들어 스프링 시큐리티를 이용해 특정 역할이 있는 인증된 사용자만 특정 엔드포인트에 접근하도록 정의할 수 있다. 하지만 데이터 수준에 제한이 없으면 다른 사용자의 데이터를 이용할 수 있는 허점이 생길 수 있다.

그림 1.5를 보자. 이 예에서 인증된 사용자는 `/products/{name}` 엔드포인트에 접근할 수 있다. 웹 앱은 브라우저에서 이 엔드포인트를 호출해 데이터베이스에서 사용자의 제품을 검색하고 화면에 표시한다. 그런데 앱이 데이터를 반환하면서 제품이 누구의 것인지 제대로 확인하지 않으면 어떻게 될까? 그러면 다른 사용자의 세부 정보를 들여다볼 수 있는 틈이 생길 수 있다. 이러한 상황은 애플리케이션 설계 초기부터 방지하기 위해 고려해야 하는 여러 예 중 하나일 뿐이다.

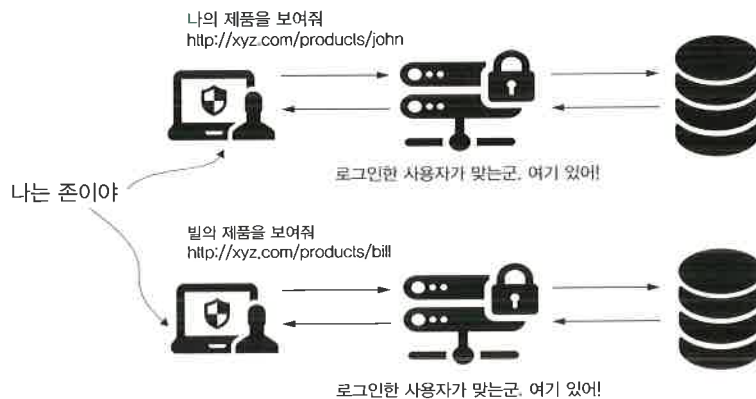


그림 1.5 로그인한 사용자는 자신의 제품을 볼 수 있다. 애플리케이션 서버가 사용자의 로그인 여부만 확인하면 사용자가 같은 엔드포인트로 다른 사용자의 제품을 검색할 수 있다. 이 예에서 사용자 존은 빌의 데이터를 볼 수 있다. 이 문제의 원인은 애플리케이션이 사용자의 데이터 검색까지 인증하지 않기 때문이다.

취약성은 이 책 전체에서 배울 내용이며 3장에서 인증과 권한 부여의 기본 구성을 살펴보는 것으로 시작한다. 그런 다음 취약성이 스프링 시큐리티와 스프링 데이터의 통합에 어떤 관련이 있는지, OAuth 2로 이러한 취약성을 방지하도록 애플리케이션을 설계하려면 어떻게 해야 하는지 논의한다.

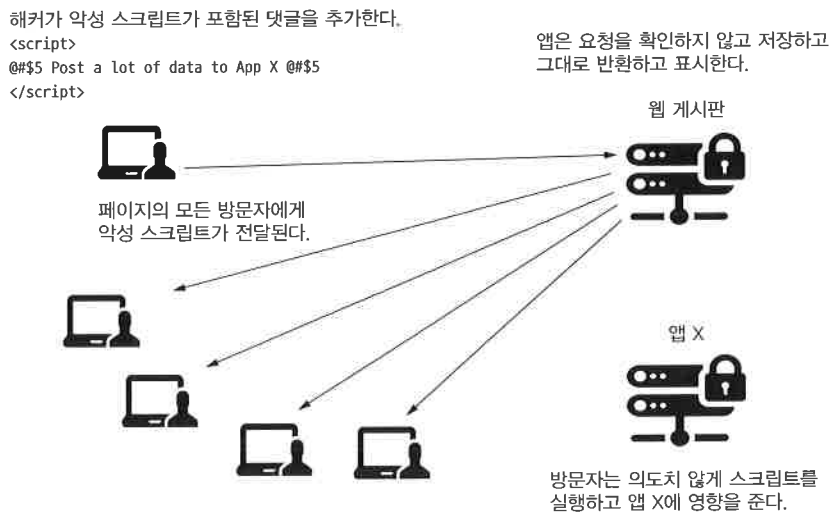


그림 1.6 한 사용자가 스크립트가 포함된 댓글을 웹 게시판에 게시한다. 이 사용자는 공격 피해자에 해당하는 다른 애플리케이션(앱 X)에 막대한 양의 데이터를 게시하거나 가져오도록 스크립트를 정의한다. 웹 게시판 앱에 XSS(교차 사이트 스크립팅)에 대한 방어 기능이 없으면 악성 댓글을 표시하는 페이지에 방문한 모든 사용자가 이를 실행하게 된다.

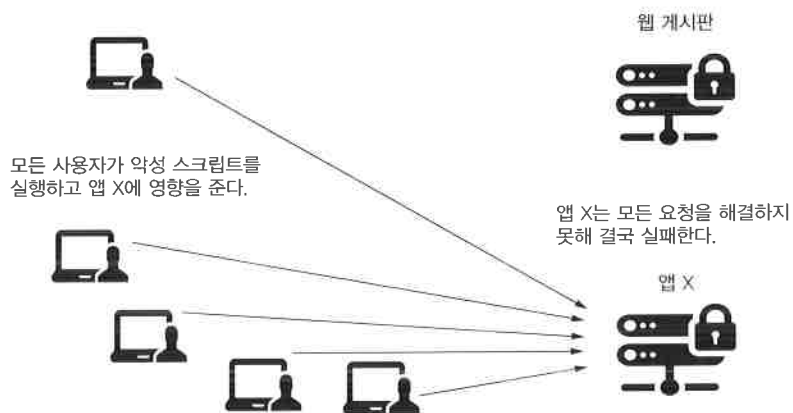


그림 1.7 사용자가 악성 스크립트를 표시하는 페이지를 방문한다. 사용자의 브라우저가 스크립트를 실행하고 앱 X에 많은 양의 데이터를 게시하거나 가져오려고 시도한다.

1.4.4 CSRF(사이트 간 요청 위조)란?

CSRF(사이트 간 요청 위조)도 웹 애플리케이션에 흔한 취약성이다. CSRF 공격은 특정 서버에서 작업을 호출하는 URL을 추출해 애플리케이션 외부에서 재사용할 수 있다고 가정한다(그림 1.8). 서버가 요청의 출처를 확인하지 않고 무턱대고 실행하면 다른 모든 곳에서 요청이 실행될 수 있다. 공격자는

1.4.6 민감한 데이터의 노출 처리하기

기밀 데이터 공개는 복잡성 측면에서 가장 기초적이고 단순한 취약성 같지만, 여전히 흔한 실수 중 하나로 남아 있다. 그 이유는 많은 온라인 자습서와 여러 서적에서 설명의 편의를 위해 구성 파일에서 직접 자격 증명을 정의하기 때문일 수 있다. 주제의 초점이 다른 곳에 맞춰져 있는 가상의 사례라면 이렇게 하는 게 적절할 것이다.

참고 대부분 개발자는 줄곧 이론적인 예제로 공부하는데, 이러한 예제는 독자가 특정 주제에 집중할 수 있도록 단순화되어 있다. 이 때문에 개발자가 잘못된 접근 방식에 익숙해질 수 있으며, 예제로 배운 모든 내용이 좋은 습관이라고 착각할 수 있다.

이러한 측면은 스프링 시큐리티에 어떤 연관이 있을까? 이 책에서도 자격 증명과 개인 키를 다루는데, 암호를 구성 파일에 넣은 예제도 있지만 중요한 데이터는 볼트에 넣어야 한다는 점을 참고 사항에서 강조할 것이다. 즉, 실제 개발된 시스템에서는 모든 환경에서 이러한 민감한 키 값을 볼 수 없어야 하고 적어도 운영 환경에서는 소수의 사람만 개인 데이터에 접근할 수 있어야 한다.

이러한 값을 스프링 프로젝트의 `application.properties` 또는 `application.yml` 파일 등의 구성 파일에서 설정하면 소스 코드를 볼 수 있는 모든 사람이 이러한 개인 값에 접근할 수 있다. 게다가 소스 코드의 버전 관리 시스템에도 이러한 값의 변경 기록이 저장되는 것을 볼 수 있다.

민감한 데이터의 노출과 관련해서는 애플리케이션에서 콘솔에 기록하거나 스플링크(Splunk)나 일래스틱서치(Elasticsearch)같이 데이터베이스에 저장하는 로그 정보도 있다. 개인적으로 개발자가 로그를 통해 민감한 데이터가 노출되는 것을 놓치는 경우를 종종 본다.

참고 공개 정보가 아닌 것은 절대 로그에 기록하지 말아야 한다. 여기서 공개라는 것은 누구든지 해당 정보에 접근하고 볼 수 있다는 뜻이다. 개인 키나 인증서는 이러한 공개 정보가 아니므로 오류, 경고, 정보 메시지와 함께 로그에 기록하지 말아야 한다.

다음은 로그에 기록해서는 안 될 정보를 포함하는 메시지의 예다.

[오류] 요청의 서명이 잘못되었습니다. 사용할 올바른 키는 X입니다.

[경고] 사용자 이름 X와 암호 Y를 이용하여 로그인하지 못했습니다. 사용자 이름 X의 암호는 Z입니다.

[정보] 사용자 X가 올바른 암호 Y를 이용하여 로그인했습니다.

취약한 종속성은 이용하지 않는 것이 최선이지만, 실수로 취약한 종속성을 이용했더라도 이를 광고할 필요는 없다. 종속성이 취약하다고 알려지지 않은 경우도 아직 취약성이 발견되지 않았기 때문일 수 있다. 앞의 예와 같이 정보가 노출되면 공격자가 해당 특정 버전의 취약성을 찾으려 하는 동기가 생길 수 있다. 즉, 공격자를 시스템으로 초대하는 것일 수 있다. 공격자는 극히 작은 세부 정보까지 활용한다. 다음의 예를 보자.

응답 A:

```
{
  "상태": 401,
  "오류": "권한 없음",
  "메시지": "사용자 이름이 올바르지 않음",
  "경로": "/login "
```

응답 B:

```
{
  "상태": 401,
  "오류": "권한 없음",
  "메시지": "암호가 올바르지 않음",
  "경로": "/login "
```

이 예에서 응답 A와 B는 같은 인증 엔드포인트를 호출한 다른 결과를 보여준다. 클래스 디자인이나 시스템 인프라에 대한 정보는 노출되지 않은 것 같지만 사실은 다른 문제가 숨겨져 있다. 컨텍스트 정보를 공개하는 메시지에는 숨겨진 취약성이 있을 수 있다. 엔드포인트에 제공된 다양한 입력에 대해 다른 메시지를 제공하면 이 메시지를 이용해 실행 컨텍스트를 분석할 수 있다. 예를 들어 사용자 이름은 맞고 암호는 틀린 상황을 식별할 수 있으므로 시스템이 무차별 대입 공격에 더 취약해질 수 있다. 클라이언트로 반환되는 응답이 특정 입력이 무엇인지 추측하게 도와줘서는 안 된다. 즉, 앞의 예에서 두 경우 모두 다음과 같이 동일한 메시지를 제공해야 한다.

```
{
  "상태": 401,
  "오류": "권한 없음",
  "메시지": "사용자 이름 또는 암호가 올바르지 않음",
  "경로": "/login "
```

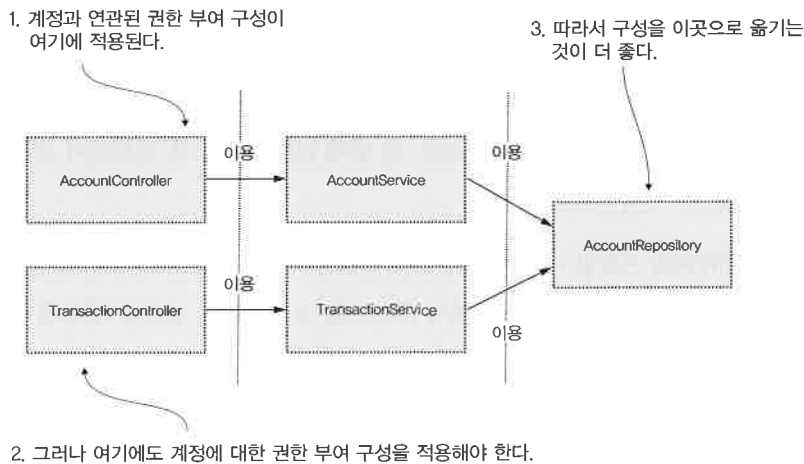


그림 1.10 새로 추가된 TransactionController는 종속성 체인의 AccountRepository를 사용한다. 개발자는 이 컨트롤러에도 권한 부여 규칙을 적용해야 하지만, 리포지토리가 인증된 사용자에게 속하지 않은 데이터를 노출하지 않는 것이 더 좋은 방법이다.

1.4.8 알려진 취약성이 있는 종속성 이용

우리가 이용하는 종속성은 스프링 시큐리티와 직접적인 연관은 없지만 애플리케이션 수준 보안의 중요한 부분이며 주의가 필요하다. 때로는 개발하는 애플리케이션이 아니라 기능을 만들기 위해 이용하는 라이브러리나 프레임워크 같은 종속성에 취약성이 있을 수 있다. 이용하는 종속성을 항상 주의 깊게 살펴보고 알려진 취약성이 있는 버전은 제거해야 한다.

다행히도 메이븐 또는 그레이들 구성에 플러그인을 추가하면 신속하게 정적 분석을 수행할 수 있다. 현재의 애플리케이션은 대부분 오픈 소스 기술을 기반으로 개발된다. 스프링 시큐리티 역시 오픈 소스 프레임워크다. 이 개발 방법론은 빠른 혁신이 가능하지만 동시에 오류가 발생하기 쉽다.

