04-2013 | Winter | www.ijug.eu

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

Java aktuell

Java ist nicht zu bremsen

D: 4,90 EUR A: 5,60 EUR CH: 9,80 CHF Benelux: 5,80 EUR ISSN 2191-6977

iJUG Verbund

2013

Programm online

DOAG

Konferenz + Ausstellung

19.-21. Nov

Nürnberg NCC Ost

Keynote

Peter Kreuz und die Anstiftung zum Querdenken

Neue Gesichter

Datenakrobatik?
Nicht ohne Sicherheitsnetz!

Schulungstag

Java-Guru Ed Burns presents:
JavaEE 7 from a JSF perspective

Eine Veranstaltung der DOAG mit

ORACLE iJUG Verbund SOUG AOUG

2013.doag.org

Wolfgang Taschner
Chefredakteur Java aktuell

Java aktuell und die Community: Eine ideale Synergie

Innerhalb von wenigen Wochen kann der Interessenverbund der Java User Groups e.V. (iJUG) gleich drei neue Mitglieder gewinnen. Mit den Java User Groups Darmstadt, Karlsruhe und Hannover sind jetzt bereits zwei Drittel aller deutschsprachigen Java User Groups im iJUG organisiert. Damit vertritt der iJUG die Interessen von rund 20.000 aktiven Java-Entwicklern. Diese starke Community ist notwendig, um bei Herstellern und in der Öffentlichkeit Gehör zu finden.

Am Tag nach der iJUG-Mitgliederversammlung trifft sich die Java-Community beim Java Forum Stuttgart. Die sechshundert Hefte der Java aktuell sind im Nu vergriffen. Am Rande der Veranstaltung treffe ich mich mit den ehemaligen Sun-Mitarbeitern Peter Doschkinow und Wolfgang Weigend, die heute unter dem Dach von Oracle Deutschland die Java-Flagge hochhalten. In ihren zahlreichen Roadshows und Vorträgen bei den deutschsprachigen Java User Groups stellen sie unermüdlich die Java-Neuheiten vor. Wir vereinbaren für die Java aktuell eine neue Reihe zum WebLogic-Server, die in dieser Ausgabe beginnt. Autorin ist Silvie Lübeck, die in der Oracle-Business-Unit Server Technologies deutschlandweit die Middleware-Themen technisch und vertriebsunterstützend verantwortet.

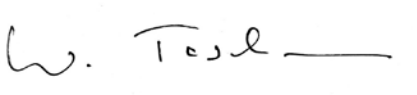
Auch auf dem JayDay 2013 erhalten alle Teilnehmer eine Ausgabe der Java aktuell. Die Konferenz für Java-Entwickler mit international anerkannten Experten findet seit Jahren in München statt. Die Talks drehen sich rund um Java 8, Java Web & EE, Java Embedded, JavaFX, Performance und Concurrency.

Einige Tage später telefoniere ich mit Ralph Müller, Director der Eclipse Foundation. Wir vereinbaren eine Medienpartnerschaft für die EclipseCon Europe 2013 im Oktober in Ludwigsburg bei Stuttgart. Auch auf dieser Veranstaltung werden wir die Java aktuell den rund 600 Teilnehmern zur Verfügung stellen. Darüber hinaus liefert die Eclipse Foundation künftig interessante Artikel für die Java aktuell.

Es bahnen sich weitere Ereignisse rund um die Java aktuell an. Lassen Sie sich überraschen. Ich freue mich wie immer auch über Ihr Feedback an redaktion@ijug.eu.

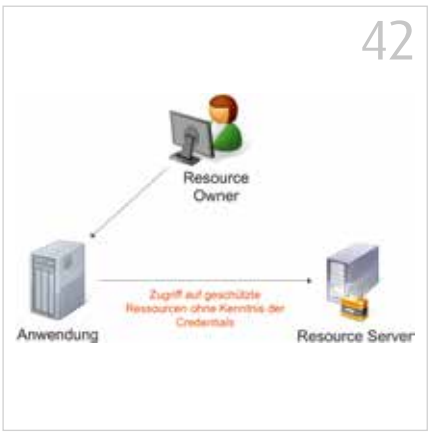
Viel Spaß beim Lesen wünscht Ihnen

Ihr



Java-Persistenz-Frameworks für MongoDB, Seite 20

Hibernate und Extra-Lazy Initialization, Seite 38



OAuth 2.0 – ein Standard, Seite 42

Universelles Ein-Klick-Log-in mit WebID, Seite 56

Das Java-Tagebuch

Andreas Badelt, Leiter der DOAG SIG Java

Das Java-Tagebuch gibt einen Überblick über die wichtigsten Geschehnisse rund um Java – in komprimierter Form und chronologisch geordnet. Der vorliegende Teil widmet sich den Ereignissen im zweiten Quartal 2013.

7. Mai 2013

Direkt vom Spec Lead: Beschreibung des „Protocol Upgrades" in Servlet 3.1

Der Veröffentlichungstermin von Java EE 7 rückt näher und die Specification Leads der darin enthaltenen JSRs machen kräftig Werbung für die Neuerungen, viele davon über praktische und anschauliche Beiträge auf „weblogs.java.net". Heute ist Shing Wai Chan, Specification Lead für JSR 340 (Java Servlet 3.1 Specification), an der Reihe mit dem Thema HTTP „Protocol Upgrade", mit dem Client und Server dynamisch alternative Protokolle aushandeln können. Vor Kurzem hatte er bereits über Non-Blocking I/O geschrieben.

https://weblogs.java.net/blog/swchan2/archive/2013/05/07/protocol-upgrade-servlet-31-example

Viele ähnliche Artikel über diesen und andere JSRs sind auch unter „blogs.oracle.com" beziehungsweise „weblogs.java.net" zu finden.

9. Mai 2013

Eclipse erhält kostenlose TCKs

In den vergangenen Jahren wurde zwischen Oracle (beziehungsweise vorher Sun) und der Apache Software Foundation viel über Test Compatibility Kits gestritten, genauer über deren kostenlose Nutzung. Diesmal gibt es eine positivere Nachricht, getreu dem Motto „Tue Gutes und rede darüber". Unter dem Scholarship Program für nichtkommerzielle Organisationen stellt Oracle der Eclipse Foundation TCKs für die Projekte „EclipseLink" (Referenz-Implementierung der Java-Persistence-API) und „Virgo" (ein leichtgewichtiger Applikationsserver) zur Verfügung. Virgo kann damit auf die Kompatibilität mit dem Java-EE-Web-Profile getestet werden (EE 6 und 7).

http://www.oracle.com/us/corporate/press/1943311

20. Mai 2013

Neues Versions-Nummerierungsschema für Java SE löst Diskussionen aus

Oracle stellt die Versions-Nummerierung für Java SE um. Auslöser waren wohl die vielen zusätzlichen Sicherheits-Patches in der letzten Zeit. Ziel ist, zwischen zwei geplanten Patch-Nummern immer noch Platz für einen kurzfristigen Patch zu haben und gleichzeitig anhand der Nummer zwischen funktionalen Erweiterungen beziehungsweise normalen Bug-Fixes (Limited-Update-Releases) und Sicherheits-Patches unterscheiden zu können. Dafür werden die Limited-Update-Releases immer mit einem Vielfachen von 20 bezeichnet (7u40, 7u60 etc.), die geplanten Sicherheits-Patches folgen dann mit um 5, 11 und 15 erhöhten Nummern (wie 7u45, 7u51, 7u55). Um dieses kleine Thema hat sich innerhalb von wenigen Tagen eine große Diskussion in der Community entwickelt. Grundstimmung: Warum wird die Versions-Nummerierung nicht grundsätzlich bereinigt, statt nun ein so verwirrendes Schema zu verwenden? Laut Oracle will man nach Problemen mit ähnlichen adhoc erfolgten Änderungen in der Vergangenheit vorsichtig sein: In Erinnerung ist noch die Umstellung des Hersteller-Strings von Sun auf Oracle – danach funktionierten tatsächlich einige Software-Produkte nicht mehr und Oracle wurde vorgeworfen, dass die Änderung ohne entsprechende Ankündigung und genügend Vorlauf erfolgt war. Eine Bereinigung komme daher erst nach Version 8 in Frage. Dabei wäre bis dahin noch ausreichend Zeit – anscheinend ist Oracle übervorsichtig geworden.

https://blogs.oracle.com/java/entry/new_java_se_version_numbering

22. Mai 2013

Offener Brief an Oracle zur Java- Versionsnummerierung

Der iJUG hat einen offenen Brief an Oracle verfasst, mit der Bitte, die Versions-Nummerierung bereits für Java 8 auf ein allgemeinverständliches Schema umzustellen. Vorgeschlagen wird, die bewährte dreiteilige Nummerierung in der Form „Version. Funktionsupdate.Sicherheitsupdate" zu nutzen, also 8.1.3 für Java 8, erstes Funktions- und drittes Security-Update. Ob Oracle seine Meinung noch ändert, bleibt abzuwarten.

http://www.ijug.eu/presse/presse-ijug/article/neue-versionierung-von-java.html

30. Mai 2013

Oracle gibt Statement zu Java-Security ab

Oracle hat in den ersten Monaten des Jahres viel Kritik für kritische Sicherheitslücken und zu langsame Reaktionen geerntet. Das hat sich der Konzern offensichtlich zu Herzen genommen. Dass mehr in den Bereich „Sicherheit" investiert wird und Oracle auf die Community eingeht (unter anderem durch Meetings mit Java User Group Leads), war bereits Thema im letzten Java-Tagebuch. Jetzt fasst Nandini Ramani, Entwicklungsleiter für die Java-Plattform, in einem Blog-Eintrag noch einmal zusammen, was Oracle bereits getan hat, und kündigt gleichzeitig weitere Maßnahmen an: Java wird nun in das reguläre Update-Konzept von Oracle aufgenommen; damit

wird es in Zukunft vier statt drei geplanter Sicherheits-Updates („Critical Patch Updates") pro Jahr geben. Darüber hinaus sei viel in den Bereich der automatisierten Security-Tests investiert worden, um Schwachstellen im Code möglichst früh zu entdecken. Außerdem will Oracle in erweiterte Möglichkeiten für Administratoren investieren („local security policies"), sodass beispielsweise Applets nur noch von definierten Hosts geladen werden dürfen. Auch das neue ServerJRE soll weiter optimiert werden (Entfernen nicht benötigter Libraries), um die Angriffsfläche weiter zu reduzieren.
https://blogs.oracle.com/security/entry/maintaining_the_security_worthiness_of
Ebenfalls auf das Thema „Qualitätsverbesserung", aber weniger mit dem Fokus auf Sicherheit, zielt ein Blog- Eintrag von Tori Wieldt, Senior Java Developer Community Manager bei Oracle: Sie wirbt darum, dass Entwickler sich bereits jetzt das neueste OpenJDK 8 herunterladen und testen, um frühzeitig Probleme melden oder Verbesserungsvorschläge einreichen zu können. Diese Bitte gebe ich hier gerne weiter.
https://blogs.oracle.com/otn/entry/feedback_wanted_java_se_8

5. Juni 2013

Sicherheitslücken in Struts
Wieder das Thema „Sicherheit", diesmal steht aber nicht die Plattform an sich im Mittelpunkt, sondern das Struts-Projekt von Apache – es geht also um serverseitige Sicherheitslücken. Heise News meldet, dass Apache innerhalb einer Woche gleich zwei Security-Updates herausbringen muss, um hochkritische Schwachstellen zu beheben, durch die ansonsten eingeschleuster Code durch doppelte Evaluierung von OGNL-Ausdrücken, einer Art Expression Language, auf dem Server zur Ausführung gebracht werden könnte. Struts ist nicht das einzige Framework, das solche Probleme hat, auch wenn es hier eine gewisse Historie gibt, gerade zum Thema OGNL-Ausdrücke. Ein kleiner Warnschuss, sich beim Thema „Security" nicht nur auf die Plattform beziehungsweise das Browser- Plug-in zu fokussieren sondern auch serverseitig immer auf der Hut zu sein und die neuesten Patches einzuspielen.
http://www.heise.de/security/meldung/Apache-muss-Struts-erneut-patchen-1883040.html

8. Juni 2013

„Java.net"-Umfrage zur Bedeutung der JVM in fünfzehn Jahren
Eine Umfrage auf „java.net" hat Folgendes ergeben: 85 Prozent der Teilnehmenden glauben daran, dass Java und andere JVM-Sprachen auch in fünfzehn Jahren noch eine bedeutende Rolle für neue Software-Projekte spielen. Die Einschränkung war immerhin auf neue Projekte bezogen, sodass die gigantische Basis von existierenden Java-Applikationen keinen Einfluss hatte. Da aber davon auszugehen ist, dass die meisten Teilnehmer sich nicht zufällig auf „java.net" verirrt haben, wäre es interessanter zu erfahren, was die anderen 15 Prozent denken – und vor allem, wie vergleichbare Umfragen etwa für „.net" ausgehen würden. Eine Analogie lässt sich leider nicht so einfach finden, aber eine interessante Zahl liefert ausgerechnet das „Visual Studio Magazine" auf seiner Webseite: Dort wurde Ende 2011 und 2012 jeweils eine Umfrage unter den Abonnenten durchgeführt. Neben einer Menge anderer Statistiken steht dort unter „Languages Used for Software Development Projects in the Last 12 Months" auch Java, mit 23,7 (2011) beziehungsweise 29,7 Prozent (2012). Ohne die Statistik jetzt überbewerten zu wollen, klingt das doch zumindest sehr nett für Java-Freunde.
https://weblogs.java.net/blog/editor/archive/2013/06/08/poll-result-java-and-other-jvm-languages-and-future-15-years-now

11. Juni 2013

Referenz-Implementierung für Java EE 7: GlassFish 4.0 erschienen
Zwei Tage vor dem offiziellen Launch von Java EE 7 ist mit GlassFish 4.0 die Referenz-Implementierung freigegeben worden. Andere Hersteller werden sicherlich in den nächsten Monaten nachziehen – zu Java EE 6 gibt es mittlerweile mehr als zehn kompatible Applikationsserver, wenn man die kommerziellen und freien Editionen getrennt zählt, sowie eine Reihe weiterer Produkte, die das eingeschränkte Web Profile implementieren.
http://www.heise.de/developer/meldung/Referenzimplementierung-fuer-Java-EE-7-GlassFish-4-0-erschienen-1886175.html

TimeUpdater wieder frei verfügbar
Mit dem letzten Update von Java SE 6 im April hatte Oracle ein unscheinbares Tool entfernt, das von da an nur noch über den kostenpflichtigen Support bezogen werden kann, den Time Zone Updater (tzupdater). Dieses Tool wird benötigt, um das JRE mit den aktuellen Informationen zu Zeitzonen zu patchen, die sich tatsächlich häufiger ändern, als man erwarten würde. Es hat einige Zeit gedauert, bis sich die Information verbreitet hat. Ein Artikel von „The Server Side" vor einigen Tagen hat eine heftige Diskussion hervorgerufen. Nachdem sich auch die Gruppe der JUG-Leader, die in regelmäßigem Kontakt mit Oracle stehen, der Sache angenommen hatte, ging es aber ganz schnell: Jetzt ist das Tool wieder zum freien Download verfügbar.
http://www.theserverside.com/discussions/thread.tss?thread_id=75820#460568
https://blogs.oracle.com/henrik/entry/tzupdater_for_jdk_7_available

12. Juni 2013

Java EE 7 ist da
Oracle hat offiziell Java Enterprise Edition in der Version 7 freigegeben. Das erste komplett unter Oracle- Obhut entstandene Release enthält nicht die ursprünglich geplanten Cloud-Funktionalitäten. Erst relativ spät war klar geworden, dass das Thema „Cloud" noch nicht reif für eine Standardisierung im JCP ist. Entsprechend wurde es, wie auch die Aufnahme des JCache JSRs, auf Release 8 verschoben, also voraussichtlich auf Anfang 2015. Dafür enthält Version 7 eine ganze Reihe von neuen und aktualisierten Spezifikationen, die Entwicklern die Arbeit erleichtern und bestehende Funktionalität sinnvoll erweitern. Unter anderem sind dies vereinfachte APIs inklusive Dependency Injection für JMS, HTML 5-Konformität für JavaServer Faces und nicht- blockierendes I/O für Servlets. Komplett neu dazugekommen sind Unterstützung für JSON, Batch-Verarbeitung und programmatisch gesteuerte

Nebenläufigkeit im Server (letztere basierend auf den Concurrency Utilities für Java SE), sowie WebSockets zur direkten Kommunikation zwischen Server und Client.
https://blogs.oracle.com/java/entry/java_ee_7_moving_java
Zeitgleich mit dem Release sind auch die neuen Java EE Tutorials fertiggestellt worden – mit Beispielen zu allen neuen und geänderten JSRs, und basierend auf Maven als Build-Tool.
https://blogs.oracle.com/thejavatutorials/entry/java_ee_7_tutorial_released

18. Juni 2013

JSR-Updates
Rund um den Launch von Java EE 7 und der entsprechenden neuen beziehungsweise auf neue Versionen gehobenen Einzel-JSRs im „Umbrella JSR" sind auch eine ganze Reihe von Maintenance-Releases älterer Versionen final geworden, beispielsweise von EJB 3.1, JCA 1.6 und JSP 2.1.
https://blogs.oracle.com/jcp/entry/jsr_updates_june_2013

GlassFish 4: Java EE 7 in der Cloud
Als „first mover" präsentiert CloudBees die Unterstützung für Java EE 7. Die Referenz-Implementierung Glassfish 4 wird als „Platform as a Service" (PaaS) verfügbar gemacht: Einrichten und Starten mit ein paar Mausklicks mit einer Einschränkung: Bislang wird nur das Web Profile unterstützt und es handelt sich bislang um einen „Community Stack", er wird also nicht offiziell von CloudBees unterstützt („use at your own risk").
http://www.cloudbees.com/press-room/cloudbees-first-paas-enable-java-ee-7-development-cloud.cb

20. Juni 2013

Eclipse-Projekte nun auch auf „Social Coding"-Plattformen
Die Eclipse Foundation erlaubt ab sofort ihren Projekten nicht nur die internen Repositories, sondern externe „Social Coding" Plattformen zu nutzen. Den Anfang macht GitHub, aber auch BitBucket soll folgen. Dies kündigt Executive Director Mike Milinkovich in einem aktuellen Blog-Eintrag an. Über diesen Schritt wurde laut Milinkovich wohl schon einige Zeit diskutiert, um die Schwelle, sich an Eclipse-Projekten zu beteiligen, möglichst gering zu halten. Der konkrete Auslöser war nun die Übergabe des „vert.x"-Projekts an die Eclipse Foundation. Dieses wird auf GitHub gehostet.

Da bereits seit einiger Zeit Eclipse-Projekte einen Mirror auf GitHub haben können, hat man dieses Prinzip nun einfach umgedreht: „vert.x" bleibt auf GitHub und wird einen Mirror im Eclipse-Git-Repository haben.
http://mmilinkov.wordpress.com/2013/06/20/embracing-social-coding-at-eclipse

21. Juni 2013

Was sollte der Java Community Process machen?
Die Londoner Java Community hat in Kooperation mit den JCP-Verantwortlichen eine Umfrage gestartet: „Was sollte der JCP machen?" Ziel der knapp gehaltenen Umfrage ist es, ein Stimmungsbild der Community einzufangen, ob die Standardisierung im JCP als sinnvoll eingeschätzt wird, welche Themen abgedeckt werden sollten, und, ob die Teilnehmer sich am JCP beteiligen, beziehungsweise wenn nicht, warum. Die Ergebnisse sollen im Rahmen des JCP öffentlich gemacht werden, im nächsten Java-Tagebuch wird es sicher einen Eintrag dazu geben.
https://blogs.oracle.com/java/entry/jcp_survey

4. Juli 2013

Java EE 7 Javadocs sind online
Mit ein bisschen Verzögerung, dafür aber im neuen Design, sind jetzt auch die Javadocs für EE 7 online verfügbar.
http://docs.oracle.com/javaee/7/api

8. Juli 2013

JCache erreicht Public Review-Phase
JCache, das (wie im letzten Java-Tagebuch berichtet) leider die Aufnahme in Java EE 7 verpasst hatte, macht trotzdem unverdrossen weiter und hat jetzt die Public-Review-Phase eingeleitet. Die Feedback-Runde läuft bis zum 5. August 2013, anschließend wird über den JSR abgestimmt. Bei einem positiven Resultat (und etwas anderes ist bei der inzwischen sehr öffentlichen Ausführung von JSR in dieser Phase eigentlich nicht mehr zu erwarten), kann zumindest für Java EE 8 nichts mehr schief gehen.
https://blogs.oracle.com/theaquarium/entry/jcache_marches_onward

19. Juli 2013

Nashorn und Lambda passen zusammen
Eine kleine Vorschau davon, was demnächst mit der neuen JavaScript-Engine Nashorn und Lambda-Expressions im JDK 8 möglich ist (oder auch jetzt schon in der Beta-Version), gibt Jim Laskey. Die beiden Projekte scheinen perfekt zueinander zu passen (eine sehr kompakte Syntax).
https://blogs.oracle.com/nashorn/entry/nashorn_and_lambda_what_the

23. Juli 2013

Neue Versionen von Java ME Embedded und Java ME SDK
Oracle adressiert die wachsende Nachfrage nach offenen und branchenübergreifenden Plattformen für das „Internet der Dinge". Dafür bieten Java ME Embedded und Java ME SDK erweiterte Unterstützung für marktführende Embedded-Chip-Architekturen und neue Binaries für Entwickler-Boards, die auf der ARM-Architektur basieren, einschließlich Raspberry Pi und Keil STM32 F200 Evaluation Board für ARM Cortex-M-Prozessor-basierte Geräte.
http://www.oracle.com/technetwork/java/embedded/downloads/javame/index.html

Andreas Badelt
Leiter der DOAG SIG Java

Andreas Badelt ist Senior Technology Architect bei Infosys Limited. Daneben organisiert er seit 2001 ehrenamtlich die Special Interest Group (SIG) Development sowie die SIG Java der DOAG Deutsche ORACLE-Anwendergruppe e.V.

UPDATE An dieser Stelle erhalten Sie in jeder Ausgabe ein Update über das Geschehen in der Java-Community

Java Enterprise Edition 7: Mobile statt Cloud

Markus Eisele, msg systems ag

Es hat nur wenig mehr als drei Jahre gedauert, um aus der „6" eine „7" zu machen. Gemeint ist die Versionsnummer der Java Enterprise Edition (EE). Ursprünglich war die siebte Ausgabe stark auf das Thema „Cloud" ausgelegt. Die Pläne stellten sich erst spät als zu ambitioniert heraus. Somit enthält die am 16. April 2013 fertiggestellte Version nur punktuell grundlegend Neues und stellt eine konsequente Abrundung der bereits vorhandenen Funktionen dar. Am 12. Juni 2013 feierte Oracle das fertige Werk mit einem internationalen Launch-Event.

Mit der Version 7 wurde die Java Enterprise Edition erstmalig komplett unter Oracles Obhut durchgeführt. Sie umfasst als sogenannter „Umbrella" (Regenschirm) eine Menge von einzelnen Technologien und liefert die grundlegenden Regelungen für deren Zusammenarbeit. Eines der Hauptziele seit Einführung der Java-EE-Plattform war, die Entwickler von allgemeinen Infrastruktur-Aufgaben zu befreien. Im Laufe der Jahre wurden dem Umbrella weitere Teile hinzugefügt und seit Anfang 2006 stand schließlich die Verbesserung der Entwicklungsfreundlichkeit an höchster Stelle. Entlang dessen folgten mit Java EE 6 weitere Verbesserungen und durch die Einführung der Profile auch eine deutliche Verschlankung der Plattform. Die neue Java EE 7 folgt diesem Trend konsequent (siehe Abbildung 1).

Den 28 Spezifikationen in EE 6 sind vier vollständig neue hinzugekommen. Bei drei bestehenden wurden umfangreiche inhaltliche Erweiterungen vorgenommen. Bei den verbleibenden wurde weitestgehend Modellpflege betrieben. Die in EE 6 bereits zum Entfernen vorgeschlagenen Spezifikationen sind in dieser Version als „optional" gekennzeichnet (EJB Entity Beans, JAX-RPC 1.1, JAXR 1.0, und JSR-88 1.2).

Komplett neu hinzugekommen sind APIs zur Arbeit mit den aktuellen Technologien im Web-Bereich. Neben vollwertigen Spezifikationen wie Web Sockets (JSR-356) und JSON (JSR-353) ist auch eine HTML5-Unterstützung in den Java Server Faces eingebaut worden.

Der JSR 352 standardisiert ein Programmier-Modell für Batch-Anwendungen und bietet eine Laufzeitumgebung für Planung und Ausführung entsprechender Jobs. Einen verbesserten Zugriff auf die Ressourcen der Container ermöglichen die Concurrency Utilities. Mit diesem API ist jetzt zuverlässig und in Synchronisation mit den Containern die Arbeit mit Threads in Java EE möglich (siehe Abbildung 2).

JAX-RS 2.0

Die Version 2.0 der „Java API for RESTful Web Services" (JAX-RS) bringt wichtige Neuerungen. Herausragendes Einzelmerkmal ist das neue Client-API, um aus Java heraus direkt auf Endpoints zuzugreifen. Damit entfällt die Notwendigkeit, Bibliotheken von Dritt-Anbietern einsetzen zu müssen. Im folgenden Beispiel werden ein neuer Client und eine Anfrage an den Server-Endpoint gestellt (siehe Listing 1).

Neu hinzugekommen ist die asynchrone Bearbeitung von Requests zwischen Client und Server. In Anlehnung an die neuen asynchronen Funktionen seit Servlet 3.0 erfolgt sie mit einem AsyncRequest (siehe Listing 2).

Mithilfe von Message-Filtern und Entity Interceptors kann man auf Request und Response zugreifen; beide können server-

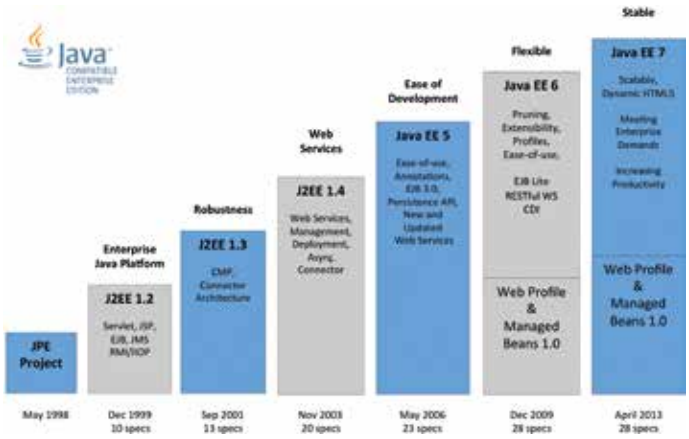


Abbildung 1: Die Geschichte von Enterprise Java

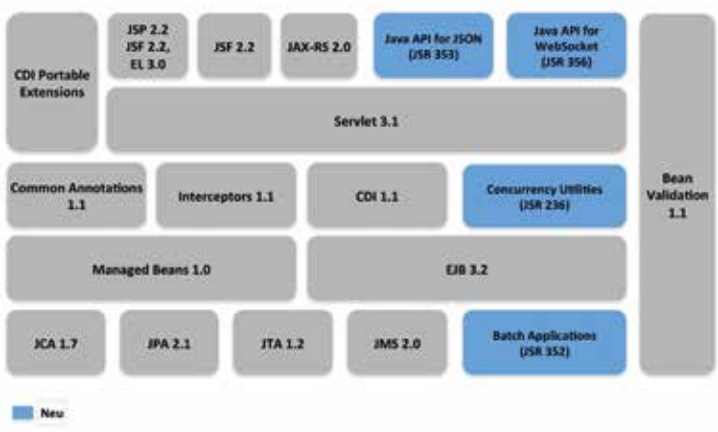


Abbildung 2: Die wichtigsten Änderungen auf einen Blick

```
Client client = ClientFactory.newClient();
String name = client.target("/category/{number}/name")
                .resolveTemplate("number", "5")
                .request()
                .get(String.class);
```

Listing 1

```
@GET
@Produces("application/plain")
public void longRunning(@Suspended final AsyncResponse ar) {
        Executors.newSingleThreadExecutor().submit(
                        new Runnable() {
                @Override
                public void run() {
                        longRunning();
                        ar.resume(„Hallo Welt!");
                }
        });
 }
```

Listing 2

```
@Provider
class LoggingFilter implements RequestFilter {
@Override
public FilterAction preFilter(FilterContext ctx) throws IOException {
        log(ctx.getRequest());
        return FilterAction.NEXT;
}
//…
}
```

Listing 3

```
@Logged
@GET
@Produces("text/plain")
@Path("/hello")
public String hello(){
return "Hello from @ijUGeV!";
}
```

Listing 4

seitig bearbeitet werden. Auch beliebte Aspekte wie Logging lassen sich elegant mithilfe von Filtern umsetzen (siehe Listing 3). Dieser Filter ist dann nur noch über „@Qualifyer" einzubinden und kann direkt an einer JAX-RS-Ressource verwendet werden (siehe Listing 4).

Das Arbeiten mit eingebauten Klassen wie „String" ist damit vergleichsweise einfach. Bei der Arbeit mit eigenen Objekten ist das Mapping zwischen diesen und der serialisierten Ausgabe (JSON, XML) manuell zu erledigen. Die technischen Möglichkeiten bietet die Spezifikation mithilfe sogenannter „Entity Provider". Diese können die Nutzlast der Nachrichten vom einen in das andere Format umwandeln. Analog dazu wurden die beiden zuständigen APIs „MessageBodyReader" und „MessageBodyWriter" benannt. Listing 5 zeigt, wie ein entsprechender Reader für ein fiktives Objekt „Person" aussieht.

Die Entity Provider ermöglichen dabei lediglich den Einsprung in JAX-RS. Die konkrete De-/Serialisierung der Objekte muss noch implementiert werden. Leider bieten hier weder Spezifikation noch Referenz-Implementierung serienmäßig etwas an. Der Entwickler hat die Wahl, einen der bereits bestehenden Mechanismen wie beispielsweise EclipseLink mit MOXy (siehe Links und Literatur) einzusetzen oder selbst tätig zu werden.

```
@Provider
@Produces("application/json")
public class PersonReader implements MessageBodyReader<Person> {
//…
@Override
public void readFrom(Person p, Class<Person> type, Type t, Annotation [ ] as, MediaType mt,
MultivaluedMap<String, Object> mm, OutputStream out) throws IOException, WebApplicationException {
//…
}
```

Listing 5

Java API for JSON Processing

Mit dem Java API for JSON Processing steht jetzt ein Werkszeug zur Arbeit mit JSON im Standard bereit. Im Angebot sind ein Streaming-API (JsonParser, JsonGenerator) und ein DOM-API (JsonReader, JsonWriter). Während das Streaming-API ereignisgetrieben agiert, kann mit dem DOM-API auf Objektebene gearbeitet werden. Um mit dem Streaming-API JSON auf System.out zu schreiben, ist lediglich der JsonGenerator erforderlich (siehe Listing 6). Der JsonWriter geht dabei objektgetrieben vor und ermöglicht die Erzeugung spezieller JSON-Datentypen (siehe Listing 7).

Java Server Faces

Passend zur Unterstützung aktueller Web-Technologien versteht sich JSF jetzt auch mit HTML5-konformem Markup. Die als „data-*" bekannten eigenen Daten-Attribute wurden bisher von den JSF-Komponenten verworfen und nicht gerendert. Genau dieses ist jetzt mit einem neuen JSF-Tag möglich (siehe Listing 8).

Alle JSF-Komponenten erkennen diesen neuen Weg. Abgesehen von dieser Kleinigkeit gibt es viele weitere Änderungen. Dazu gehören die neuen View Actions, um Request-Variablen an das Modell zu binden, eine AJAX-basierte File-Upload-Komponente, die Einführung von Stateless Views, bessere Integration von CDI inklusive der Anpassung der Scopes und vieles mehr.

```
JsonGeneratorFactory factory = Json.createGeneratorFactory();
JsonGenerator gen = factory.createGenerator(System.out);
Einzelne Objekte werden der Ausgabe direkt hinzugefügt:
gen.writeStartObject().write("Java EE","7").write("Java Aktuell","iJUG").writeEnd();
```

Listing 6

```
JsonArray jsonArray = new JsonArrayBuilder().add(new JsonObjectBuilder().add("Java EE","7")).
add(new JsonObjectBuilder().add("Java Aktuell","iJUG")).build();
```

Listing 7

```
<h:inputText value="#{person.name}">
<f:passThroughAttribute name="data-length" value="10"/>
</h:inputText>
```

Listing 8

```
@Named
@FlowScoped(id = "myFlowName")
public class MyBean {
        public String myReturn() {
                return "/myPage.xhtml";
        }
}
```

Listing 9

Eines der großen neuen Funktionspakete stellt Faces Flow dar. Hierbei handelt es sich um eine Erweiterung der Navigation, um einfacher geführte Dialogstrecken für Anwender zur Verfügung zu stellen. Diese heißen „Wizzard" und haben neben einer definierten Abfolge von Dialogen auch einen eigenen Gültigkeitsbereich für Daten. Flows können über den „<f:metadata>"-Tag direkt in Facelets verwendet, aber auch programmatisch erzeugt werden. Man startet sie ähnlich der impliziten Navigation direkt als „action", beispielsweise über „<h:commandLink value="Enter flow" action="myFlowName" />". Sofern definiert, ist ein zugehöriges JavaBean während des Flows verfügbar (siehe Listing 9).

Servlets

Hinter der neuen Minor-Version der Servlets verbergen sich hauptsächlich drei kleinere Erweiterungen. Wichtigster Aspekt ist die Umsetzung des seit HTTP 1.1 (RFC 2616) möglichen Protokoll-Upgrades. Dieser Mechanismus macht die neue WebSockets-Spezifikation erst möglich. Der neu verfügbare „javax.servlet.http.HttpUpgradeHandler" ist dafür zuständig.

Auch in Sachen „Sicherheit" hat sich etwas getan. So werden bisher nur die http-Methoden geschützt, die auch explizit in der „web.xml" definiert sind. Vielfach sind das bisher nur „get" und „post" gewesen. Die übrigen (head, put, delete etc.) verblieben zumeist per Standardeinstellung ungeschützt. Dies kann nun umgedreht und somit alles geschützt werden, was nicht explizit freigegeben wird („<deny-uncovered-http-methods/>"). Eine neue Semantik „jeder authentisierte user" vereinfacht die Prüfung auf die Gesamtsumme aller Rollen. Dabei gibt es folgenden Unterschied: „*" bezieht sich auf jede in der „web.xml" definierte Rolle und „**"auf jeden authentifizierten Nutzer.

Bean Validation 1.1 und EL 3.0

Die Zusammenarbeit mit der Bean-Validation-Spezifikation ist deutlich verbessert. So lassen sich jetzt Methoden-Parameter validieren; nicht nur mit Standard-Validatoren (@NotNull), sondern auch mit eigenen, nutzerdefinierten Constraints. Damit ist auch gleich die größte Neuerung in Bean-Validation beschrieben. Neben der Methoden-Validierung können auch Constructor-Validierungen durchgeführt werden, hier wurde ebenfalls die Integration in die Context- und Dependency-Injection-Spezifikation (CDI) vorangetrieben; es ist jetzt also beispielsweise möglich, „@Inject" in Validatoren zu verwenden. In den Meldungen gibt es nun Expression-Language-Ausdrücke. Das führt zu einer deutlich flexibleren Darstellung (siehe Listing 10).

Die Expression Language (EL) wurde in der Version 3.0 deutlich weiterentwickelt. Neben neuen Operatoren sind nun Collections und Lambda-Ausdrücke möglich. Letztere allerdings nur zukünftig unter Java SE 8. Die EL kann nun auch alleinstehend verwendet werden (siehe Listing 11).

Es lassen sich jetzt auch eigene Resolver oder Klassen hinzufügen. Eigene Typ-Con-

```
javax.validation.constraints.DecimalMax.message= Muss weniger  ${inclusive == true ? ‚gleich sein
zu , : ,'}{value}
```

Listing 10

```
ELProcessor el = new ELProcessor;
Object result = el.eval("'Hello, ' + user.nachname");
```

Listing 11

```
@Resource(lookup="DefaultManagedExecutorService")
ManagedExecutorService executor;
```

Listing 12

```
public class MeineAufhabe implements Runnable {
        @Override
        public void run() {
                //. . .
        }
}
```

Listing 13

verter und „EvaluationListener“ runden das Bild ab und machen es zu einer komplett eigenständigen und sehr mächtigen Spezifikation.

Concurrency-Utilities

Mithilfe der Concurrency-Utilities lassen sich über ein einfaches API konkurrierende Aktionen von Anwendungskomponenten ausführen, ohne die Container-Integrität zu gefährden. Die direkte Verwendung von Java-SE-Concurrency-APIs in Java-EE-Containern ist nicht erlaubt, weil diese sich der Ressourcen-Verwaltung durch die Container entziehen und zu Stabilitätsproblemen führen. Daher sind die Concurrency-Utilities im Grunde nicht mehr als eine Java-EE-konforme Erweiterung des JSR-166 in Java SE. Die dort vorhandenen Funktionen werden nunmehr vom Container verwaltet. Statt eines „java.util.concurrent.ExecutorService“ gibt es jetzt das Enterprise-Gegenstück „javax.enterprise.concurrent.ManagedExecutorService“. Eine Standard-Instanz steht via JNDI bereit, kann aber auch direkt injiziert werden (siehe Listing 12). Die auszuführende Aufgabe heißt „Task“ und implementiert dabei das Runnable oder Callable Interface (siehe Listing 13). Sie wird dann über den ManagedExecutorService ausgeführt (siehe Listing 14).

Neben der Task ohne Rückgabewert gibt noch den „Callable“, der einen Rückgabewert haben kann. CDI-Beans können ebenfalls eine Task sein, so lange sie den Scope „@Application“ oder „@Dependent“ haben. Mit „javax.enterprise.concurrent.ManagedScheduledExecutorService“ lassen sich zeitgesteuerte Aufgaben ausführen. Der Aufruf erfolgt analog zum „ExecutorService“ zuzüglich der relevanten Zeiten (siehe Listing 15).

