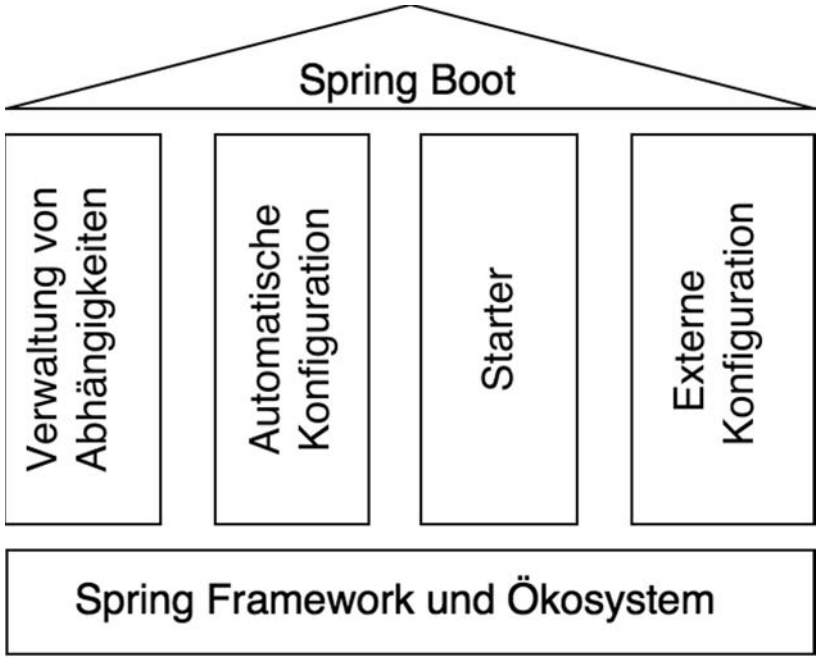
Teil II

Spring Boot

Dieser Teil stellt die Säulen von Spring Boot vor. Dazu gehören insbesondere die Verwaltung von Abhängigkeiten (*Dependency Management*), die automatische Konfiguration und die Starter, die beides zusammenfassen. Die einfache, codefreie externe Konfiguration ist eine weitere wichtige Säule. Grundlage des Systems ist das Spring-Framework selber.

Abb. 1–1
Spring Boots Säulen



Der zweite Teil baut auf dem Beispiel der Einleitung auf und erklärt sowohl das Spring Boot Dependency Management mit Maven und Gradle und wie der Spring Initializr genutzt werden kann, um Projekte und deren Abhängigkeiten zu generieren. Da Sie die Spring-Framework-Grundlagen kennen müssen, bevor ich zeige, wie Spring Boot funktioniert, werden wir diese zuerst behandeln, bevor Sie lernen, wie eine Spring-Boot-Anwendung konfiguriert wird. Das direkt anschließende Kapitel zeigt die »Magie« hinter der automatischen Konfiguration und erklärt, wie sie zu eigenen Zwecken, für eigene Funktionen nutzbar gemacht werden kann.

Ebenfalls Kernbestandteil von Spring Boot selber sind Entscheidungen hinsichtlich Logging, daher wird auch dieses Thema hier aufgegriffen.

Im abschließenden Kapitel möchte ich Ihnen die Spring Boot devtools vorstellen. Diese Werkzeuge können Ihren Entwicklungsprozess zusätzlich beschleunigen.

2 Projektstruktur

Spring Boot möchte übliche funktionale und nicht funktionale Probleme, die einer schnellen Produktivsetzung im Weg stehen, effektiv und effizient lösen, und zwar in einer konsistenten Art und Weise. Dazu gehört nicht nur ein vereinheitlichter Konfigurationsmechanismus, sondern auch eine Projektstruktur, die in gleicher Art und Weise in unterschiedlichen Projekten benutzt wird.

Letzten Endes ist Spring Boot »nur« eine Java-Bibliothek, die als Abhängigkeit eingebunden und benutzt wird – allerdings vereinfachen die folgenden Empfehlungen nicht nur den Entwicklungsprozess, sondern auch die Verteilung der fertigen Anwendungen und Pflege derselben.