소프트웨어를 개발할 때는 알려진 취약성이 있는 종속성을 이용하지 않도록 필요한 모든 조치를 취해야 한다. 이러한 의존성을 사용한 것이 발견되면 이를 빨리 수정하는 것은 물론 취약성이 이미 애플리케이션에서 악용되었는지 조사하고 필요한 조치를 해야 한다.

스프링을 이용한 웹 애플리케이션 개발과 REST 서비스 개발에 관해 자세히 배우려면 《스프링 인 액션》(제이펍, 2020)의 2장과 6장을 참고하자(아래 URL을 통해 원서의 2장과 6장 내용을 볼 수 있다).

- <https://livebook.manning.com/book/spring-in-action-sixth-edition/chapter-2/>
- <https://livebook.manning.com/book/spring-in-action-sixth-edition/chapter-6/>

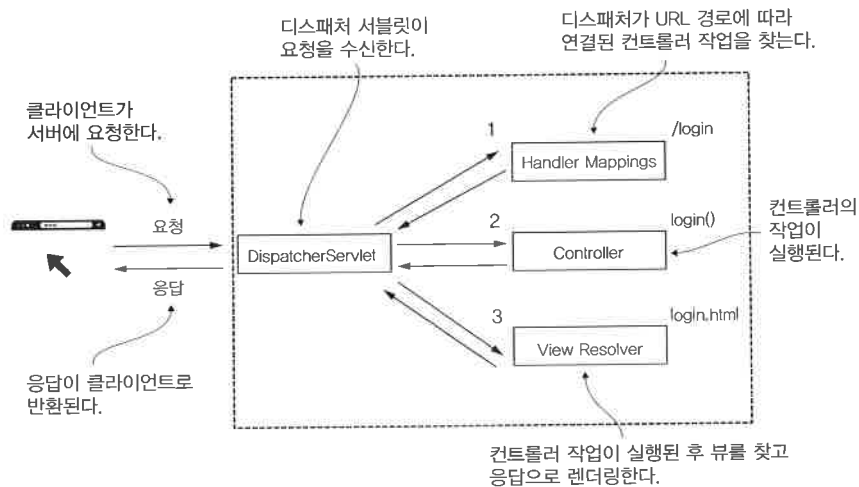


그림 1.11 스프링 MVC 흐름의 간소화된 표현. DispatcherServlet이 요청된 경로와 컨트롤러 메시지의 매핑을 찾고(1), 컨트롤러 메시지를 실행한 후(2), 렌더링된 뷰를 얻는다(3). 요청자에게 HTTP 응답을 반환하면 브라우저가 이를 해석하고 응답을 표시한다.

세션이 있는 한 세션 고정 취약성과 앞서 언급한 CSRF 가능성을 고려해야 하고 HTTP 세션에 저장하는 정보도 고려해야 한다.

서버 쪽 세션은 준 영구적이며 데이터의 상태를 저장하므로 수명이 더 길다. 메모리에 유지되는 시간이 길수록 통계적으로 접근 가능성이 커진다. 예를 들어 힙 덤프에 접근할 수 있는 사람은 앱의 내부 메모리에 있는 정보를 읽을 수 있다. 힙 덤프는 그리 어렵지 않게 얻을 수 있다는 데 주의하자. 특히 스프링 부트로 애플리케이션을 개발할 때는 애플리케이션에 액추에이터를 포함하는 경우가 많다. 스프링 부트 액추에이터는 아주 훌륭한 도구이며 구성 방법에 따라서는 엔드포인트 호출만으로도 힙 덤프를 반환할 수 있다. 즉, 덤프하려는 VM에 대한 루트 접근 권한 없이도 가능하다.

CSRF의 취약성에 관한 관점으로 돌아가서, 이 취약성을 완화하는 가장 쉬운 방법은 CSRF 방지 토큰을 이용하는 것이다. 다행히 이 기능은 스프링 시큐리티에 기본적으로 들어있고 CSRF 보호와 원점

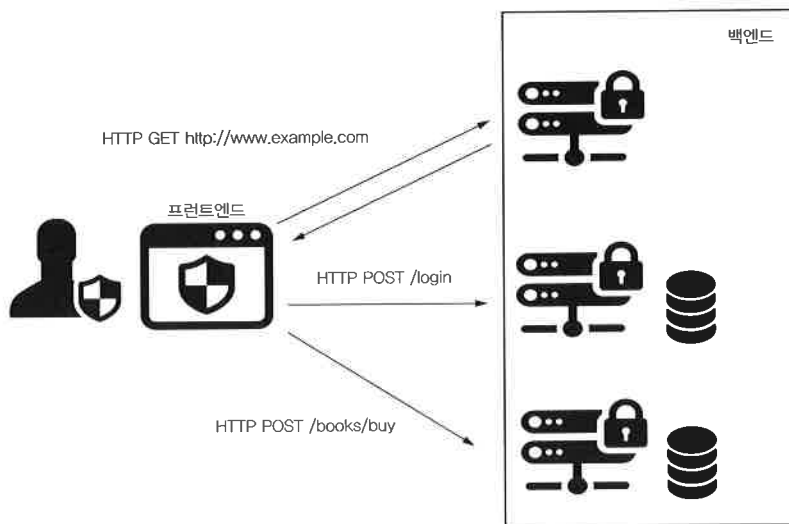


그림 1.12 브라우저가 프론트엔드 애플리케이션을 실행한다. 이 애플리케이션은 백엔드가 노출한 REST 엔드포인트를 호출하여 사용자가 요청한 여러 작업을 수행한다.

보안의 관점에서 고려할 몇 가지 다른 측면이 있다. 첫째, CSRF 및 CORS 구성은 일반적으로 더 복잡하다. 시스템을 수평적으로 확장하기를 원할 수 있는데, 반드시 백엔드와 같은 출처의 프론트엔드를 이용해야 하는 것은 아니다. 모바일 애플리케이션의 경우 출처를 확인할 수조차 없다.

엔드포인트 인증에 HTTP Basic을 이용하는 방법은 실용적이고 가장 간단하지만 바람직하지는 않다. 이 접근법은 이해하기 쉬워 인증을 설명할 때 첫 번째 이론적 예제로 자주 이용되지만 주의해야 할 단점이 있다. HTTP Basic을 이용하려면 호출마다 자격 증명을 전송해야 한다. 2장에서 확인하겠지만 자격 증명은 암호화되지 않는다. 브라우저는 Base64 인코딩을 이용해 사용자 이름과 암호를 전송하므로 각 엔드포인트 호출의 헤더에 자격 증명에 노출된다. 또한 인증 정보가 로그인한 사용자를 나타낸다고 하면 사용자가 모든 요청에 대해 자격 증명을 입력해야 하는 것은 원하지 않을 것이며 자격 증명을 클라이언트 쪽에 저장하는 것도 원하지 않을 것이다. 즉, 이 방식은 권장되지 않는다.

이러한 이유를 고려해 12장에서는 OAuth 2 흐름이라는 더 나은 접근법을 이용하는 인증과 권한 부여의 대안을 살펴본다. 다음 절에서 이 접근법에 대해 간단히 알아본다.

토큰은 사무실에 들어갈 때 이용하는 출입 카드와 비슷하다. 방문자가 건물에 들어가면 먼저 접수대로 가서 신원을 밝히고 출입 카드를 받아야 한다. 출입 카드가 있으면 건물의 일부 지역에 접근할 수 있지만 모든 문을 열 수 있는 것은 아니다. 신원에 따라 허용된 문만 열 수 있으며 나머지 문에는 접근할 수 없다. 액세스 토큰도 마찬가지다. 인증 후 호출자는 토큰을 받고, 이를 바탕으로 이용 권리가 있는 리소스에 접근할 수 있다.

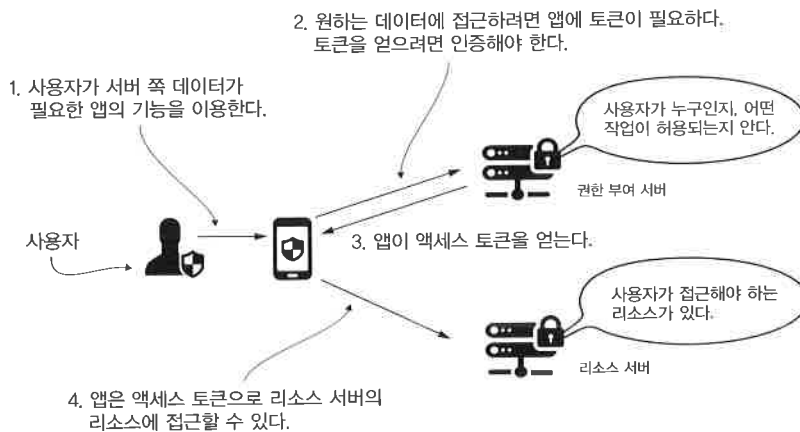


그림 1.13 암호 그랜트 유형을 이용하는 OAuth 2 권한 부여 흐름. 사용자가 요청한 작업을 실행하기 위해(1) 애플리케이션은 권한 부여 서버에서 액세스 토큰을 받아야 한다(2). 애플리케이션은 토큰을 받고(3) 액세스 토큰을 이용해 리소스 서버의 리소스에 접근한다(4).

토큰의 수명은 고정되고 일반적으로 오래 유지되지 않으며 토큰이 만료되면 앱이 새 토큰을 받아야 한다. 서버는 필요한 경우 토큰의 만료 시간보다 일찍 토큰을 실격시킬 수 있다. 다음 목록은 이 흐름의 몇 가지 이점을 나열한 것이다.

- 클라이언트는 사용자 자격 증명을 저장할 필요 없이 액세스 토큰과 (최종적으로) 갱신 토큰만 저장하면 된다.
- 애플리케이션은 사용자 자격 증명 (종종 네트워크에서) 노출하지 않는다.
- 누군가가 토큰을 가로채면 사용자 자격 증명을 무효로 할 필요 없이 토큰을 실격시킬 수 있다.
- 토큰을 이용하면 제삼자가 사용자를 가장하지 않고도 사용자 대신 리소스에 접근할 수 있다. 물론 이 경우 공격자가 토큰을 훔칠 수 있지만 토큰은 일반적으로 수명이 제한되므로 이 취약성을 악용할 수 있는 기간도 제한된다.

참고 여기에서는 간단한 개요를 제공하기 위해 **암호 그랜트 유형**이라는 OAuth 2 흐름을 설명했다. OAuth 2는 여러 그랜트 유형을 정의하며 12장부터 15장까지 배우겠지만 클라이언트 애플리케이션에 반드시 자격 증명에 있는 것은 아니다. 승인 코드 그랜트를 이용하면 애플리케이션은 인증을 브라우저에서 권한 부여 서버가 구현하는 로그인으로 리디렉션한다. 이에 대한 자세한 내용은 책의 뒷부분에 있다.

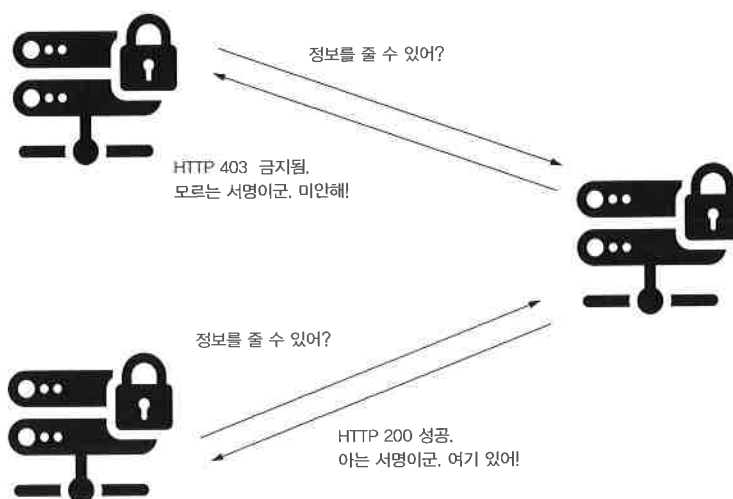


그림 1.14 다른 백엔드를 호출하려면 올바른 서명과 공유된 키를 가지고 요청해야 한다.

요청이 들어오는 특정 주소 또는 주소의 범위를 아는 경우 앞서 언급한 솔루션 중 하나를 이용해 IP 주소 검증을 적용할 수 있다. 이 방법은 애플리케이션이 수락하도록 구성한 IP 주소 외의 주소에서 오는 요청을 거부한다고 가정한다. 그러나 대부분의 경우 IP 검증은 애플리케이션 수준에서 수행되지 않고 훨씬 이전의 네트워킹 계층에서 수행된다.

1.6 이 책에서 배울 내용

이 책에서는 스프링 시큐리티를 배우는 실용적 접근법을 소개한다. 이 책의 나머지 부분에서는 스프링 시큐리티를 단계별로 깊이 파고들어 간단한 예제부터 복잡한 예제까지 개념을 증명한다. 이 책을 최대한 활용하려면 자바 프로그래밍과 스프링 프레임워크에 익숙해야 한다. 아직 스프링 프레임워크를 이용해보지 않았거나 기본에 익숙하지 않다고 느끼는 독자는 《스프링 인 액션》과 《스프링 부트 코딩 공작소》(길벗, 2016)를 먼저 공부하기를 권장한다. 이 책에서 배울 내용을 정리하면 다음과 같다.

- 스프링 시큐리티의 아키텍처와 기본 구성 요소 및 이를 이용해 애플리케이션을 보호하는 방법
- OAuth 2 및 OpenID Connect 흐름을 비롯해 스프링 시큐리티로 인증과 권한 부여를 구현하는 방법과 운영 준비 단계의 애플리케이션에 이를 적용하는 방법
- 애플리케이션의 다양한 계층에서 스프링 시큐리티로 보안을 구현하는 방법

2장

안녕! 스프링 시큐리티

이 단원의 내용

- 스프링 시큐리티로 첫 번째 프로젝트 만들기
- 인증과 권한 부여를 위한 기본 구성 요소로 간단한 기능 설계
- 이러한 구성 요소가 서로 어떻게 연관되는지 이해하기 위해 기본 계약 적용
 - 주 책임에 대한 구현 작성
- 스프링 부트의 기본 구성 재정의

스프링 부트는 스프링 프레임워크를 이용한 애플리케이션 개발에서 혁신적인 단계로 평가받는다. 스프링 부트는 미리 준비된 구성을 제공하므로 모든 구성을 작성하는 대신 자신의 구현과 일치하지 않는 구성만 재정의하면 된다. 이 접근법을 설정보다 관습(convention-over-configuration)이라고 한다.

