Christian Ullenboom

Dependency Inj
@Autowired, Mo
@Value, @Entity
JpaRepository,
@RestControlle

```
@RestController
public class StatisticRestController {

    private final ProfileRepository profiles;

    public StatisticRestController( ProfileRepository pr
        this.profiles = profiles;
    }

    @RequestMapping( "/api/stat/total" )
    public String totalNumberOfRegisteredUnicorns()
        return String.valueOf( profiles.count() );
    }
}
```

Spring Boot 3 und Spring Framework 6

Das umfassende Handbuch

- Professionelle Enterprise-Anwendungen mit Java
- Einführung, Konzepte und Werkzeuge, bewährte Praktiken
- Spring-managed Beans, Datenzugriffsschicht, RESTful Services, testgetriebene Entwicklung u. v. m.

Alle Beispielprojekte zum Download

Rheinwerk Computing

Inhalt

Vorwort

Was ist Spring Boot?

Spring Boot ist aktuell das wichtigste Java-Enterprise-Framework und bietet der Entwicklergemeinschaft viele Vorteile: Es ist leicht zu lernen und zu verwenden und erleichtert die Entwicklung von Microservices. Richtig entwickelt, sind Spring-Boot-Anwendungen skalierbar und eignen sich daher perfekt für robuste Unternehmensanwendungen.

Vorkenntnisse und Zielgruppe

Spring ist ein sehr beliebtes Framework in der Java-Community und hat sich in den vergangenen Jahren zu einem Quasistandard entwickelt. Daher wurde dieses Buch für alle konzipiert, die Unternehmensanwendungen mit Datenbankanbindung und Webservices entwickeln möchten. Es ist sowohl für Anfänger als auch für Fortgeschrittene geeignet und liefert zahlreiche praktische Beispiele.

Voraussetzung sind robuste Java-Kenntnisse und der sichere Umgang mit Werkzeugen wie Maven. Mit relationalen Datenbanken sollte die Leserschaft grundlegend vertraut sein.

Die Demoanwendung Date4u, Aufgaben und Lösungen

Mit diesem Handbuch und einer Entwicklungsumgebung des Vertrauens lassen sich Spring-Programme entwickeln. Um ein neues Framework zu erlernen, reicht das Lesen aber nicht aus. Wer eine Programmiersprache oder ein Framework erlernen möchte, muss sie bzw. es wie eine Fremdsprache üben und »sprechen«, und neben der Grammatik der Sprache ist Spring das Vokabular.

Zur Demonstration wird in diesem Buch eine Dating-Anwendung für Einhörner implementiert. Dieses unterhaltsame Beispiel demonstriert alle zentralen Bestandteile einer Enterprise-Anwendung: das Zusammenspiel von Komponenten, die Speicherung der Profildaten in einer Datenbank und den Zugriff über Webservices. Die Leserschaft wird Schritt für Schritt durch die Anwendung geleitet, und im Text eingestreut finden sich immer wieder Aufgaben, die zur Übung einladen. Zu jeder Aufgabe folgt ein Lösungsvorschlag, sodass sich der Text linear lesen lässt. Die Anwendung *Date4u* ist per Download verfügbar (siehe *https://rheinwerk-verlag.de/5980*).

Organisation der Kapitel

Das Buch gliedert sich in folgende Kapitel:

Kapitel 1, »Einleitung«, stellt das *Spring Framework* und die »Erweiterung« *Spring Boot* vor. Es wird gezeigt, wie ein Projekt angelegt wird. Außerdem werden der Aufbau der Maven-POM-Datei sowie die Bedeutung der Starter und Dependencies erläutert.

In **Kapitel 2, »Container für Spring-managed Beans«**, steht der Spring-Kontext im Mittelpunkt. Schritt für Schritt wird aus einer einzigen Klasse ein Geflecht von Spring-managed Beans aufgebaut, um Bilder im Dateisystem abzulegen und zu laden.

Die Verwaltung von Spring-managed Beans ist eine Kernaufgabe des Spring Frameworks, aber es gibt viele weitere Features, die **Kapitel 3, »Ausgewählte Module des Spring Frameworks«**, anspricht. Dazu zählen die externe Konfiguration, die Ereignisbehandlung, Konvertierungsklassen und diverse Utility-Klassen.

In **Kapitel 4, »Ausgewählte Proxies«**, geht es mit besonderen »Ringen« weiter, die sich um Komponenten legen und damit das Caching, asynchrone Aufrufe oder die Validierung übernehmen. Das bietet den Vorteil, dass bestimmte Aufgaben an das Framework verlagert werden können, sodass der eigene Code schlanker wird.

In den ersten vier Kapiteln stehen der Spring-Container und Hilfsklassen im Mittelpunkt, aber keine Speichertechnologien. Das ändert sich in **Kapitel 5, »Relationale Datenbanken anbinden«**. In diesem Kapitel wird ein Datenbankmanagementsystem vorbereitet und werden SQL-Anfragen an die Datenbank demonstriert.

Das Schreiben in Datenbanken mit der JDBC-API ist einfach und direkt, aber unkomfortabel. Daher führt **Kapitel 6, »Jakarta Persistence mit Spring«**, in das objektrelationale Mapping ein.

Datenzugriffsschichten sind in jeder größeren Anwendung nötig. **Kapitel 7, »Spring Data JPA«**, stellt ihre Möglichkeiten exemplarisch an dem Familienmitglied *Spring Data JPA* vor. Repositories lassen sich damit einfach schreiben, und viele Abfragen und Datentransformationen werden vom Framework realisiert.

Relationale Datenbanken werden flankiert von nichtrelationalen Datenbanksystemen, die ebenfalls ausgezeichnet in die Spring-Data-Familie integriert sind. **Kapitel 8, »Spring Data für NoSQL-Datenbanken«**, zeigt anhand von MongoDB und Elasticsearch, wie die Spring-Data-Konzepte in die NoSQL-Welt übertragen werden.

Nachdem mit der Datenhaltung ein wichtiger Teil der Infrastruktur beschrieben wurde, geht es mit Anwendungen des HTTP-Protokolls in **Kapitel 9, »Spring Web«**, weiter. Dort stehen RESTful Webservices und HTTP-Clients im Mittelpunkt.

Das **Kapitel 10**, **»Spring AI«**, bietet einen Einblick in die neue API zur Nutzung von Large Language Models.

Mit dem bisher erworbenen Wissen lässt sich eine Anwendung betreiben, aber es ist wichtig, auch von außen in die Anwendung schauen zu können, um zu prüfen, ob die »Betriebstemperatur« stimmt. **Kapitel 11, »Logging und Monitoring«**, stellt mit dem Actuator-Projekt eine Möglichkeit vor, Daten einer Anwendung freizugeben, die dann von externen Lösungen abgeholt und visualisiert werden können.

Dass ein Programm in der Entwicklungsumgebung läuft und bei den Testfällen nicht versagt, ist gut, aber nur die halbe Miete: Ein Programm muss ausgespielt werden. Dazu stellt **Kapitel 12, »Build und Deployment«**, unter anderem OCI-Container (aka Docker) vor.

Die *-Abschnitte

Das Spring-Universum ist reich an Feinheiten und verwirrt Einsteiger und Einsteigerinnen oft. Um das nötige Wissen vom verzichtbaren Wissen zu trennen, enden einige Überschriften mit einem *, was bedeutet, dass dieser Abschnitt übersprungen werden kann, ohne dass der Leserschaft etwas Wesentliches für die späteren Kapitel fehlt.

Welche Java-Version wird im Buch genutzt?

Die Mindestanforderung für die Nutzung von Spring Boot 3 ist Java 17. Da Java 21 jedoch die neueste Long-Term-Support-(LTS-)Version darstellt – was bedeutet, dass Laufzeitumgebungen von den Herstellern über viele Jahre hinweg mit Updates und Support versorgt werden –. basiert das Buch auf Java 21. Spring Boot 3 ist auch mit aktuelleren Java-Versionen wie Java 23 kompatibel.

Benötigte Software

Wollen wir Java-Programme laufen lassen, benötigen wir eine JVM. In der Anfangszeit war das einfach. Die Laufzeitumgebung stammte erst von Sun Microsystems, später von der Firma Oracle, die Sun übernommen hat. Heute ist das deutlich unübersichtlicher. Diverse Institutionen kompilieren das *OpenJDK* (die Originalquellen enthalten Hunderttausende Zeilen C[++]- und Java-Quellcode) und bündeln es in eine Distribution, die Java-Laufzeitumgebung. Die bekanntesten Laufzeitumgebungen sind:

- *Eclipse Adoptium* (*https://adoptium.net*),
- *Amazon Corretto* (*https://aws.amazon.com/de/corretto*),
- *Red Hat OpenJDK* (*https://developers.redhat.com/products/openjdk/overview*)

Hinzu kommen weitere, wie die von *Azul Systems* oder *Bellsoft*. Zum Lernen mit diesem Buch kann man sich frei für eine Distribution entscheiden; mit Adoptium liegt man gut.

Die Entwicklungsumgebung

Java-Quellcode ist nur Text, sodass im Prinzip ein einfacher Texteditor reicht. Allerdings können wir mit einem Editor wie Notepad keine große Produktivität erwarten. Moderne Entwicklungsumgebungen unterstützen uns in vielerlei Hinsicht: durch farbige Hervorhebung von Schlüsselwörtern, automatische Codevervollständigung, intelligente Fehlerkorrektur, durch das Einfügen von Codeblöcken, die Visualisierung von Zuständen im Debugger und durch vieles mehr. Es ist daher ratsam, eine vollständige Entwicklungsumgebung zu verwenden. Drei populäre IDEs sind: *IntelliJ*, *Eclipse* und *Visual Studio Code*. Hingegen verliert *(Apache) NetBeans* immer weiter den Anschluss.

Wie bei den Java-Laufzeitumgebungen ist es jedem und jeder überlassen, sich eine Entwicklungsumgebung auszuwählen. Eclipse, NetBeans und Visual Studio Code sind frei und quelloffen. Die *IntelliJ Community Edition* ist ebenfalls frei und quelloffen, die mächtigere *IntelliJ Ultimate Edition* kostet hingegen Geld. Die Ultimate-Version der IntelliJ ist sicherlich die leistungsfähigste Java-IDE auf dem Markt, und ihre Unterstützung von Spring ist vorbildlich und von anderen IDEs unerreicht.

Konventionen

In diesem Buch wird eine Reihe von Konventionen verwendet.

Neu eingeführte Begriffe sind *kursiv* gesetzt, und der Index verweist genau auf diese Stelle. Des Weiteren erscheinen *Dateinamen*, *HTTP-Adressen* und *Dateiendungen* (*.txt*) in kursiver Schrift.

Begriffe der Benutzeroberfläche stehen in KAPITÄLCHEN.

Für Code und Listings gelten weitere Regeln:

- Listings, Methoden und sonstige Programmelemente sind in `nichtproportionaler` Schrift gesetzt.
- An einigen Stellen wurde hinter eine Listingzeile ein abgeknickter Pfeil (↩) als Sonderzeichen gesetzt, das den Zeilenumbruch markiert. Der Code aus der nächsten Zeile gehört also noch zur vorangehenden. Bei Methodennamen und Konstruktoren folgt immer ein Klammerpaar, um die Methoden/Konstruktoren von Objekt-/Klassenvariablen abgrenzen zu können. So ist bei der Schreibweise `System.out` und

`System.gc()` klar, dass Ersteres eine statische Variable ist und Letzteres eine statische Methode.

- Hat eine Methode bzw. ein Konstruktor Parameter, so stehen sie in Klammern: `run(String... args)`. In der Regel wird der Parametername ausgelassen, also heißt es kurz: `run(String...)`. Ist der Rückgabetyp im Kontext relevant, steht er mit dabei, etwa so: `ConfigurableApplicationContext run(String... args)` bzw. `ConfigurableApplicationContext run(String...)`. Hat eine Methode bzw. ein Konstruktor eine Parameterliste, ohne dass diese gerade relevant ist, wird sie mit Auslassungspunkten abgekürzt. Beispiel: »Eine `run(…)`-Methode startet den Container.« Ein leeres Klammerpaar bedeutet demnach, dass eine Methode bzw. ein Konstruktor wirklich keine Parameterliste hat.
- Um die Parameterliste kurz und dennoch präzise zu machen, gibt es im Text teilweise Angaben wie `getProperty(String key[, String def])`, was eine Abkürzung für »`getProperty(String key)` und `getProperty(String key, String def)`« ist. Auch an anderen Stellen finden sich die Auslassungspunkte …, wenn die Implementierung oder Bildschirmausgaben für das Verständnis nicht weiter erforderlich sind.
- Um eine Gruppe von Methoden anzugeben, symbolisiert die Kennung * einen Platzhalter. So steht zum Beispiel `print*(…)` für die Methoden `println(…)`, `print(…)` und `printf(…)`. Aus dem Kontext geht hervor, welche Methoden gemeint sind.
- Lange Paketnamen werden teilweise abgekürzt. Zum Beispiel wird aus `org.springframework.data.jpa.repository` die Kurzform `o.s.d.j.r`.
- Kommen Annotationen im Text vor, ist es eigentlich doppelt gemoppelt, etwa »`@Value`-Annotation« zu schreiben, doch erlaubt das `@`-Zeichen im Text das schnellere Erfassen für die Leserschaft.

Um im Programmcode Compilerfehler oder Laufzeitfehler anzuzeigen, steht in der Zeile ein ☠. So ist auf den ersten Blick abzulesen, dass die Zeile entweder nicht kompiliert wird oder zur Laufzeit aufgrund eines Programmierfehlers eine Ausnahme auslöst. Beispiel:

```
SpringApplication.run( WebAppApplication.class, null )
// java.lang.IllegalArgumentException: Args must not be null ☠
```

Teilweise werden von der Kommandozeile (synonym: *Befehlszeile*, *Konsole*, *Shell*) aus Programme aufgerufen. Da jedes Kommandozeilenprogramm eine eigene Prompt-Sequenz hat, wird diese hier im Buch generisch durch ein `$` symbolisiert. Unsere Eingaben sind fett gesetzt. Ein Beispiel:

```
$ java -version
openjdk version "21.0.1" 2023-10-17
OpenJDK Runtime Environment (build 21.0.1+12-29)
OpenJDK 64-Bit Server VM (build 21.0.1+12-29, mixed mode, sharing)
```

An Stellen, an denen ausdrücklich die Windows-Kommandozeile gemeint ist, benutze ich das Prompt-Zeichen >:

```
> ver

Microsoft Windows [Version 10.0.19041.388]
```

Programmlistings

Bei den Listings finden sich in der Regel nur die relevanten Codeausschnitte, um den Umfang des Buches nicht zu sprengen. So tauchen Paketnamen und Importdeklarationen in der Regel nicht auf. Da der meiste Quellcode entweder als Snippet verlinkt oder Teil des Demoprojekts ist, wird der Name der Datei oder die URL zu dem Codeschnipselchen in der Listingunterschrift genannt, etwa so:

Listing 1 ABC.java

```
class ABC { }
```

Der abgebildete Quellcode befindet sich in der Datei *ABC.java*. Das Paket ist nicht angegeben, weil es in den Projekten keine zwei gleichen Typnamen in verschiedenen Paketen gibt. Für andere Ressourcen, wie XML- oder HTML-Dokumente, gilt das gleiche Muster:

Listing 2 pom.xml

```
<xml …>
```

Der komplette Quellcode ist per Download verfügbar (siehe *https://rheinwerk-verlag.de/5980*).

Einige Listings sind nicht Teil des Date4u-Projekts und werden verlinkt, etwa so:

Listing 3 https://gist.github.com/ullenboom/80cdb6c7435980c14b887a85d4b667ec

```
# http://web.archive.org/web/20111224041840/http://ww…
```

Da sich oftmals ein Codeblock im Laufe der Zeit weiterentwickelt, macht die Ergänzung »Erweiterung« darauf aufmerksam; der neue Code ist in der Regel fett gesetzt:

Listing 4 Date4uApplicationTests, Erweiterung

```
class ABC {
  public final static String INTRO = "Wann geht's endlich los?";
}
```

Über die richtige Programmier-»Sprache«

Die Programmiersprache in diesem Buch ist Englisch, um ein Vorbild für »echte« Programme zu sein. Bezeichner wie Klassennamen, Methodennamen und auch eigene API-Dokumentationen sind hier auf Englisch, um eine Homogenität mit der englischsprachigen Java-Bibliothek und dem Spring Framework zu schaffen.

Download und Onlineinformationen

Die Downloaddateien zum Date4u-Projekt sowie Updates zum Buch finden sich auf der Webseite *https://rheinwerk-verlag.de/5980*. Alle Programmteile von Date4u sind frei von Rechten und können ungefragt in eigene Programme übernommen und modifiziert werden. Das ZIP-Archiv aus dem Download enthält folgende Bestandteile:

- **main:** Dies ist die Hauptanwendung Date4u mit Services zum Upload und Download von Bildern und zum Datenbankzugriff über Jakarta Persistence. Die Spring Shell ermöglicht die interaktive Anwendung mit Shell-Kommandos, etwa zur Auflistung der Profile. RESTful Webservices machen Profildaten von außen zugänglich. Das Projekt basiert auf Maven und lässt sich nahtlos in jede Java-IDE integrieren.

- **unicorns:** Das ist ein Verzeichnis mit 120 verschiedenen Bildern von Einhörnern. Die Dateinamen lauten *unicorn001.jpg* bis *unicorn120.jpg*. Ein ZIP-Archiv mit den Bildern liegt auch unter *https://github.com/ullenboom/120-unicorn-photos*.
- **h2-2.3.232-bin:** Dieses Verzeichnis enthält das *bin*-Verzeichnis des Datenbankmanagementsystems H2. Im *bin*-Verzeichnis befinden sich die Shell-Skripte *h2.bat* (für Windows) und *h2.sh* (für Unix/Linux). Mit diesen Skripten lässt sich die Datenbank direkt starten, vorausgesetzt, die JVM ist korrekt im Systempfad konfiguriert.
- **unicorn-database:** Dieses Verzeichnis enthält die Datei *unicorns.sql* mit einem SQL-Skript für das Datenbankmanagementsystem H2 zur Initialisierung der Beispieldatenbank. Sie finden es auch unter *https://tinyurl.com/4fu3hwu4*.
- **product-database:** Dies ist ein Spring-Boot-Programm mit einem Beispiel für den Zugriff auf die NoSQL-Datenbank MongoDB. Es ist ein Maven-Projekt.
- **chat-messages:** Dieses Spring-Boot-Programm beinhaltet ein Beispiel für Elasticsearch. Es ist ein Maven-Projekt.

Über den Autor

Christian Ullenboom tippte im Alter von 10 Jahren seine ersten Zeilen Code in den C64. Nach Jahren intensiver Assembler-Programmierung führte seine Reise nach dem Studium der Informatik und der Psychologie auf die Insel Java. Urlaubsreisen nach Python, JavaScript, TypeScript und Kotlin konnten ihn bisher nicht von der Inselbegabung befreien.

Seit über 25 Jahren ist Christian Ullenboom begeisterter Softwarearchitekt, Java-Trainer (*http://www.tutego.de*) und Ausbilder für Fachinformatiker. Aus seiner Schulungstätigkeit heraus sind mehrere Fachbücher entstanden, unter anderem *Java ist auch eine Insel. Einführung, Ausbildung, Praxis* (17. Auflage Rheinwerk 2024) und *Captain CiaoCiao erobert Java: Das Trainingsbuch für besseres Java*. Die Bücher bringen seit vielen Jahren Lesern die Programmiersprache Java näher. Für sein besonderes Engagement hat Sun (heute Oracle) Christian Ullenboom im Jahr 2005 als eine Persönlichkeit, die sich besonders um Java verdient gemacht hat, mit dem Status eines *Java-Champions* ausgezeichnet.

Der Autor dieses Buches teilt sein Wissen und Erfahrungen auch über Lernvideos unter *https://tutego.learnworlds.com*. Diese Videos behandeln auch eine Einführung in Java und Spring Boot. Gelegentlich postet er Developer-News und weiteren Unsinn auf der Mastodon-Instanz *https://mas.to/@ullenboom*.

Christian Ullenboom ist in Sonsbeck am Niederrhein verwurzelt.

Danksagungen

An dieser Stelle möchte ich mich bei allen Personen bedanken, die auf unterschiedliche Art und Weise zum Gelingen dieses Buches beigetragen haben. Besonderer Dank gilt meiner Frau für ihre Liebe und Geduld und meinen Eltern, ohne die es das Buch nicht geben würde.

Feedback

Auch wenn wir die Kapitel noch so sorgfältig durchgegangen sind, sind bei über 1000 Seiten einige Unstimmigkeiten wahrscheinlich, so wie auch jede Software rein statistisch Fehler aufweist.[1] Wer Anmerkungen, Hinweise, Korrekturen oder Fragen zu bestimmten Punkten oder zur allgemeinen Didaktik hat, sollte sich nicht scheuen, mir eine E-Mail unter der Adresse *ullenboom@gmail.com* zu senden. Ich bin für Anregung, Lob und Tadel stets empfänglich. Auch für Käse und Lakritz.

Vorwort zur 2. Auflage

In dieser neuen Auflage wurden viele kleinere Fehler der ersten Ausgabe behoben, und das Buch wurde auf den neuesten Stand von Spring Boot Version 3.4 und Spring Framework 6.2 gebracht.

Maven kann (und wird) in einigen Fällen Code generieren. Dieser Prozess wird vollständig über Annotationsprozessoren gesteuert, und die dafür notwendige Konfiguration des Maven-Compiler-Plugins wird früh im Buch erklärt, da es kapitelübergreifend ist.

Die KI-generierten Einhornbilder wurden komplett neu erstellt, da die Bildgenerierung innerhalb eines Jahres enorme Fortschritte gemacht hat. Die Qualität der Bilder ist nun deutlich verbessert, und die ZIP-Datei enthält auch Vorschaubilder (Thumbnails).

Im Grundlagenteil wurde die Behandlung optionaler Abhängigkeiten im Konstruktor verbessert sowie das `MessageSource`-Konzept aufgenommen. Das UML-Diagramm zum `TaskExecutor` wurde korrigiert und um virtuelle Threads ergänzt. Der Abschnitt »Am Anfang und Ende« hat sich von Kapitel 2 in das Kapitel 3 verschoben. Der Abschnitt »Validierung testen *« sowie die Details zum `QueryByExampleExecutor` wurden entfernt, da diese Themen sehr speziell sind und ich Raum für neue Inhalte schaffen wollte. Die Annotationen `@MockBean` und `@SpyBean` sind mittlerweile deprecated und

1 Statistisch gibt es bei der Entwicklung etwa 1 bis 25 Fehler pro 1000 Zeilen Code. Für Referenzen siehe *https://stackoverflow.com/questions/2898571*.

wurden durch `@MockitoBean` und `@MockitoSpyBean` ersetzt. Dass Spring Boot auch Konfigurationen validieren kann, zeigt ein neues Beispiel und verzichtet dabei auf Lombok.

Obwohl Jakarta Persistence 3.2 erst mit Hibernate 7.0 vollständig umgesetzt wird und somit kein Bestandteil von Spring Boot 3.4 ist, so wurden doch schon die veralteten Datentypen (`Date`, `Calendar` usw.) aus dem Text genommen.

Der Datenbankabschnitt finden sich nun Informationen zu [`Fetchable`]`FluentQuery`. Die Neuerungen in Spring Data werden ebenfalls berücksichtigt: Die Scroll API findet Erwähnung, und Updates, die sich aus Spring Framework 6.1 ergeben, werden vorgestellt. Dazu gehört der `JdbcClient` mit diversen CRUD-Beispielen. Auch die `@ServiceConnection`-Annotation bei Testcontainers wird behandelt.

Im Webteil wurde das Beispiel für `@RequestParam` ausgetauscht und der `QuoteRestController` früher eingeführt. Neu hinzugekommen sind Erläuterungen zu `@JsonMixin` und zur Verwendung von Java-Records für REST-Rückgaben. Während `POST` und `PUT` sowie HTTP-Statuscodes bisher bei REST aufgeführt waren, ist dies nun getrennt, und das Kapitel über REST beschreibt nur noch die Besonderheiten von REST selbst; `@RequestBody` wird auch ohne REST benötigt. `PagedModel` wird als alternative Rückgabe zu `Page` empfohlen. Beim Testen von HTTP-Anfragen ist `MockMvc` durch `MockMvcTester` ersetzt worden.

Die Dokumentation der Klasse `RestClient` wurde deutlich ausgebaut, und der Spring-HTTP-Client wurde vom reaktiven `WebClient` auf den `RestClient` umgeschrieben. Hinzugekommen ist ein Abschnitt zur Verwendung von `QuerydslPredicate` in der Parameterliste einer Handler-Methode. Außerdem gibt es jetzt einen umfassenden Abschnitt zu Spring GraphQL, der die Implementierung sowohl auf der Server- als auch auf der Clientseite beleuchtet. Ein konkretes Beispiel zur Nutzung von MapStruct rundet die Neuerungen im Kapitel über Spring Web MVC ab.

Im Bereich Logging und Konfiguration wurde das JSON-Logging aufgenommen.

Ein ganz neues Kapitel stellt das spannende neue Modul Spring AI vor, mit dem Sprachmodelle angesprochen werden.

Und jetzt wünsche ich viele Frühlingsgefühle mit Spring Boot.

Kapitel 1
Einleitung

Spring ist ein sehr umfangreiches Framework, und daher wollen wir uns ihm Schritt für Schritt nähern. Als Erstes sind dafür die notwendigen Voraussetzungen zu schaffen, um ein Projekt mit den entsprechenden Dependencies aufzubauen. Am Ende dieses Kapitels werden wir ein erstes Spring-Boot-Projekt aufgebaut haben, das man übersetzen und starten kann. In den folgenden Kapiteln werden wir dieses Projekt immer weiter ausbauen und mit Leben füllen.

1.1 Einleitung und ein erstes Spring-Projekt

Zu Beginn wollen wir uns kurz die Geschichte des Spring Frameworks und die Entstehung von Spring Boot anschauen.

Anfangen wollen wir mit der Frage, warum die *Java Standard Edition* (Java SE) für Enterprise-Anwendungen häufig nicht ausreicht.

Die Aufgaben der Java Standard Edition

Die *Java SE* hat eine klare Aufgabe, nämlich das Fundament für beliebige Anwendungen zu schaffen. Sie enthält grundlegende Funktionalitäten für die Ein- und Ausgabe, Datenstrukturen, Netzwerkkommunikation, Datum-Zeit-Berechnungen und für die Nebenläufigkeit, und ein paar Dinge zur Sicherheit sind ebenfalls inbegriffen. Mit *AWT* und *Swing* sind auch zwei Bibliotheken für grafische Oberflächen mit an Bord.

Bei größeren Aufgaben ist man jedoch gezwungen, eigenen Code zu schreiben oder auf freie oder kommerzielle Bibliotheken zurückzugreifen.

Anforderungen im Enterprise-Bereich

Wenn man komplexe Geschäftsanwendungen schreiben möchte, gibt es eine Reihe von zusätzlichen Anforderungen. Benötigt werden RESTful Webservices, dynamisch generierte Webseiten, der Zugriff auf relationale oder nichtrelationale Datenbanken und Messaging-Systeme, das Monitoring von Anwendungen, E-Mail-Versand und mehr.

Das Problem dabei ist: Die Java SE kann das nicht leisten. Es ist ein erweiterter Technologie-Stack gefragt.

Die Entwicklung von Java-Enterprise-Frameworks

Sun Microsystems, der Java-Schöpfer, veröffentlichte das erste Java-Release Ende 1995/Anfang 1996. Als Ende der 1990er-Jahre das Internet für eine breite Masse zugänglich wurde und dynamisch generierte Webinhalte eine immer größere Rolle spielten, definierte Sun schon Ende 1996 die *Servlet-API*. Das heißt, kurz nachdem Sun Microsystems die Java SE-Plattform freigegeben hatte, dachte das Unternehmen schon in Richtung Unternehmensanwendungen weiter.