Der Code führt die Aufgabe nach 15 Sekunden Verzögerung aus. Der sogenannte „Container-Context“ stellt Funktionen von Anwendungskomponenten wie „Classloader“, „Namespaces“ und „Security“ bereit.

```
Collection tasks = new ArrayList();
tasks.add(new MeineAufhabe());
executor.invokeAny(tasks);
```

Listing 14

```
ScheduledFuture<?> f = executor.schedule(new MeineAufhabe (), 15, TimeUnit.SECONDS);
```

Listing 15

Web Sockets

Als bidirektionales Kommunikations-Protokoll über TCP erlauben WebSockets die direkte Kommunikation zwischen Server und Client. Dabei sind die WebSockets ein „IEFT RFC“, der in HTML5-fähigen Browsern mit zugehörigem JavaScript-API genutzt werden kann. Im Gegensatz zu http wird eine bestehende TCP-Verbindung dabei persistent aufrechterhalten. Der aktuelle JSR-356 fokussiert auf die serverseitige Bereitstellung der WebSocket-Endpoints, die Entgegennahme von Text, Binär und Kontroll-Nachrichten, die Lebenszyklus-Events und die Integration in Java-EE-Security. Listing 16 zeigt einen einfachen Endpunkt für einen Chat.

Die mit „@OnMessage“ annotierte Methode wird beim Empfang einer Nachricht eines Clients verwendet. Je nach Nachrichtenformat sind über den Methoden-Parameter die Behandlung von Textnachrichten via „String“, Binärnachrichten via „byte[ ]“ und Streams via „InputStream“ möglich. Mit „@PathParam“ können primitive Java-Typen oder Strings im Server-Endpoint verwendet werden. Neben „@OnMessage“ gibt es noch Lebenszyklus-Methoden (@OnOpen, @OnClose und @OnError), die ausimplementiert werden können.

CDI wird vollständig unterstützt. Alternativ zu den Annotationen kann ein Server-Endpoint auch via API komplett programmatisch erstellt und somit auch in Java SE bereitgestellt werden. Im Idealfall ist ein im Browser ausgeführter JavaScript-Client die Gegenstelle für den Server (siehe Listing 17).

Auch hier stehen wieder die Lebenszyklus-Events mit den entsprechend annotierten Methoden analog zum Server zur Verfügung. Mit Encodern und Decodern lassen sich analog zu JAX-RS applikationsspezifische Datentypen serialisieren. Provider können hier leider nicht einfach registriert, sondern müssen dem Server-Endpoint mitgegeben werden (siehe Listing 18). Die bekannten Security-Constraints aus dem „web.xml“-Deployment-Descriptor greifen auch für WebSocket-Endpoints.

```
@ServerEndpoint(“/chat/{raum}“)
public class ChatServer {
@OnMessage
public void receiveMessage(String message, @PathParam(“raum”) String raum) {
//. . .
}
}
```

Listing 16

```
var websocket = new WebSocket(“ws://localhost:8080/chatapp/chat“);
Aber auch Java Klassen können Client sein. Dafür wird die @CllientEndpoint Annotation angeboten:
@ClientEndpoint
public class ChatClient {
//. . .
}
```

Listing 17

```
@ServerEndpoint(value = “/personservice“,
encoders = {PersonEncoder.class},
decoders = {PersonDecoder.class})
```

Listing 18

```
TextMessage textMessage = context.createTextMessage(body);
 context.createProducer().setPriority(1).setProperty("foo","bar").send(demoQueue, textMessage);
```

Listing 19

```
context.createProducer().send(demoQueue,"Hallo Welt!");
```

Listing 20

```
JMSConsumer consumer = context.createConsumer(demoQueue);
String result = "Empfangen" + consumer.receiveBody(String.class, 1000);
```

Listing 21

JMS 2.0

Lange erwartet wurde die neue Version der Java-Message-Services. Mit fast neun Jahren ohne nenneswerte Weiterentwicklung hat sich hier auch am meisten verändert. Durch die Anwendung der mit Java SE 7 neu hinzugekommenen Fähigkeiten wie beispielsweise „Try-with-resources", aber auch dank CDI ist die Verwendung des API sehr schlank und einfach geworden. So sind Connection- und Session-Objekte im neuen „JMSContext" zusammengefasst. Statt mit beiden arbeiten zu müssen, reicht jetzt der neue Context, der auch injiziert werden kann. Er muss also weder explizit erstellt noch geschlossen werden: „@Inject JMXContext context;". Die Vereinfachung der APIs wurde jedoch noch weiter vorangetrieben. So kann dank neuer Producer-Methoden jetzt flüssiger gearbeitet werden. Das Senden einer JMS-Message auf dem neuen Weg schrumpft zum Zweizeiler (siehe Listing 19).

Dabei sind die Producer sehr leichtgewichtig und können bei Bedarf erzeugt werden. Die Notwendigkeit, sie zwischenzuspeichern, entfällt komplett. Überhaupt ist ein Nachrichten-Objekt nicht mehr zu instanziieren. In JMS 2.0 kann man den gewünschten Payload auch direkt dem Producer übergeben (siehe Listing 20). Andersherum entfällt beim Empfangen nunmehr ein „Casten" auf eine genaue Klasse (siehe Listing 21).

Vereinfachtes Arbeiten und weniger Code sind damit endlich auch im JMS möglich. Dabei wirken sich die Änderungen in diesem Punkt nicht nur auf die Arbeit in der EE-Spezifikation, sondern auch auf Java SE aus. Im EE-Container erfordern die Neuerungen den Einsatz eines Resource-Adapters, was aber für die Entwickler unbemerkt durch die Server erfüllt werden sollte.

Batch-Anwendungen

Angelehnt an die Konzepte von „Spring Batch" entstand unter der Regie von IBM der JSR-352. Grundidee ist die Verarbeitung einer Serie von Jobs, die ohne Nutzer-Interaktion Massendaten in lang laufenden Schritten bearbeiten. Ein „Job" ist dabei das Haupt-Objekt. Es untergliedert sich in einzelne Schritte („Steps"), die nach bekanntem Batch-Pattern (Lesen, Bearbeiten, Schreiben) Aufgaben ausführen. Gestartet werden Batch-Jobs über den sogenannten „Batch-Operator" (siehe Listing 22).

In diesem Beispiel werden dem Batch-Job noch Properties mitgegeben. Diese werden zusammen mit dem Namen der eigentlichen Batch-Beschreibung der „start"-Methode übergeben (siehe Listing 23).

```
JobOperator jo = BatchRuntime.getJobOperator();
        Properties jobParams = new Properties();
        jobParams.put("file.url","testdata.txt");
        long jobId = jo.start("extract-cities", jobParams);
```

Listing 22

```
<job id="extractCity" xmlns="http://xmlns.jcp.org/xml/ns/javaee" version="1.0">
        <step id="populateCities" >
                <chunk>
                        <reader ref="entryReader" />
                        <processor ref="entryProcessor"/>
                        <writer ref="entryWriter"/>
                </chunk>
        </step>
</job>
```

Listing 23

```
@Named("entryReader")
public class EntryReader extends AbstractItemReader {
//…
@Override
        public String readItem() {
//…
}
@Named("entryProcessor")
public class EntryProcessor implements ItemProcessor {
//…
@Override
        public Object processItem(Object o) throws Exception {
//…
}
@Named("entryWriter")
public class EntryWriter extends AbstractItemWriter {
//…
@Override
        public void writeItems(List<Object> list) throws Exception {
//…
}
```

Listing 24

Unter dem übergebenen Namen findet sich unter „META-INF/batch-jobs/" ein gleichnamiges XML-Dokument, das den Job beschreibt.

Dabei charakterisieren „job", „chunk" und „step" den Element-orientierten Bearbeitungs-Typ. Mithilfe eines sogenannten „Batchlet" kann auch der Aufgaben-orientierte Bearbeitungs-Typ umgesetzt werden. Die drei hauptsächlichen Klassen werden aus dem XML mit ihrem Bean-Namen angesprochen. Demzufolge ergibt sich, dass für Reader, Processor und Writer jeweils eine entsprechende Klasse umzusetzen ist (siehe Listing 24).

Noch vieles mehr

Vor allem in den zahlreichen Maintenance-Releases (MR) verbergen sich eine Menge Vereinfachungen und Verallgemeinerungen. Das Transaktionshandling der JTA wurde angepasst und von den EJBs entkoppelt. Damit ist die Verwendung von Transaktionen auf ManagedBeans in der Plattform vereinheitlicht worden. Im EJB-Bereich sind unter anderem lokale asynchrone Session-Beans und nicht in der Datenbank persistierte Timer-Beans in das EJB-Lite-Profil aufgenommen worden. Der Embeddable-EJB-Container implementiert jetzt auch das AutoClosable-Interface von Java SE 7.

Ähnlich viele Kleinigkeiten sind es bei CDI geworden: Fehlerbehebungen, Klarstellungen in der Spezifikation und knapp drei Handvoll neuer Funktionen wie die Möglichkeit zur Injektion des Servlet-Context oder die Bereitstellung sogenannter „applikationsübergreifender Lebenszyklus-Events" für Komponenten (wie „@Initialized" oder„@Destroyed"). Die Namespaces fast aller XML-Deployment-Deskriptoren wurden verändert und enthalten jetzt keine Referenz mehr auf „java.sun.com". Der neue Namespace beginnt mit „http://xmlns.jcp.org/xml/ns/".

GlassFish 4.0 ist die Referenz-Implementierung

Mit Verabschiedung der Spezifikation ist auch die Referenz-Implementierung des GlassFish in die letzte Runde gegangen. Seit dem Vorabend des offiziellen Launch-Events steht die finale Version 4.0 zum Download bereit.

Mit einer zweistelligen Zahl zertifizierter Java-EE-6-Server wurde in den vergangenen drei Jahren am Markt eine breite Nutzer-Basis geschaffen. Vermutlich ist dies auch der Zeitraum, bis Kunden und Hersteller auf die ab jetzt aktuelle Version umstellen. Das verschafft Entwicklern genügend Zeit, sich mit den neuen Technologien auseinanderzusetzen.

Aktuell mangelt es noch an Code-Beispielen. Verbindliche Informationen finden Interessierte meist nur in den Spezifika-

Spezifikation	JSR	Version	Link
Java Platform, Enterprise Edition	342	7	http://java.net/projects/javaee-spec/
Managed Beans	342	1.0	
Java EE Web Profile ("Web Profile")	342	1.0	
Java API for RESTful Web Services (JAX-RS)	339	2.0	http://jax-rs-spec.java.net/
Web Services for Java EE	109	1.4	
Java API for XML-Based Web Services (JAX-WS)	224	2.2	https://jax-ws.java.net/
Java Architecture for XML Binding (JAXB)	222	2.2	https://jaxb.java.net/
Web Services Metadata for the Java Platform	181	2.1	
Java API for XML-Based RPC (JAX-RPC) (Optional)	101	1.1	http://jax-rpc.java.net/
Java API for XML Registries (JAXR) (Optional)	93	1.0	
Servlet	340	3.1	
JavaServer Faces (JSF)	344	2.2	http://javaserverfaces.java.net/
JavaServer Pages (JSP)	245	2.3	
JavaServer Pages Expression Language (EL)	341	3.0	http://el-spec.java.net/
A Standard Tag Library for JavaServer Pages (JSTL)	52	1.2	https://jstl.java.net/
Debugging Support for Other Languages	45	1.0	
Contexts and Dependency Injection for the Java EE Platform (CDI)	346	1.1	https://github.com/jboss/cdi/wiki
Dependency Injection for Java (DI)	330	1.0	
Bean Validation	349	1.1	http://beanvalidation.org/
Enterprise JavaBeans (EJB)	345	3.2	http://ejb-spec.java.net/
Java EE Connector Architecture (JCA)	322	1.7	
Java Persistence (JPA)	338	2.1	http://jpa-spec.java.net/
Common Annotations for the Java Platform	250	1.2	
Java Message Service API (JMS)	343	2.0	https://jms-spec.java.net/
Java Transaction API (JTA)	907	1.2	http://jta-spec.java.net/
JavaMail	919	1.5	http://javamail.java.net/
Java Authentication Service Provider Interface for Containers (JASPIC)	196	1.1	http://jaspic-spec.java.net/
Java Authorization Contract for Containers (JACC)	115	1.5	http://jacc-spec.java.net/
Java EE Application Deployment (Optional)	88	1.2	
Java Database Connectivity (JDBC)	221	4.0	
Java Management Extensions (JMX)	255	2.0	http://openjdk.java.net/groups/jmx/
JavaBeans Activation Framework (JAF)	925	1.1	
Streaming API for XML (StAX)	173	1.0	http://sjsxp.java.net/
Java Authentication and Authorization Service (JAAS)		1.0	
Interceptors	318	1.2	https://interceptors-spec.java.net/
Batch Applications for the Java Platform	352	1.0	http://jbatch.java.net/
Java API for JSON Processing	353	1.0	http://json-processing-spec.java.net/
Java API for WebSocket	356	1.0	http://websocket-spec.java.net/
Concurrency Utilities for Java EE	236	1.0	http://concurrency-ee-spec.java.net/

Tabelle 1: Spezifikationen in Java EE 7

tionen selber oder in den zugehörigen JavaDocs der Referenz-Implementierung. Eine neue Version des Java-EE-Tutorial ebenfalls bereits. Die Java EE Expert Group geht wohl auch bald in die nächste Runde. Mit Java EE 8 soll dann doch das Thema „Cloud" kommen.

Links und Literatur

- JSR 342, Java Platform, Enterprise Edition 7 (Java EE 7) Specification: http://jcp.org/en/jsr/summary?id=342
- Neuausrichtung der Java EE 7 im August 2012: https://blogs.oracle.com/theaquarium/entry/java_ee_7_roadmap
- Transparency and Community Participation in Java EE 7: https://blogs.oracle.com/arungupta/entry/transparency_and_community_participation_in
- EclipseLink MOXy als JAX-RS JSON Provider: http://blog.bdoughan.com/2012/03/moxy-as-your-jax-rs-json-provider.html
- Neue Namespaces: http://www.oracle.com/webfolder/technetwork/jsc/xml/ns/javaee/index.html" \l „1"
- Referenz-Implementierung GlassFish 4.0: https://glassfish.dev.java.net
- Java-EE-7-Tutorial: http://docs.oracle.com/javaee/7/tutorial/doc/home.htm

Markus Eisele
markus.eisele@msg-systems.com

Markus Eisele arbeitet bei der msg systems ag in München. Er ist Java EE 7 EG und GlassFish Community Member und betreibt einen englischsprachigen Blog über Java EE, GlassFish und WebLogic unter http:// blog.eisele.net

Source Talk Tage 2013

Stefan Koospal, Sun User Group Deutschland e.V.

Auch in diesem Jahr laden die Java User Group Deutschland e.V. und die Sun User Group Deutschland e.V. Anfang Oktober wieder zu den „Source Talk Tagen" nach Göttingen. An zwei Tagen werden Vorträge, Trainings und Workshops zu den Themen „Java", „Web-Technologien", „Systemverwaltung", „Cloud Computing" und „eLearning" angeboten. Besonderen Wert legen die Veranstalter darauf, dass neben den Tracks auch die hochwertigen Trainings für Vollzeit-Studierende kostenfrei sind.

Am 1. und 2. Oktober 2013 steht das Mathematische Institut in Göttingen wieder im Zeichen der Source Talk Tage. Das Highlight für Java-Entwickler in diesem Jahr ist die Keynote von Java Champion und NetBeans-Dream-Team-Member Adam Bien. Neben den etablierten Tracks und Trainings werden in diesem Jahr erstmals Workshops angeboten, in denen die Teilnehmer im kleinen Kreis nach einem kurzen Einführungsvortrag in etwa 90 Minuten intensiv diskutieren und von eigenen Erfahrungen berichten können.

Das gesamte Programm umfasst vier Tracks „Java", „Entwicklungswerkzeuge", „Systeme & Netze" sowie „Mathematik & eLearning". Parallel dazu finden in den zwei Tagen sechs Trainings statt: „Scala", „Continuous Delivery", „Arduino", „Clean Code", „Phonegap" und „Platform as a Service". Fünf neue Workshops zu „Android", „JVM-Performance", „Java & Sicherheit", „Android Security" und „Sicherheit in mobilen Netzen" komplettieren das Programm.

Für die Mitglieder der Java User Group Deutschland e.V., der Sun User Group Deutschland e.V. sowie der DOAG und anderer im iJUG e.V. organisierter Java User Groups ist die Teilnahme an den Tracks und Workshops kostenfrei. Vollzeitstudenten haben die Möglichkeit nach vorheriger Anmeldung an allen Veranstaltungen kostenfrei teilzunehmen, sofern noch Plätze verfügbar sind. Weitere Informationen unter www.sourcetalk.de.

Stefan Koospal
office@java.de

WebLogic-Grundlagen: Die feinen Unterschiede

Sylvie Lübeck, ORACLE Deutschland B.V. & Co. KG

In dieser und in den folgenden Ausgaben werden verschiedene Themen auf Basis der neuesten WebLogic-Server-Version 12.1.2 beleuchtet. Dieser Artikel startet mit den Grundlagen wie allgemeine Architektur, Installation, Konfiguration und Administration.

Der Oracle WebLogic Server ist ein skalierbarer Java Application Server, der für den unternehmensweiten Einsatz optimiert ist. Seit Verfügbarkeit der Version 12*c* wird die Java-EE-6.0-Spezifikation voll unterstützt („Full Profile"). Dadurch ist sichergestellt, dass die erforderlichen Standard-APIs zur Entwicklung verteilter Anwendungen zur Verfügung stehen, mit denen Datenbanken und Nachrichtendienste eingebunden und die Interoperabilität mit externen Systemen, Endbenutzer-Clients, Java oder Webbrowser adressiert werden können. Ergänzend dazu spielt auch die In-Memory-Grid-Lösung „Oracle Coherence" eine immer wichtigere Rolle. Diese ermöglicht es, Zustand und Anwendungsdaten möglichst ausfallsicher und besonders skalierbar bereitzustellen.

Architektur

Bevor wir auf die Installation und Konfiguration eingehen, ist ein gemeinsames Verständnis der Grundbegriffe im WebLogic-Server-Umfeld wichtig (siehe Abbildung 1). Eine „Domain" bildet eine logische Administrationseinheit und beinhaltet WebLogic-Server-Instanzen und deren jeweilige Ressourcen. Die einzelnen WebLogic-Server-Instanzen werden als „Managed Server" bezeichnet. Auf genau einem Server je Domain wird die „Administration Console"-Anwendung bereitgestellt. Über diesen Server – „Admin Server" genannt – kann die Domain konfiguriert und administriert werden.

Die kleinste Domain besteht aus einem Admin Server, der zugleich auch als Managed Server für die eigenen Anwendungen verwendet wird. Dies ist häufig in Entwicklungsumgebungen der Fall. In der Regel werden die Anwendungen auf dedizierten Managed Servern bereitgestellt. Diese können auf mehrere physische oder virtuelle Rechner verteilt werden. Soll eine Anwendung besonders skalierbar oder ausfallsicher sein, so können mehrere Managed Server mit identischen Anwendungen, Ressourcen und Diensten als „Cluster" verbunden werden.

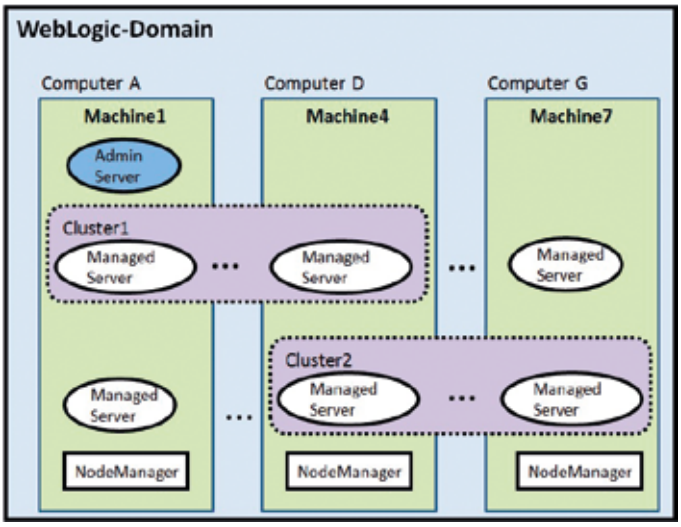


Abbildung 1: WebLogic Server – Grundbegriffe der Architektur

Die Zuordnung der Managed Server zu den jeweiligen Rechnern erfolgt über die Machine-Konfiguration. Diese Information wird zum einen durch den „Node Manager", um bei Bedarf automatisch einen ausgefallenen Managed Server neu zu starten, und zum anderen im Cluster verwendet, um den besten Ort zur Speicherung der replizierten Session-Daten zu ermitteln. Auch Coherence-Knoten können mit dem Node Manager überwacht werden.

Viele Kunden verwenden einen Cluster oder eine Domain pro Anwendung, um so Versionsabhängigkeiten zwischen Laufzeitumgebung und Anwendungen möglichst gering zu halten. Die Architektur des WebLogic Servers ist sehr flexibel. Es lassen sich dadurch auch mehrere Anwendungen auf einem Server, einer Domain oder einem Cluster betreiben.

Der WebLogic Server verarbeitet die Anfragen über einen „Thread Pool". Mithilfe des Work Manager werden die Anfragen optimal zur Abarbeitung organisiert. Die Priorisierung erfolgt auf Basis der definierten Regeln auf Domain- oder Anwendungs-Ebene sowie der Laufzeit-Metriken inklusive der aktuell benötigten Zeit zur Abarbeitung einer Anfrage. Der Work Manager spielt auch eine wichtige Rolle bei der Erkennung und Behandlung überlasteter WebLogic-Server-Knoten.

In WebLogic 12.1.2 ist das Konzept „Dynamic Server" und „Dynamic Cluster" neu eingeführt. Die Konfiguration eines dynamischen Servers wird in einem „Server Template" hinterlegt. Ein Dynamic Cluster besteht aus einem oder mehreren Dynamic Servern, die alle auf der gleichen Konfiguration basieren. Werden zusätzliche Ressourcen benötigt, können auf Basis des Server Templates ohne manuelle Konfigurationsschritte zusätzliche Dynamic Server erzeugt und gestartet werden. Im Rahmen

WebLogic Server Distribution	Inhalt	Einsatz
wls_121200.jar	WebLogic Server, Coherence, Beispiele	Produktion, Entwicklung, Test
fmw_infra_121200.jar	Java Required Files (inkl. Application Development Framework) Enterprise Manager	Produktion, Entwicklung, Test
wls_1212_dev.zip	Ohne Coherence und pre-built client jars, mit Maven-Plug-in und Sync-Plug-in	Entwicklung, Evaluierung
wls1212_dev_supplemental.zip	Beispiele, Derby-Datenbank und WebLogic Console Help („non-english") für „.zip"-Installation	Entwicklung, Evaluierung

Tabelle 1: WebLogic-Server-Distributionen

der Dynamic-Cluster-Konfiguration wird die maximale Größe des Clusters festgelegt, die er bei einer angenommenen Spitzenlast erreichen kann. Die Definition von „Server Name", „Machine Name" und „Listen Ports" wird anhand vordefinierter Regeln und zuvor im Template hinterlegter Parameter für neue Dynamic Server im Dynamic Cluster automatisch generiert. Jetzt kann der Administrator bei Bedarf einen Dynamic Server mit dem abgeleiteten Namen starten.

Das Deployment der Anwendungen verhält sich wie generell bei einem Cluster. Wurde eine Anwendung oder Ressource auf dem gesamten Cluster bereitgestellt, so steht sie auch auf den neu gestarteten Dynamic Servern zur Verfügung. Im Dynamic Cluster gibt es keine „Whole Server Migration" und „Service Migration". Anwendungen und Ressourcen müssen auf dem gesamten Cluster bereitgestellt werden und damit auch auf allen Dynamic Servern. Ein Dynamic Cluster gibt damit die Möglichkeit, schnell und einfach die Anzahl der WebLogic-Server-Instanzen zu erhöhen beziehungsweise wieder zu verringern.

Installation

Die WebLogic-Server-Installation kann wahlweise zentral oder dezentral organisiert werden. Die zentrale Installation könnte beispielsweise auf einem „Shared Storage" liegen, wobei die Konfiguration auf den jeweiligen Servern hinterlegt ist, auf denen WebLogic-Server-Instanzen (JVMs) laufen sollen (siehe Abbildung 2).

Bei der Installation des WebLogic Servers wurden schon immer Software (Binaries) und Konfiguration voneinander getrennt. Dies ist insbesondere dann ein großer Vorteil, wenn es darum geht, eine hochverfügbare Umgebung aufzusetzen. Die Software wird als „.jar"-Datei ausgeliefert. Die verschiedenen Varianten (siehe Tabelle 1) beziehen sich auf die verfügbaren Software-Downloads. Die Lizenzbedingungen sind gesondert zu betrachten und der „Licensing Information" der jeweiligen WebLogic-Server-Dokumentation zu entnehmen [1].

Für Evaluierungszwecke steht die Software auf dem Oracle Technologie Network [2] zur Verfügung. Die Code-Beispiele in diesem Artikel wurden unter dem Betriebssystem Oracle Enterprise Linux und auf Basis der WebLogic-Server-Distribution „wls_121200.jar" durchgeführt. Vor Installa-

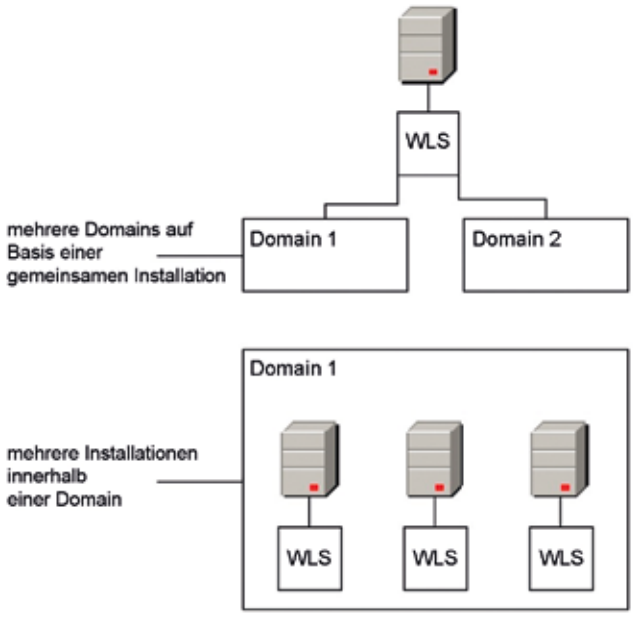


Abbildung 2: WebLogic-Server-Installation und Domain, Quelle: Figure 4-1 „Understanding WebLogic Server (12.1.2)" [1]

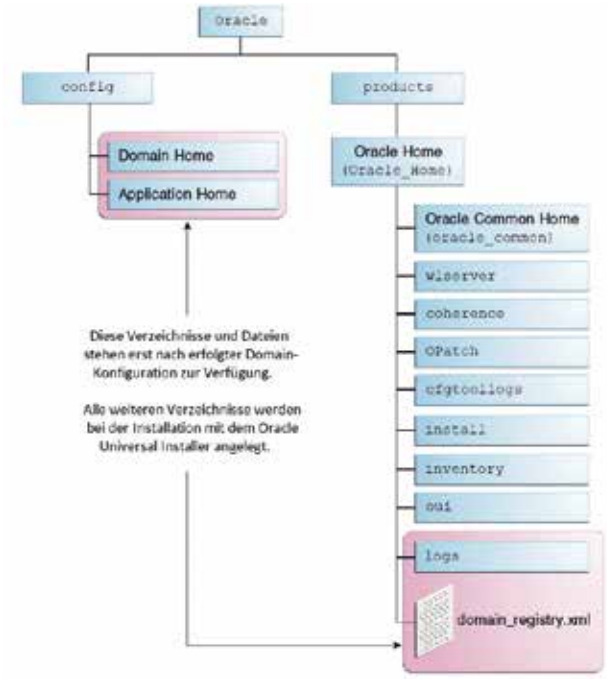


Abbildung 3: WebLogic-Server-Verzeichnisstruktur, Quelle: Anhang B-1 „Installing and Configuring Oracle WebLogic Server and Coherence 12c (12.1.2)" [1]

tion des WebLogic Servers muss zunächst das passende JDK installiert sein. Für die Version 12.1.2 ist Oracle JDK 1.7 erforderlich. Weitere Informationen zur Zertifizierung sind der Fusion-Middleware-Zertifizierungsmatrix [3] zu entnehmen.

Als Nächstes wird die Umgebungsvariable „JAVA_HOME" gesetzt und mit in den Pfad der „PATH"-Variablen aufgenommen. Die anschließende Installation erfolgt ab der Version 12.1.2 mit dem Oracle Universal Installer. Hiervon ausgenommen ist die Installation der „.zip"-Distribution, die nach Setzen der notwendigen Umgebungsvariablen einfach entpackt wird. Die Installation über den Oracle Universal Installer (OUI) wird mit „java –jar wls_121200.jar" gestartet.

Im Zusammenspiel mit anderen Oracle-Produkten und für produktive Umgebungen ist die Installation über den OUI unbedingt erforderlich. Bei Bedarf kann auch nur die mit dem OUI integrierte Distribution des WebLogic Servers gepatcht werden.

Es gibt drei verschiedene Installationstypen: „WebLogic Server", „Coherence Server" oder „Complete". Bei „WebLogic Server" werden sowohl die Komponenten für den WebLogic Server als auch für Coherence installiert. Diese Variante ist für den Aufbau einer Produktionsumgebung empfohlen. „Coherence" erzeugt keine WebLogic Client jars. Die WebLogic-Server- und Coherence-Beispiele sind nur in der Variante „Complete" enthalten. Abbildung 3 zeigt die Verzeichnisstruktur nach Installation der Software „wls_121200.jar" mit dem Oracle Universal Installer.

Das Unterverzeichnis „wlserver" beinhaltet die Produktdateien des WebLogic Servers und ist auch mit seinem Pfad als „WL_HOME" referenziert. Analog dazu sind unter „coherence" die Coherence-Produktdateien installiert. Im Verzeichnis „oracle_common" stehen die für den WebLogic Server übergreifenden Binaries und Bibliotheken. Im Falle einer Installation der Fusion-Middleware-Infrastruktur auf Basis von „fmw_infra_121200.jar" sind dort auch die „java required files" (jrf) zu finden.

Bei der Installation mit dem Oracle Universal Installer wird im Benutzerverzeichnis das „oraInventory" angelegt beziehungsweise aktualisiert. Daher ist es auch wichtig, bei einer Deinstallation das entsprechende „./deinstall.sh"-Skript zu verwenden. Alternativ gibt es auch die Möglichkeit, die Installation im Hintergrund „silent" auszuführen.

Wurde der Installationstyp „Complete" ausgewählt, können im Anschluss zur Installation die drei Beispiel-Domains über den „Quick Configuration Wizard" erzeugt werden. Die Einstellungen für den Administrator, den Pfad für Domain und Anwendungen, die Listen-Adresse sowie die Ports für die Beispiel-Domains sind im „Quick Configuration Wizard" definiert.

Konfiguration

Zur Erstellung einer Domain gibt es mit „Domain Configuration Wizard" und „WebLogic Scripting Tool" (WLST) zwei verschiedene Wege. „Domain Configuration Wizard", ein grafisches Werkzeug, führt Schritt für Schritt durch die Domain-Konfiguration. Hierbei können je nach Auswahl die Einstellungen für Admin Server, Node Manager, Managed Server, Cluster und Coherence konfiguriert beziehungsweise angepasst werden. Mit Oracle WebLogic Server 12.1.2 wurden der Node Manager und die Managed Coherence Server mit in den Wizard integriert. „cd $ORACLE_HOME/oracle_common/common/bin" und „./config.sh" starten den Domain Configuration Wizard.

Der bisherige Pfad für das „/common/bin"-Verzeichnis von WebLogic Server unter „$WL_HOME" existiert weiterhin, wobei die dortigen Skripte auf diejenigen unter „$ORACLE_HOME/oracle_common/common/bin" referenzieren. Als erste Domain führen wir beispielhaft folgende Konfiguration durch (siehe Tabelle 2).

Auf der linken Seite ist im „Domain Configuration Wizard" der Ablauf der einzelnen Schritte dargestellt. Im ersten Schritt wird definiert, wo die Domain-Konfiguration hinterlegt werden soll. Für Produktionsumgebungen empfehlen wir dafür einen von der Installation getrennten Pfad.

Die Domain Location „/home/oracle/wls/user_projects/domains/test1" ist für die Domain „test1" das sogenannte „DOMAIN_HOME". Für unsere Domain benötigen wir im zweiten Schritt lediglich das Basis Template namens „Basic WebLogic Server Domain -12.1.2.0 [wlserver*]". Alternativ könnte man hier ein zusätzliches Oracle-spezifisches oder ein eigenes Domain-Template auswählen, das als Basis für die Konfiguration der neuen Domain verwendet werden soll.

Nach Definition des Administrations-Benutzers und -Passworts im dritten Schritt werden in Schritt vier der Betriebsmodus „development" oder „production" und das zu verwendende Java Development Kit (jdk) festgelegt. In Schritt fünf folgt die Wahl der Domain-Komponenten, die angepasst beziehungsweise neu konfiguriert werden sollen.

Mit WebLogic 12.1.2 wurde die Konfiguration von Coherence-Knoten als „storage enabled"-Managed-Coherence-Server in den Domain Configuration Wizard integriert. Die restlichen Schritte wurden entsprechend Tabelle 2 durchgeführt und abschließend das Erzeugen der Domain-Konfiguration „test1" angestoßen. Die Node-Manager-Konfiguration ist Defaultmäßig unter der Domain-Konfiguration „$DOMAIN_HOME/nodemanager" hinterlegt.

Die Alternative zur Definition einer Domain ist die Konfiguration per „WebLogic Scripting Tool" (WLST). Dahinter steht eine Jython-basierte Skript-Umgebung. WLST kann interaktiv als Kommandozeilen-Werkzeug oder als Ablauf-Umgebung für vorgefertigte Skripte verwendet werden, die ohne Administrator-Eingaben im Hintergrund laufen. Oft wird WLST extensiv zur Automatisierung von Administrationsaufgaben im WebLogic-Server-Umfeld eingesetzt. Der Start erfolgt über „cd $ORACLE_HOME/oracle_common/common/bin" und „./wlst.sh".

Nach Starten des WLST befindet man sich im „offline"-Modus, was dem „Domain Configuration Wizard" entspricht. Für die Ausführung bestimmter Aktionen, die Bezug auf die Domain haben, sollte die Umgebung richtig gesetzt werden, etwa durch „cd $DOMAIN_HOME/bin" und „. ./setDomainEnv.sh".

Beispielskripte dafür, wie mit WLST eine Domain erstellt wird, sind unter „$WL_HOME/common/templates/scripts/wlst" zu finden.

Erstreckt sich eine Domain über mehrere Knoten (siehe Abbildung 1), so kann die Domain-Konfiguration mit den Befehlen „pack" und „unpack" auf die verschiedenen Knoten kopiert werden. Der Aufruf erfolgt aus dem Verzeichnis „$ORACLE_HOME\oracle_common\common\bin"

Konfigurationsschritt	Beispiel Domain test1
Administration Server	Server Name: AdminServer Listen Adress: All local Addresses Listen Port: 7001
Node Manager	Per Domain Node Manager Credentials: <nm_admin_user> <nm_admin_password>
Managed Server	Server Name: ms1 – port: 7003 Server Name: ms2 – port: 7004 Listen Address: All local Addresses
Machines (Unix Machine)	Machine Name: Machine1 Listen Address: All local Addresses Listen Port: 5556
Assign Servers to Machines	Machine1: AdminServer, ms1, ms2

Tabelle 2: Beispiel Domain „test1"

mit „./pack.sh" beziehungsweise „./unpack.sh".

Basierend auf einer gesicherten Domain-Konfiguration in Form eines Templates können neue Domains erstellt oder bestehende Domains erweitert werden. Sowohl Domain als auch Extention Templates sind über den Domain Template Builder, WLST oder den „pack"-Befehl möglich.

Administration

Das Thema „Administration" ist vielfältig. Der Artikel geht zum einen auf die verschiedenen Werkzeuge ein, die zur Administration verwendet werden können, und zum anderen auf das Starten der Server mit dem Node Manager und WLST. Für die Administration einer Domain steht die „Administration Console" zur Verfügung. Dies ist ein Web-Front-end mit Blick auf die aktuelle Domain-Konfiguration, das verfügbar ist, sobald der Admin Server gestartet ist.

Mit WLST kann man sich ebenfalls an einen Admin Server verbinden. In diesem Fall spricht man vom „WLST-‚Online'-Modus", dem die Admin Console entspricht. Wenn in der Admin Console die Aufnahmefunktion aktiviert ist, ist hinterher genau nachvollziehbar, welche WLST-Befehle ausgeführt worden sind. Beide Werkzeuge lesen die WebLogic Server MBeans, die die Domain-Konfiguration beinhalten, aus beziehungsweise aktualisieren diese. Ein Abbild der Konfiguration wird zusätzlich in der Datei „config.xml" gehalten. Änderungen an der Konfiguration sollten immer über ein Interface und nie direkt in „config.xml" erfolgen, um die Konsistenz sicherzustellen. Auch im „Offline"-Modus ist es möglich, sich per WLST an eine Domain zu verbinden, um die Konfiguration zu lesen oder zu verändern. Zu bestimmten Zeitpunkten, wie beim Stoppen eines Servers, wird die Konfiguration in die Datei „config.xml" zurückgeschrieben.

Ist der WebLogic Server zusammen mit der Fusion-Middleware-Infrastruktur installiert, steht auch die Basis-Komponente des Enterprise Manager „Fusion Middleware Control" zur Verfügung. Dabei handelt es sich ebenfalls um eine Web-basierte Administrations-Konsole, die im Gegensatz zur WebLogic Server Admin Console auch die Überwachung und das Management von Oracle-http-Server, SOA Suite, WebCenter und Identity Management umfasst.