2.1 Build-Management-Tools

Eine Anwendung, die Spring Boot benutzt, kann sowohl mit Maven oder Gradle problemlos kompiliert und verteilt werden. Beide Buildsysteme unterstützen die zentrale Verwaltung von Abhängigkeiten. Es ist prinzipiell möglich, eine Spring-Boot-Anwendung mit Ant oder anderen Systemen zu bauen, aber nicht empfehlenswert, da Sie viele der für Maven und Gradle verfügbaren Hilfsmittel nachbauen müssten.

Besonders hervorzuheben ist die kuratierte Liste von Abhängigkeiten innerhalb eines Spring-Boot-Projektes. Jedes Spring-Boot-Release wird begleitet von einer Liste von Abhängigkeiten und expliziten Versionen, gegen die Spring Boot sowie das Spring-Framework getestet wurden und von denen bekannt ist, dass sie mit Spring Boot funktionieren. Die Versionen dieser Abhängigkeiten müssen nicht angegeben werden, können aber durch Setzen von Eigenschaften überschrieben werden. Die Versionen der Abhängigkeiten sind im Anhang F der Spring-Boot-Referenz dokumentiert, und in der Regel können die zu setzenden Eigenschaften aus der ID des Artefakts abgelesen werden. Falls das nicht möglich ist, sind sie gleichermaßen für Maven- und

Gradle-Projekte im dem POM des Moduls `spring-boot-dependencies`[1] sichtbar.

Während Versionen von Bibliotheken Dritter teilweise gefahrlos überschrieben werden können, wird dringend davon abgeraten, die Version des Spring-Frameworks selber zu überschreiben, da jedes Spring-Boot-Release auf genau einer bestimmten Version des Spring-Frameworks aufbaut.

2.1.1 Maven

Maven: Sammler des Wissens

Der Begriff Maven entstammt dem Jiddischen und bedeutet in etwa soviel wie »Der, der versteht«, Connoisseur, Experte oder auch Sammler des Wissens: Maven ist ein Build-Management-Tool, das vom Gedanken der »Konvention vor Konfiguration« getrieben wird. Maven geht von einem Zyklus der Softwareerstellung aus, der häufig durchlaufen wird: Validierung, Kompilierung, Testen, Paketieren, Integrationstests, Verifizierung, Installation und Verteilung.

Die notwendigen Informationen zum Bau eines Maven-Projektes, die sich nicht aus Konventionen ableiten lassen, werden in einem *Project Object Model*, dem POM, gespeichert. Das POM wird in der Regel in einer XML-Datei namens `pom.xml` abgespeichert.

POMs können voneinander erben. Damit werden zentrale Definitionen und Konfigurationen ermöglicht, die an Teilprojekte weitergegeben werden können.

Eine wichtige Aufgabe von Maven ist die Auflösung von Abhängigkeiten, die ein Softwareprojekt hat. Dies kann aus lokalen Quellen oder Quellen im Intranet oder Internet geschehen. Beide Quellen heißen Repositorys.

Maven kann darüber hinaus zentrale Eigenschaften (»properties«) eines Projektes verwalten, Quelltexte und andere Ressourcen während des Bauens filtern und vieles mehr.

Maven hat eine modulare Architektur, die über Plugins erweitert werden kann. Mit dem Spring-Boot-Maven-Plugin steht Ihnen ein Plugin zur Verfügung, das Ihnen unter anderem beim Bau ausführbarer Artefakte hilft.

Neu generierte Spring-Boot-Projekte setzen Maven 3.5 ein.

Erben … In Abschnitt 1.1 wurde in Listing 1–2 gezeigt, wie ein Maven-Projekt aufgebaut werden muss, so dass es von `spring-boot-starter-parent` erbt und in den Genuss der zentralen »Bill of Materials« kommt.

`spring-boot-starter-parent` konfiguriert weiterhin, welche Kodierung die Quelltexte haben (per Default UTF-8), die Java-Version und

[1] https://github.com/spring-projects/spring-boot/blob/master/spring-boot-dependencies/pom.xml

einiges mehr: Spring-Boot-Projekte sind damit sehr konsistent untereinander.