Der Servlet-Standard definiert eine API, mit deren Hilfe man dafür sorgen kann, dass im Webserver eigene Programme liegen und angesprochen werden können. Die Servlet-API spielt bis heute eine wichtige Rolle und wird zum Beispiel durch den populären Servlet-Container *Tomcat* implementiert. Während es bei der Servlet-API nur darum geht, auf HTTP-Anfragen zu reagieren, stellte Sun Ende 1999, also knapp 5 Jahre nach Erscheinen der Java SE, dann die *Java Enterprise Edition* vor. Diese Spezifikation wird von einem *Applikationsserver* implementiert. Dieser kann wiederverwendbare Komponenten auf der Serverseite verwalten, die *Enterprise Java Beans* genannt werden.

Die *Java 2 Platform, Enterprise Edition*, auch *J2EE* abgekürzt, war ein interessanter Start, allerdings gab es damit auch eine Menge von Problemen. Zum Beispiel waren die Applikationsserver der ersten Generation sehr speicherintensiv und hatten eine sehr lange Startzeit. Ein weiteres Problem war, dass die Geschäftslogik mit der Java-Enterprise-API sehr eng verwoben war. Wir sprechen in diesem Zusammenhang auch von *invasiven Technologien*. Das bedeutet, dass zum Beispiel bei der Implementierung der Geschäftslogik irgendwelche Java EE-Schnittstellen implementiert werden mussten. Ein anderes Problem war, dass Anwendungen schwierig zu testen waren, im Regelfall nur innerhalb des laufenden Applikationsservers.

Die lange Startzeit bedeutete auch, dass das automatisierte Durchführen von Tests oder Deployments sehr lange dauerte. Zudem hatte die Sprache Java noch nicht so viele Features wie heute; erst in Java 5 wurden zum Beispiel Annotationen eingeführt. Spring nutzt einen stark deklarativen Ansatz, um die Anwendungen möglichst flexibel aufzubauen. Geht man in das Jahr 1999 zurück, gab es noch keine Annotationen. Stattdessen musste viel in XML-Dateien kodiert werden, was damals die einzige Möglichkeit war, etwas deklarativ auszudrücken.

Ein weiteres Problem der Java Enterprise Edition war, dass die Spezifikation zwar ein guter Start war, aber die Spezifikation eben auch viele Fälle nicht abdeckte. Die Hersteller der Applikationsserver waren schnell dabei, für diese Probleme eine eigene Lösung anzubieten; nur das Dilemma war, dass die Anwendung dann natürlich nicht mehr auf jedem Applikationsserver lief, sondern an eine technische Implementierung gebunden war. Man hatte folglich einen Vendor-Lock.

Rod Johnson entwickelt ein Framework

Diese Unzulänglichkeiten waren bekannt, und insbesondere eine Person wollte die Situation verbessern: Rod Johnson. Er entwickelte für sein Buch »J2EE Design and Development« ein neues Framework. Dieses Framework, das er *Interface 21* nannte, war mit rund 30.000 Zeilen Code damals schon recht umfangreich und wurde vollmundig als das »Interface für das 21. Jahrhundert« bezeichnet. Der Name »Spring«, unter dem wir es heute kennen, kam erst ein wenig später. Bemerkenswert ist, dass Rod Johnson ursprünglich gar nicht vorhatte, ein Framework zu entwickeln: Er verstand sich damals eher als Buchautor, der zeigen wollte, wie man mit der J2EE auch moderne Enterprise-Anwendungen entwickeln konnte.

Das Buch erschien im Wrox-Verlag[1], und Johnson bot, wie man das als Buchautor eben macht, den Quellcode seines Frameworks auf der Website des Verlags zum Download an. Im Wrox-Forum fand das Framework insbesondere das Interesse von Jürgen Höller und Yann Caroff. Diese diskutierten im Wrox-Forum den Code und motivierten Rod Johnson, den Code auf SourceForge zu setzen. Anfang 2003 wechselte der Quellcode von einem Download des Wrox-Verlags hin zu einem Projekt auf SourceForge. Yann Caroff schlug auch einen neuen Namen vor, und zwar *Spring*, was im Grunde genauso vollmundig ist wie »Interface 21«, es sollte nämlich der neue Frühling für J2EE-Anwendungen sein. Auf der Website *https://sourceforge.net/projects/springframework* sind noch die alten Artefakte und Beiträge zu finden.

Etwas später gründeten Rod Johnson, Jürgen Höller und Yann Caroff das Unternehmen *Interface 21*. Im März 2004 erschien das erste Release des Spring Frameworks (siehe Abbildung 1.1).

Ein paar Jahre verdiente Interface 21 durch Beratung und die Realisierung von Softwareprojekten Geld. Im Jahr 2007 investierte *Benchmark Capital* 10 Millionen US-Dollar in das Unternehmen Interface 21. Später wurde Interface 21 dann in *SpringSource* umbenannt. *VMware* übernahm im August 2009 SpringSource für rund 420 Millionen US-Dollar. Rod Johnson wird sich gefreut haben; Mitte 2012 verließ er VMware und zog sich aus der Spring-Entwicklung zurück.

Letztlich kann man zusammenfassen, dass das Spring Framework heute weder ein Hobbyprojekt noch ein Nebenprodukt eines Buches ist, sondern ein echtes Geschäft. VMware bezahlt Menschen dafür, das Framework weiterzuentwickeln, und wir haben damit eine robuste Grundlage für das Erstellen von Java-Enterprise-Anwendungen.

1 Der Wrox-Verlag ist seit 2003 insolvent. Einige Titel wurden von John Wiley & Sons übernommen.

Abbildung 1.1 Diese Webseite von Spring aus dem Jahr 2004 kündigte das Release 1.0 an.[2]

Das Spring Framework verlangt viele Konfigurationen

Die Anfänge des Spring Frameworks liegen ungefähr im Jahr 2002, und im März 2004 erschien das erste Release des Spring Frameworks 1.0. Das Spring Framework war eine Reaktion auf die komplexen und schwergewichtigen J2EE-Anwendungen, insbesondere die aufwendige Konfiguration, die mit ihnen einherging.[3] Allerdings waren auch die frühen Spring-Anwendungen aufwendig zu konfigurieren. Beispiele zum Spring Framework 1.0 finden sich zum Beispiel auf der GitHub-Seite *https://github.com/ullenboom/spring-framework-1.0-samples*. Es gibt eine ganze Reihe von Projekten, und ein recht bekanntes ist die *petclinic*. Das Beispiel existiert, in modernisierter Form, heute immer noch.

Im gespiegelten Verzeichnis *https://github.com/ullenboom/spring-framework-1.0-samples/tree/main/petclinic/war/WEB-INF* lassen sich mehrere XML-Dateien ausmachen; das ist typisch für die frühere Konfiguration. Wir müssen bedenken: Damals

2 Das Web Archive hat unter *https://web.archive.org/web/20040330131500/http://www.springframework.org:80/* die alte Website archiviert; die Blumen symbolisieren den Frühling. Im übertragenen Sinne: Enterprise-Anwendungen blühen mit dem Spring Framework wieder auf. SourceForge, eine Hosting-Plattform für Open-Source-Software, war vor 20 Jahren das, was heute GitHub ist.

3 Damals generierte man gerne aus Javadoc die nötigen J2EE-XML-Deskriptoren: *https://xdoclet.sourceforge.net/xdoclet/index.html*

gab es noch keine Annotationen in der Sprache Java, und wer deklarativ arbeiten wollte, hatte keine Alternative, als über XML-Dateien die Anwendung zu konfigurieren.

Schauen wir uns an einem (etwas umformatierten) Ausschnitt von *applicationContext-jdbc.xml* an, wie Komponenten unter Spring konfiguriert wurden:

```
<bean id="dataSource"
  class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName">
    <value>${jdbc.driverClassName}</value>
  </property>
  <property name="url">
    <value>${jdbc.url}</value>
  </property>
  <property name="username">
    <value>${jdbc.username}</value>
  </property>
  <property name="password">
    <value>${jdbc.password}</value>
  </property>
</bean>
<bean id="transactionManager" class=↩
  "org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource">
    <ref local="dataSource" />
  </property>
</bean>
```

Der kleine Ausschnitt zeigt die Definition für zwei Komponenten, die im Spring-Kontext *Spring-managed Beans* genannt werden, und zeigt auch, wie sie konfiguriert werden. Angegeben werden bei `<bean>` die Bean-ID, die Klasse, der Name sowie der Typ. Das geschachtelte Element `<property>` führt später zum Aufruf der Setter der Bean. Datenbankverbindungen benötigen Informationen wie zum Beispiel den Treibernamen der Datenbank, die JDBC-URL, den Benutzernamen und das Passwort. Und so geht die Deklaration dann weiter und weiter. Bei `<ref local="dataSource"/>` lässt sich eine Injektion erkennen, die besagt, dass zum Beispiel die Spring-managed Bean `transactionManager` eine `dataSource` benötigt.

Mit dem stetigen Anwachsen der XML-Dateien wird die Konfiguration immer unübersichtlicher. Aber da es damals nichts anderes als XML-Dateien gab, blieb bei der Entwicklung nur der Weg, auf diese Weise Anwendungen zu konfigurieren. Das Ganze wurde im Spring Framework 2.5 durch die Java-Annotationen und eine Java-Konfigu-

ration deutlich besser (die Spring-Komponenten mussten nicht mehr in XML, sondern konnten in Java definiert werden), was die Konfiguration enorm vereinfachte.

Es blieb jedoch ein anderes Problem: Das Spring Framework ist absolut flexibel in der Konfiguration. Man kann alles auswechseln und konfigurieren. Das klingt erst einmal wie ein Vorteil, allerdings gibt es ein kleines Problem: Wenn man alles konfigurieren kann, dann bedeutet das leider beim Spring Framework, dass man auch erst einmal alles konfigurieren muss, damit irgendetwas läuft. Das heißt, wenn man ein neues, reines Spring-Framework-Projekt aufbaut, braucht man erst mal relativ viel Anlaufzeit (engl. *ramp-up time*), um gewisse Dinge aufzusetzen. Und das war natürlich ein Problem, das gelöst werden musste. Die Lösung ist Spring Boot.

1.1.1 Spring Boot

Im Jahr 2012 schlug Mike Youngstrom in der Mailingliste vor, dass man Spring-Anwendungen eigentlich viel einfacher konfigurieren sollte. Hier ein kleiner Ausschnitt der Nachricht (*https://jira.spring.io/browse/SPR-9888*):

> *I think that Spring's web application architecture can be significantly simplified if it were to provide tools and a reference architecture that leveraged the Spring component and configuration model from top to bottom. Embedding and unifying the configuration of those common web container services within a Spring Container bootstrapped from a simple main() method.*

Auch andere fanden die Vorstellung verlockend, dass man nicht mehr einen Servlet-Container braucht, in dem die Spring-Anwendung deployt wird. Stattdessen soll die Spring-Anwendung eine eigene `main(…)`-Methode haben, um dann, wenn es zum Beispiel um Webanwendungen geht, einen Servlet-Container optional selbst zu starten.

Wenn wir heute von »Spring-Anwendungen« sprechen, dann meinen wir eigentlich meistens Spring-Anwendungen, die über Spring Boot konfiguriert werden. Das Spring Framework besitzt die eigentlichen Fähigkeiten. Gewisse Teile werden ergänzt, zum Beispiel zum Thema Testen und Datenbankzugriff. Aber alle diese Komponenten müssen, damit sie funktionieren, erst mal konfiguriert werden. Und genau das ist die Funktion von Spring Boot. Spring Boot setzt sich sozusagen als Ring um das Framework herum und schafft für alle diese Teile eine entsprechende Standardkonfiguration, sodass man sofort loslegen kann, ohne sich um die ganzen Details der Konfiguration kümmern zu müssen.

Lediglich durch die Existenz einer Klasse oder das Setzen weniger Properties stellt Spring Boot wie durch Magie komplexe Dienste im Kontext zur Verfügung. Das wird *Autokonfiguration* genannt.

Heute ist Spring Boot der Standardweg, wie man Spring-Anwendungen schreibt. Deswegen soll im Folgenden immer nur der allgemeine Begriff »Spring« stehen, wenn die

Kombination aus Spring Boot und Spring Framework gemeint ist; stammen die Technologien explizit aus dem Spring Framework oder Spring Boot, wird das genannt. Im Alltag werden heute Spring-Anwendungen in aller Regel immer über Spring Boot automatisch konfiguriert.

Spring-Boot-Versionen

Die Entwicklungen an Spring Boot begannen 2013. 2014 erschien das erste Release, *Spring Boot 1.0*. Ungefähr vier Jahre später folgte das zweite Major Release, *Spring Boot 2.0*. Das letzte Minor Release von Spring 2.7 gibt es seit Ende Mai 2022. *Spring Boot 3* erschien im November 2022 und basiert auf dem *Spring Framework 6*, das kurz vorher fertiggestellt worden war.

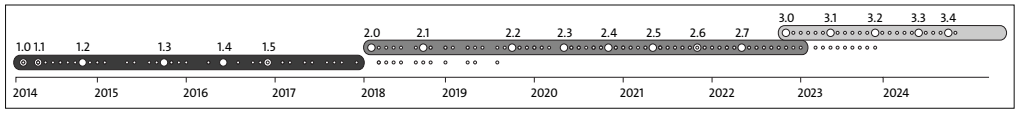


Abbildung 1.2 Release-Daten der Spring-Boot-Versionen

In den Release-Linien von Spring Boot gibt es drei wesentliche Stränge (siehe Abbildung 1.2). Die 1er-Version ist längst veraltet und wird nicht mehr unterstützt. Aktuell migrieren viele Teams von den 2er- auf die 3er-Versionen.

Regelmäßig erscheinen Updates. Während man früher sehr stark auf Feature-Releases setzte, wie in den Anfangsjahren der Java SE und Jakarta EE, so ist auch das Spring-Boot-Team mittlerweile dazu übergegangen, nicht mehr nach Features ein Release zu veröffentlichen, sondern in regelmäßigen Abschnitten. Beim Spring Framework vergehen ungefähr 6 bis 8 Wochen von einem Release zum nächsten. Deswegen sind die Abstände zwischen den Spring-Boot-Versionen 2.2, 2.3 ... in etwa gleich. Zusätzlich gibt es nach den Updates eines Major/Minor Release regelmäßig Patches. Das kann man ganz gut an den kleinen Punkten in den Kreisen in Abbildung 1.2 ablesen. Seit Version 2.3 gibt es im Monatsrhythmus Updates.

Das hat eine wichtige Konsequenz, denn Spring Boot definiert, wie wir später sehen werden, eine ganze Reihe von Unterversionen für Bibliotheken. Diese Versionen müssen wir nicht selbst aktualisieren, diese Aufgabe übernimmt das Spring-Team für uns. Deswegen sollten wir jedoch die Spring-Boot-Versionen regelmäßig aktualisieren, um von sämtlichen Unter-Updates ebenfalls zu profitieren.

Zeitraum für Unterstützung

Das Spring Framework und auch Spring Boot sind quelloffen und stehen unter der Apache-Lizenz 2.0. Gleichwohl gibt es kommerziellen Support von VMware; die Vorteile dokumentiert die Website unter *https://tanzu.vmware.com/spring*. Unterstützung ist besonders wichtig, wenn eine ältere Version weiter genutzt werden muss, wie

es bei Spring Boot 2 der Fall ist, das seit dem 18.11.2023 keinen Support mehr erhält. Kunden mit kommerzieller Unterstützung haben mehr Zeit für eine Umstellung. Einige Unternehmen können nicht auf Java 17 für Spring Boot 3 umsteigen, sodass die kommerzielle Unterstützung die einzige sichere Option ist. Die Webseite *https://spring.io/projects/spring-boot#support* gibt eine Übersicht über die Support-Zeiträume (siehe Abbildung 1.3).

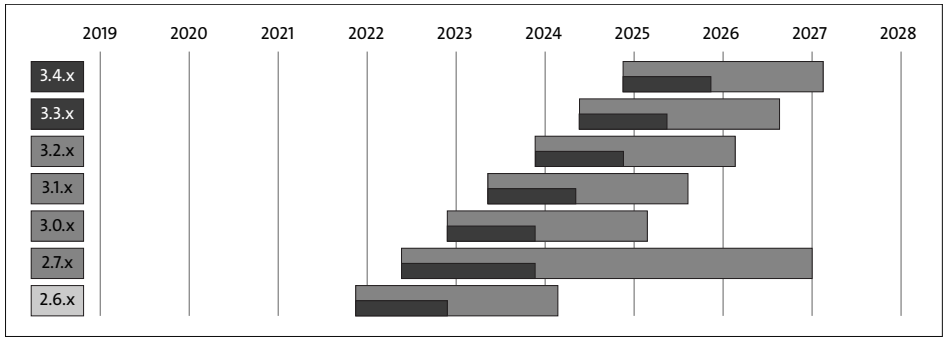


Abbildung 1.3 Support-Zeiträume der verschiedenen Spring-Boot-Versionen

Die längeren Balken zeigen die kommerzielle Unterstützung; die kürzeren inneren Balken davor zeigen die Bugfixes und Updates der Open-Source-Versionen.

Alternativen zu Spring

Wenn man ehrlich ist, dann muss man ganz klar sagen, dass das Spring Framework nur deswegen entstanden ist, weil die J2EE damals so kompliziert und unbequem war. Das ist jetzt allerdings schon ziemlich lange her. Daher stellt sich die Frage, wie es heute aussieht: Gelten die Aussagen für die aktuelle Java Enterprise Edition noch? Gibt es Alternativen? Ist Spring immer noch modern?

Aus J2EE wurde später die *Java Enterprise Edition*, kurz *Java EE*, und nach etwas Herumgezanke (Oracle wollte auf das Namensrecht »Java« nicht verzichten) dann die *Jakarta EE*. Die Jakarta EE ist eine umfangreiche Weiterentwicklung und hat mit der ursprünglichen J2EE wenig gemeinsam. Heute ist die Jakarta EE durchaus ein gutes Modell zum Entwickeln von großen Enterprise-Java-Anwendungen.

Neben den genannten Alternativen zu Spring gibt es noch eine ganze Reihe von weiteren Enterprise-Frameworks auf dem Markt. Eine kurze Übersicht:

- *Dropwizard*, *https://dropwizard.io* (v1, Juli 2016): Mike Youngstrom selbst nimmt Dropwizard zum Vorbild für Spring Boot.
- *Micronaut*, *https://micronaut.io* (v1, Mai 2018)
- *Helidon*, *https://helidon.io* (v1, Februar 2019)
- *Quarkus*, *https://quarkus.io* (v1, März 2019)

Fassen wir zusammen: Es gibt prinzipiell eine ganze Reihe von spannenden Alternativen. Auch die Jakarta EE erfüllt heute alle wichtigen Anforderungen an ein Enterprise-Framework, sodass man damit Geschäftsanwendungen schreiben kann. Warum also dennoch Spring?

Ein Vergleich zwischen Jakarta EE und Spring ist eigentlich nicht möglich, schon eher der Vergleich zwischen einem Jakarta-EE-Application-Server und einer Spring-Anwendung. Wir dürfen nicht vergessen, dass die Jakarta EE im Grunde nur ein Bündel von rund 20 Teilspezifikationen ist, die ein Application-Server vollständig implementiert. Zu den Teilen zählen:

- Jakarta Bean Validation,
- Jakarta Concurrency,
- Jakarta Enterprise Beans,
- Jakarta Enterprise Web Services,
- Jakarta Expression Language,
- Jakarta Mail,
- Jakarta Persistence,
- Jakarta RESTful Web Services,
- Jakarta Faces,
- Jakarta Server Pages,
- Jakarta Servlet

und noch viele weitere. Wir haben also eine ganze Reihe an Teilspezifikationen. Spring setzt sehr viele dieser Teile auch ein, zum Beispiel die *Jakarta Bean Validation*. Spring nutzt diesen guten Standard und die Referenzimplementierung – warum sollten die Spring-Macher etwas komplett Neues entwickeln? Dafür gibt es überhaupt keinen Grund! Das Gleiche bei der *Mail-API* – es gibt keinen Grund, die Mail zu verschmähen, denn das Senden und Empfangen von E-Mails ist nicht gerade trivial. Das heißt, im Spring-Universum entwickelt man nicht einfach neue Standards, wenn das Problem schon gelöst ist. Ein weiteres Beispiel ist die *Jakarta Persistence*. Objektrelationales Mapping mit der Jakarta Persistence ist schon komplex genug, und es gibt ausgereifte Implementierungen.

Deswegen ist der Vergleich schwierig, denn wenn wir uns das Spring Framework anschauen, ist es eher eine Art Integrations-Framework, das verschiedene Technologien umfasst. Wenn man einen Unterschied zwischen Jakarta EE und Spring sucht, dann ist es der *Applikationsserver*. Dieser wird gestartet, läuft den lieben Tag vor sich hin, und immer wieder werden Anwendungen hinzugefügt, entfernt und ausgewechselt. Ein Applikationsserver enthält einen Webserver, der auch immer mitläuft. In einer Spring-Anwendung ist üblicherweise ein Embedded Webserver enthalten, das heißt,

Spring-Anwendungen werden in der Regel nicht in einem Container deployt, sondern enthalten alle Serverbestandteile.

Jakarta EE und Application-Server sind nicht wirklich eine Konkurrenz für das Spring Framework, aber es gibt durchaus andere Entwicklungen, wo alternative Frameworks ein bisschen die Nase vorn haben. Moderne Frameworks, wie zum Beispiel *Quarkus*, sind für Microservices optimiert und nutzen die sogenannte *native compilation*. Damit gibt es keine traditionelle Java Virtual Machine, die gestartet wird, zur Laufzeit den Bytecode einliest und diesen zur Laufzeit in mehreren Iterationen in Maschinencode übersetzt. Die Übersetzung wird vor der Ausführung gemacht und nennt sich daher auch *ahead-of-time compilation* (kurz *AOT compilation*). Oracle bietet einen solchen Compiler an, die Technologie wird *GraalVM Native Image* genannt. Damit entsteht am Ende eine direkt ausführbare Datei, also z. B. unter Windows eine *.exe*-Datei, die wir mit dem Doppelklick einfach starten können. Native Java-Anwendungen haben eine radikal reduzierte Startzeit, und auch ihr Speicherverbrauch ist geringer. Spring hat lange gebraucht, die native Kompilation zu unterstützen; in dieser Zeit haben andere Lösungen wie Quarkus Fans gefunden. Doch seit dem Spring Framework 6 und Spring Boot 3 ist auch die native Kompilation eingezogen. Zudem sprechen andere Features für Spring.

Der große Vorteil des Spring Frameworks ist, dass es im Grunde alles integrieren kann, um es noch mal mit der Jakarta EE zu vergleichen. Während wir im Jakarta-Umfeld genau eine API für eine Aufgabe haben (zum Beispiel E-Mail oder OR-Mapping) und es dafür verschiedene Implementierungen gibt, ist es bei Spring genau andersherum: Es gibt verschiedene APIs und damit auch verschiedene Implementierungen. Das lässt sich gut am Beispiel eines OR-Mappers ablesen. Die Jakarta Persistence definiert eine API für das objektrelationale Mapping, aber es gibt durchaus alternative Ansätze und Erweiterungen, etwa mit *JOOQ*, *Querydsl* oder *Spring-Data-JDBC*; es gibt mehr als nur einen objektrelationalen Mapper.

Während auf der einen Seite eine Spring-Anwendung alle Jakarta-Technologien einsetzen kann, geht Spring viel weiter. Sind zum Beispiel bei Jakarta EE lediglich Jakarta Faces und Jakarta Server Pages (JSP) standardisiert, lassen sich im Spring-Universum beliebige Template-Engines einsetzen. Dazu gehören *Thymeleaf*, *Free Marker*, *Velocity* oder natürlich auch *Jakarta Faces*. Einer der zentralen Ansätze von Spring ist, nicht invasiv zu sein, also nicht Spring-spezifische Datentypen mit Geschäftslogik zu vermischen. Im Spring-Umfeld haben wir selten Verbindung zu einer tatsächlichen Technologie, sondern diese wird häufig durch eine Abstraktion von Spring gekapselt.

Spring bietet auch Lösungen, für die es im Jakarta-EE-Standard gar nichts gibt. Schaut man sich die Projekte auf der Spring-Seite an, tauchen dort sehr viele Lösungen im Cloud-Umfeld auf, mit denen man leicht Microservices entwickeln und verwalten kann.

Spring ist heute immer noch eine der besten Möglichkeiten, Enterprise-Anwendungen zu realisieren, und das zeigen auch unterschiedliche Erhebungen. JRebel macht regelmäßig eine Umfrage, welche Technologien in der Softwareentwicklung eingesetzt werden (siehe Abbildung 1.4); Spring Boot ist ganz vorne mit dabei.[4]

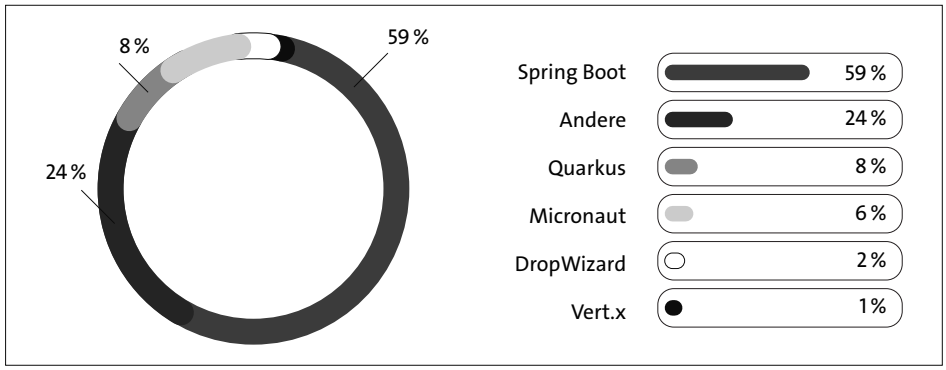


Abbildung 1.4 Verbreitung von Spring Boot nach einer Umfrage von JRebel

Die Kombination aus Spring Boot und dem Tomcat-Servlet-Container ist das, was die meisten Teams heute für moderne Java-Enterprise-Anwendungen einsetzen. Natürlich werden auch andere Java-Enterprise-Frameworks eingesetzt, der Anteil ist allerdings im Moment nicht hoch.

Bemerkung

Die Entwicklung ist schon irgendwie merkwürdig: Erst liefen Java-Programme in einem Container, dem Enterprise-Application-Server, dann wurden die Container geächtet, und Anwendungen enthielten alle Komponenten. Betrachtet man die aktuelle Situation, gibt es wieder einen Trend hin zu einer Art Container, nur liegen die Anwendungen nach dem Serverless-Modell typischerweise in der Cloud.

1.1.2 Ein Spring-Boot-Projekt aufsetzen

Es gibt mehrere Möglichkeiten zum Aufbau neuer Spring-Boot-Projekte.

Letztendlich ist ein Spring-Boot-Projekt nichts anderes als ein reguläres Java-Projekt mit ein paar Klassen im Klassenpfad. Solche Abhängigkeiten werden heute nicht mehr von Hand in den Klassenpfad aufgenommen, sondern wir nutzen zur Verwaltung normalerweise Werkzeuge wie *Maven* oder *Gradle*. Obwohl es nicht schwierig ist, ein solches Projekt von Hand aufzubauen, empfiehlt sich eine der beiden folgenden Möglichkeiten.

4 *https://www.jrebel.com/system/files/jrebel-2023-java-developer-productivity-report.pdf*

Spring Initializr unter https://start.spring.io/

Mit dem *Initializr* gibt es einen webbasierten Dienst, den wir direkt im Browser nutzen können. In den meisten modernen Entwicklungsumgebungen ist er auch über Dialoge direkt integriert.

Schauen wir uns den Initializr unter *https://start.spring.io* an (siehe Abbildung 1.5).