이 애플리케이션 개발 방법이 보급되기 전에는 개발자가 새로운 앱을 개발할 때마다 매번 수십 행의 비슷한 코드를 반복해서 입력했다. 과거 대부분의 아키텍처를 모놀리식으로 개발할 때는 이러한 상황이 그리 문제가 되지 않았다. 모놀리식 아키텍처에서는 처음 한 번만 구성을 작성하며 나중에 수정하는 경우가 드물었다. 서비스 지향 소프트웨어 아키텍처가 발전함에 따라 각 서비스를 구성하는 상용구 코드를 작성하는 일이 고통스럽게 느껴지기 시작했다. 이 주제에 관심이 있다면 《스프링 프레임워크의 설계》(ITC, 2015)의 3장을 읽어보자. 조금 오래된 책의 이 단원은 스프링 3로 웹 애플리케이션을 작성하는 방법을 다루는데, 작은 싱글 페이지 웹 애플리케이션을 만들기 위해 얼마나 많은 구성을 작성해야 하는지 확인할 수 있다. 다음 주소에서 이 원서의 해당 단원을 볼 수 있다.

- <https://livebook.manning.com/book/spring-in-practice/chapter-3/>

링 부트로 웹 앱을 만드는 과정이 정확하게 설명돼 있다. 다음 주소에서 원서의 해당 단원을 볼 수 있다: <https://livebook.manning.com/book/spring-boot-in-action/chapter-2/>.

이 책의 예제는 미리 작성된 소스 코드를 참조한다. 각 예제마다 pom.xml 파일에 추가해야 하는 종속성도 지정되어 있다. 이 책에서 다루는 프로젝트의 소스 코드는 <https://github.com/wikibook/spring-security>에서 다운로드할 수 있으므로 미리 받아두면 직접 진행하다가 막히는 부분이 있을 때 도움이 될 것이다. 제공된 프로젝트는 자신이 완성한 솔루션을 검증하는 데도 사용할 수 있다.

참고 이 책의 예제는 어느 빌드 툴을 선택하든 영향을 받지 않으며 메이븐이나 그레이들을 이용할 수 있다. 다만 일관성을 위해 필자는 모든 예제에 메이븐을 이용했다.

첫 번째 예제는 이 책에서 가장 작은 예제이며, 호출하면 그림 2.1과 같이 응답을 반환하는 REST 엔드포인트 하나를 노출하는 간단한 애플리케이션이다. 이 프로젝트는 스프링 시큐리티로 애플리케이션을 개발하는 첫 번째 단계를 배우는 데 충분하며 인증과 권한 부여를 위한 스프링 시큐리티 아키텍처의 기본 사항을 보여준다.

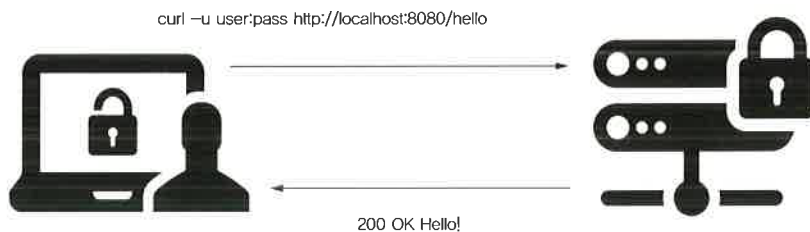


그림 2.1 첫 번째 예제는 HTTP Basic을 이용해 엔드포인트에 대해 사용자를 인증하고 권한을 부여한다. 이 애플리케이션은 정의된 경로(/hello)에 REST 엔드포인트를 노출한다. 호출이 성공하려면 응답이 HTTP 200 상태 메시지와 본문을 반환해야 한다. 이 예제는 스프링 시큐리티의 기본 인증과 권한 부여 구성이 어떻게 작동하는지 보여준다.

가장 먼저 비어 있는 프로젝트를 만들고 이름을 ssia-ch2-ex1로 지정한다. (이 책과 함께 제공되는 프로젝트에 같은 이름의 예제가 있다.) 예제 2.1에서 볼 수 있듯이 첫 번째 프로젝트에는 spring-boot-starter-web 및 spring-boot-starter-security 종속성만 있으면 된다. 프로젝트를 만든 후 이러한 종속성을 pom.xml 파일에 추가한다. 이 프로젝트의 주된 목적은 스프링 시큐리티의 기본 구성 애플리케이션이 어떻게 작동하는지 확인하고 이 기본 구성에 속한 구성 요소와 그 목적을 알아보는 것이다.

야 한다고 지정한다. @GetMapping 어노테이션은 /hello 경로를 구현된 메서드에 매핑한다. 애플리케이션을 실행하면 콘솔에 다음과 비슷한 결과가 표시된다.

```
Using generated security password: 93a01cf0-794b-4b98-86ef-54860f36f7f3
```

애플리케이션을 실행할 때마다 새 암호가 생성되고 이전 코드와 비슷하게 표시된다. HTTP Basic 인증으로 애플리케이션의 엔드포인트를 호출하려면 이 암호를 이용해야 한다. 먼저 Authorization 헤더를 이용하지 않고 엔드포인트를 호출해보자.

```
curl http://localhost:8080/hello
```

참고 이 책에서는 모든 예제의 엔드포인트를 호출하는 데 cURL을 이용한다. 개인적으로는 cURL이 가장 이용하기 쉬운 솔루션이라고 생각하지만, 선호하는 다른 툴이 있으면 써도 된다. 예를 들어 알아보기 쉬운 그래픽 인터페이스를 원한다면 포스트맨(Postman)이 좋은 선택이다. 이용 중인 운영 체제에 이러한 툴이 설치되지 않은 경우 직접 설치해야 할 수 있다.

호출에 대한 응답은 다음과 같다.

```
{
  "status":401,
  "error":"Unauthorized",
  "message":"Unauthorized",
  "path":"/hello"
}
```

응답 상태로 HTTP 401 권한 없음이 반환됐다. 인증을 위한 올바른 자격 증명을 제공하지 않았기 때문에 예상된 결과다. 기본적으로 스프링 시큐리티는 기본 사용자 이름(user)과 제공된 암호(이 책의 경우 93a01로 시작하는 암호)를 사용할 것으로 예상된다. 이번에는 올바른 자격 증명을 지정하고 다시 호출해보자.

```
curl -u user:93a01cf0-794b-4b98-86ef-54860f36f7f3 http://localhost:8080/hello
```

호출 결과는 다음과 같다.

Hello!

기본 프로젝트에는 주의할 별다른 보안 구성이 없다. 첫 번째 예제에서는 기본 구성으로 올바른 종속성이 이용되도록 하는 것이 주된 내용이므로 인증과 권한 부여 측면에서는 별다른 작업을 하지 않았다. 물론 이 구현은 운영 단계의 애플리케이션에서 볼 수 있는 수준은 아니지만 출발점 역할의 기본 프로젝트로는 충분하다.

첫 번째 예제에서는 스프링 시큐리티가 작동하는 것을 확인한 것으로 만족하자. 다음 단계에서는 요구 사항을 프로젝트에 적용하기 위해 구성을 변경해보자. 먼저 스프링 부트가 스프링 시큐리티의 무엇을 구성하는지 알아보고 구성을 재정의하려면 어떻게 해야 하는지 확인해보자.

2.2 기본 구성이란?

이 절에서는 전체 아키텍처에서 인증과 권한 부여를 처리하는 데 참여하는 주 구성 요소에 대해 논의한다. 그 이유는 이러한 사전 구성된 요소를 애플리케이션의 필요에 맞게 재정의해야 하기 때문이다. 먼저 스프링 시큐리티 아키텍처가 인증과 권한 부여 측면에서 어떻게 작동하는지 알아본 후 배운 내용을 이 단원의 프로젝트에 적용한다. 전체 내용을 한꺼번에 배우기는 부담스러우므로 먼저 각 구성 요소의 개요를 그림으로 살펴보자. 각 구성 요소에 대한 자세한 내용은 이후 단원에서 배운다.

2.1절에서 인증과 권한 부여를 위한 몇 가지 논리를 살펴보았다. 예제에서는 기본 사용자를 이용했고 애플리케이션이 시작될 때마다 임의의 암호를 받았는데 이 기본 사용자와 암호로 엔드포인트를 호출할 수 있었다. 그렇다면 이 논리는 어디에 구현되어 있을까? 짐작할 수 있겠지만, 스프링 부트는 우리가 이용하는 종속성에 따라 몇 가지 구성 요소를 대신 설정해준다.

그림 2.2를 보면 스프링 시큐리티 아키텍처의 주 구성 요소와 이들 간의 관계를 큰 그림으로 볼 수 있다. 첫 번째 프로젝트에서는 이러한 구성 요소의 사전 구성된 구현을 이용했다. 이 단원에서는 애플리케이션에서 스프링 부트가 스프링 시큐리티의 무엇을 구성하는지 살펴보고 인증 흐름에 참여하는 엔티티 간의 관계를 알아본다.

그림 2.2에도 해당 항목이 있다. 인증 공급자는 이러한 빈을 이용해 사용자를 찾고 암호를 확인한다. 인증에 필요한 자격 증명을 제공하는 방법부터 시작해보자.

사용자에 관한 세부 정보는 스프링 시큐리티로 UserDetailsService 계약을 구현하는 객체가 관리한다. 지금까지는 스프링 부트가 제공하는 기본 구현을 사용했는데, 이 구현은 애플리케이션의 내부 메모리에 기본 자격 증명을 등록하는 일만 한다. 이러한 기본 자격 증명에서 사용자 이름은 'user'이고 기본 암호는 UUID(Universally Unique Identifier) 형식이며 암호는 스프링 컨텍스트가 로드될 때 자동으로 생성된다. 현재 애플리케이션은 암호를 볼 수 있도록 콘솔에 출력한다. 따라서 이 단원에서 방금 작성한 예제에 사용할 수 있다.


이 기본 구현은 개념 증명의 역할을 하며 종속성이 작동하는 것을 확인해준다. 이 구현은 자격 증명을 메모리에 보관한다. 즉 애플리케이션은 자격 증명을 보존하지 않는다. 이 접근법은 예제나 개념 증명에 적합하지만 운영 단계 애플리케이션에서는 피해야 한다.

다음으로 PasswordEncoder가 있다. PasswordEncoder는 두 가지 일을 한다.

- 암호를 인코딩한다.
- 암호가 기존 인코딩과 일치하는지 확인한다.

UserDetailsService 객체와 마찬가지로, PasswordEncoder 객체도 Basic 인증 흐름에 꼭 필요하다. 가장 단순한 구현에서는 암호를 일반 텍스트로 관리하고 인코딩하지 않는다. 이 객체의 구현에 관해서는 4장에서 자세히 다룬다. 현재로서는 PasswordEncoder가 기본 UserDetailsService와 함께 존재한다고만 알아두자. UserDetailsService의 기본 구현을 대체할 때는 PasswordEncoder도 지정해야 한다.

스프링 부트는 기본 HTTP Basic 접근 인증을 구성할 때 인증 방식도 선택하며 이는 가장 직관적인 접근 인증 방식이다. Basic 인증에서는 클라이언트가 사용자 이름과 암호를 HTTP Authorization 헤더를 통해 보내기만 하면 된다. 클라이언트는 헤더 값에 접두사 Basic을 붙이고 그 뒤에 콜론(:)으로 구분된 사용자 이름과 암호가 포함된 문자열을 Base64 인코딩하고 붙인다.

참고  HTTP Basic 인증은 자격 증명의 기밀성을 보장하지 않는다. Base64는 단지 전송의 편의를 위한 인코딩 방법이고 암호화나 해싱 방법이 아니므로 전송 중에 자격 증명을 가로채면 누구든지 볼 수 있다. 일반적으로는 최소한의 기밀을 위해 HTTPS를 함께 이용할 때가 아니면 HTTP Basic 인증은 이용하지 않는다. HTTP Basic의 자세한 정의는 RFC 7617(<https://tools.ietf.org/html/rfc7617>)에서 볼 수 있다.

```
winpty openssl req -newkey rsa:2048 -x509 -keyout key.pem -out cert.pem -days 365
winpty openssl pkcs12 -export -in cert.pem -inkey key.pem -out certificate.p12
-name "certificate"
```

자체 서명 인증서가 준비되면 엔드포인트를 위해 HTTPS를 구성할 수 있다. certificate.p12 파일을 스프링 부트 프로젝트의 resources 폴더에 복사하고 application.properties 파일에 다음 행을 추가한다.

```
server.ssl.key-store-type=PKCS12
server.ssl.key-store=classpath:certificate.p12
server.ssl.key-store-password=12345
```

← 암호의 값은 PKCS12 인증서 파일을 생성하는 두 번째 명령을 실행할 때 지정한 것이다.

인증서를 생성하는 명령어를 실행하면 암호를 입력하라는 메시지가 표시되므로(여기에서는 '12345'로 지정) 이 책에는 표시되지 않았다. 이제 애플리케이션에 테스트용 엔드포인트를 추가하고 HTTPS를 이용해 호출한다.

```
@RestController
public class HelloController {
    @GetMapping("/hello")
    public String hello() {
        return "Hello!";
    }
}
```

자체 서명 인증서를 이용할 때는 엔드포인트를 호출하는 툴에서 인증서 신뢰성 테스트를 생각하도록 구성해야 한다. 툴이 인증서의 신뢰성을 테스트하면 인증서를 정식으로 인식하지 않고 호출이 작동하지 않는다. cURL에서는 -k 옵션을 이용해 인증서의 신뢰성 테스트를 생략할 수 있다.

```
curl -k https://localhost:8080/hello
```

호출 응답은 다음과 같다.

```
Hello!
```

HTTPS를 이용해도 시스템의 구성 요소 간 통신이 완벽하게 보호되지는 않는다는 점을 기억하자. "HTTPS를 이용하니까 이제 암호화할 필요는 없어."라는 말을 자주 듣는다. HTTPS는 통신을 보호하는 데 도움이 되기는 하지만, 시스템을 보호할 벽을 세울 벽돌에 지나지 않는다. 항상 책임감을 갖고 시스템 보안을 처리하고 그와 관련된 모든 계층에 주의를 기울여야 한다.

