Spring Boot 3 und Spring Framework 6

Das umfassende Handbuch

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 Dependencys 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, Netzwerk-Kommunikation, 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, hat das erste Java-Release Ende 1995 / Anfang 1996 veröffentlicht. 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 freigab, 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 *Application Server* 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 auch, 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 wie wir es heute kennen, kam erst ein wenig später. Das Interessante daran ist, dass Rod Johnson nicht wirklich geplant hatte, ein Framework zu schreiben: 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 Webseite 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 Hobby-Projekt 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. Man muss ganz klar sagen, dass damals das Spring Framework als Alternative zu J2EE-Anwendungen galt, denn diese waren schwergewichtig, der Standard nicht besonders umfangreich und die Applikationsserver aufwendig zu konfigurieren.[3] Allerdings waren auch die Spring-Anwendungen aufwendig zu konfigurieren. Beispiele vom 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 gab es noch

2 Das Web Archive hat unter *https://web.archive.org/web/20040330131500/http://www.springframework.org:80/* die alte Webseite 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 hat man gerne aus Javadoc die nötigen J2EE-XML-Deskriptoren generiert: *https://xdoclet.sourceforge.net/xdoclet/index.html*

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. Die geschachtelte Angabe `<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-Konfiguration 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 provided 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, Datenbankzugriff usw. 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 Propertys 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 haben 2013 begonnen. 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 wurde.

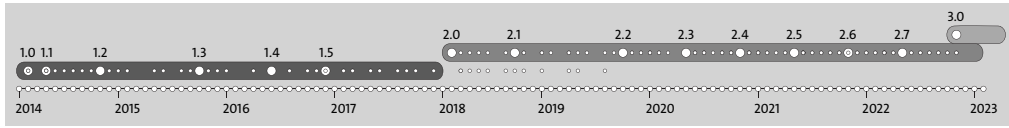

Abbildung 1.2 Release-Daten der Spring-Boot-Versionen

In den Releases gibt es bisher zwei große Linien (siehe Abbildung 1.2). Die 1er-Version von Spring Boot ist lange schon veraltet, und der 2er-Zweig ist der, der heute am weitesten verbreitet ist. Wie schnell Spring Boot 3 adaptiert wird, ist heute natürlich noch nicht abzusehen.

Regelmäßig erscheinen Updates. Während man früher sehr stark auf Feature-Releases gesetzt hat, 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 Release. 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; in der Anfangszeit der 2er-Versionen gab es noch Patches für die 1er-Version, bis dann etwa Mitte 2019 die 1er-Reihe auslief und es seitdem nur noch mit dem 2er-Release weitergeht. Ab 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 Webseite unter *https://tanzu.vmware.com/spring-runtime*. Support ist dann angebracht, wenn eine ältere Version weiterhin unterstützt werden muss, wie sich das für Spring Boot 2 abzeichnet, das nach dem 18.11.2023 keinen Support mehr erhalten wird. Kunden mit kommerzieller Unterstützung haben nach aktuellem Stand bis 18.02.2025 Zeit für eine Umstellung. Viele 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 Support-Zeitraum (siehe Abbildung 1.3).

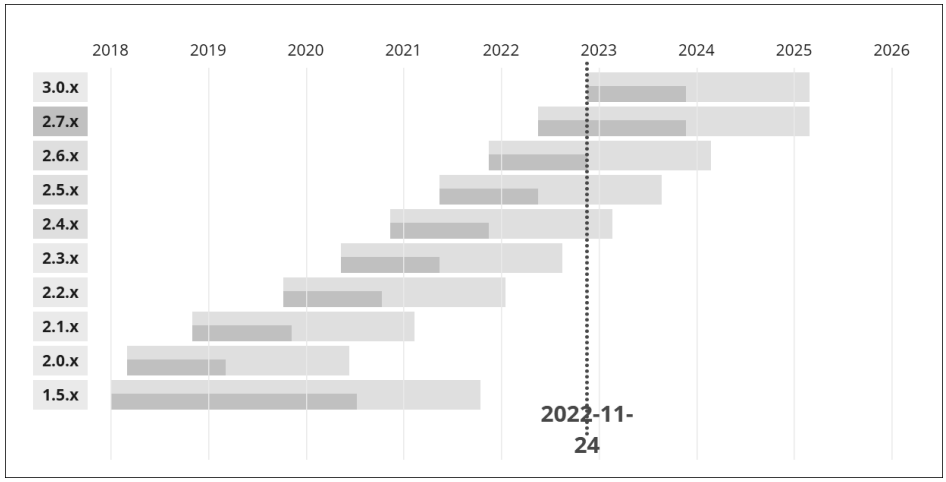

Abbildung 1.3 Support-Zeiträume der verschiedenen Spring-Boot-Versionen

Die Angabe in Gelb zeigt die kommerzielle Unterstützung; die Angabe in Grün zeigt 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*. Ich werde im Folgenden immer nur die moderne Bezeichnung »Jakarta EE« verwenden, wenn ich Java-Enterprise-Technologien aus dem Oracle-Umfeld meine.

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://www.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-API 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-API*. Objektrelationales Mapping mit der Jakarta-Persistence-API 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 Integrationsframework, 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 Byte-Code 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, 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 JSF und JSP standardisiert, lassen sich im Spring-Universum beliebige Template-Engines einsetzen. Dazu gehören *Thymeleaf*, *Free Marker*, *Velocity* oder natürlich auch *JSF*. 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-Enterprise-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, um 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 (hat sogar 12 Prozentpunkte gegenüber dem Vorjahr zugelegt).[4]

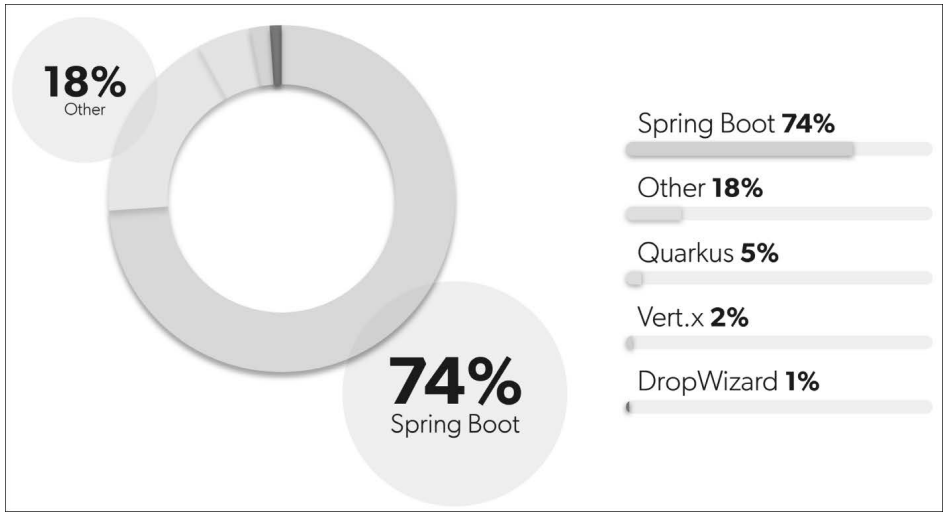

Abbildung 1.4 Verbreitung von Spring Boot nach einer Umfrage von JRebel

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.

4 *https://www.jrebel.com/system/files/jrebel-2022-java-developer-productivity-report.pdf*

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.

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 dieser auch über Dialoge direkt integriert.

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

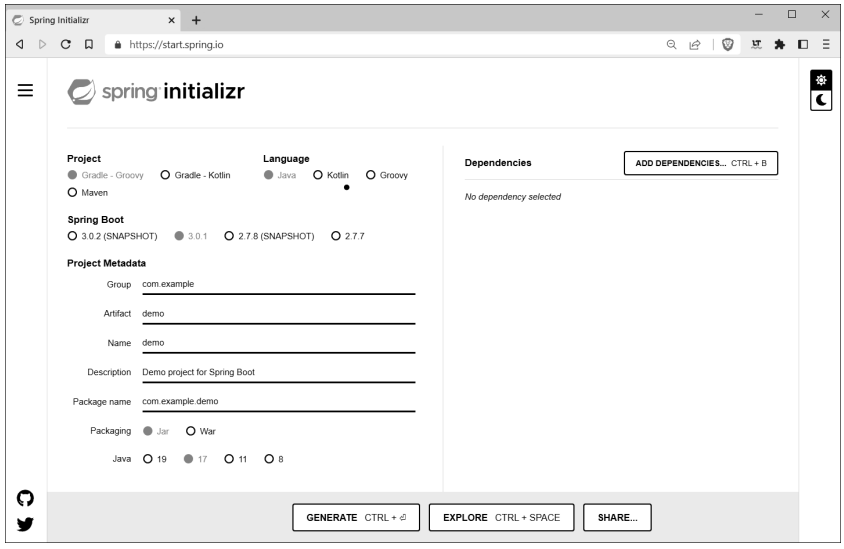

Abbildung 1.5 Die Webseite »https://start.spring.io/«

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.

Als Nächstes lässt sich die Versionsnummer von Spring Boot auswählen – da es im Monatsrhythmus Updates gibt, dürfte diese 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.

Auf der rechten Seite lassen sich unter ADD DEPENDENCIES Abhängigkeiten hinzufügen. Dabei handelt es sich nicht um Dependencys auf beliebige Java-Projekte, sondern um speziell ausgewählte Projekte aus dem Spring-Universum. 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 Dependencys 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).

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.

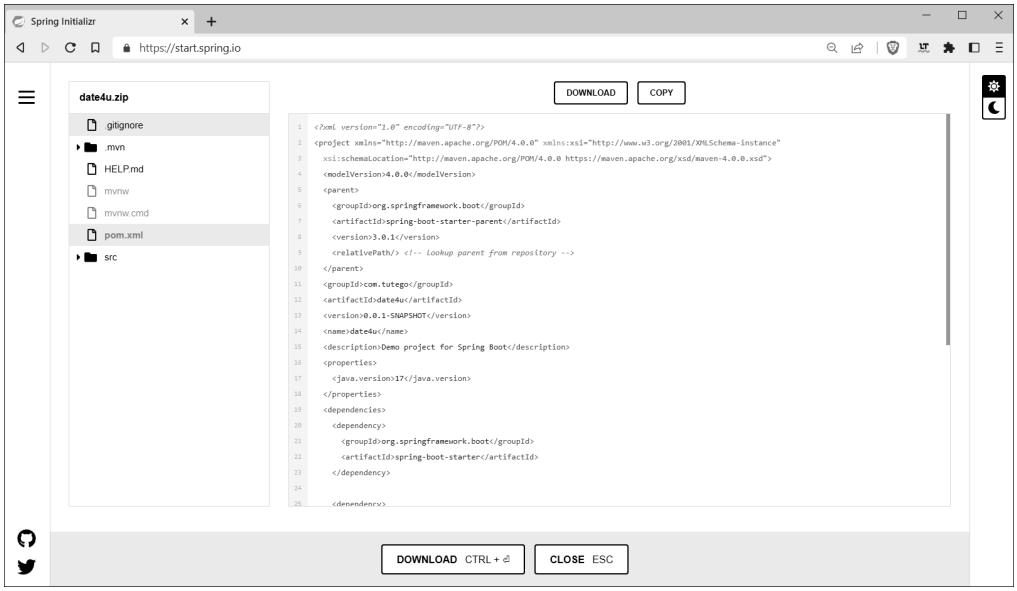

Abbildung 1.6 Ein erster Blick auf das Projekt

Ebenfalls gibt es eine *HELP.md*-Datei; sie enthält eine kompakte Dokumentation über die Dependencys. 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.)