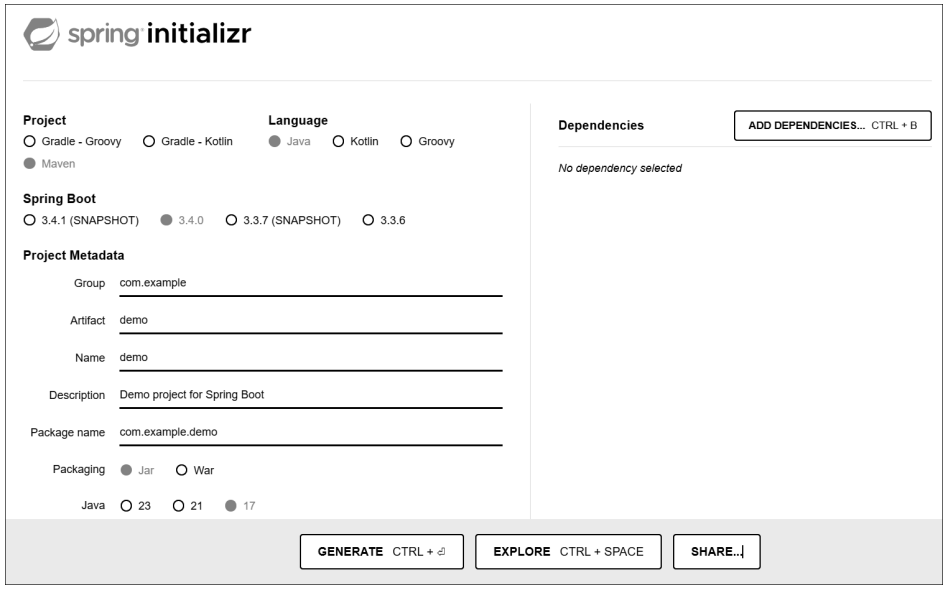


Abbildung 1.5 Die Website »https://start.spring.io«

Es lässt sich einstellen, ob ein Maven- oder ein Gradle-Projekt gewünscht ist; wir wählen MAVEN PROJECT und nicht Gradle. Als Nächstes lässt sich die Programmiersprache bestimmen. Neben Java werden auch Kotlin und Groovy unterstützt. Mittlerweile zeigt die Referenzdokumentation von Spring die Programmierbeispiele je nach Wunsch in Java und Kotlin an; das macht deutlich, wie wichtig Kotlin mittlerweile im Backend ist. Java ist voreingestellt, und wir belassen das auch so.

Als Nächstes lässt sich die Versionsnummer von Spring Boot auswählen – da es im Monatsrhythmus Updates gibt, dürfte sie bei der Leserschaft aktueller sein. Anschließend lassen sich die üblichen Projekt-Metadaten eintragen, was typisch ist für Maven. Das hat mit dem Spring-Projekt erst einmal nichts zu tun.

Wir wollen ein Projekt aufbauen und die Eingabefelder füllen. Im Textfeld bei der GROUP-ID tragen wir »com.tutego« ein und im Feld bei der ARTIFACT-ID »date4u«. Der Name ist ebenfalls als »date4u« vorinitialisiert, was in Ordnung ist. Die Eintragung unter DESCRIPTION »Demo Project for Spring Boot« ist auch in Ordnung. Der Paketname, der automatisch aus der Group-ID und der Artifact-ID generiert wird, passt so weit.

Als Nächstes sehen wir das PACKAGING: Soll die Anwendung als JAR (Java-Archiv) verpackt werden (das wäre der Standard) oder als Webanwendung (WAR) generiert werden? Webanwendungen können später in einem Servlet-Container wie Tomcat deployt werden. Zusätzlich ist die Java-Version anzugeben; Spring Boot 3 benötigt mindestens Java 17, Java 21 bietet sich als letztes LTS-Release allerdings in der Praxis an.

Auf der rechten Seite lassen sich unter ADD DEPENDENCIES Abhängigkeiten hinzufügen. Dabei handelt es um speziell unterstützte und ausgewählte Projekte aus dem Spring-Universum und nicht um x-beliebige Java-Projekte. Zum Beispiel könnte man sagen: Ich will »irgendwas mit Webentwicklung« oder »irgendwas mit einem Werkzeug wie Lombok« machen, sodass es über einen Compiler-Hack automatisch Setter und Getter gibt. Die Angaben werden später automatisch über den Initializr in die POM-Datei eingetragen (oder im Fall von Gradle in eine Gradle-Datei). Diese Dependencies lassen sich mit dem Minuszeichen wieder aus der Liste löschen.

Mit EXPLORE gibt es eine Vorschau in das Projekt, so wie es generiert werden würde (siehe Abbildung 1.6).

Abbildung 1.6 Ein erster Blick auf das Projekt

Faltet man den Strukturbaum bei *src* auf, sieht man das typische *Standard Directory Layout* von Maven: *src/main/java*, *src/main/resources* und *src/test/java*, aber standardmäßig kein *src/test/resources*. Das müsste später von Hand ergänzt werden.

Zusätzlich lassen sich weitere Dateien ablesen: eine Hauptklasse mit der `main(…)`-Methode, eine leere *application.properties*-Datei zur Konfiguration, eine automatisch generierte Testklasse – und wir haben natürlich, wie schon gesehen, die POM-Datei.

Ebenfalls gibt es eine *HELP.md*-Datei; sie enthält eine kompakte Dokumentation über die Dependencies. Zusätzlich wird eine Datei *.gitignore* generiert, weil die Entwickelnden heute üblicherweise mit Git als Versionsverwaltungsprogramm arbeiten.

Auf der Webseite mit dem Dateibaum befindet sich die Schaltfläche DOWNLOAD (siehe Abbildung 1.6). Ein Klick darauf gibt uns ein ZIP-Archiv. Auf der Hauptseite heißt die Schaltfläche GENERATE (siehe Abbildung 1.5). Mit anderen Worten: Der Spring Initializr ist im Wesentlichen ein spezieller Webservice, der ein Quellcode-Archiv mit dem entsprechenden Projekt liefert.

Hat man das ZIP-Archiv heruntergeladen und ausgepackt, ist das Ergebnis ein reguläres Maven-Projekt, das in jeder modernen Entwicklungsumgebung geöffnet werden kann. (Wer zum Beispiel die freie *Community Edition* von IntelliJ verwendet, muss das an dieser Stelle tatsächlich als Maven-Projekt tun, weil die freie Community Edition keine Unterstützung für Spring-Boot-Projekte hat.)

Auch auf der Kommandozeile kann das Projekt jetzt gebaut und ausgeführt werden. Der Spring Initializr erstellt dabei mit dem *Maven Wrapper* eine lokale, projektspezifische Maven-Installation. Die Funktionalität lässt sich mit dem Befehl `mvnw -version` (oder kurz `mvnw -v`) überprüfen. Beim ersten Ausführen kann es etwas länger dauern, da erforderliche Ressourcen aus dem Internet heruntergeladen werden.

```
$ mvnw -version
Apache Maven 3.9.9 (8e8579a9e76f7d015ee5ec7bfcdc97d260186937)
Maven home: …\.m2\wrapper\dists\apache-maven-3.9.9\
977a63e90f436cd6ade95b4c0e10c20c
Java version: 21.0.1, vendor: Oracle Corporation, ↩
runtime: C:\Program Files\Java\latest
Default locale: de_DE, platform encoding: UTF-8
OS name: "windows 11", version: "10.0", arch: "amd64", family: "windows"
```

Tipp

Maven braucht eine gesetzte Umgebungsvariable `JAVA_HOME`. Wenn das beschriebene Vorgehen zu einem Problem führt, liegt dies vermutlich an der nicht gesetzten Umgebungsvariablen. Maven zeigt das allerdings in einer Fehlermeldung auch an.

In der POM-Datei ist bereits ein Plugin eingetragen, mit dem sich das Spring-Programm auch ausführen lässt. Nach Eingabe des Befehls `mvnw spring-boot:run` werden alle Phasen abgearbeitet. Das heißt, das Projekt wird erst kompiliert und dann ausgeführt.

```
$ mvnw spring-boot:run
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------< com.tutego:date4u >---------------------
[INFO] Building date4u 0.0.1-SNAPSHOT
[INFO]   from pom.xml
[INFO] --------------------------------[ jar ]---------------------------
[INFO]
[INFO] >>> spring-boot:3.4.0:run (default-cli) > test-compile @ date4u >>>
[INFO]
[INFO] --- resources:3.3.1:resources (default-resources) @ date4u ---
[INFO] Copying 1 resource from src\main\resources to target\classes
[INFO] Copying 0 resource from src\main\resources to target\classes
[INFO]
[INFO] --- compiler:3.13.0:compile (default-compile) @ date4u ---
[INFO] Recompiling the module because of changed source code.
[INFO] Compiling 1 source file with javac [debug parameters release 17] to
       target\classes
[INFO]
[INFO] --- resources:3.3.1:testResources (default-testResources) @ date4u ---
[INFO] skip non existing resourceDirectory …\date4u\src\test\resources
[INFO]
[INFO] --- compiler:3.13.0:testCompile (default-testCompile) @ date4u ---
[INFO] Recompiling the module because of changed dependency.
[INFO] Compiling 1 source file with javac [debug parameters release 17] to
       target\test-classes
[INFO]
[INFO] <<< spring-boot:3.4.0:run (default-cli) < test-compile @ date4u <<<
[INFO]
[INFO]
[INFO] --- spring-boot:3.4.0:run (default-cli) @ date4u ---
[INFO] Attaching agents: []


  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/

 :: Spring Boot ::                (v3.4.0)

2024-11-28T07:13:42.397+01:00  INFO 20848 --- [date4u] [           main]
```

```
com.tutego.date4u.Date4uApplication      : Starting Date4uApplication using
Java 21.0.1 with PID 20848 (…\date4u\target\classes started by Christian in
…\date4u)
2024-11-28T07:13:42.401+01:00  INFO 20848 --- [date4u] [           main]
com.tutego.date4u.Date4uApplication      : No active profile set, falling back
to 1 default profile: "default"
2024-11-28T07:13:42.838+01:00  INFO 20848 --- [date4u] [           main]
com.tutego.date4u.Date4uApplication      : Started Date4uApplication in 0.786
seconds (process running for 1.076)
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  4.005 s
[INFO] Finished at: 2024-11-28T07:13:42+01:00
[INFO] ------------------------------------------------------------------------
```

In der Konsole sehen wir das Ergebnis: Die Spring-Anwendung fährt hoch und wieder herunter. Weil die Anwendung keinen Serverteil enthält, gibt es für die Spring-Anwendung nichts Weiteres zu tun, und sie beendet sich automatisch.

Der Maven-Wrapper ist komfortabel und bietet zwei Vorteile: Die Empfänger des Projektes können das Projekt direkt ausführen, ohne dass sie eine eigene lokale Maven-Installation haben müssen. Und der zweite Vorteil ist der, dass diese lokale Version nicht mit der anderen globalen Maven-Installation kollidiert. Das kann ein Vorteil sein, da es wichtig ist, einen sogenannten *reliable build* zu produzieren. Mit anderen Worten: So, wie die Software heute gebaut wird, soll sie auch in 10 Jahren gebaut werden können. Wenn eine Software Maven selbst bereitstellen kann, ist die Build-Umgebung ein geschlossenes System und funktioniert ohne besondere äußere Einstellungen. Der Maven-Wrapper lädt automatisch alle erforderlichen Komponenten vom Maven Central Repository herunter. Ein installiertes JDK ist das einzige notwendige Tool. Weitere Informationen finden sich auf der Webseite *https://maven.apache.org/wrapper/index.html*.

1.1.3 Spring-Projekte in Entwicklungsumgebungen aufbauen

Sich über den Initializr oder die *Spring Boot CLI* das Projekt generieren zu lassen, ist in Ordnung. Allerdings helfen Entwicklungsumgebungen dabei, schnell und unkompliziert Spring-Projekte aufzubauen. Zusätzliches Tooling hilft insbesondere bei der Auflistung der Komponenten (zum Beispiel der HTTP-Endpunkte), bei gewissen Gültigkeitsprüfungen von Methodennamen oder bei der Abfrage von Datenbanken.

Für alle modernen Entwicklungsumgebungen gibt es entsprechende Plugins:

- **Eclipse**: Es gibt für die *Eclipse IDE*, die »offizielle IDE« von VMware, die *Spring Tool Suite* (*https://spring.io/tools*). Dabei wird eine Eclipse-Installation um entsprechende Plugins erweitert. Das bietet auch eine komfortable Möglichkeit zum Anlegen von Projekten.
- **IntelliJ-IDE**: Nur die Ultimate-Edition bringt eine entsprechende Spring-Unterstützung mit. Wer die Community Edition einsetzt, wird leider keine Unterstützung für Spring-Projekte vorfinden. Das ist ein bisschen bedauerlich, aber letztendlich ist es auch das, was die freie Version von der teuren Version unterscheidet. Es gibt auf dem Markt einige Plugins, die die freie Community Edition ein bisschen erweitern. Sie sind allerdings proprietär und wirklich nur die letzte Option, wenn man die *Ultimate Edition* nicht nutzen kann. In diesem Buch gibt es vereinzelt Screenshots der IntelliJ IDE Ultimate Edition.
- **Visual Studio Code**: VSC ist ein Editor, der immer populärer wird. Von VMware gibt es ein Plugin, das sich installieren lässt. Dann kann man innerhalb von Visual Studio Code sehr einfach und komfortabel neue Spring-Boot-Projekte anlegen und laufen lassen.

Die nächsten Abschnitte zeigen, wie die unterschiedlichen IDEs helfen, ein Spring-Boot-Projekt ohne weitere Dependencies aufzubauen. Wer das schon kennt, kann diese Abschnitte überspringen und eigenständig ein Projekt mit den folgenden Maven-Koordinaten aufbauen:

- Group-ID: `com.tutego`
- Artifact-ID: `date4u`
- Paket: `com.tutego.date4u`

IntelliJ Ultimate

Grundsätzlich kann man mit allen großen Entwicklungsumgebungen Spring-Boot-Projekte realisieren, doch die Unterstützung von IntelliJ ist am besten.

Wenn wir ein neues Projekt aufbauen wollen, gehen wir im Menü auf FILE und klicken dann auf NEW und PROJECT. Von dort können wir diverse Projekttypen anlegen. Auf der linken Seite sind wir bei SPRING INITIALIZER richtig (siehe Abbildung 1.7).

Der Dialog ist stark mit der Webseite vom Initializr verwandt. Es sind gültige Daten eingetragen, sodass man mit NEXT einfach weitermachen könnte, doch wir wollen einige Werte anpassen.

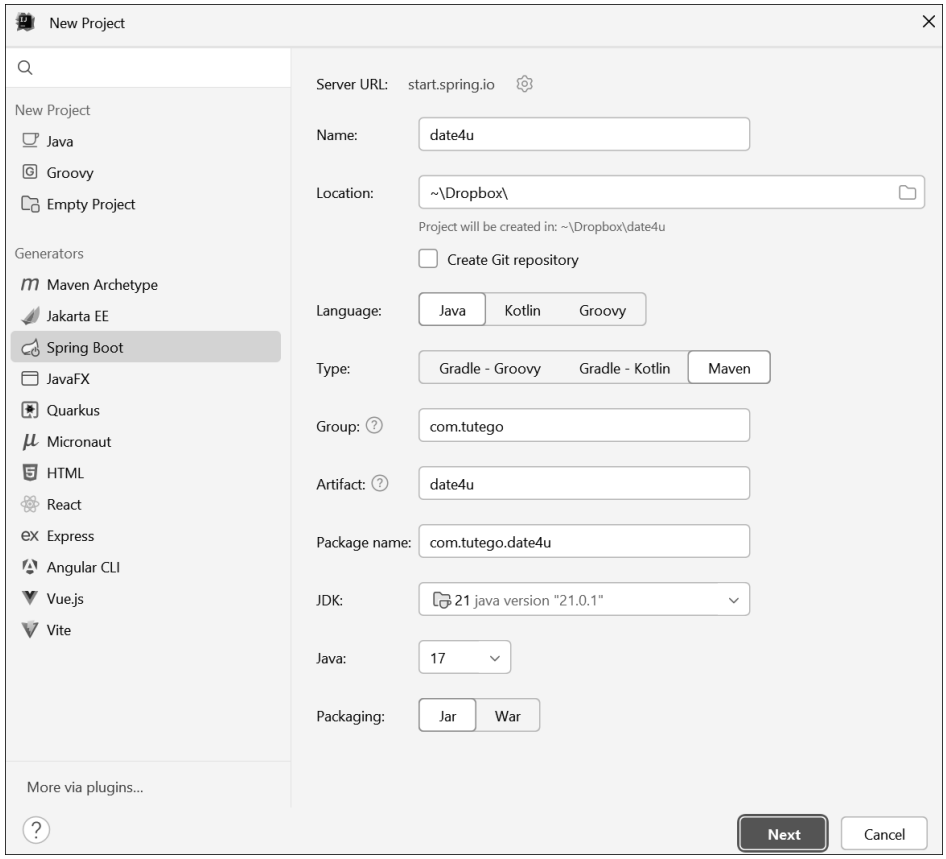


Abbildung 1.7 Dialog zum Erzeugen eines neuen Projekts

Wir haben die Möglichkeit, zwischen den Programmiersprachen Java, Kotlin und Groovy auszuwählen. Wir bleiben bei Java. Auch wählen wir MAVEN aus. Als NAME setzen wir »date4u« ein. Der Zielordner (LOCATION) lässt sich anpassen. Bei der GROUP sage ich »com.tutego«, bei ARTIFACT trage ich »date4u« ein. Der PACKAGE NAME soll »com.tutego.date4u« sein. IntelliJ hat Java 17 hat als JDK erkannt. Bei der JAVA-Version wählen wir mindestens 17, vorzugsweise Java 21. Die Version von Spring Boot ist in dem Dialog nicht zu finden, aber auf der nächsten Dialogseite lässt sie sich auswählen.

Ein Klick auf den NEXT-Button bringt uns zu den Abhängigkeiten im nächsten Dialog (siehe Abbildung 1.8).

Bei den *Dependencies* handelt es sich um Abhängigkeiten aus dem Spring Universum, die Spring »kennt«. Für unser erstes Projekt werden wir das noch nicht benötigen, und wir werden später auch über die POM-Datei unsere Dependencies hinzunehmen.

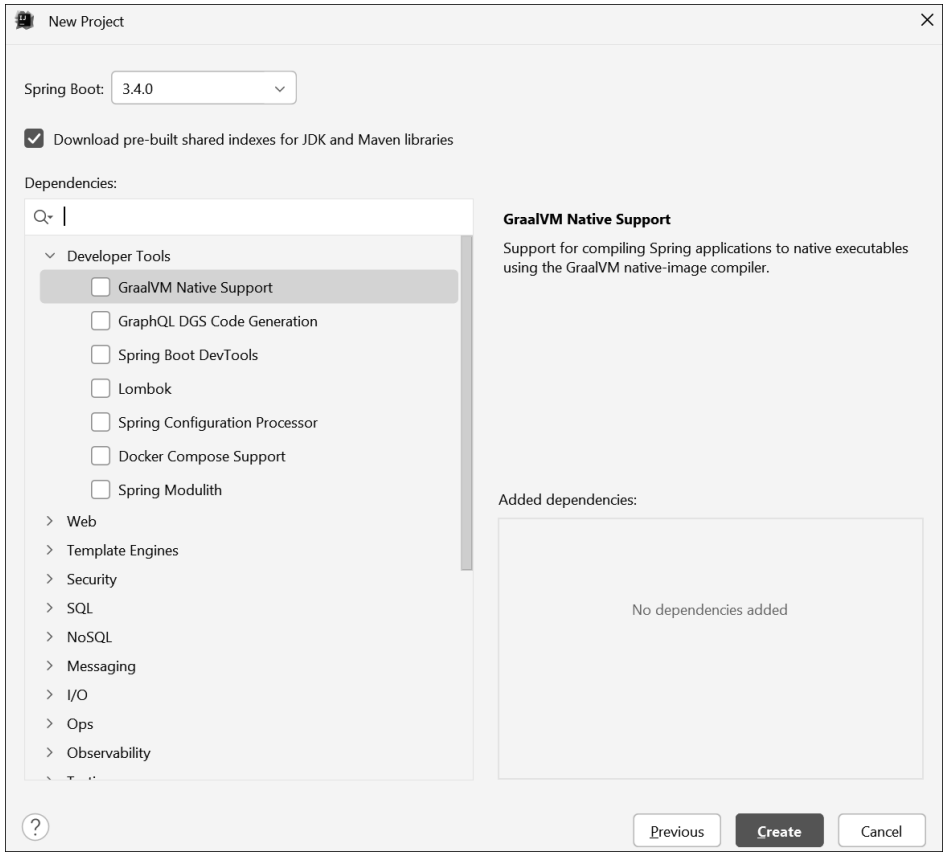


Abbildung 1.8 Dependencies eintragen

Ein Klick auf CREATE schließt den Prozess ab. Nun entsteht ein Maven-Projekt, das IntelliJ auch direkt startet. Wir können zum Hauptprogramm unter `Date4uApplication` gehen und es in der IDE starten. In unter 2 Sekunden wird die Anwendung gestartet und endet dann automatisch wieder. Das heißt, nach dem Aufruf der `run(…)`-Methode, die den Spring-Container startet, wird der Spring-Container automatisch beendet.

IntelliJ Community Edition

Die *IntelliJ Community Edition* hat von Haus aus keine Framework-Unterstützung, weder für Spring noch für Jakarta EE. Eine Option ist, sich von der Initializr-Webseite ein ZIP-Archiv herunterzuladen, es auszupacken und das Verzeichnis als Maven-Projekt zu importieren.

Es gibt allerdings Plugins, die uns helfen und wie die Ultimate Edition einen Dialog für neue Projekte nachbilden. Diese Plugins sind unterschiedlich gut bewertet und un-

terschiedlich vollständig. Eine Empfehlung gibt es nicht. Die freien Erweiterungen ersetzen auf keinen Fall die kommerzielle »große« Ultimate Edition, die viele Konfigurationsfehler erkennt oder SQL-Abfragen direkt im Java-Code ausführen kann.

STS Spring Tool Suite

VMware hat für die Eclipse-IDE ein eigenes Plugin erschaffen, die *Spring Tool Suite*, kurz *STS* (siehe Abbildung 1.9). Wenn man die STS nutzen möchte, gibt es zwei Möglichkeiten:

1. Ein installiertes Standard-Eclipse wird nachträglich über den Marketplace um STS-Plugins erweitert.
2. Man wählt eine Installation aus, die auf der aktuellen Version der Eclipse-IDE basiert und die Plugins schon enthält. Regelmäßig gibt es dann, wenn es eine neue Version der Eclipse-IDE gibt, auch aktualisierte Versionen der STS.

Die zweite Möglichkeit ist meistens unproblematischer, weil sie Plugin-Konflikte vermeidet.

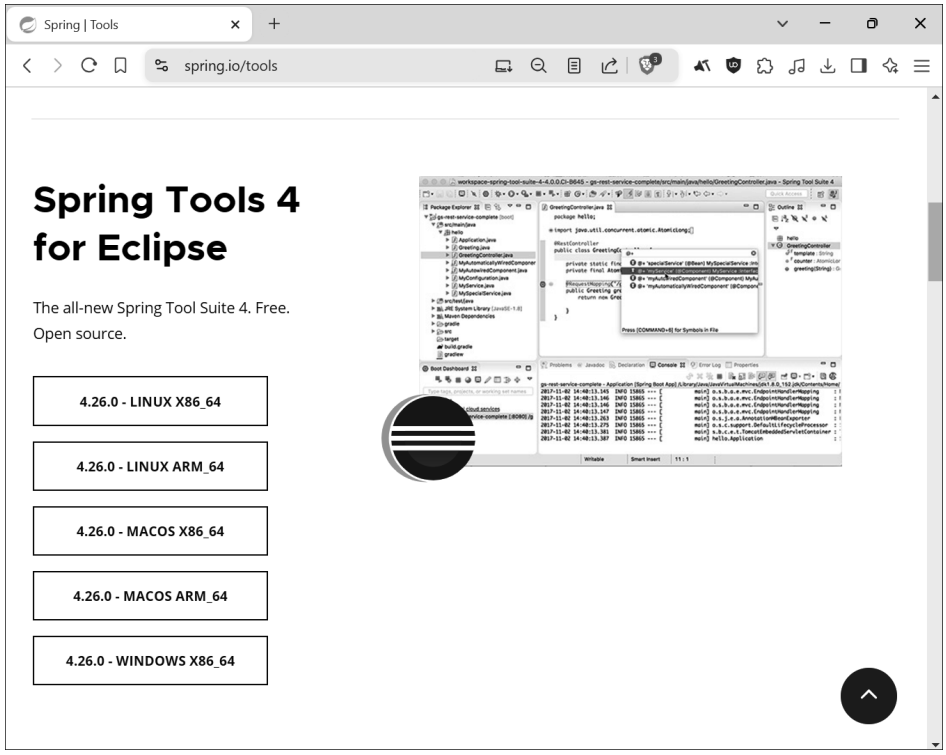


Abbildung 1.9 Die Eclipse-STS-Startseite »https://spring.io/tools« mit Downloadlinks

Die Datei zum Herunterladen ist ein ZIP. Nach dem Auspacken gibt es unter Windows eine EXE-Datei, und Eclipse lässt sich starten. Nach kurzer Wartezeit erscheint der *Tools Launcher*. Bei Eclipse gibt es immer einen *Workspace*, einen Ort für zentrale Konfigurationen, an dem auch Projekte hinterlegt sind. Nach der Dialogbestätigung fährt Eclipse hoch, und auf der linken Seite taucht der Punkt CREATE NEW SPRING STARTER PROJECT auf; das liegt an der besonderen Konfiguration der Eclipse IDE. Wenn wir diesen Eintrag nicht sehen sollten, können wir jederzeit unter FILE • NEW • SPRING STARTER PROJECT auf ein Spring-Projekt aufbauen. Der Dialog erinnert an das, was wir auf der Initializr-Webseite gesehen haben. Diese Webseite wird tatsächlich auch als Endpunkt eingesetzt, denn sie generiert dann das ZIP, das Eclipse auspackt und als Basis für das Projekt verwendet.

Ich nenne das Projekt »date4u«. Die Default-Location ist standardmäßig der Workspace-Ordner. Bei TYPE wählen wir MAVEN aus. Alle Java-Versionen ab Java 17 sind erlaubt; Java 21 ist als letztes LTS-Release eine gute Wahl. Als GROUP-ID trage ich »com.tutego« ein. Die ARTIFACT-ID ist »date4u«. Die Versionsnummer passt, die DESCRIPTION auch. Bei dem PACKAGE gebe ich ein: »com.tutego.date4u«. Diese Einstellungen haben nichts mit Spring zu tun, das sind die typischen Angaben, die wir auch bei jedem Maven- oder Gradle-Projekt vornehmen müssten.

Ein Klick auf NEXT bringt uns zum nächsten Dialog. An dieser Stelle wird es interessant, denn wir können die Dependencies auswählen, die typisch für Spring-Projekte sind. Wenn wir zum Beispiel etwas mit einem Webserver machen wollten, dann könnten wir im Suchfeld den Begriff WEB eingeben; das schränkt die Suche ein. Für unsere ersten Projekte brauchen wir jedoch keine Dependencies, deswegen lässt sich dieser Teil überspringen. Klicken wir auf FINISH, wird das ZIP-Archiv bezogen und das Projekt aufgebaut.

Navigiert man zur Hauptklasse `Date4uApplication`, lässt diese sich ausführen. Im Kontextmenü gibt es dazu unter RUN AS zwei Möglichkeiten: JAVA APPLICATION und SPRING BOOT APP – letztere Option führt zur farbigen Ausgabe.

Tipp

Ein Problem, das man allerdings ernsthaft berücksichtigen sollte, besteht darin, dass Eclipse (zumindest in der aktuellen Version) große Schwierigkeiten hat, diese farbigen Ausgaben performant auf den Bildschirm zu bringen. Mit anderen Worten: Wenn man eine träge Ausgabe bemerkt und das Scrollen viel Zeit braucht, sollte man die Farbdarstellung abschalten. Dazu wird in der Konfiguration der Punkt ANSI CONSOLE OUTPUT deaktiviert.

1.2 Dependencies und Starter eines Spring-Boot-Projekts

An dieser Stelle wollen wir uns die Dependencies eines regulären Spring-Boot-Projekts genauer anschauen.