Eine weitere Aufgabe der Administration ist das Starten und Stoppen der Server in der Umgebung. Hier gibt es mehrere Möglichkeiten. Wie der Admin Server und die Managed Server per WLST über den Node Manager gestartet werden, kann aus dem Kasten auf dieser Seite entnommen werden. Dies könnte auch Inhalt einer Skript-basierten Lösung sein (siehe Kasten).

Alle Server, die über den Node Manager gestartet wurden, sind auch von diesem überwacht und werden im Falle eines ungeplanten Ausfalls automatisch wieder gestartet. Ein geplantes Herunterfahren muss dann auch immer per WLST via Node Manager mit "nmKill(‚ServerName')" oder über die Admin Console erfolgen.

In der Regel ist ein WebLogic Server für alle unterstützten Protokolle über einen Port erreichbar. Für administrative Aufgaben gibt es die Möglichkeit, einen eigenen Port zu konfigurieren, für den implizit ein eigener „Administration Channel" mit „Protokoll:Listenadresse:Port" generiert wird. Dies ermöglicht das Starten von Servern im Standby-Modus und die Durchführung administrativer Aufgaben, ohne dass die Anwendungen verfügbar sind. Dadurch wird unter anderem der Netzwerk-Verkehr

Den WebLogic Server starten

```
cd $DOMAIN_HOME/bin
. ./setDomainEnv.sh

WebLogic Scripting Tool starten
java weblogic.WLST

Node Manager starten
startNodeManager(verbose='true',Node
ManagerHome='/home/oracle/wls1212/
user_projects/domains/test1/nodeman-
ager',
ListenPort='5556')

Mit Node-Manager verbinden
nmConnect('<nm_admin_user>','<nm_
admin_password>','localhost','5556','te
st1','/home/oracle/wls1212/user_proj-
ects/
domains/test1','ssl','true')

Admin Server über Node Manager das
erste Mal starten
prps=makePropertiesObject(,Username
=weblogic;Password=welcome1')
nmStart(,AdminServer',props=prps)

Admin Server ab dem zweiten Mal starten
nmStart(,AdminServer')

WLST mit Admin Server verbinden
connect(,weblogic',‘welcome1',‘localho
st:7001')

Benutzername und Passwort verschlüsselt
mitgeben, siehe WLST-Befehl „storeUser-
Config()"

prps =
makePropertiesObject(„AdminURL
=http://wlsdev.de.oracle.com:7001
;Username=weblogic;Password=welco
me1")

Managed Server das erste Mal starten
nmStart('ms1',props=prps)
nmStart('ms2',props=prps)

Managed Server ab dem zweiten Mal starten
nmStart('ms1')
nmStart('ms2')
```

auf dem produktiven Port minimiert. Zusätzlich können „Network Channel", bestehend aus „Protokoll:Listenadresse:Port", für verschiedene Protokolle jeweils mit dediziertem Port definiert werden, die physisch bestimmten Netzwerkkarten zugeordnet sind.

Die Admin Console ermöglicht den Blick auf eine Domain. Im Gegensatz dazu bietet Oracle Enterprise Manager Cloud Control 12*c* mit dem WebLogic-Management-Pack die Möglichkeit, mehrere Domänen zu überwachen und zu administrieren. Ein Kernbereich des Enterprise Manager ist die Möglichkeit der Diagnose, um möglichst frühzeitig Probleme zu erkennen und beheben zu können. Dazu gehören sowohl der tiefe Blick in die JVM bis hin auf die Datenbank als auch die Nachverfolgbarkeit von Transaktionen über mehrere Schichten hinweg. In vielen Bereichen lassen sich die Informationen in Echtzeit und aus der Historie betrachten. Der Automatisierungsgrad für wiederkehrende Aufgaben kann erheblich erhöht werden.

Insgesamt sichert der Einsatz des WebLogic-Management-Packs eine höhere Service-Qualität und ein Minimum an geplanten und ungeplanten Ausfallzeiten. Der Betrieb wird im gesamten Lebenszyklus einer Applikation unterstützt und der Administrator behält zu jeder Zeit den Überblick über die gesamten WebLogic-Server-Umgebungen, auch wenn diese komplexer sind.

Hinweis

In der nächsten Ausgabe beschäftigen wir uns mit den Themen „Hochverfügbarkeit" und „Skalierbarkeit".

Weiterführende Links

[1] WebLogic Server Dokumentation: http://www.oracle.com/technetwork/middleware/weblogic/documentation/index.html

[2] WebLogic Server Software Download: http://www.oracle.com/technetwork/middleware/weblogic/downloads/index.html

[3] Fusion Middleware Zertifizierungsmatrix: http://www.oracle.com/technetwork/middleware/ias/downloads/fusion-certification-100350.html

Sylvie Lübeck
Sylvie.Luebeck@oracle.com

Sylvie Lübeck arbeitet bei der ORACLE Deutschland B.V. & Co. KG in der Abteilung „Business Unit Server Technologies – Fusion Middleware", die deutschlandweit die Middleware-Themen technisch und vertriebsunterstützend verantwortet. Ihr Schwerpunkt ist der Oracle WebLogic Server.

Morphia, Spring Data & Co. – Java-Persistenz-Frameworks für MongoDB

Tobias Trelle, codecentric AG

Nach einer kurzen Einführung in MongoDB betrachten wir zunächst, wie Anwendungen mittels des Java-Treibers mit dieser NoSQL-Datenbank interagieren können. Anschließend werden eine Auswahl an Persistenz-Frameworks wie Spring Data, Hibernate OGM oder Morphia vorgestellt, die beispielsweise ein komfortables Objekt-Mapping und Support für Data Access Objects (DAO) anbieten.

MongoDB [1] ist eine hochperformante, dokumentenorientierte NoSQL-Datenbank, die als Open Source [2] verfügbar ist. Der Name leitet sich vom englischen „humongous" ab, was so viel wie gigantisch oder riesig groß bedeutet. Ein Dokument ist eine geordnete Menge von Key/Value-Paaren, wobei unter einem Schlüssel auch Arrays oder wiederum Dokumente abgelegt werden können. Objekt-Netze lassen sich so sehr einfach auf ein Dokument abbilden, der sogenannte „Impedance Mismatch" relationaler Datenbank-Systeme entfällt.

Intern speichert MongoDB die Dokumente im BSON-Format [3] ab, das sich strukturell an JSON anlehnt, aber über differenziertere Datentypen verfügt und effizienter traversierbar ist. Da BSON nicht menschenlesbar ist, werden Beispiele in der Regel in JSON angegeben, zum Beispiel „{ hallo: "MongoDB", sub: {x:1}, a: [1,2,3,5,8] }". Zentrale Features von MongoDB sind:

- *Schema-Freiheit*
  Dokumente werden innerhalb von Datenbanken in logischen Namensräumen, den „Collections", zusammengefasst. Dabei existiert kein Schema, das diesen Dokumenten Restriktionen auferlegt, außer dem Vorhandensein eines Primärschlüssel-Felds. Somit lassen sich auch gut semi-strukturierte Daten abspeichern.

- *Replikation*
  Replikation der Daten ist integraler Bestandteil eines MongoDB-Setups. Da bei MongoDB bewusst auf das ACID-Paradigma relationaler Datenbanken verzichtet wurde, sollten produktive System stets in einem „Replica Set" betrieben werden. Diese bieten sich auch zur Skalierung von Lesezugriffen an.

- *Sharding*
  Die horizontale Skalierung („scale-out") wird bei MongoDB über das „Sharding" erreicht. Damit wird der gesamte Datenbestand disjunkt auf (viele) verschiedene Shard-Knoten verteilt, sodass zum einen Volumina im Petabyte-Bereich adressiert werden können, zum anderen eine hinreichend gute Performance bei Schreibzugriffen gegeben ist.

Die Kommunikation mit einem einzelnen MongoDB-Server oder einem Replica Set erfolgt über ein proprietäres, binäres TCP/IP-basiertes Protokoll. Für viele gängige Programmiersprachen [4] gibt es entsprechende Treiber, so auch für Java.

Ein Fallbeispiel

Um die Eigenschaften des jeweiligen API beziehungsweise Frameworks besser vergleichen zu können, werden wir durchgängig ein Beispiel persistieren und Abfragen darauf definieren. Es handelt sich um eine Bestellung mit Unterposition (siehe Abbildung 1). Die verwendete Abfrage, die nach Bestellungen mit Unterpositionen einer bestimmten Menge sucht, sieht in der Mongo-Shell (der Administrationskonsole von MongoDB) wie folgt aus: „db.order.find({"items.quantity": ? })".

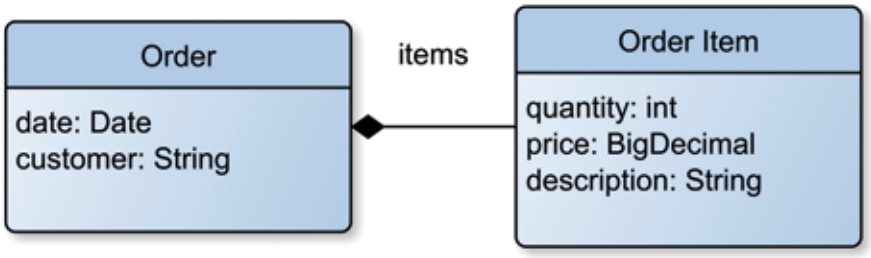


Abbildung 1: Aufbau des Beispiels

Java-Treiber

Der Java-Treiber [5] kann in Form eines einzelnen JAR direkt herunterladen werden oder zum Beispiel in Build-Tools wie Maven als Abhängigkeit definiert werden (siehe Listing 1). Zentraler Einstiegspunkt in das Client-API ist die Klasse „com.mongodb.MongoClient“ (siehe Listing 2). Listing 3 zeigt den Zugriff auf eine bestimmte Collection innerhalb einer Datenbank.

Datenbanken und Collections müssen nicht explizit angelegt sein, das erfolgt automatisch bei der ersten schreibenden Operation auf einer Collection. Über eine „DBCollection“-Instanz werden unter anderem CRUD-Operationen und Queries ausgeführt. Zur Repräsentation von Dokumenten verwendet man die Klasse „com.mongodb.BasicDBObject“, die das „Interface com.mongodb.DBObject“ implementiert und im Wesentlichen eine geordnete Hashmap darstellt (siehe Listing 4).

Sofern nicht vorhanden, erzeugt der Treiber ein Feld namens „_id“, das den Primärschlüssel innerhalb der Collection enthält. Vergibt man den Primärschlüssel in Eigenregie, ist man selbst für die Eindeutigkeit der Werte innerhalb der Collection zuständig. Mit dieser Art der API erzeugen selbst relativ kleine Objektnetze schnell jede Menge Quellcode (siehe Listing 5).

Abfragen werden in MongoDB ebenfalls in Form von Dokumenten formuliert, was das API klein hält, bei nichttrivialen Queries aber auch schnell wieder eine Menge an Quellcode erzeugt. Listing 6 zeigt, wie die Abfrage aus unserem Fallbeispiel formuliert wird. Neben diesen einfachen Operationen stellt der Java-Treiber alle Funktionalitäten zur Verfügung, die durch das Mongo-Protokoll angeboten werden.

Nachfolgend betrachten wir verschiedene Persistenz-Frameworks für MongoDB, die vor allem das Objekt-Mapping, aber auch die Formulierung von Abfragen wesentlich vereinfachen. Die Abbildung von Java-Objekten auf MongoDB-Dokumente wird – in Anlehnung an das O/R-Mapping relationaler Datenbank-Systeme – als „O/D-Mapping“ bezeichnet. Dabei werden in der Regel Java-Klassen auf Collections und einzelne Attribute auf Felder eines Dokuments abgebildet.

```
<dependency>
        <groupId>org.mongodb</groupId>
        <artifactId>mongo-java-driver</artifactId>
        <version>2.11.1</version>
</dependency>
```

Listing 1

```
import com.mongodb.MongoClient;
// Default: localhost:27017
MongoClient mongo = new MongoClient();
// Dedizierter Server
MongoClient mongo = new MongoClient("mongo-prod.firma.de", 4711);
```

Listing 2

```
import com.mongodb.DB;
import com.mongodb.DBCollection;
DB db = mongo.getDB("foo");
DBCollection collection = db.getCollection("bar");
```

Listing 3

```
DBObject doc = new BasicDBObject();
doc.put("date", new Date());
doc.put("i", 42);
// Dokument einfügen
collection.insert(doc);
```

Listing 4

Morphia

Morphia ist der älteste O/D-Mapper, der als Google-Code-Projekt [6] entstanden ist, mittlerweile aber auf GitHub [7] weitergepflegt wird. Das O/D-Mapping wird mithilfe eigener Annotationen realisiert (siehe Listing 7).

Ist für den Klassen- beziehungsweise Attribut-Namen keine Annotation angegeben, werden die jeweiligen Namen als Feld-Namen des MongoDB-Dokuments verwendet. Neben den Annotationen aus dem Beispiel gibt es noch weitere, etwa für das Anlegen von Indizes und einigem mehr. Zur typsicheren Formulierung von Queries bietet Morphia eine Basisklasse für DAOs, die man beerben kann und die bereits alle CRUD-Operationen zur Verfügung stellt (siehe Listing 8).

Abfragen lassen sich mit einer kleinen DSL auf zwei verschiedene Arten formulieren: zum einen als Filter, bei dem Kriterien über den Feldnamen einschließlich Operatoren abgebildet werden, zum anderen in Form eines sogenannten „Fluent-API“, bei dem die Relationen über eigene Methoden realisiert sind. Neben der Formulierung der Abfrage können auch sehr bequem Manipulationen der Ergebnismenge mit Sortierung, Weitersetzen des Cursors etc. angegeben werden. Listing 9 zeigt, wie das Ganze aufgerufen wird. „MongoClient()“ ist dabei die Instanziierung des zuvor besprochenen MongoDB-Treibers,

```
DB db = mongo.getDB("test");
DBCollection collection = db.getCollection("order");
DBObject order;
List<DBObject> items = new ArrayList<DBObject>();
DBObject item;
// order
order = new BasicDBObject();
order.put("date", new Date());
order.put("custInfo" ,"Tobias Trelle");
order.put("items", items);
// items
item = new BasicDBObject();
item.put("quantity", 1);
item.put("price", 47.11);
item.put("desc","Item #1");
items.add(item);
item = new BasicDBObject();
item.put("quantity", 2);
item.put("price", 42.0);
item.put("desc","Item #2");
items.add(item);
collection.insert(order);
```

Listing 5

```
DB db = mongo.getDB("test");
DBCollection collection = db.getCollection("order");
DBObject query;
DBObject order;
DBCursor cursor;
query = new BasicDBObject("items.quantity", 2);
cursor = collection.find(query);
while ( cursor.hasNext() ) {
        order = cursor.next();
        System.out.println.(order);
}
```

Listing 6

```
@Entity("orders")
public class Order {
        @Id private ObjectId id;
        private Date date;
        @Property("custInfo") private String customerInfo;
        @Embedded List<Item> items;
        ...
}

public class Item {
        private int quantity;
        private double price;
        @Property("desc") private String description;
        ...
}
```

Listing 7

```
public class OrderDao extends BasicDAO<Order, ObjectId> {
        List<Order> findByItemsQuantity(int quantity) {
                return
                find( createQuery().filter("items.quantity", quantity))
                .asList();
        }

        List<Order> findByItemsPriceGreaterThan(double price) {
                return
                find( createQuery().field("items.price").greaterThan(price) )
                .asList();
        }
        …
}
```

Listing 8

Morphia baut – wie die anderen O/D-Mapper auch – darauf auf.

Spring Data

Spring Data [8, 9] ist ein Teilprojekt des Spring-Frameworks und bietet ein gemeinsames Programmiermodell [10] für diverse NoSQL-Datenbanken, Map/Reduce-Frameworks und auch relationale Datenbanken an. Spring Data bettet sich nahtlos in das restliche Spring-Ökosystem ein und bietet folgende Features an:

- *Templating*
  Zur Kapselung von Ressourcen und Übersetzung von Exceptions
- *O/X-Mapping*
  Zum Mapping von Java-Klassen auf die entsprechenden Datentöpfe
- *Repository-Unterstützung*
  Zur vereinfachten Definition von Abfragen

Nachfolgend konzentrieren wir uns auf „Spring Data MongoDB". Für den Zugriff auf eine MongoDB-Datenbank definieren wir zunächst ein entsprechendes Template.

Listing 10 zeigt eine Minimalausprägung davon und Listing 11 das Objekt/Dokument-Mapping unseres Beispiels. Beim Schreiben eigener Abfragen reicht es nun aus, die Query als Methode an einem „Interface" zu definieren, Spring Data injiziert dann zur Laufzeit ein Proxy-Objekt mit der entsprechenden Implementierung (siehe Listing 12).

```
private OrderDao dao;
dao = new OrderDao(new Morphia().createDatastore(
new MongoClient(),"<datenbankname>")
);
dao.deleteAll();
// Dokument speichern
Order order = new Order("Tobias Trelle, gold customer");
List<Item> items = new ArrayList<Item>();
items.add( new Item(1, 47.11,"Item #1") );
items.add( new Item(2, 42.0,"Item #2") );
order.setItems(items);
dao.save(order);
// Dokument suchen
List<Order> orders = dao.findByItemsQuantity(2);
```

Listing 9

Über bestimmte Namenskonventionen und Teile der Methodensignatur leitet Spring die passende Query ab. So wird aus der Methode „findByItemQuantity(int)“ durch Abschneiden des Prefix „findBy“ und Interpretation des restlichen Teils als Suchpfad genau die gewünschte Abfrage. Durch den Einsatz bestimmter Schlüsselwörter wie „And“ oder „GreaterThan“ können auch komplexere Abfragen abgeleitet werden.

Alternativ kann die Query aber auch immer in JSON-Notation mit Platzhaltern durch die „@Query“-Annotation formuliert werden, was für Abfragen mit vielen Feldern und/oder Operatoren sicher zu bevorzugen ist. Das Interface „MongoRepository“ bietet bereits eine Vielzahl an CRUD-Operationen an, sodass auch diese nicht mehr implementiert werden müssen. Über weitere Konventionen können auch Sortierung und Pagination einer Menge von Objekten ohne eigene Implementierung genutzt werden.

Hibernate OGM

Mit Hibernate OGM (Object/Grid Mapper) [11] wird versucht, verschiedenen NoSQL-Datenbanken das allseits bekannte JPA-API überzustülpen. Dazu sind entsprechende JPA-Provider implementiert, unter anderem auch für MongoDB, die man wie gewohnt in der Persistence Unit konfiguriert (siehe Listing 13). Die JPA-Annotation definiert das O/D-Mapping und die Abfrage (siehe Listing 14).

CRUD-Operationen und Abfragen werden wie bei jedem JPA-Provider über einen „EntityManager“ durchgeführt. Bei der aktuellen Implementierung war es allerdings nur möglich, Dokumente zu speichern, da die Methoden für die Ausführung von NamedQueries in der Klasse „OgmEntityManager“ allesamt eine „NotSupportedException“ werfen.

Jongo

Unter dem Namen Jongo [12] verbirgt sich ein relativ junges Framework, das nach eigenen Aussagen „der bessere MongoDB-Treiber“ ist. Jongo bietet daher auch ein sehr schmales, aber mächtiges API. Zum Mappen wird die Jackson-Bibliothek verwendet, deren Annotation man auch verwendet (siehe Listing 15). Listing 16 zeigt, wie man Zugriff auf eine Collection erhält.

```
<!-- Connection to MongoDB server -->
<mongo:db-factory host="localhost" port="27017" dbname="test" />
<!-- MongoDB Template -->
<bean id="mongoTemplate"
 class="org.springframework.data.mongodb.core.MongoTemplate">
<constructor-arg name="mongoDbFactory" ref="mongoDbFactory"/>
</bean>
<!-- Package w/ automagic repositories -->
<mongo:repositories base-package="mongodb" />
```

Listing 10

```
public class Order {
        @Id private String id;
        private Date date;
        @Field("custInfo") private String customerInfo;
        List<Item> items;
...
}
public class Item {
        private int quantity;
        private double price;
        @Field("desc") private String description;
...
}
```

Listing 11

```
public interface OrderRepository extends MongoRepository<Order, String> {
        List<Order> findByItemsQuantity(int quantity);
        @Query("{ \"items.quantity\": ?0 }")
        List<Order> findWithNativeQuery(int quantity);
}
```

Listing 12

```
<persistence version="2.0" …>
  <persistence-unit name="primary">
<provider>org.hibernate.ogm.jpa.HibernateOgmPersistence</provider>
  <class>hibernate.Order</class>
  <class>hibernate.Item</class>
  <properties>
    <property name="hibernate.ogm.datastore.provider" value="org.hibernate.ogm.datastore.
mongodb.impl.MongoDBDatastoreProvider"/>
    <property name="hibernate.ogm.mongodb.database" value="odm"/>
    <property name="hibernate.ogm.mongodb.host" value="localhost"/>
    <property name="hibernate.ogm.mongodb.port" value="27017"/>
  </properties>
  </persistence-unit>
</persistence>
```

Listing 13

Über die „MongoCollection“ erhält man Zugriff auf CRUD-Operationen und dort lassen sich auch eigene Abfragen in JSON-Notation mit Platzhaltern durchführen (siehe Listing 17).

Fazit

Wir haben an einem durchgehenden Fallbeispiel gezeigt, wie der eher mit dem JDBC-API vergleichbare MongoDB-Java-Treiber und die abstrakteren, eher JPA-ähnlichen O/D-Mapper verwendet werden können, um Java-Objekt-Netze in einer MongoDB-Datenbank zu persistieren. Jedes der Frameworks bietet ein O/D-Mapping auf Basis von Annotationen mit sinnvollen Defaults an sowie mehr oder weniger abstrakte Ansätze für CRUD-Operationen und eigene Abfragen, die in Tabelle 1 zusammengefasst sind.

```
@Entity
@NamedQuery(
        name="byItemsQuantity",
        query = "SELECT o FROM Order o JOIN o.items i WHERE i.quantity = :quantity"
        )
public class Order {
        @GeneratedValue(generator = "uuid")
        @GenericGenerator(name = "uuid", strategy = "uuid2")
        @Id private String id;
        private Date date;
        @Column(name = "custInfo") private String customerInfo;
        @ElementCollection
        private List<Item> items;
…
}
@Embeddable
public class Item {
        private int quantity;
        private double price;
        @Column(name="desc") private String description;
        ...
}
```

Listing 14

```
public class Order {
        private ObjectId id;
        private Date date;
        @JsonProperty("custInfo") private String customerInfo;
        List<Item> items;
… }
public class Item {
        private int quantity;
        private double price;
        @JsonProperty("desc") private String description;
        …
}
```

Listing 15

```
// Java-Treiber API
MongoClient mc = new MongoClient();
DB db = mc.getDB("odm_jongo");
// Jongo API
Jongo jongo = new Jongo(db);
MongoCollection orders = jongo.getCollection("order");
```

Listing 16

```
Iterable<Order> result =
        orders.find("{\"items.quantity\": #}", 2).as(Order.class);
```

Listing 17

Es stellt sich nun die Frage aller Fragen: „Welches Framework nehme ich?" Sie lässt sich (wie immer) nicht pauschal beantworten. Der Java-Treiber eignet sich grundsätzlich immer. Er hat den größten Funktionsumfang und wird auch regelmäßig mit neuen Versionen von MongoDB veröffentlicht, wohingegen die anderen Frameworks nicht immer so schnell nachziehen (können, da es sich teilweise um Einzelpersonen handelt).

Wenn allerdings viele oder komplexere Abfragen und/oder große Objekt-Netze umzusetzen sind, kann der Code auf Basis des Java-Treibers schnell unübersichtlich werden. Dann bietet sich der Einsatz eines der anderen Frameworks an.

Wenn man ohnehin den Spring-Stack einsetzt, ist Spring Data sicher die natürliche Wahl. Mit dem Repository-Support ist ein Großteil aller Abfragen sehr bequem und auch wartbar umzusetzen. Zudem wird über das Template ein Großteil des MongoDB-API direkt zugreifbar.

Abseits des Spring-Wegs besteht im Prinzip nur die Wahl zwischen Morphia und Jongo, da die MongoDB-Implementierung von Hibernate OGM maximal ein frühes Beta-Stadium darstellt. Daneben zeigt der Versuch, das JPA-API auf NoSQL-Datenbanken aufzupfropfen, dass man die Grundidee hinter NoSQL (noch) nicht vollumfänglich verstanden hat. „@Column"- und „@Join"-Annotation und JP/QL sind bei Datenbanken, denen diese Konzepte nicht bekannt sind, aus Sicht des Autors wenig sinnvoll.

Bei der Wahl zwischen Morphia und Jongo spielen eher nur noch Feinheiten bei den API-Unterschieden eine Rolle, die Mächtigkeit ist in etwa gleich. Morphia hat den Vorteil der Ausgereiftheit, Jongo die Frische (und die Bugs) eines Neulings.

Framework	O/D-Mapping Klasse	O/D-Mapping Attribut	DAO/Repository-Support
Java-Treiber	–	–	–
Morphia	@Entity	@Property	Abstrakte Basis-Klasse „BasicDAO"
Spring Data	@Document	@Field	Interface-Abstraktion mit Sortierung und Pagination
Hibernate OGM	@Entity @Table bzw. @Embeddable	@Column	JPA EntityManager
Jongo		@JsonProperty	„MongoCollection"

Tabelle 1

Quellen

[1] http://www.mongodb.org
[2] https://github.com/mongodb/mongo
[3] http://bsonspec.org
[4] http://docs.mongodb.org/ecosystem/drivers
[5] https://github.com/mongodb/mongo-java-driver/downloads
[6] http://code.google.com/p/morphia
[7] https://github.com/jmkgreen/morphia
[8] http://www.springsource.org/spring-data
[9] Spring Data; M. Pollack, O. Gierke et. al.; O'Reilly Media, 2013
[10] http://www.infoq.com/articles/spring-data-intro
[11] http://docs.jboss.org/hibernate/ogm/4.0/reference/en-US/html_single
[12] http://jongo.org

Vollständige Source-Code-Beispiele stehen unter http://www.github.com/ttrelle.

Tobias Trelle
tobias.trelle@codecentric.de

Dipl.-Math. Tobias Trelle ist Senior IT Consultant bei der codecentric AG, Düsseldorf. Er ist seit knapp 20 Jahren im IT-Business unterwegs und interessiert sich für Software-Architekturen und skalierbare Lösungen. Tobias Trelle organisiert die Düsseldorfer MongoDB-Usergruppe und hält Vorträge zum Thema „MongoDB" auf Konferenzen und bei Usergruppen. Außerdem schreibt er gerade an einem deutschsprachigen Buch über MongoDB im dpunkt-Verlag.

Noch mehr Web-Apps mit „Play!“ entwickeln

Andreas Koop, enpit consulting OHG

In der letzten Ausgabe wurden die Grundlagen des Play-Frameworks vorgestellt. In diesem Artikel geht es jetzt darum, fortgeschrittene Konzepte und Möglichkeiten kennenzulernen, um das notwendige Rüstzeug für das erste Projekt zu haben und die Produktivität des Frameworks in vollem Maße ausschöpfen zu können.

Nahezu jede Web-Anwendung benötigt in irgendeiner Weise einen Session-Handling-Mechanismus, um benutzerbezogene Daten vorzuhalten. Im Gegensatz zum klassischen HTTP-Session-Konzept, bei dem die Daten serverseitig zu einer Benutzer-ID gespeichert sind, nutzt Play das clientseitige Cookie. Dies impliziert, dass die Größe für Session-Daten sehr eingeschränkt ist, nämlich maximal bis zu 4 KB, und dass nur String-Werte möglich sind. Selbstverständlich wird das Cookie mit einem geheimen Schlüssel signiert, damit es nicht manipuliert werden kann. Sollte das Cookie doch in irgendeiner Weise geändert worden sein, wird die Session automatisch invalidiert.

Die Play-Session ist nicht dazu gedacht, Daten zu cachen – wie in der Praxis in Java-EE-Projekten häufig anzutreffen –, sondern maximale Skalierbarkeit zu gewährleisten. Sollten die 4 KB jedoch nicht ausreichen, kann man sich des integrierten Play-Cache (auf Basis von Ehcache [1]) bedienen. Dazu wird einfach eine UUID generiert, die sowohl in der Session abgelegt als auch als Teil der Schlüssel für die im Cache abgelegten Daten verwendet wird.

Wie wird die Session nun in Play-Anwendungen genutzt? Ganz einfach: In allen Play-Controllern stehen Methoden zur Verfügung, um Session-Daten zu speichern, zu lesen oder zu entfernen. Listing 1 zeigt für alle gängigen Operationen ein Beispiel. Das war es dann auch schon zum Thema „Session-Handling“.

Mehrsprachigkeit

Jedes Enterprise-Framework sollte natürlich auch Unterstützung für Mehrsprachigkeit mitbringen, was bei Play auch der Fall ist. Die unterstützten Sprachen werden per ISO-Sprachcode, gefolgt von optionalem ISO-Ländercode, in der Konfigurationsdatei „conf/application.conf“ spezifiziert: „application.langs=de,en,en-US“.

Die mehrsprachigen Texte müssen per Konvention in den Dateien „conf/messages.xxx“ abgelegt sein. Je Sprache hinterlegt man die Texte also zum Beispiel in „conf/messages.de“ oder „conf/messages.en-US“. Daneben gibt es die Standard-Datei „conf/messages“, die auf alle Sprachen zutrifft. Der aktuelle Sprachcode (default) wird aus dem HTTP-Header „Accept-Language“ ermittelt und muss in der „application.conf“ konfiguriert sein. Andernfalls greift die Standard-Sprachdatei.

Für den Zugriff existiert ein – wie schon erwartet – einfach zu benutzendes API „String title = Messages.get(“home.title”)“. Will man explizit auf die Texte einer gewünschten Sprache zugreifen, dann kann der Sprachecode mitgegeben werden: „String title = Messages.get(new Lang(“en-US”), “home.title”)“. Bei komplexeren Texten mit Parametern kommt die bekannte „java.text.MessageFormat“-Bibliothek ins Spiel. Zum Beispiel lassen sich folgende Texte hinterlegen: „contacts.found=In Kategorie {0} wurden {1} Kontakte gefunden.“. Die Parameter werden dann mit „Messa-

```
public static Result login() {
        // TODO do login here...

        session(“username”,“ak@enpit.de”);
        return ok(“Welcome!”);
}

public static checkLogin() {
        String user = session(“username”);

        if(user != null) {
                return ok(“Hello “ + user);
        } else {
                return unauthorized(“Oops, you are not connected”);
        }
}

public static Result logout() {
        String username = session().remove(“username”);
        // oder gleich alle Sessiondaten entfernen
        //session.clear();

        return ok(“Ciao “ + username);
}
```

Listing 1: Session-Daten speichern, lesen, entfernen

```
@Transactional(readOnly = true)
public static Result contacts() {
        final List<Contact> contacts = Contact.find(“”,”asc”,“lastname”);
        return ok( Json.toJson(contacts) );
}
```

Listing 2: JSON-Service-Implementierung im Controller

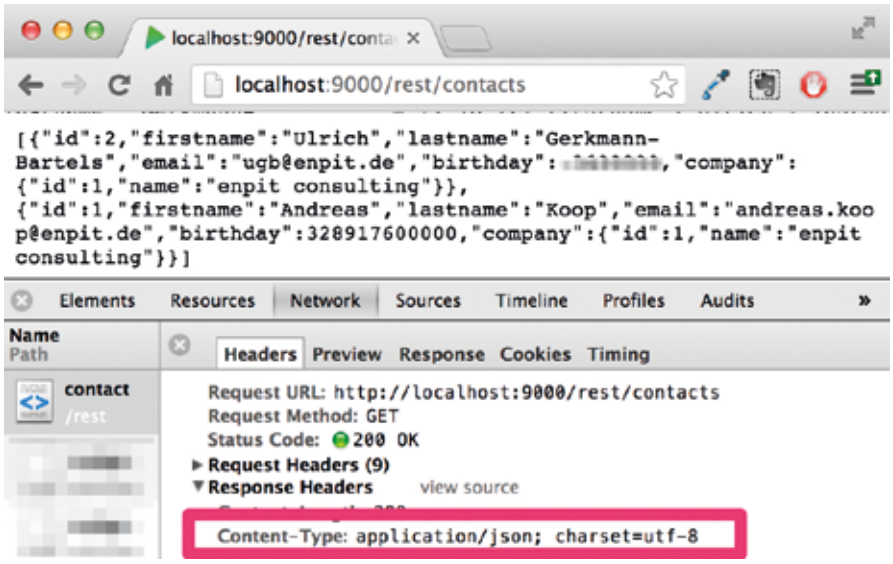


Abbildung 1: Beispiel für einen REST-Service-Call im Browser

ges.get("contacts.found", category, result.size())" übergeben.

Damit nicht genug. Es ist sogar möglich, folgenden Textausdruck zu hinterlegen: „contact.list.result={0,choice,0#Keine Kontakte|1#Ein Kontakt|1<{0,number,integer} Kontakte} gefunden". Die Verwendung im View-Template mittels „@Messages("contact.list.result", contacts.size())" hat zur Folge, dass abhängig von der Ergebnismenge jeweils die Texte „Keine Kontakte", „Ein Kontakt" oder „<Anzahl> Kontakte" ermittelt werden.

RESTful-Services erstellen

RESTful-Services sind sehr populär. Per HTTP lassen sich Ressourcen ansprechen und je nach HTTP-Methode (get, post, put, delete) die typischen CRUD-Operationen ausführen, meist im XML- oder JSON-Format. Durch die Request-basierte Architektur des Play-Frameworks ist es ein Leichtes, derartige Services bereitzustellen, zumal auch ein entsprechendes API für XML beziehungsweise JSON bereitsteht. Um beispielsweise eine Liste von Kontaktdaten per Web-Service im JSON-Format verfügbar zu machen, definiere man im „conf/routes" die gewünschte URL, also „GET / rest/contacts controllers.ContactService.contacts()" und implementiere den Service, wie in Listing 2 dargestellt.

Durch diese wenigen Eingriffe lässt sich der Service bereits nutzen, in der lokalen Entwicklungsumgebung per Aufruf von http://localhost:9000/rest/contacts. Abbildung 1 zeigt das Ergebnis. Analog zum „Get" eines Service lassen sich Operationen für „post", „put" und „delete" implementieren.

Realtime-Push mit WebSockets

Moderne Web-Anwendungen erfordern immer häufiger einen Push-Mechanismus, um Daten in Echtzeit vom Server an den Browser-Client zu senden – sei es für einen Chat oder für die Einblendung von Benachrichtigungen. Obwohl es mit Comet eine Art „Hack" für den Push gibt, hat sich mit WebSocket in den letzten Jahren doch ein Standard herausgebildet, der auch schon von aktuellen Browsern implementiert ist. Vorteil des WebSocket-Protokolls ist, dass es im Gegensatz zu Comet einen bidirektionalen Kommunikationskanal von Client und Server ermöglicht.

Um in Play die neue Technologie nutzen zu können, muss die „Action"-Methode in

```
public static WebSocket<String> echo() {
        return new WebSocket<String>() {
                public void onReady(final WebSocket.In<String> in, final WebSocket.Out<String> out) {
                        in.onMessage(new Callback<String>() {
                                public void invoke(String event) {
                                        out.write("echoing:" + event);
                                }
                        });

                        in.onClose(new Callback0() {
                                public void invoke() {
                                        Logger.info("disconnected");
                                }
                        });

                        // wird einmalig gesendet
                        out.write("connected");
                }
        };
}
```

Listing 3: WebSocket-Action im Controller

```
function setupWebSocket(){
        websocket = new WebSocket("ws://localhost:9000/echo");
        websocket.onopen    = function(evt) { onOpen(evt) };
        websocket.onclose   = function(evt) { onClose(evt) };
        websocket.onmessage = function(evt) { onMessage(evt) };
        websocket.onerror   = function(evt) { onError(evt) };
}
$(document).ready(setupWebSocket);
```

Listing 4: WebSocket-Handler in JavaScript

```
import play.mvc.*;
...
@Security.Authenticated(Security.Authenticator.class)
```

Listing 5: Annotation zum Schutz

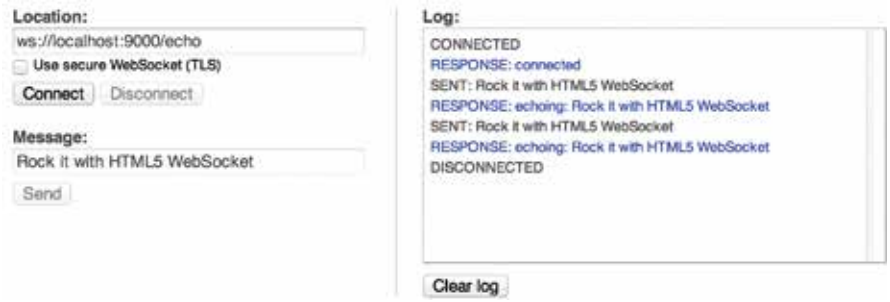


Abbildung 2: Testverlauf des Echo-WebSocket

```
public class Secured extends Security.Authenticator {
        @Override
        public Result onUnauthorized(Context ctx) {
                return redirect(routes.Application.login());
        }
}
```

Listing 6: Weiterleitung zur Login-Maske

```
...
object ApplicationBuild extends Build {
 val appDependencies = Seq(
        ...
        "securesocial" % "securesocial_2.9.1" % "2.0.6"
)
...
```

Listing 7

```
linkedin {
        requestTokenUrl="https://api.linkedin.com/uas/oauth/requestToken"
        accessTokenUrl="https://api.linkedin.com/uas/oauth/accessToken"
        authorizationUrl="https://api.linkedin.com/uas/oauth/authenticate"
        consumerKey="<hier einfuegen>"
        consumerSecret="<hier einfuegen>"
}
```

Listing 8

einem Controller statt eines Result-Objekts ein „WebSocket<Type>"-Objekt zurückliefern. Listing 3 zeigt die Implementierung für einen „Echo-WebSocket".