참고 자바의 인터페이스는 객체 간의 계약을 정의하며 애플리케이션의 클래스 디자인에서 서로를 이용하는 개체를 분리하는 역할을 한다. 이 책에서는 이러한 인터페이스의 특징을 강조하기 위해 주로 계약이라는 용어를 이용한다.

여기서 선택한 구현으로 이 구성 요소를 재정의하는 방법을 보여주기 위해 첫 번째 예제와는 다르게 작업할 것이다. 이를 통해 자체적으로 관리하는 자격 증명을 인증에 이용할 수 있다. 이 예제에서는 직접 클래스를 구현하지는 않고 스프링 시큐리티에 있는 `InMemoryUserDetailsManager` 구현을 이용한다. 이 구현은 `UserDetailsService`보다 복잡하지만, 여기에서는 `UserDetailsService`와 다를 바 없이 이용한다. 이 구현은 메모리에 자격 증명을 저장해서 스프링 시큐리티가 요청을 인증할 때 이용할 수 있게 한다.

참고 `InMemoryUserDetailsManager` 구현은 운영 단계 애플리케이션을 위한 것은 아니며 예제나 개념 증명용으로 좋은 툴이다. 사용자만으로 충분할 때는 다른 부분을 구현하는 데 시간을 쓸 필요가 없다. 여기에서는 `UserDetailsService` 구현을 재정의하는 방법을 알아보는 것이 목적이다.

먼저 구성 클래스를 정의한다. 일반적으로는 `config`라는 별도의 패키지에 구성 클래스를 선언한다. 예제 2.3에 구성 클래스의 정의가 나온다. `ssia-ch2-ex2` 프로젝트에서도 예제를 볼 수 있다.

참고 이 책의 예제는 최신 장기 지원 자바 버전인 자바 11에 맞게 설계됐다. 앞으로 더 많은 애플리케이션이 자바 11로 개발될 것이므로 이 책에서도 이 버전을 이용했다.

이 책의 코드에서는 `var`를 자주 이용한다. `var`는 자바 10에 도입된 예약 형식 이름이며 로컬 선언에만 이용한다. 이 책에서는 코드를 간단하게 만들고 변수 형식을 숨기기 위해 이용했다. `var`가 숨긴 형식에 대해서는 이후 단원에서 논의할 것이므로 그때까지는 신경 쓰지 않아도 된다.

예제 2.3 `UserDetailsService` 빈에 대한 구성 클래스

```
@Configuration
public class ProjectConfig {

    @Bean
    public UserDetailsService userDetailsService() {
        var userDetailsService =
            new InMemoryUserDetailsManager();
    }
}
```

`@Configuration` 어노테이션은 클래스를 구성 클래스로 표시한다.

`@Bean` 어노테이션은 반환된 값을 스프링 컨텍스트에 빈으로 추가하도록 스프링에 지시한다.

`var` 키워드는 구문을 간소하게 만들어주고 세부 정보를 감춘다.

```

public UserDetailsService userDetailsService() {
    var userDetailsService =
        new InMemoryUserDetailsManager();

    var user = User.withUsername("john")
        .password("12345")
        .authorities("read")
        .build();
    userDetailsService.createUser(user);
    return userDetailsService;
}

```

주어진 사용자 이름, 암호, 권한 목록으로 사용자 생성

← UserDetailsService에서 관리하도록 사용자 추가

참고 User 클래스는 org.springframework.security.core.userdetails 패키지에 있으며 사용자를 나타내는 객체를 만드는 빌더 구현이다. 또한 이 책에서는 예제 코드에 특정 클래스를 작성하는 방법을 언급하지 않으면 스프링 시큐리티에 기본적으로 있다는 의미다.

예제 2.4에 나온 것처럼 사용자 이름과 암호에 값을 지정하고 권한에도 하나 이상의 값을 지정해야 한다. 하지만 아직은 엔드포인트를 호출할 수 없다. PasswordEncoder도 선언해야 한다.

기본 UserDetailsService를 이용하면 PasswordEncoder도 자동 구성되지만, UserDetailsService를 재정의 하면 PasswordEncoder도 다시 선언해야 한다. 현재 상태로 예제를 실행하면 엔드포인트를 호출할 때 예외가 발생한다. 인증하려고 하면 스프링 시큐리티는 암호를 관리하는 방법을 모른다는 것을 인식하고 오류를 생성한다. 다음과 같은 예외가 애플리케이션의 콘솔에 표시된다. 클라이언트는 HTTP 401 권한 없음 메시지와 빈 응답 본문을 받는다.

```
curl -u john:12345 http://localhost:8080/hello
```

애플 콘솔에는 다음과 같은 호출 결과가 표시된다.

```

java.lang.IllegalArgumentException: There is no PasswordEncoder mapped for the id "null"
    at org.springframework.security.crypto.password
    ↳.DelegatingPasswordEncoder$UnmappedIdPasswordEncoder

```

```

    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return NoOpPasswordEncoder.getInstance();
    }
}

```

PasswordEncoder를 컨텍스트에 추가하기 위해 @Bean 어노테이션이 지정된 새 메서드

이제 사용자 이름이 john이고 암호가 12345인 새 사용자로 엔드포인트를 호출해보자.

```

curl -u john:12345 http://localhost:8080/hello
Hello!

```

참고 단위 테스트와 통합 테스트의 중요성을 아는 독자들은 예제를 위한 테스트를 작성하지 않는 이유가 궁금할 것이다. 사실 이 책의 모든 예제에 대해 스프링 시큐리티 통합 테스트가 함께 제공되고 있다. 그러나 각 장에 제시된 주제에 집중할 수 있도록 스프링 시큐리티 통합 테스트에 대한 논의를 분리해서 20장에서 자세히 설명했다.

2.3.2 엔드포인트 권한 부여 구성 재정의

2.3.1절에서 설명한 것처럼 사용자에게 대한 새로운 관리 방식을 소개했으나 이제 엔드포인트의 인증 방식과 구성에 관해 논의할 수 있다. 권한 부여 구성에 대해서는 7장부터 9장까지 자세히 다루겠지만 자세한 내용을 배우기 전에 큰 그림을 이해해야 한다. 가장 좋은 방법은 첫 번째 예제를 다시 이용하는 것이다. 기본 구성에서 모든 엔드포인트는 애플리케이션에서 관리하는 유효한 사용자가 있다고 가정한다. 또한 기본적으로 HTTP Basic 인증을 권한 부여 방법으로 이용하지만 이 구성은 손쉽게 재정의할 수 있다.

다음 단원에서 배우겠지만 HTTP Basic 인증은 대부분의 애플리케이션 아키텍처에 적합하지 않다. 종종 애플리케이션에 맞게 변경하고 싶을 때가 있다. 마찬가지로 애플리케이션의 모든 엔드포인트를 보호할 필요는 없으며 보안이 필요한 엔드포인트에 다른 권한 부여 규칙을 선택해야 할 수도 있다. 이러한 변경을 위해 WebSecurityConfigurerAdapter 클래스를 확장하는 것부터 시작해보자. 이 클래스를 확장하면 다음 예제 코드에 나온 것처럼 configure(HttpSecurity http) 메서드를 재정의할 수 있다. 이 예제에서는 ssia-ch2-ex2 프로젝트의 코드를 계속 작성한다.

```
// 생략된 코드

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.httpBasic();
    http.authorizeRequests()
        .anyRequest().permitAll();
}
}
```

인증 없이 요청할 수 있다.

이제는 자격 증명 없이 /hello 엔드포인트를 호출할 수 있다. 구성에서 anyRequest() 메서드와 함께 permitAll()을 호출하면 모든 엔드포인트를 자격 증명 없이 접근할 수 있게 된다.

```
curl http://localhost:8080/hello
```

호출의 응답 본문은 다음과 같다.


```
Hello!
```

이 예제의 목적은 기본 구성을 재정의하는 방법을 이해하는 것이며 권한 부여에 대한 자세한 내용은 7장과 8장에서 다룬다.

2.3.3 다른 방법으로 구성 설정

스프링 시큐리티의 구성을 작성할 때 혼동되는 점은 여러 가지 방법으로 같은 구성을 만들 수 있다는 것이다. 이 절에서는 UserDetailsService와 PasswordEncoder를 구성하는 다른 방법을 배운다. 이 책이나 다른 블로그와 설명서의 예제를 볼 때 관련 내용을 이해하려면 다른 방법도 숙지하고 있어야 한다. 애플리케이션에서 이런 다른 방법을 이용하는 절차와 때를 이해하는 것도 중요하다. 이후 단원에서 이 절에서 소개한 내용을 확장하는 예제를 추가로 다룬다.

다시 첫 번째 프로젝트를 살펴보자. 기본 애플리케이션을 만든 후에 새 구현을 스프링 컨텍스트에 빈으로 추가해 UserDetailsService와 PasswordEncoder를 재정의했다. 이제 UserDetailsService와 PasswordEncoder에 대해 동일한 구성을 수행하는 다른 방법을 찾아보자.

참고 WebSecurityConfigurerAdapter 클래스에는 오버로드된 세 가지 다른 configure() 메서드가 있으며 예제 2.9에  서는 예제 2.8과 다른 메서드를 재정의했다. 이 세 가지 메서드는 다음 단원에서 자세히 살펴본다.

예제 2.10 구성 클래스의 전체 정의

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        var userDetailsService =
            new InMemoryUserDetailsManager(); ← InMemoryUserDetailsManager의 인스턴스 생성

        var user = User.withUsername("john")
            .password("12345")
            .authorities("read")
            .build(); ← 새 사용자 생성

        userDetailsService.createUser(user); ← UserDetailsService에서 관리하도록 사용자 추가

        auth.userDetailsService(userDetailsService)
            .passwordEncoder(
                NoOpPasswordEncoder.getInstance()); ← UserDetailsService 및 PasswordEncoder 구성
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic();
        http.authorizeRequests()
            .anyRequest().authenticated(); ← 모든 요청에 인증을 요구하도록 지정
    }
}
```

이 구성 옵션은 모두 사용이 가능하다. 컨텍스트에 빈을 추가하는 첫 번째 옵션을 이용하면 필요할 가능성이 있는 다른 클래스에 값을 주입할 수 있다. 하지만 그럴 필요가 없을 때는 두 번째 옵션도 마찬가지로 좋다. 하지만 구성을 혼합하면 헷갈릴 수 있어 권장하지 않는다. 예를 들어 다음 예제 코드를 보면 UserDetailsService 및 PasswordEncoder의 연결이 어디인지 알기가 쉽지 않다.

이 예제는 인-메모리 방식으로 사용자를 구성했기 때문에 괜찮게 보일 수 있지만, 운영 단계 애플리케이션은 상황이 다르며 보통은 사용자를 데이터베이스에 저장하거나 다른 시스템에서 가져와야 한다. 이 경우처럼 구성이 아주 길고 복잡해질 수 있다. 예제 2.12에는 인-메모리 사용자를 위한 구성을 작성하는 방법이 나온다. ssia-ch2-ex4 프로젝트에 이 예제가 적용된 것을 볼 수 있다.

예제 2.12 인-메모리 사용자 관리 구성

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication()
        .withUser("john")
        .password("12345")
        .authorities("read")
        .and()
        .passwordEncoder(NoOpPasswordEncoder.getInstance());
}
```

가능하면 애플리케이션의 책임을 분리해서 작성하는 것이 좋기 때문에 일반적으로 이 접근 방식은 권장하지 않는다.

2.3.4 AuthenticationProvider 구현 재정의

지금까지 살펴본 것처럼 스프링 시큐리티 구성 요소는 상당히 유연하므로 애플리케이션 아키텍처에 적용할 때 다양한 옵션을 선택할 수 있다. 지금까지는 스프링 시큐리티 아키텍처에서 UserDetailsService 및 PasswordEncoder의 목적과 이를 구성하는 여러 방법을 배웠다. 이제 이들 구성 요소에 작업을 위임하는 AuthenticationProvider도 맞춤 구성할 수 있다는 것을 배울 차례다.

예제 2.13 AuthenticationProvider 인터페이스 구현

```

@Component
public class CustomAuthenticationProvider implements AuthenticationProvider {

    @Override
    public Authentication authenticate
    ↪(Authentication authentication) throws AuthenticationException {

        // 인증 논리를 추가할 위치
    }

    @Override
    public boolean supports(Class<?> authenticationType) {

        // Authentication 형식의 구현을 추가할 위치
    }
}

```

authenticate(Authentication authentication) 메서드는 인증의 전체 논리를 나타내므로 예제 2.14와 같은 구현을 추가하면 된다. supports() 메서드를 이용하는 방법은 5장에서 자세히 다룬다. 현재는 그다지 중요한 내용이 아니므로 이 구현을 그대로 이용하자.

예제 2.14 인증 논리의 구현

```

@Override
public Authentication authenticate(Authentication authentication)
    throws AuthenticationException {

    String username = authentication.getName();
    String password = String.valueOf(authentication.getCredentials());

    if ("john".equals(username) &&
        "12345".equals(password)) {
        return new UsernamePasswordAuthenticationToken
        ↪(username, password, Arrays.asList());
    } else {
        throw new AuthenticationCredentialsNotFoundException("Error in authentication!");
    }
}

```

Principal 인터페이스의 getName() 메서드를 Authentication에서 상속받는다.

이 조건은 일반적으로 UserDetailsService 및 PasswordEncoder를 호출해서 사용자 이름과 암호를 테스트한다.