Doch auch auf der Kommandozeile lässt sich das Projekt nun bauen und ausführen. Denn der Initializr generiert mit dem *Maven Wrapper* eine Art eigene lokale Maven-Installation. Mit `mvnw -version` lässt sich die Installation testen:

```
$ ./mvnw -version

Apache Maven 3.8.6 (84538c9988a25aec085021c365c560670ad80f63)
Maven home: …\.m2\wrapper\dists\apache-maven-3.8.6-bin\ ↩
1ksOnkde5v1pk9vtc31i9dOlcd\apache-maven-3.8.6
```

```
Java version: 17, vendor: Oracle Corporation, runtime: ↩
…\jdk-17
Default locale: de_DE, platform encoding: Cp1252
OS name: "windows 10", 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 Umgebungsvariable. 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] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] >>> spring-boot-maven-plugin:3.0.0:run (default-cli) > test-compile @
date4u >>>
[INFO]
[INFO] --- maven-resources-plugin:3.3.0:resources (default-resources) @ date4u
---
[INFO] Copying 1 resource
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.10.1:compile (default-compile) @ date4u ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to …\target\classes
[INFO]
[INFO] --- maven-resources-plugin:3.3.0:testResources (default-testResources)
@ date4u ---
[INFO] skip non existing resourceDirectory …\src\test\resources
[INFO]
[INFO] --- maven-compiler-plugin:3.10.1:testCompile (default-testCompile) @
date4u ---
```

```
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to …\target\test-classes
[INFO]
[INFO] <<< spring-boot-maven-plugin:3.0.0:run (default-cli) < test-compile @
date4u <<<
[INFO]
[INFO]
[INFO] --- spring-boot-maven-plugin:3.0.0:run (default-cli) @ date4u ---
[INFO] Attaching agents: []


  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v3.0.0)

2022-11-28T18:38:14.846+01:00  INFO 61404 --- [           main]
com.tutego.date4u.Date4uApplication      : Starting Date4uApplication using
Java 17 with PID 61404 (…\target\classes started by Christian in …)
2022-11-28T18:38:14.850+01:00  INFO 61404 --- [           main]
com.tutego.date4u.Date4uApplication      : No active profile set, falling back
to 1 default profile: "default"
2022-11-28T18:38:15.252+01:00  INFO 61404 --- [           main]
com.tutego.date4u.Date4uApplication      : Started Date4uApplication in 0.71
seconds (process running for 0.998)
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  3.103 s
[INFO] Finished at: 2022-11-28T18:38:15+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 sich alle nötigen Bestandteile vom Maven-Central-Server herunter, und bis auf ein installiertes JDK muss kein weiteres Werkzeug vorhanden sein. Weitere Details gibt die Webseite unter *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. Diese 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.
- **NetBeans**: Fans von NetBeans können ebenfalls ein Plugin (*https://plugins.netbeans.apache.org/catalogue/?id=4*) installieren, das sogar das zweitpopulärste Plugin überhaupt ist.

Die nächsten Abschnitte zeigen, wie die unterschiedlichen IDEs helfen, ein Spring-Boot-Projekt ohne weitere Dependencys 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 die Schaltfläche 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).

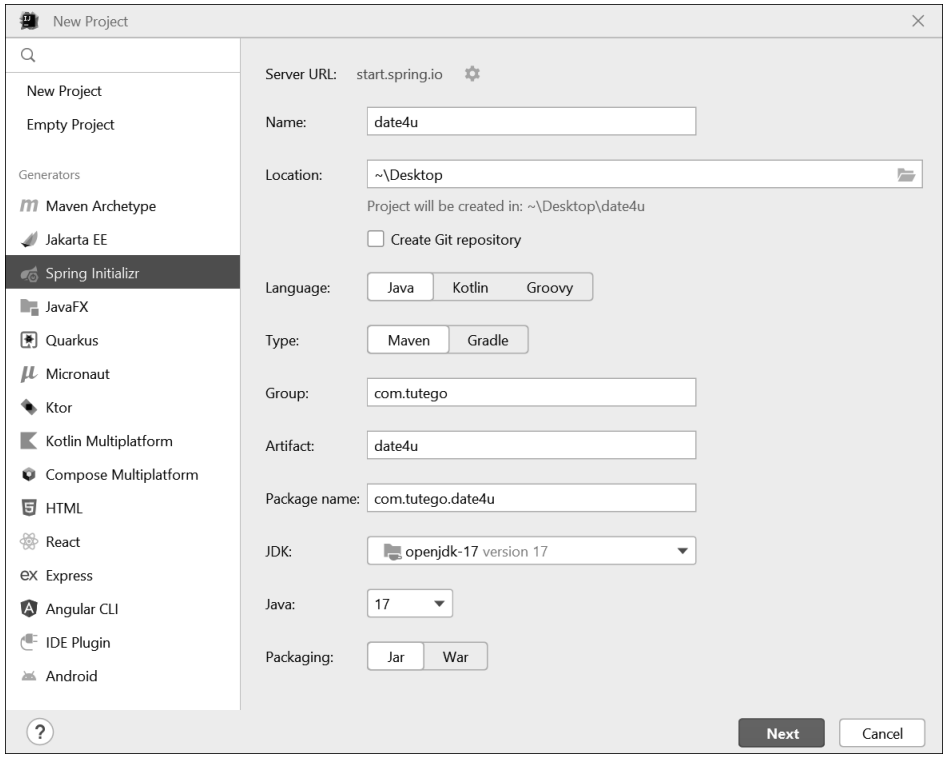

Abbildung 1.7 Dialog zum Erzeugen eines neuen Projekts

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.

Wir haben die Möglichkeit, zwischen den Programmiersprachen Java, Kotlin und Groovy auszuwählen. Wir bleiben bei Java. Auch bleiben wir bei Maven. Beim 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. Java 17 hat IntelliJ als JDK erkannt, und bei der JAVA-Version wählen wir mindestens 17. 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).

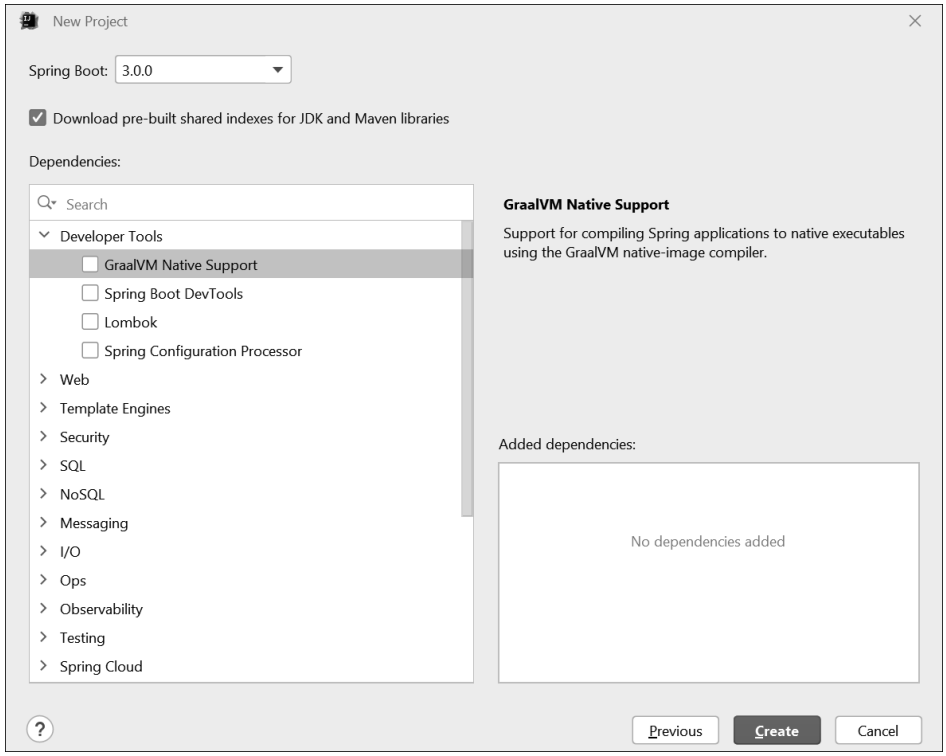

Abbildung 1.8 Dependencys eintragen

Bei den *Dependencys* 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 Dependencys hinzunehmen.

Ein Klick auf CREATE schließt den Prozess ab. Nun entsteht ein Maven-Projekt, was 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 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 unterschiedlich 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.

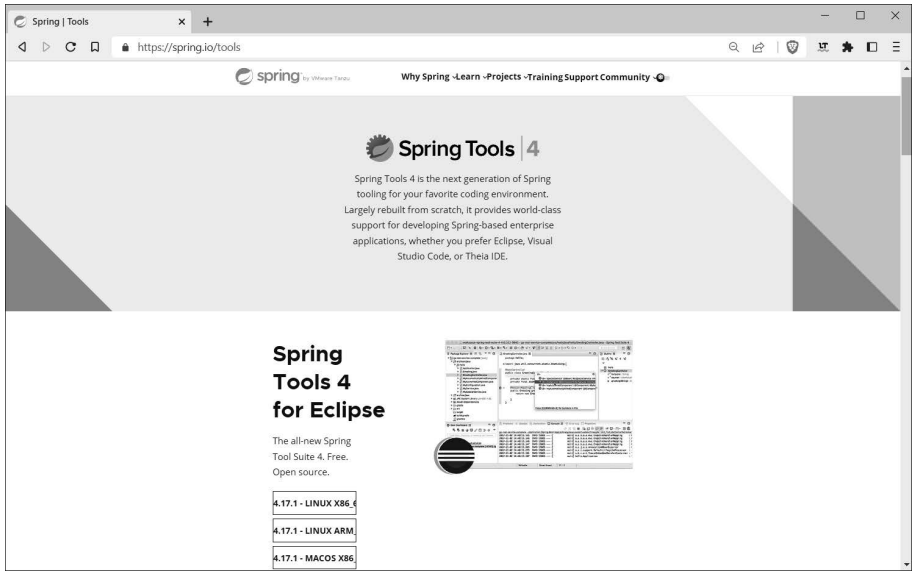

Abbildung 1.9 Die Eclipse-STS-Startseite »https://spring.io/tools« mit Download-Links

Die Datei zum Herunterladen ist ein *Self-Extracting-Jar*, das nur gestartet werden muss. 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.

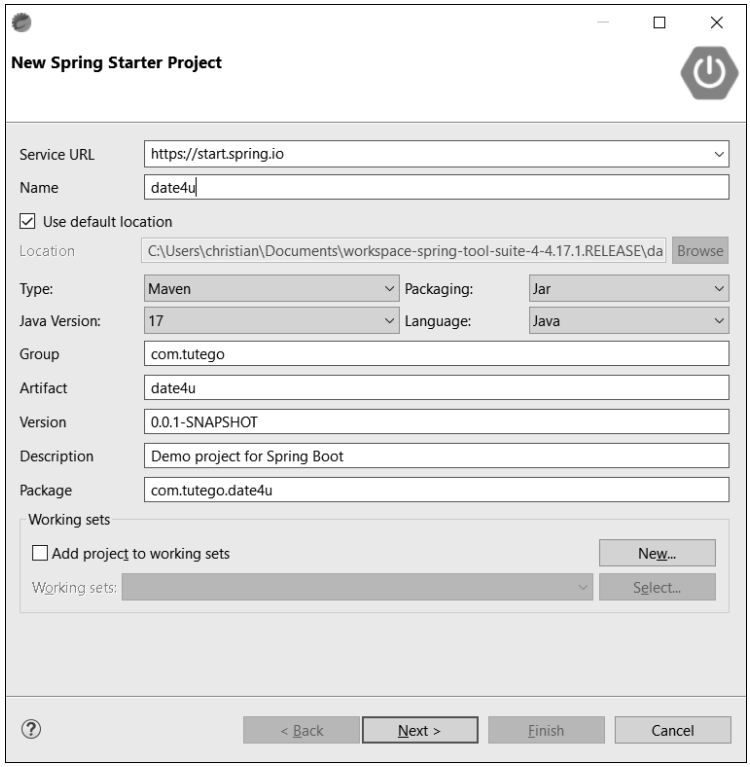

Abbildung 1.10 Ein neues Spring-Boot-Projekt in STS anlegen

Wir sehen an dem Dialog aus Abbildung 1.10, dass der ein bisschen ähnlich aussieht wie 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. Standardmäßig wählt der Initilizr bei TYPE Gradle aus, wir ändern das in Maven. Alle Java-Versionen ab Java 17 sind erlaubt; Java 17 ist eine gute Wahl und vorausgewählt. 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 Dialog aus Abbildung 1.11. An dieser Stelle wird es interessant, denn wir können die Dependencys auswählen, die typisch für Spring-Projekte sind.

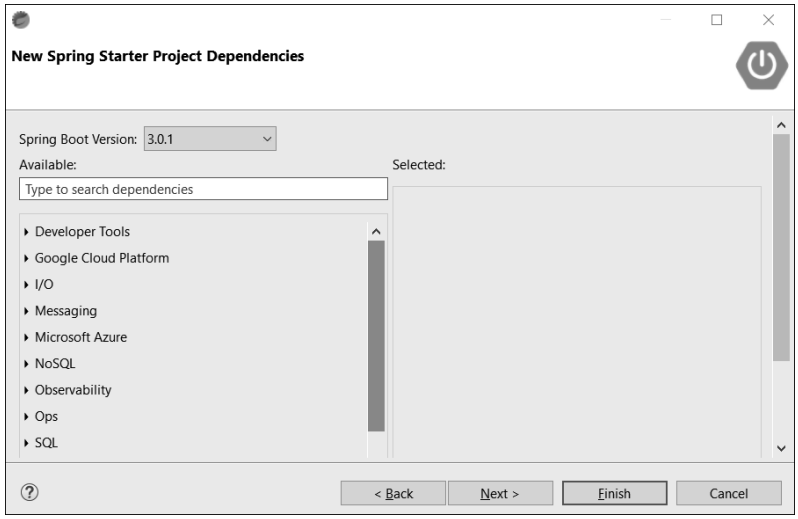

Abbildung 1.11 Dependencys eintragen

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 Dependencys, 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 Dependencys und Starter eines Spring-Boot-Projekts

An dieser Stelle wollen wir uns die Dependencys 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 POM-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 Dependencys, 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>3.0.0</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 ein Parent-POM mit BOM

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

Bei der Deklaration dieser BOM geht das Spring-Team zweistufig vor. Im ersten Schritt wird eine große Anzahl von Propertys definiert:

Listing 1.3 Ausschnitt aus der »org.springframework.boot:spring-boot-dependencies«-POM