Zudem ist eine Route (in „conf/routes") erforderlich, um die WebSocket-Action zu adressieren, also „GET /echo controllers. Application.echo()". Nun kann der Echo-WebSocket bereits getestet werden. Am einfachsten geht dies über einen bestehenden Client, der unter „http://websocket.org/echo.html" zu finden ist. Als WebSocket-Lokation wird „ws://localhost:9000/echo eingetragen". Einen Beispielverlauf des Echo-WebSocket zeigt Abbildung 2.

Abbildung 3: Standardmeldung bei Aufruf einer zugriffsgeschützten Seite

In echten Projekten wird der clientseitige Code in der Regel selbst implementiert. Das WebSocket-API ist recht überschaubar. Es werden ein WebSocket-Objekt erzeugt und Callback-Handler für „onopen", „onclose", „onmessage" und „onerror" implementiert (siehe Listing 4).

Mithilfe der registrierten Callback-Funktionen kommuniziert der Client über die geöffnete WebSocket-Verbindung mit dem Server. Der WebSocket wird typischerweise beim „onload"-Event der Webseite geöffnet

Security- und Social-Sign-on

Security ist in Cloud-Zeiten zu einem noch wichtigeren Thema geworden, als es ohnehin schon ist. Erst kürzlich mussten beim führenden Cloud-Anbieter für Notizen über 50 Millionen Passwörter zurückgesetzt werden, da sich Hacker Zugang zu Benutzernamen, E-Mail-Adressen und verschlüsselten Passwörtern verschafft hatten. Play 2 bietet in Puncto Zugriffsschutz von Haus aus ein simples API an. Mittels einer Annotation (siehe Listing 5) können ganze Controller oder einzelne Controller-Actions geschützt werden. Bei Aufruf einer auf diese Weise geschützten URL liefert der Server ein „HTTP 401 Unauthorized" (siehe Abbildung 3).

In der Praxis wird als „Authenticator" eine Subklasse von „Security.Authenticator" implementiert, wo man zum Beispiel anstelle der Standardseite auf die Login-Maske umleitet (siehe Listing 6). Ob ein Benutzer authentifiziert ist, wird per Konvention vom Framework geprüft. In der Session muss ein Benutzername unter dem Schlüssel „username" abgelegt sein.

Die eigentliche Authentifizierungsprüfung obliegt dem Anwendungsentwickler. Er ist dafür verantwortlich, mithilfe des Session-Mechanismus und spezifischer Prüfung der Anmeldedaten die Login/Logout-Funktionalität bereitzustellen. Es gibt keine Framework-Unterstützung für die Authentifizierung mittels LDAP-Provider. Feingranulare Autorisierungsprüfungen müssen ebenso selbst implementiert werden.

Eine elegante Authentifizierung kann heutzutage auch über bestehende Social-Network-Provider erfolgen. Erfreulicherweise existiert dafür ein sehr mächtiges Plug-in für Play-Anwendungen: SecureSocial [2]. Es funktioniert mit den gängigsten Plattformen wie Twitter, Google, Facebook, LinkedIn, XING oder GitHub und hat ein API, das die Implementierung eigener Authentifizierungsroutinen erlaubt. Folgende Schritte sind notwendig, um beispielsweise die Authentifizierung mit LinkedIn durchzuführen:

Als erstes die SecureSocial-Dependencies im Projekt konfigurieren (siehe Listing 7). Gemäß Dokumentation die Konfigurationsdatei „conf/securesocial.conf" einbinden und die LinkedIn-Einstellungen vervollständigen. Die jeweiligen Keys erhält man im Developer-Bereich von LinkedIn,

indem man eine neue Applikation registriert (siehe Listing 8).

Gemäß Dokumentation sind die SecureSocial-Routen in der „conf/routes"-Datei hinzuzufügen. Aufgrund der Menge an Routen sind hier beispielhaft nur zwei URLs definiert (siehe Listing 9). Nun können per Annotation die gewünschten Controller-Actions vor unberechtigtem Zugriff geschützt werden (siehe Listing 10).

Mit der beschriebenen Konfiguration der Play-Anwendung wird bei Zugriff auf die Kontaktseite („localhost:9000/contacts") automatisch zu LinkedIn geleitet, um sich zu authentifizieren und bei erstmaligem Zugriff die Erlaubnis zu erteilen, auf die Profildaten zuzugreifen. Damit können dann beispielsweise der Name und das Profilbild in der eigenen Anwendung verwendet werden. Abbildung 4 zeigt diesen Prozess.

```
# User Registration
GET  /signup securesocial.controllers.Registration.startSignUp
POST /signup securesocial.controllers.Registration.handleStartSignUp
```

Listing 9

```
@SecureSocial.SecuredAction
@Transactional(readOnly = true)
public static Result contacts(....) { ...
```

Listing 10

```
app/assets/javascripts/main.coffee
$ ->
        alert "Mmmmh, lecker Kaffee.."
```

Listing 11

```
...
<script type="text/javascript" src="@routes.Assets.at("javascripts/main.js")" />
</script>
...
```

Listing 12

Abbildung 4: Social-Sign-on mit SecureSocial

CoffeeScript- und LESS-Integration

Das „Convention over Configuration"-Prinzip in Play ist wirklich mächtig. Neben den Standard-Webskripten wie JavaScript und CSS werden CoffeeScript [3] und LESS [4] unterstützt. Beides sind einfache Skriptsprachen, die einige Nachteile und Einschränkungen von JavaScript und CSS umgehen. Allerdings müssen diese Skripte zunächst nach JavaScript beziehungsweise CSS kompiliert sein, um von Browsern interpretiert werden zu können. Genau dafür bietet Play von Haus aus einzigartige Unterstützung. Um ein CoffeeScript zu verwenden, muss es lediglich im Verzeichnis „app/assets/javascripts" abgelegt werden (siehe Listing 11).

Dieser Zweizeiler sorgt dafür, dass beim „document.ready"-Event ein JavaScript-Alert mit dem entsprechenden Text angezeigt wird. Damit dieses Skript zur Laufzeit wirksam wird, muss das generierte, korrespondierende JavaScript im View-Template referenziert sein. Dies geschieht per Konvention (siehe Listing 12).

Es wird einfach angenommen, dass eine zur CoffeeScript-Datei korrespondierende JavaScript-Datei vorliegt. Die Erstellung des JavaScripts übernimmt ein in Play integrierter Compiler, der bei jeder Änderung der CoffeeScript-Datei zuschlägt und das jeweilige JavaScript im target-Verzeichnis erzeugt: „target/scala-<version>/classes/public/javascripts/main.js" (siehe Listing 13) – dank Hot-Reloading extrem produktiv.

LESS CSS in Play-Projekten zu verwenden, ist genauso einfach. Die LESS-Dateien werden per Konvention in „app/assets/stylesheets/" abgelegt, also beispielsweise „app/assets/stylesheets/main.less" (siehe Listing 14). Analog zum CoffeeScript-Compiler wird von Play der LESS-Compiler bei Änderungen angestoßen, um die entsprechende CSS-Datei zu generieren. Die Einbindung in View-Templates geschieht ebenso plausibel (siehe Listing 15). Zur Laufzeit wird valides CSS dargestellt (siehe Listing 16).

Less-CSS ist eine wertvolle Sprache, um effektives CSS zu schreiben. Neben Variablen können auch eine Art „Makros" und Berechnungen vorgenommen werden.

WebJars

Im Zuge der Popularität von immer mehr clientseitigen Web-Bibliotheken wie jQuery oder Twitter Bootstrap wird es zunehmend schwieriger, in Web-Projekten den Überblick über die Abhängigkeiten und jeweiligen Versionen zu behalten. Aus diesem Grund hat James Ward (Typesafe Inc.) letztes Jahr (2012) eine Initiative gestartet und das Projekt WebJars [5] ins Leben gerufen. Die Idee ist genial: Die clientseitigen Web-Bibliotheken werden analog zu Java-Bibliotheken in JAR-Dateien gepackt und in der jeweils passenden Version per Maven, Gradle oder SBT in die gewünschte Projekt-Umgebung geladen. Für das Laden der Dateien aus dem JAR existiert ein ResourceLoader beziehungsweise ein Routing, das die Referenzierung der Dateien direkt in View-Templates ermöglicht. Um WebJars in eine Play-2-Anwendung zu integrieren, müssen also lediglich die gewünschten Dependencies in die „project/Build.scala"-Konfigurationsdatei eingebunden werden (siehe Listing 17).

Die WebJars-Hilfsbibliothek „webjars-play" enthält den notwendigen Ressourcen-Controller, um die clientseitigen Dateien direkt aus dem JAR-File zu referenzieren. Das Ganze erfolgt über eine entsprechende Routing-Konfiguration in

```
(function() {
        $(function() {
                return alert("Mmmmh, lecker Kaffee..");
        });
}).call(this);
```

Listing 13

```
@mainBgColor: white;
@mainColor:   blue;

body {
        background-color: @mainBgColor;
        color: @mainColor;
}

h2 {
        color: @mainColor;
}
```

Listing 14

```
<link rel="stylesheet" media="screen"
href="@routes.Assets.at("stylesheets/main.
css")">
```

Listing 15

```
body {
        background-color: #ffffff;
        color: #0000ff;
}
h2 {
        color: #0000ff;
}
```

Listing 16

```
...
object ApplicationBuild extends Build {
 val appDependencies = Seq(
        ...
        "org.webjars" % "webjars-play" % "2.1.0-1",
        "org.webjars" % "bootstrap" % "2.1.1"
)
...
```

Listing 17

```
<link rel='stylesheet' href='@routes.
WebJarAssets.at(WebJarAssets.locate("css/
bootstrap.min.css"))'>

<script type='text/javascript' src='@
routes.WebJarAssets.at(WebJarAssets.
locate("jquery.min.js"))'></script>
```

Listing 18

„conf/routes": „GET /webjars/*file controllers.WebJarAssets.at(file)". Um die URL der clientseitigen Dateien aus den WebJars zu referenzieren, verwendet man den Reverse-Routing-Mechanismus von Play in den View-Templates (siehe Listing 18).

Mittels WebJars lässt sich das Dependency-Management clientseitiger Web-Bibliotheken stark vereinfachen. Um eine clientseitige Bibliothek auf die nächste Version zu aktualisieren, muss lediglich die Versionsnummer in der Projekt-Konfiguration geändert werden. Beim nächsten „play run" werden die notwendigen JavaScript- und CSS-Dateien als WebJar gegebenenfalls vom Repository geladen und automatisch in der richtigen Version im View-Template eingebunden.

Deployment in der Cloud

Da Play-Anwendungen auf Skalierbarkeit ausgelegt sind, ist die Cloud die prädestinierte Runtime-Umgebung. Cloud-Anbieter für Play-Anwendungen der ersten Stunde war Heroku [6]. Um eine Anwendung auf der Heroku-Cloud zu installieren, sind nur wenige Handgriffe notwendig. Intern verwendet Heroku Git [7]. Mit nur wenigen Handgriffen lässt sich eine lokal laufende Play-Anwendung in die Cloud heben. Der erste Schritt ist die Installation eines Git-Clients und der Heroku-Toolbelt-Kommandozeilen-Tools. Dann gilt es sicherzustellen, dass die Tools „git" und „heroku" im Suchpfad des Terminals/der Konsole enthalten sind. Da Heroku PostgreSQL als Datenbank nutzt, muss der entsprechende Treiber in die Dependencies aufgenommen werden (siehe Listing 19).

Anschließend legt man „$project/Procfile" an (siehe Listing 20). Mithilfe des „Procfile" werden Konfigurations-Parameter der Heroku-Umgebung an die jeweilige Anwendung übergeben. Die Projekt-Sourcen werden in ein lokales Git-Repository eingecheckt (siehe Listing 21) und die neue Anwendung in Heroku registriert (siehe Listing 22).

Alle vorhergehenden Schritte sind einmalig für eine neue Anwendung auszuführen. Alle nachfolgenden inkrementellen Cloud-Deployments erfolgen per Git-Push

```
val appDependencies = Seq(
..
        "postgresql" % "postgresql" % "9.1-901-1.jdbc4"
..
)
```

Listing 19

```
web: target/start -Dhttp.port=${PORT} -DapplyEvo-
lutions.default=true -Ddb.default.driver=org.post-
gresql.Driver -Ddb.default.url=$DATABASE_URL
${JAVA_OPTS}
```

Listing 20

```
$ git init
Initialized empty Git repository in .../enpit.sample.
        play2/contact/.git/
$ git add .
$ git commit -m "Initial commit"
[master (root-commit) 3fac732] Initial commit
134 files changed, 2721 insertions(+), 0 deletions(-)
```

Listing 21

```
$ heroku login
Enter your Heroku credentials.
Email: andreas.koop@enpit.de
Password ****
Authentication successful.
$ heroku create -s cedar myapp-name
$ git remote add heroku git@heroku.com:myapp-name.git
$ heroku keys:add ~/.ssh/id_rsa.pub
```

Listing 22

„$ git push heroku master". In unserem Beispiel funktionierte alles ohne Vorkommnisse beim ersten Mal. Neben dem Deployment auf Heroku gibt es mittlerweile auch „first class"-Support auf der CloudBees-Plattform. CloudBees unterstützt sogar das WebSocket-Protokoll, sodass sämtliche Play-Features auch in der Cloud genutzt werden können.

Fazit

Das Play-Framework ist sehr mächtig, aber in Version 2 noch sehr jung. Die stark ausgeprägte „Featuritis" bringt mit jedem Release teils gravierende Änderungen mit sich. Viele Plug-ins von Version 2.0 sind nicht mehr mit Version 2.1 kompatibel. Aus der Version 1.2.x hat sich ein Fork namens Yalp [8] herausgelöst, nachdem sich eine Mehrheit in der Community mit 66,2 Prozent dafür ausgesprochen hat [9]. Blickt man ein wenig zurück, so muss man eingestehen, dass

Version 1.2.x von Play einen wirklich runden Eindruck hinterließ. Es gab für alle Belange einer Web-Anwendung exakt eine Best-Practice-Implementierung. Diesen Rundumschlag lässt Play 2 leider vermissen.

Aus rein technologischer Sicht mag Play 2 mit der Umstellung auf Scala seine Berechtigung haben. Betrachtet man das Framework unter dem Gesichtspunkt der Nachhaltigkeit und strategischen Ausrichtung bei der Implementierung von Unternehmensanwendungen, so hat das Framework leider noch nicht die notwendige Reife. Zudem bleibt fraglich, ob es sich langfristig auszahlt, auf ein Framework in der Java-Welt jenseits des Java-EE-Standards zu setzen. Für kritische, nachhaltige Unternehmensanwendungen bietet beispielsweise Oracle ADF eine auf Standards basierende Alternative, in der neueste Technologien wie CSS3 und HTML5 integriert sind – mit der Essentials Edition sogar lizenzkostenfrei.

Trotz dieser kritischen Punkte bleibt das Play-2-Framework interessant und unter der Federführung der Typesafe Inc. eines mit möglicherweise rosiger Zukunft. Es bleibt spannend in der Welt der Web-Frameworks.

Weitere Informationen und Verweise

- „Webapps mit Play! entwickeln – Nichts leichter als das", JavaAktuell 2/2013. Es empfiehlt sich, diesen Artikel im Vorfeld gelesen zu haben. Dort finden sich alle notwendigen Verweise zum Play-Framework. Der Einstieg ist http://www.playframework.org

[1] Ehcache: http://ehcache.org
[2] SecureSocial: http://securesocial.ws
[3] CoffeeScript: http://coffeescript.org
[4] LESS: http://www.lesscss.de
[5] WebJars: http://www.webjars.org
[6] Heroku, Cloud: http://www.heroku.com
[7] Git, SCM: http://git-scm.com
[8] Yalp-Framework: https://github.com/yalp-framework/yalp
[9] Play 1 Fork Umfrage: https://www.surveymonkey.com/sr.aspx?sm=uRhTJLButShLupD_2bknn5YKhStcjfFZ1eHw4i7wg5X9w_3d

Andreas Koop
andreas.koop@enpit.de

Dipl.-Inf. Andreas Koop ist Geschäftsführer des Beratungsunternehmens enpit consulting OHG. Sein beruflicher Schwerpunkt liegt in der Beratung, dem Training und der Architektur für Oracle-ADF- und WebLogic-Server-Projekte. Daneben beschäftigt er sich mit der Entwicklung von mobilen Apps auf Basis von HTML5 sowie Oracle ADF Mobile und evaluiert aktuelle Web-Technologien und -Frameworks.

Flexible Suche mit Lucene

Florian Hopf, Freiberuflicher Software-Entwickler

Um große Textmengen zu durchsuchen, reichen herkömmliche Mechanismen oft nicht mehr aus. Eine indexbasierte Suche auf Basis der Open-Source-Bibliothek Apache Lucene bietet dann viele Vorteile.

Wenn große Mengen semistrukturierter Daten vorliegen, ergibt sich häufig das Problem, bestimmte Informationen aufzufinden. Während eine SQL-Datenbank gute Zugriffsmöglichkeiten auf strukturierte Daten bietet, kommt man beim Suchen in Texten schnell an die Grenzen. Unter anderem ergeben sich Probleme hinsichtlich der Skalierbarkeit und bei großen Datenmengen ist ein Durchsuchen der Daten zum Abfragezeitpunkt, beispielsweise über eine LIKE-Query, nicht mehr sinnvoll. Zu-

sätzlich fehlen Features, die bei modernen Such-Lösungen erwartet werden: Die Suche soll tolerant hinsichtlich der Schreibweise des Suchbegriffs sein und die Ergebnisse sollen sortiert nach Relevanzkriterien zurückgeliefert werden. Eine dafür besser geeignete Technologie ist die indexbasierte Suche, wie sie zum Beispiel von der Bibliothek Lucene und den darauf aufbauenden Suchservern Solr und ElasticSearch zur Verfügung gestellt wird.

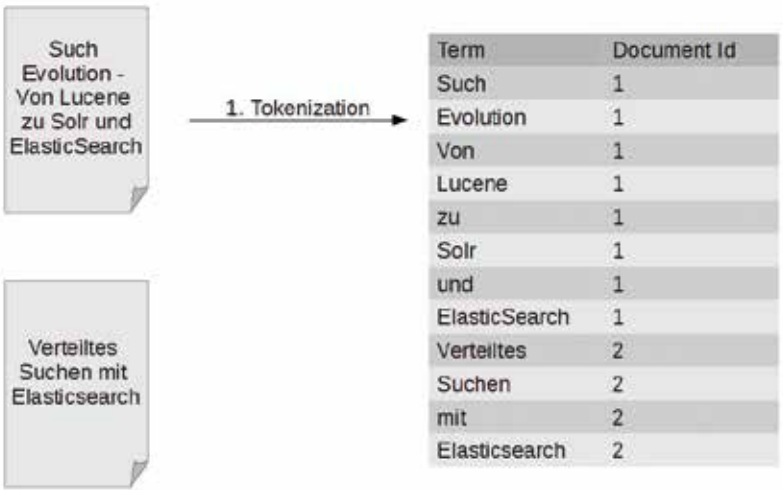
Indexbasierte Suche

Das Grundprinzip einer indexbasierten Suche ist die Aufbereitung der zu durchsuchenden Inhalte und die Ablage in einem für die Suche optimierten Format in einer separaten Datenstruktur, dem invertierten Index. Der Index ähnelt im Kern einer Map, in der für einzelne Teile der Texte, die sogenannten „Terme" als Referenzen auf die Dokumente gespeichert werden, in denen sie auftauchen. Eine Suche besteht dann nur noch aus einem Lookup auf den Term und ist dadurch sehr schnell.

Der komplexe Vorgang, den Eingabetext zu parsen, wird der Suche vorgelagert und beim Indizieren, dem Aufbau der Datenstruktur, ausgeführt. Da der Index eine zusätzliche Ablage darstellt, ist es auch gleichgültig, woher die Daten ursprünglich stammen. Es ist möglich, Daten aus unterschiedlichen Quellen, beispielsweise aus einer Datenbank und aus Dateien im Filesystem, gemeinsam in einem Index abzulegen und zu durchsuchen.

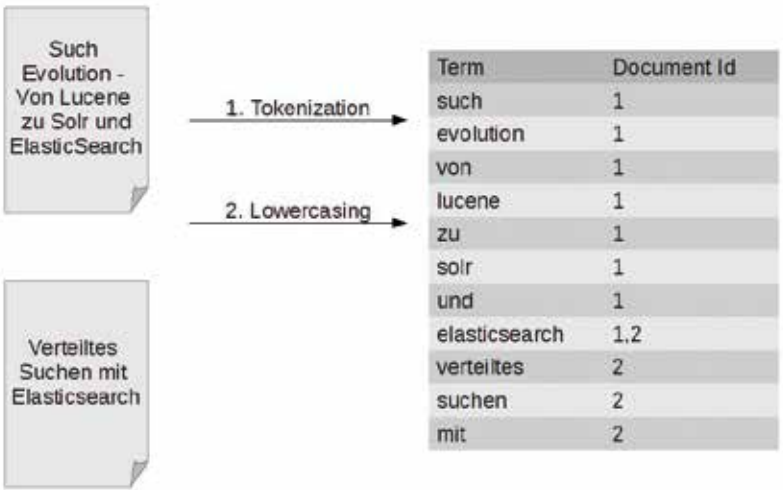
Welche Einheiten eines Textes die Terme bilden, ist anwendungsabhängig, das Aufsplitten erfolgt beim sogenannten „Analyzing". Meist werden dabei mehrere Verarbeitungsschritte durchgeführt, wie wir im folgenden Beispiel sehen werden. Als Eingabe verwenden wir zwei Dokumente, die jeweils aus einem kurzen Vortragstitel bestehen: „Such-Evolution – von Lucene zu Solr und ElasticSearch" und „Verteiltes Suchen mit ElasticSearch". Jedes indizierte Dokument bekommt eine Id zugewiesen, die als Verweis verwendet werden kann.

In einem ersten Schritt werden die Wörter des Textes extrahiert sowie Leer- und Satzzeichen entfernt. Abbildung 1 zeigt den Inhalt des Index nach der Verarbeitung. Um unabhängig von der Groß- und Kleinschreibung zu bleiben, können alle Terme in Kleinbuchstaben umgewandelt werden. Danach sieht unser Index dann wie in Abbildung 2 aus.



Term	Document Id
Such	1
Evolution	1
Von	1
Lucene	1
zu	1
Solr	1
und	1
ElasticSearch	1
Verteiltes	2
Suchen	2
mit	2
Elasticsearch	2

Abbildung 1: Index nach dem Aufsplitten der Sätze in Wörter (Tokenization)



Term	Document Id
such	1
evolution	1
von	1
lucene	1
zu	1
solr	1
und	1
elasticsearch	1,2
verteiltes	2
suchen	2
mit	2

Abbildung 2: Index nach dem Umwandeln in Kleinbuchstaben (Lowercasing)

Wenn man sich den Prozess beim Auffinden eines Dokuments vor Augen führt, wird man feststellen, dass eine Verarbeitung nur beim Indizieren nicht ausreichend ist. Zwar ist es jetzt möglich, kleingeschriebene Suchbegriffe zu matchen, großgeschriebene Begriffe führen allerdings aufgrund des Lowercasing während der Indizierung zu keinen Treffern. Um die Modifikationen an den Termen beim Indizieren zu kompensieren, sind die gleichen Schritte auch für den Suchbegriff durchzuführen. Dadurch ist garantiert, dass gleiche Wörter beim Indizieren und Suchen auf denselben Term zurückgeführt werden.

Eine einfache Suche mit exakten Treffern ist damit bereits möglich, der Eindruck einer intelligenten Suche entsteht allerdings erst durch weitere Schritte. Der oft verwendete sprachabhängige Prozess „Stemming" führt Begriffe auf ihren Wortstamm zurück. So werden die Begriffe „Such" und „Suchen" beide in den Term „such" überführt. Dadurch sind Abweichungen wie Einzahl oder Mehrzahl nicht mehr relevant, da sie auf denselben Stamm zurückgeführt werden. Unser Index könnte dann nach Durchführung des Stemming wie in Abbildung 3 aussehen.

Es wurde bereits erwähnt, dass der Analyzing-Prozess anwendungsabhängig ist. Die meisten der Schritte sind verlustbehaftet, das heißt, es gehen potenziell Unterschiede zwischen Wörtern verloren. Groß- und Kleinschreibung kann für einzelne Anwendungen eine Rolle spielen, außerdem kann das Stemming, da es sich um einen algorithmischen Prozess handelt, Be-

griffe, die nicht nach den entsprechenden Regeln aufgebaut sind, falsch modifizieren.

Generell gilt es bei diesen Überlegungen abzuwägen, ob möglichst viele eventuell passende Ergebnisse zurückgeliefert werden oder ob die zurückgelieferten Ergebnisse auf jeden Fall passen sollen, mit der Gefahr, dass nicht alle passenden Dokumente gefunden werden. Im Information Retrieval beschreiben diese Aspekte die Begriffe „Recall" (möglichst viele gute Ergebnisse zurückliefern) und „Precision" (möglichst wenige schlechte Ergebnisse zurückliefern).

Neben den hier gezeigten Schritten sind noch viele weitere denkbar: Terme können automatisch mit Synonymen angereichert werden, häufig vorkommende Füllwörter können entfernt oder Sonderzeichen normalisiert werden. Wie der Analyzing-Prozess einer Suchanwendung aussieht, muss sorgfältig mit den eigenen Anforderungen abgeglichen werden, da er die Qualität der Suchergebnisse maßgeblich beeinflusst.

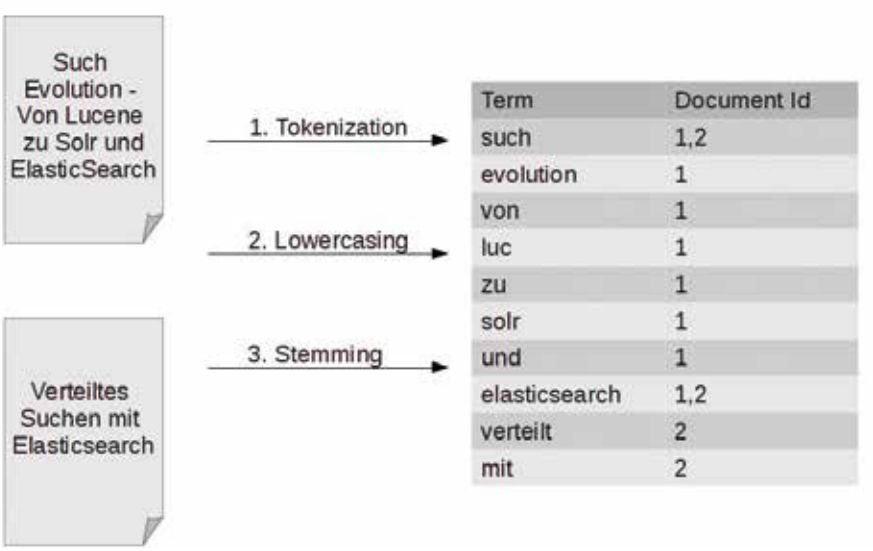


Abbildung 3: Index nach Durchführen des Stemming

Lucene

In der Java-Welt ist die Bibliothek Lucene für Such-Anwendungen seit Jahren fest etabliert. Sie bietet die Implementierung eines invertierten Index mit unterstützenden Datenstrukturen und Möglichkeiten, diese Strukturen zu schreiben, zu lesen und zu durchsuchen. Im Kern und als zusätzliche Module sind zahlreiche für das Analyzing notwendige Klassen verfügbar, insbesondere „Tokenizer", die Texte aufsplitten, „TokenFilter", die Änderungen wie das Lowercasing und Stemming durchführen, und „Analyzer", die aus Tokenizern und TokenFiltern bestehen und für häufige Einsatzzwecke schon fertig zur Verfügung stehen. Für die Suche können unterschiedliche Queries verwendet werden, die entweder in Textform verarbeitet oder programmatisch konstruiert werden können. Häufige Anwendungsfälle wie Keyword-Suche (Term-Query), Ähnlichkeitssuche (Fuzzy-Query) oder Phrase-Queries werden direkt unterstützt.

Um einen ersten Eindruck vom Indizieren und Suchen mit Lucene zu bekommen, betrachten wir die Verarbeitung von Vortragsankündigungen. Diese bestehen teils aus strukturierten Daten wie Sprecher und Datum, aber auch aus textuellen Daten, die durchsucht werden sollen, wie dem Titel. Ein ausführlicheres Beispiel in Form einer Webanwendung findet sich auch auf GitHub unter https://github.com/fhopf/lucene-solr-talk.

Die Ein- und Ausgabe erfolgt in Lucene jeweils durch Instanzen der Klasse „Document". Ein Document ist eine Sammlung von Feldern, die jeweils aus einem Namen und einem Wert bestehen. Welche Felder verfügbar sind, wird von der Anwendung bestimmt; Lucene ist intern schemafrei und daher vergleichbar mit Document-Stores aus der NoSQL-Welt. Zur Bestimmung, wie ein Feld verarbeitet wird, können weitere Eigenschaften konfiguriert werden:

- *Indexed*
  Bestimmt, ob ein Feld in den Index geschrieben wird und ob es vom Analyzer verarbeitet werden soll
- *Stored*
  Bestimmt, ob der Originalinhalt zusätzlich gespeichert werden soll, um ihn bei Anzeige der Suchergebnisse verfügbar zu haben

Seit Lucene 4 müssen diese Eigenschaften nicht mehr direkt gesetzt werden; für häufige Anwendungsfälle stehen unterschiedliche Field-Implementierungen zur Verfügung, beispielsweise „TextField", wenn der Inhalt indiziert und „analyzed" sein soll, oder „StringField", wenn zwar die Indizierung, nicht aber das Analyzing durchgeführt werden soll. Für unser Beispiel entscheiden wir, dass alle Felder bei der Darstellung der Suchergebnisse zur Verfügung stehen sollen. Titel und Datum werden in den Index geschrieben, allerdings soll lediglich der Titel vom Analyzer verarbeitet werden (siehe Listing 1).

Zum Indizieren benötigen wir einen „IndexWriter", der Daten in den Index schreiben kann. Der Speicherort des Index ist durch die Klasse „Directory" abstrahiert, von der es unterschiedliche Implementierungen gibt, die die Daten beispielsweise optimiert für unterschiedliche Betriebssysteme auf der Festplatte ablegen oder die Struktur direkt im Speicher halten. Der „IndexWriter" wird unter anderem mit dem „Analyzer", der für das Aufsplitten der Inhalte verantwortlich ist, und dem „Directory" konfiguriert. Anschließend können eines oder mehrere Dokumente über „addDocument" zum Index hinzugefügt werden. In diesem Moment stehen die Dokumente allerdings noch nicht zur Suche zur Verfügung. Lucene speichert den Index in Segmenten, die jeweils einen Teil des Index enthalten. Erst nach einem „commit" kann dieses Segment durchsucht werden (siehe Listing 2).

Für den lesenden Zugriff auf den Index steht der „IndexSearcher" zur Verfügung. Dieser bietet Methoden, um für übergebene Query-Instanzen die zugehörigen Ergebnisse aus dem Index zu lesen. Eine Query-Instanz kann entweder über einen „QueryParser" aus einem String erzeugt oder programmatisch zusammengebaut werden. Die vom Standard-QueryParser an-

gebotene Lucene-Syntax bietet viele Möglichkeiten, etwa um Teilbegriffe als optional zu markieren, Begriffe auszuschließen oder Phrase- und Fuzzy-Queries durchzuführen. Entscheidet man sich dafür, die Query programmatisch selbst aufzubauen, muss man beachten, dass man das Analyzing selbst durchzuführen hat, da dieses ansonsten im „QueryParser" stattfindet (siehe Listing 3).

Zur Ausgabe der Suchergebnisse werden ebenfalls wieder Instanzen von „Document" verwendet, die über die „TopDocs" identifiziert werden können und über den „IndexSearcher" auslesbar sind. Im Gegensatz zu den beim Indizieren übergebenen Dokumenten sind nun allerdings nur noch die als „stored" markierten Felder verfügbar (siehe Listing 4).

Die Reihenfolge der zurückgelieferten Dokumente ist durch den Relevanzmechanismus in Lucene bestimmt, standardmäßig kommt hier der TF/IDF-Algorithmus zum Einsatz. Dieser basiert auf der Annahme, dass Dokumente, die einen Begriff häufiger enthalten, eine höhere Relevanz für den User haben. Gleichzeitig werden Begriffe, die im Gesamt-Index sehr häufig vorkommen, als weniger wichtig angesehen. Die Relevanz ist durch Mechanismen wie „Boosting" zur Index- oder Suchzeit auch beeinflussbar. Dabei werden bestimmte Dokumente, Felder oder Terme mit einem Boost-Faktor versehen, der den Scoring-Wert eines Dokuments und damit die Sortierung beeinflusst. Zusätzlich ist es auch möglich, den verwendeten Algorithmus auszutauschen, entweder durch eine der weiteren mitgelieferten Implementierungen oder durch einen eigenen Mechanismus.

Die Sortierung nach Relevanz ist einer der großen Vorteile bei der Implementierung einer Anwendung mit Lucene. Es ist bei Bedarf jedoch auch möglich, die Ergebnisse nach einem bestimmten Feld zu sortieren, beispielsweise bei Indizierung des Datumsfeldes die Anzeige sortiert nach der Aktualität.

```
Document doc = new Document();
doc.add(new TextField("title", "Such Evolution", Field.Store.YES));
doc.add(new StoredField("speaker", "Florian Hopf"));
doc.add(new StringField("date", "20130704", Field.Store.YES));
```

Listing 1

```
Directory directory = FSDirectory.open(new File("/tmp/talk-index"));
IndexWriterConfig config = new IndexWriterConfig(Version.LUCENE_43, new
        GermanAnalyzer(Version.LUCENE_43));
try (IndexWriter writer = new IndexWriter(directory, config)) {
        writer.addDocument(doc);
        writer.commit();
}
```

Listing 2

```
IndexReader reader = DirectoryReader.open(directory);
IndexSearcher searcher = new IndexSearcher(reader);
QueryParser parser = new QueryParser(Version.LUCENE_43, "title", new
        GermanAnalyzer(Version.LUCENE_43));
Query query = parser.parse("suchen");
TopDocs topDocs = searcher.search(query, 5);
assertEquals(1, topDocs.totalHits);
```

Listing 3

```
ScoreDoc scoreDoc = topDocs.scoreDocs[0];
Document result = searcher.doc(scoreDoc.doc);
assertEquals("Florian Hopf", result.get("speaker"));
```

Listing 4

Ausblick

Neben den hier angesprochenen Features von Lucene sind noch sehr viele weitere Funktionen verfügbar. Sehr populär ist beispielsweise das in der nächsten Ausgabe beschriebene „Faceting", mit dem Suchergebnisse kategorisiert werden können. „Highlighting" ermöglicht das Hervorheben der Fundstelle im Text, um den Kontext schnell zu erfassen. „Result Grouping" dient dem Zusammenfassen von Suchergebnissen anhand bestimmter Kriterien, über „Suggester" können Vorschläge zu sinnvollen Suchen gegeben werden.

Mit Lucene kann mit relativ wenig Code das Grundgerüst einer Such-Anwendung implementiert werden. Die wenigen Zeilen, die wir gesehen haben, ermöglichen uns bereits das Indizieren von Dokumenten und die flexible Suche darin. Geht es jedoch darum, eine Anwendung in Produktion zu nehmen, sind noch weitere Faktoren zu beachten, die eine höhere Komplexität bringen können. Beispielsweise sollten wir vermeiden, den „IndexReader" bei jeder Anfrage zu erzeugen, da es sich dabei um eine sehr teure Operation handelt.

Zur Implementierung fortgeschrittener Features ist teilweise ein tieferes Verständnis der Interna von Lucene notwendig. Schließlich kann es noch passieren, dass unsere Datenmenge ansteigt und wir die Dokumente auf mehrere Indizes zu verteilen haben, oder wir müssen uns um Ausfallsicherheit Gedanken machen. Diese Anforderungen sind mit den auf Lucene aufbauenden Suchservern Apache Solr und ElasticSearch oftmals einfacher umzusetzen. In der nächsten Ausgabe werden wir deshalb einen Blick auf diese beiden Systeme werfen und sehen, dass sie uns einiges an Arbeit abnehmen können.

Links

- http://lucene.apache.org
- https://github.com/fhopf/lucene-solr-talk

Florian Hopf
mail@florian-hopf.de

Florian Hopf arbeitet als freiberuflicher Software-Entwickler mit den Schwerpunkten „Content Management" und „Suchlösungen" in Karlsruhe. Er setzt Lucene und Solr seit Jahren in unterschiedlichen Projekten ein und ist einer der Organisatoren der Java User Group Karlsruhe.

Automatisiertes Behavior Driven Development mit jbehave

Matthias Balke und Sebastian Laag, adesso AG

Eine der großen Herausforderungen der agilen Software-Entwicklung ist die Zusammenarbeit der Bereiche „Entwicklung", „Qualitätsmanagement" und „Business-Analyse". Behavior Driven Development (BDD) ist ein Ansatz, um diese Herausforderung zu meistern. Dabei wird das Zusammenspiel der einzelnen Bereiche gestärkt, indem man Anforderungen durch sogenannte „Szenarien" gemeinsam beschreibt. So lassen sie sich später auf einfache Weise automatisieren.

Zur Realisierung des Behavior Driven Development existieren Frameworks, um das gemeinsam festgelegte Verhalten einer Anwendung sowohl textuell als auch programmatisch festzuhalten. Zudem ermöglichen sie die Einbindung der geschriebenen Tests in einen Continuous-Integration-Prozess. Einige Frameworks bieten auch die spätere Integration zur übersichtlichen Ansicht der Testergebnisse in Continuous-Integration-Server. Der Artikel beschreibt die einzelnen Schritte des BDD-Ansatzes bis hin zum automatisierten Testen einer Anwendung mit JBehave.