예제 2.16 새로운 AuthenticationProvider 구현 등록

```

@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private CustomAuthenticationProvider authenticationProvider;

    @Override
    protected void configure(AuthenticationManagerBuilder auth) {
        auth.authenticationProvider(authenticationProvider);
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic();
        http.authorizeRequests().anyRequest().authenticated();
    }
}

```

이제 인증 논리에 정의된 유일하게 인식된 사용자 존과 암호 12345를 이용해 엔드포인트를 호출할 수 있다.

```
curl -u john:12345 http://localhost:8080/hello
```

응답 본문은 다음과 같다.

```
Hello!
```

5장에서는 AuthenticationProvider에 대해 자세히 알아보고 인증 프로세스에서 이 동작을 재정의하는 방법을 배운다. 또한 Authentication 인터페이스와 그것의 구현인 UserPasswordAuthenticationToken 등에 대해 알아본다.

2.3.5 프로젝트에 여러 구성 클래스 이용

앞서 구현한 여러 예제에서는 하나의 구성 클래스만 사용했다. 하지만 구성 클래스도 책임을 분리하는 것이 좋다. 이러한 분리가 필요한 이유는 구성이 복잡해지기 때문이다. 운영 단계 애플리케이션에는 당

예제 2.18 권한 부여 관리를 위한 구성 클래스 정의

```
@Configuration
public class WebAuthorizationConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic();
        http.authorizeRequests().anyRequest().authenticated();
    }
}
```

여기에서 `WebAuthorizationConfig` 클래스는 `WebSecurityConfigurerAdapter`를 확장하고 `configure` (`HttpSecurityhttp`) 메서드를 재정의해야 한다.

참고 여기에서 두 클래스가 모두 `WebSecurityConfigurerAdapter`를 확장할 수는 없으며 그렇게 하면 종속성 주입이 실패한다. `@Order` 어노테이션으로 주입의 우선순위를 설정하면 종속성 주입 문제는 해결할 수 있지만 구성이 병합되지 않고 서로를 제외하므로 기능상 작동하지 않는다.

요약

- 스프링 시큐리티를 애플리케이션의 종속성으로 추가하면 스프링 부트가 약간의 기본 구성을 제공한다.
- 인증과 권한 부여를 위한 기본 구성 요소인 `UserDetailsService`, `PasswordEncoder`, `AuthenticationProvider`를 구현했다.
- `User` 클래스로 사용자를 정의할 수 있다. 사용자는 사용자 이름, 암호, 권한을 가져야 한다. 권한은 사용자가 애플리케이션의 컨텍스트에서 수행할 수 있는 작업을 지정한다.
- 스프링 시큐리티는 `UserDetailsService`의 간단한 구현인 `InMemoryUserDetailsManager`를 제공한다. `UserDetailsService`의 인스턴스와 같은 사용자를 추가해서 애플리케이션의 메모리에서 사용자를 관리할 수 있다.
- `NoOpPasswordEncoder`는 `PasswordEncoder` 계약을 구현하며 암호를 일반 텍스트로 처리한다. 이 구현은 학습 예제와 개념 증명에 적합하지만 운영 단계 애플리케이션에는 적합하지 않다.
- `AuthenticationProvider` 계약을 이용해 애플리케이션의 맞춤형 인증 논리를 구현할 수 있다.
- 구성을 작성하는 방법은 여러 가지가 있지만, 한 애플리케이션에서는 한 방법을 선택하고 고수해야 코드를 깔끔하고 이해하기 쉽게 만들 수 있다.



15부 休養시간

오늘날 많은 앱, 특히 클라우드에 배포된 시스템이 OAuth 2 사양에 따라 인증과 권한 부여를 구현하고 있다. 12~15장에서는 OAuth 2에서 스프링 시큐리티를 이용해 인증과 권한 부여를 구현하는 방법을 배운다. 16장과 17장에서는 메서드 수준에 권한 부여 규칙을 적용하는 방법을 설명한다. 이 접근법으로 스프링 시큐리티에 관해 배운 내용을 웹 이외의 앱에 적용할 수 있다. 또한 웹 앱에도 유연하게 제한을 적용할 수 있다. 19장에서는 스프링 시큐리티를 리액티브 앱에 적용한다. 또한 테스트 없는 개발 프로세스는 없으므로 20장에서는 보안 구현을 위한 통합 테스트를 작성하는 방법을 배운다.

2부 전체의 단원에서 당면한 주제를 다루는 풍부한 정보를 접할 수 있을 것이다. 각 단원에서 배운 내용을 다시 확인하고, 논의한 주제가 서로 얼마나 잘 맞는지 이해하며, 새로운 내용에 대한 애플리케이션을 배우는 데 도움이 되는 요구 사항을 다룰 것이다. 이러한 단원을 '실전' 단원이라고 부르겠다.

- 스프링 시큐리티에서 사용자를 기술하는 UserDetails
- 사용자가 실행할 수 있는 작업을 정의하는 GrantedAuthority
- UserDetailsService 계약을 확장하는 UserDetailsManager. 상속된 동작 외에 사용자 만들기, 사용자의 암호 수정이나 삭제 등의 작업도 지원한다.

앞서 2장에서 인증 프로세스에서의 UserDetailsService 및 PasswordEncoder의 역할을 간단하게 알아봤는데 스프링 부트가 구성한 기본 인스턴스를 이용하지 않고 개발자가 정의한 인스턴스를 연결하는 방법만 알아봤다. 이 단원에서는 다음과 같이 더 수준 높은 내용을 다룬다.

- 스프링 시큐리티에 있는 구현과 이를 이용하는 방법
- 계약을 위한 맞춤형 구현을 정의하는 방법과 시기
- 실제 애플리케이션에서 볼 수 있는 인터페이스를 구현하는 방법
- 이러한 인터페이스 이용의 모범 사례

먼저 스프링 시큐리티가 사용자 정의를 이해하는 방법부터 시작해보자. 이를 위해 UserDetails 및 GrantedAuthority 계약을 살펴보고 UserDetailsService를 자세히 알아본 후 UserDetailsManager가 이 계약을 확장하는 방법을 알아본다. 그리고 이러한 인터페이스의 구현(예: InMemoryUserDetailsManager, JdbcUserDetailsManager, LdapUserDetailsManager)을 적용하고 이러한 구현이 시스템에 적합하지 않을 때는 맞춤형 구현을 작성한다.

3.1 스프링 시큐리티의 인증 구현

앞의 단원에서 스프링 시큐리티를 시작했는데, 첫 번째 예제에서 스프링 부트가 새 애플리케이션이 처음부터 작동하도록 몇 가지 기본값을 정의한다는 것을 알아봤다. 또한 앱에서 흔히 볼 수 있는 다양한 대안으로 이러한 기본값을 재정의하는 방법도 배웠다. 하지만 실제로는 작업의 개념을 간략하게 알아봤을 뿐이다. 이번 단원과 4~5장에서 이러한 인터페이스와 다양한 구현을 더 자세하게 논의하고 실제 애플리케이션에서 어떤 경우에 이러한 내용을 적용할 수 있는지 알아본다.

그림 3.1에 스프링 시큐리티의 인증 흐름이 나온다. 이 아키텍처는 스프링 시큐리티가 구현하는 인증 프로세스의 근간이며 모든 스프링 시큐리티 구현이 이에 의존하므로 정확하게 이해하는 것이 중요하

한 사용자 추가, 수정, 삭제 작업을 추가한다. 두 계약 간의 분리는 인터페이스 분리 원칙의 훌륭한 예다. 인터페이스를 분리하면 앱에 필요 없는 동작을 구현하도록 프레임워크에서 강제하지 않기 때문에 유연성이 향상된다. 사용자를 인증하는 기능만 필요한 경우 UserDetailsService 계약만 구현하면 필요한 기능을 제공할 수 있다. 사용자를 관리하려면 UserDetailsService 및 UserDetailsManager 구성 요소에 사용자를 나타내는 방법이 필요하다.

개발자는 스프링 시큐리티에 있는 UserDetails 계약을 구현해서 프레임워크가 이해할 수 있게 사용자를 기술해야 한다. 이 단원에서 배우겠지만, 스프링 시큐리티에서 사용자는 사용자가 수행할 수 있는 작업을 나타내는 이용 권리의 집합을 가진다. 이용 권리에 관해서는 7장과 8장에서 권한 부여를 논의할 때 자세히 다룬다. 일단 여기에서는 사용자가 수행할 수 있는 작업을 GrantedAuthority 인터페이스로 나타낸다는 것만 알아두자. 이를 종종 권한이라고 하며 사용자는 하나 이상의 권한을 가진다. 그림 3.2에는 인증 흐름의 사용자 관리 부분에 해당하는 구성 요소 간의 관계가 나온다.

스프링 시큐리티 아키텍처에서 이러한 객체 간의 연결과 구현하는 방법을 이해하면 애플리케이션을 개발할 때 광범위한 옵션을 선택할 수 있다. 이러한 옵션은 개발하려는 앱에 딱 맞는 퍼즐의 조각일 수 있으므로 상황에 맞게 현명하게 선택할 수 있어야 한다. 물론 선택하기 위해서는 먼저 어떤 옵션이 있는지를 알아야 한다.

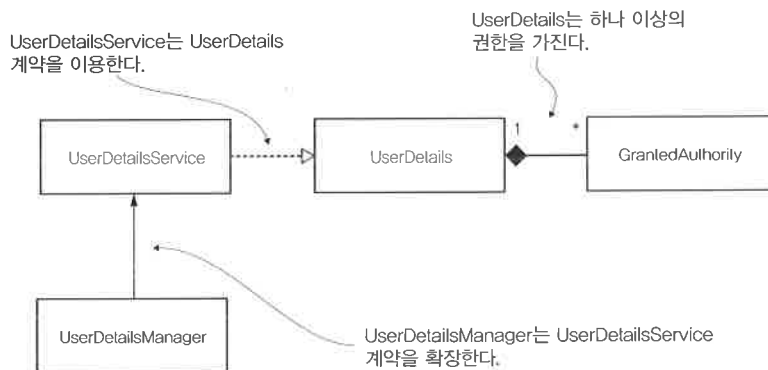


그림 3.2 사용자 관리를 수행하는 구성 요소 간의 종속성. UserDetailsService는 사용자 이름으로 찾은 사용자 세부 정보를 반환한다. UserDetails 계약은 사용자를 기술한다. 사용자는 GrantedAuthority 인터페이스로 나타내는 권한을 하나 이상 가진다. UserDetailsManager 계약은 UserDetailsService를 확장해서 암호 생성, 삭제, 변경 등의 작업을 추가한다.

일반적으로 앱은 사용자가 애플리케이션에서 의미 있는 작업을 수행하도록 허용해야 한다. 예를 들어 사용자는 데이터를 읽고, 쓰고, 삭제할 수 있어야 한다. 사용자에게 작업을 수행할 이용 권리가 있거나 없다고 말하며 사용자가 가진 이용 권리를 나타내는 것이 권한이다. `getAuthorities()` 메서드는 사용자에게 부여된 권한의 그룹을 반환하도록 구현한다.

참고 7장에서 배우겠지만 스프링 시큐리티에서는 세분화된 이용 권리, 또는 이용 권리의 그룹인 역할을 권한이라고 한다. 이 책에서는 이해를 돕기 위해 세분된 이용 권리를 권한이라고 한다.

`UserDetails` 계약을 보면 사용자는 다음과 같은 작업을 할 수 있다.

- 계정 만료
- 계정 잠금
- 자격 증명 만료
- 계정 비활성화

애플리케이션의 논리에서 이러한 사용자 제한을 구현하려면 `isAccountNonExpired()`, `isAccountNonLocked()`, `isCredentialsNonExpired()`, `isEnabled()` 메서드를 재정의해서 `true`를 반환하게 해야 한다. 모든 애플리케이션에서 특정 조건에 계정이 만료되거나 잠기는 것은 아니다. 애플리케이션에서 이러한 기능을 구현할 필요가 없다면 단순히 네 메서드가 `true`를 반환하게 하면 된다.

참고 `UserDetails` 인터페이스의 마지막 네 메서드의 이름은 다소 이상해 보일 수 있으며 깔끔한 코딩이나 유지 관리 측면에서 잘못됐다고 생각하는 사람도 있을 것이다. 예를 들어 `isAccountNonExpired()`는 이중 부정에 가까워서 언뜻 혼동을 일으킬 수 있다. 하지만 메서드 이름이 이렇게 정해진 데는 이유가 있다. 이들 메서드 이름은 권한 부여가 실패해야 하면 `false`를 반환하고, 그렇지 않으면 `true`를 반환하도록 지정됐다. 보통은 'false'를 부정적으로, 'true'를 긍정적으로 인식하므로 이것이 올바른 접근법이다.

```
GrantedAuthority g1 = () -> "READ";
GrantedAuthority g2 = new SimpleGrantedAuthority("READ");
```



참고 람다 식으로 구현하기 전에 `@FunctionalInterface` 어노테이션을 지정해 인터페이스가 함수형임을 지정하는 것이 좋다. 그 이유는 인터페이스가 함수형으로 표시되지 않으면 인터페이스의 개발자가 향후 버전에 더 많은 추상 메서드를 추가할 권리가 있다는 의미일 수 있기 때문이다. 스프링 시큐리티에서 `GrantedAuthority` 인터페이스는 함수형으로 표시되지 않았다. 개인적으로 실제 프로젝트에는 권장하지 않지만, 여기에서는 인터페이스를 구현할 때 코드를 짧고 이해하기 쉽게 하려고 람다 식을 이용했다.

3.2.3 최소한의 UserDetails 구현 작성

이 절에서는 `UserDetails` 계약의 첫 번째 구현을 작성한다. 우선 각 메서드가 정적 값을 반환하는 기본 구현으로 시작해보자. 그런 다음 실제 시나리오에서 더 쉽게 찾아볼 수 있는 버전으로 변경하고 여러 사용자 인스턴스를 이용할 수 있는 버전으로 변경해보자. 이제 `UserDetails` 및 `GrantedAuthority` 인터페이스를 구현하는 방법을 배웠으므로 애플리케이션 사용자의 가장 단순한 정의를 작성할 수 있다.