Von einiger Wichtigkeit für zentrale Module des Spring-Ökosystems ist Java-8s Möglichkeit, die Namen von Konstruktor- und Methodenparameter aus den Class-Files zu ermitteln. Diese Namen können zum Beispiel in SpEL-Ausdrücken, in benannten SQL-Abfragen im Spring-Data-Modul oder als Namen von Pfadparametern wie im Spring-Web-MVC-Modul genutzt werden. Vor Java 8 waren diese Namen über Javas Reflection-API nur dann ableitbar, wenn die Quelltexte mit der Option `-debug` übersetzt wurden. Der Java 8 Compiler stellt `-parameter` zur Verfügung. Der `spring-boot-starter-parent` konfiguriert diesen Schalter per Default. Falls Sie dies nicht wünschen, müssen Sie in Ihrem POM das Compiler-Plugin explizit konfigurieren. Umgekehrt gilt das Gleiche: Wenn Sie das Parent-POM nicht nutzen und trotzdem diese Funktion haben möchten, konfigurieren Sie das Compiler-Plugin entsprechend:

Listing 2–1
Konfiguration des Compiler-Plugins, wenn das Parent-POM nicht genutzt wird

```
<plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
                <compilerArgs>
                        <arg>-parameters</arg>
                </compilerArgs>
        </configuration>
</plugin>
```

Quelltexte

override_versions auf GitHub:
https://github.com/springbootbuch/override_versions
override_versions_with_imports auf GitHub:
https://github.com/springbootbuch/override_versions_with_imports

Die geerbten Informationen aus dem `spring-boot-starter-parent`-POM sind nicht in Stein gemeißelt. Ergänzend zum Beispiel aus 1–2 zeigt das Projekt *override_versions*, wie die Version einer transitiven Abhängigkeit, nämlich die des eingebetteten Tomcats, durch Setzen von einer Property überschrieben werden kann, ohne manuell Abhängigkeiten auszuschließen oder selber konsistent Versionen verwalten zu müssen:

Listing 2–2
Erben von »spring-boot-starter-parent« und Überschreiben einzelner Versionen

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.0.RELEASE</version>
</parent>

<properties>
    <!-- Use a certain tomcat version -->
    <tomcat.version>8.5.9</tomcat.version>
</properties>

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

… oder importieren

Falls bereits ein firmenweites Standard-POM vorliegt, von dem abgeleitet werden muss, oder es sonstige Gründe gibt, das Parent-POM nicht zu nutzen, kann die Verwaltung von Abhängigkeiten durch die Spring »Bill of Materials« (BOM) mit Einschränkungen dennoch genutzt werden. Sie kann über den Maven-Gültigkeitsbereich »import« importiert werden, wie Listing 2–3 zeigt:

Listing 2–3
»Manuelles« Importieren der Spring Boot BOM

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>
                spring-boot-dependencies
            </artifactId>
            <version>2.0.0.RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Durch den Import der BOM kommt Ihr Projekt zwar in den Genuss der kuratierten Abhängigkeiten, kann aber nicht von den weiteren Einstellungen des Parent-POM wie dem oben erwähnten Compiler-Plugin profitieren. Darüber hinaus ist das Überschreiben von Versionen deutlich weniger komfortabel und fehleranfälliger: In der »Bill of Materials«

werden durchgängig Propertys für Versionen genutzt. Erben Sie vom Parent-POM, können Sie mit dem Setzen einer Property korrekt die Versionen aller notwendigen Abhängigkeiten ebenfalls setzen. Wenn Sie die BOM wie in Listing 2–3 importieren, können Sie die Abhängigkeiten im Dependency Management nur wie in Listing 2–4 gezeigt überschreiben, das Setzen einer Property wie in Listing 2–2 reicht nicht:

Listing 2–4
Überschreiben von Abhängigkeiten im Dependency Management ohne Parent-POM

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.apache.tomcat.embed</groupId>
            <artifactId>tomcat-embed-core</artifactId>
            <version>8.5.9</version>
        </dependency>
        <dependency>
            <groupId>org.apache.tomcat.embed</groupId>
            <artifactId>tomcat-embed-el</artifactId>
            <version>8.5.9</version>
        </dependency>
        <dependency>
            <groupId>org.apache.tomcat.embed</groupId>
            <artifactId>
                tomcat-embed-websocket
            </artifactId>
            <version>8.5.9</version>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Während Sie mit dem Parent-POM auch von verwalteten Plugin-Versionen profitieren, ist dies ohne Parent-POM nicht möglich. Zu guter Letzt müssen Sie beim reinen Import der Bill of Materials explizit die Java-Version angeben. Seit Spring Boot 2.0 ist Java 8 der Default, Java 6 und 7 werden nicht mehr unterstützt:

Listing 2–5
Die Java-Version muss beim Import der BOM explizit angegeben werden.

```
<properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
</properties>
```

Zusätzliche Werkzeuge

Spring Boot bietet Ihnen zusätzliche Werkzeuge für Maven und Gradle. Zu den Hauptfunktionen zählt die Paketierung der Anwendungen als ausführbares Fat Jar beziehungsweise als ausführbare War-Datei.