1.2.1 POM mit Parent-POM

Blicken wir in die vom Initializr angelegte POM-Datei. Der Grundaufbau sieht so aus:

Listing 1.1 pom.xml

```
<?xml …?>
<project …>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    …
  </parent>

  <groupId>com.tutego</groupId>
  <artifactId>date4u</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <properties>
    <java.version>17</java.version>
  </properties>

  <dependencies>…</dependencies>

  <build>…</build>
</project>
```

Spring Boot nutzt eine Parent-POM

Der Initializr legt ein Maven-Projekt an, das standardmäßig eine Parent-POM referenziert. Außerdem tauchen die Angaben für die Maven-Koordinaten auf, also die Group-ID, Artifact-ID und Versionsnummer. Zudem finden wir Properties mit der Versionsnummer der gewünschten Java-Version. Anschließend folgt ein Block mit Dependencies, und zu guter Letzt wird ein Maven-Plugin referenziert.

Eine Parent-POM ist eine Art »Oberklasse« für Maven-Projekte. Damit können Informationen aus der Parent-POM auf das eigene Projekt übertragen werden. Spring verweist dabei auf `org.springframework.boot:spring-boot-starter-parent`:

Listing 1.2 pom.xml

```
<?xml …?>
<project …>
  …
  <parent>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-parent</artifactId>
   <version>3.4.0</version>
  </parent>
  …
</project>
```

Die Parent-POM definiert einige Properties. Dazu zählt etwa `java.version` für die Java-Versionsnummer, die Spring Boot 3 standardmäßig auf Java 17 setzt. Mit `java.version` werden auch `maven.compiler.source` und `maven.compiler.target` initialisiert. Standardmäßig werden auch UTF-8-Encodings gesetzt, was sowieso üblich ist. Es folgen Lizenzen und ein paar Developer-Informationen.

Parent-POM hat auch eine Parent-POM mit BOM

Interessant ist, dass der *Spring Boot Starter Parent* selbst wiederum eine Parent-POM referenziert: `org.springframework.boot:spring-boot-dependencies`. Diese POM definiert eine sogenannte *BOM* (*Bill of Material*), was übersetzt »Stückliste« bedeutet. Darin werden in erster Linie Versionsnummern von Abhängigkeiten deklariert.

Bei der Deklaration der Dependencies verwendet das Spring-Team einen zweistufigen Ansatz. Im ersten Schritt werden zahlreiche Properties definiert. Hier ist ein Beispiel für die POM-Datei aus `org.springframework.boot:spring-boot-dependencies`[5]:

```
<properties>
  <activemq.version>6.1.4</activemq.version>
  <angus-mail.version>2.0.3</angus-mail.version>
  <artemis.version>2.37.0</artemis.version>
  <aspectj.version>1.9.22.1</aspectj.version>
  <assertj.version>3.26.3</assertj.version>
  …
</properties>
```

Die Schlüssel werden in der Regel aus der Artifact-ID und der Versionsnummer zusammengesetzt. Das bestimmt ganz viele Versionsnummern von diversen Projekten, die im Spring-Umfeld häufig eingesetzt werden. Es ist nicht so, dass dort alle Java-Pro-

5 *https://repo1.maven.org/maven2/org/springframework/boot/spring-boot-dependencies/3.4.0/spring-boot-dependencies-3.4.0.pom*

jekte auf der Welt vorkommen (das wäre wenig sinnvoll), aber es erscheinen zumindest die Projekte, die im Spring-Universum eine wichtige Rolle spielen.

Das Spring-Team sorgt dafür, dass diese Versionsnummern der referenzierten Projekte perfekt zusammenpassen. Wer möchte das schon selbst übernehmen und prüfen, dass zum Beispiel die Logging-Bibliothek mit der Versionsnummer 1.2.3 perfekt mit dem Webserver der Version 3.4.5 zusammenpasst? Einfach alle Versionsnummern immer auf die letzte Version hochzusetzen, funktioniert nicht immer. Wenn man auf der einen Seite für eine Bibliothek die Versionsnummer hochnimmt, kann es passieren, dass diese höhere Versionsnummer für eine andere Bibliothek zum Problem wird. Das heißt, es gibt ein feines Spiel von Versionen, die alle perfekt aufeinander abgestimmt sein müssen, sonst könnte es vielleicht beim Betrieb mit irgendwelchen veralteten Java-Archiven Probleme geben, weil Variablen, Methoden oder Typen fehlen.

Nach dieser Deklaration der Variablen kommt später in der POM ein Dependency-Management-Block:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.apache.activemq</groupId>
      <artifactId>activemq-amqp</artifactId>
      <version>${activemq.version}</version>
    </dependency>
    …
</dependencyManagement>
```

Dependency-Management bedeutet, dass die genannten Dependencies nicht wirklich mitgenommen, sondern lediglich mit ihrer Versionsnummer deklariert werden. Beim Festsetzen der Versionen greift der `<dependency>`-Block auf die vorher deklarierten Properties zurück.

Wenn wir später eine Dependency brauchen, werden wir ausschließlich einen Dependency-Block mit einer Group-ID und Artifact-ID schreiben, allerdings keine Versionsnummern nennen müssen. In unserer POM-Datei kommen daher nicht viele Versionsnummern vor. Natürlich bleiben weiterhin die Versionen der referenzierten Java-Projekte, die nicht unter dem Mantel von Spring Boot stehen, allerdings ist keine Versionsnummer von Projekten nötig, die etwas mit Spring zu tun haben. Für diese Projekte werden die Versionsnummern automatisch passend gesetzt.

Das hat aber auch zur Folge, dass wir nicht vergessen dürfen, unsere Spring-Boot-Version regelmäßig anzupassen, denn nur so bekommen wir auch die Version der referenzierten Projekte neu angepasst. Das ist ein wichtiges Feature, und wenn man sich die Release-Notes der entsprechenden Spring-Boot-Versionen unter *https://git-*

hub.com/spring-projects/spring-boot/releases anschaut, dann tauchen die Dependency-Upgrades immer auf (siehe Abbildung 1.10).

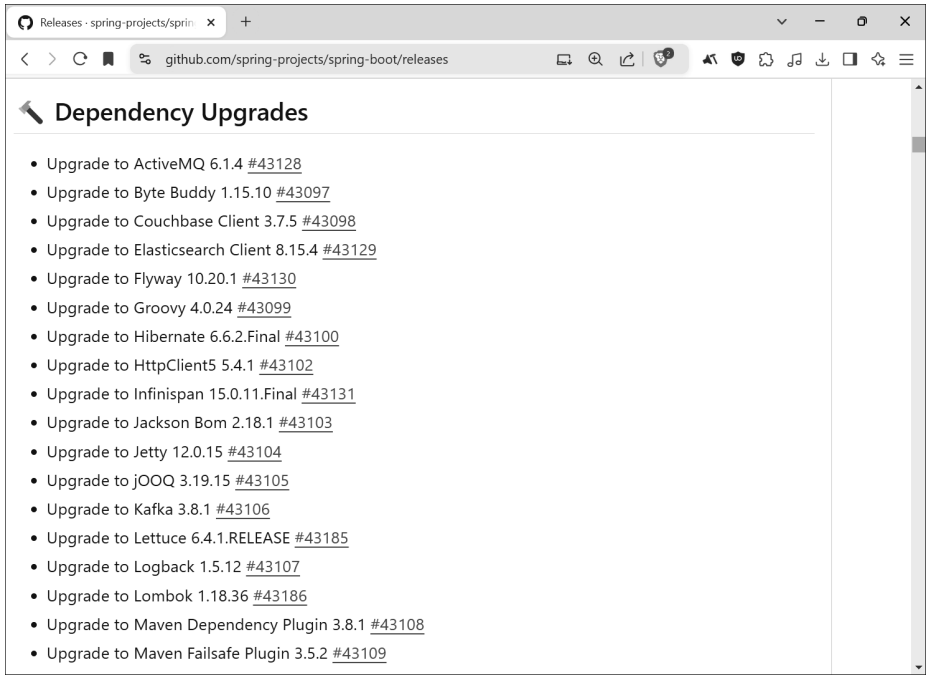


Abbildung 1.10 Dependency-Updates der Spring-Boot-Versionen in den Release-Notes

Es ist möglich, die Versionsnummern einzelner Projekte zu korrigieren. Dafür lässt sich im Properties-Block unserer POM-Datei die vordefinierte Variable überschreiben. Nehmen wir als Beispiel H2:

```
<properties>
  <h2.version>1.4.200</h2.version>
</properties>
```

Der Block setzt die vordefinierte Property `h2.version` auf die gewünschte Version. Wenn anschließend eine Dependency auf H2 in die POM kommt, nimmt Maven die gesetzte Versionsnummer. Ob sie höher oder niedriger ist, spielt keine Rolle.

1.2.2 Dependencies als Import

Spring-Boot-Dependencies müssen nicht zwingend über eine Parent-POM referenziert werden, sondern auch ein Import ist möglich. Das ist genau dann praktisch, wenn man eine eigene Parent-POM verwenden möchte. Es ist nicht ungewöhnlich, dass einige Unternehmen eine eigene Parent-POM definieren, in der zum Beispiel auch die Versionsnummer, Lizenzen oder sonstige Informationen hinterlegt sind.

Wenn man eine eigene Parent-POM haben möchte, kann man sich natürlich trotzdem das Projekt mit dem Initializr anlegen lassen. Doch dann muss man anschließend die Parent-POM durch die eigene ersetzen. Das Problem ist dann allerdings, dass die Dependencies fehlen, und die sind ja sehr wichtig. Aus diesem Grunde kann ich einen eigenen Dependency-Management-Block mit einer Import-Dependency setzen. Das sieht dann so aus:

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

Hinweis

Es gibt zwischen der Spring-Parent-POM und einem Spring-POM-Import einen wichtigen Unterschied. Der Spring-Boot-Starter-Parent »vererbt« den Kindern Properties, wie UTF-8-Encoding. Diese Eigenschaften müssen händisch ergänzt oder in einer eigenen Parent-POM definiert werden.

1.2.3 Milestones und das Snapshots-Repository

Auf dem Maven-Central-Server werden nur Releases von Bibliotheken hochgeladen und gehostet. Was wir dort nicht finden, sind Entwicklungsversionen, also Snapshots oder Milestones. Im Spring-Umfeld gibt es einen eigenen Server, den das Spring-Team betreibt. Dort finden wir auch entsprechende Milestones oder Snapshots, und das ist immer dann ganz nützlich, wenn man Dinge benutzen möchte, die gerade erst in Entwicklung sind. Es könnte auch sein, dass zum Beispiel irgendwelche Fehler gefixt wurden, und vielleicht macht es das nötig, auf eine tagesaktuelle Version zu setzen, auch wenn das kein Release ist.

Für Milestones und Snapshots ist in der POM Folgendes einzutragen:

```
<repositories>
  <repository>
    <id>spring-milestones</id>
    <name>Spring Milestones</name>
```

```
    <url>https://repo.spring.io/milestone</url>
    <snapshots><enabled>false</enabled></snapshots>
  </repository>
  <repository>
    <id>spring-snapshots</id>
    <name>Spring Snapshots</name>
    <url>https://repo.spring.io/snapshot</url>
    <releases><enabled>false</enabled></releases>
  </repository>
</repositories>
```

Tipp

https://start.spring.io/ erlaubt auch das Anlegen von Spring-Boot-Projekten mit einer Milestone- oder Snapshot-Version. Wählt man etwa einen Snapshot aus und geht auf EXPLORE, kann man das Fragment aus der angezeigten POM in die eigene POM-Datei kopieren.

1.2.4 Annotationsprozessoren konfigurieren

Der Spring Initializr baut eine POM-Datei auf, die wir an verschiedenen Stellen erweitern, in der Regel durch weitere Dependencies. Auch wird das Compiler-Plugin oft mit Annotationsprozessoren konfiguriert. Ein Annotationsprozessor (engl. *annotation processor*) kann Annotationen im Quellcode verarbeiten, zusätzlichen Code generieren oder bestimmte Aktionen basierend auf den Annotationen ausführen. Im Spring-Umfeld werden etwa JSON-Dateien zur Beschreibung von externen Konfigurationen oder Meta-Model-Klassen über Annotationsprozessoren erzeugt.

Die Einbindung eines Annotationsprozessors kann unterschiedlich erfolgen, etwa über eine `<annotationProcessorPaths>`-Konfiguration:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <annotationProcessorPaths>
        <! -- -->
        </annotationProcessorPaths>
      </configuration>
    </plugin>
</build>
```

Wenn wir einen Annotationsprozessor einbinden, werden wir im weiteren Verlauf des Buches noch auf die Konfiguration eingehen.

1.2.5 Starter: Dependencies des Spring Initializrs

In der POM-Datei hat der Initializr zwei Dependencies eingesetzt:

```
<?xml version="1.0" encoding="UTF-8"?>
<project …>
  …
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>
  …
</project>
```

Es gibt eine Dependency auf den `spring-boot-starter` und eine zweite Dependency auf `spring-boot-starter-test`, wobei die zweite Dependency im Scope `test` ist.

Die Funktion eines *Starters* ist, dass wir uns nicht um feine Abhängigkeiten kümmern müssen, sondern dass wir ein ganzes Bündel, also eine Sammlung von Abhängigkeiten, über diesen Starter bekommen.

Die offiziellen Starter beginnen alle mit einem Präfix, nämlich mit `spring-boot-starter-`, und die Group-ID ist immer `org.springframework.boot`. Der kleinste Starter nennt sich *Core-Starter* und bringt alles mit, was man für Spring-Boot-Anwendungen braucht.

In Abbildung 1.11 lassen sich die Abhängigkeiten gut ablesen: `spring-boot-starter` steht oben und hat eine ganze Reihe von Dependencies. Mit anderen Worten: Wenn wir eine Dependency auf den `spring-boot-starter` setzen, bekommen wir das Gezeigte mitreferenziert. Wir müssen also nicht mehr selbstständig zum Beispiel eine Logging-Bibliothek aufnehmen oder das Spring Framework selbst. Das steckt alles als transitive Abhängigkeit in diesem `spring-boot-starter`.

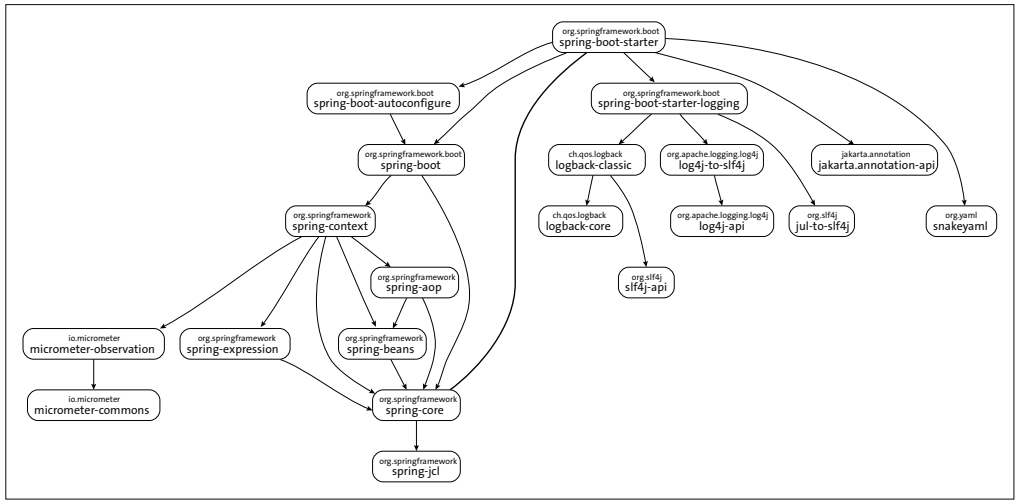


Abbildung 1.11 Dependencies des Core-Starters

Auf der linken Seite kann man ablesen, dass bei `spring-context` das Spring Framework selbst referenziert wird, inklusive der Abhängigkeiten. Auf der rechten Seite sehen wir die Abhängigkeit zu `spring-boot-starter-logging`, das heißt zur Logging-Infrastruktur. Dann werden noch ein paar Standardannotationen eingebunden, und der YAML-Parser *SnakeYAML* ist ebenfalls mit dabei.

Weitere Spring Boot Application Starter

Es gibt nicht nur einen Starter, sondern eine große Anzahl von Startern. Die folgende Liste zeigt nur eine Auswahl:

- `spring-boot-starter-jdbc` für Anwendungen mit Datenbankzugriffen über JDBC und DataSources
- `spring-boot-starter-data-jpa` für Anwendungen mit Jakarta Persistence
- `spring-boot-starter-json` für das Objekt-JSON-Mapping
- `spring-boot-starter-web` für Webservices und dynamische Webseiten inklusive des Servlet-Containers Tomcat

Eine Übersicht aller Starter liefert die Website *https://docs.spring.io/spring-boot/reference/using/build-systems.html#using.build-systems.starters*. Mit allen Startern lässt sich eine spezifische Technologie einfach einbinden, ohne dass wir uns im Detail um »Kleinigkeiten« kümmern müssen, sondern wir sagen: »Ich möchte irgendwas mit Datenbankzugriffen machen«, »Ich möchte Webentwicklung machen«, »Ich möchte irgendwas mit Websockets machen«, »Ich muss irgendwas mit Security tun« – und dahinter steckt dann ein ganzer Batzen von Unterabhängigkeiten.

Die Starter sind dabei nicht exklusiv und schließen sich nicht gegenseitig aus. Starter können also kombiniert verwendet werden: Wer eine Webanwendung mit einer Datenbankanbindungen bauen möchte, nimmt einen Starter für das Web selbstverständlich zusammen mit einem Starter für zum Beispiel Jakarta Persistence.

Manche Starter enthalten den Core-Starter, sodass man ihn nicht zwingend einbinden müsste. Man muss das allerdings nicht im Kopf haben, denn den Core-Starter mit einzubinden und zusätzlich zum Beispiel noch Web mit einzubinden, ist kein Fehler.

Hinweis

Die offiziellen Starter *beginnen* alle mit `spring-boot-starter-`, etwa `spring-boot-starter-web` oder `spring-boot-starter-jdbc`. Starter, die nicht von VMware kommen, sollten einen wohldefinierten Namen haben und mit dem Projektnamen beginnen, und *danach* sollte `-spring-boot-starter` stehen. Beispiele aus der Open-Source-Welt wären `grpc-spring-boot-starter` oder `okta-spring-boot-starter`.

1.3 Einstieg in die Konfigurationen und das Logging

Der Initializr legt diverse Dokumente an, darunter war auch die Datei *application.properties*. Diese Datei ist nützlich für die Konfiguration von Spring-Boot-Anwendungen. Spring Boot hat diverse Property-Quellen. Darunter versteht man verschiedene Quellen, aus denen Spring Boot Konfigurationen zieht; *application.properties* oder *application.yml* sind zwei dieser Quellen, die Kommandozeile wäre eine andere.

Üblicherweise liegt die Datei im Klassenpfad, also im Quellcodeordner *src/main/resources*. Alles, was in diesem Ordner liegt, wird später im Wurzelverzeichnis der gebauten Anwendung liegen.

1.3.1 Banner abschalten

Als Erstes wollen wir das beim Start angezeigte Banner abschalten:

Listing 1.3 application.properties

```
spring.main.banner-mode=off
```

Startet man das Programm erneut, ist das Banner verschwunden.

Es gibt drei Belegungen für `spring.main.banner-mode`:

- `off`: ganz ausschalten
- `console`: Das ist der Standard: Das Banner erscheint auf `System.out`.
- `log`: schreibt das Banner in den aktuellen Log-Strom.

IDE-Tipp

Die Entwicklungsumgebung »kennt« fast alle Konfigurations-Properties, und eine Tastaturvervollständigung ist möglich. Auch bei den Werten. Schreibt man `spring.main.banner-mode=`, so kennt die Entwicklungsumgebung die drei möglichen Konfigurationsbelegungen `off`, `console` und `log`. Auch wird die Dokumentation angezeigt.

1.3.2 Die Logging-API und SLFJ

Der Spring-Boot-Starter hat eine Dependency auf `spring-boot-starter-logging`. Dieser Starter bietet eine Logging-Infrastruktur und konfiguriert zwei Logging-Fassaden:

- *Simple Logging Facade for Java* (SLFJ; *https://sslf4j.org*)
- *Commons Logging API* (*https://commons.apache.org/proper/commons-logging*). Das Spring Framework nutzt diese Fassade intern; man möchte aber seit Jahren eigentlich von ihr weg.

Direkt mit einer Logging-Bibliothek zu arbeiten, zum Beispiel mit *Logback* oder mit *Log4j*, ist eher unüblich. Man arbeitet üblicherweise mit sogenannten *Log-Fassaden*. Eine Fassade ist ein klassisches Design-Pattern, das Anfragen von außen entgegennimmt und dann delegiert – es ist eine einfache Schnittstelle für ein kompliziertes Subsystem. Der Vorteil ist, dass die Details der konkreten Logging-Bibliotheken hinter der Fassade versteckt werden, wenn wir ausschließlich über eine Fassade loggen. So ist es einfach möglich, das Logging-Framework später auszuwechseln. Heute könnte es Logback, morgen Log4j2 sein.

Die SLF4J-API nutzen

Wir wollen ein kleines Beispiel mit der SLF4J-Bibliothek programmieren. Letztendlich sind dafür nur drei Schritte nötig:

1. Wir müssen eine Importdeklaration für die Typen setzen:

   ```
   import org.slf4j.*;
   ```

2. Wir deklarieren eine Variable `log`:

   ```
   private final Logger log = LoggerFactory.getLogger( getClass() );
   ```

   Ob die Variable statisch ist oder nicht, kommt ein wenig auf den Anwendungsfall an.

3. Und dann können wir die entsprechenden Logging-Methoden einsetzen:

   ```
   log.info( "Log with arguments{}, {} und {}", 1, "2", 3.0 );
   ```

Logger-Methoden heißen `debug(…)`, `error(…)`, `info(…)` usw. Sie zeigen die Dringlichkeit an, mit der etwas gemeldet werden soll.

Bei dem `log.info(…)`-Methodenaufruf lässt sich etwas Vergleichbares wie bei `System.out.printf(…)` ablesen: ein Formatstring, der aber mit den geschweiften Klammern als Platzhalter viel einfacher ist als ein Java-`Formatter`. Bei den Platzhaltern geht es nur nach der Reihenfolge. Wenn es `log.info("Log mit Argumenten {}, {} und {}", 1, "2", 3.0)` heißt, kommt am Ende `Log mit Argumenten 1, 2 und 3.0` heraus.

Log-Level

Am `Logger`-Objekt hängen unterschiedliche Log-Methoden, zum Beispiel `debug(…)`, `info(…)`, `error(…)`. Es ist nicht selbstverständlich, dass alle Ausgaben erscheinen. Nehmen wir Folgendes:

```
log.debug( "Debug Level Log" );
log.info( "Info Level Log" );
log.error( "Log with arguments {}, {} und {}", 1, "2", 3.0 );
```

Es werden nicht alle drei Ausgaben auf der Konsole stehen, denn standardmäßig erscheint die Debug-Meldung nicht auf der Konsole. Der Grund liegt im *Log-Level*.

Der Log-Level definiert eine Dringlichkeit, die mindestens gegeben sein muss, damit eine Meldung geschrieben wird. SLF4J definiert die Log-Level `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`, `FATAL` und grundsätzlich auch `OFF` für das Abschalten von Meldungen. Wenn die `debug(…)`-Meldungen nicht erscheinen, liegt das daran, dass aktuell der Log-Level ein anderer ist, das heißt die Dringlichkeit über `DEBUG` liegt.

Standardmäßig ist der Log-Level auf `INFO` eingestellt. Das bedeutet, dass nur `INFO`-Meldungen und »dringlichere« geschrieben werden, also Meldungen mit den Leveln `INFO`, `WARN`, `ERROR` und `FATAL`, aber nichts, was weniger wichtig ist. Das heißt, beim Log-Level `INFO` werden `DEBUG`- und `TRACE`-Meldungen nicht geloggt. Die Hierarchie der Level ist: `TRACE` < `DEBUG` < `INFO` < `WARN` < `ERROR` < `FATAL`.

Log-Level setzen

Die Log-Level lassen sich über Konfigurations-Properties setzen. In der Datei *application.properties* könnte Folgendes gesetzt werden:

```
logging.level.com.tutego.date4u=DEBUG
logging.level.org.springframework=ERROR
```

Die Konfigurations-Properties beginnen mit dem Präfix `logging.level`. Anschließend folgt eine Paketangabe oder ein vollqualifizierter Typ. In unserem Beispiel setzen wir den Log-Level von `com.tutego.date4u` auf `DEBUG`. Der Log-Level wird dabei nicht nur exakt für dieses Paket gesetzt, sondern die Angabe gilt ebenso für alle Unterpakete. Gibt

es zum Beispiel unter `com.tutego.date4u.very.deep.nested` eine Klasse mit einer `debug(…)`-Meldung, so wird die Log-Meldung ebenfalls erscheinen.

Hinter `logging.level` kann auch ein konkreter Typ stehen. Das ist nützlich, wenn zum Beispiel für ganz konkrete Klassen der Log-Level gesetzt werden soll, aber nicht für alle anderen Typen im gleichen Paket gleich mit. Ein spezieller Log-Level für einen Typ oder ein Paket überschreibt eine übergeordnete Einstellung.

Auch die Logging-Ausgaben des Spring Frameworks lassen sich so konfigurieren. Ist zum Beispiel der `logging.level` vom `springframework` auf `ERROR` gesetzt, wird alles, was weniger wichtig ist als `ERROR`, nicht ausgegeben. Es ist durchaus ein Experiment, den Log-Level von `org.springframework` auf `TRACE` zu setzen. Dann sieht man nämlich alles, was das Spring Framework loggt. Allerdings verlangsamt die massive Ausgabe dieser Log-Ausgaben die Anwendung.

Tipp

Auf Produktivsystemen sind Ausgaben (wie Log-Meldungen) über die Konsole selten, weil diese langsam sind; anders sieht das aus, wenn die Log-Ströme in Dateien umgeleitet werden. Obendrein sollte ein Programm nur loggen, was später auch ausgewertet wird. Es ist darauf zu achten, dass keine sicherheitsrelevanten Daten geloggt und gespeichert werden. Bei Java-Anwendungen im Container (Docker, Kubernetes-Cluster) werden Log-Ausgaben in die Konsole geschrieben.

Ausklang

Zur Einführung zeigte dieses Kapitel den Unterschied zwischen der Java SE-Plattform, Jakarta EE und Spring auf. Spring nutzt viele Jakarta EE-Standards, ist aber kein kompatibler Jakarta-Enterprise-Application-Server.

Als Nächstes haben wir gesehen, wie der Spring Initializr entweder über die Webseite oder eingebaut in der Entwicklungsumgebung hilft, ein neues Spring-Boot-Projekt aufzubauen. Es gibt noch eine weitere Möglichkeit über das Spring Boot CLI, ein Kommandozeilenwerkzeug. Da das in der Praxis selten gebraucht wird, verweise ich auf die Referenzdokumentation *https://docs.spring.io/spring-boot/cli/index.html*.

Nachdem wir das Projekt aufgebaut haben, können wir uns im nächsten Schritt intensiver mit dem Spring-Container beschäftigen. Der Spring-Container verwaltet sogenannte Spring-managed Beans. Wir wollen lernen, wie man neue Komponenten in dem Spring-Container aufnimmt und wie man sie erfragt.

Kapitel 7
Spring Data JPA

In diesem Kapitel wollen wir ein neues Themengebiet angehen, und zwar *Spring Data*. Mit Spring Data lassen sich schnell und unkompliziert Repositories für unterschiedliche Datenbanksysteme realisieren.

7.1 Welche Aufgaben erfüllt Spring Data?

Natürlich hat jedes Datenbankmanagementsystem eine eigene API: Relationale Datenbanken werden über JDBC oder JPA angesprochen, MongoDB hat eine eigene API, eine Textsuchswmaschine wie Elasticsearch ebenso. Diese APIs arbeiten oftmals auf einem sehr niedrigen Level und sind völlig verschieden.