예제 3.2에 나온 것처럼 `DummyUser`라는 클래스로 최소한의 사용자 기술을 구현해보자. 이 클래스로 `UserDetails` 계약의 메서드를 구현하는 방법을 볼 수 있다. 이 클래스의 인스턴스는 사용자 이름이 "bill"이고 암호는 "12345"이며 "READ" 권한이 있는 한 사용자를 나타낸다.

예제 3.2 DummyUser 클래스

```
public class DummyUser implements UserDetails {

    @Override
    public String getUsername() {
        return "bill";
    }

    @Override
    public String getPassword() {
        return "12345";
    }
}
```



```

    }

    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @Override
    public boolean isEnabled() {
        return true;
    }

    // 생략된 코드
}

```

이 최소화된 구현에서 클래스의 모든 인스턴스는 같은 사용자를 나타내는데, 이는 계약을 이해하기 위한 용도이고 실제 애플리케이션과는 거리가 멀다. 실제 애플리케이션에서는 다른 사용자를 나타내는 인스턴스를 생성할 수 있도록 클래스를 작성해야 한다. 이 경우 다음 코드 예제와 같이 클래스가 사용자 이름과 암호를 특성으로 포함하도록 정의해야 한다.

예제 3.5 더 현실적인 UserDetails 인터페이스의 구현

```

public class SimpleUser implements UserDetails {

    private final String username;
    private final String password;

    public SimpleUser(String username, String password) {
        this.username = username;
        this.password = password;
    }

    @Override

```

방법은 UserDetails의 다른 인스턴스에서 시작하는 것이다. 예제 3.7의 첫 번째 행은 사용자 이름을 문자열로 지정하는 것으로 시작해서 UserBuilder를 만든다. 뒤에서는 이미 존재하는 UserDetails의 인스턴스로 빌더를 만드는 방법을 알아본다.

예제 3.7 User.UserBuilder 인스턴스 만들기

```
User.UserBuilder builder1 = User.withUsername("bill");  ← 주어진 사용자 이름으로 사용자 생성

UserDetails u1 = builder1
    .password("12345")
    .authorities("read", "write")
    .passwordEncoder(p -> encode(p))  ← 암호 인코더는 인코딩을 수행하는 함수에 불과하다.
    .accountExpired(false)
    .disabled(true)
    .build();  ← 빌더 파이프라인의 끝에서 build() 메서드를 호출한다.

User.UserBuilder builder2 = User.withUserDetails(u);  ← 기존의 UserDetails 인스턴스에서 사용자를
                                                       만들 수도 있다.

UserDetails u2 = builder2.build();
```

예제 3.7에 나온 모든 빌더를 이용해 UserDetails 계약으로 표현되는 사용자를 얻을 수 있다. 빌더 파이프라인의 끝에서 build() 메서드를 호출하며, 별도로 암호 인코딩 함수를 지정한 경우 이를 적용해서 암호를 인코딩하고 UserDetails의 인스턴스를 구성한 후 반환한다.

참고 암호 인코더는 2장에 나온 빈과는 다르다. 이름이 약간 혼동될 수 있지만, 여기에는 Function<String, String>만 있다. 이 함수는 암호를 지정한 인코딩으로 변환하는 일만 한다. 다음 절에서는 2장에서 이용했던 스프링 시큐리티의 PasswordEncoder 계약을 더 자세하게 다룬다.

3.2.5 사용자와 연관된 여러 책임 결합

앞의 절에서 UserDetails 인터페이스를 구현하는 방법을 배웠는데 실제 시나리오는 더 복잡할 때가 많고 한 사용자가 여러 책임을 갖는 것이 일반적이다. 그리고 사용자를 데이터베이스에 저장하면 애플리케이션에 지속성 엔티티를 나타내는 클래스가 필요하다. 또는 다른 시스템에서 웹 서비스를 통해 사용자를 가져오면 사용자 인스턴스를 나타내는 데이터 전송 객체가 필요하다. 간단하고 일반적인 첫 번째

```

        return this.password;
    }

    public String getAuthority() {
        return this.authority;
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return List.of() -> this.authority;
    }

    // 생략된 코드

}

```

이 클래스에는 JPA 어노테이션, getter, setter가 포함되어 있고, 그중 getUsername() 및 getPassword()는 모두 UserDetails 계약의 메서드를 재정의한다. getAuthority() 메서드는 String을 반환하고 getAuthorities() 메서드는 Collection을 반환하며 getAuthority() 메서드는 클래스의 단순한 getter이고 getAuthorities()는 UserDetails 인터페이스의 메서드를 구현한다. 여기에 다른 엔티티에 대한 관계를 추가하면 더욱더 복잡해진다. 다시 말하지만 이런 코드는 절대 바람직하지 않다!

어떻게 해야 더 깔끔한 코드를 작성할 수 있을까? 이 코드가 복잡한 이유는 두 책임을 혼합했기 때문이다. 애플리케이션에 두 책임이 필요한 것은 맞지만, 모두 한 클래스에 넣을 필요는 없다. User 클래스를 장식하는 SecurityUser라는 별도의 클래스를 정의해서 책임을 분리해보자. 예제 3.10에 나오듯이 SecurityUser 클래스는 UserDetails 계약을 구현하고 이를 이용해 사용자를 스프링 시큐리티 아키텍처에 연결한다. User 클래스에는 JPA 엔티티 책임만 남아 있다.

예제 3.10 JPA 엔티티 책임만 있는 User 클래스 구현

```

@Entity
public class User {

    @Id
    private int id;
    private String username;
    private String password;
}

```

보다시피 `SecurityUser` 클래스는 시스템에서 사용자 세부 정보를 스프링 시큐리티가 이해하는 `UserDetails` 계약에 매핑하는 일만 한다. `SecurityUser`에 `User` 엔티티가 반드시 필요하다는 것을 지정하기 위해 필드를 `final`로 지정했다. 사용자는 생성자를 통해 지정해야 한다. `SecurityUser` 클래스로 `User` 엔티티 클래스를 장식하고 스프링 시큐리티 계약에 필요한 코드를 추가해서 JPA 엔티티에 코드를 섞어 결과적으로 여러 다른 작업을 구현하지 않도록 했다.

참고 두 책임을 분리하는 다른 방법도 있으며 이 절에 나온 방법이 최상이거나 유일한 방법은 아니다. 클래스 디자인을 구현하기 위해 선택하는 방법은 일반적으로 사례별로 크게 다르지만 기본 개념은 동일하다. 즉, 애플리케이션의 유지 관리성을 높이려면 책임을 혼합하지 말고 최대한 분리해서 코드를 작성해야 한다.

3.3 스프링 시큐리티가 사용자를 관리하는 방법 지정

앞의 절에서 `UserDetails` 계약을 구현해 스프링 시큐리티가 이해할 수 있게 사용자를 기술했다. 그렇다면 스프링 시큐리티는 사용자를 어떻게 관리할까? 자격 증명을 비교할 때는 어디에서 가져오고, 새 사용자를 추가하거나 기존 사용자를 변경할 때는 어떻게 해야 할까? 2장에서 프레임워크가 정의하는 `UserDetailsService`라는 특정 구성 요소로 인증 프로세스가 사용자 관리를 위임한다고 배웠다. 또한 `UserDetailsService`를 정의해서 스프링 부트가 제공하는 기본 구현을 재정의했다.

이 절에서는 `UserDetailsService` 클래스를 구현하는 다양한 방법을 실험해본다. `UserDetailsService` 계약에 기술된 책임을 예제로 구현해서 사용자 관리가 작동하는 방식을 배우고 그다음에는 `UserDetailsManager` 인터페이스로 `UserDetailsService`로 정의된 계약에 더 많은 동작을 추가하는 방법을 배운다. 마지막으로 스프링 시큐리티에 있는 `UserDetailsManager` 인터페이스의 구현을 이용하고 스프링 시큐리티에 있는 가장 잘 알려진 구현인 `JdbcUserDetailsManager`를 이용해 예제 프로젝트를 작성한다. 이 절을 완료하면 인증 흐름에 필수적인 사용자를 찾을 위치를 스프링 시큐리티에 알려주는 방법을 이해할 수 있게 된다.

3.3.2 UserDetailsService 계약 구현

이 절에서는 UserDetailsService의 구현을 시연하는 실용적인 예제를 살펴본다. 애플리케이션은 사용자의 자격 증명과 다른 측면의 세부 정보를 관리한다. 이러한 정보는 데이터베이스에 저장하거나 웹 서비스 또는 기타 방법으로 접근하는 다른 시스템에서 관리할 수 있다(그림 3.3). 시스템이 어떻게 작동하는지와 관계없이 스프링 시큐리티에 필요한 것은 사용자 이름으로 사용자를 검색하는 구현을 제공하는 것이다.

다음 예제에서는 사용자의 메모리 내 목록을 이용하는 UserDetailsService를 작성한다. 2장에서 같은 일을 하는 InMemoryUserDetailsManager 구현을 이용했다. 구현이 작동하는 방법에 이미 익숙하므로 비슷한 기능을 선택했지만, 이번에는 우리가 직접 구현했다. UserDetailsService 클래스의 인스턴스를 만들 때 사용자의 목록을 제공한다. ssia-ch3-ex1 프로젝트에서 이 예제를 볼 수 있다. model로 명명된 패키지에서 다음 예제 코드와 같이 UserDetails를 정의한다.

예제 3.12 UserDetails 인터페이스의 구현

```
public class User implements UserDetails {
```

```
    private final String username;
```

```
    private final String password;
```

```
    private final String authority; ← 예제를 간단하게 하기 위해 한 사용자에게 하나의 권한만 적용한다.
```

```
    public User(String username, String password, String authority) {
```

```
        this.username = username;
```

```
        this.password = password;
```

```
        this.authority = authority;
```

```
    }
```

```
@Override
```

```
public Collection<? extends GrantedAuthority> getAuthorities() {
```

```
    return List.of(() -> authority); ← 인스턴스를 만들 때 지정한 이름의 GrantedAuthority
```

```
    }
```

```
@Override
```

```
public String getPassword() {
```

```
    return password;
```

```
}
```

User 클래스는 변경할 수 없다. 인스턴스를 만들 때 세 특성의 값을 지정하며 이러한 값은 나중에 변경할 수 없다.

객체만 포함하는 목록을 반환한다.

```

@Override
public UserDetails loadUserByUsername(String username)
    throws UsernameNotFoundException {

    return users.stream()
        .filter( ← 사용자 목록에서 요청된 사용자 이름과 일치하는 항목을 필터링한다.
            u -> u.getUsername().equals(username)
        )
        .findFirst() ← 일치하는 사용자가 있으면 반환한다.
        .orElseThrow( ← 이 사용자 이름이 존재하지 않으면 예외를 투척한다.
            () -> new UsernameNotFoundException("User not found")
        );
}
}

```

loadUserByUsername(String username) 메서드는 주어진 사용자 이름으로 사용자의 목록을 검색하고 원하는 UserDetails 인스턴스를 반환하고 주어진 사용자 이름이 발견되지 않으면 UsernameNotFoundException 을 투척한다. 이제 이 구현을 UserDetailsService로 이용할 수 있다. 다음 예제 코드에는 이를 구성 클래스에 빈으로 추가하고 여기에 한 사용자를 등록하는 방법이 나온다.

예제 3.14 UserDetailsService를 구성 클래스에 빈으로 등록

```

@Configuration
public class ProjectConfig {

    @Bean
    public UserDetailsService userDetailsService() {
        UserDetails u = new User("john", "12345", "read");
        List<UserDetails> users = List.of(u);
        return new InMemoryUserDetailsService(users);
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return NoOpPasswordEncoder.getInstance();
    }
}

```

```
void changePassword(String oldPassword, String newPassword);
boolean userExists(String username);
}
```

2장에서 이용한 `InMemoryUserDetailsManager` 객체는 사실 `UserDetailsManager`였다. 2장에서는 `UserDetailsService` 특성만 생각했지만, 이제 그 인스턴스에서 `createUser()` 메서드를 호출할 수 있었던 이유를 알 수 있게 됐다.

사용자 관리에 `JdbcUserDetailsManager` 이용

`InMemoryUserDetailsManager` 외에 다른 `UserDetailManager`인 `JdbcUserDetailsManager`도 자주 이용한다. `JdbcUserDetailsManager`는 SQL 데이터베이스에 저장된 사용자를 관리하며 JDBC를 통해 데이터베이스에 직접 연결한다. 이처럼 `JdbcUserDetailsManager`는 데이터베이스 연결과 관련한 다른 프레임워크나 사양으로부터 독립적일 수 있다.

`JdbcUserDetailsManager`가 작동하는 방법을 이해하려면 예제로 직접 이용해보는 것이 최선이다. 다음 예제에서는 `JdbcUserDetailsManager`를 이용해 MySQL 데이터베이스에 저장된 사용자를 관리하는 애플리케이션을 구현한다. 그림 3.4는 `JdbcUserDetailsManager` 구현이 인증 흐름에서 차지하는 위치를 개략적으로 보여준다.

데이터베이스 한 개와 테이블 두 개를 만들고 `JdbcUserDetailsManager` 이용 방법을 보여주는 데모 애플리케이션을 작성해보자. 이 예제에서 데이터베이스 이름은 `spring`으로 지정하고 테이블 이름은 각각 `users` 및 `authorities`로 지정한다. 이러한 이름은 `JdbcUserDetailsManager`가 인식하는 기본 테이블 이름이다. 이 절의 마지막 부분에서 배우겠지만, `JdbcUserDetailsManager` 구현은 아주 유연하므로 원하면 이러한 기본 이름을 바꿀 수 있다. `users` 테이블의 목적은 사용자 레코드를 저장하는 것이다. `JdbcUserDetailsManager` 구현은 `users` 테이블에 사용자 이름, 암호, 그리고 사용자 활성화 여부를 저장하는 세 열이 있다고 가정한다.