Für Maven ist dies das Spring-Boot-Maven-Plugin, das mit

Listing 2–6 Ergänzung des Spring-Boot-Maven-Plugins

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>
                spring-boot-maven-plugin
            </artifactId>
        </plugin>
    </plugins>
</build>
```

eingebunden wird. Auf die Angabe einer Versionsnummer kann bei Verwendung des Parent-POM verzichtet werden.

Wird das Plugin wie oben gezeigt eingebunden, führt ein `mvn package` zu zwei Jar- beziehungsweise War-Dateien: `artefakt.jar` sowie `artefakt.jar.original`, da das Goal `spring-boot:repackage` automatisch während der Maven-Package-Phase aufgerufen wird. Die ursprüngliche Datei beinhaltet dabei nur die Klassen und Ressourcen, die sich aus der Anwendung ergeben, die neu paketierte Jar-Datei hingegen liegt im Executable Jar-Format vor.

In Abschnitt Artefakte wird detailliert beschrieben, wie die Neupaketierung der Anwendung gesteuert werden kann, insbesondere wie Artefakte wie die devtools vom Fat Jar ausgeschlossen werden können.

Filterung von Ressourcen

Das Spring-Boot-Maven-Plugin hat noch eine wichtige Funktion: die Konfiguration der Filterung von Ressourcen (Quelltexte, Propertys und mehr). Maven kann Platzhalter in beliebigen Dateien durch Maven-Propertys, System-Propertys oder Umgebungsvariablen ersetzen. Normalerweise werden diese Platzhalter in `${...}`-Platzhaltern eingefasst. Dieser Platzhalter wird aber ebenfalls für Spring-eigene Eigenschaften beziehungsweise innerhalb der SpEL genutzt. Daher wird dieser Platzhalter umdefiniert und ist in Spring-Boot-Projekten `@...@` (die Maven-Property `resource.delimiter` dient zum Setzen des Platzhalters). Standardmäßig werden alle Konfigurationsdateien (unter anderem `application.properties`, `application.yml`) sowie deren profilspezifische Abkömmlinge gefiltert.

Build-Informationen in der Anwendung

Oftmals werden Sie die Anforderung bekommen, Informationen über den Build in der Anwendung anzuzeigen. Das Spring-Boot-Maven-

Plugin stellt zu diesem Zweck das Goal `build-info` zur Verfügung, das Sie allerdings manuell konfigurieren müssen:

Listing 2–7
Ausführung des Goals build-info vor der Paketierung

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <executions>
        <execution>
            <goals>
                <goal>build-info</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

2.1.2 Gradle

Gradle: Erwarte das Unerwartete

Gradle ist ein auf Groovy basierendes Build-Management-Tool. Im Gegensatz zu Maven-Projektdefinitionen sind Gradle-Skripte direkt ausführbarer Code.

Gradle versucht, das Prinzip »Konvention vor Konfiguration« aus Maven-Projekten mit der Flexibilität anderer Build-Management-Tools wie *ant* zusammenzubringen.

Da Gradle für Builds von Softwaresystemen mit einer Vielzahl von Teilprojekten entworfen wurde, unterstützt Gradle insbesondere inkrementell und parallel ablaufende Build-Prozesse.

Viele Standardlebenszyklen wurden aus Maven übernommen. Ein Java-Projekt, das sich an diese Konventionen hält, kann zum Beispiel alleine mit `apply plugin: 'java'` gebaut werden.

Verschiedene Abschnitte des Builds werden in Gradle »Tasks«, Aufgaben, genannt. Sie können beliebig komplexe Aufgaben erledigen und auch nahezu beliebig voneinander abhängen: Erwarte das Unerwartete.

Die Definition eines Projektes wird in der Datei `build.gradle` hinterlegt.

Projekte auf Basis von Spring Boot 2 benötigen mindestens Gradle in Version 4, damit alle Plugins korrekt funktionieren.

Das Build-Management-Tool Gradle kann auf die gleichen Repositorys wie Maven zugreifen und beinhaltet im Kern ein System zur Verwaltung von Abhängigkeiten. Allerdings müssen Sie das zentrale Dependency Management, bei dem Versionen in einer zentralen Stelle vorgegeben werden, über ein Plugin, dem Spring-Boot-Gradle-Plugin, nachrüsten.

Es kann nach Deklaration im Gradle-`buildscript`-Block in der Datei `build.gradle`, dem Pendant zur `pom.xml`, mittels `apply` genutzt werden:

Listing 2–8 *Nutzung des Spring-Boot-Gradle-Plugins*