Mit dem Spring-Data-Projekt lassen sich mühelos Datenzugriffsschichten realisieren. Die Besonderheit ist, dass Spring Data eine eigene API definiert und von der konkreten Datenbanktechnologie weit abstrahiert (siehe Abbildung 7.1).

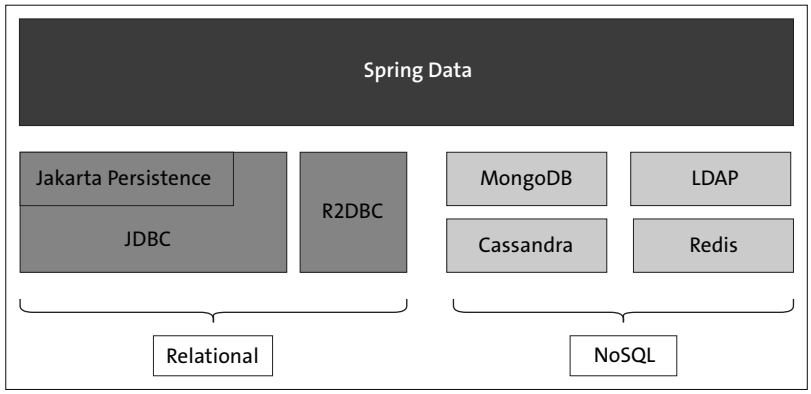


Abbildung 7.1 Spring Data: eine API für alle Systeme

Es ist eine großartige Leistung von Spring Data, eine gemeinsame API zu schaffen, die relationale und auch nichtrelationale Datenbankmanagementsysteme vereinheitlicht und unter einem Dach zusammenfasst. Entwickelnde müssen nicht mehr die API für unterschiedlichste Systeme lernen, sondern es reicht die API von Spring Data. Im Umfeld von Jakarta EE gibt es mittlerweile Jakarta Data. Das ist relativ neu und das Spring-Team hat sich bewusst gegen die Unterstützung entschieden, da mit Spring Data schon eine Lösung existiert.

Die Möglichkeiten von Spring Data in einer Kurzübersicht:

- codegleiche Repository-Schnittstellen für relationale und nichtrelationale Datenbanken
- Aus bestimmten Methodennamen lassen sich Queries aufbauen, sodass keine Abfragesprache nötig ist.
- Spring Data unterstützt das Auditing. Vermerkt wird, wer wann wo Daten verändert hat.
- Leicht lassen sich durch das Teilprojekt Spring Data REST-Repositories als REST-Endpunkte veröffentlichen.

Für welche Systeme gibt es Spring Data?

Spring Data kann man sich als Familie mit unterschiedlichen Familienmitgliedern vorstellen. Es gibt sozusagen die »Familien-API«, und die einzelnen Mitglieder der Familie implementieren diese API für ihr zentrales Datenbankmanagementsystem.

Die Website von Spring Data[1] listet auf, für welche Datenbankmanagementsysteme VMware Implementierungen bereitstellt. Tabelle 7.1 fasst das zusammen.

Offizielle Spring-Data-Module	Community-Module
JPA	Neo4j
JDBC	Elasticsearch
MongoDB	Couchbase
KeyValue	Hazelcast
Redis	Azure Cosmos DB
LDAP	Google Spanner
Cassandra	und weitere ...
Geode	
GemFire	
R2DBC	

Tabelle 7.1 Offizielle Module und Community-Module aus dem Spring-Data-Projekt

Ganz prominent als offizielles Modul steht *Spring Data JPA* zur Verfügung, das wir für die Beispiele verwenden. Aber auch für andere wichtige Datenbankmanagementsys-

1 *https://projects.spring.io/spring-data/*

teme (wie MongoDB und LDAP) oder für Key-Value-Stores wie Redis gibt es von VMware Unterstützung.

Einige Spring-Data-Implementierungen werden nicht von VMware aufgebaut und gepflegt, sondern von den Herstellern der Datenbankmanagementsysteme. Dazu zählen etwa die Graphdatenbank Neo4j, die Textsuchmaschine Elasticsearch oder die Column-White-Space-Database Couchbase. Hier möchte VMware nicht die Verantwortung übernehmen und investieren, sondern das sollen die Hersteller dieser Datenbanksysteme bitte schön selbst übernehmen.

Hinweis

Dieses Kapitel beschreibt die verschiedenen Möglichkeiten von Spring Data am Beispiel von Spring Data JPA. Wer nicht mit Jakarta Persistence arbeitet, sondern zum Beispiel mit MongoDB, ist dennoch angehalten, dieses Kapitel zu lesen, um sich mit den Datentypen vertraut zu machen.

7.2 Spring Data Commons: CrudRepository

Spring Data verwaltet Entitäten, und jede dieser Entitäten hat eine ID. Es ist egal, in welchem Datenbankmanagementsystem man sich befindet – die Datensätze müssen grundsätzlich identifiziert werden. Bei einem relationalen Datenbankmanagementsystem ist das der Primärschlüssel einer Zeile; aber auch bei einem dokumentenorientierten Datenbankmanagementsystem, wie MongoDB, haben die Einträge eine ID und müssen eindeutig identifiziert werden können.

Objekte, die Spring Data verwaltet, nennen sich *Entitäten*. Für das Finden, Schreiben, Aktualisieren und Löschen dieser Entitäten deklariert Spring Data diverse Schnittstellen, unter anderem `CrudRepository`[2]. Wir haben diese Schnittstelle in Abschnitt 6.14.3, »SimpleJpaRepository«, kurz kennengelernt. Die Schnittstelle `CrudRepository` ist ein Teil des Spring-Data-Commons-Projekts und etwas, was später alle konkreten Familienmitglieder für ihre Datenbankmanagementsysteme implementieren. Die Schnittstelle wird also nie von uns implementiert, sondern wir bekommen eine Implementierung von Spring Data zur Laufzeit für unser konkretes Datenbankmanagementsystem.

2 *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html*

7.2.1 Der Typ CrudRepository

Die Schnittstelle `CrudRepository` ist ein generischer Typ mit zwei Typvariablen `T` und `ID`:

```
package org.springframework.data.repository;
import java.io.Serializable;

@NoRepositoryBean
public interface CrudRepository<T, ID> extends Repository<T, ID> {
  …
}
```

Die Typvariable `T` steht für den Typ der Entität, die dieses `CrudRepository` übernimmt. Das kann immer nur eine Entität sein, beispielsweise ein `Unicorn`, ein `Profile`, eine Bestellung, ein Kunde oder etwas anderes. Die Typvariable `ID` steht für den Typ des Schlüssels. Häufig dürfte das ein `Long`, ein `String` oder eine `UUID` sein.

Die Schnittstelle `CrudRepository` erweitert die Schnittstelle `Repository`, die keine Methoden hat. Welchen Nutzen das hat, werden wir in Abschnitt 7.3.3, »Ausgewählte Methoden über Repository«, besprechen.

Methoden von CrudRepository

`CrudRepository` selbst deklariert eine ganze Reihe von Methoden für CRUD-Operationen, also Methoden für das *Create*, *Read*, *Update* und *Delete* von Entity-Beans.

Der folgende Codeausschnitt gruppiert die Methoden um die folgenden vier Operationen: Create und Update (zum Anlegen, Speichern/Aktualisieren), Read (Operationen rund ums Lesen) und Delete (Operationen, die löschen):

```
@NoRepositoryBean
public interface CrudRepository<T, ID> extends Repository<T, ID> {
  // C(reate) / U(pdate)
  <S extends T> S save(S entity);
  <S extends T> Iterable<S> saveAll(Iterable<S> entities);

  // R(ead)
  long count();
  boolean existsById(ID id);
  Optional<T> findById(ID id);
  Iterable<T> findAll();
  Iterable<T> findAllById(Iterable<ID> ids);

  // D(elete)
  void delete(T entity);
```

```
  void deleteById(ID id);
  void deleteAll(Iterable<? extends T> entities);
  void deleteAll();
  void deleteAllById(Iterable<? extends ID> ids);
}
```

Die Methode `save(…)` speichert eine Entität, und die Rückgabe ist die gespeicherte Entität. `saveAll(…)` nimmt ein `Iterable`, also eine Sammlung von Entitäten an, wie eine `ArrayList`; das Ergebnis ist wieder ein `Iterable` mit den gespeicherten Entitäten. Die Besonderheit bei Spring Data ist, dass es im `CrudRepository` nur eine `save(…)`-Methode gibt und dass es egal ist, ob die Entität neu gespeichert oder aktualisiert gespeichert wird. Bei der Jakarta Persistence muss der Client entweder die `persist(…)`- oder die `merge(…)`-Methode aufrufen, denn SQL unterscheidet zwischen dem Neuspeichern (SQL-`INSERT`) und dem Aktualisieren (SQL-`UPDATE`). Spring Data JPA wählt die korrekte `EntityManager`-Methode für SQL-basierte Datenbanken intern aus. Da bei anderen Datenspeichern die Unterscheidung zwischen Neuspeichern und Aktualisierung oft nicht gemacht wird, definiert Spring Data eine `save*(…)`-API für alle möglichen Datenspeicher.

Schauen wir uns als Nächstes die lesenden Methoden an. `count()` liefert die Anzahl der Datensätze zurück. `existsById(…)` ermittelt, ob eine Entität zu einem gewissen Schlüssel existiert. `findById(…)` versucht, eine Entität zu laden. Falls die Entität nicht gefunden wurde, gibt es ein `Optional.empty()`, andernfalls ist das `Optional` mit der geladenen Entität präsent. Die Methode `findAll()` liefert alle Entitäten aus der Datenbank. Die zu ladenden Entitäten können auch als `Iterable` aufgeführt werden; die Methode `findAllById(…)` liefert ein `Iterable` genau dieser geladenen Objekte zurück.

Zum Schluss gibt es verschiedene `delete*(…)`-Methoden, die Varianten sind selbsterklärend.

Das sind alle vordefinierten Methoden der Schnittstelle `CrudRepository`. Im nächsten Schritt müssen wir uns ein `CrudRepository` von Spring geben lassen.

7.2.2 JPA-basierte Repositories

Wir haben schon mit dem Spring-Data-JPA-Projekt gearbeitet und selbst mit dem `EntityManager` die Entity-Beans geladen und uns die Speicheroperationen angeschaut. Nutzt man Repositories von Spring Data JPA, hat man mit der `EntityManager`-API nichts mehr zu tun. Intern geht die Realisierung dieser CRUD-Methoden weiterhin auf den `EntityManager`. Das ist auch der Grund, warum wir den `EntityManager` in der Vergangenheit nicht so oft für Beispiele verwendet haben, denn üblicherweise nutzen Spring-Boot-Anwendungen Spring Data.

Ein eigenes ProfileRepository

Wir wollen ein Repository deklarieren, das CRUD-Operationen für `Profile` anbietet. Dazu schreibt man ein eigenes Interface, das vom Basistyp `CrudRepository` erbt. (Es existieren andere Basistypen, was einige Vorteile bietet; das schauen wir uns auch an.) So sieht das aus:

```
public interface ProfileRepository extends CrudRepository<Profile, Long> { }
```

Die Typargumente bei den Generics stehen für den Typ der Entity (`Profile`) und den Typ des Schlüssels (`Long`).

Die Besonderheit ist, dass die eigenen Schnittstellen nicht von uns implementiert werden: Wir deklarieren nur eine Schnittstelle, die zwar weitere abstrakte Methoden bekommen kann, aber in der Regel keine Default-Methoden enthält.

Wenn wir eine Schnittstelle deklarieren müssen und keine Klasse, stellt sich die Frage, wer die Schnittstelle implementiert. Das macht Spring Data zur Laufzeit! Ein Spring-Data-Familienmitglied baut zur Laufzeit ein Objekt auf, das zum Beispiel `ProfileRepository` erweitert. Auf dieser `ProfileRepository`-Implementierung lassen sich alle Methoden aufrufen. Die Spring-Data-JPA-Implementierung bildet die Methoden intern auf den `EntityManager` ab.

Bei der Schnittstelle ist es nicht angebracht, Komponentenannotationen, wie `@Repository`, zu setzen. Das braucht man nur, wenn man eigene Repository-Klassen programmiert. Gehen wir über Schnittstellen, dann ist `@Repository` unnötig.

Ein Minimalprogramm mit `ProfileRepository` könnte so aussehen:

```
@SpringBootApplication
public class SpringDataJpaApplication {

  private final Logger log = LoggerFactory.getLogger( getClass() );

  public SpringDataJpaApplication( ProfileRepository profiles ) {
    log.info( "Profile with id=1: {}", profiles.findById( 1L ) );
    log.info( "All profiles: {}", profiles.findAll() );
  }

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

Über eine Constructor-Injection bekommen wir die Laufzeitimplementierung von `ProfileRepository`, und es lassen sich alle vordefinieren CRUD-Methoden aufrufen.

Aufgabe: RepositoryCommands mit neuen Shell-Methoden

Wer mag, kann selbst Hand am Code anlegen und folgende Aufgabe umsetzen:

1. Lege eine neue Klasse `RepositoryCommands` an, die auch wiederum mit `@ShellComponent` annotiert ist.
2. Injiziere das `ProfileRepository` in die neue Spring-Bean `RepositoryCommands`.
3. Schreibe ein neues Kommando `list`, das alle Profile ausgibt.
4. Probiere diverse `CrudRepository`-Methoden in eigenen neuen Kommandos aus, etwa indem ein neues `Profile` aufgebaut, in der Datenbank gespeichert und später gelöscht wird. Alle modifizierenden Methoden wie `save(…)` sind automatisch transaktional, so wie es in Abschnitt 6.14.3, »SimpleJpaRepository«, schon deutlich wurde.

PS: `EntityManagerCommands` kann auskommentiert werden.

Lösungsvorschlag:

```
@ShellComponent
public class RepositoryCommands {

  @Autowired
  private ProfileRepository profiles;

  @ShellMethod( "Display all profiles" )
  public void list() {
    profiles.findAll().forEach( System.out::println );
  }

  @ShellMethod( "random" )
  public Optional<Profile> random() {
    Profile generate = new Profile(
        "JuicyJenelyn",
        LocalDate.now().minusYears( 22 ),
        12,
        Profile.FEE,
        null,
        "",
        LocalDateTime.now().minusDays( 12 ) );
    profiles.save( generate );
    return profiles.findById( generate.getId() );
  }

  @ShellMethod( "update" )
  public Optional<Profile> update() {
    Optional<Profile> maybeProfile = profiles.findById( 1L );
```

```
    maybeProfile.ifPresent( profile -> {
      System.out.println( profile );
      profile.setNickname( "King" + profile.getNickname() + "theGreat" );
      profiles.save( profile );
    } );
    return profiles.findById( 1L );
  }
}
```

Interner Ablauf: Vom Client zur Datenbank

Auf dem Weg vom Spring-Data-JPA-Projekt über den `EntityManager` und den JDBC-Treiber zur Datenbank gibt es verschiedene Abstraktionen; gehen wir diese durch. Wird auf dem Repository eine Methode aufgerufen, so wird sie von einem Proxy realisiert, der genau die Schnittstelle implementiert. Bei einem Spring-Data-JPA-Projekt werden im Hintergrund entsprechende `EntityManager`-Methoden aufgerufen. Diese wiederum gehen über den JDBC-Treiber zur Datenbank.

Es gibt eine Reihe von Abstraktionsschichten, die wir auswechseln könnten:

- Es lässt sich die Datenbank ändern, und der Client sollte davon nichts mitbekommen.
- Der JDBC-Treiber könnte ausgetauscht werden; das bekommt der Client nicht mit. Das ist in diesem Szenario am unrealistischsten, denn JDBC-Treiber werden von den Herstellern der RDBMS in enger Abstimmung entwickelt, und hier sollte man keine Experimente machen. Oracle ist dafür bekannt, dass die Versionsnummern der JDBC-Treiber mit den Versionsnummern der Oracle-Datenbank Schritt halten müssen.
- Der Client würde auch nicht bemerken, wenn der Jakarta Persistence Provider ausgewechselt wird. Egal, ob es Hibernate ORM oder EclipseLink JPA ist, beide implementieren die Spezifikation.
- Die einzige API, mit der der Client etwas zu tun hat, ist aktuell das `CrudRepository`. Selbst wenn das Datenbankmanagementsystem ausgewechselt wird und die Anwendung von einer relationalen Datenbank zum Beispiel zur MongoDB wechselt, hat das `ProfileRepository`, so wie es aktuell programmiert ist, keinen Bezug zu Jakarta Persistence, zu Tabellen oder Spalten. Der einzige Bezug ist der Behälter, die Entity-Bean-Klasse, weil sie Annotationen trägt.

7.3 Untertypen von CrudRepository: JpaRepository etc.

Die Schnittstellen `CrudRepository` und `Repository` sind sehr allgemein. Es gibt andere Schnittstellen, von denen es sich zu erben lohnt. Ein paar weitere Basistypen werden wir in diesem Abschnitt besprechen.

7.3.1 ListCrudRepository

Während der Basistyp bei `CrudRepository` Sammlungen nur als `Iterable` herausgibt, nutzt der Untertyp `ListCrudRepository`[3] (siehe Abbildung 7.2) eine Spezialisierung beim Rückgabetyp in Form von Listen – wenn ein Untertyp eine Methode überschreibt und eine Spezialisierung in der Rückgabe nutzt, nennt man das in Java einen *kovarianten Rückgabetyp*.

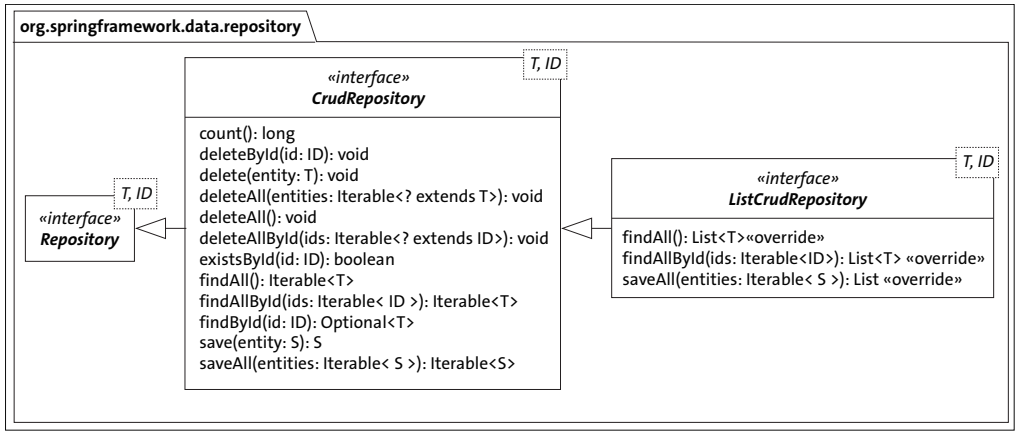


Abbildung 7.2 Methoden von »ListCrudRepository« liefern »List« statt »Iterable«.

Ein `Iterable` liefert lediglich einen `Iterator`, und ein `Iterator` ermöglicht es nur, eine Datenquelle sequenziell abzulaufen – das ist nicht komfortabel.

7.3.2 Technologiespezifische [List]CrudRepository-Untertypen

Das [`List`]`CrudRepository` ist völlig unabhängig vom Datenspeicher und der gemeinsame Nenner für alle Spring-Data-Projekte. Keine der Methoden verrät etwas von SQL, Relationen oder Dokumenten. Der Vorteil ist, dass jedes Datenbankmanagementsystem [`List`]`CrudRepository` implementieren kann. Allerdings kann man das auch als Nachteil betrachten, weil technologiespezifische Besonderheiten wünschenswert sind, wie Batch-Operationen bei RDBMS oder Gewichtungen von Suchergebnissen bei Textsuchen.

Spring Data kann man sich in etwa so vorstellen wie JDBC – es gibt eine API und Implementierungen. Im Fall von JDBC sind das die JDBC-Treiber: Sie implementieren die JDBC-API und »wissen«, wie eine konkrete Datenbank angesprochen wird. Beim Spring-Data-Projekt ist das ähnlich: Spring Data Commons deklariert gemeinsame Datentypen (wie die Schnittstellen `Repository` oder [`List`]`CrudRepository`) und Hilfs-

3 *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/ListCrudRepository.html*

typen (wie `Sort` oder `Pageable`). Eine Implementierung für einen Datenspeicher ist Spring Data Commons nicht. Das ist die Aufgabe der Spring-Data-Teilprojekte, wie Spring Data JPA. Sie implementieren die API und deklarieren in der Regel zusätzliche Datentypen. Das Diagramm in Abbildung 7.3 macht das deutlich.

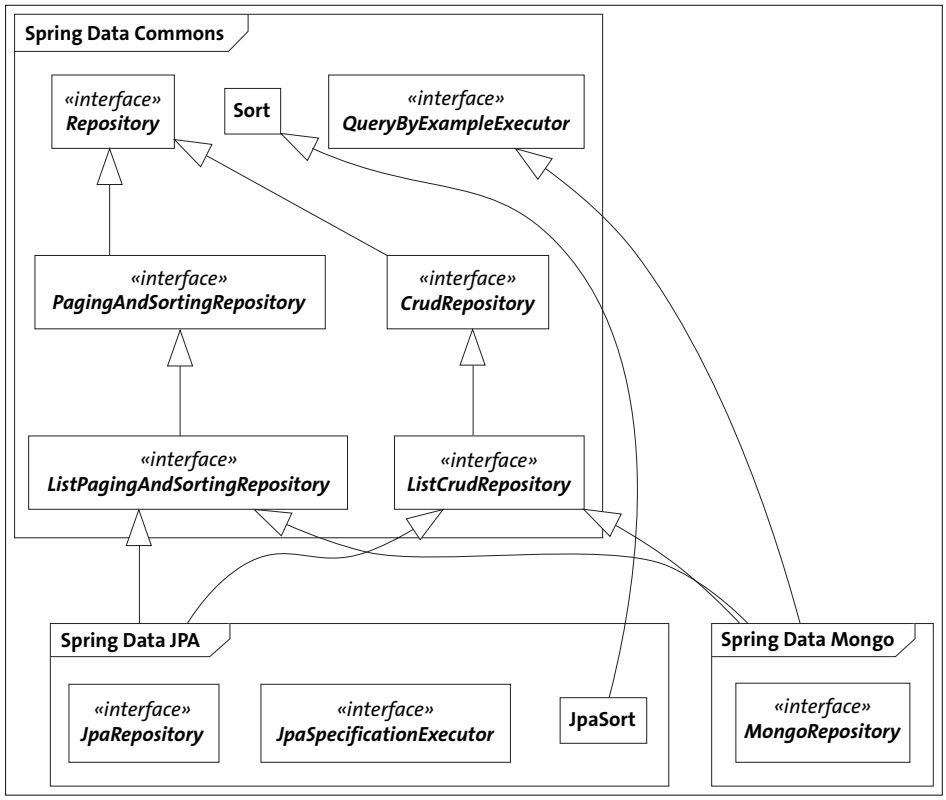


Abbildung 7.3 Typbeziehungen der Spring-Data-Repositories

Aus den Typbeziehungen lässt sich ablesen, dass die verschiedenen Projekte Untertypen von [`List`]`CrudRepository` bilden. So können die zugrunde liegenden Speichersysteme optimaler unterstützt werden. Für JPA gibt es etwa `JpaRepository`, für die MongoDB-Datenbank `MongoRepository` usw.

JpaRepository nutzen

In unserem Fall können wir das `ProfileRepository` »verbessern«, indem wir statt `CrudRepository` den Untertyp `JpaRepository`[4] verwenden. Aus

```
public interface ProfileRepository extends CrudRepository<Profile, Long> { }
```

4 *https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html*

wird:

Listing 7.1 ProfileRepository.java, Änderung

```
public interface ProfileRepository extends JpaRepository<Profile, Long> { }
```

Methoden von JpaRepository und Basistypen

Das `JpaRepository` statt des [`List`]`CrudRepository`s zu nutzen, bietet mehrere Vorteile. Das UML-Diagramm in Abbildung 7.4 macht das deutlich.

Der erste Unterschied ist, dass neue Methoden durch weitere Obertypen dazukommen. Ein `JpaRepository` erweitert `ListCrudRepository` und `ListPagingAndSortingRepository` (zu diesem Typ werden wir in Abschnitt 7.4, »Blättern und Sortieren mit [List]PagingAndSortingRepository«, mehr lernen.) Aber dadurch kommen zwei zusätzliche Methoden in das `JpaRepository`. Außerdem erbt `JpaRepository` von `QueryByExampleExecutor`, was noch einmal eine Reihe von zusätzlichen Methoden ergibt. Die Methoden sind im UML-Diagramm ausgeblendet, und wir kommen auf den Datentyp in Abschnitt 7.5, »QueryByExampleExecutor *«, zu sprechen.

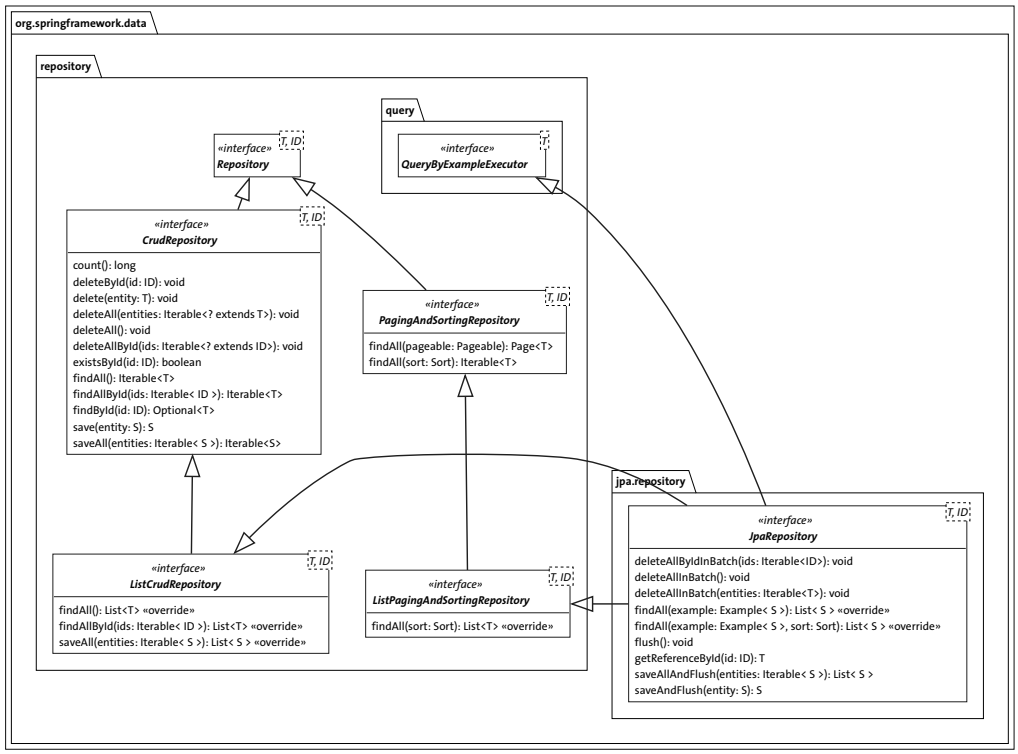


Abbildung 7.4 Obertypen und Methoden von »JpaRepository«

Dazu kommt, dass `JpaRepository` Methoden hat, die es in den Obertypen nicht gibt, wie die Batch-Methoden. Relationale Datenbanken können im Batch-Processing sehr performant Massendaten verarbeiten – den Vorteil möchte man nutzen. Andere Datenspeicher bieten keine Batch-Operationen, daher können sie nicht im Basistyp stehen.

Es gibt einen dritten Unterschied, der kaum zu erkennen, aber nützlich ist: `JpaRepository` überschreibt zwei Methoden aus `QueryByExampleExecutor` mit kovarianten Rückgabetypen, um `List` statt `Iterable` zurückzugeben.

Tipp

In der Praxis ist der Basistyp gar nicht so wichtig, denn Methoden und Rückgabetypen sind immer vorhanden und lassen sich ergänzen, wie der folgende Abschnitt zeigen wird.

7.3.3 Ausgewählte Methoden über Repository

Der Nachteil einer Erweiterung von Schnittstellen wie `JpaRepository` oder [`List`]`CrudRepository` ist, dass sie einen ganzen Satz von Methoden in die eigenen Repositories bringen, die vielleicht nicht willkommen sind. Insbesondere die `delete*(…)`-Methoden können unerwünscht sein, wenn Datensätze nicht gelöscht werden sollen.

Um Methoden »auszublenden«, kommt der allgemeinste Typ, `Repository`[5], ins Spiel. Von dieser Schnittstelle sind in Spring Data alle `Repository`-Schnittstellen abgeleitet. Der Basistyp deklariert keine Methoden (siehe Abbildung 7.5).

Eine Schnittstelle ohne CRUD-Methoden ergibt auf den ersten Blick wenig Sinn. Der Trick ist aber der: Wir erweitern `Repository` und setzen sozusagen per Copy and Paste genau die Methoden in die Schnittstelle, die unser Repository haben soll. Das könnte so aussehen:

```
public interface ReadOnlyProfileRepository
    extends Repository<Profile, Long> {
  boolean existsById( Long id );
  List<Profile> findAll();
  List<Profile> findAllById( Iterable<Long> ids );
  Optional<Profile> findById( Long id );
}
```

5 *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/Repository.html*

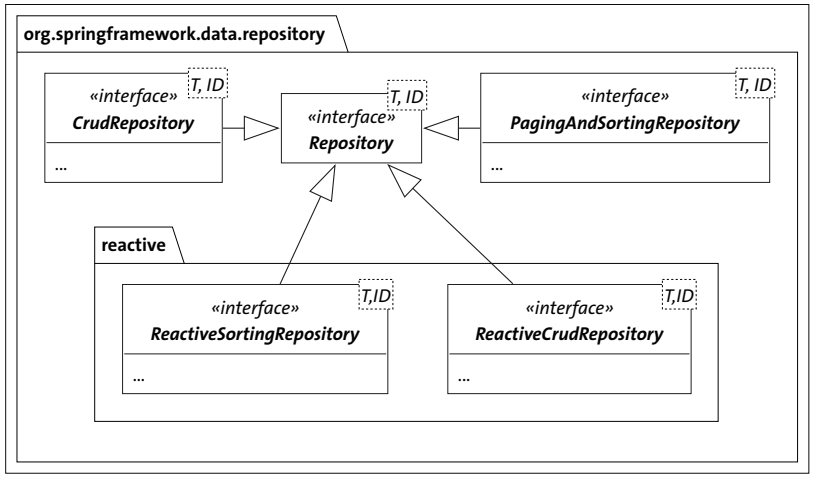


Abbildung 7.5 UML-Diagramm von »Repository« und seinen Untertypen

Diese Form der Copy-and-Paste-Programmierung ist vielleicht nicht optimal, aber so bekommt der Client nur die API, die er benötigt – das ist eine Anwendung des *Interface Segregation Principle* (ISP), das wir in Abschnitt 7.11.2, »Denke an ISP und die Zwiebel!«, erneut ansprechen.

Schreibfehler bei den Methoden werden bis zu einem gewissen Grad erkannt. Dass ein Repository »magische« Methoden mit speziellen Namen haben kann, die dann zur Laufzeit implementiert werden, ist ein Vorgeschmack auf Abschnitt 7.8, »Derived Query Methods«.

7.4 Blättern und Sortieren mit [List]PagingAndSortingRepository

In diesem Abschnitt geht es um die Typen [`List`]`PagingAndSortingRepository`[6], die in Abbildung 7.5 schon auftauchten. Der Datentyp erlaubt:

- das »Blättern« durch große Ergebnismengen, das auch *Paginierung* genannt wird
- das Sortieren von Ergebnissen nach gewissen Eigenschaften

Das `PagingAndSortingRepository` erweitert `Repository` (siehe Abbildung 7.6).

Die Schnittstelle `PagingAndSortingRepository` führt zwei neue Methoden ein: einmal `findAll(Sort)` und einmal `findAll(Pageable)`. Dabei repräsentiert `Sort` die Ordnungskriterien und `Pageable` den Ausschnitt/Block, den man vor- und zurücksetzen kann. Der Untertyp `ListPagingAndSortingRepository`[7] nutzt wieder einen kovarianten Rückgabetyp – `List` statt `Iterable`.

6 *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/PagingAndSortingRepository.html*

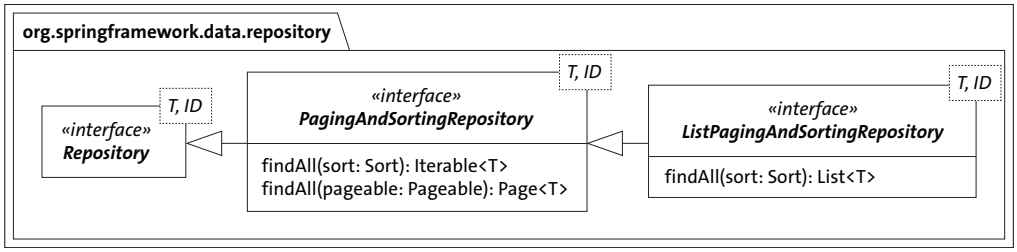


Abbildung 7.6 Typbeziehungen von »[List]PagingAndSortingRepository«

JpaRepository: Untertyp von ListPagingAndSortingRepository und ListCrudRepository

Das `JpaRepository` liegt unter `ListPagingAndSortingRepository` und `ListCrudRepository`, vereint also beide Schnittstellen (siehe Abbildung 7.7).

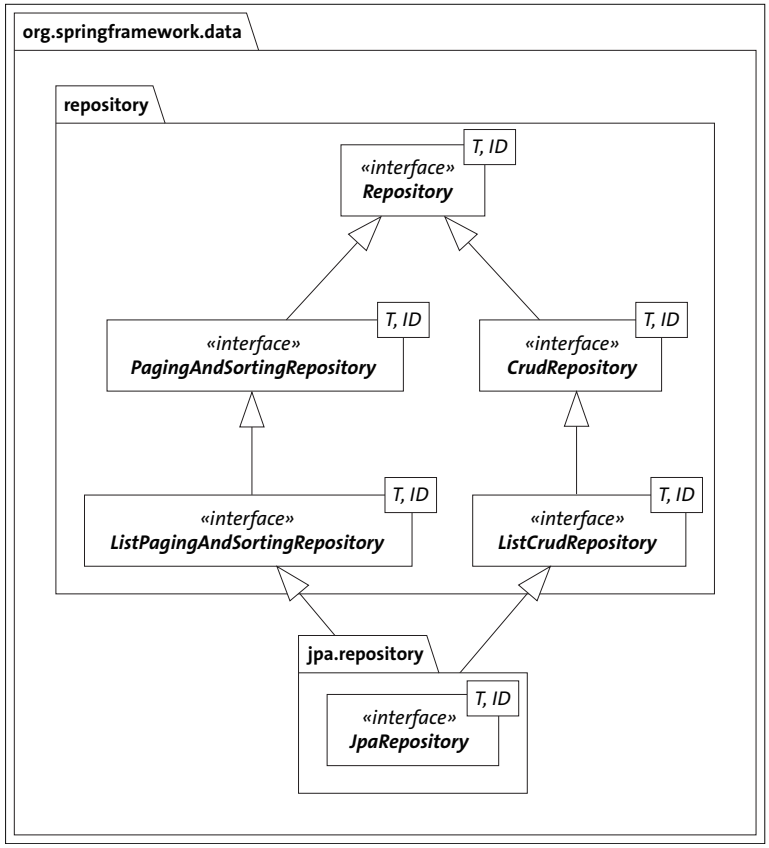


Abbildung 7.7 Ein »JpaRepository« ist ein »ListCrudRepository« und »ListPagingAndSortingRepository«.

7 *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/ListPagingAndSortingRepository.html*

Folglich hat auch unser `ProfileRepository` die mit `Sort` und `Pageable` parametrierten `findAll(…)`-Methoden.

7.4.1 Der Typ Sort

Beginnen wir mit der Sortierung der Ergebnisse. `Sort`[8] ist eine Klasse, die keine Konstruktoren hat. Stattdessen bauen statische `by(…)`-Fabrikmethoden die Objekte auf:

- `Sort.by(String... properties)`
- `Sort.by(Order... orders)`
- `Sort.by(List<Order> orders)`
- `Sort.by(Direction direction, String... properties)`

Schauen wir uns einige Beispiele zum Aufbau eines `Sort`-Objekts an:

```
Sort sort = Sort.by( "manelength" ).ascending();
```

Die einfachste Möglichkeit ist, der `by(…)`-Methode den Namen des persistenten Attributs zu übergeben. Obacht: Bei der Jakarta Persistence wird *nicht* der Spaltenname angegeben, sondern der Name des persistenten Attributs. Das angehängte `ascending()` bräuchten wir nicht, denn der Standard ist eine aufsteigende Sortierung.

Es sind mehrere Sortierkriterien möglich. Das heißt, wenn die ersten Elemente gleichwertig sind, kann ein zweites Sortierkriterium die Reihenfolge bestimmen. Eine Schreibweise ist diese:

```
Sort sort =       Sort.by( "manelength" ).ascending()
           .and( Sort.by( "lastseen"   ).descending() );
```

Mit `.and()` wird ein anderes `Sort`-Objekt angehängt. Das Ergebnis wäre eine aufsteigende Sortierung nach den Mähnenlängen, und wenn zwei Profile die gleichen Mähnenlängen haben, wird als zweites Kriterium `lastseen` genommen, und zwar absteigend.

Die weitere Möglichkeit zur Initialisierung von `Sort`-Objekten bieten `Order`-Objekte[9]:

```
Sort sort = Sort.by( Sort.Order.asc(  "manelength" ),
                     Sort.Order.desc( "lastseen" ) );
```

8 *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/Sort.html*

9 *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/Sort.Order.html*

Die Klasse `Order` ist ein geschachtelter Typ von `Sort`. Statische Methoden wie `by(…)`, `asc(…)` (für *ascending*, aufsteigend) und `desc(…)` (für *descending*, absteigend) erzeugen `Order`-Objekte. Die `Order`-Objekte füllen das Vararg der `Sort`-Methode `by(Order...)`.

Das generierte SQL ist nicht weiter auffällig. Hibernate fügt Folgendes an das SQL an:

```
order by p1_0_.manelength asc, p1_0_.lastseen desc
```

Der große Vorteil ist, dass wir das `Sort`-Objekt dynamisch aufbauen können; man denke nur an eine Tabellenkalkulation und Spalten, die man für die Sortierung anklickt. Unser JPQL-String ändert sich nicht. Wäre die Sortierung im JPQL-String hart kodiert, wäre das wenig flexibel. Doch die Methode `findAll(…)` erlaubt mit dem `Sort`-Objekt die dynamische Änderung der Reihenfolge.

Fehler bei falschen Property-Namen

Der Jakarta Persistence Provider prüft beim `Sort`-Objekt, ob es sich um ein gültiges persistentes Attribut handelt. Wird ein `Sort`-Objekt mit einem fehlerhaften Namen aufgebaut, folgt eine `PropertyReferenceException`. Nehmen wir an, es heißt:

```
profiles.findAll( Sort.by( "len" ) );
```

Dann wird Spring Folgendes melden:

```
Caused by: org.springframework.data.mapping.PropertyReferenceException: ↩
No property 'len' found for type 'Profile'!
```

Das stimmt, denn es hätte `manelength` heißen müssen. Das Beispiel macht deutlich, dass beliebige Ordnungskriterien nicht möglich sind, die in SQL prinzipiell möglich wären.

Dass die persistenten Attribute bei der Sortierung mit `Sort` als String angegeben werden müssen, ist ein kleines Ärgernis. Es gibt verschiedene Möglichkeiten, von diesen Strings wegzukommen. Einen Ansatz bietet Spring Data mit einem weiteren Datentyp `TypedSort`, und ein weiterer ist Thema von Abschnitt 7.10.1, »Querydsl«.

7.4.2 Sort.TypedSort<T>

Die Klasse `TypedSort`[10] drückt über den Typnamen schon aus, dass es um eine getypte Sortierung geht. `TypedSort` ist ein geschachtelter Typ und wird exemplarisch wie folgt aufgebaut:

```
Sort.TypedSort<Profile> sortedProfile = Sort.sort( Profile.class );
```

10 *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/Sort.TypedSort.html*

Die Idee hinter `TypedSort` ist clever, und zwar müssen wir auf der Entity-Bean die Getter aufrufen. Ein Getter wiederum ist an das persistente Attribut geknüpft; würde es eine Umbenennung geben, würde das im Quellcode auffallen.

```
Sort sort = sortedProfile.by(
    ( Profile profile ) -> profile.getManelength()
);
```

Der Lambda-Ausdruck lässt sich in eine Methodenreferenz umschreiben. Verknüpfen wir zwei Sortierkriterien mit `and(…)`, und führen wir die Sortierung aus:

```
Sort.TypedSort<Profile> sortedProfile = Sort.sort( Profile.class );
Sort sort = sortedProfile.by( Profile::getManelength )
    .and( sortedProfile.by( Profile::getLastseen ).descending() );
List<Profile> sortedProfiles = profiles.findAll( sort );
```

Spring Data JPA generiert für das `Profile-Class`-Objekt einen Proxy, auf dem wir den Getter aufrufen. Der Proxy fängt den Methodenaufruf ab und weiß dadurch, welche persistenten Properties das Sortierkriterium bilden.

7.4.3 Pageable und PageRequest

Wir haben gerade die erste Funktion von [`List`]`PagingAndSortingRepository` besprochen, nämlich dass die `findAll(…)`-Methode mit einem `Sort`-Parameter überladen ist, der die Ergebnisse sortiert.

Die zweite Methode aus dem [`List`]`PagingAndSortingRepository` ist `findAll(Pageable)`. Der Typ `Pageable`[11] ist eine Schnittstelle, und die Implementierung ist `PageRequest`[12] (siehe Abbildung 7.8).

Es gibt zwei Möglichkeiten zum Aufbau von `Pageable`-Objekten: zum einen über statische `of*(…)`-Methoden der implementierenden Klasse `PageRequest`, und zum anderen hat `Pageable` auch eine statische Fabrikmethode, die im Hintergrund ein `PageRequest`-Objekt aufbaut:

- `PageRequest.of(int page, int size)`
- `PageRequest.of(int page, int size, Sort sort)`
- `PageRequest.of(int page, int size, Direction direction, String... properties)`
- `PageRequest.ofSize(int pageSize)` → `ofSize(0, pageSize)`
- `Pageable.ofSize(int pageSize)` → `PageRequest.of(0, pageSize)`

11 *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/Pageable.html*

12 *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/PageRequest.html*

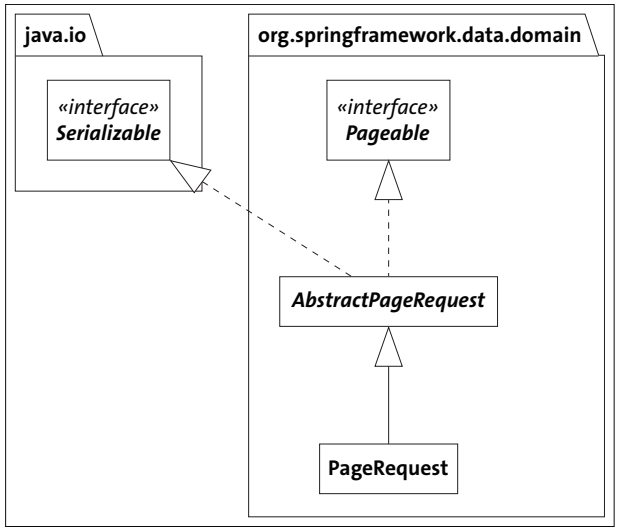


Abbildung 7.8 Typbeziehungen von »Pageable« und Implementierung

Im Wesentlichen gilt es, zwei Informationen anzugeben: die Seite und die Anzahl der Elemente auf der Seite. Bei der Paginierung ändert sich normalerweise die Anzahl der Elemente pro Seite nicht, sondern nur die Seite ändert sich. Außerdem lassen sich bei einigen `of(…)`-Methoden Sortierkriterien mitgeben, entweder über ein `Sort`-Objekt oder über eine `Direction` mit einem Vararg für die Namen der persistenten Attribute.

Die statische Methode `PageRequest.ofSize(pageSize)` ist eine Abkürzung für das Beziehen der ersten Seite.

Um folglich die erste Seite mit zehn Elementen zu selektieren, lässt sich zum Beispiel schreiben:

```
Pageable pageable = PageRequest.of( 0 /* page */, 10 /* size */ );
```

Die Rückgabe ist ein `Pageable`-Objekt, das in die `findAll(Pageable)`-Methode geht.

Page

Es mag überraschen, dass die Methode nicht wie die anderen `findAll(…)`-Methoden direkt die Ergebnisse liefert, sondern ein `Page`-Objekt[13]:

```
Page<Profile> page1 = profiles.findAll( PageRequest.of( 0, 10 ) );
// page1.toString() -> "Page 1 of … containing … instances"
List<Profile> content = page1.getContent();
```

13 *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/Page.html*

Den Inhalt der Seite liefert die Methode `getContent()`. Das mag seltsam erscheinen, hat aber einen logischen Grund.

Seiten ablaufen

Ein `Page`-Objekt ist wie ein `Iterator`, mit dem man durch die Seiten laufen kann. Man kann navigieren, also auf die nächste oder vorhergehende Seite kommen. Auch beantwortet die `Page` die Frage, ob sie die erste oder letzte Seite ist. Alle diese Navigationsmöglichkeiten stecken in dem `Page`-Objekt bzw. im Basistyp. Ein Beispiel:

```
Page<Profile> page1 = profiles.findAll( PageRequest.ofSize( 10 ) );
Page<Profile> page2 = profiles.findAll( page1.nextPageable() );
```

Von der Seite navigiert `nextPageable()` auf die nächste Seite. Die Methode `nextPageable()` liefert ein `Pageable` zurück, das wieder an `findAll(…)` übergeben werden kann.

Mit dieser Konstruktion ist es etwa möglich, über alle Seiten mit einer bestimmten Größe zu laufen:

```
Pageable pageRequest = Pageable.ofSize( 10 );
Page<Profile> profilePage;
do {
  profilePage = profiles.findAll( pageRequest );
  log.info( "{}", profilePage.getContent() );
  pageRequest = pageRequest.next();
} while ( ! profilePage.isLast() );
```

Das ist nicht wahnsinnig performant (in Abschnitt 7.4.5, »Performanceüberlegungen«, folgt noch mehr zu diesem Thema), aber technisch möglich.

Typbeziehungen von Page

Schauen wir uns die Typbeziehungen von `Page` an (siehe Abbildung 7.9).

Der Datentyp kommt aus Spring Data Commons und wird nicht nur für Jakarta-Persistence-Repositories eingesetzt, sondern auch für die anderen Spring-Data-Projekte.

Die Schnittstelle `Page` erweitert die Schnittstelle `Slice`, die ein `Streamable` ist, und ein `Streamable` ist ein `Iterable` und ein `Supplier`. Der Datentyp `Page` hat nicht so viele Methoden:

- `static <T> Page<T> empty()`
- `static <T> Page<T> empty(Pageable pageable)`
- `long getTotalElements()`
- `int getTotalPages()`
- `<U> Page<U> map(Function<? super T,? extends U> converter)`

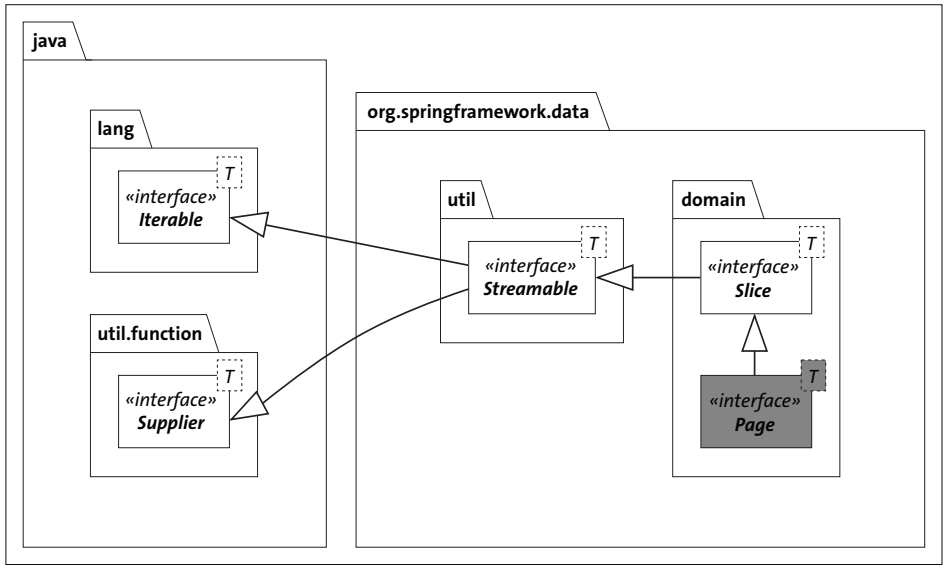


Abbildung 7.9 UML-Diagramm der Typbeziehungen von »Page«

Die Methode `getTotalElements()` liefert die Gesamtanzahl aller Elemente, nicht nur die Anzahl der Elemente der Seite. Zur Ermittlung der Gesamtzahl muss der Jakarta Persistence Provider eine zweite `SELECT`-Anweisung ausführen, um sich mit `COUNT` diese Anzahl zu holen. `getTotalPages()` liefert die Anzahl der Seiten. Die Anzahl der Seiten berechnet sich durch `getTotalElements()`, geteilt durch die aktuelle Seitengröße.

Im Scheibchentyp `Slice` kommen deutlich mehr Methoden hinzu:

- `List<T> getContent()`
- `int getNumber()`
- `int getNumberOfElements()`
- `default Pageable getPageable()`
- `int getSize()`
- `Sort getSort()`
- `boolean hasContent()`, `boolean hasNext()`, `boolean hasPrevious()`, `boolean isFirst()`, `boolean isLast()`
- `Pageable nextPageable()`, `Pageable previousPageable()`
- `default Pageable nextOrLastPageable()`, `default Pageable previousOrFirstPageable()`
- `<U> Slice<U> map(Function<? super T,? extends U> converter)`

Der Datentyp `Slice` hat die wichtige Methode `getContent()` und weitere Methoden zum Erfragen von Größen. Es gibt Methoden für Existenztests und Navigationsmethoden. Praktisch zum Navigieren sind die Default-Methoden `nextOrLastPageable()` und `previousOrFirstPageable()`, denn steht man auf der ersten oder letzten Seite, lässt sich nicht weiterlaufen.

Eine Methode `map(…)` hat die Schnittstelle `Slice` ebenso. Damit lassen sich Objekte aus dem Teilstück problemlos in neue Objekte transformieren.

Die Schnittstelle `Slice` selbst erweitert die Schnittstelle `Streamable`. Dieser Datentyp erlaubt es uns, direkt ausgewählte `Stream`-Methoden aufzurufen. Wir hatten diesen Datentyp schon früh in Abschnitt 3.9 besprochen.

Aufgabe: Durch die Profile navigieren

In der interaktiven Anwendung soll man durch alle Profile der Datenbank navigieren können. Dazu soll `RepositoryCommands` erweitert werden und einen Zustand für die aktuelle Seite speichern. Das Shell-Kommando `list` soll von der aktuellen Seite 10 Profile anzeigen.

Zwei neue Shell-Kommandos sollen ergänzt werden:

- **`pp`** (*previous page*): Geht auf die vorherige Seite und zeigt 10 Profile an.
- **`np`** (*next page*): Geht auf die nächste Seite und zeigt 10 Profile an.

Der Lösungsvorschlag:

Es gibt verschiedene Ansätze, die Aufgabe zu lösen. Wir können uns intern die Seite als Ganzzahl merken (Seite 1, Seite 2 …) oder als `Page`-Objekt. Wenn wir uns die Seite als Zahl merken, hat das zwei Nachteile:

- Wir müssen Daten immer wieder neu laden, wenn mehrmals hintereinander `list` aufgerufen wird – was auch von Vorteil sein kann, weil die Daten dann immer frisch sind –, und
- wir können nicht von den schönen `Slice`-Navigationsmethoden Gebrauch machen.

Daher wählen wir einen anderen Lösungsansatz. Wir wollen uns die `Page` merken und damit navigieren. Aber an welcher Stelle baut man die `Page` auf? Man könnte eine Objektvariable nutzen, die mit `null` so lange belegt bleibt, bis das erste Mal `list` oder ein Kommando `pp`/`np` aufgerufen wird. Oder man könnte die `Page` immer im Konstruktor aufbauen. Beides ist nicht perfekt: `null`-Prüfungen sind fehleranfällig, und in Vorleistung wollen wir auch nicht gehen. Was wir benötigen, ist eine Art Behälter, der immer dann gefüllt wird, wenn es den ersten Zugriff gibt. Spring Data Commons hat so eine Klasse, die wir in Abschnitt 3.11.3 schon kennengelernt haben: `Lazy`.

Damit kann die Lösung so aussehen:

Listing 7.2 RepositoryCommands.java

```
@ShellComponent
public class RepositoryCommands {
  private static final int PAGE_SIZE = 10;
  private final ProfileRepository profiles;
  private Lazy<Page<Profile>> currentPage;

  public RepositoryCommands( ProfileRepository profiles ) {
    this.profiles = profiles;
    currentPage =
      Lazy.of( () -> profiles.findAll(PageRequest.ofSize(PAGE_SIZE)) );
  }

  @ShellMethod( "Display all profiles" )
  public List<Profile> list() {
    return currentPage.get().getContent();
  }

  @ShellMethod( "Set current page to previous page, display the current page" )
  List<Profile> pp() {
    currentPage = currentPage.map(
                page -> profiles.findAll( page.previousOrFirstPageable() ) );
    return list();
  }

  @ShellMethod( "Set current page to next page, display the current page" )
  List<Profile> np() {
    currentPage = currentPage.map(
                    page -> profiles.findAll( page.nextOrLastPageable() ) );
    return list();
  }
}
```

Das `Lazy`-Objekt wird mit einem `Supplier` versorgt, der die Daten holt. Wenn es einen Zugriff gibt, sei es durch das Kommando `list` oder durch die `map(…)`-Methoden, dann wird der `Supplier` aufgerufen, und die Daten werden einmalig besorgt und im `Lazy`-Objekt abgelegt.

Die `map(…)`-Methode hat den Vorteil, aus einem `Lazy`-Objekt ein neues `Lazy`-Objekt zu machen, wobei der Inhalt transformiert wird. Der `Supplier` wird aber auch erst dann gefragt, wenn es wirklich einen Zugriff gibt – den wird es geben, denn es folgt nach `pp()`/`np()` ein Aufruf von `list()`.

7.4.4 Sortieren von paginierten Seiten

Ein `Pageable`-Objekt lässt sich auch mit einem Sortierkriterium aufbauen. Dann wird erst die Datenmenge sortiert und anschließend paginiert. Es ist also nicht so, dass Spring erst diese Seite heraussucht und nur diese sortiert.

Beim Aufbau der Sortierkriterien können wir auf das Wissen aus dem vorherigen Abschnitt zurückgreifen. Schauen wir uns drei Beispiele an:

```
Pageable sortedByManelength =
   PageRequest.of( 2, 10, Sort.by("manelength") );

Pageable sortedByManelengthDesc =
   PageRequest.of( 2, 10, Sort.by("manelength").descending() );