DBMS(Database Management System)의 명령줄 툴이나 클라이언트 애플리케이션으로 데이터베이스와 그 구조를 직접 만들 수 있다. 예를 들어 MySQL의 경우 MySQL 워크벤치를 이용할 수 있다. 하지만 가장 쉬운 방법은 스프링 부트가 대신 스크립트를 실행하도록 하는 것이다. 이를 위해서는 프로젝트의 `resources` 폴더에 `schema.sql` 및 `data.sql`의 두 파일만 추가하면 된다. `schema.sql` 파일에는 테이블 만들기, 수정, 삭제 같은 데이터베이스의 구조에 관한 쿼리를 추가한다. `data.sql` 파일에는 `INSERT`, `UPDATE`, `DELETE` 같은 테이블 안의 데이터를 처리하는 쿼리를 추가한다. 스프링 부트는 애플리케이션

```
`password` VARCHAR(45) NOT NULL,  
`enabled` INT NOT NULL,  
PRIMARY KEY (`id`));
```

authorities 테이블은 사용자별 권한을 저장한다. 각 레코드는 사용자 이름과 해당 사용자에게 허가된 권한을 저장한다.

예제 3.17 authorities 테이블을 생성하는 SQL 쿼리

```
CREATE TABLE IF NOT EXISTS `spring`.`authorities` (  
  `id` INT NOT NULL AUTO_INCREMENT,  
  `username` VARCHAR(45) NOT NULL,  
  `authority` VARCHAR(45) NOT NULL,  
  PRIMARY KEY (`id`));
```

참고 편의를 위해 이 책의 예제에서는 인덱스나 외래 키 정의를 생략했다.



테스트에 필요한 사용자를 준비하기 위해 각 테이블에 레코드를 삽입한다. 이를 위해 스프링 부트의 resources 폴더에 있는 data.sql 파일에 다음 쿼리를 추가한다.

```
INSERT IGNORE INTO `spring`.`authorities` VALUES (NULL, 'john', 'write');  
INSERT IGNORE INTO `spring`.`users` VALUES (NULL, 'john', '12345', '1');
```

프로젝트를 위해 최소한 다음 코드 예제에 나온 종속성을 추가해야 한다. pom.xml 파일을 열어서 이러한 종속성을 추가했는지 확인한다.

예제 3.18 예제 프로젝트를 개발하는 데 필요한 종속성

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-security</artifactId>  
</dependency>  
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-web</artifactId>  
</dependency>
```



```
public PasswordEncoder passwordEncoder() {
    return NoOpPasswordEncoder.getInstance();
}
}
```

애플리케이션의 엔드포인트에 접근하려면 이제 데이터베이스에 저장된 사용자 중 하나와 HTTP Basic 인증을 이용해야 한다. 다음 예제 코드와 같이 새 엔드포인트를 만들고 cURL을 이용해 호출해보자.

예제 3.20 구현을 확인하기 위한 테스트 엔드포인트

```
@RestController
public class HelloController {

    @GetMapping("/hello")
    public String hello() {
        return "Hello!";
    }
}
```

다음 코드에서 올바른 사용자 이름과 암호로 엔드포인트를 호출하면 결과가 표시된다.

```
curl -u john:12345 http://localhost:8080/hello
```

호출 응답은 다음과 같다.

```
Hello!
```

JdbcUserDetailsManager에 이용되는 쿼리도 구성할 수 있다. 이전 예제에서는 JdbcUserDetailsManager 구현이 예상하는 테이블과 열의 이름을 그대로 이용했지만, 자신의 애플리케이션에 이러한 이름이 최고의 선택이 아닐 수 있다. 다음 예제 코드에 JdbcUserDetailsManager의 쿼리를 재정의하는 방법이 나온다.

예제 3.21 사용자를 찾도록 JdbcUserDetailsManager의 쿼리를 변경

```
@Bean
public UserDetailsService userDetailsService(DataSource dataSource) {
    String usersByUsernameQuery =
        "select username, password, enabled from users where username = ?";
    String authsByUserQuery =
```

```
dn: uid=john,ou=groups,dc=springframework,dc=org ← 사용자 정의
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: John
sn: John
uid: john
userPassword: 12345
```

LDIF 파일 끝에 앱의 동작을 테스트하기 위해 한 사용자만 추가했다. LDIF 파일은 resources 폴더에 곧바로 추가할 수 있다. 그러면 자동으로 클래스 경로에 추가되므로 나중에 손쉽게 참조할 수 있다. LDIF 파일의 이름은 server.ldif로 지정했다. LDAP를 이용하고 스프링 부트가 내장형 LDAP 서버를 시작하도록 허용하려면 다음 코드와 같이 종속성을 pom.xml에 추가해야 한다.

```
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-ldap</artifactId>
</dependency>
<dependency>
  <groupId>com.unboundid</groupId>
  <artifactId>unboundid-ldapsdk</artifactId>
</dependency>
```

application.properties 파일에도 다음 코드에 나온 것처럼 임베디드 LDAP 서버에 대한 구성을 추가해야 한다. 앱이 임베디드 LDAP 서버를 부팅하려면 LDIF 파일의 위치, LDAP 서버의 포트, 기본 도메인 구성 요소(DN) 레이블 값이 필요하다.

```
spring.ldap.embedded.ldif=classpath:server.ldif
spring.ldap.embedded.base-dn=dc=springframework,dc=org
spring.ldap.embedded.port=33389
```

인증을 위한 LDAP 서버가 준비되면 이를 이용하도록 애플리케이션을 구성할 수 있다. 다음 코드 예제에서 앱이 LDAP 서버를 통해 사용자를 인증하도록 LdapUserDetailsManager를 구성하는 방법을 볼 수 있다.

이제 앱을 시작하고 /hello 엔드포인트를 호출한다. 사용자 존으로 인증해 앱이 엔드포인트를 호출할 수 있게 해야 한다. 다음 코드는 cURL로 엔드포인트를 호출한 결과를 보여준다.

```
curl -u john:12345 http://localhost:8080/hello
```

호출 응답은 다음과 같다.

```
Hello!
```

요약

- UserDetails 인터페이스는 스프링 시큐리티에서 사용자를 기술하는 데 이용되는 계약이다.
- UserDetailsService 인터페이스는 애플리케이션이 사용자 세부 정보를 얻는 방법을 설명하기 위해 스프링 시큐리티의 인증 아키텍처에서 구현해야 하는 계약이다.
- UserDetailsManager 인터페이스는 UserDetailsService를 확장하고 사용자 생성, 변경, 삭제와 관련된 동작을 추가한다.
- 스프링 시큐리티는 UserDetailsManager 계약의 여러 구현을 제공한다. 이러한 구현에는 InMemoryUserDetailsManager, JdbcUserDetailsManager, LdapUserDetailsManager가 있다.
- JdbcUserDetailsManager는 JDBC를 직접 이용하므로 애플리케이션이 다른 프레임워크에 고정되지 않는다는 이점이 있다.

한 심도 있는 내용과 구현 방법을 알아본다. 그림 4.1을 보면서 PasswordEncoder가 인증 프로세스의 어떤 부분을 담당하는지 기억을 되살려보자.

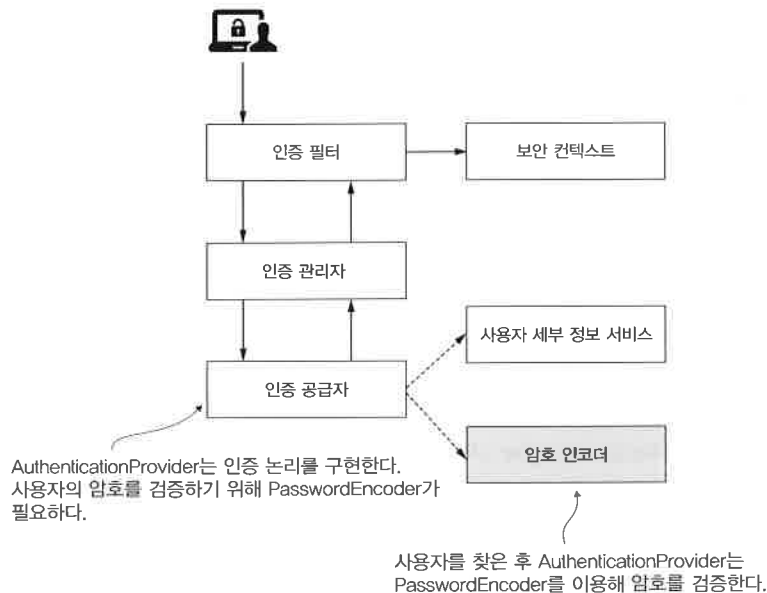


그림 4.1 스프링 시큐리티의 인증 프로세스. AuthenticationProvider는 인증 프로세스에서 PasswordEncoder를 이용해 사용자의 암호를 검증한다.

일반적으로 시스템은 암호를 일반 텍스트로 관리하지 않고 공격자가 암호를 읽고 훔치기 어렵게 하기 위한 일종의 변환 과정을 거친다. 스프링 시큐리티에는 이 책임을 위해 정의된 별도의 계약이 있다. 이를 쉽게 설명하기 위해 PasswordEncoder 구현에 관한 아주 많은 코드 예제를 준비했다. 우선 계약을 이해하는 것부터 시작하고 프로젝트를 진행하며 직접 구현을 작성해보자. 그리고 4.1.3절에서 스프링 시큐리티에 있는 PasswordEncoder의 구현 중 가장 잘 알려지고 많이 이용되는 구현을 알아본다.

4.1.1 PasswordEncoder 계약의 정의

이 절에서는 PasswordEncoder 계약의 정의를 알아본다. 이 계약을 구현해 스프링 시큐리티에 사용자 암호를 검증하는 방법을 알려줄 수 있다. PasswordEncoder는 인증 프로세스에서 암호가 유효한지를 확인한다. 모든 시스템은 어떤 방식으로든 인코딩된 암호를 저장하며 아무도 암호를 읽을 수 없게 해시를 저장하는 것이 좋다. PasswordEncoder도 암호를 인코딩할 수 있다. 계약에 선언된 encode() 및 matches() 메서드는 사실상 계약의 책임을 정의한다. 이 둘은 서로 강력하게 연결되어 있으며 같은 계약의 일부

를 관리하는 방법을 선택할 수 있다. 가장 직관적인 암호 인코더 구현은 암호를 인코딩하지 않고 일반 텍스트로 간주하는 것이다.

2장의 첫 번째 예제에 나온 `NoOpPasswordEncoder`의 인스턴스가 이렇게 암호를 일반 텍스트로 취급했다. 이 클래스를 직접 작성하면 다음 코드 예제와 비슷할 것이다.

예제 4.1 PasswordEncoder의 가장 단순한 구현

```
public class PlainTextPasswordEncoder
    implements PasswordEncoder {

    @Override
    public String encode(CharSequence rawPassword) {
        return rawPassword.toString();  ← 암호를 변경하지 않고 그대로 반환한다.
    }

    @Override
    public boolean matches(
        CharSequence rawPassword, String encodedPassword) {
        return rawPassword.equals(encodedPassword);  ← 두 문자열이 같은지 확인한다.
    }
}
```

인코딩 결과는 원래 암호와 같으므로 일치하는지 확인하려면 `equals()`로 문자열을 비교하면 된다. 다음 코드 예제에 해싱 알고리즘 SHA-512를 이용하는 `PasswordEncoder`의 간단한 구현이 나온다.

예제 4.2 SHA-512를 이용하는 PasswordEncoder 구현

```
public class Sha512PasswordEncoder
    implements PasswordEncoder {

    @Override
    public String encode(CharSequence rawPassword) {
        return hashWithSHA512(rawPassword.toString());
    }

    @Override
    public boolean matches(
```

- `NoOpPasswordEncoder` - 암호를 인코딩하지 않고 일반 텍스트로 유지한다. 이 구현은 예제 용도로만 적당하며 암호를 해시하지 않기 때문에 실제 시나리오에는 절대 쓰지 말아야 한다.
- `StandardPasswordEncoder` - SHA-256을 이용해 암호를 해시한다. 이 구현은 이제는 구식이며 새 구현에는 쓰지 말아야 한다. 이 구현이 구식인 이유는 강도가 약한 해싱 알고리즘을 쓰기 때문이지만 가끔 기존 애플리케이션에는 이 구현이 이용되는 경우가 있다.
- `Pbkdf2PasswordEncoder` - PBKDF2를 이용한다.
- `BCryptPasswordEncoder` - `bcrypt` 강력 해싱 함수로 암호를 인코딩한다.
- `SCryptPasswordEncoder` - `scrypt` 해싱 함수로 암호를 인코딩한다.

해싱과 이러한 알고리즘에 관한 자세한 내용은 《Real-World Cryptography》(Manning, 2020)의 2장에서 볼 수 있다. 다음에 원서를 볼 수 있는 주소가 있다.

- <https://livebook.manning.com/book/real-world-cryptography/chapter-2/>

이러한 `PasswordEncoder` 구현의 인스턴스를 만드는 방법을 몇 가지 예제로 확인해보자. `NoOpPasswordEncoder`는 암호를 인코딩하지 않으며 예제 4.1의 `PlainTextPasswordEncoder`와 비슷한 구현을 가지고 있다. 따라서 이 암호 인코더는 이론적인 예제에만 이용한다. 또한 `NoOpPasswordEncoder` 클래스는 싱글톤으로 설계돼서 클래스 바깥에서는 생성자를 직접 호출할 수 없지만, 다음과 같이 `NoOpPasswordEncoder.getInstance()` 메서드로 클래스의 인스턴스를 얻을 수 있다.

```
PasswordEncoder p = NoOpPasswordEncoder.getInstance();
```

스프링 시큐리티에 있는 `StandardPasswordEncoder` 구현은 SHA-256으로 암호를 해싱한다. `StandardPasswordEncoder`를 이용하면 해싱 프로세스에 적용할 비밀을 지정할 수 있다. 비밀의 값은 생성자의 매개 변수로 전달하며 인수가 없는 생성자를 호출하면 빈 문자열이 키 값으로 이용된다. 그러나 `StandardPasswordEncoder`는 이제 구식이므로 새 구현에는 쓰지 않는 것이 좋다. 오래된 애플리케이션이나 레거시 코드에서 종종 이 구현을 볼 수 있다. 다음 코드 예제에 이 암호 인코더의 인스턴스를 만드는 방법이 나온다.