```
<properties>
  <angus-mail.version>1.0.0</angus-mail.version>
  <artemis.version>2.26.0</artemis.version>
  <aspectj.version>1.9.9.1</aspectj.version>
  <assertj.version>3.23.1</assertj.version>
  <awaitility.version>4.2.0</awaitility.version>
  <brave.version>5.14.1</brave.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-

Projekte 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 hochzusetzen funktioniert nicht. 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:

Listing 1.4 Ausschnitt aus der »org.springframework.boot:spring-boot-dependencies«-POM

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.eclipse.angus</groupId>
      <artifactId>angus-core</artifactId>
      <version>${angus-mail.version}</version>
    </dependency>
    …
</dependencyManagement>
```

Dependency-Management bedeutet, dass die genannten Dependencys nicht wirklich mitgenommen, sondern lediglich mit ihrer Versionsnummer deklariert werden. Bei den Versionen greift die POM auf die vorher deklarierten Variablen 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://github.com/spring-projects/spring-boot/releases* anschaut, dann tauchen die Dependency-Upgrades immer auf (siehe Abbildung 1.12).

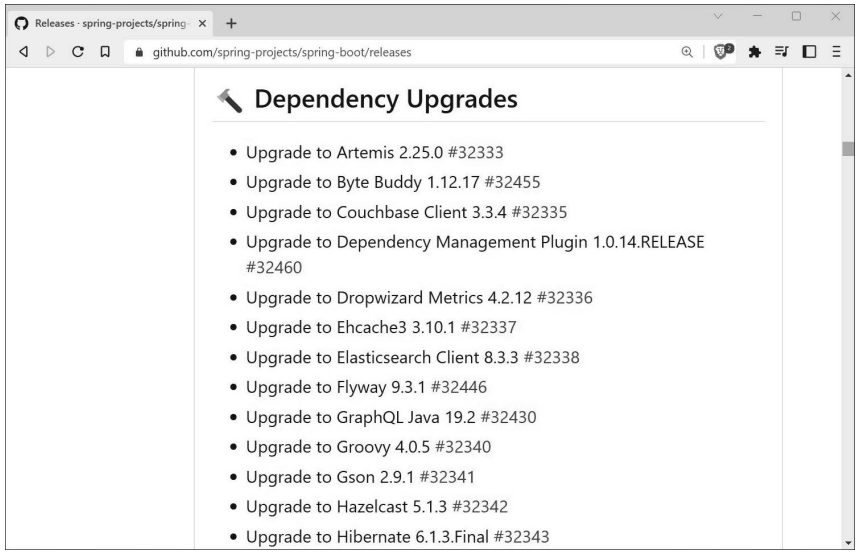

Abbildung 1.12 Dependency-Updates der Spring-Boot-Versionen in den Release-Notes

Es ist möglich, die Versionsnummern einzelner Projekte zu korrigieren. Dafür lässt sich im Propertys-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 Dependencys als Import

Spring-Boot-Dependencys 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 anschlie-

ßend die Parent-POM durch die eigene ersetzen. Das Problem ist dann allerdings, dass die Dependencys 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 Propertys, wie UTF-8-Encoding. Diese Eigenschaften müssen händisch ergänzt oder in einem eigenen Parent-POM definiert werden.

1.2.3 Milestones und das Snapshots-Repository

Auf dem Maven-Central-Server werden nur endgültige Versionen hochgeladen und gehostet. Was wir dort nicht finden, sind Entwicklungsversionen, also keine 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 Starter: Dependencys des Spring Initializr

In der POM-Datei hat der Initializr zwei Dependencys 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.

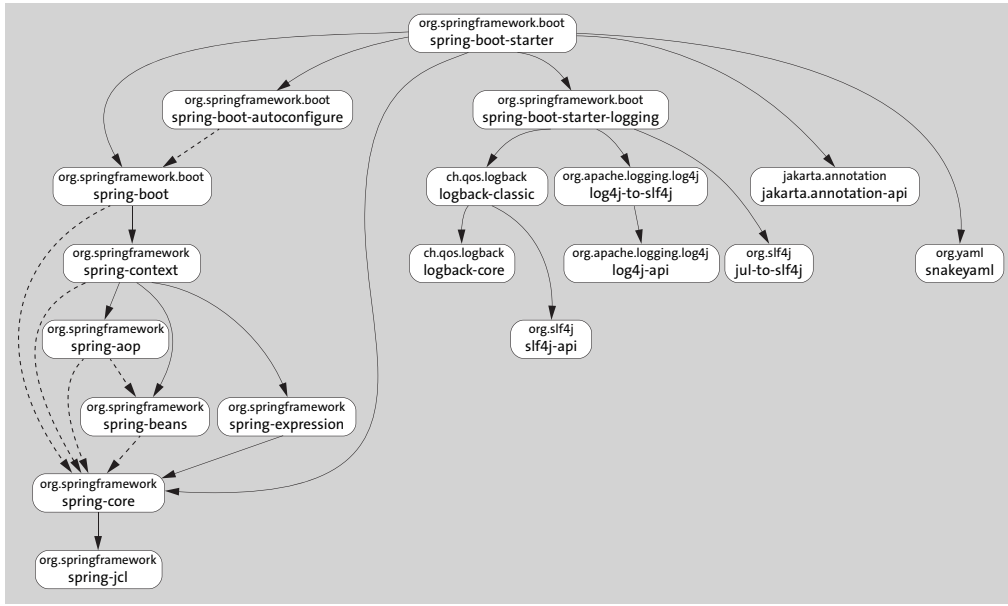

Abbildung 1.13 Dependencys des Core-Starters

In Abbildung 1.13 lassen sich die Abhängigkeiten gut ablesen: `spring-boot-starter` steht oben und hat eine ganze Reihe von Dependencys. 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 in diesem `spring-boot-starter`.

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 Starters

Es gibt nicht nur einen Starter, sondern eine große Anzahl von Startern. Die 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/docs/current/reference/htmlsingle/#using-boot-starter*. 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 `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*, die standardmäßig leer ist. 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 1.5 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 SL4J

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://www.slf4j.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 hin-

ter 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 formulieren. 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() );
   ```

   Die Variable ist statisch, da es eine Log-Meldung aus einer statischen Methode geben soll.