Die Wurzeln des BDD

Im Jahr 2006 veröffentlichte Dan North den ersten Artikel zum Thema „Behavior Driven Development" als Antwort auf die bisherigen Erfahrungen mit Test Driven Development (TDD) [1]. Die Erfahrung in seinen Projekten zeigte, dass er bei der Software-Entwicklung mit TDD oft das Gefühl hatte, „wenn mir das doch jemand vorher gesagt hätte". Daraufhin erfand er das heute bekannte Verfahren „BDD". Es ermöglicht auf einfache Weise, ein in einer natürlichen Sprache genau spezifiziertes Verhalten einer Anwendung direkt in einen Test zu überführen. Um das Verfahren in der Praxis anwenden zu können, entwickelte Dan North das Framework JBehave. Bei BDD geht es nicht nur um das Testen einer Anwendung, sondern vielmehr darum festzulegen, was umgesetzt werden soll, bevor es angegangen wird [2].

BDD in der Theorie …

Für Behavior Driven Development existiert eine Vielzahl verschiedener Frameworks. Dabei unterscheiden sich diese in der unterstützten Programmiersprache, der zu verwendenden Domain Specific Language (DSL) oder der Ausgabe der Test-Reports. Alle Frameworks unterstützen Englisch als Beschreibungssprache. Für die meisten gibt es allerdings auch Möglichkeiten zur Internationalisierung, sodass auch Deutsch, Französisch oder andere Sprachen möglich sind.

Jede Sprache nutzt bei der Spezifikation des Verhaltens die Schlüsselwörter „GIVEN", „WHEN" und „THEN". „GIVEN" ist dabei die Annahme, auf der das Verhalten basiert, „WHEN" die Aktion, die im Rahmen des Verhaltens ausgeführt wird und „THEN" definiert das erwartete Ergebnis des Verhaltens. Aktion und Ergebnis lassen sich mit dem Schlüsselwort „AND" erweitern, sodass mehrere Aktionen beziehungsweise Ergebnisse möglich sind. Mithilfe dieser Schlüsselwörter entsteht eine sogenannte „Story-Datei", um ein spezielles Szenario einer Anwendung zu beschreiben.

… und in der Praxis

Nachfolgend ist in einer solchen Story-Datei das Verhalten einer Einkaufsliste beschrieben, um den Ablauf der Implementierung eines Tests mit JBehave zu verdeutlichen (siehe Listing 1). Diese müssen im Ordner „src/test/resources" innerhalb des Projekts abgelegt sein. Zudem ist bei der Einrichtung des „jbehave-maven"-Plug-ins auf den korrekten Scope, in diesem Fall „test", zu achten (siehe Listing 4). Der Einkaufsliste können neue Einträge hinzugefügt sowie vorhandene bearbeitet oder entfernt werden.

Die Story-Datei definiert also in natürlicher Sprache, wie sich die Einkaufsliste verhalten soll. Diese Spezifikation kann durch den Anwender geschrieben oder zumindest überprüft werden. Ideal wäre es, wenn sie außerdem automatisiert ausgeführt würde. Dazu fehlt nur die Logik, mit der die Story-Datei interpretiert und das passende Szenario getestet werden kann. Dafür kann ein Grundgerüst für JUnit-Tests mithilfe des „maven-jbehave"-Plug-ins generiert werden, indem „mvn test" ausgeführt wird. Diese Vorlage wird auf der Konsole ausgegeben und kann übernommen werden.

```
Eintrag anlegen
GIVEN an empty shopping list
WHEN I add "Buy milk" to the list
THEN the list contains 1 item with the
description "Buy milk"
```

Listing 1: Beispiel einer Story-Datei

```
@Given("an empty shopping list")
@Pending
public void givenAnEmptyShoppingList() {
// PENDING
}
```

Listing 2: Aus der Story-Datei generierter JUnit-Testcode

```
@Given("an empty shopping list")
public void givenAnEmptyShoppingList() {
this.shoppingList = new ShoppingList();
}
```

Listing 3: Implementierter JUnit-Testcode

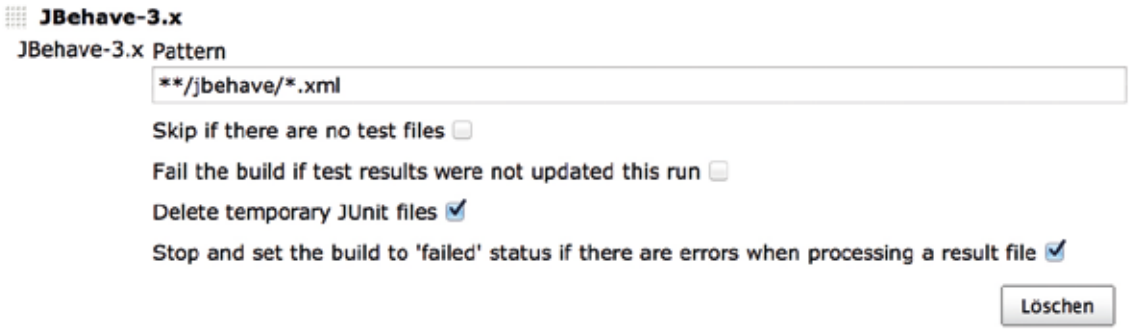


Abbildung 1: Integration von JBehave im CI-Server Jenkins

Es erfolgt jedoch keine Generierung eigenständiger Testdateien. JBehave nutzt für die bereits bekannten Schlüsselwörter „GIVEN", „WHEN" und „THEN" jeweils Annotationen innerhalb der Tests. Der Benutzer muss die so generierten Testmethoden (siehe Listing 2) abschließend nur noch um die eigentliche Testlogik erweitern (siehe Listing 3).

Wird nun ein weiteres Mal der Build-Prozess durchlaufen, erfolgt ein Abgleich von Story-Datei und Test, um festzustellen, ob die Story-Datei geändert wurde und daher eine neue Generierung notwendig ist. Damit dies erfolgen kann, ist eine weitere Java-Datei erforderlich, die dieses Mapping definiert. Die entsprechende Klasse muss von „org.jbehave.JunitStories" abgeleitet sein und die Methoden „configuration()", „stepsFactory()" und „storyPaths()" implementieren.

Ist der Abgleich erfolgreich, wird der Test automatisch ausgeführt. Falls der Abgleich fehlschlägt, setzt JBehave den Status des Tests auf „Pending", sodass er nicht ausgeführt wird. Der Benutzer kann konfigurieren, ob dieser Status den Build fehlschlagen lässt oder nicht. Jede Änderung einer Story-Datei führt somit dazu, dass die geschriebenen JUnit-Tests ebenfalls anzupassen sind. Auf den ersten Blick erscheint dieser Umstand als zusätzlicher Aufwand; er hilft dem Entwickler jedoch, da ein geändertes Verhalten zwangsläufig auch zu einer Änderung des bisherigen Tests führen muss.

Internationalisierung von Tests

Bisher sind in den Beispielen alle Stories in Englisch verfasst worden. Häufig ist es jedoch sinnvoll, die Beschreibung der Tests in der Muttersprache der Anwender zu schreiben. JBehave bietet eine solche Möglichkeit für verschiedene Sprachen an. Dabei werden die in den Story-Dateien verwendeten Schlüsselwörter übersetzt, die in den Unit-Tests verwendeten Annotationen jedoch nicht.

Ist die gewünschte Sprache nicht verfügbar, lässt sich JBehave um diese erweitern [3]. Beim Versuch, eine nicht unterstützte Sprache zu verwenden, wird die Systemsprache genutzt.

Continuous Integration mit JBehave

Der geschriebene JUnit-Test lässt sich problemlos in bestehende Build-Prozesse integrieren. JBehave bietet dafür Ant Tasks [4] beziehungsweise ein Maven-Plug-in [5] an, das in die „build.xml" beziehungsweise in die „pom.xml" integriert werden kann. In diesem Beispiel wird Maven und somit das „jbehave-maven"-Plug-in eingesetzt. In der Konfiguration des Plug-ins wird definiert, welche Klassen JBehave-Tests enthalten. JBehave führt nun die Tests mit der Endung „Stories.java" im Build-Prozess aus (siehe Listing 4).

Der CI-Server Jenkins [6] bietet sich an, um die Applikation mithilfe der definierten Tests kontinuierlich zu überprüfen. Er verfügt jedoch standardmäßig über keinerlei Unterstützung für das Anzeigen der Test-Ergebnisse von BDD-Frameworks. Dies kann mit Plug-ins nachgerüstet werden. In diesem Fall werden die Plug-ins „xUnit" [7] und „Jbehave" [8] benötigt.

Sobald beide Plug-ins installiert sind, können die Testresultate über einen separaten Schritt des Jenkins-Jobs gesammelt werden (siehe Abbildung 1). Dabei sind einige Anpassungsmöglichkeiten vorhanden. Anhand des Patterns sucht das Plug-in die Test-Resultate im Workspace zur spä-

```
<plugin>
<groupId>org.jbehave</groupId>
<artifactId>jbehave-maven-plugin</artifactId>
<version>${jbehave.version}</version>
<executions>
<execution>
<id>run-stories-as-embeddables</id>
<phase>integration-test</phase>
<configuration>
<!-- If the story file is placed in src/test/resources, the test scope has to be used -->
<scope>test</scope>
<includes>
<include>**/*Stories.java</include>
</includes>
<ignoreFailureInStories>true</ignoreFailureInStories>
<ignoreFailureInView>false</ignoreFailureInView>
</configuration>
<goals>
<goal>run-stories-as-embeddables</goal>
</goals>
</execution>
</executions>
</plugin>
```

Listing 4: Integration in Maven

Alle Tests

Testname	Dauer	Status
Adding numbers {a=-1, b=-1, result=-2}	0 ms	Erfolg
Adding numbers {a=-1, b=-3, result=-4}	0 ms	Erfolg
Adding numbers {a=-1, b=1, result=0}	0 ms	Erfolg
Adding numbers {a=1, b=2, result=3}	0 ms	Erfolg

Abbildung 2: Darstellung der Test-Ergebnisse in Jenkins

teren Anzeige. Mittels Checkboxen kann der Job zusätzlich so konfiguriert sein, dass ein Build fehlschlägt, falls keine neuen Ergebnisdaten vorhanden sind oder die Test-Resultate nicht ausgewertet werden konnten.

Bei der Konfiguration ist zu beachten, dass das JBehave-Plug-in die Testergebnisse des Maven-Builds im XML-Format benötigt, um sie auswerten zu können. Zudem muss das Pattern des JBehave-Plug-ins so konfiguriert sein, dass die erzeugten XML-Dateien gefunden werden.

Die Anzeige der Test-Ergebnisse erfolgt auf der Standard-Jenkins-Seite für Test-Ergebnisse (siehe Abbildung 2). Eine weitere Einschränkung besteht darin, dass nur die Job-Typen „free-style", „multi-configuration" und „monitor an external job" unterstützt werden. Wird ein Maven-2- oder -3-Job verwendet, werden die Test-Ergebnisse nicht angezeigt, sofern Maven „Build"-, „Surefire"- oder „Failsafe"-Reports erzeugt.

Ein Jenkins „Freestyle"-Projekt, konfiguriert mit den Plug-ins „xUnit" und „JBehave", ermöglicht die Integration von BDD in einen bestehenden Continuous-Integration-Prozess. Das Beispielprojekt umfasst eine Konfigurationsvorlage für diese Jenkins-Jobs („jenkins-config.xml"). Insgesamt gestaltet sich die Einbindung, bedingt durch die Beschränkung auf Freestyle-Projekte, sehr umständlich. Zudem ist das JBehave-Plug-in nur über die Produkt-Webseite des Herstellers und nicht im Update-Center von Jenkins zu finden.

Fazit

Behavior Driven Development erleichtert die Zusammenarbeit der Projektbeteiligten, indem eine allgemein verständliche Sprache verwendet wird. JBehave bietet dabei eine umfassende Unterstützung beim Schreiben und Ausführen von Unit-Tests.

Die Gliederung in die Testmethoden „GIVEN", „WHEN" und „THEN" bietet eine klare Strukturierung von Vorbedingung, Aktion und Nachbedingung. Das JBehave-Maven-Plug-in integriert die Tests im Rahmen eines Build-Prozesses. Die Anzeige der Test-Ergebnisse kann dabei in verschiedenen Formaten wie HTML, XML oder einfachem Text gewählt werden, sodass für jedes Test-Szenario das passende Format bereitsteht.

Erfolgt der Test innerhalb von Eclipse mittels JUnit, wird nicht deutlich, welche Test-Methoden erfolgreich beziehungsweise fehlerhaft waren. Die Anzeige kann jedoch durch „jbehave-junit-runner" sinnvoll erweitert werden, sodass jede einzelne Methode innerhalb des Test-Ergebnisses angezeigt wird [9]. Eine Ebene weiter können die Tests sogar kontinuierlich mithilfe von Plug-ins und CI-Server Jenkins ausgeführt werden.

In vielen Details zeigt sich diese Integration allerdings etwas problematisch, denn hier wird nur Jenkins unterstützt und keine Maven-2- beziehungsweise -3-Projekte, die vermutlich die meisten Benutzer verwenden werden.

Zur Internationalisierung bietet JBehave Möglichkeiten, sodass Stories und Tests in der Muttersprache der zusammenarbeitenden Fachbereiche geschrieben werden können. Bis auf die Annotationen in den JUnit-Tests lässt sich dabei jeder Bereich anpassen. Alle bislang nicht von Haus aus unterstützten Sprachen können mit einem eigenen Verfahren von Hand in JBehave integriert werden. Der vollständige Quellcode zu allen verwendeten Beispielen sowie eine Vorlage zur Integration in den CI-Server Jenkins sind auf GitHub zu finden [10].

Weitere Informationen

[1] http://dannorth.net/introducing-bdd
[2] A New Look at Test-Driven Development, David Astels, 2006: http://techblog.daveastels.com/files/BDD_Intro.pdf
[3] http://jbehave.org/reference/stable/stories-in-your-language.html
[4] http://jbehave.org/reference/stable/ant-tasks.html
[5] http://jbehave.org/reference/stable/maven-goals.html
[6] http://jenkins-ci.org
[7] https://wiki.jenkins-ci.org/display/JENKINS/xUnit+Plugin
[8] http://jbehave.org/reference/stable/hudson-plugin.html
[9] https://github.com/codecentric/jbehave-junit-runner
[10] https://github.com/matthiasbalke/artikel-bdd-2013

Matthias Balke
matthias.balke@adesso.de

Matthias Balke (Dipl. Inf., Univ.) ist als Software-Engineer bei der adesso AG in Dortmund tätig und arbeitet dort als Java-Entwickler mit Schwerpunkt auf Web-Anwendungen.

Sebastian Laag
sebastian.laag@adesso.de

Sebastian Laag (Dipl. Inf., Univ.) ist als Software-Engineer bei der adesso AG in Dortmund tätig und arbeitet dort als Java-Entwickler mit Schwerpunkt auf Web-Anwendungen.

Heute mal extra faul – Hibernate und Extra-Lazy Initialization

Martin Dilger, effectivetrainings & consulting

Bei der Arbeit mit JPA und Hibernate gibt es einige Probleme, die echte Klassiker sind. Der Versuch, dafür beim Entwickler am Schreibtisch gegenüber oder in der Kaffeeküche eine Lösung zu finden, endet bestenfalls mit einem verlegenen Schulterzucken. Im schlimmsten Fall bekommt man eine alternative Lösung präsentiert, umgangssprachlich als Workaround gefürchtet. Ein Klassiker dieser Problem-Gattung ist, die Größe einer lazy-initialisierten Collection zu erfragen, ohne diese vollständig zu initialisieren.

Ein konkretes Beispiel für dieses Problem: Nehmen wir einen Blog. Dieser besitzt Einträge und Kommentare. Eine einfache Entität für einen Blog-Eintrag besteht aus einem Titel, einer Liste von Kommentaren sowie dem Datum, an dem der Eintrag erstellt wurde (siehe Listing 1). Die Entität für Kommentare wurde der Übersichtlichkeit halber ausgelassen.

Nehmen wir an, dass dieser Blog sehr stark besucht ist und viele Besucher Kommentare hinterlassen. Pro Eintrag soll jedoch zu Beginn nur die Anzahl der vorhandenen Kommentare angezeigt werden.

Die eigentlichen Kommentare sollen erst dann erscheinen, wenn ein Benutzer diese explizit auswählt. Es kommt also relativ selten vor, dass Kommentare geladen werden, aber ständig, dass die Anzahl der Kommentare gebraucht wird. Ist es nun sinnvoll, nur um die Größe einer Collection zu erfragen, die ganze Liste von Kommentaren zu initialisieren?

Setup mit JPA/Hibernate/Spring

Eine einfache Möglichkeit für erste Experimente ist ein Setup mit Hibernate, JPA und Spring. Der Artikel geht nicht auf die Einzelheiten dieses Setups ein, denn darüber wurde bereits viel geschrieben. Der Sourcecode ist außerdem unter „https://github.com/dilgerma/hibernate-extra-lazy-initialization-example“ zu finden. Neben Spring arbeitet der Autor auch mit der hervorragenden Spring-Data-Bibliothek. Mithilfe von Spring-Data ist das Repository für Blog-Einträge aus Listing 2 definiert. Wem Spring-Data noch nicht geläufig ist, sollte das schleunigst nachholen. Mehr als dieses Interface muss im Code nicht definiert sein, um Blog-Einträge zu laden, zu speichern, zu löschen und zu ändern.

Echte Softwareentwickler brauchen keine Benutzeroberfläche, um Software zu entwickeln. Und genau aus diesem Grund schreiben wir zuerst einen Unit-Test, um die Konfiguration und die Funktionalität des Repository sicherzustellen (siehe Listing 3). Zusätzlich lassen wir uns direkt das von Hibernate generierte SQL mit ausgeben, um zu verstehen, was passiert (siehe Listing 4). Für das Speichern des Blog-Eintrags generiert Hibernate das SQL (siehe Listing 5).

Bis jetzt war es einfach

Soweit sieht alles sehr einfach aus und funktioniert. Versuchen wir allerdings jetzt im Test die Anzahl der vorhandenen Kommentare zu ermitteln, indem wir den Test um das Assert-Statement „assertEquals(new Integer(2), post.getCommentsCount());“ ergänzen, tritt das erste Problem auf. Es funktioniert nicht, da Hibernate die Collection nicht standardmäßig direkt beim Laden der Entität initialisiert (Lazy Initialization). Die Kommentare werden nur geladen, wenn auch wirklich auf sie zugegriffen wird. Das Nachladen einer derart initialisierten Collection funktioniert aber leider nur im Kontext einer Transaktion und nicht mit einer Entität, die keinem „PersistenceContext“

mehr zugeordnet ist („detached state"). Der Test spiegelt das auch sofort wider, denn eine Transaktion ist nur innerhalb der Methoden im „PostRepository" aktiv. Das Transaktionshandling wird automatisch von Spring-Data übernommen. Im Test selbst haben wir keine Möglichkeit mehr, auf nicht initialisierte Daten zuzugreifen (siehe Listing 6).

Eager Initialization

Wir können Hibernate anweisen, die Collection direkt zu initialisieren. Immer wenn eine Post-Entity geladen wird, wird Hibernate auch gleich alle Kommentare laden (siehe Listing 7). Das von Hibernate generierte SQL sieht in diesem Fall wie in Listing 8 aus.

Aus Performance-Gründen kann das nicht erwünscht sein, da wir für die meisten Blog-Einträge völlig umsonst alle Kommentare laden würden. Jetzt haben wir ein Dilemma, aus dem es scheinbar keinen Ausweg gibt. Doch Lazy Initialization ist gut, Extra-Lazy Initialization (manchmal) besser.

Extra-Lazy Initialization

Hibernate bietet ein Feature, das sich „Extra-Lazy Initialization" nennt. Hier initialisiert Hibernate eine Collection teilweise, jedoch nur die Elemente, die tatsächlich gebraucht werden, das können sogar einzelne Elemente in der Liste sein. Wir ändern hierfür das Collection-Mapping für die Kommentare (siehe Listing 9).

Die Annotation „@LazyCollection" stammt nicht aus der JPA-Spezifikation, sondern ist Hibernate-spezifisch. Führen wir den Test erneut aus, sehen wir aber leider, dass dieser nach wie vor mit der „LazyInitializationException" fehlschlägt. Wie bereits vorhin erwähnt, initialisiert Hibernate intelligent. Es wird nur das initialisiert, worauf auch tatsächlich zugegriffen wird. Wir ergänzen deshalb die Post-Entität um eine Methode, die nach jedem Laden ausgeführt wird (siehe Listing 10).

Jedes Mal, wenn ein Post geladen wird, fragen wir die Größe der Kommentar-Collection ab – und nur diese. Listing 11 zeigt das SQL, das Hibernate jetzt anschließend generiert.

Hibernate erkennt, dass nicht die Collection selbst, sondern nur deren Größe initialisiert wird. Deswegen wird auch kein „select" auf alle Kommentare, sondern nur ein „select(count)" ausgeführt, was aus Performance-Sicht optimal ist. Das lässt sich auch beweisen, indem wir einfach mit

```
@Entity
@Table(name="POST")
public class Post {

@Id
@GeneratedValue(strategy=GenerationType.AUTO)
@Column(name="POST_ID")
private Integer postId;

@Column(name="TITLE")
private String title;

@Column(name="POST_DATE")
private Date postDate;

@OneToMany(cascade = CascadeType.ALL)
@JoinColumn(name="POST_ID",referencedColumnName = "POST_ID")
private List<Comment> comments = new ArrayList<Comment>();

public Integer getPostId() {
return postId;
}

public void setPostId(Integer postId) {
this.postId = postId;
}

public String getTitle() {
return title;
}

public void setTitle(String title) {
this.title = title;
}

public Date getPostDate() {
return postDate;
}

public List getComments(){
return comments;
}

public void addComment(Comment comment){
this.comments.add(comment);
}

public int getCommentsCount(){
return comments.size();
}

public boolean hasComments(){
return !comments.isEmpty();
}

@PrePersist
public void setPostDate() {
this.postDate = new Date();
}
}
```

Listing 1

```
public interface PostRepository extends
JpaRepository<Post, Integer> { }
```

Listing 2

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations="classpath:META-INF/test-context.xml")
public class PostRepositoryTest {

@Autowired
PostRepository repository;

@Test
public void test() {
Post post = new Post();
post.setTitle("First Post");
//adding some comments
Comment firstComment = new Comment("Kommentar Text1","Martin");
post.addComment(firstComment);
Comment secondComment = new Comment("Kommentar Text2","Markus");
post.addComment(secondComment);

repository.save(post);

Post dbpost = repository.findOne(post.getPostId());
assertNotNull(dbpost);
}
}
```

Listing 3

```
Hibernate: create table POST (POST_ID integer generated by default as identity, POST_DATE timestamp, TITLE varchar(255), primary key (POST_ID))
Hibernate: create table COMMENTS (ID bigint generated by default as identity, AUTHOR varchar(255), COMMENT varchar(255), POST_ID integer,
primary key (id))
Hibernate: alter table COMMENTS add constraint FKDC17DDF4D3BC1BD8 foreign key (POST_ID) references POST
```

Listing 4

```
Hibernate: insert into POST (POST_ID, POST_DATE, TITLE) values (null, ?, ?)
Hibernate: insert into comments (id, author, comment) values (null, ?, ?)
Hibernate: insert into comments (id, author, comment) values (null, ?, ?)
Hibernate: update comments set POST_ID=? where id=?
Hibernate: update comments set POST_ID=? where id=?
```

Listing 5

```
org.hibernate.LazyInitializationException: failed to lazily initialize a collection of role: de.effectivetrainings.entities.Post.comments, could not initialize
proxy - no Session
```

Listing 6

```
@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
@JoinColumn(name="POST_ID",referencedColumnName = "POST_ID")
private List<Comment> comments = new ArrayList<Comment>();
```

Listing 7

```
Hibernate: select post0_.POST_ID as POST1_0_0_, post0_.POST_DATE as POST2_0_0_, post0_.TITLE as TITLE0_0_ from POST post0_ where
post0_.POST_ID=?
Hibernate: select comments0_.POST_ID as POST4_0_1_, comments0_.id as id1_1_, comments0_.id as id1_0_, comments0_.author as author1_0_,
comments0_.comment as comment1_0_ from comments comments0_ where comments0_.POST_ID=?
```

Listing 8

```
@OneToMany(cascade = CascadeType.ALL)
@JoinColumn(name="POST_ID",referencedColumnName = "POST_ID")
@LazyCollection(LazyCollectionOption.EXTRA)
private List<Comment> comments = new ArrayList<Comment>();
```

Listing 9

```
@PostLoad
 private void initCommentCount(){
   comments.size();
 }
```

Listing 10

```
Hibernate: select post0_.POST_ID as POST1_1_0_, post0_.POST_DATE as POST2_1_0_,
post0_.TITLE as TITLE1_0_ from POST post0_ where post0_.POST_ID=?

Hibernate: select count(id) from comments where POST_ID =?
```

Listing 11

„dbpost.getComments().get(0);“ auf ein Element in der Collection zugreifen. Beim Zugriff auf das erste Element in der Liste wird nach wie vor die „LazyInitializationException“ geworfen, da wir nichts weiter als die Größe initialisiert haben.

Fazit

Wer das nächste Mal in der Kaffeeküche darauf angesprochen wird, wie schwierig es ist, die Größe einer Collection abzufragen, die lazy-initialisiert wird, kann auf das Schulterzucken verzichten und ein Siegerlächeln aufblitzen lassen. Diese Lösung ist elegant und praxistauglich, sicherlich nicht für jede Situation geeignet, doch der Autor hat bereits einige Male von dieser Möglichkeit profitiert.

Martin Dilger
martin@effectivetrainings.de

Martin Dilger ist freiberuflicher Software-Entwickler, Consultant und Trainer aus Leidenschaft. Er beschäftigt sich mit der Entwicklung von Enterprise-Java-Anwendungen, agiler Software-Entwicklung und der pragmatischen Arbeit mit GIT.

Programmieren lernen mit Java

Gelesen von Jürgen Thierack

Titel: Programmieren lernen mit Java

Autor: Hans-Peter Habelitz

Verlag: Galileo Computing

Umfang: 512 Seiten

Preis: 19,90 Euro inkl. DVD

ISBN: 978-3-8362-1788-0

Das Buch von Dipl.-Ing. Hans-Peter Habelitz ist für Einsteiger in die Programmierung überhaupt gedacht. Über den Autor erfahren wir, dass er an einer Berufsschule Informatik lehrt. „Keine Vorkenntnisse erforderlich“ wird auf dem Cover zugesichert. Die Einführung zu Kapitel 1 wiederholt das Versprechen: „Besonderer Wert wurde bei der Erstellung darauf gelegt, dass keine Programmierkenntnisse vorausgesetzt werden, sodass Sie das Programmieren von Grund auf lernen können.“

Zunächst werden Java und Eclipse vorgestellt und installiert. Die DVD unterstützt dabei Linux, Mac OS X und Windows. In der heutigen Zeit kommen buchbegleitende CDs/DVDs immer mehr aus der Mode. Hier sieht man auch warum: Weder Eclipse noch der Java-7-SDK sind auf dem neuesten Stand. Bei einem so grundlegenden Werk schadet das aber nichts. Ebenfalls auf der DVD: die Beispiele und Übungen.

Ganz systematisch geht es von den Grundlagen, Variablen, Operatoren, Kontroll-Strukturen, Ein- und Ausgaben, grafischen Benutzeroberflächen (Swing) zu Objekten, Ereignissen und Datenbanken. Die dem Buch eigene Ausführlichkeit macht den Weg zwar länger, die Steigung aber flacher. Fazit: Für den absoluten Neueinsteiger gut geeignet.

Jürgen Thierack
thierack@igfm-muenchen.de

OAuth 2.0 – ein Standard wird erwachsen

Uwe Friedrichsen, codecentric AG

OAuth ist ein weitverbreiteter Standard im Web-Umfeld, mit dem ein Nutzer Dritt-Anwendungen den Zugriff auf eigene geschützte Web-Ressourcen erlauben kann, ohne dass Passwörter oder andere sensible Credentials weitergegeben werden müssen.

Nach zähem Ringen und vielen schwierigen Diskussionen über zweieinhalb Jahre hinweg ist OAuth 2.0 im Oktober 2012 freigegeben worden. Die Version 2.0 ist ein vollständig neuer Standard, nicht abwärtskompatibel zu 1.0, der den Anspruch hat, die Stärken der Version 1.0 zu übernehmen und dabei die bekannten Schwächen zu beheben.

Obwohl OAuth 2.0 im Grunde recht einfach ist, wenn man sich erst einmal an ein paar Security-typische Eigenheiten gewöhnt hat, würde jeder Versuch, OAuth 2.0 vollumfänglich zu beschreiben, unweigerlich den Rahmen sprengen. Deshalb beschränkt sich dieser Artikel auf einen Überblick über OAuth 2.0 in Form eines etwas ausführlicheren Steckbriefs.

OAuth 2.0 kurzgefasst

In einem Satz ist OAuth ein Standard, der Anwendungen ermöglicht, im Namen eines Nutzers auf geschützte Ressourcen zuzugreifen, ohne dass der Anwendung die Credentials, mit denen die Ressourcen geschützt sind, bekannt gegeben werden müssen. Da dieser Satz zwar inhaltlich korrekt ist, man ihn aber mehrere Male lesen muss, um ihn richtig zu verstehen, zeigt Abbildung 1 eine Erläuterung als Bild. Wie dort zu sehen, gibt es auf logischer Ebene drei beteiligte Parteien:

- *Resource Owner*
  In der Regel eine natürliche Person, der die geschützten Ressourcen gehören
- *Resource Server*
  Server, auf dem die geschützten Ressourcen gespeichert sind
- *Anwendung*
  Anwendung, die im Namen des Resource Owner auf die geschützten Ressourcen zugreifen will

Das typische Musterbeispiel dafür ist ein Foto-Server, auf dem man Bilder gespeichert hat, als „Resource Server" und ein Fotobuch-Service als „Anwendung", der man Zugriff auf seine Bilder gewähren will. Das Problem ist jetzt, dass man seine Credentials (wie Username und Passwort), mit denen man seine Bilder auf dem Foto-Server geschützt hat, dem Fotobuch-Service nicht bekannt geben will.

Genau dieses Problem löst OAuth. Man kann damit dem Foto-Service einen sicheren Zugriff auf seine Fotos ermöglichen, ohne dass die Credentials den Foto-Server verlassen müssen. Prinzipiell im Web- und Browser-Umfeld angesiedelt, kann OAuth 2.0 diese Aufgabe auch in anderen Umfeldern, etwa innerhalb von Unternehmen erfüllen. Die einzige Voraussetzung ist die Verfügbarkeit von HTTP bei allen beteiligten Parteien, da OAuth auf HTTP aufsetzt.

Die Entstehung von OAuth 2.0

Im Jahr 2006 hat Twitter ein Protokoll entwickelt, mit dem Dritt-Anwendungen im Namen von Twitter-Nutzern auf deren Tweets und einige weitere Informationen zugreifen konnten, ohne dass man Passwörter weitergeben musste. Andere soziale Plattformen wie Facebook hatten einen ähnlichen Bedarf und so hat man sich zusammengetan und binnen eines Jahres „OAuth 1.0" spezifiziert. Erst als reiner Industrie-Standard wurde OAuth 1.0 mit dem RFC 5849 [1] im Frühjahr 2010 zum offiziellen IETF-Standard.

Trotz des Erfolgs von OAuth 1.0 gab es aber auch einige kritische Stimmen. So hatten viele Client-Entwickler Probleme mit dem komplizierten Security-Handling auf der Client-Seite. Außerdem störten die Beschränktheit auf genau den einen Use-Case und das Browser/Web-Umfeld sowie die fehlende Erweiterbarkeit. Gerade Un-

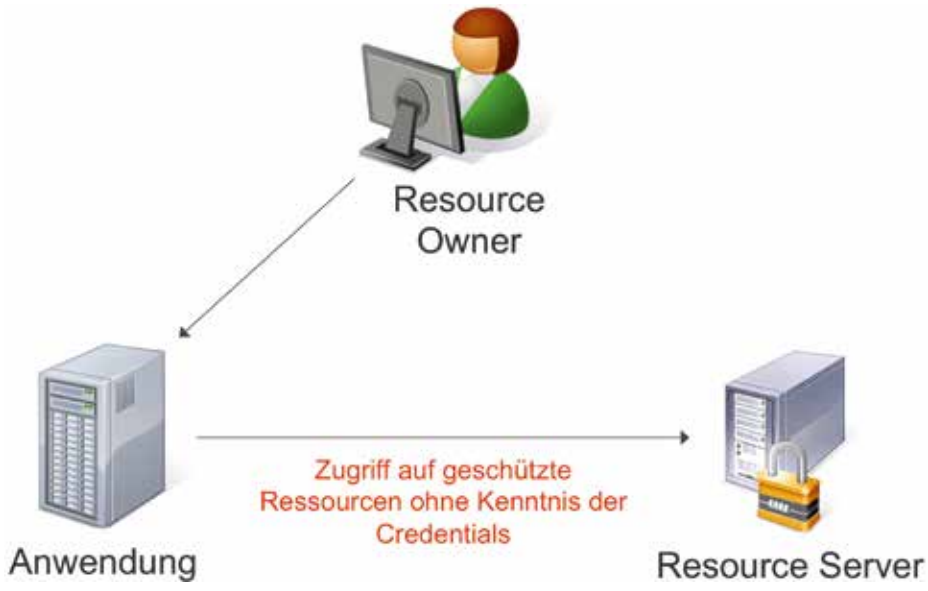


Abbildung 1: Das Problem, das OAuth löst

ternehmen, die OAuth gerne für internen Zwecke nutzen wollten, bemängelten die fehlende „Enterprise-Tauglichkeit".

Diese Kritikpunkte führten dazu, dass sich im Frühjahr 2010 eine neue Arbeitsgruppe zur Spezifikation von OAuth 2.0 formierte. Sie wollte die Stärken von OAuth 1.0 in Version 2.0 übernehmen, jedoch auch Antworten auf die genannten Kritikpunkte liefern.

Die Arbeiten an dem neuen Standard erwiesen sich als sehr zäh und schwierig, weil sehr viele, sehr unterschiedliche Interessen aufeinanderprallten. Die Web-Fraktion wollte die Leichtgewichtigkeit und Sicherheit von OAuth 1.0 erhalten, während die Enterprise-Fraktion an maximaler Flexibilität und Erweiterbarkeit interessiert war.

Nach 31 Draft-Versionen mit viel Gerangel und viel bösem Blut bis hin zum Ausstieg zentraler Personen [2] wurde OAuth 2.0 im Oktober 2012 endlich offiziell als IETF RFC 6749 [3] freigegeben.

Wie OAuth 2.0 funktioniert

OAuth ist im Kern ein Protokoll, das die Kommunikation zwischen den beteiligten Parteien definiert. OAuth 2.0 definiert mehrere solcher Protokollflüsse. Wir werden hier aus Platzgründen nur einen dieser Flüsse, den sogenannten „Authorization Code Grant", näher betrachten. OAuth 2.0 unterscheidet vier Parteien, indem es den zuvor beschriebenen Resource Server in zwei Bereiche aufteilt:

- *Authorization Server*
  Zuständig für Authentifizierung und Autorisierung sowie Token-Vergabe
- *Resource Server*
  Speicherort der geschützten Ressourcen

Die Dritt-Anwendung heißt im OAuth-Jargon „Client". Jeder Client, der Zugriff auf die Ressourcen haben möchte, muss einmalig beim Authorization Server registriert werden. OAuth 2.0 legt dabei nicht fest, wie die Registrierung zu erfolgen hat. Es werden zwei Client-Typen unterschieden:

- *Confidential Clients*
  Können ein bei der Registrierung erhaltenes Secret sicher speichern
- *Public Clients*
  Können das nicht

In die erste Kategorie fallen zum Beispiel Web-Server, die in einer geschützten Umgebung betrieben werden, in die zweite etwa JavaScript-Clients, die in einem Browser laufen. Der Authorization Code Grant ist für Confidential Clients konzipiert worden. Deshalb werden wir im Folgenden diesen Client-Typ annehmen.

Bei der Registrierung beim Authorization Server gibt der Client optional einen Redirection-Endpoint an (dazu später mehr). Als Ergebnis der Registrierung erhält der Client eine „Client Id" sowie ein „Client Secret" oder ein alternatives Credential, denn OAuth 2.0 erlaubt verschiedene Credential-Arten.

Ist der Client registriert, kann er ein „Access Token" anfordern. Das geschieht in zwei Teilschritten. Im ersten (siehe Abbildung 2) erwirbt er einen „Authorization Code" – daher der Name des Flusses. Dafür übergibt der Client beispielsweise per Redirect oder per Link, den der Anwender anklicken muss, die Kontrolle an den Authorization Server (Schritte 1 und 2). Listing 1 zeigt die Protokoll-Ebene.