Pageable sortedByManelengthAndLastseen =
   PageRequest.of( 2, 10, Sort.by( Sort.Order.asc("manelength"),
                           Sort.Order.desc("lastseen") ) );
```

Natürlich wird erst sortiert und dann paginiert.

7.4.5 Performanceüberlegungen

Für die Navigation durch Datensätze haben wir haben wir das `PagingAndSortingRepository` kennengelernt. Der Code für die Seitennavigation ist elegant und schnell geschrieben. Spring Data JPA realisiert die Paginierung über `LIMIT` (`FETCH` zählen wir dazu) und `OFFSET`.[14] Die Technik heißt *Offset-basierte Pagination* (auch *Page-based Pagination*). So einfach das Ganze auch sein mag, es hat unter Umständen einige Kosten in der Performance. Das Problem ist, dass eine Sortierung in Zusammenhang mit `LIMIT` und `OFFSET` ziemlich teuer ist. Bei nicht sortierten Mengen ist das noch in Ordnung, aber eine Sortierung macht `LIMIT` und `OFFSET` teuer. Es ist schon gemein: Die Datenbank muss die Daten erst sortieren, dann werden vielleicht viele Daten weggeschmissen. Das ist ungünstig. Deswegen sollte man sich gut überlegen, ob das `PagingAndSortingRepository` wirklich der richtige Datentyp ist.

Neben der Offset-basierten Paginierung bietet die *Keyset-basierte Paginierung* eine Alternative zur effizienten Navigation durch umfangreiche Datensätze. Diese Methode nutzt eindeutige oder sequenzielle Schlüsselwerte – typischerweise IDs oder Zeitstempel – als Referenzpunkte für die Abfrage nachfolgender Daten. Der Schlüsselwert des letzten Elements einer Ergebnisseite dient als Ausgangspunkt für die Ab-

14 Intern sendet `SimpleJpaRespository` über die Criteria API eine Abfrage mit einer Paginierung ab; JPQL oder SQL wird nicht direkt aufgebaut: *https://github.com/spring-projects/spring-data-jpa/blob/main/spring-data-jpa/src/main/java/org/springframework/data/jpa/repository/support/SimpleJpaRepository.java*

frage der nächsten Seite. Dieser Ansatz vermeidet das `OFFSET` in SQL-Abfragen und bietet zusätzliche Stabilität bei gleichzeitigen Datenänderungen, da neue Einträge die Paginierung laufender Abfragen nicht beeinflussen. Ein Nachteil ist, dass ein direkter Zugriff auf bestimmte Seitenzahlen nur über Umwege möglich ist. Die Methode eignet sich daher besonders gut für Anwendungsfälle, in denen hauptsächlich sequenziell durch die Daten navigiert wird, also vorwärts oder rückwärts zur jeweils nächsten oder vorherigen Seite.

Ein weiterer Nachteil der Keyset-basierten Paginierung ist, dass die API Schlüsselparameter offenlegen muss, die direkt auf bestimmte Spalten in der Datenbank verweisen. Diese Informationen müssen bei jeder Anfrage übergeben werden, obwohl sie eigentlich ein Implementierungsdetail sein sollten. In Webservices wird daher häufig die *Cursor-basierte Pagination* verwendet. Dabei bezeichnet der Begriff *Cursor* (auch *Pointer* genannt) einen kodierten String, der die aktuelle Position im Datensatz enthält. Der Cursor wird bei jeder Anfrage zurückgegeben und dient als Ausgangspunkt für die nächste Abfrage, um die folgenden Ergebnisse abzurufen. Typischerweise wird der Cursor-String in Base64 kodiert, um Implementierungsdetails zu verbergen und den Wert kompakt zu halten. Dadurch kann die Paginierungsstrategie flexibel angepasst werden, ohne dass die Clients die Unterschiede bemerken oder Änderungen vornehmen müssen. Intern kann diese Methode auf Keyset-basierte Paginierung abgebildet werden oder auf eine andere Technik, was für den Client jedoch transparent bleibt.

Die Spring-Bibliotheken unterstützen Keyset-basierte und Cursor-basierte Paginierung nicht »out-of-the-box«, wie es bei Offset-basierter Paginierung der Fall ist. Daher müssen wir diese Paginierungstechniken selbst implementieren. Seit Spring Boot 3.1 kommt Offset- or Keyset-based Scrolling hinzu, aber die API-Unterstützung ist erst in den Anfängen.[15]

7.5 QueryByExampleExecutor *

In diesem Abschnitt wollen wir uns mit einem weiteren Basistyp von Spring Data beschäftigen: `QueryByExampleExecutor`. Auch dieser Typ bietet vordefinierte Methoden, wie das [`List`]`CrudRepository` oder das `JpaRepository`. Die Funktion des Typs ist, eine Suche zu erlauben, so wie Menschen eher an Beispielen beschreiben, was sie suchen:

- Gesucht ist *ein Buch*, dessen *Titel* »*Java*« enthält, das auf *Deutsch* ist und *länger als 10 Jahre* auf dem Markt ist.
- Gesucht sind alle *Profile*, die in den *vergangenen 3 Monaten gesehen* wurden und *älter als 100 Jahre* sind.

15 *https://docs.spring.io/spring-data/commons/docs/current/reference/html/#repositories.scrolling*

Anfragen dieser Art nennt man *Probe*. Über die Probe kann die Bibliothek eine Abfrage für die Datenbank formulieren, die Ergebnisse liefert, die zu der Probe passen.

Genau darum geht es bei dem `QueryByExampleExecutor`.

7.5.1 Die Probe

Nehmen wir an, wir möchten bestimmte `Profiles` suchen. Dann bauen wir zuerst eine Probe auf:

```
Profile p = new Profile( null, null,
                         /* manelength = */ 10,
                         /* gender = */     0,
                          null, null, null );
```

Fast alle Eigenschaften sind über den Konstruktor auf `null` gesetzt, nur die gewünschte Mähnenlänge soll `10` und das gesuchte Gender `0` sein.

7.5.2 QueryByExampleExecutor

Der Datentyp, der zu einer Probe entsprechende Ergebnisse liefern kann, ist `QueryByExampleExecutor`[16]. Unsere Repository-Schnittstelle muss dazu `QueryByExampleExecutor` erweitern, und es gibt zwei Optionen:

- **Variante 1**: `JpaRepository` erweitert `QueryByExampleExecutor`, und die Suchmethoden sind vorhanden:

  ```
  interface ProfileRepository extends JpaRepository<Profile, Long> { }
  ```

  Das heißt: Wenn ein Repository `JpaRepository` erweitert – wie wir das gemacht haben –, muss `QueryByExampleExecutor` nicht erneut bei `extends` auftauchen.

- **Variante 2**: Erweitert das eigene Repository nur [`List`]`CrudRepository`, so muss von `QueryByExampleExecutor` ebenfalls abgeleitet werden, denn [`List`]`CrudRepository` erweitert `QueryByExampleExecutor` nicht:

  ```
  interface ProfileRepository extends CrudRepository<Profile, Long>,
                                      QueryByExampleExecutor<Profile> { }
  ```

  Natürlich ist es zulässig, wenn das `ProfileRepository` auch nur von `QueryByExampleExecutor` erbt.

Bei `QueryByExampleExecutor<Profile>` fällt auf, dass nur das Typargument `Profile` vorkommt, die ID vom Typ `Long` spielt keine Rolle.

16 *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/query/QueryByExampleExecutor.html*

QueryByExampleExecutor-API

Werfen wir einen Blick auf alle Methoden der Schnittstelle `QueryByExampleExecutor`:

```
package org.springframework.data.repository.query;

import …

public interface QueryByExampleExecutor<T> {

  <S extends T> Optional<S> findOne(Example<S> example);
  <S extends T> Iterable<S> findAll(Example<S> example);
  <S extends T> Iterable<S> findAll(Example<S> example, Sort sort);
  <S extends T> Page<S> findAll(Example<S> example, Pageable pageable);
  <S extends T,R> R findBy(Example<S> example,
    Function<FluentQuery.FetchableFluentQuery<S>,R> queryFunction);
  <S extends T> long count(Example<S> example);
  <S extends T> boolean exists(Example<S> example);
}
```

In der Parameterliste ist überall der Datentyp `Example`[17] zu finden. Ein `Example`-Objekt kapselt die Probe.

7.5.3 Probe in das Example

Die `Example`-Objekte baut eine Fabrikmethode `of(…)` auf:

```
Profile p = new Profile( null, null, /* manelength */ 10, /* gender */ 0,
                         null, null, null );
Example<Profile> example = Example.of( p );
```

Das `Example`-Objekt mit der Probe wird im nächsten Schritt den Methoden von `QueryByExampleExecutor` übergeben:

```
List<Profile> result = profiles.findAll( example );
```

Ein `Example`-Objekt ist immutable.

Die Ausführung erzeugt im Hintergrund SQL mit einer `WHERE`-Klausel, die so aussieht:

```
where p1_0.manelength=10 and p1_0.gender=0
```

Es lässt sich ablesen, dass Spring die Informationen aus der Probe nimmt und daraus die `WHERE`-Klausel formuliert; abzulesen ist die Prüfung auf die Mähnenlänge `10` und

17 *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/Example.html*

Gender `0`. Die Anfragen werden automatisch Und-verknüpft. Das lässt sich über einen `ExampleMatcher` anpassen.

Die `of(…)`-Methode zum Aufbau der `Example`-Objekte ist überladen, zusätzlich lässt sich ein `ExampleMatcher` übergeben (siehe Abbildung 7.10).

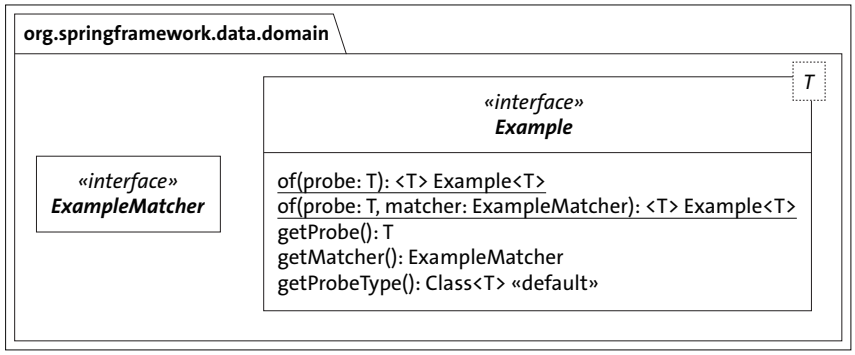


Abbildung 7.10 Operationen in »Example«

7.5.4 ExampleMatcher aufbauen

Die Schnittstelle `ExampleMatcher`[18] bietet statische Fabrikmethoden zum Aufbau der Instanzen:

- **`matchingAll()`**, alternativ **`matching()`**: Alle Werte müssen – wie in einer Und-Verknüpfung – mit der Probe übereinstimmen. Das ist der Standard.
- **`matchingAny()`**: Es reicht, wenn einzelne Werte übereinstimmen, wie in einer Oder-Verknüpfung.

Folgende Zeilen sind gleichbedeutend:

```
var example = Example.of( p );
var example = Example.of( p, ExampleMatcher.matching() );
var example = Example.of( p, ExampleMatcher.matchingAll() );
```

Die `ExampleMatcher`-Exemplare sind immutable. Über eine Fluent-API lassen sich die Zustände weiter verfeinern; es liefern also alle Methoden wieder `ExampleMatcher` zurück.

Oder-Verknüpfung mit ExampleMatcher

Nehmen wir an, wir möchten wissen, welche Profile entweder die Mähnenlänge `10` oder das Gender `0` haben:

18 *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/ExampleMatcher.html*

```
Profile p = new Profile( null, null, 10, 0, null, null, null );
ExampleMatcher matcher = ExampleMatcher.matchingAny();
Example<Profile> example = Example.of( p, matcher );
```

Beim `Example`-Objekt wird für die Oder-Abfrage `ExampleMatcher.matchingAny()` übergeben.

Im generierten SQL wird statt `AND` ein `OR` stehen.

7.5.5 Properties ignorieren

Während `null`-Referenzen bei einer Abfrage ignoriert werden, sind primitive Datentypen immer belegt und Teil der Abfrage. Es ist aber wichtig, gewisse persistente Attribute zu ignorieren. Oder wie können wir ausdrücken, dass das Geschlecht egal ist?

```
Profile p = new Profile( null, null, /* manelength */ 10, /* gender */ ????,
                         null, null, null );
```

Es lässt sich `0` nicht übergeben, und `null` ist ebenfalls nicht erlaubt. Über die Probe ist es nicht möglich, primitive Werte zu ignorieren. Der `ExampleMatcher` bietet eine Lösung.

ExampleMatcher + withIgnorePaths(...)

Der `ExampleMatcher` bietet die Möglichkeit, bestimmte Attribute aus der SQL-Abfrage auszuschließen. Diese tauchen dann nicht in der `WHERE`-Klausel auf. Soll ein persistentes Attribut nicht betrachtet werden, so nutzt man `withIgnorePaths(String... ignoredPaths)`:

```
Profile p = new Profile( null, null, /* manelength */ 10, 0,
                         null, null, null );
var matcher = ExampleMatcher.matching().withIgnorePaths( "gender" );
Example<Profile> example = Example.of( p, matcher );
List<Profile> result = profiles.findAll( example );
```

In der Profilprobe wird zwar `gender` auf `0` gesetzt, aber im Grunde ist der Wert egal, er wird ignoriert.

Hinweis

Referenzvariablen, die auf `null` gesetzt sind, werden von der Abfrage standardmäßig ignoriert. Möchte man dennoch, dass `null` als Abfragekriterium einbezogen wird, kann dies mit `withIncludeNullValues()` geändert werden.

Mit dem `ExampleMatcher` kann man auch noch ein wenig mehr machen.

7.5.6 String-Vergleichs-Techniken setzen

Strings werden standardmäßig vollständig verglichen, müssen also Zeichen für Zeichen übereinstimmen. Das lässt sich steuern, denn es muss auch die Frage möglich sein, ob ein Teilstring vorkommt oder ob die Groß-/Kleinschreibung keine Rolle spielt.

Die Eigenschaft lässt sich mit einem `StringMatcher` steuern, der bei der `ExampleMatcher`-Methode `withStringMatcher(StringMatcher)` übergeben wird. Der Parametertyp `StringMatcher`[19] ist ein Aufzählungstyp mit den Konstanten `EXACT`, `STARTING`, `ENDING`, `CONTAINING` und `REGEX`; der Standard ist `DEFAULT`.

Die Methode `withStringMatcher(…)` setzt das Verhalten für alle String-Properties. Wir werden gleich sehen, wie sich das individuell für die verschiedenen persistenten Attribute steuern lässt.

Eine weitere Methode ist `withIgnoreCase(String... propertyPaths)`; sie hilft, die Groß-/Kleinschreibung zu ignorieren. Es werden die Namen der persistenten Attribute übergeben, und die Methode liefert einen `ExampleMatcher` zurück, der eine Fluent-API erlaubt. Schauen wir uns dazu ein Beispiel an:

```
final int IGNORE = 0;
Profile p = new Profile( "st", null, IGNORE, IGNORE, null, null, null );

ExampleMatcher matcher = ExampleMatcher.matching()
    .withIgnorePaths( "manelength", "gender" )
    .withStringMatcher( ExampleMatcher.StringMatcher.CONTAINING )
    .withIgnoreCase();

Example<Profile> example = Example.of( p, matcher );
```

Das Ziel ist es, alle `Profiles` zu bekommen, die den Nickname `st` enthalten. Die persistenten Attribute `manelength` und `gender` sollen beide komplett ignoriert werden. Um das etwas lesbarer zu machen, gibt es im Programm eine Konstante, die mit jedem Wert belegt werden könnte.

7.5.7 Bewertung von Query-By-Example (QBE)

Der `QueryByExampleExecutor` hat Vor-, aber eher Nachteile. Einer seiner Vorteile ist auf jeden Fall, dass Refactorings gut überstanden werden, denn die Abfragen sind relativ frei von Strings (nur bei zu ignorierenden persistenten Attributen). Sollten sich intern Typen oder Bezeichner in der Probe ändern, funktioniert die Abfrage vermutlich wie

19 *https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/ExampleMatcher.StringMatcher.html*

vorher. Zudem wird der QueryByExampleExecutor von diversen Datenspeichern unterstützt – ob relationale Datenbank oder MongoDB, beide unterstützen QueryByExampleExecutor.

Daneben gibt es eine Reihe von Nachteilen. Auch wenn unterschiedliche Spring-Data-Implementierungen prinzipiell QueryByExampleExecutor unterstützen, heißt das nicht, dass jedes Feature der API unterstützt wird, etwa ein StringMatcher mit REGEX. Ein weiterer Haken ist, dass Bereichsangaben nicht möglich sind. Bei der Mähnenlänge können wir nicht prüfen, welche Profile eine Mähnenlänge zwischen 1 und 100 haben; das ist für Prüfungen aber angebracht. Zudem sind aktuell keine verschachtelten Abfragen der Art »Mähnenlänge ist X oder (Geburtstag am YY und Geschlecht = Z)« möglich.

Da die Nachteile überwiegen, spielt der QueryByExampleExecutor in der Praxis in dieser Form keine besonders große Rolle.

7.6 Eigene Abfragen mit @Query formulieren

Die Methoden, die das [List]CrudRepository bietet, sind praktisch. Wir haben wichtige Methoden zum Beispiel zum Ermitteln aller Datensätze, zum Speichern mehrerer Entity-Beans und zum Löschen von Datensätzen nach IDs. Das ist mehr, als der EntityManager bietet. Allerdings reichen diese vorgefertigten Methoden in der Praxis nicht aus. Ein echtes Repository stellt über Methoden Anfragen nach ausgewählten Daten zur Verfügung. Profile sollen sicherlich nach Kriterien gefiltert werden können – oder Unicorns nach E-Mail-Adressen. So etwas lässt sich mit dem [List]CrudRepository nicht realisieren. Wieder zum EntityManager zurückkehren muss man im Allgemeinen nicht, denn es ist möglich, JPQL oder native Queries im eigenen Repository zu ergänzen.

7.6.1 Die @Query-Annotation

Spring Data bietet im Paket eine @Query-Annotation[20], die an eigene neue Methoden in der Repository-Schnittstelle gesetzt wird. Die @Query enthält über ein Annotationsattribut die Abfrage in der nativen Datenbanksprache, etwa JPQL, SQL oder MongoDB-Queries. Während [List]CrudRepository und [List]PagingAndSortingRepository vom Datenspeicher unabhängig sind, sind es die Methoden mit den @Query-Annotationen nicht mehr. Das heißt, die Abfragen sind datenbankspezifisch. Wer das Datenbankmanagementsystem wechselt, muss die Queries vermutlich anpassen.

20 *https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/Query.html*

Nehmen wir an, es sind alle Profile zu finden, die sich für andere Profile beliebigen Geschlechts interessieren. In der Datenbank wird das durch `NULL` bei `attractedToGender` kodiert. Entweder steht für das Interesse in der Spalte `1` oder `2`, oder es ist `NULL` eingetragen, wenn das Profil offen für beide Geschlechter ist. So könnte der Code aussehen:

```
public interface ProfileRepository extends JpaRepository<Profile, Long> {
  @Query( "SELECT p FROM Profile p WHERE p.attractedToGender IS NULL" )
  List<Profile> findBiUnicorns();
}
```

Die neue Methode kommt in das `ProfileRepository` und wird mit `@Query` annotiert. `@Query` stammt aus dem Paket `org.springframework.data.jpa.repository`. Andere Spring-Data-Projekte nutzen die gleiche Annotation, aber aus unterschiedlichen Paketen!

Die Abfrage ist regulär in JPQL formuliert. Bei den Methoden sind unterschiedliche Rückgabetypen möglich. In unserem Fall sind mehrere Profile möglich, und daher ist eine `List` gut; eine `Collection` oder ein `Stream` wäre auch möglich gewesen. Es gibt natürlich Abfragen, die nur zu exakt einem einzigen Profil führen, dann könnte der Rückgabetyp `Profile` sein; falls es kein Ergebnis gibt, wäre die Rückgabe `null`. `Optional<Profile>` wäre vom API-Design üblicher, wenn es keine Rückgaben geben könnte.

@Query-Annotation für parametrisierte Methoden nutzen

Es können Parameter in die `@Query`-Methode aufgenommen werden, und dazu wird die Methode parametrisiert. Es gibt zwei Möglichkeiten: Positionsparameter oder benannte Parameter. Schauen wir uns die gleiche Abfrage mit den zwei Möglichkeiten an. Eine Abfrage soll die Profile auflisten, die nach einem gewissen Stichtag gesehen wurden.

Zur ersten Option, den *Positionsparametern*:

```
@Query( "SELECT p FROM Profile p WHERE p.lastseen > ?1" )
List<Profile> findProfilesLastSeenAfter( LocalDateTime timestamp );
```

Das JPQL schränkt mit einer `WHERE`-Klausel die Ergebnisse ein, in denen `profilesLastSeen` größer als der übergebene Wert sein muss. Das `?` ist der Platzhalter für das Prepared Statement und wird mit dem gefüllt, was der Methode übergeben wurde. Der Positionsparameter wird auch *Indexparameter* genannt.

Eine alternative Schreibweise ist, sich mit einem Doppelpunkt auf einen Bezeichner zu beziehen. Das nennt sich *benannter Parameter*:

```
@Query( "SELECT p FROM Profile p WHERE p.lastseen > :lastseen" )
List<Profile>
   findProfilesLastSeenAfter( LocalDateTime lastseen );
// findProfilesLastSeenAfter( @Param( "lastseen" ) LocalDateTime timestamp );
```

In JPQL heißt der mit einem Doppelpunkt geschriebene Bezeichner genauso wie die Parametervariable `lastseen`. Wenn die Parametervariable anders lauten sollte, etwa `timestamp`, muss die Annotation `@Param` den benannten Parameter in dem JPQL-String bestimmen.

Aufgabe: Neue @Query-annotierte Methoden einführen

Wer üben möchte, kann folgende drei Methoden mit entsprechender `@Query` implementieren:

- `findProfileByNickname(String name)`
- `findProfilesByContainingName(String name)`
- `findProfilesByManelengthBetween(short min, short max)`

Lösungsvorschlag:

```
@Query( "SELECT p FROM Profile p WHERE p.nickname = :name" )
Optional<Profile> findProfileByNickname( String name );

@Query( "SELECT p FROM Profile p WHERE p.nickname LIKE %:name%" )
List<Profile> findProfilesByContainingName( String name );

@Query( "SELECT p FROM Profile p WHERE p.manelength BETWEEN :min AND :max" )
List<Profile> findProfilesByManelengthBetween( short min, short max );
```

7.6.2 Verändernde @Query-Operationen mit @Modifying

Ein JPQL-Ausdruck kann neben `SELECT` auch ein `UPDATE` oder `DELETE` enthalten. Wir hatten in Abschnitt 6.8.9, »Query mit UPDATE und DELETE«, im Zusammenhang mit JPQL-Ausdrücken gesehen, dass das üblicherweise für Batch-Aktualisierungen eingesetzt wird. Wenn man `UPDATE`- oder `DELETE`-Methoden in JPQL-Strings über den `EntityManager` absendet, dann muss eine andere `EntityManager`-Methode benutzt werden als bei `SELECT`-Abfragen – `executeUpdate(…)` steht den Methoden `getResultList()`, `getResultStream()` bzw. `getSingleResult()` gegenüber. Da Spring Data JPA aus dem String nicht ablesen kann, ob es ein `SELECT`, `UPDATE` oder `DELETE` ist, muss bei verändernden Operationen zusätzlich `@Modifying` an der `@Query`-Methode ergänzt werden.

Schauen wir uns ein Beispiel an. Der Zeitstempel `lastseen` soll für ein Profil mit einer gewissen ID aktualisiert werden:

```
@Modifying
@Query( "UPDATE Profile p SET p.lastseen =:lastseen WHERE p.id = :id" )
int updateLastSeen( long id, LocalDateTime lastseen );
```

Das Beispiel greift auf zwei benannte Parameter zurück. Das zu aktualisierende Profil wird über die `id` identifiziert, und `lastseen` wird wieder als `LocalDateTime` übergeben.

Ruft man `@Modifying`-Operationen von außen auf, sind es Veränderungen, und diese müssen in einem transaktionalen Kontext stattfinden. Schematisch sieht das so aus:

```
@Transactional
@Override public void run() {
  long id = …
  profiles.updateLastSeen( 1, LocalDateTime.now() );
}
```

7.6.3 IN-Parameter durch Array/Vararg/Collection füllen

JPQL – und auch SQL – erlaubt mit `IN` eine Art kompakte Oder-Abfrage, ob ein persistentes Attribut gewisse Werte in der Datenbank annimmt. Ein Beispiel: Eine Abfrage soll Profile liefern, die sich für bestimmte Geschlechter interessieren:

```
@Query( "SELECT p FROM Profile p WHERE p.attractedToGender IN :genders" )
List<Profile>    findProfilesAttractedToGender( Byte... genders );
// List<Profile> findProfilesAttractedToGender( List<Byte> genders );
// List<Profile> findProfilesAttractedToGender( Byte[] genders );
```

Die Werte hinter `IN` können von der Java-Seite aus über ein Vararg, eine Liste oder über ein Array in die Query gelangen. Neben `IN` gibt es auch die Negation `NOT IN`.

7.6.4 @Query mit JPQL-Projektion

Die mit `@Query` annotierten Methoden haben entweder eine oder mehrere Entity-Beans zurückgegeben. Das muss aber nicht so sein, denn wir haben in Abschnitt 6.6.9, »Projektion auf mehrere Werte«, schon gesehen, dass ein JPQL-Ausdruck auch eine Projektion enthalten kann. Dann ist das Ergebnis keine Entity-Bean mehr, sondern vielleicht ein Ganzzahlwert, ein String oder ein Assoziativspeicher mit Schlüssel-Wert-Paaren. Dazu ein Beispiel: Die `SELECT`-Anweisung soll keine `Profile`-Entity-Beans liefern, sondern nur die ID eines Profils sowie den Nickname:

```
@Query(
 "SELECT p.id AS id,p.nickname AS nickname FROM Profile p WHERE id=:id"
)
Map<String, Object> findSimplifiedProfile( long id );
```

Die Methode `findSimplifiedProfile(…)` liefert eine `Map` mit Schlüssel-Wert-Paaren, wobei die Schlüssel in unserem Fall die `id` und der `nickname` sind und die damit assoziierten Werte ein `Long` und ein `String`. Da die ID nur einmal vorkommen kann, reicht eine `Map` für die »Zeile«.

Wenn mehr Werte zurückkommen, ist das Ergebnis eine Liste von kleinen Assoziativspeichern. Ein weiteres Beispiel: Wir wollen alle Profile selektieren und auch auf ID und Nickname projizieren:

```
@Query("SELECT p.id AS id,p.nickname AS nickname FROM Profile p")
List<Map<String, Object>> findAllSimplifiedProfiles();
```

Ruft man die Methode auf, bekommt man eine `Map` bzw. eine `List` von `Map`s. So könnte man die Inhalte loggen:

```
Map<String, Object> map = profiles.findSimplifiedProfile( 1 );
log.info( "id={}, nickname={}", map.get( "id" ),
                                map.get( "nickname" ) );

List<Map<String, Object>> maps = profiles.findAllSimplifiedProfiles();
maps.forEach( map -> log.info( "id={}, nickname={}",
                               map.get( "id" ), map.get( "nickname" ) ) );
```

7.6.5 Sort- und Pageable-Parameter

`Sort` und `Pageable` sind wichtige Datentypen von Spring Data. `Sort` repräsentiert ein Ordnungskriterium, und `Pageable` beschreibt eine Seite. `Sort`- und `Pageable`-Objekte lassen sich zusätzlich den `@Query`-Methoden übergeben. In diesem Fall wird Spring Data JPA automatisch diese Sortierung beziehungsweise das `Pageable` in die Query integrieren. Der Vorteil ist, dass das Ordnungskriterium flexibel ist und nicht verschiedene `@Query`-Methoden mit hartkodierten Kriterien existieren müssen. Ein Beispiel: Wir wollen alle Profile finden, die nach einem gewissen Zeitpunkt gesehen wurden:

```
@Query( "SELECT p FROM Profile p WHERE p.lastseen > :lastseen" )
List<Profile> findProfilesLastSeenAfter( LocalDateTime lastseen,
                                         Sort sort );
```

Anders als die Abfragen vorher greift der JPQL-Ausdruck nicht auf den Parameter zurück. Spring Data erkennt, dass `Sort` ein besonderer Datentyp ist, und ergänzt das Sortierkriterium im JPQL-String.

Mit `Pageable` werden Seiten und Ausschnitte erfragt. Wir können zur Methode `findProfilesLastSeenAfter(…)` eine Variante ergänzen:

```
@Query( "SELECT p FROM Profile p WHERE p.lastseen > :lastseen" )
// mit getTotalElements()
Page<Profile>  findProfilesLastSeenAfter( LocalDateTime lastseen,
                                          Pageable p );
```

Der Unterschied ist, dass wir neben dem `lastseen` das `Pageable` mit übergeben, das natürlich selbst wieder eine Sortierung erhalten kann.

Wird ein `Pageable` übergeben, sind verschiedene Rückgabetypen erlaubt, und zwar `Page`, `Slice` oder ein Sammlungstyp wie `List`. Ist die Rückgabe ein `Page`-Objekt, ist die Anzahl der Gesamtelemente wichtig und muss über ein eigenes `SELECT COUNT` erfragt werden. Statt einer Abfrage sind bei dem Rückgabetyp `Page` zwei SQL-Abfragen nötig. Bei den Datentypen `Slice` und `List` ist das nicht mehr der Fall.