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

   ```
   log.info( "Log mit Argumenten {}, {} 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 Format-String, 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 mit Argumenten {}, {} 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*.

Das 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.

Log-Level setzen

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

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

Die Konfigurations-Propertys 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 die *Spring Boot CLI*, ein Kommandozeilenwerkzeug. Da das in der Praxis selten gebraucht wird, verweise ich auf die Referenzdokumentation *https://docs.spring.io/spring-boot/docs/current/reference/html/cli.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 4
Ausgewählte Proxys

Bei Spring-Anwendungen spielt die Injektion eine wichtige Rolle. Durch sie werden Objekte an die Stellen gebracht, an denen sie gebraucht werden. Es kann aber sein, dass der Client keine Referenz auf das gewünschte Objekt selbst bekommt, sondern dass Spring einen Proxy aufbaut. Einen Proxy kann man sich als einen Ring vorstellen, der sich um ein Objekt legt und alle Methodenaufrufe abfängt und gegebenenfalls weiterleitet (siehe Abbildung 4.1).

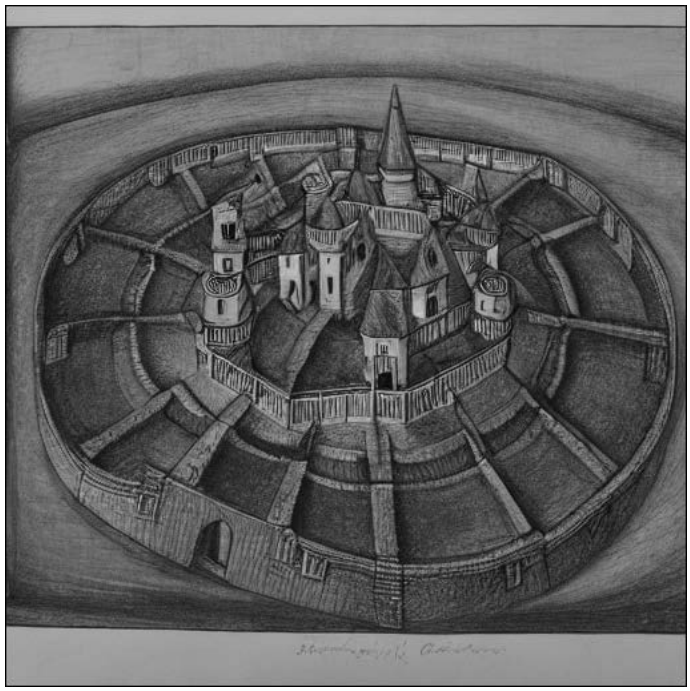

Abbildung 4.1 Proxys sind wie Ringe um einen Kern.

Auf dem Weg zum Zielobjekt im Kern könnte es verschiedene Prüfungen geben, die auch so weit gehen können, dass der Zugriff nicht gelingt.

4.1 Proxy-Pattern

Was anhand von Ringen anschaulich erklärt werden kann, wollen wir mit einem Sequenzdiagramm in die Welt der Softwaretechnik übersetzen. Der Ring veranschaulicht das *Proxy-Pattern*, ein bekanntes Entwurfsmuster (engl. *design pattern*).

Was in Abbildung 4.1 im Kern lag, ist das *Zielobjekt* in Abbildung 4.2. Dieses Zielobjekt hat eine Schnittstelle, nennen wir sie `I`, mit Operationen. Um dieses Zielobjekt legt sich das *Proxy-Objekt*. Es hat nach außen die gleiche Schnittstelle `I` wie das Zielobjekt, das auch *Subjekt* genannt wird. Hat ein Client eine Referenz auf »etwas vom Typ `I`«, weiß der Client nicht, ob er mit dem Proxy redet oder mit dem Zielobjekt. Im Deutschen spricht man statt von einem Proxy auch von einem *Stellvertreter*.

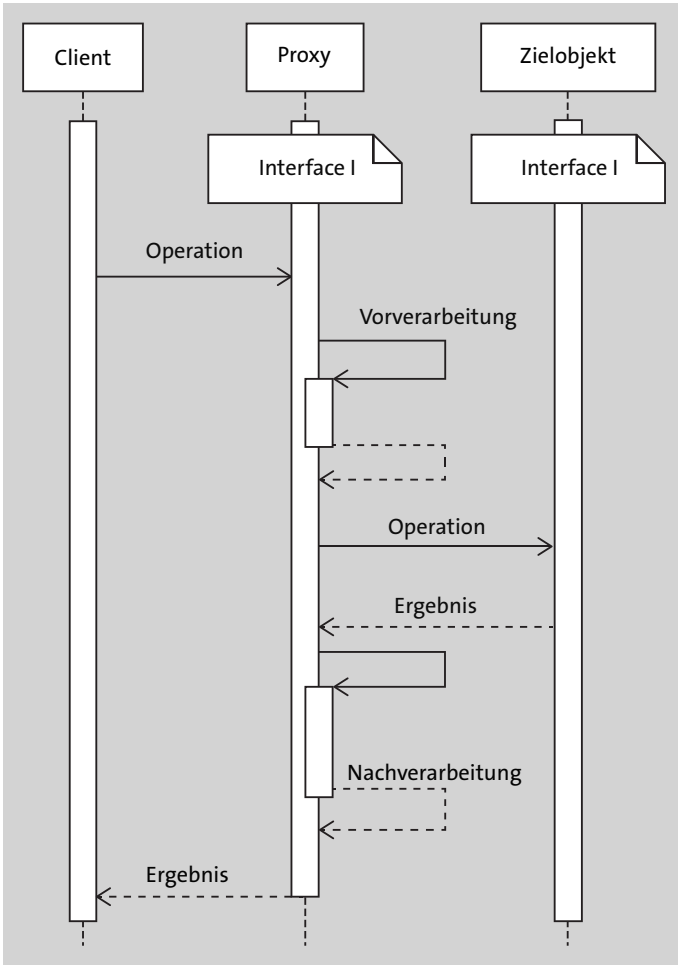

Abbildung 4.2 Sequenzdiagramm des Proxy-Patterns

Ein Proxy klingt nicht spannend, doch er kann zwei nützliche Dinge leisten:

- Bevor der Proxy mit der Operation an das Zielobjekt geht, kann er eine Vorverarbeitung durchführen. Das heißt, er kann interne Methoden aufrufen, und er könnte die Parameter modifizieren. Anschließend kann der Proxy zum Zielobjekt gehen oder den Aufruf blockieren.

- Wenn das Subjekt dem Proxy das Ergebnis zurückliefert, kann der Proxy eine Nachverarbeitung vornehmen. Zum Beispiel kann er die Daten cachen. Später liefert der Proxy das Ergebnis wieder an den Client.

Hinweis

Das `Class`-Objekt lässt sich erfragen, und darüber kann man herausfinden, ob man ein Proxy-Objekt vor sich hat oder nicht. Auch die `toString()`-Methode antwortet anders. Spring enthält in `org.springframework.aop.support.AopUtils` die Hilfsmethode `isAopProxy(…)` für Spring-Proxys:

```
public static boolean isAopProxy(@Nullable Object object) {
  return
    (object instanceof SpringProxy &&
 (Proxy.isProxyClass(object.getClass()) ||
    object.getClass().getName().contains(
    ClassUtils.CGLIB_CLASS_SEPARATOR)));
}
```

Bei der Implementierung von `equals(…)` muss stellenweise berücksichtigt werden, dass Vergleiche mit `instanceof` statt mit `getClass(…)` programmiert werden.

4.1.1 Proxy-Einsatz in Spring

Ein Proxy macht eigentlich nicht viel, insbesondere kann er nicht in das Verhalten der Subjekt-Operation eingreifen. Aber diese Vorverarbeitung und Nachverarbeitung reicht für viele Aufgaben aus, und daher nutzt Spring Proxys an vielen Stellen. Häufig bekommen der Code einen Stellvertreter injiziert und wir merken es überhaupt nicht; der Proxy hat ja die gleiche Schnittstelle wie das »Original«.

Ein paar Beispiele, wie Spring Proxys nutzt:

- **Transaktionsmanagement:** Methoden können neue Transaktionen anstoßen, und alles, was die Methode mit der Datenbank macht, findet in einer transaktionalen Klammer statt. Wenn es in der Methode Ausnahmen gibt, so würde der Proxy diese Ausnahmen abfangen und ein Rollback realisieren; wenn es keine Fehler gibt, dann folgt am Methodenende ein Commit.
- **Caching:** Geht ein Client mit einem gewissen Argument zu einer Methode, kann ein Proxy feststellen, dass die Methode das Argument in der Vergangenheit schon einmal gesehen hat, und kann das früher berechnete Ergebnis zurückliefern.
- **Validierung:** Die Aufgabe der Validierung ist, dass Methoden nur mit gültigen Werten aufrufen werden. Ein Proxy kann prüfen, ob die Argumente in den richtigen Wertebereichen liegen, und sie nur bei Gültigkeit an das Subjekt weiterleiten.

- **Asynchrone Aufrufe:** Normalerweise wird eine Methode synchron und blockierend aufgerufen. Ein besonderer Proxy kann einen Hintergrund-Thread starten – oder auf einen Thread eines Thread-Pools zurückgreifen – und dann asynchron diese Operation abarbeiten.
- **Spring Retry:** Es können – besonders bei entfernten Zugriffen – Abbrüche auftreten; Spring Retry kann einen Proxy einsetzen, der beim Misslingen einer Subjekt-Operation einen weiteren Versuch startet, bis der Aufruf letztendlich glückt oder man aufgibt.

Übersetzen wir das in Java-Quellcode, um ein besseres Gefühl für die Arbeitsweise zu bekommen.

Proxy-Pattern im Code

Das Proxy-Pattern, in Code programmiert, hätte folgende Grundform:

```
class Subject {
  public String operation( String input ) { return input; }
}

class Proxy extends Subject {
  private final Subject subject;

  Proxy( Subject subject ) { this.subject = subject; }

  @Override public String operation( String input ) {

    // Vorverarbeitung mit input

    String result = subject.operation( input );

    // Nachverarbeitung mit Ergebnis

    return result;
  }
}
```

Der Proxy muss mit dem Subjekt verbunden werden. So speichert der Konstruktor die Referenz.

Was hier in Java-Code symbolisch steht, wird im Spring Framework automatisch zur Laufzeit generiert.

4.1.2 Proxys dynamisch generieren

Das Spring Framework kann zur Laufzeit Proxys generieren; deswegen sprechen wir bei dieser Form von Proxys auch von *dynamischen Proxys*. Das Spring Framework greift dazu auf zwei Techniken zurück.

- Das JDK kann dynamische Proxys aufbauen; diese nennt man *JDK Dynamic Proxys*. In diesem Fall gibt es eine Einschränkung, denn die Java SE kann nur Proxys für Schnittstellen realisieren – für Klassen geht das nicht.
- Wenn Spring für Klassen einen Proxy generiert, dann greift das Framework auf die Bibliothek *cglib* (kurz für *Byte Code Generation Library*)[1] zurück. Diese Byte Code Generation Library ist relativ alt und wird nicht mehr gepflegt, und daher hat das Spring-Team die Version stark gepatcht, um sie für die aktuellen Java-Versionen fit zu machen.

Gebote und Verbote für Subjekte

Wenn Proxys gut funktionieren sollen, dann müssen die Klassen einige Regeln berücksichtigen.

Baut Spring Proxy-Objekte für Klassen auf, dann erzeugt eine Bibliothek Bytecode mit einer Unterklasse. Daraus folgt, dass die Oberklasse (das Subjekt) keinen privaten Konstruktor haben darf. Das ist klar, denn eine Unterklasse ruft immer den Konstruktor der Oberklasse auf, und wenn die Oberklasse keinen sichtbaren Konstruktor hat, kann die Konstruktion nicht funktionieren. Fazit: Die Konstruktoren im Subjekt dürfen nicht privat sein.

Wenn Spring einen Proxy durch eine Unterklasse realisiert, so wie wir das in unserem eigenen Proxy-Beispiel mit den Typen `Proxy` und `Subject` im vorangegangenen Abschnitt gemacht haben, dann muss der Proxy die Methode aus dem Subjekt überschreiben. Deswegen darf die Methode nicht `final` sein. Andernfalls könnte sie nicht überschrieben werden.

Grundsätzlich sollten alle Methoden, für die ein Proxy generiert wird, `public` sein. Spring greift auf sein eigenes AOP-Framework zurück, das `public`-Methoden voraussetzt. Zwar lassen sich alle diese genannten Einschränkungen aushebeln, dann muss allerdings der Bytecode über AspectJ modifiziert werden. Wir werden hier ausschließlich den »üblichen« Weg nutzen.

Fasst man die oben genannten Ge- und Verbote zusammen, gibt es ein paar Einschränkungen, wenn das Framework Proxy-Objekte aufbauen soll. Es gibt eine Sache, auf die man achten muss: Ein Proxy kann nur dann seine Aufgabe erledigen, wenn er zwischen dem Client und dem Subjekt steht. Ruft das Subjekt im eigenen Code unter-

1 *https://github.com/cglib/cglib*

einander Methoden auf, dann geht das nicht durch den Proxy. Das kann schnell zu einem Problem führen, etwa beim Refactoring.

Ein erfundenes Beispiel soll das Problem verdeutlichen. Stellen wir uns vor, es gäbe eine Proxy-Klasse `TrimmingProxy`, die immer dann, wenn Parametervariablen mit `@Trim` annotiert sind, von diesen Strings den Weißraum vorn und hinten entfernt und sie, auf diese Weise getrimmt, an das Subjekt übergibt. So könnte eine Nutzung aussehen:

```
IntParser proxy = new TrimmingProxy( new IntParser() );
proxy.parseInt( "  12423\t   " );
```

Jetzt schauen wir den Code der Klasse `IntParser` mit der Methode an:

```
class IntParser {
  public int parseInt( String input ) {
    return parseInt( input, 10 );
  }
  public int parseInt( @Trim String input, int radix ) {
    return Integer.parseInt( input, radix );
  }
}
```

Achtet der Proxy auf ein `@Trim`, wird er nur beim *direkten* Aufruf von `parseInt(String, int)` aktiviert, sonst nicht. Unser Beispiel ruft `parseInt(String)` auf und die Parametervariable hat kein `@Trim`, also wird auch der Proxy keinen Weißraum entfernen. Wenn `parseInt(String)` später `parseInt(String, int)` aufruft, geschieht das sozusagen innerhalb des »Kerns« und der Proxy ist nicht beteiligt.

Hinweis

In der Praxis ist das ein oft gemachter Fehler, daher muss man sich immer wieder in Erinnerung rufen, dass ein Proxy immer »von außen« in den Kern geht und erst dann seinen Job erledigt.

Proxies selber bauen mit ProxyFactory *

Spring bietet die Klasse `ProxyFactory`[2], mit der auch wir Proxy-Objekte aufbauen können. Ein Beispiel: Es soll ein Proxy-Objekt für eine `java.util.List` erzeugt werden. Auf dieser besonderen Liste soll `add(null)` ignoriert werden:

2 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/aop/framework/ProxyFactory.html*

```
ProxyFactory factory = new ProxyFactory( new ArrayList<>() );
MethodInterceptor methodInterceptor = invocation ->
  (invocation.getMethod().getName() == "add"
  && isNull( invocation.getArguments()[ 0 ] )) ? false
                                               : invocation.proceed();
factory.addAdvice( methodInterceptor );
```

Im Konstruktor von `ProxyFactory` wird das Subjekt übergeben, eine `ArrayList`. Das Ziel ist, dass `ProxyFactory` uns einen Proxy generiert und wir gewisse Methoden im Hintergrund an die Liste weitergeben.

Der Proxy fängt jeden Methodenaufruf ab. Elementar ist daher eine Konfiguration, wie sich der Proxy bei welcher Methode mit welchen Argumenten zu verhalten hat. Ein `MethodInterceptor`[3] hilft uns dabei; die funktionale Schnittstelle ist wie folgt deklariert:

```
public interface MethodInterceptor extends Interceptor {
  @Nullable
  Object invoke(@Nonnull MethodInvocation invocation) throws Throwable;
}
```

Wird von außen eine Methode auf dem Proxy aufgerufen, so gibt er den Aufruf an den konfigurierten `MethodInterceptor` weiter und ruft die `invoke(…)`-Methode auf. Anders gesagt: Egal auf welche Methode von außen zugegriffen wird, sie wird immer auf eine `invoke(…)`-Methode abgebildet. Daher ist der Parameter `MethodInvocation` so wichtig, denn das Objekt enthält Informationen darüber, welche Methode mit welchen Argumenten aufgerufen wurden.

Unsere Implementierung testet mit `invocation.getMethod().getName() == "add"` zuerst, ob die Methode wirklich `add` heißt. Da die Strings der Methodennamen intern sind, ist es kein Problem, `==` für den Vergleich zu nutzen – so etwas ist allerdings selten, und hier ist es wirklich eine Ausnahme. Die anschließende Prüfung mit `isNull( invocation.getArguments()[ 0 ] )` fragt, ob der `add(…)`-Methode `null` übergeben wurde. Es gibt zwei Ausgänge für die Vergleiche:

- Es wurde `add(null)` aufgerufen. In diesem Fall geben wir einen `boolean`-Wert zurück, denn das ist auch der Rückgabetyp der `add(…)`-Methode. Die Rückgabe `false` drückt aus, dass keine Veränderung an der Datenstruktur vorgenommen wurde. An das Subjekt leiten wir nichts weiter. Zwar liefert `invoke(…)` nur `Object`, doch ist es wichtig darauf zu achten, dass der Rückgabetyp mit dem vom Subjekt bei der Methode übereinstimmt.

3 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/aopalliance/intercept/MethodInterceptor.html*

- Wurde nicht `add(null)` aufgerufen, also entweder eine andere Methode oder `add(…)` mit einem anderen Wert, dann geht der Proxy mit `invocation.proceed()` an den Kern, und was das Subjekt liefert, liefern wir bei `invoke(…)` zurück.

Damit ist der `MethodInterceptor` fertig. Nachdem er bei `addAdvice(…)` übergeben wurde, können wir von der `ProxyFactory` das Proxy-Objekt beziehen und nutzen. Ein Beispiel:

```
List list = (List) factory.getProxy();
list.add( "One" );
System.out.println( list );
list.add( null );
list.add( "Two" );
System.out.println( list );
```

Die Ausgaben sind:

```
[One]
[One, Two]
```

Genug zu den selbst programmierten Proxys, schauen wir uns nun einige Standard-Proxys aus dem Spring Framework an.

4.2 Caching

In diesem Abschnitt wollen wir uns intensiver mit dem Thema Caching beschäftigen.

4.2.1 Optimierung durch Caching

Caching ist wichtig, wenn Daten bezogen werden, die im Heranschaffen teuer sind und wenn auf einen schnellen Zwischenspeicher zurückgegriffen werden kann, in dem das Objekt zeitweise gespeichert werden kann.

Bekannte Anwendungen von Caching sind:

- Ein DNS-Cache wird auf jedem PC und Smartphone eingesetzt. Er merkt sich die IP-Adresse zu einem Hostnamen.
- Unternehmen setzen einen Web-Proxy und Cache ein. Gehen Abfragen von dem Firmennetzwerk nach außen, wird der Proxy abgerufene Teile zwischenspeichern.
- Bei einem Webserver wird nicht jede Abfrage zu einem Dateizugriff führen, sondern der Server hält die Dateiinhalte vor und würde nur bei Dateiänderungen den Cache-Eintrag aktualisieren.

- Bei Spring-Anwendungen werden oft Inhalte einer Datenbank im Cache gehalten, um Datenbankzugriffe einzusparen.

Herausforderungen beim Caching

Caching lohnt sich nur dann, wenn das erfragte Element in der Erstellung oder in der Abfrage relativ teuer ist.[4] Eine besondere Herausforderung sind gespeicherte und dann ungültig gewordene Elemente. Bei einer reinen idempotenten Operation, zum Beispiel bei einer Sinusfunktion (es kommt etwas rein und immer das Gleiche kommt heraus), ist das Cachen einfach. Doch viele Einträge werden irgendwann ungültig. Ist zum Beispiel mit einer ID ein Datensatz assoziiert, dann kann sich der Datensatz ändern. Folglich muss der Cache-Inhalt als veraltet markiert werden; im Englischen wird das *stale* genannt.

Ein Cache muss außerdem wie das menschliche Gehirn arbeiten: Er muss vergessen können. Es bringt wenig, wenn der Cache den ganzen Hauptspeicher belegt und das zum `OutOfMemoryError` der Anwendung führt. Der Cache muss Elemente vergessen, die lange nicht mehr benutzt wurden, so wie vermutlich alle von uns die Bestandteile einer Blütenpflanze aus dem 6. Schuljahr vergessen haben.

4.2.2 Caching in Spring

Spring bietet eine sehr einfache Möglichkeit, ein Caching zu aktivieren. Die Grundidee ist, dass ein Spring-Proxy eine Assoziation zwischen Methodenargumenten und der Rückgabe aufbaut. Auch das Auswechseln verschiedener Caching-Implementierungen ist mit minimalem Aufwand möglich.

Es gibt mehrere Ansätze, einen Caching-Mechanismus in Spring zu realisieren. Üblicherweise wird Caching deklarativ verwirklicht, das heißt, der eigentliche Cache-Rückgriff ist im eigenen Code nicht sichtbar. Es gibt zwei Ansätze für den deklarativen Weg: *annotationsbasiert* oder *XML-basiert*. Den XML-Weg lassen wir außen vor, denn er ist heutzutage unüblich, wie auch die gesamte Container-Konfiguration.

Bei den Annotationen gibt es zwei Optionen:

- Spring-Annotationen wie `@Cacheable`, `@CacheEvict`, `@CachePut`, `@Caching`, `@CacheConfig`

4 In der Anfangszeit von Java hat man Angst vor den Kosten einer Objekterzeugung gehabt und Objekte für später »aufgehoben«. Heute ist das im Allgemeinen keine gute Strategie, denn die Objekterzeugung und das Aufräumen sind verhältnismäßig billig. Wird ein Objekt künstlich lebendig gehalten, wird der generationenbasierte Garbage Collector die Objekte in die »old generation« verschieben, und dort ist die Speicherfreigabe nicht mehr so schnell wie bei neuen, frischen Objekten.

- JCache-Annotationen wie `@CacheResult`, `@CachePut`, `@CacheRemove`, `@CacheRemoveAll`, `@CacheDefaults`, `@CacheKey`, `@CacheValue`. JCache ist der Standard aus dem JSR 107 (*https://www.jcp.org/en/jsr/detail?id=107*).

Die Spring- und JCache-Annotation sind nicht nur anders benannt, auch die Semantik ist stellenweise anders; bei JCache werden auch Ausnahmen gecacht. Spring Cache kann auch mit einer JCache-Implementierung konfiguriert werden.

Im folgenden Beispiel werden die Spring-Annotationen eingesetzt.

@EnableCaching

Die eigentliche Arbeit im Hintergrund übernehmen Proxy-Objekte. Ein Proxy wird sich später um die Objekte legen und das Ergebnis der annotierten Methode in den Cache legen. Die Proxys werden nicht automatisch aufgebaut, sondern dafür ist an einer beliebigen Konfigurationsklasse die Annotation `@EnableCaching`[5] zu setzen:

```
@EnableCaching
@Configuration
class CacheConfig { }
```

Unsere Hauptklasse, die mit `SpringBootApplication` annotiert war, war eine besondere `@Configuration`-Klasse, also würde es auch dort funktionieren.

Werden später Methoden aufgerufen, die `@Cacheable`, `@CacheEvict` usw. sind, wird ein Proxy die Aufrufe empfangen.

4.2.3 Komponente mit @Cacheable implementieren

Schauen wir uns ein kleines Beispiel für eine Operation an, deren Resultat im Cache landen soll:

```
@Component
class HotProfileToYamlConverter {
  private final Logger log = LoggerFactory.getLogger( getClass() );