```
GET /authorize?
        response_type=code&
        client_id=s6BhdRkqt3&
        state=xyz&
        scope=abc&
        redirect_uri=https%3A%2F%2Fclient%
2Eexample%2Ecom%2Fcb HTTP/1.1
Host: server.example.com
```

Listing 1

Der Aufruf wird an den Authorization Endpoint („/authorize") geschickt. Als zusätzliche Query-Parameter gibt er den gewünschten Grant Type („code" – Authorization Code Grant), seine Client-Id, optional außerdem seinen Zustand („state"), den gewünschten Scope („scope") und seinen Redirection-URI („redirect_uri") mit.

Der Zustand wird vom Authorization Server am Ende des Teilschritts unverändert an den Client zurückgegeben (siehe Schritte 4 und 5). Der Client kann ihn mitgeben, um seinen Anwendungskontext wiederherzustellen, wenn er die Kontrolle zurückerhält.

Mit dem Scope-Parameter kann der Client einen bestimmten Zugriffsumfang anfordern. Mögliche Scope-Ausprägungen werden vom Authorization Server definiert.

Der Redirection-URI ist die Adresse, an die der Authorization Server am Ende des Teilschritts die Kontrolle zurückgibt. Ist sie nicht angegeben, muss der Client sie bei der Registrierung hinterlegt haben. Ist sie angegeben und hinterlegt, muss der angegebene URI mit dem hinterlegten URI übereinstimmen oder eine Unter-Ressource des hinterlegten URI sein.

Der nächste Schritt (Schritt 3) wird durch OAuth 2.0 nur grob umrissen. Die Spezifikation fordert, dass der Authorization Server vom Resource Owner explizit die Zustimmung für den gewünschten Zugriff mit dem

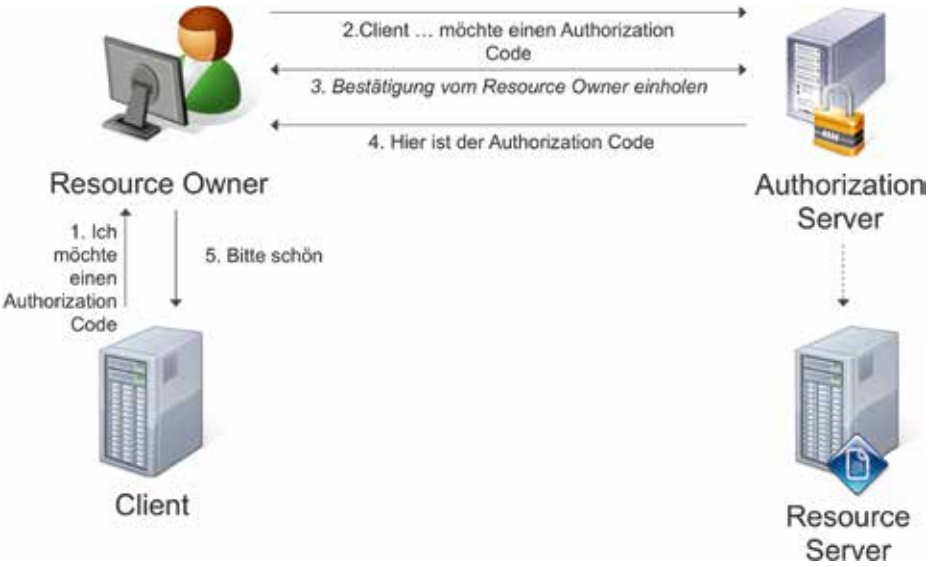


Abbildung 2: Authorization Code Grant, Anfordern eines Authorization Code

geforderten Scope einholt, lässt jedoch offen, wie das zu erfolgen hat. Diese Unschärfe bietet einerseits eine hohe Flexibilität, andererseits aber keine Unterstützung für eine konkrete Umsetzung – inklusive der Gefahr ungewollter Sicherheitslücken aufgrund fehlenden Security-Wissens.

Hat der Resource Owner der Zugriffsanfrage zugestimmt, schickt der Authorization Server seine Antwort via Browser-Redirect an den Redirection-Endpoint des Clients (Schritte 4 und 5). Listing 2 zeigt wieder die Protokoll-Ebene.

```
HTTP/1.1 302 Found
Location: https://client.example.com/cb?
        code=SplxlOBeZQQYbYS6WxSbIA&
        state=xyz
```

Listing 2

Als Werte werden dem Client der Authorization Code („code“) und der Zustand („state“) zurückgegeben, falls er diesen in Schritt 1 mitgegeben hat. OAuth 2.0 definiert auch das Format aller möglichen Fehlermeldungen, etwa für den Fall, dass der Resource Owner die Zugriffsanfrage ablehnt. Aus Platzgründen werden wir uns dies allerdings nicht näher ansehen.

Der Authorization Code Grant ist in zwei Teilschritte unterteilt, weil die Kommunikation im ersten Teilschritt über den potenziell unsicheren Browser des Resource Owner läuft. Aus diesem Grund soll der Authorization Code auch nur eine kurze Lebensdauer haben; die Spezifikation empfiehlt maximal zehn Minuten. Außerdem muss der Authorization Server sicherstellen, dass ein Authorization Code nicht mehr als einmal verwendet wird, und sollte im Missbrauchsfall alle Access Token sperren, die auf dem Authorization Code basieren.

Der zweite Teilschritt ist das Eintauschen des Authorization Code gegen das eigentliche Access Token (siehe Abbildung 3). Hier kommunizieren Client und Authorization Server direkt miteinander. Der Client stellt die Anfrage, seinen Authorization Code gegen ein Access Token einzutauschen (Schritt 6). Listing 3 zeigt wieder die Protokoll-Ebene.

Der Aufruf wird an den Token Endpoint („/token“) geschickt. Im Authorization Header authentifiziert sich der Client. Die

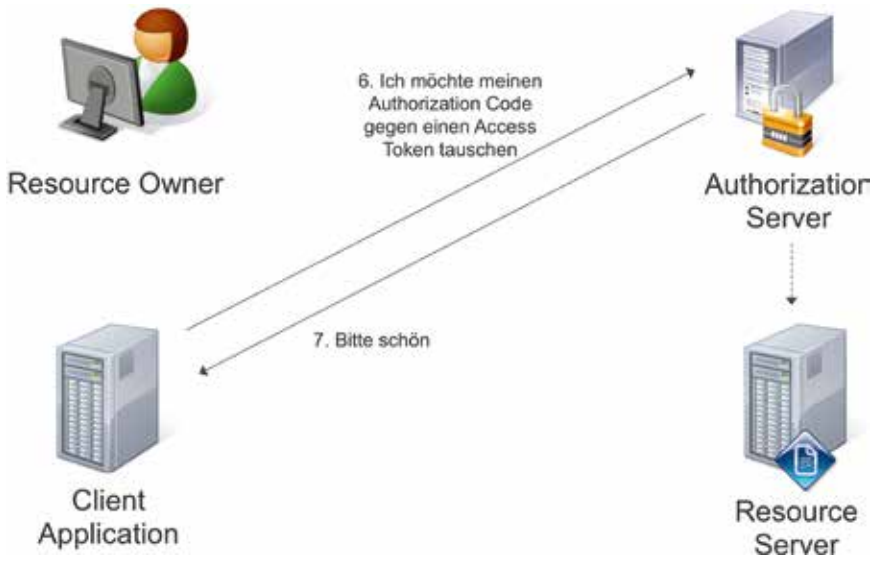


Abbildung 3: Authorization Code Grant, Tauschen des Authorization Code gegen ein Access Token

```
POST /token HTTP/1.1
Host: server.example.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&
        code=SplxlOBeZQQYbYS6WxSbIA&
        redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
```

Listing 3

Zeichenfolge hinter Basic sind die Client-Id und das Client-Secret, gemäß den Vorgaben für die Basic-Authentication durch Doppelpunkt getrennt hintereinander geschrieben und per „BASE64“ kodiert. OAuth 2.0 erlaubt auch andere Authentifizierungsverfahren, aber dieses Verfahren muss jeder Authorization Server anbieten.

Als weitere Parameter werden dem Authorization Server der Grant Type („authorization_code“ – Authorization Code Grant), der zuvor erhaltene Authorization Code („code“) und der Redirection-URI („redirect_uri“) mitgegeben, der mit dem Redirection-URI aus dem ersten Teilschritt übereinstimmen muss. OAuth 2.0 definiert auf recht detailreiche Weise, wann und wo man den Redirection-URI mitgeben muss und wann nicht, was wir hier aber nicht vertiefen werden. Ist der Authorization Code gültig und noch nicht verwendet worden, gibt der Authorization Server ein neues Access Token zurück (Schritt 7). Listing 4 zeigt wieder die Protokoll-Ebene.

Je nach Art des Access Tokens können unterschiedliche Informationen in dem JSON-Dokument übermittelt werden. Die Spezifikation schreibt vor, dass jeder Authorization Server mindestens Bearer-Token (“token_type”:”bearer“) unterstützen muss. OAuth erlaubt aber auch die Verwendung beliebiger anderer Token-Typen. Im Falle von Bearer Token erhält man das eigentliche Token („access_token“). Außerdem sollte noch eine Lebensdauer („expires_in“) mitgegeben werden, die aber nur informativen Charakter hat.

Darüber hinaus kann optional noch ein Refresh-Token („refresh_token“) mitgegeben werden, über das sich der Client nach Ablauf der Lebensdauer ein neues Access Token holen kann, ohne noch einmal den gesamten Fluss durchlaufen zu müssen.

Hatte der Client in Schritt 1 einen bestimmten Scope angefordert, der Resource Owner jedoch einen anderen Scope berechtigt, muss an der Stelle auch noch der tatsächlich berechtigte Scope mit zurückgegeben werden („scope“, im Listing nicht enthalten).

Das war der Authorization Code Grant im Schnelldurchlauf. Mit dem erhaltenen Access Token kann der Client jetzt auf die

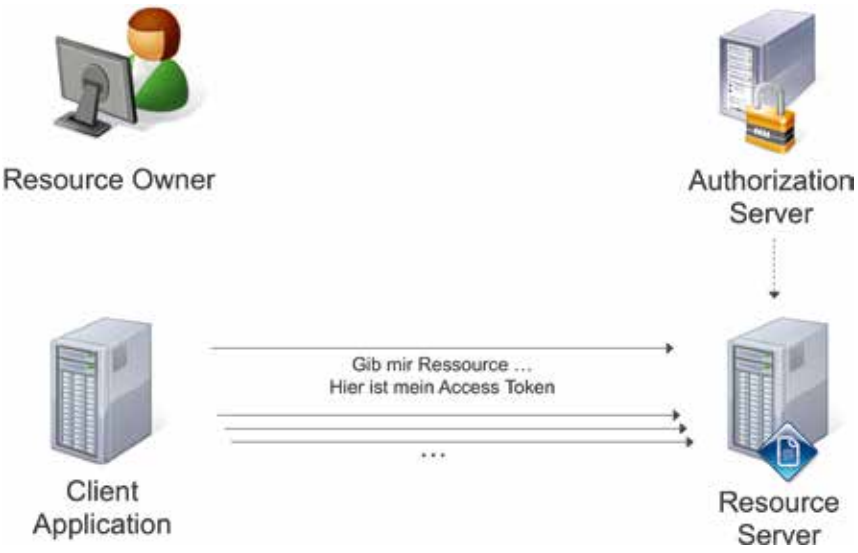


Abbildung 4: Zugriff auf geschützte Ressourcen mit einem Access Token

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
 "access_token":"2YotnFZFEjr1zCsicMWpAA",
 "token_type":"bearer",
 "expires_in":3600,
 "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA"
}
```

Listing 4

```
GET /resource/1 HTTP/1.1
Host: example.com
Authorization: Bearer 2YotnFZFEjr1zCsic-
MWpAA
```

Listing 5

geschützten Ressourcen auf dem Resource Server zugreifen (siehe Abbildung 4). Listing 5 zeigt wieder die Protokoll-Ebene.

Neben den normalen Parametern für den Zugriff auf die Ressource wird zusätzlich ein Authorization-Header mit dem Authentifizierungsschema „Bearer" und dem Access Token mitgegeben. Damit eine solche Anfrage und auch die zuvor gesehenen Anfragen und Antworten sicher sind, muss die Kommunikation verschlüsselt werden. OAuth 2.0 schreibt deshalb an vielen Stellen die Verwendung von TLS (SSL) vor.

Die Unterschiede zu OAuth 1.0

OAuth 1.0 ist wesentlich enger umrissen. OAuth 1.0 kennt genau zwei Flüsse, häufig „3-Legged-Flow" und „2-Legged-Flow" genannt. Ersterer entspricht inhaltlich dem Authentication Code Grant. Mit letzterem kann man einem Client unabhängig von einem Resource Owner Zugriff auf geschützte Ressourcen gewähren. OAuth 2.0 hat dafür den „Client Credentials Grant" als Gegenstück. Darüber hinaus definiert OAuth 2.0 aber noch zusätzliche Flüsse und erlaubt die Erweiterung um weitere Flüsse.

OAuth 1.0 ist auch ohne den Einsatz von TLS sicher, da es ein anderes Verfahren zum Erwerben und zur Nutzung eines Tokens verwendet. In einem Erweiterungsstandard zu OAuth 2.0 [4] werden HMAC-Token (HTTP Message Authentication Code) definiert, die einen Zugriff auf die geschützten Ressourcen ohne den Einsatz von TLS ermöglichen. Der Standard befindet sich allerdings noch im Draft-Zustand.

Wie aus dem HMAC-Token bereits ersichtlich, erlaubt OAuth 2.0 neben weiteren Flüssen auch die Definition weiterer Token-Typen sowie aller zusätzlich dafür benötigten Informationen (zusätzliche Parameter, Antwort-Typen und Fehlermeldungen). Damit ist OAuth 2.0 offen für alle möglichen Arten von Erweiterungen, während OAuth 1.0 genau auf seinen initialen Einsatzzweck zugeschnitten ist und keinerlei Erweiterungen darüber hinaus vorsieht. Letztlich weicht OAuth 2.0 gegenüber OAuth 1.0 einige Sicherheitsanforderungen zugunsten der Flexibilität und Erweiterbarkeit auf. Wo es in OAuth 1.0 ganz strikt „MUST" heißt, findet man bei OAuth 2.0 häufig einmal ein „SHALL" oder „MAY".

Was die Kritiker sagen

Gerade wegen der gegenüber OAuth 1.0 reduzierten Sicherheits-Anforderungen wird OAuth 2.0 teilweise heftig kritisiert. Dass es keine eingebaute Sicherheit hat, sondern sich – zumindest in der Minimalversion – vielfach ausschließlich auf TLS verlässt, wird von den Kritikern als großer Rückschritt gegenüber OAuth 1.0 angesehen. Auch die vielen Zugeständnisse zugunsten der größeren Flexibilität und Erweiterbarkeit stoßen bei den Kritikern auf wenig Gegenliebe. Und der Bearer-Token ist aus ihrer Sicht vollständig unsäglich.

Außerdem stört die Kritiker, dass der neue Standard nicht mehr so schön leichtgewichtig und fokussiert ist wie OAuth 1.0, sondern wesentlich schwammiger, von vielfältigen Kompromissen geprägt und deutlich schwergewichtiger. Diese Kritikpunkte erwecken den Eindruck, dass man besser bei OAuth 1.0 bleiben und die Finger von OAuth 2.0 lassen sollte. Hier gibt es jedoch Entwarnung. Man kann mit OAuth 2.0 zwar potenziell unsichere Systeme entwickeln, muss es aber nicht. Niemand zwingt einen, nur den Minimal-Standard einzusetzen und alles zu ignorieren, an dem nicht ein „MUST" steht.

Mit ein wenig Umsicht lassen sich mit OAuth 2.0 auf einfache Weise sichere Systeme entwickeln und spätestens mit dem Einsatz von HMAC-Token sollten alle Kritikpunkte entkräftet sein. Allerdings sind solide Security-Kenntnisse erforderlich, um die richtigen Entscheidungen treffen zu können.

Was noch fehlt

Dies war ein kurzer Überblick über OAuth 2.0. Aus Platzgründen musste eine ganze Reihe von Aspekten leider unbehandelt bleiben:

- Weitere Flüsse: Neben dem „Authorization Code Grant“ definiert OAuth 2.0 zusätzlich noch den „Implicit Grant“ (für Public Clients), den „Resource Owner Password Credentials Grant“ (mit User und Passwort, für gesicherte Umgebungen, führt die Grund-Idee von OAuth ein wenig ad absurdum), den „Client Credentials Grant“ sowie den „Refresh Token Grant“ (beide bereits zuvor erwähnt).
- Extensions: Wie bereits erwähnt, kann man OAuth 2.0 um weitere Flüsse, Token, Typen und die dafür zusätzlich benötigten Informationen erweitern. Es existiert auch bereits eine Reihe mehr oder minder standardisierter Erweiterungen.
- Alternative Implementierungen: An diversen Stellen schreibt OAuth 2.0 nur eine Minimal-Implementierung vor, die alle Implementierungen unterstützen müssen, erlaubt aber alternative (bessere) Umsetzungen.
- Darüber hinaus gibt es eine Menge interessanter Details in der Spezifikation, auf die wir hier nicht eingehen konnten.
- Auch eine ausführliche Sicherheitsbetrachtung fehlt: Wie sicher ist OAuth 2.0, wo liegen die Angriffspunkte und was kann man dagegen tun (die Spezifikation [4] liefert dazu einige Informationen).
- Zudem haben wir nicht betrachtet, wie sich OAuth-Client und -Server für einen Java-Entwickler anfühlen, inklusive der verfügbaren Frameworks.

OAuth 2.0 ist leider viel zu ergiebig, um das alles in einen Artikel packen zu können. Eventuell werden wir den einen oder anderen Aspekt in einem Folgeartikel beleuchten.

Fazit

OAuth ist ein Standard für den Zugriff auf geschützte Ressourcen, ohne dass zugehörige Credentials verbreitet werden müssten. Im Oktober 2012 ist Version 2.0 des Standards freigegeben worden. In dem Artikel haben wir uns die Funktionsweise von OAuth am Beispiel des wichtigsten Flusses, des Authorization Code Grant, im Detail angesehen.

OAuth 2.0 ist im Gegensatz zu Version 1.0 stark auf Flexibilität und Erweiterbarkeit ausgelegt und opfert der zusätzlichen Offenheit in der Minimalimplementierung einen Teil der Sicherheit von Version 1.0. Mit ein wenig Augenmaß und Security-Wissen kann man mit OAuth 2.0 aber ebenfalls auf einfache Weise sichere Systeme realisieren.

Weiterführende Links

[1] RFC 5849, The OAuth 1.0 Protocol: http://tools.ietf.org/html/rfc5849

[2] Eran Hammer-Lahav, OAuth 2.0 and the road to hell: http://hueniverse.com/2012/07/oauth-2-0-and-the-road-to-hell/

[3] RFC 6749, The OAuth 2.0 Authorization Framework: http://tools.ietf.org/html/rfc6749

[4] OAuth 2.0 Message Authentication Code (MAC) Tokens, Draft 03: http://tools.ietf.org/html/draft-ietf-oauth-v2-http-mac-03

Uwe Friedrichsen

uwe.friedrichsen@codecentric.de

Uwe Friedrichsen ist ein langjähriger Reisender in der IT-Welt. Als CTO der codecentric AG darf er seine Neugierde auf neue Ansätze und Konzepte sowie seine Lust am Andersdenken ausleben. Seine aktuellen Schwerpunktthemen sind verteilte, hochskalierbare Systeme und post-agile Architekturarbeit. Er teilt und diskutiert seine Ideen häufig auf Konferenzen, als Autor von Artikeln, Blog Posts, Tweets und mehr.

„Der JCP bestimmt die Evolution der Programmiersprache …“

Die beiden JCP-EC-Repräsentanten der Credit Suisse, Dr. Susanne Cech Previtali und Victor Grazi, erläutern im Interview mit dem Oracle Java Magazine ihre Arbeit im Java Community Process.

Dr. Susanne Cech Previtali ist im Infrastruktur-Architektur-Team der Credit Suisse verantwortliche Java-Architektin und damit auch zuständig für die Java Application Platform (JAP) der Credit Suisse und die damit verbundenen Konstruktions-Richtlinien für die Entwicklung von Enterprise Java-EE-Applikationen. Victor Grazi hat im September 2012 den jährlichen JCP-Award als Best Spec Lead für seine Arbeit am JSR 354, Money and Currency API, erhalten. Er ist Java Champion und hat die Credit Suisse im Dezember 2012 für eine neue Herausforderung verlassen. Der neue „Alternate“ ist seit Februar 2013 Scot Baldry, Investment-Banking-Chefarchitekt.

Credit Suisse wurde erneut in das Executive Committee des JCP gewählt. Ihr Positionierungs-Statement ist online verfügbar (siehe http://www.jcp.org/aboutJava/communityprocess/elections/2012/Credit_Suisse_JCP_Position_Statement.pdf) – können Sie Ihre Pläne für die nächste Periode zusammenfassen?

Credit Suisse: Wir freuen uns sehr über das großartige und ermutigende Resultat und werden weiterhin unsere Kundensicht einbringen, um für starke, offene und stabile Standards zu argumentieren,

die kompetitiv von beliebigen Software-Firmen oder Open Source Communities implementiert werden können. Wir bringen auch unsere Erfahrung und Expertise in der Standardisierung und Architektur-Governance ein und führen unsere aktive Teilnahme in Expert Groups und als Spec Leads fort. Wir möchten auch helfen, dass wieder vermehrt Universitäten teilnehmen; beispielsweise auf dem Fakultätslevel als Spec Leads oder Studierende auf dem Entwicklerlevel, etwa über die Adopt-a-JSR-Initiative der Java User Groups.

Credit Suisse als erster Kundenvertreter im JCP wird als Java User Company charakterisiert, die das Ziel hat, Java zu verbessern, um es besser sowohl den eigenen als auch den Bedürfnissen anderer Kunden anzupassen. Damit bringt Credit Suisse eine andere Perspektive als eine Software-Firma ein. Gibt es hier Reibungsmomente mit anderen JCP-Mitgliedern oder verstehen dadurch die anderen EC-Mitglieder besser, wie große Firmen Java einsetzen?

Credit Suisse: Wir sehen unsere Rolle allgemein in der Java-Community und speziell im Executive Committee als repräsentativer Kunde, der geschäftskritische Applikationen basierend auf starken, offenen und stabilen Standards entwickelt, auch um unsere jahrelangen Investitionen zu sichern. Credit Suisse hat schon davor aktiv zur Entwicklung von Java-Standards beigetragen, etwa durch Teilnahme in verschiedenen Customer Advisory Boards, die Beschreibung von konkreten Anforderungen an Java-Technologien sowie durch aktive Teilnahme in Expert Groups. Das JCP Executive Committee wird jedoch immer noch von Software-Firmen dominiert. Große Kunden von Java-Technologien waren im Prozess nicht berücksichtigt und konnten daher nicht an der Entwicklung und Evolution von Java teilnehmen. Credit Suisse steht in enger Verbindung mit Software-Firmen. Unsere Erfahrungen mit der konkreten Anwendung von Java-Technologien, beispielsweise mit unserer Java Application Platform, sind auch für Software-Firmen interessant und relevant.

Warum hat sich die Credit Suisse im Jahr 2010 entschieden, dem JCP beizutreten? Wie hilft die Teilnahme im JCP der Credit Suisse?

Credit Suisse: Oracle hat Credit Suisse für einen Sitz im Java SE/EE Executive Committee nominiert. Credit Suisse wurde dann von der JCP-Community mit 77 Prozent für den „ratified Seat" ab Juni 2010 gewählt und im Oktober 2012 erneut mit 85 Prozent bestätigt. Credit Suisse war eines der ersten Finanz-Institute, das Web- und Java-Technologien eingesetzt hat. Sie war auch die erste Schweizer Bank, die ihren Kunden bereits 1997 eine webbasierte e-Banking-Lösung zur Verfügung gestellt hat. YouTrade war im Jahr 1999 eine der ersten Client-Trading-Applikationen auf dem Markt. Credit Suisse hat eine interne Applikationsplattform basierend auf Java-EE-Standards namens Java Application Platform (JAP) entwickelt, mit der Entwicklung und Betrieb der Java-EE-Applikationen industrialisiert wurden. Schon 2004 hat JAP viele der Platform-as-a-Service-Konzepte realisiert und trägt so zu einer Kostenvermeidung von jährlich mehr als 40 Millionen Dollar bei. JAP realisierte eine der ersten großen SOA-Implementierungen mit einer Public-Key-Infrastruktur und reduzierte die Anzahl der physischen Server durch Konsolidierung auf virtuelle Server. Zusätzlich wurden zahlreiche Java-EE-Technologien wie Transaktionsmonitore, Java Server Faces (JSF) und Portal frühzeitig eingesetzt. Heutzutage investiert Credit Suisse signifikant in die Entwicklung neuer und in die Wartung bestehender Java-EE-Applikationen und besitzt mehr als 30 Millionen Zeilen Java-Code. Credit Suisse beabsichtigt auch weiter in Java-Technologien zu investieren, da wir mehr und mehr Workload vom Mainframe auf JAP verschieben.

Dr. Susanne Cech Previtali, Java-Architektin der Credit Suisse Group

Unterscheidet sich die Java-EE-Vision der Credit Suisse von der anderer JCP-Executive-Committee-Mitglieder?

Credit Suisse: Im Zentrum unserer Java-EE-Vision stehen die Unterstützung von Cloud-Features und Modularisierung. Die Standardisierung von PaaS-Features wie Provisionierung, Multi-Tenancy, Elastizität und das Deployment von Applikationen in der Cloud sind ausschlaggebend für Entwicklung und Betrieb. Die Modularisierung der Java-SE-Plattform und der daraus folgenden Möglichkeit der besseren Verwaltung von Abhängigkeiten und Versionen über Modulgrenzen hinweg sind wichtig für Enterprise-Applikationen. Solche Änderungen haben größere Auswirkungen auf die Java-Plattform und wurden deshalb von Java EE 7 auf Java EE 8 verschoben. Wir haben ähnliche Visionen, aber teilweise andere Vorstellungen von Roadmaps und deren zeitlicher Umsetzung.

Wie hat sich der JCP nach der Übernahme von Sun Microsystems durch Oracle verändert?

Credit Suisse: Die größte Änderung ist die JCP.next-Initiative, die sich zu höherer Transparenz, Teilnahme und Öffnung verpflichtet. JCP.next.1 (JSR 348, „Towards a New Version of the Java Community Process", siehe http://jcp.org/en/jsr/detail?id=348) hat den Grundstein für größere Transparenz gelegt, indem verlangt wird – und nicht nur empfohlen wurde –, dass die Entwicklung des JSR auf öffentlichen Mailinglisten und Issue Trackers stattfindet. Zusätzlich sind der Rekrutierungsprozess von Expert-Group-Mitgliedern sowie die Disclosure Agreements öffentlich einsehbar. Die Resultate des Technology-Compatibility-Kits-Testprozesses werden zur Verfügung gestellt; davor wusste die Öffentlichkeit kaum Bescheid über diese Resultate. All diese Entwicklungen beabsichtigen einen offenen, zugänglichen und transparenten JCP. JCP.next.2 (JSR 355, „JCP Executive Committee Merge",

siehe http://jcp.org/en/jsr/detail?id=355) adressiert das Zusammenlegen der beiden Executive Committees (Java SE/EE und Java ME) und bildet damit die Grundlage für eine ganzheitliche Plattform. JCP.next.3 (JSR 358, „A Major Revision of the Java Community Process", siehe http://jcp.org/en/jsr/detail?id=358) revidiert das Java Specification Participation Agreement, das eigentliche Prozess-Dokument, und die Standing Rules für das Executive Committee. Ziele sind, die Prozesse zu verbessern, Probleme, die über die Jahre bemerkt wurden, zu korrigieren sowie die Sprache zu vereinfachen und zu bereinigen, um Unklarheiten und Doppeldeutigkeiten zu reduzieren.

Welche strukturellen Änderungen wollen Sie im JCP sehen?

Credit Suisse: Offenheit, transparente Lizenzmodelle, einfache Teilnahme-Möglichkeiten, eine breite Teilnahme der Java-Community und das Bewusstsein für die Relevanz der Technologie-Governance.

Hat Oracle sein Versprechen von erhöhter Transparenz und Offenheit eingehalten?

Credit Suisse: Ja, als Mitglied des Executive Committee hat Oracle dem JCP.next.1 und JCP.next.2 zugestimmt und sich damit dazu verpflichtet. Jetzt arbeitet man mit den anderen Executive-Committee-Mitgliedern am JCP.next.3.

Wie könnte Oracle die Teilnahme am JCP erhöhen?

Credit Suisse: Die Aufnahme der JUGs ins Executive Committee ist bereits ein großer Erfolg, weil es Bewusstsein auf der richtigen Ebene bringt.

Wie könnte der JCP die Java-Community besser erreichen und dieser besser dienen?

Credit Suisse: Der JCP bestimmt die Evolution der Programmiersprache sowie der Plattform und ist dadurch sehr wichtig für die Java-Community. Indem wir die Bedürfnisse von Anwendern und Technologie-Anbietern zusammenbringen, kann der JCP sicherstellen, dass Java weiterhin eine der Top-Programmiersprachen bleibt. Wir sollten auch nicht die nächste Generation vergessen und dafür sorgen, dass Schulen und Universitäten weiterhin auf Java setzen.

Was stört Sie am meisten am JCP?

Credit Suisse: Die Java-Community ist großartig und sehr aktiv, aber nicht alle Anwender der Java-Technologie kennen den Wert und die Relevanz der Technologie-Governance. Es ist ja die Java-Community, die JSRs spezifiziert und entwickelt. Vielleicht könnten wir uns besser darstellen, damit wir mehr Leute und Firmen in die Evolution der Java-Technologie einbeziehen können.

Es scheint das Problem zu geben, dass Expert-Group-Mitglieder Spezifikationen beeinflussen, um ihre eigenen Use Cases durchzusetzen – es ist also schwierig, während des Entwicklungsprozesses objektiv zu bleiben. Wie kann man als Spec Lead unvoreingenommen sein und nicht die eigenen Use Cases bevorzugen?

Credit Suisse: Es ist nicht einfach, einen Konsens zu finden, denn Menschen haben nun mal höchst individuelle Vorstellungen, die natürlich manchmal Konflikte erzeugen können. Natürlich ist auch ein gewisses Ego dabei, aber das übergeordnete Ziel ist es, alle Use Cases zu finden und eine entsprechend robuste Spezifikation und Implementierung zu erarbeiten. Alle Spec Leads teilen ähnliche Herausforderungen, etwa auch, um neben der normalen Tätigkeit noch freie Zeit zu finden. Und selbst wenn man Zeit hat, ist es schwierig, die anderen EG-Mitglieder zu involvieren, weil die ja auch einen Job haben.

Können Sie in Ihrer Rolle als Spec Lead einige Änderungen für Java.net vorschlagen?

Credit Suisse: Java.net ist der beste Freund eines JSR Spec Lead. Es stellt alles, was es für die Leitung eines JSR braucht, zur Verfügung: Wiki, Source Control, Mailingliste, Archive. Auch das schnelle Feedback vom JCP bezüglich Support-Anfragen ist beeindruckend. Zu überlegen wäre, ob es nicht Performance-basierte Anreize für die Expert Groups geben könnte, etwa einen Preis für den wertvollsten Beitrag.

Erzählen Sie uns von den Verbesserungen für die Benutzer und Entwickler von Finanz-Software, die aus JSR 354 resultieren werden. Was wird deren Arbeit vereinfachen?

Credit Suisse: Entwickler werden nicht jedes Mal das Rad neu erfinden müssen, wenn sie Finanz-Applikationen entwickeln. Ein häufiger Fehler ist, dass Floats oder Doubles verwendet werden, um monetäre Werte zu repräsentieren. Allerdings sind diese Typen nicht genau und es können Rundungsfehler entstehen. Weil die Buchhalter Rundungsfehler gar nicht schätzen, muss einiges überlegt werden, damit monetäre Werte entsprechend genau sind. Zusätzlich wird der JSR die Formatierung verbessern. Die meisten Länder der USA und Europa verwenden Standardformate, aber Indische Rupien werden beispielsweise auch in Lakh und Crore ausgedrückt, die mit den Standard-Java-Formatters kaum darzustellen sind. Außerdem sind auch geeignete Rundungsregeln wichtig. In den USA wird typischerweise ab „5" auf- und zwischen „1" und „4" abgerundet. Aber es gibt auch Länder, die recht exotische Rundungsregeln verwenden.

Es gibt eine feine Grenze zwischen Standardisierung und Innovation – in welche Richtung bewegt sich der JCP?

Credit Suisse: Als Vertreter der Kundensicht setzen wir auf Standardisierung, weil wir damit relativ flexibel Produkte aussuchen können und trotzdem den Vendor-Lock-in verringern. Einige argumentieren, dass zumindest zwei Implementierungen existieren sollten, bevor man daraus einen Standard entwickelt. Andere bemerken, dass Kunden, die bereits existierende Implementierungen verwenden, bei der Migration auf den neuen Standard zusätzliche Aufwände tragen müssten. Man muss also eine Balance finden: Einerseits nicht zu früh standardisieren, weil das zu schwachen Standards führt, aber andererseits nicht zu spät, weil das wiederum hohe Migrationskosten bewirkt.

Hinweis: Das Interview führte Janice J. Heiss, Java Acquisitions Editor bei Oracle und Technologie-Redakteurin für das Java Magazine. Ins Deutsche übersetzt von Dr. Susanne Cech Previtali.

Java ohne Schwankungen

Matthew Schuetze, Azul Systems

Moderne Geschäftsanwendungen sind auf extrem niedrige Latenzzeiten im Bereich von wenigen Millisekunden angewiesen, wenn sie die Erwartungen der Kunden erfüllen und die strengen Vorgaben der Service Level Agreements einhalten sollen.

Die Anforderungen an die meisten Business-Systeme fallen immer strenger aus. Galten vor wenigen Jahren noch Reaktionszeiten unter 100 Millisekunden als ultraniedrig latent, bewegen sich die Zeiten heute bereits oft im Mikrosekundenbereich. Dieser Bereich gehörte noch nie zu den Stärken von Java, herkömmliche Java Virtual Machines (JVM) unterliegen Einschränkungen, die die Gewährleistung einer ultraniedrigen Latenz unmöglich machen. Zu den wichtigsten dieser Faktoren gehört die Speicherbereinigung (Garbage Collection).

Dieser Artikel erläutert die bisherigen Probleme im Zusammenhang mit der Latenz und den Schwankungen von Java-Anwendungen und zeigt, wie moderne Java-Anwendungen das erstrebte Ziel einer kontinuierlich beständigen Latenz bei anhaltend hohem Durchsatz erreichen können. Zudem wird beschrieben, wie sich die neue JVM-Technologie dazu nutzen lässt, eine Plattform-Latenz von höchstens zehn Millisekunden oder weniger „out of the box" und eine maximale Plattform-Latenz von ein bis zwei Millisekunden mit geringem Tuning-Aufwand zu erzielen.

Das Problem mit der Garbage Collection

Viele Anwendungen, die auf eine niedrige Latenz angewiesen sind, wurden ursprünglich in Java geschrieben, um die Vorteile und das Ökosystem dieser Sprache zu nutzen. Doch für Anwendungen, die Leistung mit ultraniedriger Latenz benötigen, weist Java Einschränkungen auf. Zu den wichtigsten davon gehört die Garbage Collection (GC) von Java. Da die JVM für den Entwickler die Speicherverwaltung übernimmt, muss sie die Ausführung der Anwendungen von Zeit zu Zeit unterbrechen, um alte Objekte wegzuräumen und den Heap zu defragmentieren. Je nach Größe kann dies einige Sekunden oder sogar Minuten in Anspruch nehmen, in denen sich die Transaktionen verzögern, der Durchsatz der Anwendung beeinträchtigt ist und bis zu einem Timeout der Benutzersitzung führen kann.

Die gute Nachricht ist, dass Java ein gewisses Maß an Kontrolle über die Speicherbereinigung ermöglicht. Entwickler und Architekten können Entscheidungen treffen, um die Leistung der Anwendung zu verbessern. Ziel ist, diese Unterbrechungen so weit wie möglich in die Zukunft zu verschieben, damit neu gebootet werden kann, bevor die Unterbrechung tatsächlich eintritt.

Häufigere kürzere Unterbrechungen – normalerweise im Bereich von mehreren Hundertstel Sekunden – lassen sich hingegen nicht vermeiden. Wenn jedoch alle Entwickler darauf achten, möglichst keinen Speicherplatz zuzuweisen, lässt sich die Dauer dieser Unterbrechungen auf wenige Millisekunden reduzieren. Doch mehr ist aus den derzeitigen JVMs nicht herauszuholen. Java selbst weist systeminhärente Schwankungen von deutlich mehr als einer Millisekunde auf. Das ist nicht die Schuld der Anwendung oder des Systems, sondern liegt vielmehr in der JVM und ihrem Verhalten begründet.

Moderne Server sind keine Lösung

Moderne x86-Server bieten Speicherkapazitäten von mehreren Hundert Gigabytes sowie ein Dutzend oder mehr vCores zu äußerst erschwinglichen Preisen. Doch die meisten Unternehmen stellen den Java-Anwendungsinstanzen auf ihren Servern nur einen Bruchteil dieser Ressourcen zur Verfügung. Das hängt damit zusammen, dass traditionelle JVMs bei der Speicherbereinigung Unterbrechungen verursachen, die bei einem großen Heap stark ins Gewicht fallen. Für Anwendungen mit Benutzerkontakt liegt diese Schmerzgrenze bei einer Reaktionszeit von mehr als einigen Sekunden. Bei Maschine-zu-Maschine- oder anderen niedriglatenten Anwendungen liegt die Grenze im Milli- oder sogar Mikrosekundenbereich.

Nicht mehr benötigte Objekte, die weggeräumt werden müssen, sammeln sich schnell an. Jeder Kern eines Servers erzeugt im Normalbetrieb pro Sekunde 0,25 bis 1,5 Gigabyte solcher Objekte. Sobald sich im Heap des Systems mehrere Gigabyte Daten angesammelt haben, liegen die Unterbrechungszeiten bereits bei mehreren Sekunden. In der Welt der niedrigen Latenz werden diese längeren Unterbrechungen in der Hoffnung hinausgezögert, dass die Server vor ihrem Auftreten neu gestartet werden können; zu kürzeren Unterbrechungen kommt es jedoch nach wie vor. Dies führt dazu, dass Unternehmen Java auf Servern mit umfangreichen Ressourcen mit immer mehr winzigen Instanzen einsetzen.

Bei Anwendungen mit niedriger Latenz geht es nicht mehr nur darum, die gesamte Speicherkapazität des Servers zu nutzen – es geht darum, dass Java als Sprache nicht mehr genutzt werden kann, weil eine immer niedrigere Latenz gefordert wird. Den Kern des Problems bilden monolithische Speicherbereinigungen vom Typ „Stop the world". Das System muss für die Speicherbereinigung sämtliche Vorgänge anhalten, und die Speicherbereinigung muss ohne Unterbrechung vollständig durchgeführt werden („monolithisch"). Wäre das nicht der Fall, könnte man die Arbeit in kleinere, inkrementelle Abschnitte aufteilen, um die gesamte Latenz niedrig zu halten.