```
// kennt keine Gesamtanzahl
Slice<Profile> findProfilesLastSeenAfter(LocalDateTime lastseen, Pageable p);
// keine Navigation mehr
List<Profile>  findProfilesLastSeenAfter(LocalDateTime lastseen, Pageable p);
```

Wenn wir ein `Slice` benutzen, gibt es weiterhin die Möglichkeit zur Navigation, allerdings fehlt dem `Slice` die Information über die Gesamtanzahl der Elemente. Wenn wir noch weiter abstrahieren, also bis auf den Datentyp `List`, sind es nur noch die Daten der Abfrage selbst.

Hinweis

Kommen in der Parameterliste `Sort` oder `Pageable` vor, dürfen Aufrufer nicht `null` übergeben. Wenn keine Sortierung oder Paginierungsdetails gewünscht oder bekannt sind, lässt sich `Sort.unsorted()` bzw. `Pageable.unpaged()` nutzen.

JpaSort

Der Datentyp `Sort` hat die Besonderheit, dass er nur mit bekannten persistenten Attributen funktioniert. Andere Ordnungskriterien sind nicht möglich. Es ist aber nützlich, nach anderen Ordnungskriterien zu sortieren, und dafür deklariert Spring Data den Datentyp `JpaSort`[21] , eine Unterklasse von `Sort`. Eine statische Fabrikmethode `JpaSort.unsafe(…)` liefert die `JpaSort`-Objekte. Damit kann etwa eine Funktion als Sortierkriterium eingesetzt werden, wie es das nächste Beispiel zeigt:

```
var result = profiles.findProfilesLastSeenAfter(
                LocalDateTime.now().minusYears( 10 ),
                JpaSort.unsafe( "LENGTH(nickname)" ) );
```

21 *https://docs.spring.io/spring-data/data-jpa/docs/current/api/org/springframework/data/jpa/domain/JpaSort.html*

Das Sortierkriterium ist die Länge der Nicknames. So etwas könnte das `Sort`-Objekt nicht ausdrücken.

Hinweis

Die Klasse `JpaSort` erweitert `Sort` und erbt die `by(…)`-Methode, sodass `JpaSort.by(…)` auch nur ein `Sort`-Objekt liefert. Es ist wichtig, die Methode `unsafe(…)` einzusetzen.

7.6.6 Neue Query-Methoden ergänzen

Bei einer `@Query`-Methode aus Spring Data müssen die Parameteranzahl und die Typen zu den Platzhaltern der JPQL-Anfrage passen. Das ist aber unter Umständen unflexibel, weil man dem Client ein gutes »Aufruferlebnis« geben möchte. Falls neue Methoden im Repository gewünscht sind, die auch eine Vorverarbeitung der Parameter vornehmen sollen, sind drei Optionen denkbar:

- eigene Default-Methoden ergänzen
- SpEL-Ausdrücke in der JPQL einführen
- ein Fragment-Interface mit eigener Implementierung realisieren

Wir schauen uns die ersten beiden Möglichkeiten jetzt an, das Fragment-Interface später in Abschnitt 7.11.3, »Das Fragment-Interface«.

Default-Methoden

Wir hatten unsere erste `@Query`-Methode so formuliert:

```
@Query( "SELECT p FROM Profile p WHERE p.lastseen > ?1" )
List<Profile> findProfilesLastSeenAfter( LocalDateTime timestamp );
```

Was wäre, wenn die Abfrage nicht nur nach einem präzisen Datum möglich sein sollte, sondern nach einem Jahr, wobei dann der Startpunkt der Anfang des Jahres ist?

Dann lässt sich eine neue Default-Methode einführen, die ein `Year`-Objekt annimmt, es in ein `LocalDateTime` umwandelt und damit die eigene Methode `findProfilesLastSeenAfter(LocalDateTime)` aufruft:

```
default List<Profile> findProfilesLastSeenAfter( Year lastseen ) {
  return findProfilesLastSeenAfter( lastseen.atDay( 1 ).atStartOfDay() );
}
```

Default-Methoden können komplexe Operationen realisieren, aber zwei Nachteile sind:

- Sie ergänzen nur Methoden, blenden aber keine anderen Methoden aus.
- Da sie sich in einem Interface befinden, können sie auf keine anderen Spring-managed Beans zurückgreifen.

7.6.7 Queries mit SpEL-Ausdrücken

JPQL-Strings in einer `@Query` können SpEL-Ausdrücke enthalten. Auf diese Weise lassen sich bei Anfragen andere Spring-managed Beans einbeziehen und Parameter transformieren, bevor sie in die JPQL-Query eingesetzt werden.

Kommen wir erneut auf die Frage zurück, welche Profile sich nach einem gewissen Jahr angemeldet haben. Mit SpEL lässt sich unsere vorherige `Default`-Methode einsparen:

```
@Query( """
        SELECT p FROM Profile p
        WHERE p.lastseen > ?#{#lastseen.atDay(1).atStartOfDay()}""" )
List<Profile> findProfilesLastSeenAfter( Year lastseen );
```

Wenn man dann SpEL-Ausdrücke einsetzen möchte, müssen diese im `@Query`-String angekündigt werden. Andernfalls weiß Spring Data JPA nicht, dass der String, bevor er zum `EntityManager` geht, transformiert werden muss. Die SpEL-Ausdrücke werden deswegen besonders markiert: Sie fangen mit einem Doppelpunkt oder einem Fragezeichen an; in unserem Beispiel ist es das Fragezeichen.

Die Parameter lassen sich über ihren Namen oder über ihren Index mit eckigen Klammern ansprechen. Letztlich ergeben sich damit also vier verschiedene Schreibweisen, die wir hätten wählen können. Die anderen sind:

- … `WHERE p.lastseen > ?#{[0].atDay(1).atStartOfDay()}`
- … `WHERE p.lastseen > ?:{[0].atDay(1).atStartOfDay()}`
- … `WHERE p.lastseen > :#{#lastseen.atDay(1).atStartOfDay()}`

Wie üblich gilt im Leben: Egal, wie man es macht, Hauptsache konsistent.

Die SpEL hat eine weitere Variable im Namensraum, nämlich `{#entityName}`. Damit lässt sich der Name der Entity-Bean dynamisch erfragen. Das hätte den Vorteil, dass man beim Refactoring den Entity-Namen nicht ändern müsste.

Die Wahl dieser beiden Zeichen – Fragezeichen oder Doppelpunkt – ist kein Zufall, weil wir gesehen haben, dass wir mit Fragezeichen einen Indexparameter ansprechen und mit dem Doppelpunkt einen benannten Parameter. Diese beiden Symbole haben in der `@Query` ohnehin schon eine besondere Bedeutung.

7.6.8 Die @NamedQuery einer Entity-Bean verwenden

Die `@Query`-Definitionen haben wir bisher an die Methode gesetzt, doch das muss nicht sein. Wir hatten schon in Abschnitt 6.6.10 sogenannte *benannte deklarative Abfragen* (Named Queries) kennengelernt, also JPQL-Ausdrücke, die an der `@Entity`-Klasse festgemacht werden (oder in der XML-Datei *orm.xml* definiert werden). Der Vorteil der benannten Queries ist, dass sich JPQL-Ausdrücke nicht im Quellcode verteilen, sondern lokal bei der Entität bleiben. Ändert sich die Entität, weil zum Beispiel etwas umbenannt wird, gibt es einen kurzen Weg zum JPQL-String oben an der Klasse.

Setzen wir an die Entity-Bean eine `@NamedQuery`:

```
@Entity
@NamedQuery( name  = "Profile.findByNickname",
             query = "SELECT p FROM Profile p WHERE p.nickname = :nickname" )
public class Profile …
```

Der Trick ist der: Wenn es im Repository eine Methode `findByNickname(…)` gibt, dann ist die `@Query`-Annotation nicht mehr nötig, weil Spring Data JPA nach einer benannten Abfrage sucht. Der Aufbau des Namens muss allerdings mit dem Namen der Entity (bei uns: `Profile`) beginnen, dann muss ein Punkt als Separator folgen und dann der Name der Methode.

Gültig wäre im `ProfileRepository` also:

```
public interface ProfileRepository extends JpaRepository<Profile, Long> {
  Optional<Profile> findByNickname( String nickname );
}
```

Hinweis

Wenn man diese Methode umbenennt, muss natürlich auch die benannte Query umbenannt werden, andernfalls wird das aufgrund von *Derived Query Methods* (mehr dazu folgt in Abschnitt 7.8) merkwürdige Effekte haben. An sich sind benannte Abfragen gut, aber mit dem Mapping auf Methoden ist das ein bisschen wackelig.

7.6.9 Die @Query-Annotation mit nativem SQL

Bisher haben wir bei den Queries auf JPQL gesetzt. Doch der `EntityManager` erlaubt auch das Absenden von nativem SQL, und somit ist es auch mit Spring Data JPA möglich. Wenn in der `@Query` natives SQL eingesetzt wird, kann Spring aus dem `@Query`-String nicht ablesen, ob es sich um JPQL oder natives SQL handelt. Zur Unterscheidung muss ein Annotationsattribut `nativeQuery` wie im nächsten Beispiel auf `true` gesetzt werden:

```
@Query( value       = "SELECT * FROM Profile WHERE manelength > ?1",
        nativeQuery = true )
List<Profile> findProfilesWithManelengthGreaterThan( short minManelength );
```

Die native Query selektiert alle Spalten der Tabelle `Profile` und fragt über eine `WHERE`-Klausel nach, welches Profil eine Mähnenlänge größer als die Bind-Variable hat. Innerhalb dieser nativen Query können manche Jakarta Persistence Provider auf benannte Parameter (`:name`) zurückgreifen, allerdings ist das implementierungsabhängig und nicht standardisiert. Was standardisiert ist und genutzt werden sollte, sind Indexparameter. Die werden mit einem Fragezeichen eingeführt; der Index beginnt bei 1.

Veränderungen sind mit nativen SQL-Abfragen möglich, aber auch dann muss wieder `@Modifying` gesetzt werden, wie wir das in Abschnitt 7.6.2, »Verändernde @Query-Operationen mit @Modifying«, kennengelernt haben.

Paginierung mit nativen Abfragen

Wir haben gerade `findProfilesWithManelengthGreaterThan(…)` als native Query gesehen. Bei Methoden mit nativen SQL-Anweisungen lässt sich ebenfalls ein `Pageable`-Objekt übergeben, wie wir das in JPQL-Abfragen in Abschnitt 7.6.5, »Sort- und Pageable-Parameter«, gesehen haben:

```
@Query( value = "SELECT * FROM Profile WHERE manelength > ?1",
        nativeQuery = true )
Page<Profile> findProfilesWithManelengthGreaterThan( short minManelength,
                                                     Pageable pageable );
```

Ist die Rückgabe ein `Page`-Objekt, sind zwei SQL-Abfragen nötig: einmal für die Daten selbst und dann für die Gesamtanzahl der Elemente. In dem Moment, wo wir gewisse Einschränkungen haben, gilt es natürlich, dass die Anzahl der Elemente kleiner wird. Das heißt, ein `SELECT COUNT(*)` ist nicht richtig, sondern die Einschränkung in der `WHERE`-Klausel muss auch beim `SELECT COUNT` übernommen werden, denn ohne `WHERE`-Klausel wäre die Anzahl im Allgemeinen zu hoch.

Spring Data JPA kann die `WHERE`-Klausel automatisch setzen und würde zum Beispiel Folgendes generieren:

```
SELECT COUNT(*) FROM profile WHERE manelength > ?
```

Es kann sein, dass eine native Query mit der `WHERE`-Klausel so kompliziert ist, dass Spring kein korrektes `SELECT COUNT` generieren kann. Daher lässt sich ein `SELECT COUNT` von Hand setzen. Exemplarisch sieht das so aus:

```
@Query( value = "…",
        countQuery = "SELECT count(*) FROM … WHERE …",
        nativeQuery = true )
```

Es ist mehr als sinnvoll, das generierte SQL von Spring Data JPA daraufhin zu prüfen, ob die Abfrage korrekt ist.

Sortierungen von nativen Queries

Ein `Pageable`-Objekt kann ein `Sort`-Objekt für eine Sortierung enthalten. Ein wenig merkwürdig ist, dass man `Sort`-Objekte nicht bei den nativen `@Query`-Methoden übergeben kann, aber die Sortierung über die `Pageable`-Objekte in die Abfrage hineinschmuggeln kann.

Nehmen wir die Deklaration `findProfilesWithManelengthGreaterThan(…)` von eben, aber geben wir beim Aufruf dem `Pageable` ein `Sort`-Kriterium mit:

```
var page = profiles.findProfilesWithManelengthGreaterThan(
                        (short) 4,
                        PageRequest.of( 0, 10, Sort.by( "nickname" ) ) );
```

Die generierte SQL-Anweisung enthält dieses Sortierkriterium und auch über dieses `PageRequest` das `LIMIT` und möglicherweise das `OFFSET`:

```
SELECT * FROM Profile WHERE manelength > ? order by nickname asc limit ?
select count(*) FROM profile WHERE manelength > ?
```

Spring Data JPA hängt an unser SQL ein `ORDER BY` an.

Das Ganze wird noch ein bisschen interessanter, denn wenn wir schon ein `ORDER BY` haben, muss Spring Data JPA das berücksichtigen. Nehmen wir an, die Abfrage wäre:

```
@Query( value = """
                SELECT * FROM Profile
                WHERE manelength > ?1 ORDER BY manelength DESC",
        nativeQuery = true )
Page<Profile> findProfilesWithManelengthGreaterThan( short minManelength,
                                                     Pageable pageable );
```

Dann würde folgendes SQL generiert:

```
SELECT *
FROM Profile
WHERE manelength > ?
ORDER BY manelength DESC, nickname asc
limit ?
```

In diesem Fall ist es natürlich ein nachgeschobenes Sortierkriterium, aber trotzdem ist es beeindruckend, dass Spring Data JPA SQL-Anweisungen erkennt und umschreiben kann.[22]

7.7 Stored Procedures (gespeicherte Prozeduren) *

Eine gespeicherte Prozedur ist eine Besonderheit mancher Datenbanken. So lassen sich mehrere SQL-Anweisungen zu einer Art Unterprogramm zusammenfassen. Das hat eine Reihe von Vorteilen:

- Der Client muss nicht mehrere SQL-Anfragen realisieren, sondern nur die gespeicherte Prozedur starten. Das verringert die Kommunikation, und letztlich ist es schneller, nur einen einzigen Aufruf zur Datenbank zu starten, anstatt mehrfach zum Server zu gehen und unterschiedliche Aufrufe nacheinander zu realisieren.
- Es gibt Parameter und Rückgaben, was eine API für Geschäftsdaten bildet. Gespeicherte Prozeduren sind somit eine Art API zur Kommunikation mit der Datenbank. So können Details (zum Beispiel Namen von Tabellen oder Spalten) in den gespeicherten Prozeduren verborgen bleiben und sind für Außenstehende nicht sichtbar.
- Diese Abstraktion erhöht die Sicherheit, denn sind nur gespeicherte Prozeduren nach außen sichtbar und ist alles andere unsichtbar, dann kommt man mit den rohen Daten nicht mehr in Kontakt.

In der Praxis hat man oft beides: ausgewählte Tabellen und gespeicherte Prozeduren. Aber wenn man es auf die Spitze treiben wollte, könnte man ausschließlich über gespeicherte Prozeduren mit der Datenbank kommunizieren.

Obwohl gespeicherte Prozeduren ihre Vorteile haben, gibt es auch Nachteile:

- Das Hauptproblem ist, dass gespeicherte Prozeduren kein Teil des SQL-Standards sind. Bisher wurde eine Standardisierung nie forciert, das heißt, die gespeicherten Prozeduren sind nicht portabel. Oft realisieren die Datenbankmanagementsysteme eine große Anzahl von Erweiterungen, sodass gespeicherte Prozeduren sich fast wie imperativ programmierte Programme lesen: Es gibt Variablen, Fallunterscheidungen, Schleifen usw. Die bekanntesten Dialekte sind *PL/SQL* (von Oracle) oder *Transact SQL*, kurz *T/SQL* (von Microsoft). Manche Datenbankmanagementsysteme haben keine gespeicherten Prozeduren.
- Schwieriger ist auch die Entwicklung von gespeicherten Prozeduren. Gibt es diese ausschließlich auf der Datenbankseite, sind Entwickelnde auf die Werkzeuge der

22 Wer für JPQL-Ausdrücke Details lernen möchte, die Klasse `QueryUtils` ist hier erläutert: *https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/query/QueryUtils.html*.

Hersteller und deren Debug-Möglichkeiten angewiesen. Die Möglichkeit, Breakpoints in Skripte zu setzen, ist zum Beispiel unterschiedlich gut ausgeprägt.

- Weil gespeicherte Prozeduren auf der Datenbankseite liegen, gibt es ein Problem, wenn diese gelöscht werden oder wenn man eine neue Version in die Versionsverwaltung einpflegen möchte: Sie sind nicht zwangsläufig Teil des eigenen Programmcodes. Es ist unglücklich, wenn die gespeicherte Prozedur über mehrere Generationen nur innerhalb der Datenbank weiterentwickelt wird und außerhalb der Datenbank dieses Skript überhaupt nirgendwo auftaucht. Deswegen ist es auch wichtig, dass Installations- und Migrationsskripte außerhalb der Datenbank verwaltet werden, um die Datenbank automatisiert aufzusetzen. Das hilft auch bei Testfällen, die nur in Zusammenarbeit mit einer Datenbank möglich sind. Hier helfen Werkzeuge wie *Flyway* oder *Liquibase* ungemein – damit werden wir uns in Abschnitt 7.15, »Incremental Data Migration«, beschäftigen.

7.7.1 Eine gespeicherte Prozedur in H2 definieren

Das RDBMS H2 ist, was gespeicherte Prozeduren angeht, ein Sonderfall: In H2 werden keine SQL-Anweisungen zu einem Skript zusammengefasst, sondern die Skripte sind kleine Java-Programme mit besonderen Methoden. Java-Methoden können von außen Argumente annehmen und Tabellen als `ResultSet`, einfache Listen oder primitive Elemente zurückgeben. Sehen wir uns dazu ein Beispiel an.

Eine virtuelle Tabelle mit einer Spalte soll zufällige Namen generieren. Die gespeicherte Prozedur heißt `GET_RANDOM_NAMES`:

```
DROP ALIAS IF EXISTS GET_RANDOM_NAMES;
CREATE ALIAS GET_RANDOM_NAMES AS $$
import java.util.concurrent.ThreadLocalRandom;
@CODE
java.sql.ResultSet getRandomNames( java.sql.Connection __, int size ) {
  org.h2.tools.SimpleResultSet rs = new org.h2.tools.SimpleResultSet();
  rs.addColumn( "RANDOM_NAME", java.sql.Types.VARCHAR, 255, 0 );
  for ( int i = 0; i < size; i++ )
    rs.addRow( getName() );
  return rs;
}
static String getName() {
  int size = ThreadLocalRandom.current().nextInt( 6, 16 );
  StringBuilder newName = new StringBuilder();
  for ( int i = 0; i < size; i++ )
    newName.append( i % 2 == 0 ?
      "aeiou".charAt( ThreadLocalRandom.current().nextInt( 5 ) ) :
      "bcdfghklmnprstvwyz".charAt(ThreadLocalRandom.current().nextInt(18)) );
```

```
  newName.setCharAt( 0, Character.toUpperCase( newName.charAt( 0 ) ) );
  return new String( newName );
}
$$;
CALL GET_RANDOM_NAMES (10);
```

Zu den Details:

- `DROP ALIAS` wäre nicht wirklich nötig, würde aber die gespeicherte Prozedur löschen, wenn sie schon existiert, denn eine gespeicherte Prozedur wird nicht automatisch überschrieben; es würde einen Fehler geben.
- Die Schreibweise `CREATE ALIAS` ist spezifisch für H2, denn das definiert die Funktion. Hinter `$$` beginnt der eigentliche Java-Code. Erst kommen die `import`-Deklarationen und nach dem Trenner `@CODE` die Java-Methoden, sozusagen der Rumpf der Klasse.
- Es werden zwei Methoden deklariert: `getRandomNames(…)` und die Hilfsmethode `getName()`. Die Hilfsmethode liefert Fantasienamen, wobei einfach zwischen Vokalen Konsonanten liegen; die Stringlänge ist zufällig.
- `getRandomNames(…)` hat zwei Parameter. Die `java.sql.Connection` ermöglicht die Verbindung zur Datenbank (hier nicht genutzt, aber notwendig). Der Parameter `size` bestimmt die Anzahl generierter Namen. Die Rückgaben kommen bei uns über eine virtuelle Tabelle, die ein besonderes `ResultSet` aufbaut. `SimpleResultSet` ist ein H2-Datentyp und ein beschreibbares `ResultSet`. Es werden `size` viele Zufallsnamen in die Tabelle gesetzt, jeder Name in eine eigene Zeile.
- `CALL GET_RANDOM_NAMES (10)` ist ein Beispiel für einen Aufruf dieser gespeicherten Prozedur.

Nun soll der Aufruf der gespeicherten Prozedur nicht über SQL geschehen, sondern über ein Spring-Programm. Dazu gibt es zwei Möglichkeiten, die wir uns in den folgenden Abschnitten ansehen.

7.7.2 Eine gespeicherte Prozedur über eine native Query aufrufen

Eine der Möglichkeiten nutzt die `@Query`-Annotation mit einer nativen Abfrage:

```
@Query( value       = "CALL GET_RANDOM_NAMES(?1)",
        nativeQuery = true )
List<String> getRandomNames( int size );
```

Der SQL-String ist fast identisch wie vorher, nur haben wir hier eine Übergabe mit `?1`. Der Repository-Methode `getRandomNames(…)` lässt sich ein Argument übergeben. Das füllt den Prepared-Statement-Platzhalter, also die Bind-Variable.

Die Rückgabe ist in unserem Fall eine Liste von Strings. Falls die Stored Procedure die Spalten einer Entity-Bean liefert, so kann die Rückgabe auch auf eine Entity-Bean übertragen werden.

Der Einsatz des Schlüsselwortes `CALL` ist spezifisch für das RDBMS, weil es natives SQL und kein JPQL ist. Daher gibt es eine zweite Möglichkeit.

7.7.3 Eine gespeicherte Prozedur mit @Procedure aufrufen

Eine Repository-Methode kann mit `@Procedure`[23] annotiert werden, sodass die native Query entfallen kann:

```
@Procedure
List<String> GET_RANDOM_NAMES( int size );
```

Die Java-Methode heißt genauso wie die gespeicherte Prozedur. An die Methode übergebene Argumente werden beim Aufruf der gespeicherten Prozedur übergeben. Der Aufruf ist intern und versteckt von Spring. Auch die Rückgabe ist, wie vorher, eine Liste von Strings.

Da der Methodenname gegen die JavaBean-Konvention verstößt und vielleicht mit einer anderen Methode oder einem Java-Schlüsselwort kollidieren könnte oder weil es bei einer Umbenennung der gespeicherten Prozedur zu einem Problem kommen könnte, lässt sich bei `@Procedure` der Name der gespeicherten Prozedur angeben:

```
@Procedure( "GET_RANDOM_NAMES" )
List<String> getRandomNames( int size );
```

Damit kann der Name auf der Java-Seite unterschiedlich lauten, und das macht den Code unabhängig von dem Namen der gespeicherten Prozedur.

Wichtig

Die `@Procedure-Methode` muss transaktional aufgerufen werden, auch dann, wenn sie nichts schreibt:

```
@Transactional( readOnly = true )
public void run() {
  List<String> names = profiles.getRandomNames( 10 );
  …
}
```

Mit dem Flag `readOnly = true` kann der Treiber oder die Datenbank die Abfrage optimiert ausführen, weil die Transaktion nichts verändert, sondern nur liest.

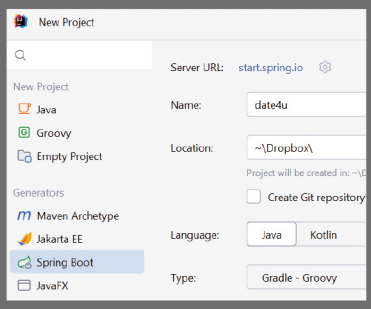
23 *https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/query/Procedure.html*

Spring Boot 3 und Spring Framework 6

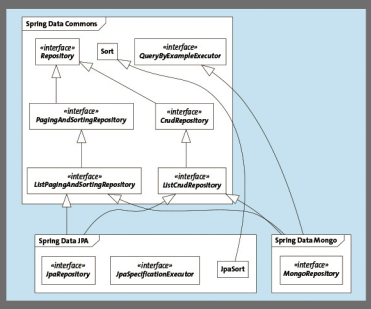
»Alles für robuste Enterprise-Anwendungen«

Moderne Softwareentwicklung mit Java

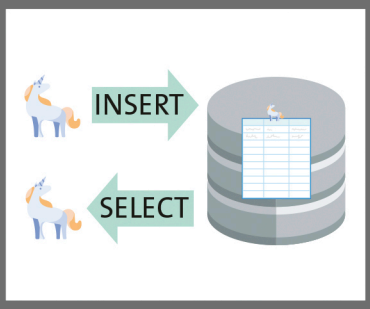
Leistungsfähige Anwendungen mit Java, ohne Ballast und voll auf der Höhe der Zeit? Willkommen in der Welt von Spring und Spring Boot! Dieses umfassende Handbuch zeigt Ihnen, was Sie über das Framework wissen sollten. Mit dem richtigen Know-how im Rücken sind Sie bereit für professionelle und moderne Softwareentwicklung mit Spring und Java.



Das erste Projekt starten



Zusammenhänge verstehen



Anschauliche Beispiele

Installation und praktischer Einstieg

Setzen Sie Ihr erstes Projekt auf und programmieren Sie direkt mit. Code und Aufgaben haben sich schon in vielen Schulungen bewährt. Wichtige Konzepte wie Dependency Injection oder die Autokonfiguration werden von Anfang an praxisnah eingeführt.

Hilfreiche Details, auch für Profis

Brauchen Sie mehr Know-how zum Caching, zur Datenzugriffsschicht oder zum Deployment? Kein Problem: Die Kapitel lassen sich auch gut einzeln lesen, enthalten fortgeschrittene Tipps und bewahren Sie vor Stolperfallen.

Lehrreiche Beispielanwendung

Tauchen Sie tief ein und lernen Sie das Framework von allen Seiten kennen. Eine durchgehende Demo-Anwendung veranschaulicht die Konzepte und Funktionen. Zahlreiche Beispiele und Praxistipps helfen Ihnen, diese schnell zu verstehen und anzuwenden.

Alle Codebeispiele zum Download

Christian Ullenboom, Dipl.-Informatiker und Java Champion, ist erfahrener Trainer und Gründer des IT-Schulungsunternehmens tutego. Der Klassiker »Java ist auch eine Insel« und das Java-Trainingsbuch »Captain CiaoCiao erobert Java« stammen ebenfalls aus seiner Feder.

Aus dem Inhalt

Rheinwerk Computing

Gedruckt in Deutschland
Mineralölfreie Druckfarben
Zertifiziertes Papier

€ 49,90 [D] € 51,30 [A]
Für Windows, macOS und Linux
Programmierung
ISBN 978-3-367-10421-5