```
PasswordEncoder p = new StandardPasswordEncoder();
PasswordEncoder p = new StandardPasswordEncoder("secret");
```

지정하는 로그 라운드 값은 해싱 작업이 이용하는 반복 횟수에 영향을 준다. 반복 횟수는 2로그 라운드로 계산된다. 반복 횟수를 계산하기 위한 로그 라운드 값은 4~31 사이여야 한다. 이 값을 지정하려면 이전 코드에 나온 것처럼 두 번째나 세 번째 오버로드된 생성자를 호출하면 된다.

마지막으로 살펴볼 옵션은 `SCryptPasswordEncoder`다(그림 4.2). 이 암호 인코더는 `scrypt` 해싱 함수를 이용한다. `SCryptPasswordEncoder`의 인스턴스를 만드는 데는 두 가지 옵션이 있다.

```
PasswordEncoder p = new SCryptPasswordEncoder();
PasswordEncoder p = new SCryptPasswordEncoder(16384, 8, 1, 32, 64);
```

위 예제의 값은 인수가 없는 생성자를 호출해 인스턴스를 만들 때 이용되는 값이다.

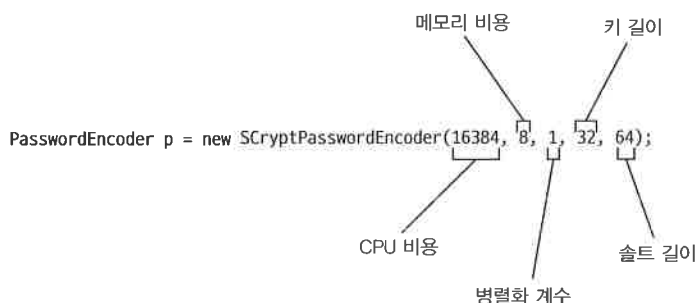


그림 4.2 `SCryptPasswordEncoder` 생성자는 다섯 개의 매개 변수를 받아 CPU 비용, 메모리 비용, 키 길이, 솔트 길이를 구성할 수 있게 해준다.

4.1.4 DelegatingPasswordEncoder를 이용한 여러 인코딩 전략

이 절에서는 인증 흐름에 암호 일치를 위해 다양한 구현을 적용해야 할 때를 설명한다. 또한 애플리케이션에서 `PasswordEncoder`로 작동하는 유용한 툴을 적용하는 방법도 배운다. 이 툴은 자체 구현이 없고 `PasswordEncoder` 인터페이스를 구현하는 다른 객체에 위임한다.

일부 애플리케이션에서는 다양한 암호 인코더를 갖추고 특정 구성에 따라 선택하는 방식이 유용할 수 있다. 운영 단계 애플리케이션에 `DelegatingPasswordEncoder`가 사용되는 일반적인 시나리오로는 특정 애플리케이션 버전부터 인코딩 알고리즘이 변경된 경우가 있다. 현재 사용되는 알고리즘에서 취약성이 발견되어 신규 등록 사용자의 자격 증명을 변경하고 싶지만, 기존 자격 증명을 변경하기가 쉽지 않다고 생각해보자. 이때는 여러 종류의 해시를 지원해야 하는데, 이 상황을 어떻게 해결해야 할까? `DelegatingPasswordEncoder` 객체는 이 상황을 해결하는 유일한 방법은 아니지만 좋은 선택이다.

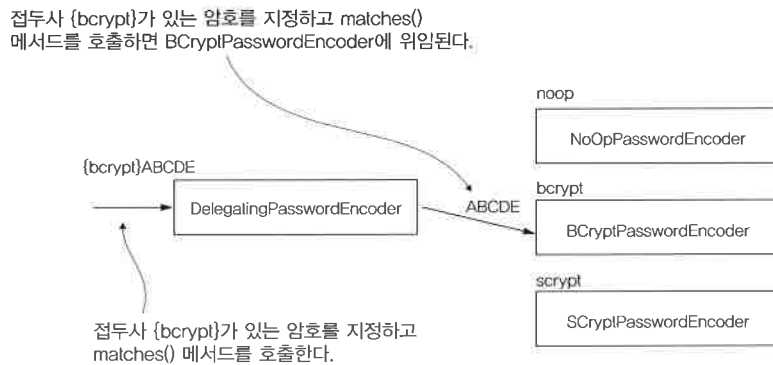


그림 4.4 여기에서 DelegatingPasswordEncoder는 접두사 {noop}에 대해 NoOpPasswordEncoder, 접두사 {bcrypt}에 대해 BCryptPasswordEncoder, 접두사 {scrypt}에 대해 SCryptPasswordEncoder를 등록한다. 암호에 접두사 {bcrypt}가 있으면 DelegatingPasswordEncoder는 BCryptPasswordEncoder 구현에 작업을 위임한다.

다음으로, DelegatingPasswordEncoder를 정의하는 방법을 알아보자. 다음 예제에 나오는 것처럼 먼저 원하는 PasswordEncoder 구현의 인스턴스 컬렉션을 만들고, 이를 DelegatingPasswordEncoder에 넣는다.

예제 4.4 DelegatingPasswordEncoder의 인스턴스 만들기

```

@Configuration
public class ProjectConfig {

    // 생략된 코드

    @Bean
    public PasswordEncoder passwordEncoder() {
        Map<String, PasswordEncoder> encoders = new HashMap<>();

        encoders.put("noop", NoOpPasswordEncoder.getInstance());
        encoders.put("bcrypt", new BCryptPasswordEncoder());
        encoders.put("scrypt", new SCryptPasswordEncoder());

        return new DelegatingPasswordEncoder("bcrypt", encoders);
    }
}
  
```

DelegatingPasswordEncoder는 구현의 컬렉션 중에서 선택할 때 쓸 수 있는 PasswordEncoder처럼 작동한다. 예제 4.4에서 DelegatingPasswordEncoder의 선언된 인스턴스는 NoOpPasswordEncoder, BCryptPasswordEncoder, SCryptPasswordEncoder에 대한 참조를 가지며 기본적으로 BCryptPasswordEncoder

$(x, k) \rightarrow y$

여기에서 x 는 입력이고 k 는 키며 y 는 암호화 결과다. 이 방식으로 알려진 함수에 키를 이용해 출력에서 입력을 얻을 수 있다 $(y, k) \rightarrow x$. 이를 **역함수 복호화**(Reverse Function Decryption)라고 한다. 암호화에 쓰는 키와 복호화에 쓰는 키가 같으면 일반적으로 이를 대칭 키라고 한다.

암호화 $((x, k1) \rightarrow y)$ 및 복호화 $((y, k2) \rightarrow x)$ 에 다른 키를 쓰면 **비대칭 키**(Asymmetric Key)로 암호화가 수행된다고 말한다. 그리고 $(k1, k2)$ 를 **키 쌍**(Key Pair)이라고 한다. 암호화에 이용되는 키 중 $k1$ 을 **공개 키**(Public Key)라고 하고 $k2$ 를 **개인 키**(Private Key)라고 한다. 이와 같이 개인 키의 소유자는 데이터를 복호화할 수 있다.

해싱(Hashing)은 함수가 한 방향으로만 작동하는 특정한 유형의 인코딩이다. 따라서 해싱 함수의 출력 y 에서 입력 x 를 얻을 수 없다. 그러나 출력 y 가 입력 x 에 해당하는지 확인할 수 있는 방법이 반드시 있어야 하므로 해시는 인코딩과 일치를 위한 한 쌍의 함수로 볼 수 있다. 해싱 함수가 $x \rightarrow y$ 라면 일치 함수 $(x, y) \rightarrow \text{boolean}$ 도 있다.

때때로 해싱 함수는 입력에 임의의 값을 추가할 수도 있다. $(x, k) \rightarrow y$. 이 값을 **솔트**(Salt)라고 한다. 솔트는 함수를 더 강하게 만들어 결과에서 입력을 얻는 역함수의 적용 난도를 높인다.

지금까지 이 책에서 소개하고 이용한 계약을 요약하기 위해 표 4.1에 각 구성 요소를 간단하게 정리했다.

표 4.1 스프링 시큐리티에서 인증 흐름을 위한 주 계약을 나타내는 인터페이스

설명	설명
UserDetails	스프링 시큐리티가 관리하는 사용자를 나타낸다.
GrantedAuthority	애플리케이션의 목적 내에서 사용자에게 허용되는 작업을 정의한다(예: 읽기, 쓰기, 삭제 등).
UserDetailsService	사용자 이름으로 사용자 세부 정보를 검색하는 객체를 나타낸다.
UserDetailsManager	UserDetailsService의 더 구체적인 계약이다. 사용자 이름으로 사용자를 검색하는 것 외에도 사용자 컬렉션이나 특정 사용자를 변경할 수도 있다.
PasswordEncoder	암호를 암호화 또는 해시하는 방법과 주어진 인코딩된 문자열을 일반 텍스트 암호와 비교하는 방법을 지정한다.

4.2 스프링 시큐리티 암호화 모듈에 관한 추가 정보

이 절에서는 스프링 시큐리티에서 암호화를 담당하는 스프링 시큐리티 암호화 모듈(SSCM)을 살펴봤다. 암호화 및 복호화 함수와 키 생성 기능은 자바 언어에서 기본 제공되지 않기 때문에 개발자가 이러한 기능에 보다 쉽게 접근하기 위한 종속성을 추가할 때 제약이 있다.

```
public interface BytesKeyGenerator {

    int getKeyLength();
    byte[] generateKey();

}
```

이 인터페이스에는 byte[] 키를 반환하는 generateKey() 메서드 외에도 키 길이(바이트 수)를 반환하는 메서드가 있다. 기본 BytesKeyGenerator는 8바이트 길이의 키를 생성한다.

```
BytesKeyGenerator keyGenerator = KeyGenerators.secureRandom();
byte [] key = keyGenerator.generateKey();
int keyLength = keyGenerator.getKeyLength();
```

위 코드 예제의 키 생성기는 8바이트 길이의 키를 생성한다. 다른 키 길이를 지정하려면 키 생성기 인스턴스를 얻을 때 KeyGenerators.secureRandom() 메서드에 원하는 값을 전달하면 된다.

```
BytesKeyGenerator keyGenerator = KeyGenerators.secureRandom(16);
```

KeyGenerators.secureRandom() 메서드로 생성한 BytesKeyGenerator는 generateKey() 메서드가 호출될 때마다 고유한 키를 생성한다.

같은 키 생성기를 호출하면 같은 키 값을 반환하는 구현이 적합할 때가 있다. 이때는 KeyGenerators.shared(int length) 메서드로 BytesKeyGenerator를 생성할 수 있다. 다음 코드에 key1 및 key2는 값이 같다.

```
BytesKeyGenerator keyGenerator = KeyGenerators.shared(16);
byte [] key1 = keyGenerator.generateKey();
byte [] key2 = keyGenerator.generateKey();
```

4.2.2 암호화와 복호화 작업에 암호기 이용

이 절에서는 스프링 시큐리티에 있는 암호기의 구현을 코드 예제에 적용해본다. 암호기는 암호화 알고리즘을 구현하는 객체다. 암호화와 복호화는 보안을 위한 공통적인 기능이므로 애플리케이션에 이러한 기능이 필요할 가능성이 크다.

차이는 작고 보이지 않는 곳에 있으며, 256바이트 AES 암호화의 작업 모드로 GCM(갈루아/카운터 모드)을 이용한다. 표준 모드는 더 약한 방식인 CBC(암호 블록 체인)를 이용한다.

TextEncryptors는 세 가지 주요 형식이 있으며 Encryptors.text(), Encryptors.delux(), Encryptors.queryableText() 메서드를 호출해 이러한 형식을 생성할 수 있다. 암호기를 생성하는 이러한 메서드 외에도 값을 암호화하지 않는 더미 TextEncryptor를 반환하는 메서드도 있다. 암호화에 시간을 소비하지 않고 애플리케이션의 성능을 테스트하기를 원할 때나 데모 예제에는 더미 TextEncryptor를 이용할 수 있다. Encryptors.noOpText() 메서드가 이 더미 암호기를 반환한다. 다음 코드에 TextEncryptor 이용 예가 나온다. 암호기에 대한 호출이기는 하지만, 이 예제에서 encrypted와 valueToEncrypt는 같다.

```
String valueToEncrypt = "HELLO";
TextEncryptor e = Encryptors.noOpText();
String encrypted = e.encrypt(valueToEncrypt);
```

Encryptors.text() 암호기는 Encryptors.standard() 메서드로 암호화 작업을 관리하고 Encryptors.delux() 메서드는 다음과 같이 Encryptors.stronger() 인스턴스를 이용한다.

```
String salt = KeyGenerators.string().generateKey();
String password = "secret";
String valueToEncrypt = "HELLO";

TextEncryptor e = Encryptors.text(password, salt); ← 솔트와 암호를 이용하는 TextEncryptor 객체 생성
String encrypted = e.encrypt(valueToEncrypt);
String decrypted = e.decrypt(encrypted);
```

Encryptors.text() 및 Encryptors.delux()의 경우 같은 입력으로 encrypt() 메서드를 반복 호출해도 다른 출력이 반환된다. 그 이유는 암호화 프로세스에 임의의 초기화 벡터가 생성되기 때문이다. 실제 상황에서는 이러한 작동 방식을 원하지 않을 수 있다(예: OAuth API 키). OAuth 2에 관해서는 12장~15장에서 다룬다. 이러한 유형의 입력을 쿼리 가능 텍스트라고 하며 이 상황에서는 Encryptors.queryableText() 인스턴스를 이용한다. 이 암호기는 순차 암호화 작업에서 입력이 같으면 같은 출력을 생성하도록 보장한다. 다음 예제에서 encrypted1 변수의 값은 encrypted2 변수의 값과 같다.

```
String salt = KeyGenerators.string().generateKey();
String password = "secret";
String valueToEncrypt = "HELLO";
```