Wäre jeder Abschnitt klein genug (etwa unter einer Millisekunde) und könnte das System die Abschnitte über einen größeren Zeitraum verteilen, wäre die erhöhte Latenz weniger problematisch. Doch eine herkömmliche JVM ist auf eine monolithi-

sche Speicherbereinigung angewiesen. Das hat folgenden Grund:

Angenommen, der Garbage Collector der JVM muss den Speicher kompaktieren, also Objekte verschieben, um größere Blöcke mit freiem Speicherplatz zu erzeugen. Dazu verschiebt er ein Objekt von Punkt A nach Punkt B. Allerdings können Tausende, Millionen oder sogar Milliarden Zeiger auf dieses Objekt verweisen. Bevor die Anwendung den nächsten Befehl ausführen kann, muss der Garbage Collector jeden dieser Zeiger finden und aktualisieren. Die Verschiebung des Objekts lässt sich nicht rückgängig machen; sobald das Objekt verschoben wurde, muss die Anwendung daher warten, bis jeder einzelne Zeiger gefunden und aktualisiert wurde. Auf diese Weise arbeiten heute beinahe alle JVMs, was dazu führt, dass die Anwendungen keine beständig niedrige Latenz erzielen.

Ultraniedrige Latenz in Java mit der Zing JVM

Die Zing JVM löst sämtliche Probleme der Speicherbereinigung für Unternehmens- und niedriglatente Anwendungen. Im Gegensatz zu anderen JVMs, die zur Minimierung der GC-Unterbrechungen aufwändiges Tuning oder Anpassungen erforderlich machen, beseitigt Zing diese Unterbrechungen vollständig. Aspekte der Skalen-Metrik, wie größere Heaps, mehr Transaktionen oder mehr Anwender, werden vollständig von der Reaktionszeit abgekoppelt. Ebenso ist Zing die einzige JVM mit variablen Speicher-Funktionen, die selbst in unvorhersehbaren Situationen Stabilität gewährleisten. Listing 1 zeigt ein Beispiel für die C4-Collector-Einstellungen in Zing.

Zing bietet ohne spezielles Tuning beständige Reaktionszeiten unter zehn Millisekunden, sofern die Schwankungen des Systems im Leerlauf unter zehn Millisekunden liegen. Mit etwas Tuning, das in der Regel etwa ein bis zwei Tage in Anspruch nimmt, lässt sich mit Zing eine Reaktionszeit der JVM-Plattform von höchstens ein bis zwei Millisekunden problemlos verwirklichen. In diesem Bereich bewegen sich Anwendungen, die hundert Mikrosekunden Arbeit verrichten. Die Schwankung einer Anwendung, die zehn Millisekunden Arbeit verrichtet, würde sich durch die JVM nur um ein bis zwei Millisekunden erhöhen. Auch die Tuning-Parameter werden durch Zing erheblich vereinfacht. Zing ist eine 100-prozentige Java-kompatible JVM auf der Grundlage von HotSpot (siehe Abbildung 1).

GC-Tuning

Selbst wenn man sich mit den Besonderheiten der jeweiligen Anwendung auskennt, bereitet das richtige Tuning der GC meist einige Schwierigkeiten. Die obige Abbildung zeigt zwei Sätze von Tuning-Parametern für den Oracle CMS Collector. Obwohl in diesen Sätzen ähnliche Parameter zum Einsatz kommen, sind die Sätze grundverschieden und in einigen Bereichen diametral entgegengesetzt. Dennoch lässt sich die Leistung der Anwendung mit jedem Satz je nach den speziellen Merkmalen optimieren.

Für die meisten GCs gibt es keine Einheitslösung. Vielmehr muss die Speicherbereinigung von den Entwicklern und Architekten sorgfältig abgestimmt werden; dieses Tuning muss zudem bei jeder Veränderung der Anwendung, Umgebung oder voraussichtlichen Last wiederholt werden. Ein Fehler bei diesen Parametern kann zu unerwarteten langen Unterbrechungen während der Spitzenlastzeiten führen.

Mit Zing ist die Leistung einer Anwendung nicht von den üblichen Tuning-Parametern abhängig. Einzig wichtig ist die Größe des Heaps. Allerdings arbeitet Zing durchaus mit Tuning-Markierungen, die für niedriglatente Anwendungen zuweilen nützlich sein können. So lässt sich zum Beispiel die Speicherbereinigung auf einen einzigen Thread beschränken, um die Planungsverzögerungen auf unter zehn Millisekunden zu begrenzen. Ein spezielles Tuning für Reaktionszeiten von unter zehn Millisekunden ist mit Zing hingegen nicht erforderlich.

Einige Zing-Anwender haben Reaktionszeiten von höchstens eine Millisekunde erreicht (siehe Abbildung 2); Voraussetzung hierfür sind allerdings ein aufwändigeres Tuning sowie eventuell eine Veränderung der Anwendung, eine Zuweisung von Prozessoren für bestimmte Funktionen oder die Weiterleitung von Unterbrechungen an bestimmte Kerne (siehe „http://mechanical-sympathy.blogspot.de“).

Fazit

In zahlreichen Branchen sinken die erforderlichen maximalen Latenzen für Java-Anwen-

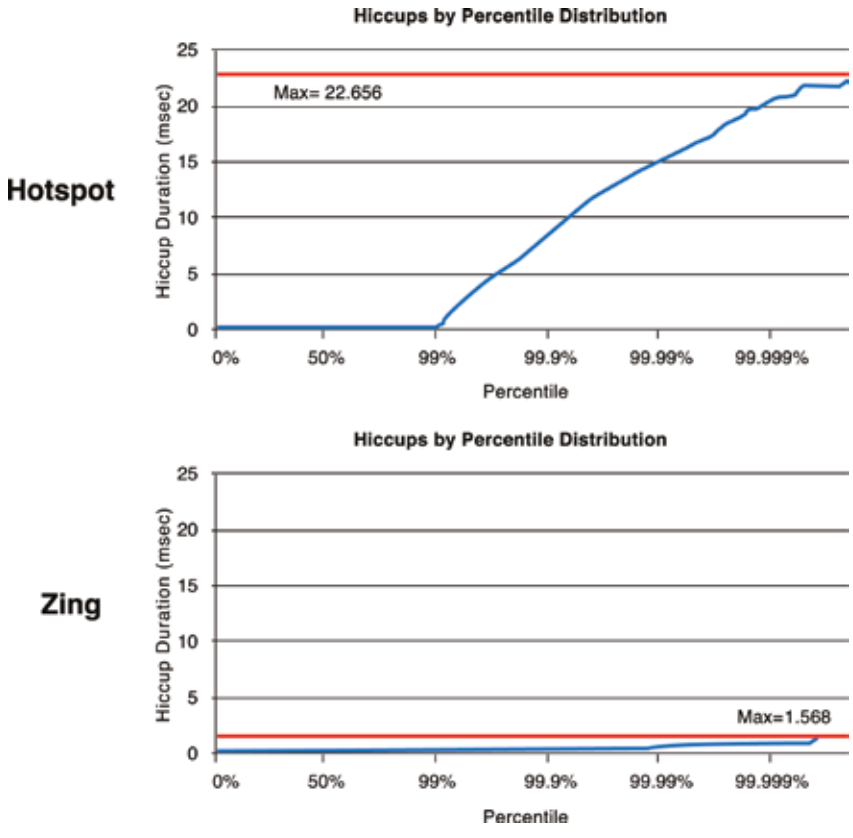


Abbildung 1: Tatsächliche Leistung: Zing im Vergleich zu Hotspot

dungen auf ein immer niedrigeres Niveau. Leider sind herkömmlichen JVMs Grenzen gesetzt, die durch Unterbrechungen bei der Speicherbereinigung verursacht werden; hierdurch erhöht sich die Latenz selbst dann, wenn man größere Speicherbereinigungsvorgänge hinauszögert, bis das System neu gebootet werden kann.

```
Java -Xmxl2g -XX:MaxPermSize=64M -XX:PermSize=32M -XX:MaxNewSize=2g
        -XX:NewSize=lg -XX:SurvivorRatio=l28 -XX:+UsePartNewGC
        -XX:+UseConcMarkSweepGC -XX:MaxTenuringThreshold=0
        -XX:CMSInitiatingOccupancyFraction=60 -XX:+CMSParallelRemarkEnabled
        -XX:+UseCMSInitiatingOccupancyOnly -XX:ParallelGCThreads=l2
        -XX:LargePageSizeInBytes=256m ...

Java –Xms8g -Xmx8g -Xmn2g -XX:PermSize=64M -XX:MaxPermSize=256M
-XX:-OmitStackTraceInFastThrow -XX:SurvivorRatio=2 -XX:-UseAdaptiveSizePolicy
-XX:+UscConcMarkSweepGC -XX:+CMSConcurrentMTEnabled
-XX:+CMSParallelRemarkEnabled -XX:+CMSParallelSurvivorRemarkEnabled
        -XX:CMSMaxAbortablePrecleanTime=10000 -XX:+UseCMSInitiatingOccupancyOnly
        -XX:CMSInitiatingOccupancyFraction=63 -XX:+UseParNewGC -Xnoclassgc ...
```

Listing 1

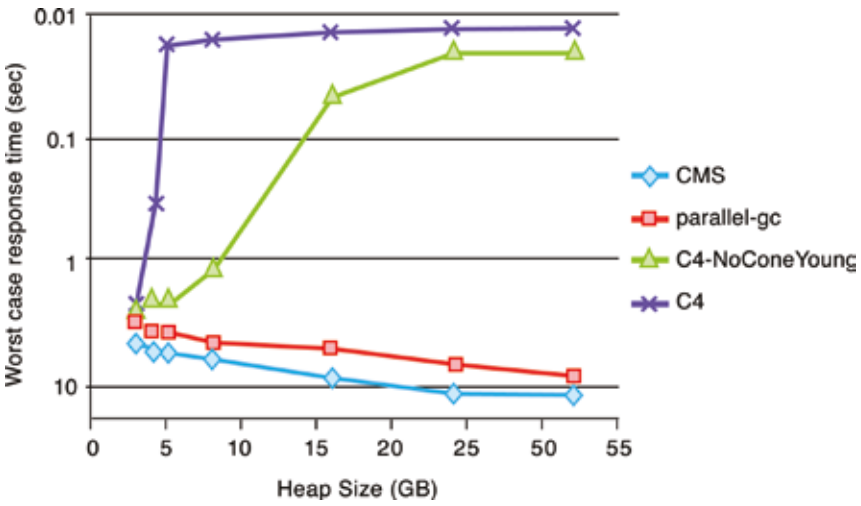


Abbildung 2: Reaktionszeiten im schlimmsten Fall

Die Zing JVM ermöglicht als ungetuntes „Out of the box"-System unübertroffene Reaktionszeiten von höchstens zehn Millisekunden. Als einzige handelsübliche JVM koppelt sie Aspekte der Skalenmetrik, wie größere Heaps, mehr Transaktionen oder mehr Anwender, vollständig von der Reaktionszeit ab.

Matthew Schuetze
uroesch@azulsystems.com

Matthew Schuetze ist Produkt-Manager für die Zing-Proukt-famlie bei Azul Systems. Er arbeitet gemeinsam mit Kunden und Partnern an Ideen für neue Features und verfügt über mehr als fünfzehn Jahre Erfahrung bei der Erstellung von Software-Anwendungen und mit Entwicklungs-Tools, davon knapp zehn Jahre mit statischen und dynamischen Code-Analyzern für Java und JVMs. Matthew Schuetze hat Ingenieur-Abschlüsse am Massachusetts Institute of Technology (MIT) und an der University of Michigan.

Wenn die „ADF Task Flows" in JavaServer Faces einfließen, heißen sie „Faces Flows" – Grundlagen eines neuen Features von JSF 2.2

Die Version 2.2 von JavaServer Faces (JSF) ist mit etlichen neuen Features herausgekommen. Für Entwickler sind im Standard-Framework unter den siebzig neuen Funktionen und Verbesserungen drei besonders interessant: „HTML5 Friendly Markup", „Resource Library Contracts" und „Faces Flows". DOAG Online hat die „Faces Flows" herausgepickt und Ed Burns darum gebeten, die Grundlagen dieser Funktionalität zu erläutern.

Lesen Sie den Artikel auf www.doag.org/go/java-serverfaces

Ed Burns
Consulting Member of the Technical Staff
Oracle America, Inc.

Dynamische Reports innerhalb von Oracle-Datenbanken

Michael Simons, ENERKO Informatik GmbH

Etliche Java-Entwickler kennen ihre Datenbank nur über eine objektrelationale Abbildung. Umgekehrt ist für viele datenbanknahe Entwickler Java nur die Sprache der Anwendung, die ständig umständliche Statements an die Datenbank übergibt. Allerdings gibt es sehr wohl elegante Zugriffsmöglichkeiten aus Java auf Datenbanken und man kann ebenso objektorientiert in einer Datenbank arbeiten.

Oracle stellt bereits seit 1999 eine native JVM [1] innerhalb der Datenbank zur Verfügung, in der Geschäftslogik in nahezu jeder Form entwickelt und über eine PL/SQL-Schnittstelle anwendungsweit zur Verfügung gestellt werden kann. Anhand der ENERKO-Report-Engine wird gezeigt, wie man SQL, PL/SQL und Java-Datenbank einfach und gewinnbringend verbinden kann.

Im Rahmen eines Oracle-Forms-6*i*-Ablöseprojekts stellte sich die Frage nach dem Reporting. Oracle Reports ist Teil der Developer Suite und wurde entsprechend mit abgelöst. Da die vorhandene Applikation nicht mit einer Big-Bang-Ablösung umzustellen war, mussten vorhandene und neue Reports gleichermaßen im alten Forms und im neuen Java-SE-Client zur Verfügung stehen. Weiterhin sollte es möglich sein, Reports mit dem vorhandenen Wissen über SQL sowie Excel-Dateien als Vorlagen ohne Java, XML oder ähnliche Kenntnisse erstellen zu können.

Die Datenquellen der vorhandenen Reports konnten als SQL-Statements aus den Report-Definitionen archiviert werden, das Layout wurde in Form von Excel-Dateien extrahiert. Das Ziel war somit klar: Die Datenbank ist der zentrale Speicherort für Report-Definitionen jeglicher Art und die Erzeugung der Reports soll ebenfalls in der Datenbank stattfinden.

```
public class CellDefinition {
        /** The name of the sheet inside the Excel document */
        public final String sheetname;
        /** Column Index (0-based) */
        public final int column;
        /** Row Index (0-based) */
        public final int row;
        /** Cellreference ("A1" notation),
        *only used for output) */
        public final String name;
        /** Contains either type or type and a reference cell
        *  as  datentyp;"SHEETNAME" CELLREFERENCE */
        private final String type;
        /** A string representation of the value */
        public final String value;
}
```

Listing 1: CellDefinition

```
private void addCell(
        final Workbook workbook,
        final Sheet sheet,
        final CellDefinition cellDefinition
) {
        final int columnNum = cellDefinition.column,
                rowNum = cellDefinition.row;

        Row row = sheet.getRow(rowNum);
        if(row == null)
                row = sheet.createRow(rowNum);
        Cell cell = row.getCell(columnNum);
        if(cell != null &&
                cell.getCellType() != Cell.CELL_TYPE_BLANK) {
                        cell = fill(workbook, cell, cellDefinition, false);
        } else {
                cell = row.createCell(columnNum)
                cell = fill(workbook, cell , cellDefinition, true);     }
}
```

Listing 2: Erzeugen und Füllen einer Zelle

Java und Excel

Wenn es um Reporting geht, liegt im Java-Kontext der Gedanke an JasperReports nahe, allerdings konnte man aufgrund der genannten Ziele nicht auf diese Lösung setzen und benutzte stattdessen Apache POI [2]. Die Erzeugung der Arbeitsmappen liegt somit in der eigenen Verantwortung. Das Modul „Apache POI-HSSF" ist eine reine Java-Implementierung des „Excel 97-2007"-Formats. POI steht dabei für „Poor Obfuscation Implementation" und HSSF für „Horrible Spreadsheet Format". Leider ist diese Erklärung nur noch in Wikipedia zu finden, „um die Bibliothek für Geschäftsanwendungen attraktiver zu machen, die Humor dieser Art für unangemessen halten".

Das Usermodel von POI-HSSF unterstützt das Lesen und insbesondere Erzeugen

```
CREATE TYPE t_er_cell_definition AS OBJECT (
        sheetname       VARCHAR2(512),
        cell_column     INTEGER,
        cell_row        INTEGER,
        cell_name       VARCHAR2(64),
        cell_type       VARCHAR2(512),
        cell_value      VARCHAR2(32767),

        CONSTRUCTOR FUNCTION t_er_cell_definition(
                p_sheetname     VARCHAR2,
                p_cell_column   INTEGER,
                p_cell_row      INTEGER,
                p_cell_type     VARCHAR2,
                p_cell_value    VARCHAR2
        ) RETURN self AS result
)
/
```

Listing 3: Benutzerdefinierte PL/SQL „Typ t_er_cell_definition“

```
CREATE OR REPLACE
FUNCTION f_say_hello(num_rows NUMBER, p_hello_to IN VARCHAR2) RETURN table_of_
er_cell_definitions pipelined IS
BEGIN
        FOR i IN 0 .. (num_rows - 1) LOOP
                pipe row(
                        t_er_cell_definition(
                                'Hello',
                                0,
                                i,
                                'string',
                                'Hello, ' || p_hello_to
                        )
                );
        END LOOP;
        RETURN;
END f_say_hello;
/
SELECT cell_column, cell_row, cell_value
FROM table(f_say_hello(5,'JavaAktuell'));
```

Listing 4: Table function „f_say_hello“

```
FUNCTION f_create_report(
        p_method_name IN VARCHAR2,
        p_template IN BLOB,
        p_args IN t_vargs
) RETURN BLOB IS LANGUAGE JAVA NAME 'de.enerko.reports2.PckEnerkoReports2.
createReport(java.lang.String, oracle.sql.BLOB, oracle.sql.ARRAY) return oracle.sql.BLOB';
```

Listing 5: Call specification „f_create_report“ für „PckEnerkoReports2.createReport“

```
private static String[ ] extractVargs(final ARRAY arguments) throws SQLException {
        final String[ ] $arguments;
        if(arguments == null)
                $arguments = new String[0];
        else {
                $arguments = new String[arguments.length()];
                final ResultSet hlp = arguments.getResultSet();
                int i = 0;
                while(hlp.next())
                        $arguments[i++] = hlp.getString(2);
        }
        return $arguments;
}
```

Listing 6: Verarbeitung von PL/SQL-Arrays

von Arbeitsmappen und Zellen, die Auswertung von Formeln, benutzerdefinierte Funktionen zur Verwendung in Formeln und vieles mehr. Arbeitsmappen können von Grund auf neu erzeugt oder auf Basis bestehender Arbeitsmappen (Templates) definiert werden.

Apache-POI-HSSF benötigt mindestens eine Java-1.5-Laufzeit-Umgebung. Für das Projekt ist das ein kleiner Nachteil, da erst ab Oracle 11g R1 eine 1.5-JVM in der Datenbank implementiert ist. Falls eine 10er-Datenbank benutzt werden soll, kann beispielsweise das Java-Excel-API [3] eine Alternative darstellen. Der große Nachteil hier ist, dass Formeln zwar gelesen und geschrieben, aber nicht ausgewertet werden.

Apache-POI-HSSF (und die einzige, weitere Abhängigkeit „commons-codec“) müssen als Bibliotheken in die Datenbank geladen sein. Dies kann über das Tool „loadjava“ erfolgen, das Teil des Oracle-Clients ist, oder über das Package „dbms_java“, wenn die Jar-Dateien in einem vom Datenbank-Server lesbaren Verzeichnis liegen. Man hat die Möglichkeit, Java-Quellen oder vorkompilierte Klassen zu laden.

Im Falle von Java-1.5-Code gibt es einige Besonderheiten der Oracle-JVM. Das „For-each“ und einige weitere, insbesondere generische Sprach-Konstrukte sind nicht unterstützt. Quellen mit diesen Konstrukten lassen sich nicht in der Datenbank kompilieren. Werden Bibliotheken und Klassen, die diese Konstrukte benutzen, extern kompiliert, lassen sie sich allerdings sehr wohl verwenden.

Definition einer Zelle

Eine Zelle einer Arbeitsmappe kann mit wenigen Attributen beschrieben werden (siehe Listing 1). Die Zell-Definition ist bewusst nicht typisiert und der Wert wird als String abgespeichert, um später beim Abruf der Werte aus der Datenbank maximale Flexibilität zu haben. Bereits mit dieser Definition ist es möglich, die Zellen einer Excel-Datei zu füllen (siehe Listing 2).

Indem die Methode „addCell“ als Teil der Report-Klasse und nicht als Teil der Definition der Zelle implementiert ist, kann man die Report-Klasse gegebenenfalls durch ein Interface ersetzen und mehrere Implementierungen (etwa zusätzlich durch das Java-Excel-API) anbieten.

PL/SQL-Objekte

PL/SQL-Typen können genau wie Java-Klassen Konstruktoren, Typ-Methoden (statische Methoden) und Objekt-Methoden haben. Listing 3 zeigt die Spezifikation des Typs, der eine Zelle einer Arbeitsmappe repräsentiert und mit der Klasse „CellDefinition“ aus Listing 1 korrespondiert.

Der „Object-Relational Developer's Guide“ [4] bietet eine sehr schöne Übersicht darüber, was Objekte in der Datenbank leisten können. Dieser Typ könnte nun einzeln als „java.sql.Struct“ beziehungsweise als Liste innerhalb eines „java.sql.Array“ an die Report Engine übergeben werden, widerspräche aber der Anforderung, Reports durch Statements oder Funktionen definieren zu können. Daher werden Zell-Definitionen entweder direkt aus SQL-Statements erzeugt („connection.createStatement().executeQuery(“ “)“) oder durch den Aufruf sogenannter „pipelined functions“. Diese sind eine Besonderheit innerhalb von Oracle-Datenbanken: „Pipelined oder Table-Funktionen sind Funktionen, die eine Menge von Zeilen eines bestimmten Typs produzieren, die genau wie physikalische Tabellen abgefragt werden können“ [5]. Listing 4 zeigt ein typisches „Hello World“ und seinen Aufruf.

Java-Funktionen aus SQL und PL/SQL aufrufen

Innerhalb der Datenbank geschieht der Aufruf eines Java-Programms ähnlich wie in einer „normalen“ JVM, nämlich über eine statische Methode, da vor dem Aufruf noch kein Objekt existiert. Der Aufruf geschieht dabei über eine „Call Specification“, in der der vollständig qualifizierte Name der statischen Java-Methode sowie die vollständig qualifizierten Typen der Parameter angegeben werden müssen (siehe Listing 5).

In diesem Beispiel sieht man, dass auch problemlos benutzerdefinierte PL/SQL-Typen („t_vargs“) an Java übergeben werden können. Listings 6 und 7 zeigen die Verarbeitung beziehungsweise die Erzeugung von benutzerdefinierten PL/SQL-Typen innerhalb von Java.

Report-Erzeugung

An dieser Stelle stehen alle Teile zur Verfügung, die zur Erzeugung von Reports benötigt werden: Definitionen der Zellen auf

```
private static ARRAY convertListOfCellsToOracleArray(final List<CellDefinition> cellDefinitions)
throws SQLException {
	final StructDescriptor resultStruct = StructDescriptor.createDescriptor("T_ER_CELL_
DEFINITION", connection);
	final ArrayDescriptor  arrayDesc = ArrayDescriptor.createDescriptor("TABLE_OF_ER_
CELL_DEFINITIONS", connection);

	final STRUCT[] rv = new STRUCT[cellDefinitions.size()];
	int i=0;
	for(CellDefinition cellDefinition : cellDefinitions)
		rv[i++] = new STRUCT(resultStruct, connection, cellDefinition.toSQLStructOb-
ject());
	return new ARRAY(arrayDesc, connection, rv);
}
```

Listing 7: Erzeugung von PL/SQL-Arrays mit benutzerdefinierten Typen

```
try {
	connection = (OracleConnection) DriverManager.getConnection("jdbc:default:connecti
on:");
	reportEngine = new ReportEngine(connection);
} catch (SQLException e) {
	throw uncheck(e);
}

public class ReportEngine {
	private final OracleConnection connection;
	private final Map<String, FreeRefFunction> customFunctions = new HashMap<String, Fre
	eRefFunction>();

	public ReportEngine(OracleConnection connection) {
		this.connection = connection;
		this.addCustomFunction("Enerko_NormInv", new NormInv());
	}
}
```

Listing 8: „ReportEngine.java“

```
DECLARE
	v_report BLOB;
BEGIN
	-- Create the report
	v_report :=
		pck_enerko_reports2.f_create_report(
			'f_say_hello', t_vargs('5', 'JavaAktuell')
		);
	-- Store it into a server side file
	pck_enerko_reports2.p_blob_to_file(
		v_report, 'enerko_reports', 'f_say_hello.xls'
	);
END;
/
```

Listing 9: Erzeugung eines Reports und Speicherung als BLOB

```
SELECT *
FROM table(
	pck_enerko_reports2.f_eval_report(
		f_say_hello', null, t_vargs('5', 'JavaAktuell')
	)
) src;
```

Listing 10: „On the fly“-Evaluierung von Reports/Arbeitsmappen

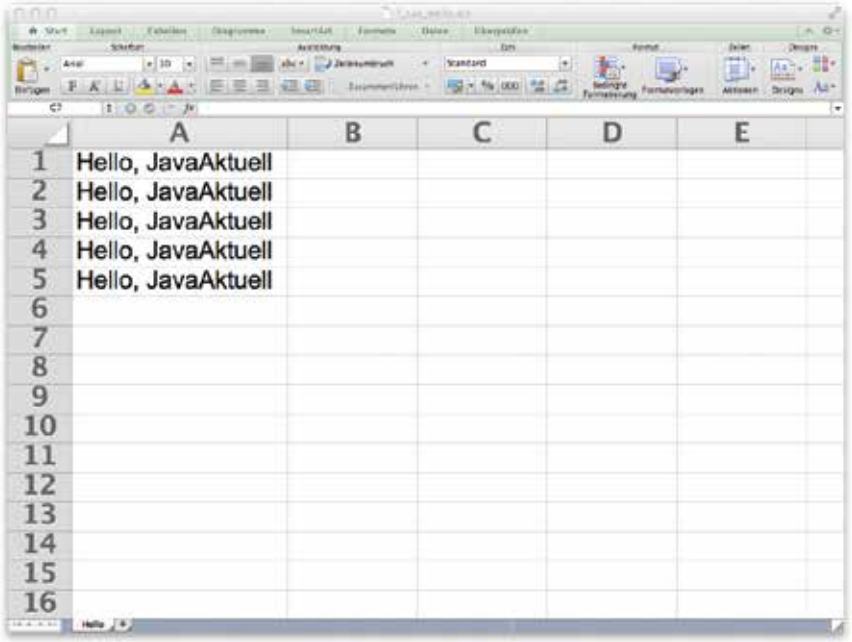


Abbildung 1: Das Ergebnis des Hello-World-Reports aus Listing 4

SQL, PL/SQL und Java-Ebene, Datenquellen für Statements und Funktionen auf Java-Ebene sowie eine Factory-Klasse, die auf Apache-POI-HSSF basierende Reports erzeugt und im passenden Format ausgibt (siehe Listing 8). Die Factory-Klasse hält dabei nicht nur die interne JDBC-Verbindung zur Datenbank, sondern auch eine Map mit benutzerdefinierten Excel-Funktionen, die als Java-Klassen implementiert sind.

Das Package „pck_enerko_reports2" stellt darüber hinaus überladene Funktionen zur Verfügung, die einen Report auf Basis eines Templates erzeugen können. Ein Template kann jede Excel-Arbeitsmappe sein, die als BLOB innerhalb der Datenbank zur Verfügung steht.

In einigen Fällen ist keine Excel-Arbeitsmappe als Ausgabe erwünscht, sondern ein Report, der wie eine Tabelle abgefragt werden kann. In diesem Fall kommen wieder „pipelined functions" ins Spiel. Benutzerdefinierte Typen können, wie in Listing 7 gezeigt, auf Java-Ebene instanziiert und genutzt werden, um neu erstellte Reports oder vorhandene Excel-Arbeitsmappen wie physikalische Tabellen abfragen zu können (siehe Listing 10).

Fazit

Die ENERKO-Report-Engine ist seit Anfang 2010 in mehreren Projekten, unter anderem bei Vertriebsmanagement-Aufgaben im Strom- und Gasbereich, im Einsatz und erzeugt komplexe Reports mit teilweise weit mehr als 50.000 Zellen, aufwändigen Grafiken und Makros. Sie wird zudem in der Preiskalkulation verwendet und dient zur Auswertung von Rechen-Schemata, die als Excel-Arbeitsmappen zur Verfügung gestellt werden. SQL-Entwickler nutzen die Engine gern, da sie die gewohnte Arbeitsumgebung nicht verlassen müssen, und die Kunden freuen sich über Auswertungen innerhalb eines bekannten Werkzeugs. Es hat sich also gelohnt, um die Ecke zu denken, gewohnte Bahnen zu verlassen und mit der Report-Engine zwei Welten zu verbinden.

Im Rahmen des Artikels ist die Report Engine vollständig überarbeitet und von Altlasten befreit worden. Sie steht nun unter Apache-2-Lizenz auf GitHub [6] zum Download bereit; Jar-Dateien zur direkten Verwendung innerhalb der Datenbank stehen im Central Repository zur Verfügung. Die Schnittstelle für benutzerdefinierte Funktionen soll in Zukunft noch erweitert werden. Darüber hinaus ist geplant, zusätzliche Infrastruktur-Klassen für die Verwaltung von Vorlagen zu implementieren.

Links

[1] Oracle Database: http://en.wikipedia.org/wiki/Oracle_database#History
[2] Apache POI: http://poi.apache.org
[3] Java Excel API: http://jexcelapi.sourceforge.net
[4] Object-Relational Developer's Guide: http://docs.oracle.com/cd/E11882_01/appdev.112/e11822/adobjint.htm#ADOBJ001
[5] Pipelined / Table functions: http://docs.oracle.com/cd/E11882_01/appdev.112/e10765/pipe_paral_tbl.htm
[6] ENERKOs Report Engine: https://github.com/michael-simons/enerko-reports2
[7] ENERKO Informatik GmbH: http://enerkogruppe.de

Michael Simons
michael@simons.ac

Michael Simons ist Software-Architekt bei ENERKO Informatik [7] in Aachen und entwickelt dort GIS-, EDM- und Vertriebsmanagement-Systeme für Stromnetz-Betreiber und Energie-Lieferanten. Dabei setzt er auf Java SE, Java EE sowie Oracle-Datenbanken. Er bloggt über Lösungen für tägliche und nicht alltägliche Probleme der Entwicklung auf http://info.michael-simons.eu.

Universelles Ein-Klick-Log-in mit WebID

Angelo Veltens, http://datenwissen.de

WebID bietet eine globale Identität im World Wide Web. In der letzten Ausgabe wurden die theoretischen Grundlagen dieser Technik behandelt. Dieser Artikel zeigt nun, wie sich mit WebID ein universelles Ein-Klick-Log-in für eine Web-Anwendung realisieren lässt.

Womöglich arbeitet jemand gerade an einer großartigen neuen Web-Anwendung. Doch statt sich auf deren Kern-Features konzentrieren zu können, muss er sich Gedanken um Account-Verwaltung, Log-in und Passwort-Sicherheit machen. Natürlich dürfen heutzutage auch soziale Funktionen nicht fehlen: Die Nutzerinnen und Nutzer sollen sich befreunden und Inhalte teilen können. Nicht selten kommt es vor, dass neue Web-Dienste ihr eigenes kleines Social Network implementieren, von ihren Nutzern abverlangen, abermals ein Passwort zu wählen, ein Profil auszufüllen und die Accounts ihrer Freunde zu finden. Statt das Rad neu zu erfinden, kann man WebID verwenden, um seinen Nutzern ein Ein-Klick-Log-in zu ermöglichen, das Account-Informationen und soziale Vernetzung gleich mitliefert.

Rückblick: Das Web als Social Network

Werfen wir zunächst einen kurzen Blick zurück auf Linked Data und wie damit Personen und deren Beziehungen abgebildet werden können. Wie alle Dokumente und Dinge im Linked-Data-Web werden auch Personen über einen URI identifiziert. Mithilfe des Ressource Description Frameworks (RDF) und der Friend-of-a-Friend-Ontologie (FOAF) können wir maschinenlesbare Aussagen über eine so identifizierte Person treffen. Listing 1 zeigt einen kurzen Ausschnitt aus dem FOAF-Profil des Autors.

```
<http://me.desone.org/person/aveltens#me>
        a foaf:Person;
        foaf:name "Angelo Veltens";
        foaf:homepage <http://datenwissen.de/>;
        foaf:knows <http://www.w3.org/People/Berners-Lee/card#i>.
```

Listing 1

Es enthält sowohl einfache Literale wie den Namen als auch Links zu anderen Ressourcen im Web. Diese Links sind der Schlüssel zum Aufbau eines dezentralen sozialen Netzwerks. Profile von Personen können auf unterschiedlichen Servern gehostet und trotzdem über ihren URI miteinander verlinkt werden. Bei Dokumenten ist dies seit Jahrzehnten gelebte Praxis. Linked Data wendet dieses Prinzip auch auf Daten, Dinge und Personen an. Verlinkt man die Nutzer auf diese Weise untereinander und mit den von ihnen verbreiteten Inhalten, entsteht ein globales, Server- und Dienste-übergreifendes Social Network. Das WebID-Protokoll dient dabei zur Authentifizierung der Nutzerinnen und Nutzer.

Linked Data und WebID wurden in den letzten Ausgaben ausführlich beschrieben. Wer die Artikel verpasst hat, kann sie unter [1] herunterladen. Nachfolgend wird gezeigt, wie die Authentifizierung über WebID in einer Anwendung implementiert werden kann.

Authentifizierung einer WebID

Das WebID-Protokoll kommt ohne Nutzernamen und Passwörter aus. Die Nutzer werden stattdessen über ein X.509-Zertifikat und ihren URI identifiziert. Das Zertifikat wird im Browser hinterlegt und enthält den URI des FOAF-Profils als „Subject Alternative Name". Das Profil wiederum enthält den öffentlichen RSA-Schlüssel. Um eine Nutzerin über WebID zu authentifizieren, muss ein Web-Server also zunächst eine SSL-Client-Authentifizierung durchführen und nach einem Zertifikat verlangen. Dieses Zertifikat muss gültig sein in dem Sinne, dass privater und öffentlicher Schlüssel zusammenpassen. Jedoch muss das Zertifikat nicht von einer Certificate Authority (CA) signiert oder dem Server bekannt sein. Stattdessen nutzt der Server den im Zertifikat hinterlegten URI, um die Authentifizierung und gegebenenfalls eine Autorisierung durchzuführen. Stimmt der im verlinkten Profil hinterlegte öffentliche Schlüssel mit dem des Zertifikats überein, ist nachgewiesen, dass es sich bei der Besitzerin des Zertifikats auch um die Besitzerin des Profils handelt.

Um eine Authentifizierung mit WebID zu ermöglichen, können die genannten Schritte manuell implementiert werden. Nachfolgend werden jedoch zwei alternative Wege aufgezeigt, die auf bestehende Lösungen setzen und die Sache deutlich vereinfachen. Die erste Variante delegiert die Authentifizierung an einen externen Dienst. Die zweite nutzt das Apache-Modul „mod_authn_webid", um die Authentifizierung auf einem eigenen Server durchzuführen. Wer sich dennoch im Detail mit einer eigenen Implementierung der WebID-Spezifikation auseinandersetzen möchte, findet sie unter [2].

Eine WebID erstellen

Bevor wir uns an die eigentliche Implementierung machen, wollen wir uns zunächst selbst eine WebID erstellen. Schließlich soll die Authentifizierung auch getestet werden. Am einfachsten geht dies über einen Profile-Hoster wie „my-profile.eu". Unter [3] gibt man lediglich einen frei wählbaren Namen und einen eindeutigen lokalen Benutzer-Namen an. Letzterer wird

Abbildung 1: WebID-Log-in bei my-profile.eu

Teil des URI. Nach einem Klick auf „New Profile" generiert der Browser ein Zertifikat und speichert es in der Zertifikatsverwaltung ab. Anschließend kann man sich mit einem Klick auf „WebID Log-in" einloggen (siehe Abbildung 1).

Wer sein Profil lieber selbst hosten möchte, schreibt einfach eine RDF/XML- oder Turtle-Datei und hinterlegt sie auf seinem Server. Wichtig dabei ist, dass der persönliche URI sich von dem des Dokuments unterscheidet, also beispielsweise durch ein „#me" ergänzt wird. Ein Zertifikat lässt sich beispielsweise über das Kommandozeilen-Tool „openssl" erzeugen. Der öffentliche Schlüssel wird anschließend, wie in der letzten Ausgabe beschrieben, in seinem Profil eintragen. Unter [4] gibt es ein fertiges Shell-Skript, das ein WebID-taugliches Zertifikat generiert und die zugehörigen RDF-Tripel ausgibt.

Authentifizierung delegieren

Die Authentifizierung an einen externen Dienst zu delegieren, hat den Vorteil, dass lediglich das API des Dienstes implementiert werden muss und nicht das komplette WebID-Protokoll. Der Nachteil ist, dass zusätzliche HTTP-Roundtrips entstehen und dem Dienst vertraut werden muss. Für den Einstieg ist dies jedoch die einfachste Variante. Wir werden im Folgenden den Dienst „foafssl.org" für die Authentifizierung heranziehen. Als Web-Framework kommt Grails zum Einsatz, die Prinzipien lassen sich jedoch leicht auf andere Web-Frameworks übertragen. Die Authentifizierung der Nutzer erfolgt in drei Schritten:

1. Weiterleitung des Nutzers zu „foafssl.org"
2. Auswertung der Antwort von „foafssl.org"
3. Verknüpfung des WebID-Nutzers mit der eigenen Anwendung

Im ersten Schritt werden die Besucher der Website also zu „foafssl.org" weitergeleitet – und zwar zu „https://foafssl.org/srv/idp?rs=<Antwort-URI>". Dabei ist „<Antwort-URI>" ein bestimmter URI der Web-Anwendung, der die Antwort von „foafssl.org" entgegennehmen und weiterverarbeiten kann. Die Nutzer werden bei Aufruf von „/webId/login" weitergeleitet und die Antwort von „foafssl.org" unter „/webId/verify" erwartet. Dazu legt man einen „WebIdController" mit den Actions „login" und „verify" an (siehe Listing 2).