  @Cacheable( "date4u.yamlhotprofiles" )
  // @Cacheable( cacheNames = "date4u.yamlhotprofiles" )
  public String hotAsYaml( List<Long> ids ) {
    log.info( "Generating YAML for list {}", ids );
    return new Yaml().dump( Map.of( "trendy", ids ) );
  }
}
```

5 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/annotation/EnableCaching.html*

Die Methode `hotAsYaml(…)` bekommt eine Liste von IDs mit den angesagtesten Einhörnern und antwortet mit einem YAML-String. Das Programm greift direkt auf die YAML-Bibliothek *SnakeYAML* (*https://bitbucket.org/snakeyaml/snakeyaml/src/master/*) zurück, die der Spring-Boot-Starter integriert.

Schauen wir uns drei Dinge an, die im Rahmen von Caching wichtig sind:

- Die Annotation `@Cacheable`[6] wird genau an die Methoden gesetzt, deren Argumente wir mit der Methodenrückgabe assoziieren wollen. Ein Cache ist im Kern nichts anderes als ein Assoziativspeicher, der einen Schlüssel mit einem Wert assoziiert: Der Schlüssel für den Cache ist das Argument, das der Methode übergeben wurde. Der Proxy merkt sich den Rückgabewert, der bei jedem individuellen Argument zurückkam.
- Ein Cache braucht grundsätzlich einen Namen; ein hierarchischer Aufbau ist sinnvoll, um Konflikte mit anderen Anwendungen zu vermeiden. Der Name lässt sich über das Annotationsattribut `cacheNames` oder `value` setzen. Der Name ist aus zwei Gründen wichtig:
  - Es können innerhalb einer Klasse verschiedene Caches benutzt werden.
  - Verschiedene Klassen können den gleichen Cache verwenden.
- Die Methoden, die der Proxy aufruft, müssen `public` sein. Das hatten wir in Abschnitt 4.1.2 unter »Gebote und Verbote für Subjekte« schon besprochen. Prinzipiell wären andere Sichtbarkeiten möglich, doch der Proxy muss auf diese Methoden zurückgreifen, und das funktioniert am besten, wenn die Methode `public` ist. Andere Sichtbarkeiten sind möglich, doch dann muss mit AspectJ der Bytecode umgebogen werden.

Schauen wir uns nun an, wie wir den `HotProfileToYamlConverter` nutzen können.

4.2.4 @Cacheable-Proxy nutzen

Der `HotProfileToYamlConverter` ist eine Spring-managed Bean und lässt sich injizieren:

```
@Autowired HotProfileToYamlConverter converter;
```

Das, was Spring injiziert, ist *nicht* unser eigenes Objekt! Wir bekommen etwas, was vom Typ `HotProfileToYamlConverter` ist, aber es ist der Proxy. Im Debugger (oder über das `Class`-Objekt) ist das gut zu sehen:

```
log.info( converter.getClass().getName() );
// c.t.d.HotProfileToYamlConverter$$EnhancerBySpringCGLIB$$8d40339b
```

6 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/annotation/Cacheable.html*

Der Name des `Class`-Objekts enthält den String `EnhancerBySpringCGLIB`, und das ist die typische Identifikation eines zur Laufzeit generierten Proxys.

Dieser Proxy hat die Methode `hotAsYaml(…)`, die wir nutzen können:

```
String yml1 = converter.hotAsYaml( Arrays.asList( 1L, 2L, 3L ) );
String yml2 = converter.hotAsYaml( Arrays.asList( 1L, 2L, 3L ) );
// assertThat( yml1 ).isSameAs( yml2 );
```

Die Strings, die am Ende herauskommen, sind identisch und nicht nur gleichwertig. Das ist ein wichtiger Punkt und darauf zurückzuführen, wie der Proxy arbeitet.

Schon der erste Aufruf von `hotAsYaml(…)` nimmt der Proxy entgegen. Der Proxy bildet das Argument auf einen Schlüssel ab (wie das auch angepasst geht, zeigt Abschnitt 4.2.9, »Eigene Schlüsselgeneratoren angeben«) und geht intern an den Cache mit der Frage: »Gibt es für die Liste mit den Werten 1, 2, 3 schon ein assoziiertes Ergebnis?« Nein, gibt es nicht bei der ersten Abfrage. Daher geht der Proxy zum Kern, bekommt den String und vermerkt ihn im Cache. Beim zweiten Aufruf von `hotAsYaml(…)` sieht es schon anders aus: Auch dann geht der Proxy zuerst an den Cache, doch dieser hat für die Liste mit 1, 2, 3 einen Eintrag. Mit anderen Worten: Der zweite Aufruf geht nicht zum Kern, sondern der Proxy liefert den gecachten Wert zurück. Daher taucht er in der Log-Ausgabe auch nur einmal auf:

```
Generating YAML for list [1, 2, 3]
```

Würde man mit einer zweiten Liste arbeiten, würde das Gleiche von vorne beginnen:

```
String yml3 = converter.hotAsYaml( Arrays.asList( 1L ) );
// assertThat( yml1 ).isNotSameAs( yml3 );
```

Die Liste mit 1 hat der Cache noch nicht gesehen, also geht er an den Kern und cacht das Ergebnis. Natürlich sind die YAML-Strings der Listen 1, 2, 3 und 1 völlig unterschiedlich.

Bilder cachen

Betrachten wir das Caching an einem praktischen Beispiel in unserer Anwendung *Date4u*.

Im `PhotoService` existiert eine `download(…)`-Methode mit folgender Implementierung:

Listing 4.1 PhotoService.java, Ausschnitt

```
public Optional<byte[]> download( String imageName ) {
  try {
    return Optional.of( fs.load( imageName + ".jpg" ) );
  }
```

```
  catch ( UncheckedIOException e ) {
    return Optional.empty();
  }
}
```

Übergeben wird der `download(…)`-Methode ein Bildname, und die Rückgabe ist ein `Optional` mit einem `byte`-Array. Intern geht die Methode an das Dateisystem.

Wird die `download(…)`-Methode zweimal aufgerufen, könnte problemlos das Bild als `byte`-Array aus dem Cache kommen. Umgesetzt werden kann es wie folgt:

1. **Schritt 1:** An einer beliebigen Konfiguration muss `@EnableCaching` gesetzt werden – das Hauptprogramm ist mit `@SpringBootApplication` annotiert, da wäre das gültig.

   Listing 4.2 Date4uApplication.java, Erweiterung

   ```
   @SpringBootApplication
   @EnableCaching
   public class Date4uApplication {
     public static void main( String[] args ) {
       SpringApplication.run( Date4uApplication.class, args );
     }
   }
   ```

2. **Schritt 2:** Bei dem `PhotoService` muss die Annotation `@Cacheable` mit einem Namen ergänzt werden:

   Listing 4.3 PhotoService.java, Erweiterung

   ```
   @Cacheable( "date4u.filesystem.file" )
   public Optional<byte[]> download( String imageName )
   ```

Der Cache-Name ist frei wählbar, aber `date4u.filesystem.file` sollte eindeutig sein.

4.2.5 @Cacheable + condition

Unter Umständen sollen bestimmte Objekte nicht in den Cache, etwa wenn sie sehr groß sind oder vorher bekannt ist, dass ein Objekt nicht häufig erfragt wird. Daher lässt sich das Caching an Bedingungen knüpfen, zum Beispiel: Wenn das Objekt bestimmte Eigenschaften hat, dann soll es nicht in den Cache kommen.

Bei Spring steuert das Annotationsattribut `condition`[7], ob ein Objekt in den Cache soll. Eine `condition` enthält eine Bedingung, und wenn sie wahr ist, wird das Objekt in den Cache aufgenommen.

7 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/annotation/Cacheable.html#condition--*

Im Code sieht das so aus:

```
@Cacheable( cacheNames = "date4u.yamlhotprofiles",
            condition  = "true" )
```

Diese Bedingung wird als SpEL-Ausdruck (siehe Abschnitt 2.11, »Spring Expression Language (SpEL)«) geschrieben. Die SpEL ist sehr leistungsfähig und erlaubt Rückgriffe auf den Objektgraph und auf alle Parameter. `condition = true` ist ein sehr einfacher SpEL-Ausdruck und steht exemplarisch für die Schreibweise. Die Belegung `condition= true` ist übrigens der Default.

Nehmen wir an, bei `hotAsYaml(…)` lohnt sich das Caching nur für große Listen – kleine Listen sollen nicht im Cache landen. Das lässt sich so ausdrücken:

```
@Cacheable( cacheNames = "date4u.yamlhotprofiles",
            condition  = "#ids.size() > 10" )
public String hotAsYaml( List<Long> ids )
```

Die Bedingung `#ids.size() > 10` enthält die Abfrage, ob die Liste mehr als 10 Elemente enthält. Die Variable `ids` ist der Parametername, auf den der SpEL-Ausdruck zugreifen kann. Der Parametertyp ist `java.util.List`, und der hat eine Methode `size()`. Ist die Anzahl der Listenelemente größer als 10, dann sollte der Cache gefragt und auch gefüllt werden. Das ist einleuchtend: große Listen cachen, kleine Listen nicht. Wenn die Bedingung falsch ist, also die Liste klein ist, dann geht es direkt zum Kern und auch wieder vom Kern direkt zum Client, ohne das Objekt in den Cache zu übernehmen.

Für die Bedingung wird üblicherweise auf die Parameter zurückgegriffen, und in der SpEL sind weitere Bezeichner vordefiniert:

- `#root.method`, `#root.target`, `#root.caches`
- `#root.methodName`, `#root.targetClass`
- `#root.args[0]`, `#p0`, `#a0`, `root.args[1]` usw. oder einfach `#Parametername`

Es lässt sich auf das Reflection-`Method`-Objekt zurückgreifen oder alternativ auf den Methodennamen bzw. `Class`-Objekt oder den Klassennamen.

Es gibt verschiedene Schreibweisen, um auf die Parameter zurückzugreifen, wobei die Variante am lesbarsten ist, die unser Beispiel nutzt: mit dem Doppelkreuz und dem Parameternamen.

4.2.6 @Cacheable + unless

Ein Veto verhindert, dass ein bisher noch nicht gecachtes Ergebnis nach dem Aufruf des Datengebers in den Cache aufgenommen wird. Das ist nützlich für Fälle, in denen der Kern nichts zurückliefert, zum Beispiel wenn die Ergebnisse `null` oder leer sind.

Beim Spring Caching wird das Veto mit dem Annotationsattribut `unless`[8] ausgedrückt:

```
@Cacheable( cacheNames = "date4u.yamlhotprofiles"
            unless     = "false" )
```

Der SpEL-Ausdruck ist hier `false`, das heißt, es gibt *kein* Veto, also kommt das Ergebnis in den Cache.

Dass `unless` auf `false` steht, ist nur ein Beispiel zur Verdeutlichung. Eingesetzt wird in der Praxis wieder ein SpEL-Ausdruck, und der kann auf die bekannten Informationen zurückgreifen, auf den Methodennamen, die Parameterliste usw. Es gibt eine zusätzliche Variable `result`, mit ihr ist das Ergebnis aus dem Kern im SpEL-Ausdruck zugänglich. Damit lässt sich eine Entscheidung fällen, ob das Objekt in den Cache kommt oder nicht.

Kommen wir zu etwas Praktischerem.

Beispiel 1: Ein YAML-Ergebnis unter 100 Zeichen soll nicht gecacht werden:

```
@Cacheable( cacheNames = "date4u.yamlhotprofiles",
            unless     = "#result.length() < 100" )
```

Wenn das Ergebnis – `result` ist hier ein `String` – weniger als hundert Zeichen hat, dann ist es ein Veto und der String soll nicht in den Cache. `unless` steuert auf diese Weise, dass kleine Strings nicht gecacht werden und dass sich Caching nur dann lohnt, wenn die Strings größer werden.

Beispiel 2: Wenn eine Methode eine Sammlung liefert, das Ergebnis jedoch `null` oder die Sammlung leer ist, soll das ein Veto sein; der SpEL-Ausdruck könnte so aussehen:

```
#result == null || #result.size() == 0
```

Wenn das Veto wahr ist, kommt die Sammlung nicht in den Cache. Anders gesagt: Wenn es kein Veto gibt, kommt die Sammlung in den Cache und war dann nicht `null` und nicht leer.

Bemerkungen

Kommt es im Kern, den der Cache-Proxy ummantelt, zu einer Ausnahme, werden diese Ausnahmen nicht gecacht. Das heißt, es gibt keine Assoziation zwischen einem Wert und der Exception, dass es mit dem Wert nicht funktioniert.

8 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/annotation/Cacheable.html#unless--*

Liefert eine Methode ein `Optional` zurück, steht nicht das `Optional` in `result`, sondern in `result` steht entweder `null` (wenn das `Optional` `isEmpty()` ist) oder der Wert aus dem `Optional`.

4.2.7 @CachePut

Ist eine Methode mit `@CachePut`[9] annotiert, lässt sich der Cache direkt beschreiben. Das ist praktisch, wenn der Cache initial gefüllt werden soll, denn wenn die Elemente nicht vorhanden sind, muss der Cache nicht erst erfragt werden. `@CachePut` ist zudem nützlich, wenn Elemente im Cache aktualisiert und überschrieben werden sollen, etwa bei veralteten Werten. Die darunter liegende Methode im Kern wird also immer aufgerufen, der Wert ermittelt und gespeichert.

Bei `@CachePut` und bei `@Cacheable` gelten ähnliche Annotationsattribute, da sind sich die beiden Annotationen recht ähnlich. Beide Annotationen zusammen ergeben im Übrigen keinen Sinn, wir müssen also entweder `@CachePut` oder `@Cacheable` verwenden.

4.2.8 @CacheEvict

Der Cache darf nicht ewig wachsen, sondern die Elemente müssen aus einem Cache auch entfernt werden. Hierzu gibt es zwei Strategien:

- Der Cache verliert aufgrund von gewissen Bedingungen seine Elemente selbstständig und vergisst.
- Wir selbst löschen bewusst Elemente aus dem Cache oder löschen vielleicht den gesamten Cache.

Der erste Punkt ist eine Konfigurationsfrage, und das Vorgehen schauen wir uns in Abschnitt 4.2.12, »Caching mit Caffeine«, an einem lokalen Cache an.

Eine mit `@CacheEvict`[10] annotierte Methode kann Einträge aus dem Cache löschen:

```
@CacheEvict( cacheNames = "date4u.yamlhotprofiles" )
public void removeHotAsYaml( List<Long> ids )
```

Aufgeführt wird wiederum der Cache-Name in der Annotation, und die Parameterliste ist wie bekannt.

9 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/annotation/CachePut.html*

10 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/annotation/CacheEvict.html*

Rufen wir die Methode von außen auf, wird der Cache das Element entfernen. Mit der Belegung `allEntries=true` lassen sich alle Elemente für den genannten Cache löschen.

```
@CacheEvict( cacheNames = "date4u.yamlhotprofiles",
             allEntries = true )
public void removeAllHotAsYaml()
```

Die Parameterliste ist leer, ein Parameter ist nicht nötig. `condition` ist bei dieser Annotation ebenfalls erlaubt.

Die Methoden, die mit `CacheEvict` annotiert sind, sind typischerweise die Methoden, die wirklich auf einem darunterliegenden Datenspeicher die Daten löschen. Das heißt, dann sollen sie auch aus dem Cache fliegen.

4.2.9 Eigene Schlüsselgeneratoren angeben

Ein Cache assoziiert immer einen Schlüssel mit einem Wert. Der Cache-Proxy muss aus den übergebenen Argumenten einen Schlüssel bilden. Die Notwendigkeit besteht bei allen mit `@Cachable`, `@CachePut` oder `@CacheEvict` annotierten Methoden. Die Schlüsselbildung ist einfach, wenn eine Methode nur einen Parameter hat und ein »dankbarer« Schlüsseltyp wie `long` oder `String` vorkommt.

Das Problem beginnt, wenn eine Methode keinen oder mehr als einen Parameter deklariert. Außerdem kann es Parametertypen geben, die nicht so gut als Schlüssel taugen.

Um aus übergebenen Argumenten der Methode einen Schlüssel zu generieren, greift Spring auf einen `KeyGenerator`[11] zurück (siehe Abbildung 4.3). Die Schnittstelle verdichtet die übergebenen Argumente einer Methode in ein `Object`, den Schlüssel für den Cache.

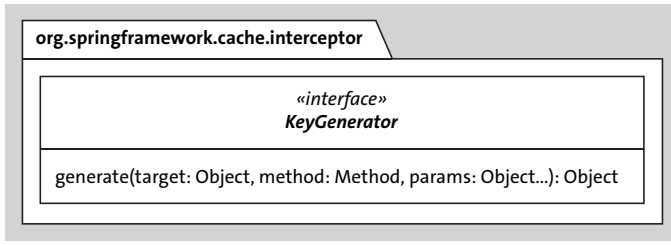

Abbildung 4.3 Die funktionale Schnittstelle »KeyGenerator«

11 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/interceptor/KeyGenerator.html*

SimpleKeyGenerator

Das Spring Framework nutzt standardmäßig für die Abbildung der Argumente auf einen Cache-Schlüssel die `KeyGenerator`-Implementierung `SimpleKeyGenerator`[12]. Der Code[13] ist übersichtlich:

```
public class SimpleKeyGenerator implements KeyGenerator {

    @Override
    public Object generate(Object target, Method method, Object... params) {
        return generateKey(params);
    }

    public static Object generateKey(Object... params) {
        if (params.length == 0) {
            return SimpleKey.EMPTY;
        }
        if (params.length == 1) {
            Object param = params[0];
            if (param != null && !param.getClass().isArray()) {
                return param;
            }
        }
        return new SimpleKey(params);
    }
}
```

Die überschriebene Methode `generate(…)` aus der Schnittstelle ruft die eigene öffentliche statische Methode `generateKey(…)` auf, die die eigentliche Abbildung vornimmt. `target` und das `Method`-Objekt werden nicht vom `SimpleKeyGenerator` verwendet – die Implementierung bildet nur aus den Argumenten einer Methode den Cache-Schlüssel.

Die Methode `generateKey(…)` berücksichtigt für die Übergabe an die Methode, die vom Cache-Proxy abgefangen wird, vier Fälle:

- Die Methode hat keine Parameter. Das kommt selten vor, ist aber gültig, und in diesem Fall ist die Rückgabe `SimpleKey.EMPTY`. Auf diese Weise lassen sich teure Fabrikmethoden deklarieren, deren Ergebnisse zeitweise zwischengespeichert werden können.

12 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/interceptor/SimpleKeyGenerator.html*).

13 *https://github.com/spring-projects/spring-framework/blob/main/spring-context/src/main/java/org/springframework/cache/interceptor/SimpleKeyGenerator.java*

- Die Methode hat exakt einen Parameter, und der ist nicht `null` und auch kein Array. Dann bildet das übergebene Argument wirklich direkt den Schlüssel. Das war bei unserer Liste von IDs der Fall, und das ist performant und speicherschonend, weil keine zusätzlichen Behälterobjekte nötig sind.
- Die Methode hat exakt einen Parameter, und der ist `null` oder ein Array wurde als Datenstruktur übergeben. Dann wird ein `SimpleKey`-Objekt mit `null` oder dem Array aufgebaut.
- Falls mehrere Parameter existieren, wird ein `SimpleKey` für die Argumente generiert.

Das SimpleKey-Objekt

In vielen Fällen wird der Cache als Schlüssel ein `SimpleKey`[14]-Objekt referenzieren. Der Code[15] sieht wie folgt aus; `readObject(…)` für die Serialisierung wurde ausgespart:

Listing 4.4 SimpleKey.java

```
package org.springframework.cache.interceptor;

import …

public class SimpleKey implements Serializable {

    public static final SimpleKey EMPTY = new SimpleKey();


    private final Object[] params;

    private transient int hashCode;


    public SimpleKey(Object... elements) {
        Assert.notNull(elements, "Elements must not be null");
        this.params = elements.clone();
        this.hashCode = Arrays.deepHashCode(this.params);
    }
```

14 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/interceptor/SimpleKey.html*

15 *https://github.com/spring-projects/spring-framework/blob/main/spring-context/src/main/java/org/springframework/cache/interceptor/SimpleKey.java*

```
    @Override
    public boolean equals(@Nullable Object other) {
        return (this == other ||
                (other instanceof SimpleKey &&
                        Arrays.deepEquals(this.params,
                          ((SimpleKey) other).params)));
    }

    @Override
    public final int hashCode() {
        return this.hashCode;
    }

    @Override
    public String toString() {
        return getClass().getSimpleName() + " ["
        + StringUtils.arrayToCommaDelimitedString(this.params) + "]";
    }
    …
}
```

Der Konstruktor nimmt die an den Proxy übermittelten Methodenargumente an, kopiert die Werte und berechnet im Vorfeld den Hashwert, damit die spätere Abfrage billig ist. Wichtig sind die Implementierungen der Methoden `equals(…)` und `hashCode()`, denn der Schlüssel muss im Assoziativspeicher gefunden werden. Eine `toString()`-Implementierung ist nützlich, aber für den Cache unnötig.

Einen eigenen Schlüsselgenerator über key oder keyGenerator festlegen

Der `SimpleKeyGenerator` ist ein guter Standard, aber er hat Nachteile, wenn zum Beispiel `equals(…)`/`hashCode()` nicht überschrieben werden, nicht gut implementiert sind oder wenn das Objekt nicht als Schlüssel im Cache gewünscht ist. Daher lässt sich ein Schlüsselgenerator selbst angeben; Spring bietet dazu zwei Möglichkeiten.

Die Annotationstypen `@Cacheable`, `@CachePut` und `@CacheEvict` bieten zwei verschiedene Möglichkeiten, über eine Annotationsattribut einen Schlüsselgenerator zu setzen:

```
public @interface Cacheable / CachePut / CacheEvict {
   …
  // Option 1:
  String key() default "";           (1)

  // Option 2:
  String keyGenerator() default "";  (2)
   …
}
```

Option 1 ist ein SpEL-Ausdruck zur Extraktion eines Schlüssels; bei Option 2 setzen wir den Bean-Namen einer Spring-managed Bean , die die Schnittstelle Typ `KeyGenerator` implementiert. Die Details zu Option 2 sehen wir uns weiter unten an.

Bei Option 1 wird ein SpEL-Ausdruck benutzt, der vom übergebenen Objekt den Schlüssel extrahiert. Nehmen wir an, ein komplexes Objekt wie ein Buch liegt vor; dann ist das gesamte Buch als Schlüssel für den Cache nicht nötig und vielleicht sogar unmöglich, wenn keine sinnvollen `equals(…)`/`hashCode()`-Methoden implementiert sind. Sinnvoller ist der Rückgriff auf die ISBN; wenn das ein String ist, ist das ein wunderbarer Schlüssel für ein Objekt.

Den Schlüsselgenerator über key festlegen

Stellen wir uns den Typ `Profile` für ein Dating-Profil vor. `Profile`-Objekte sollten nicht als Schlüssel verwendet werden, aber eine Profil-ID vom Datentyp `long` wäre ein ausgezeichneter Schlüssel für den Cache:

```
@Cacheable( cacheNames = "…",
            key        = "#prof.id" )
Object method( Profile prof )
```

Ist bei `key` ein SpEL-Ausdruck angegeben, wird Spring für die Schlüsselbildung nicht mehr den `SimpleKeyGenerator` einsetzen, sondern auf den SpEL-Ausdruck bei der Schlüsselbildung zurückgreifen.

Die SpEL kann, wie wir das schon bei `condition` und `unless` gesehen haben, auf Kontext-Objekte zurückgreifen:

- `#root.args[0]`, `#p0`, `#a0` oder einfach `#Parametername`
- `#root.method`, `#root.methodName`, `#root.target`, `#root.targetClass` …

So greift `"#prof.id"` auf die Parametervariable `prof` zurück und anschließend auf die `id`. Im Cache sind dann die Schlüssel vom Typ `Long` (Wrappertypen) und nicht vom Typ `Profile`.

Aufgabe: Den Schlüsselgenerator über key festlegen

Mit der nächsten Aufgabe lässt sich das Caching üben. Zur Erinnerung: `PhotoService` hatte eine Methode `download(String)`:

Listing 4.5 PhotoService.java