```
buildscript {
    ext {
        springBootVersion = '2.0.0.RELEASE'
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-
        ↪  plugin:${springBootVersion}")
    }
}

apply plugin: 'org.springframework.boot'
apply plugin: 'io.spring.dependency-management'
```

Ein vollständiges Beispiel steht in Abschnitt 16.1 im Listing 16–1 zur Verfügung. Im `dependencies`-Block können nun – wie auch mit Maven – die Versionsnummern weggelassen werden:

Listing 2–9 *Nutzung von Dependency Management mit Gradle*

```
dependencies {
    compile "org.codehaus.groovy:groovy"
    compile "org.springframework.boot:spring-boot-starter-web"
    testCompile "org.springframework.boot:spring-boot-starter-test"
}
```

Das Spring-Boot-Plugin konfiguriert den Build-Prozess automatisch so, dass ausführbare Jar-Dateien erzeugt werden. Darüber hinaus sind die zur Verfügung stehenden neuen Gradle-Aufgaben ähnlich den Maven-Zielen, wie Tabelle 2–1 zeigt.

Tab. 2–1 *Aufgaben des Spring-Boot-Gradle-Plugins*

Name	Funktion
`gradle bootRun`	Führt die Anwendung aus dem aktuellen Verzeichnis heraus aus; über das Attribut `addSources` kann definiert werden, ob statische Ressourcen automatisch neugeladen werden, wenn die Entwicklungswerkzeuge im Klassenpfad sind
`gradle bootJar`	Erzeugt eine ausführbare Jar-Datei oder War-Datei
`gradle bootWar`	Erzeugt eine ausführbare War-Datei

Um die Datei `build-info.properties` mit den Metainformationen des aktuellen Builds zu erzeugen, muss der Spring-Boot-Task ähnlich wie im Maven-Pendant angepasst werden:

```
springBoot  {
    buildInfo()
}
```

Listing 2–10
Erzeugung der Build--Metainformationen mit Gradle

(...)

9.1 Minimale Autokonfiguration

Während der `spring-boot-starter-security` in Spring-Boot-1-Anwendungen noch etliche Annahmen getroffen hat und viele Konfigurationseigenschaften bereitstellte, diese Annahmen zu überschreiben, so wurde dieses Verhalten mit Spring Boot 2 grundsätzlich geändert. Falls Sie Spring Boot Security 2 über den Starter einbinden, werden alle Web-Endpunkte – inklusive der Actuator-Endpunkte – geschützt. Der Starter stellt dabei einen Default-Benutzer bereit, dem – je nach angefragtem Inhaltstyp – Basic- oder Form-Login angeboten wird. Dieses Verhalten entspricht dem Standardverhalten von Spring Security. Damit stellt dieser Starter, der im Folgenden auch als Spring Boot Security 2 bezeichnet wird, nur eine minimale Autokonfiguration zur Verfügung.

Jegliche Konfiguration, die über die Abschaltung der vollständigen HTTP-Sicherheit mittels `security.basic.enabled = false` (für den Fall, dass Sie zum Beispiel nur Methodensicherheit nutzen möchten) und der Bereitstellung eigener Zugangsdaten hinausgeht, erfordert das Vorhandensein einer Bean vom Typ `WebSecurityConfigurerAdapter`. Sobald eine solche Bean im Kontext vorhanden ist, bewirkt der Starter lediglich ein grundsätzliches `@EnableWebSecurity` und stellt darüber hinausgehende Autokonfiguration ein. Diese Entscheidung wurde getroffen, damit Sie sich als Entwickler explizit mit der Konfiguration von Security beschäftigen müssen. Die Gefahr, etwas falsch zu konfigurieren oder von übrig gebliebenen Fragmenten der automatischen Konfiguration überrascht zu werden, ist im Bereich Security mit größerem Risiko verbunden.

`WebSecurityConfigurerAdapter` ist eine abstrakte Klasse und stellt Schnittstellen zur Verfügung, die als Parameter einen Builder zur Konfiguration von Spring Security erhalten.

9.2 Die Grundlagen verstehen

Spring Security nutzt folgende Hauptbestandteile, um die Themen Authentifizierung und Autorisierung zu realisieren:

- Der `SecurityContextHolder` realisiert den Zugriff auf den `SecurityContext`.
- Der `SecurityContext` beinhaltet die `Authentication`-Instanz sowie gegebenenfalls Request-spezifische Informationen.
- `Authentication` repräsentiert einen erfolgreich authentifizierten Principal.
- Ein oder mehrere `GrantedAuthority`-Instanzen beschreiben die Rechte einer `Authentication`.

- Eine Instanz des `UserDetailsService` stellt eine Schnittstelle bereit, um mittels eines Benutzernamens oder eines Zertifikats die `UserDetails` zu ermitteln, deren Authentizität überprüft wird.

Der `SecurityContextHolder` ist zentrales Element von Spring Security. In der Standardkonfiguration speichert diese Klasse alle Informationen, insbesondere über den aktuellen Principal in einem `ThreadLocal`: Damit stehen diese Informationen innerhalb des ausführenden Threads immer zur Verfügung. Spring Security trägt weiterhin Sorge dafür, dass der `ThreadLocal` nach Ablauf eines Requests bereinigt wird.

Damit ist auch klar, dass Spring Security nicht für alle Arten von Anwendungen ohne zusätzliche Konfiguration gleichermaßen gut geeignet ist: Anwendungen, in denen jeder Thread demselben Principal zugeordnet wird, zum Beispiel in einer JavaFX-Anwendung, oder reaktive Anwendungen, in denen Principals einer Kette von Ereignissen und nicht einzelnen Threads zugeordnet werden müssen, müssen gesondert betrachtet werden. In diesem Kapitel werden wir nur über Spring Security im Webkontext sprechen, allerdings sind die im Folgenden vorgestellten Konzepte unabhängig vom `SecurityContextHolder` gültig.