```
public class PhotoService {
  …
  public Optional<byte[]> download( String name ) {
    try { return Optional.of( fs.load( name + ".jpg" ) ); }
```

```
    catch ( UncheckedIOException e ) { return Optional.empty(); }
  }
  …
}
```

In `PhotoService` soll eine weitere überladene Methode `download(Photo)` ergänzt werden, sodass Aufrufer entscheiden können, ob sie das Bild über den Dateinamen oder über das `Photo` beziehen wollen (`Photo` enthält den Dateinamen). Mit `Photo` liegt ein komplexer Datentyp vor, und es wird gleich ein Schlüsselgenerator nötig:

```
public Optional<byte[]> download( Photo photo ) { implementMe }
```

Der Parametertyp `Photo` ist neu und soll so aussehen:

Listing 4.6 Photo.java

```
public class Photo {
  public Long          id;
  public Long          profile;
  public String        name;
  public boolean       isProfilePhoto;
  public LocalDateTime created;
}
```

Später werden wir die Klasse so weiter ausbauen, dass Fotos auch in der Datenbank gespeichert werden können.

Ein SpEL-Schlüsselgenerator soll implementiert werden, der `name` von `Photo` als Cache-Schlüssel extrahiert.

Der Lösungsvorschlag:

Listing 4.7 PhotoService.java, Erweiterung

```
@Cacheable( cacheNames = "date4u.filesystem.file",
            key        = "#photo.name" )
public Optional<byte[]> download( Photo photo ) {
  return download( photo.getName() );
}
```

Die Parametervariable heißt `photo`, und der SpEL-Ausdruck `#photo.name` referenziert damit den Namen, der ein String ist. Intern bildet der Cache somit Assoziationen zwischen dem Dateinamen und dem `byte`-Array.

Den Schlüsselgenerator über KeyGenerator festlegen

Kommen wir zur Option 2 zurück, einer eigenen `KeyGenerator`-Implementierung. SpEL-Ausdrücke sind praktisch und kompakt, aber schlecht typisiert und weniger effizient als direkter Java-Code.

Die Schnittstelle `KeyGenerator` schreibt eine Methode vor:

```
Object generate(Object target, Method method, Object... params)
```

Das folgende Beispiel liefert über eine `@Bean`-Methode einer `@Configuration`-Klasse eine `KeyGenerator`-Implementierung:

```
@Bean
public KeyGenerator photoNameKeyGenerator() {
  return ( Object __, Method ___, Object... params ) -> {
    if ( params.length == 1 && params[ 0 ] instanceof Photo photo )
      return photo.name;
    throw new UnsupportedOperationException(
        "Can't apply this KeyGenerator here" );
  };
}
```

Da `KeyGenerator` eine funktionale Schnittstelle ist, lässt sie sich kompakt über einen Lambda-Ausdruck implementieren. Aus der Parameterliste werden `Object target` und `Method method` nicht gebraucht, nur `params`, und zwar für die an den Cache-Proxy übergebenen Methodenargumente.

Es kann sein, dass der `KeyGenerator` falsch eingesetzt wird, daher folgt eine Konsistenzprüfung: `params.length` muss `1` sein, und der Typ muss `Photo` sein; gilt das, wird der Name extrahiert und zurückgegeben, andernfalls folgt eine Ausnahme.

Gesetzt wird die `KeyGenerator`-Implementierung nicht über ein `Class`-Objekt, sondern über den Bean-Namen; die Methode heißt `photoNameKeyGenerator`, und das ist auch der Name der Komponente. Er wird im Annotationsattribut `keyGenerator` gesetzt:

```
@Cacheable( cacheNames   = "date4u.filesystem.file",
            keyGenerator = "photoNameKeyGenerator" )
public Optional<byte[]> download( Photo photo )
```

Da `key` und `keyGenerator` beides Strings sind, besteht prinzipiell Verwechslungsgefahr.

4.2.10 @CacheConfig

Falls Angaben wie der Cache-Name für alle Methoden einer Klasse gelten, ist die Codeduplikation an jeder annotierten Caching-Methode nicht optimal. Eine andere Lösung bietet die Annotation `CacheConfig`[16], die an einem Typ gültig ist und bei `cacheNames` den Namen bestimmt.

Ein Beispiel:

```
@Component
@CacheConfig( cacheNames = "date4u.yamlhotprofiles" )
class HotProfileToYamlConverter {

  @Cacheable
  public String hotAsYaml( List<Long> ids ) { … }

}
```

Der Cache-Name kann bei den einzelnen Methoden entfallen. Methoden könnten den Namen allerdings auch neu setzen.

Insgesamt deklariert der Annotationstyp `org.springframework.cache.annotation.CacheConfig` vier Annotationsattribute:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface CacheConfig {
    String[] cacheNames() default {};
    String keyGenerator() default "";
    String cacheManager() default "";
    String cacheResolver() default "";
}
```

4.2.11 Cache-Implementierungen

Bisher haben wir über Annotationen deklarativ das Caching aktiviert, uns aber noch keine Gedanken über die eigentliche Cache-Implementierung gemacht. Standardmäßig nutzt Spring die Datenstruktur `java.util.concurrent.ConcurrentHashMap`. Eine `ConcurrentHashMap` ist ein besonderer Assoziativspeicher, der auf Hashtabellen basiert und nebenläufige Veränderungen erlaubt. Allerdings ist diese Datenstruktur kein

16 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/annotation/CacheConfig.html*

echter Cache, weil eine wichtige Eigenschaft fehlt: Ein guter Cache vergisst Elemente, die lange Zeit nicht verwendet wurden. Eine `ConcurrentHashMap` würde aber niemals Werte vergessen. Apropos Schule und vergessen: Eine Blütenpflanze besitzt einen Blütenboden, Kelchblätter, Kronblätter, Staubblätter und einen Stempel.

Verteilte vs. lokale Caches

Auf dem Markt gibt es eine große Anzahl von Cache-Implementierungen, die sich wie folgt einteilen lassen:

- **Verteilter (engl. distributed) Cache:** Daten liegen auf einem Cache-Server und nicht lokal. Alle Anwendungen teilen sich die Daten auf dem Cache-Server. Typische Produkte im Einsatz sind: *Redis*, *EhCache*, *Hazelcast*, *Infinispan*, *Couchbase*.
- **Lokaler Cache:** Jede Anwendung besitzt einen eigenen Cache im Hauptspeicher. Typische Java-Bibliotheken sind *Caffeine* und *cache2k*.

Spring erkennt anhand des Eintrags im Klassenpfad über eine Autokonfiguration, was vorhanden ist, und der Proxy leitet die Cache-Operationen an die Implementierungen weiter. Oft lassen sich über Konfigurations-Propertys Lebensdauer, Größe etc. extern konfigurieren, sodass der eigene Code nichts von einer Änderung der Cache-Implementierung mitbekommt.

4.2.12 Caching mit Caffeine

Als Beispiel für eine Cache-Implementierung schauen wir uns die Open-Source-Bibliothek *Caffeine* (*https://github.com/ben-manes/caffeine*) an, die aus dem Google-*Guava Cache* entstanden ist.

Zwei Dependencys sind in der POM nötig:

Listing 4.8 pom.xml, Erweiterung

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
<dependency>
  <groupId>com.github.ben-manes.caffeine</groupId>
  <artifactId>caffeine</artifactId>
  <version>3.1.2</version>
</dependency>
```

`spring-boot-starter-cache` fasst die zwei Dependencys `spring-boot-starter` und `spring-contextsupport` zusammen und hat selbst keinen Code.

Als Nächstes muss der Cache konfiguriert werden; das ist zum Beispiel über die *application.properties* möglich:

```
spring.cache.caffeine.spec=maximumSize=1,expireAfterWrite=10s
```

Hinter dem Schlüssel `spring.cache.caffeine.spec` steht ein String mit mehreren Segmenten; die einzelnen Segmente sind mit Komma separiert. Zum Testen lässt sich `maximumSize` auf `1` setzen, was bedeutet, dass nur maximal ein Element im Cache sein kann. (Grundsätzlich lässt sich Maximum-Size auf `0` setzen, das würde das Caching abschalten.) `expireAfterWrite` steht auf `10` Sekunden; also fliegt der Eintrag automatisch nach 10 Sekunden aus dem Cache.

Die Webseite *https://www.javadoc.io/doc/com.github.ben-manes.caffeine/caffeine/latest/com.github.benmanes.caffeine/com/github/benmanes/caffeine/cache/CaffeineSpec.html* erklärt die Konfigurationen etwas genauer. Kurz zusammengefasst, lässt sich Folgendes einstellen:

- `initialCapacity=[integer]`
- `maximumSize=[long]`
- `maximumWeight=[long]`
- `expireAfterAccess=[duration]`
- `expireAfterWrite=[duration]`
- `refreshAfterWrite=[duration]`
- `weakKeys`
- `weakValues`
- `softValues`
- `recordStats`

Gibt es keine Konfiguration von uns, gilt eine Standardkonfiguration.

4.2.13 Das Caching im Test abschalten

Im Testfall wird das Caching in der Regel abgeschaltet. Es gibt verschiedene Lösungsansätze, die auch auf andere Technologien übertragbar sind:

Option 1: Meistens gibt es einen Schalter, der etwas abschalten oder einschalten kann. So kann `spring.shell.interactive.enabled=false` die Shell abschalten und `spring.main.banner-mode=off` das Banner. Für das Caching lautet diese Konfigurations-Property `spring.cache.type`:

```
spring.cache.type=NONE
```

Die Belegung mit `NONE` schaltet das Caching ab.

Option 2: Spring Boot aktiviert über eine Autokonfiguration einen `CacheManager`[17]: Wenn es keinen Cache-Manager gibt, baut Spring eine Instanz auf. Wenn wir nun einen Cache-Manager selbst aufbauen, hat die Autokonfiguration nichts mehr zu tun und es gibt keinen Cache:

```
@Bean
CacheManager cacheManager() {
  return new NoOpCacheManager();
}
```

Die Fabrikmethode befindet sich wie üblich in einer `@Configuration`-Klasse und liefert den `NoOpCacheManager` zurück – wie der Name schon sagt: »No Op(eration)«, es passiert nichts.

Option 3: Das Caching benötigt einen Proxy-Generator, und den aktiviert `@EnableCaching`. Soll der Cache abgeschaltet werden, lässt sich das über die Aktivierung von `@EnableCaching` steuern. Ein Profil könnte zum Beispiel eine `@Configuration` aktivieren. Ein Beispiel: Befindet sich die Anwendung nicht im Development-Modus, also im Produktiv-Modus, soll erst diese Konfiguration mit `@EnableCaching` aktiviert werden:

```
@Profile( "!dev" )
@EnableCaching
@Configuration
public class CachingConfig { }
```

Ist man im Development-Modus, ist das Profil nicht aktiviert, folglich gibt es auch keine Konfiguration. Das bedeutet: Ohne Konfiguration gibt es auch kein `@EnableCaching`, also keinen Cache.

4.3 Asynchrone Aufrufe

Ein Spring-Proxy kann Operationen an einen Hintergrund-Thread abgeben. Damit kann ein Hauptprogramm die Arbeit fortführen und später die nebenläufig berechneten Ergebnisse abrufen.

17 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/CacheManager.html*