9.2.1 Authentifizierung

Spring Security ermöglicht Authentifizierung über folgende Technologien, die entweder Teil des Spring-Security-Projekts sind oder von Dritten bereitgestellt werden:

- HTTP-Basic-Authentifizierung
- HTTP-Digest-Authentifizierung
- HTTP X.509 client certificate exchange
- LDAP
- Form-based-Authentifizierung
- OpenID-Authentifizierung
- Java Authentication and Authorization Service (JAAS)
- und viele weitere mehr

Abbildung 9–1 zeigt die Kernkomponenten von Spring Security. Die Authentifizierung wird an Provider delegiert, die vollständig initialisierte `Authentication`-Objekte zurückgeben oder eine Exception werfen, wenn die Authentifizierung nicht erfolgreich war.

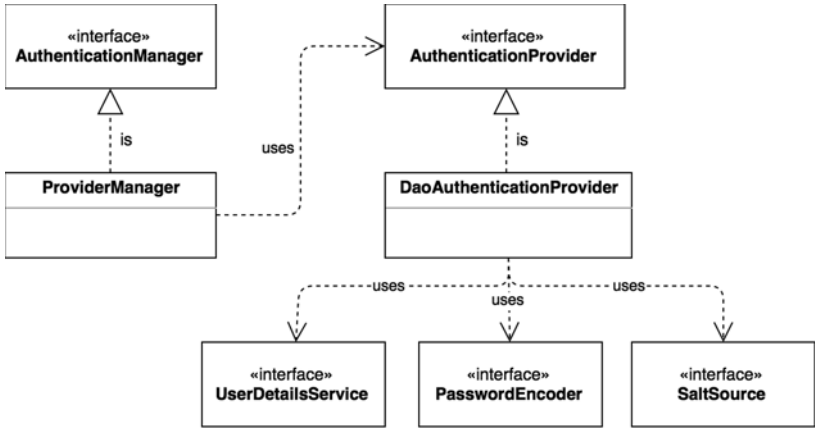


Abb. 9–1
Kernkomponenten von Spring Security

Spring Security stellt eine ganze Reihe von Implementierungen des `AuthenticationProvider` bereit, sowohl intern als auch über externe Module. So gibt es LDAP-, OpenID- oder Keycloak-spezifische Implementierungen. Der im Bild gezeigte `DaoAuthenticationProvider` ist eine der ältesten Implementierungen. Er nutzt einen `UserDetailsService`, um Benutzerdaten aus verschiedenen Quellen zu laden. Das Projekt selber stellt eine In-Memory- sowie eine Jdbc-Implementierung zur Verfügung.

Der `spring-boot-starter-security` konfiguriert für Sie einen `AuthenticationManager` mit einem In-Memory-`UserDetailsService`. Dieser Service enthält genau einen Benutzer. Ebenfalls konfiguriert werden Ereignisse, die über den in Abschnitt A.1 beschriebenen Mechanismus veröffentlicht werden. Zu den veröffentlichten Ereignissen gehören unter anderem `AuthenticationSuccessEvent` und `AbstractAuthenticationFailureEvent`, Letzteres in verschiedenen Ausprägungen. Zugriffe auf Ressourcen durch einen authentifizierten Principal ohne entsprechende Berechtigung lösen Access-Denied-Ereignisse aus.

9.2.2 Autorisierung

Autorisierung beziehungsweise die Durchsetzung von Berechtigungen wird üblicherweise als querschnittlicher Aspekt einer Anwendung betrachtet; innerhalb von fachlichem Code sollten explizite Abfragen, ob eine bestimmte Aktion erlaubt ist oder nicht, vermieden werden. Spring Security realisiert das in Hinblick auf Methodenaufrufe über Springs AOP-Support (siehe Abschnitt 3.2) und für HTTP-Requests über Standard-Servlet-Filter.

Abbildung 9–2 zeigt die Komponenten von Spring Security, die an der Entscheidungsfindung, ob ein Principal für ein bestimmtes Objekt autorisiert ist, beteiligt sind. Es wird eine von mehreren Implementie-

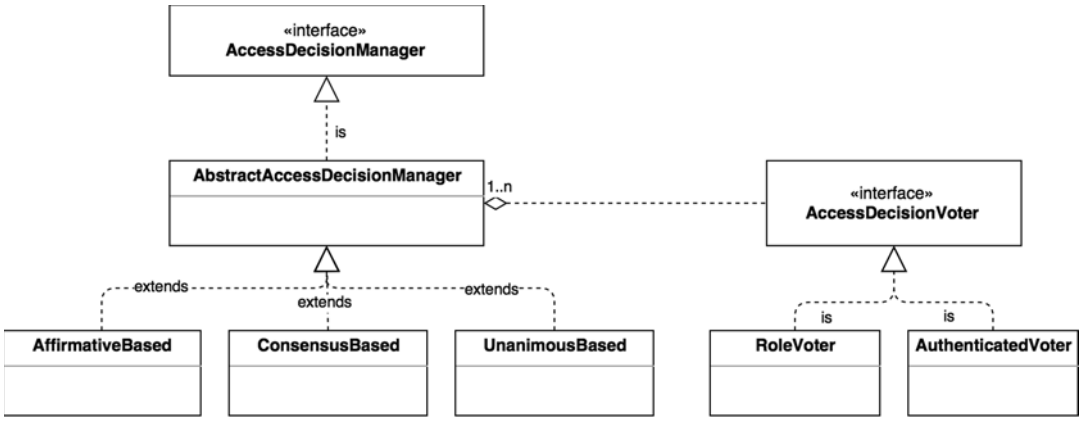


Abb. 9–2
Entscheidungsfindung

rungen des `AccessDecisionManager` genutzt, um zu entscheiden, ob ein Principal, repräsentiert durch ein `Authentication`-Objekt, Zugriff auf ein gesichertes Objekt hat oder nicht. Dabei kann die Entscheidung einheitlich, bestätigend oder über einen Konsens erfolgen. Wie eine Entscheidung getroffen wird, hängt dabei von einem oder mehreren `AccessDecisionVoter` ab. Diese Objekte können zum Beispiel auf den Authentifizierungsstatus oder die Rollen eines Principals zugreifen.

`spring-boot-starter-security` nutzt dieselbe Default-Konfiguration wie Spring Security und stellt für Ihre Anwendung einen `AffirmativeBased AccessDecisionManager` zur Verfügung, der sowohl `RoleVoter` als auch `AuthenticatedVoter` nutzt.

9.2.3 Spring Security und Spring Web MVC

(...)