Mittels „redirect()" wird bei Grails eine Weiterleitung eingeleitet. Den für den „rs"-Parameter benötigten absoluten URI erzeugen wir zuvor mit „createLink()". Er verweist auf die „verify"-Action. Die Antwort von „foafssl.org" erfolgt durch eine Weiterleitung an diesen „verify"-URI, inklusive der folgenden drei Parameter:

- *webid*
  Die WebID des authentifizierten Users
- *ts*
  Ein Zeitstempel, wann die Authentifizierung durchgeführt wurde
- *sig*
  Die Signatur von „foafssl.org"

Um sicherzustellen, dass die angegebene WebID tatsächlich korrekt authentifiziert wurde, müssen sowohl die Signatur als auch der Zeitstempel überprüft werden. Bei der Signatur handelt es sich um den uns bekannten „verify"-URI, ergänzt um die oben genannten Parameter „webid" und „ts", der mit dem RSA-Schlüssel von „foafssl.org" signiert wurde. Mit dem öffentlichen Schlüssel von „foafssl.org" können wir diese Signatur überprüfen. Den Schlüssel findet man ebenso wie die Dokumentation des Dienstes unter [5]. Java bietet alles, was wir zur Überprüfung der Signatur benötigen (siehe Listing 3).

In den ersten fünf Zeilen wird eine Public-Key-Instanz aus den öffentlichen Daten „MODULUS" und „PUBLIC_EXPONENT" erzeugt. In der sechsten Zeile rekonstruieren wir den URI, den „foafssl.org" signiert haben sollte. Mithilfe eines Signatur-Objekts prüfen wir anschließend, ob dies tatsächlich die Daten sind, die von „foafssl.org" signiert wurden. Dabei ist zu beachten, dass die Signatur „Base64-URL"-encodiert (im Gegensatz zur gewöhnlichen Base64-Encodierung werden hierbei die Zeichen „+" und „/" durch „–" und „_" ersetzt) ist und zunächst decodiert werden muss. Dies erledigt „Base64.decodeBase64()" aus Apache Commons für uns.

```
class WebIdController {
        def login() {
                def rs = createLink (
                        action: 'verify',
                        absolute: true
                ).encodeAsURL()
                redirect(url: "https://foafssl.org/srv/idp?rs=${rs}")
        }
        def verify() {
                // Signatur und Timestamp prüfen (Listings 3 und 4)
        }
}
```

Listing 2

Um Replay-Attacken zu erschweren, ist auch der Zeitstempel unbedingt zu überprüfen. Gelangt ein Angreifer zum Beispiel an den „verify"-URI von WebID-Nutzerin „Alice", könnte er sich damit jederzeit als „Alice" bei unserem Dienst einloggen. Der „verify"-URI sollte daher eine möglichst kurze Zeitspanne gültig sein. Allerdings darf diese Zeitspanne auch nicht zu kurz sein, da sonst gültige Requests abgewiesen werden, weil sie etwas länger dauern oder die Zeiten des „foafssl.org"-Servers und des von uns verwendeten Web-Servers zu stark voneinander abweichen. Hat man sich für eine angemessene Lebensdauer entschieden, fällt die Implementierung leicht (siehe Listing 4).

Der von „foafssl.org" übermittelte Zeitstempel entspricht dem Datumsformat „yyyy-MM-dd'T'HH:mm:ssZ" und kann entsprechend geparst werden. Anschließend muss er nur noch mit der aktuellen Uhrzeit verglichen werden. Ist die Differenz größer als die tolerierte Lebenszeit, muss die Authentifizierung scheitern.

Nachdem so sichergestellt wurde, dass die Antwort von „foafssl.org" sowohl korrekt als auch aktuell ist, kann die WebID beispielsweise genutzt werden, um den zugehörigen lokalen Account zu finden oder gegebenenfalls einen solchen anzulegen. Anschließend kann die Anwendung wie gewohnt arbeiten, als ob der Nutzer sich mit Benutzername und Passwort angemeldet hätte (siehe Listing 5 und Abbildung 2). Eine große Stärke von WebID liegt jedoch auch darin, dass über den URI weitere Informationen nachgeladen werden können, wie wir später noch sehen werden.

```
BigInteger MODULUS = new BigInteger("9093ac0285..." , 16);
BigInteger PUBLIC_EXPONENT = new BigInteger("65537", 10);
def publicKeySpec = new RSAPublicKeySpec(MODULUS, PUBLIC_EXPONENT);
KeyFactory keyFactory = KeyFactory.getInstance("RSA");
PublicKey publicKey = keyFactory.generatePublic(publicKeySpec);
byte[ ] data = "$verifyUrl?webid=${URLEncoder.encode(webId)}&ts=${URLEncoder.
encode(timestamp)}".getBytes()
Signature sig = Signature.getInstance("SHA1withRSA");
sig.initVerify(publicKey);
sig.update(data);
return sig.verify(Base64.decodeBase64(signatureBase64));
```

Listing 3

```
Date verifiedDate =
        new SimpleDateFormat(TIMESTAMP_FORMAT).parse(timestamp)
def differenceInMillis =
        new Date().getTime() - verifiedDate.getTime()
return (differenceInMillis / 1000) <= TIMESTAMP_LIFESPAN
```

Listing 4

```
session.user = webIdAccountService.loadOrCreateUserAccount(webId)
flash.message = "Successfully logged in with WebID $webId"
redirect (controller:'myAccount')
```

Listing 5

Abbildung 2: Externe WebID-Authentifizierung über „foafssl.org"

Authentifizierung selbst gemacht

Die Delegation der Authentifizierung an einen externen Dienst wie „foafssl.org" ist einfach zu realisieren, bringt jedoch auch Nachteile mit sich. Es sind mehrere HTTP-Weiterleitungen erforderlich, was den Prozess in die Länge zieht und noch nicht viel vom versprochenen Ein-Klick-Log-in erahnen lässt. Weiterhin ist es nicht immer angemessen, einem externen Dienst die Authentifizierung anzuvertrauen. Dieses Problem steht bereits der Verbreitung von OpenID im Wege. Bis heute sind viele OpenID-Provider trotz des offenen Standards nicht miteinander kompatibel, da sie sich gegenseitig nicht vertrauen. Bei WebID tritt dieses Problem nicht auf, da jeder Dienst die Authentifizierung selbst durchführen kann.

Mithilfe des Apache-Moduls „mod_authn_webid" lässt sich die Authentifizierung leicht auf dem eigenen Server konfigurieren. In folgendem Beispiel-Szenario läuft eine Grails-Anwendung auf einem Apache Tomcat. Ein Apache-httpd-Server ist mittels „mod_jk" als Reverse-Proxy vor diesen geschaltet. Die Authentifizierung wird der Apache-httpd übernehmen und das Ergebnis an Apache Tomcat weiterreichen.

Das Modul „mod_authn_webid" muss zunächst aus den Sourcen [6] kompiliert werden, was jedoch problemlos gelingen sollte, wenn die in der README-Datei genannten Voraussetzungen erfüllt sind. Listing 6 zeigt,

```
apt-get install librdf0-dev
apt-get install openssl
autoconf
./configure
make install
```

Listing 6

wie der Autor das Modul auf einem Ubuntu-System installieren konnte. Anschließend muss das Modul geladen und der Apache neu gestartet werden (siehe Listing 7).

Sobald das Modul verfügbar ist, kann ein virtueller Host für die Grails-Anwendung eingerichtet werden. Angenommen, die Anwendung soll über die Domain „webid.example.org" erreichbar sein und unter „/login" ein WebID-Log-in ermöglichen. Da WebID auf SSL aufsetzt, muss mindestens dieser Log-in-URI über „HTTPS" abgesichert sein. Um dies sicherzustellen, richten wir für diesen Pfad einen Redirect zur HTTPS-Variante ein (siehe Listing 8). Die Konfiguration für die HTTPS-Variante ist in Listing 9 ersichtlich.

```
echo 'LoadModule authn_webid_module /usr/lib/apache2/modules/mod_auth_webid.so' >> /etc/
apache2/mods-available/authn_webid.load
a2enmod authn_webid
service apache2 restart
```

Listing 7

```
<virtualhost *:80>
        ServerName webid.example.org
        Redirect permanent /login https://webid.example.org/login
        JkMount /* worker1
</virtualhost>
```

Listing 8

```
<virtualhost *:443>
        ServerName webid.example.org
        SSLEngine On
        SSLCertificateFile /etc/apache2/ssl/server.pem
        SSLOptions +StdEnvVars +ExportCertData
        SSLVerifyClient none
        <Location /login>
                SSLVerifyClient optional_no_ca
                SSLVerifyDepth 1
                AuthType WebID
                Require valid-user
        </Location>
        JkMount /* worker1
</virtualhost>
```

Listing 9

```
def login () {
        String webId = request.getRemoteUser ()
        if (webId){
                loadOrCreateUserAccount(webId)
                flash.message = "Successfully logged in with WebID $webId"
                redirect (controller: 'myAccount')
        } else {
                flash.message = 'Login failed.'
                session.user = null
                redirect (controller: 'home')
        }
}
```

Listing 10

```
def rdfLoader = new JenaRdfLoader ()
RdfResource person = rdfLoader.loadResource(webId)
String name = person(foaf.name)
String imgUri = person(foaf.img).uri
String cityUri = person(foaf.basedNear).uri
RdfResource city = rdfLoader.loadResource(cityUri)
String cityName = city(geo.name)
```

Listing 11

Die Option „SSLOptions +StdEnvVars +ExportCertData" ist wichtig, damit die Zertifikat-Informationen für Apache Tomcat und die Grails-Anwendung verfügbar sind. Über „ SSLVerifyClient none" wird grundsätzlich die Notwendigkeit zur Authentifizierung ausgeschaltet. Lediglich für den Log-in-URI wird im Location-Abschnitt die WebID-Authentifizierung eingeschaltet. „SSLVerifyClient optional_no_ca" stellt dabei sicher, dass alle gültige Zertifikate akzeptiert werden, unabhängig davon, ob sie von einer CA signiert und dem Server bekannt sind.

Mit „AuthType WebID" kommt das soeben installierte Modul ins Spiel. Der Browser fragt nun beim Aufruf des Log-in-URI nach einem Zertifikat und prüft, ob es sich um eine gültige WebID handelt. Falls eine Nutzerin erfolgreich authentifiziert werden konnte, steht deren URI anschließend in der Umgebungsvariable „REMOTE_USER".

In der Grails-Anwendung lässt sich diese bequem über „request.getRemoteUser()" abfragen. Ist sie gesetzt, liegt ein authentifizierter WebID-Nutzer vor, der beispielsweise mit einem lokalen Account verbunden werden kann. Ist die Variable leer, ist das Log-in fehlgeschlagen (siehe Listing 10).

Das war´s dann. Ein Klick auf den Log-in-Link führt nun dazu, dass der Browser um die Auswahl eines Zertifikats bittet. Nachdem das WebID-Zertifikat ausgewählt ist, ist man eingeloggt. Da sich der Browser das ausgewählte Zertifikat merkt, ist es nun bei allen Diensten im Web, die WebID unterstützen, möglich, sich mit einem einzigen Klick anzumelden.

Daten bitte

Über den WebID-URI können wir nicht nur Nutzer global identifizieren, sondern auch zusätzliche Informationen laden. Denn schließlich verbirgt sich dahinter ein gegebenenfalls sehr komplexes, maschinenlesbares FOAF-Profil. In der vorletzten Ausgabe haben wir mithilfe der Bibliothek

„groovyrdf“ [7] bereits Daten über Hotels aus dem Web-of-Data geladen. Genauso können wir auch mit Informationen über Personen verfahren (siehe Listing 11).

Neben einfachen Literalen wie dem Namen lassen sich auch verlinkte Ressourcen abrufen und Informationen über diese nachladen. In dem Beispiel-Listing wird der Link zum Ort der Person verfolgt und dessen Name abgerufen. Da sich die Profil-Informationen beliebig zusammensetzen können und nicht unter Kontrolle der Anwendung liegen, sollte die Abfrage der Daten möglichst tolerant erfolgen und fehlende Informationen sollten verschmerzbar sein. Möglicherweise enthält ein Profil beispielsweise gar kein „foaf:basedNear“-Prädikat oder dieses enthält Geo-Koordinaten, anstatt auf eine andere Ressource zu verlinken. Die Profil-Informationen sind außerdem ebenso vorsichtig zu behandeln wie gewöhnliche Nutzer-Eingaben. Wird der Name einer Person achtlos in die HTML-Seite einbettet, ist die Anwendung zum Beispiel für Cross-Site-Scripting (XSS) anfällig.

Um tatsächlich ein globales, dezentrales Social-Network aufzubauen, ist es sehr zu empfehlen, dass man die URIs seiner Nutzer mit den Inhalten der Anwendung verlinkt. Ermöglicht es die Anwendung beispielsweise, Nutzer-Inhalte zu „liken“, sollten man diese Information auch selbst wieder als RDF-Tripel veröffentlichen oder ein semantisches Pingback [8] senden, um das „Social Web of Data“ zu erweitern. Veröffentlicht man alle Inhalte auch als Linked Data, haben andere Dienste die Möglichkeit, auf sie zu verlinken.

Fazit und Ausblick

Mithilfe von WebID bietet man Nutzern eine bequeme Log-in-Möglichkeit mit einem bestehenden und gegebenenfalls sogar selbst-gehosteten Profil. Selbst wenn die Nutzer noch keine WebID besitzen, kann man sie zur Profil-Anlage und -Verwaltung leicht an einen externen Dienst wie „myprofile.eu“ verweisen und dennoch die Authentifizierung nicht aus der Hand geben.

Es ist auch problemlos möglich, WebID als Ergänzung zu etablierten Log-in-Möglichkeiten anzubieten. Mit dem Aufkommen neuer WebID-tauglicher Dienste wird die Verbreitung von WebID weiter steigen und sich der Nutzen für die Besitzer einer WebID und die Betreiber von Diensten vervielfachen. Je mehr Dienste und Nutzer nach diesem Prinzip das Netz verwenden, desto stärker wird die Verlinkung von Daten und sozialen Interaktionen. Letztlich wird die Summe an kleinen, vernetzten Diensten im Social-Web die Möglichkeiten eines zentralisierten sozialen Netzwerks weit übersteigen.

Referenzen

[1] https://datenwissen.de/hintergrundinfos/zeitschriftenartikel/
[2] http://www.w3.org/2005/Incubator/webid/spec/
[3] https://my-profile.eu/profile
[4] https://gist.github.com/njh/2432427
[5] https://foafssl.org/srv/idp
[6] https://github.com/linkeddata/mod_authn_webid
[7] http://angelo-v.github.io/groovyrdf/
[8] http://www.w3.org/wiki/Pingback#Semantic_Pingback

Angelo Veltens
angelo.veltens@online.de
http://datenwissen.de

Angelo Veltens studierte Angewandte Informatik an der Dualen Hochschule Baden-Württemberg Karlsruhe und befasste sich in Studienarbeiten und Abschlussarbeit mit Linked Data und semantischem Wissensmanagement. Heute ist er als Software-Entwickler in Braunschweig tätig, arbeitet in seiner Freizeit am „Social Web of Data“ und referiert auf Konferenzen zu diesem Thema.

Leben Sie schon oder programmieren Sie noch UIs?

Jonas Helming, EclipseSource München GmbH

Business-Anwendungen sind häufig auf die Ein- und Ausgabe von Daten fokussiert. Zu diesem Zweck sind Formular-basierte Oberflächen notwendig, um die verwalteten Entitäten der Anwendung anzuzeigen und die verfügbaren Attribute einer oder mehrerer Entitäten zur Bearbeitung anzubieten.

Das manuelle Programmieren von Formular-basierten Oberflächen und besonders das Erstellen von Layouts sind aufwändig und erzeugen inhomogene Ergebnisse. Während der Kunde mit einfachen Konzepten wie Feldern, Spalten oder Attribut-Gruppen spezifiziert, muss sich der Entwickler durch die mächtigen, aber auch komplexen Layout-Manager aktueller UI-Frameworks quälen. Da jede Anforderung erst langwierig umgesetzt werden muss, kann der Kunde erst spät Feedback geben – jede weitere Änderung kann erneute aufwändige Entwicklung erfordern.

Bekannte UI-Editoren ermöglichen zwar im Gegensatz zu handgeschriebenem Code ein früheres Feedback. Wer damit professionelle, skalierbare und vor allem funktionale Oberflächen erstellen will, wird früher oder später trotzdem den erzeugten Code erweitern und anpassen müssen. Insbesondere ist die entstehende Oberfläche meist noch manuell an Entitäten anzubinden, selbst implementierte Controls sind häufig nicht unterstützt. Die in Business-Anwendungen typischerweise geforderte Homogenität der Oberflächen stellen diese Ansätze nicht sicher. Am Ende erzeugt jede entstandene Zeile Code, egal ob geschrieben oder generiert, dauerhaft hohe Wartungskosten, insbesondere wenn Änderungen am Layout gewünscht sind, die alle Formulare einer Anwendung betreffen.

Dieser Artikel beschreibt einen Ansatz, in dem Formular-basierte Oberflächen automatisch gerendert werden können, ohne eine Zeile Code zu schreiben. Die Oberflächen verbinden sich ohne manuelles Zutun mit den entsprechenden Entitäten und erlauben damit vom ersten Projekttag an die Ein- und Ausgabe und sogar Validierung von Daten. Anpassungen am Layout der Oberflächen werden Tool-gestützt mit einem einfachen Modell ausgedrückt. Dieses reduziert dabei die Komplexität der Layout-Technologien auf das benötigte Maß an Mächtigkeit und ist idealerweise sogar für den Kunden zu verstehen.

Die eigentlichen Formulare werden erst zur Laufzeit von einem Renderer gezeichnet. Dieser setzt die in der Anwendung unterstützten und erlaubten Layout-Konzepte zentral um und stellt damit eine Homogenität der Oberfläche sicher. Sind im Layout-Konzept Änderungen notwendig, beispielsweise das Anpassen von Zeilenabständen auf neue Bildschirme, ist lediglich der Renderer an zentraler Stelle zu modifizieren.

Der Ansatz ist effektiver, spart viele Zeilen Code und ermöglicht es, mit wenigen Klicks die Oberfläche einer Anwendung an neue Anforderungen, geänderte Entitäten und sogar neue Oberflächen-Technologien anzupassen.

Der Artikel bezieht sich auf das Open Source Framework „EMF Client Platform", das den technologischen Kern sowie produktiv einsetzbare Modelle und Renderer bereitstellt. Es lässt sich in beliebige Java-Anwendungen integrieren und ist um eigene Komponenten erweiterbar.

Der Artikel zeigt zunächst im Abschnitt „Kein Aufwand", wie ohne jeglichen UI-Code und zunächst ohne zusätzliches Modell voll automatisch eine erste funktionierende Oberfläche erstellt werden kann. Im Anschluss wird diese Oberfläche um eigene Elemente erweitert und schließlich ein „Eigenes Layout" definiert. Zuletzt kommen das für die View-Modellierung unterstützende Tooling sowie die Möglichkeiten der Anpassung der EMF Client Platform an eigene Anforderungen.

Vollautomatik

Business-Anwendungen bauen häufig auf einem zu definierenden Datenmodell auf. Dieses beschreibt die Entitäten, die mit der Anwendung verwaltet werden können. Das einfachste denkbare User-Interface stellt alle als öffentlich markierten Attribute des Datenmodells in einem Formular dar und erlaubt dem Benutzer, diese Attribute in entsprechenden Controls zu editieren. Gerade in der Anfangsphase eines Projekts ist das genaue Layout der Oberfläche oft noch nicht festgelegt, der Kunde möchte aber dennoch gern eine erste Version der Anwendung „anfassen" können. Für diese erste Demonstration sind nun ohne Framework-Unterstützung bereits etliche Zeilen UI-Code zu schreiben. Diese Tatsache verlängert von Beginn an den sogenannten „Turnaround", also die Zeit, nach der eine Änderung für den Kunden sichtbar gemacht werden kann. Das ist besonders bei der Definition der durch die Anwendung verwalteten Entitäten kritisch. Ob ein Feld in einem Datenbank-Schema fehlt, können meist nur eher technisch versierte Ansprechpartner beantworten. Wenn jedoch ein Textfeld in einer Eingabemaske fehlt, werden das auch andere Anwender des Systems schnell feststellen.

Technisch gesehen ist für eine erste einfache Version einer Formular-basierten Oberfläche meist wenig bis keine zusätzliche Information notwendig. Die Information darüber, welche Entitäten und welche zugehörigen Attribute es in der Anwendung gibt, wird in den meisten Fällen bereits außerhalb des User Interface festgelegt, beispielsweise in einem Datenbank-Schema. Genau diese Information macht sich die EMF Client Platform für das Anzeigen von Entitäten in einem Formular-basierten Editor zunutze.

Im Folgenden soll der Ansatz anhand eines einfachen Beispiels vorgestellt werden. Das Beispiel definiert eine Entität „User" mit folgenden drei Attributen:

- First Name: String
- Last Name: String
- E-Mail-Adresse: Liste von Strings

Das Beispiel ist bewusst einfach gehalten, der Ansatz spielt seine Stärken natürlich insbesondere bei einer großen Anzahl von Entitäten und Attributen voll aus. Wird nun eine Entität vom Typ „User", konkreter eine Instanz der Java-Klasse „User", mit dem Editor der EMF Client Platform geöffnet, wertet dieser die Entität reflexiv zur Laufzeit aus und zeichnet eine erste Version einer Formular-basierten Oberfläche (siehe Abbildung 1). Für die beiden String-Attribute wurden jeweils ein Textfeld und links daneben ein Label mit dem Attribut-Namen gerendert. Das dritte Attribut wird mit einem Multi-String-Control angezeigt, um das Hinzufügen, Bearbeiten, Sortieren und Löschen von E-Mail-Adressen des Users zu ermöglichen. Die Oberfläche ist bereits voll funktional, das bedeutet, Änderungen werden in die Entität übertragen. Zu diesem Zweck wird jedes Control an ein oder mehrere Attribute automatisch angebunden. Die EMF Client Platform bietet für jeden einfachen Datentyp sowie für Referenzen ein entsprechendes Standard-Control an.

Diese erste Version der Oberfläche ist natürlich in den allermeisten Fällen nicht die endgültige Lösung. Die EMF Client Platform erlaubt sowohl die Anpassung der verwendeten Controls als auch die des Layouts. Die erste Version der Oberfläche, wenn auch noch nicht perfekt, bietet zwei entscheidende Vorteile: Sie erfordert nicht eine Zeile UI-Code und kann dem Kunden direkt nach der Definition der angezeigten Entitäten präsentiert werden. Ergeben sich daraus Änderungswünsche, entdeckt der Kunde beispielsweise ein fehlendes Attribut, kommt der dritte entscheidende Vorteil des Ansatzes zum Tragen. Durch das reflexive Auslesen der Informationen aus den Entitäten ist die EMF Client Platform robust gegen Änderungen.

Soll nun beispielsweise auf Kundenwunsch der Entität „User" ein neues Attribut hinzugefügt werden, das in einer Enumeration das Geschlecht eines Users angibt, ist dieses lediglich der Entität hinzuzufügen. Nach einem erneuten Öffnen des Formular-basierten Editors der EMF Client Platform ist das Attribut sichtbar und wird als Enumeration mit einer Drop-Down-Box gerendert. Wäre die Oberfläche hingegen manuell implementiert, wäre bei jeder Änderung an den Entitäten eine manuelle Anpassung aller Oberflächen notwendig gewesen, die die Entität anzeigen, und das, bevor sie durch den Kunden abschließend als sinnvoll beurteilt werden kann. Ist die Änderung am Ende doch unerwünscht, sind die entstandenen Kosten verloren.

In Business-Anwendungen müssen neben der reinen Anzeige und Bearbeitung von Attributen typischerweise die Eingaben des Benutzers validiert werden. Im Beispiel könnten die Attribute „FirstName" und „LastName" Pflichtfelder sein, die also zwingend die Eingabe eines Wertes erfordern. Auch diese Anforderung lässt sich zu Beginn ohne manuelle Entwicklungsschritte abdecken. Dazu muss allerdings die Information über die zu überprüfenden Regeln vorliegen. Ähnlich wie die Liste der verfügbaren Attribute ist die Information über den gültigen Wertebereich eines Attributs meist schon bei der Definition der Entitäten, beispielsweise in Form eines Datenbank-Schemas, festgelegt worden. Die EMF Client Platform kann diese Informationen auslesen und entsprechend nutzen. Gibt die Entität „User" aus dem Beispiel an, dass das Feld „LastName" nicht leer sein darf, zeigt das Text-Control der EMF Client Platform bei leerer Angabe einen entsprechenden Fehler an (siehe Abbildung 3).

Alle Standard-Controls können derartige Eingabe-Validierungen anzeigen. In vielen Fällen wird man aber früher oder später zusätzlich eigene, stark angepasste „Controls" für bestimmte Attribute verwenden wollen. Wie das geht, zeigt der folgende Absatz.

Eigene Controls

Der vorgestellte, reflexive Ansatz erzeugt basierend auf Entitäten schnell und ohne zusätzlichen Aufwand eine erste Formular-basierte Oberfläche. Dazu stellt die EMF Client Platform für jeden Grund-Attribut-Typ wie beispielsweise für Strings ein Standard-Control bereit. Dieses erlaubt die Bearbeitung des Attributs und unterstützt sogar dessen Validierung. Es wird allerdings in den meisten Projekten mindestens einige Fälle geben, in denen die Funktionalität dieser Standard-Controls auf Dauer nicht ausreichend ist. Trotzdem muss nicht auf die Vorzüge des automatischen Aufbaus der Oberfläche verzichtet werden. Die EMF Client Platform unterstützt die Verwendung eigener Controls. Diese werden registriert und anschließend vom Formular-basierten Editor verwendet. Dabei kann genau angegeben werden, für welche Attribute ein bestimmtes neues Control verwendet werden soll. Es lassen sich einfach Zuordnungen treffen, beispielsweise dass ein eigenes Control für alle String-Attribute oder nur für ein ganz bestimmtes Attribut wie „LastName" verwendet wird. Aber auch komplexe Regeln, etwa ein Control für Zahlenfelder länger als zehn Stellen, sind einfach zu realisieren.

Noch einfacher ist es in vielen Fällen, existierende Controls um fehlende Funktionen zu erweitern. Abbildung 4 zeigt

Abbildung 1: Bereits ohne jegliche Zusatz-Informationen kann die EMF Client Platform eine Formular-basierte Oberfläche für eine Entität rendern

Abbildung 2: Neue Attribute werden ohne zusätzlichen Aufwand in der Oberfläche angezeigt

Abbildung 3: Die verwendeten Standard-Controls unterstützen eine Eingabe-Validierung

Abbildung 4: Die EMF Client Platform erlaubt die Verwendung von eigenen und angepassten Controls

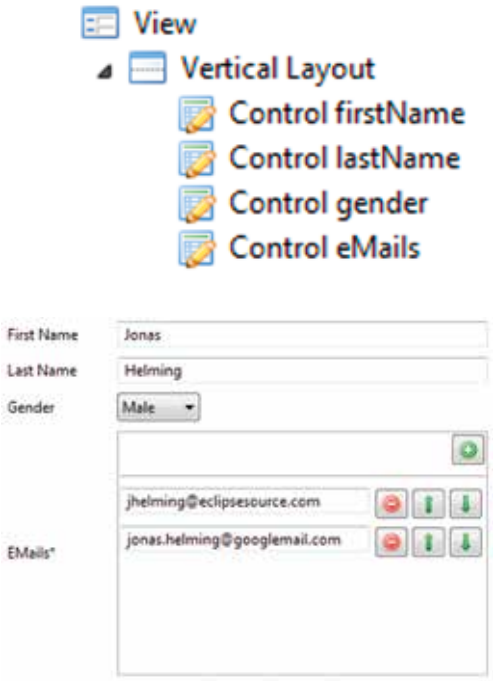


Abbildung 5: Das Modell (oben, nicht für den User sichtbar) und unten die resultierende Oberfläche

eine solche Erweiterung. Neben dem Textfeld, das die Mail-Adresse anzeigt, wurde ein Button eingefügt, um direkt eine Mail zu versenden. Entsprechend registriert erscheint der Button neben allen E-Mail-Attributen. Auf diese Weise können in kleinen Schritten einzelne Controls erweitert oder ersetzt werden.

Die zentrale Verwaltung der Controls stellt sicher, dass einmal implementierte Controls leicht auffindbar bleiben und für alle gleich zu behandelnden Attribute wiederverwendet werden. Dies sorgt damit für ein homogenes Anwendungsbild und wenig zu wartenden Code.

Das punktuelle Erweitern und Ersetzen einzelner Controls fokussiert den notwendigen Aufwand von Beginn an nur auf die Änderungen, die wirklich notwendig sind. Für alle unangepassten Attribute sind weiterhin die Standard-Controls verfügbar. Das bedeutet, dass die Anwendung kontinuierlich benutz- und demonstrierbar bleibt. Selbst wenn bei Fertigstellung der Anwendung die meisten existierenden Controls durch eigene Implementierungen ersetzt wurden, wird man keine Zeile Code entwickelt haben, die nicht ohnehin notwendig gewesen wäre. Im Gegenteil, durch die bessere Wiederverwendung von Controls und das vollständige Wegfallen des manuellen Layouts schrumpft die zu entwickelnde und zu pflegende Code-Basis massiv. Weiterhin bleibt die Oberfläche robust gegen Änderungen an den Entitäten. Wird beispielsweise ein neues String-Attribut hinzugefügt, wird ohne weiteres Zutun das Control für String-Attribute gezeichnet, egal, ob es das Standard-Control oder ein eigenes ist.

Eigenes Layout

Durch den bisher beschriebenen Ansatz ist es mit sehr wenig Aufwand und kaum manueller Entwicklung möglich, erste Formular-basierte Oberflächen aufzubauen. Diese stellen alle verfügbaren Attribute in einer Art Liste dar. Durch das Austauschen einzelner Controls kann das Aussehen und Verhalten einzelner Attribute angepasst werden. Damit bekommt der Kunde bereits einen sehr guten Eindruck von der laufenden Anwendung und kann sie sogar schon benutzen. Trotzdem entwickelt sich in den allermeisten Projekten recht bald die Anforderung nach einem angepassten, komplexeren Layout, das die einfache Liste an Controls ersetzt. Das ist typischerweise der Zeitpunkt, an dem sich Entwickler mit vielen Zeilen Code durch die Verwendung von Layout-Manager, vertikalen Ausdehnungen, maximalen Höhen und relativen Positionen quälen muss. Und das alles oft nur, damit der Kunde nach einer Demonstration am Ende doch alles ganz anders haben möchte. Außerdem geht durch die manuelle Implementierung des Layouts die Robustheit gegen Änderungen verloren, da jedes Control einen expliziten Platz im Layout zugewiesen bekommt.

Die EMF Client Platform geht auch hier einen radikal anderen Weg. Der Ansatz verzichtet selbst für das finale Layout vollständig auf manuellen Code. Stattdessen sind die Layout-Informationen in einem einfachen Modell spezifiziert, das den gewünschten Aufbau der Oberfläche beschreibt. Ein View-Modell besteht aus Layout-Informationen sowie aus Controls selbst, die an bestimmter Stelle im Layout platziert werden. Controls sind wie zuvor immer an bestimmte Attribute angebunden und erlauben deren Ein- und Ausgabe.

Die Erstellung des Modells wird von einem leicht zu verwendenden Tool unterstützt. Das fertige Modell wird schließlich zur Laufzeit von einem Renderer interpretiert.

Ein sehr einfaches Beispiel für ein Layout ist das bereits zuvor beschriebene Listen-Layout, in dem jedes Control in einer eigenen Zeile dargestellt ist. Das Layout ist zwar, wie zuvor beschrieben, ohne jegliche Zusatzschritte bereits automatisch verfügbar, im Folgenden wird jedoch als erstes einfaches Beispiel beschrieben, wie genau dieses Layout Modell-basiert ausgedrückt werden kann. Für die Darstellung von Elementen in einer vertikalen Liste unterstützt das UI-Modell das Element Vertical-Layout. Das Layout stellt alle enthaltenen Elemente untereinander als Zeilen dar. Die enthaltenen Elemente sind in diesem einfachen Beispiel direkt die gewünschten Controls.

Die EMF Client Platform bietet einen Editor für die IDE an, mit dem das Modell der Oberfläche komfortabel bearbeitet werden kann, mehr dazu im folgenden Abschnitt „Mehr Tooling". Ein View-Modell besteht immer aus einem Root-Knoten (View), der per Kontextmenü mit Unterelementen befüllt wird. Abbildung 5 zeigt das Modell, das das bisher automatisch erstellte Layout explizit ausdrückt. Wie in der Abbildung zu erkennen, kann das Modell bereits in der IDE live in einem Preview gerendert werden. Dadurch sieht der Oberflächendesigner direkt, welches Ergebnis er zu erwarten hat, ohne die Anwendung dazu erst starten zu müssen. Dieses frühe Feedback vereinfacht das Erstellen eines korrekten Layouts zusätzlich.

Das so erstellte View-Modell (Layout) wird nun als Datei abgespeichert. Diese ist einem bestimmten Typ von Entität zugeordnet. Zur Laufzeit liest der Editor der EMF Client Platform beim Öffnen einer Entität die entsprechende Datei ein und rendert die Oberfläche. Ist für eine Entität kein View-Modell angegeben, wird weiterhin das Default-Layout verwendet. Das Lay-

Abbildung 6: Das Modell (links, nicht für den User sichtbar) und rechts die resultierende Oberfläche

out der Oberflächen kann also in kleinen Schritten iterativ spezifiziert werden.

Durch den Modell-basierten Ansatz lässt sich das Modell nun einfach verfeinern und damit ein besser angepasstes Oberflächen-Layout erstellen. Im bisherigen Layout wird gerade auf breiten Bildschirmen viel Platz verschwendet. Eine einfache Möglichkeit, den Platz besser zu nutzen, wäre, nicht nur eine, sondern zwei vertikale Reihen als parallele Spalten aufzubauen. In diesem Layout (siehe Abbildung 6) gibt es also zwei „VerticalLayout"-Elemente. Da diese beiden Container horizontal nebeneinander angeordnet werden, sind sie in einem Horizontal-Layout enthalten, das alle Kind-Elemente nebeneinander darstellt.

Auch das zweite Beispiel ist im Vergleich zu den Formularen echter Business-Anwendungen immer noch sehr einfach gehalten. Natürlich existieren neben den beiden beschriebenen Layout-Typen weitere Layouts (wie GridLayout) sowie Layout-Elemente (wie Separatoren). Darüber hinaus können Layouts mit Labels und Rahmen versehen werden, die bestimmte Gruppen von Attributen kennzeichnen. Durch den Modell-basierten Ansatz behält man auch komplexe Layouts gut im Griff.

Mehr Tooling

Der Modell-basierte Ansatz, ein Layout zu erstellen, ist nicht nur einfacher und effektiver. Durch das bessere Abstraktions-Niveau ist es, anders als in Technologie-abhängigem Source Code, möglich, nützliches Tooling bereitzustellen. Ein Beispiel dafür ist die bereits beschriebene Preview, die das aktuelle Layout live anzeigt, ohne dafür die gesamte Anwendung starten zu müssen. Aber auch bei der Modellierung selbst gibt es Unterstützung. Es wäre beispielsweise ein recht großer manueller Aufwand, für viele Attribute manuell Controls an einer bestimmten Stelle im bestehenden Layout zu erzeugen, die diese in der Oberfläche repräsentieren.

Da die verfügbaren Attribute jedoch bekannt sind, bietet das Tooling der EMF Client Platform einen Wizard an, mit dem in einem ausgewählten Container alle in einer Checkbox angebotenen Attribute als Controls erzeugt werden. Dabei lassen sich auch bereits vorhandene Controls ausblenden, meistens soll ja ein Attribut nur einmal pro Formular angeboten werden.

In dem beschriebenen, zweispaltigen Layout lassen sich so beispielsweise zunächst per Auswahl die Attribute für die erste Spalte erzeugen und anschließend mit einem Klick alle weiteren in der zweiten Spalte platzieren. Im Anschluss korrigiert man die Aufteilung leicht per „Drag & Drop". Schlussendlich lässt sich per Mausklick sogar validieren, ob alle Attribute ihren Platz im Layout gefunden haben; gerade bei einer großen Anzahl von Attributen ist eine manuelle Überprüfung auf Vollständigkeit äußerst mühsam. Die Überprüfung ist insbesondere bei Änderungen an den Entitäten nützlich, wenn also potenziell neue Attribute hinzugekommen sind, die erst noch platziert werden müssen.

Gerade bei Änderungen an Entitäten wird auch klar, dass, sobald ein explizites Layout angegeben wird, die Robustheit der automatisch berechneten Lösung verloren geht. Neue Attribute werden nicht mehr ohne weiteres Zutun in die Oberfläche aufgenommen. Umgekehrt ist das nach Erstellung eines expliziten Layouts auch meistens nicht mehr wünschenswert, neue Attribute müssen ja einen passenden Platz zugewiesen bekommen. Es gibt jedoch gerade für frühere Phasen eines Projekts, in denen noch viele Änderungen auftreten, aber schon erste Layouts definiert sind, eine sinnvolle Zwischenlösung. Die EMF Client Platform bietet eine Schnittstelle für das sogenannte „Postprocessing" von View-Modellen an. Ein bereits mitgelieferter Processor untersucht auf Wunsch ein existierendes Layout. Alle noch nicht explizit im Layout platzierten Attribute werden unten an das Layout angefügt. Dort sind sie zunächst einmal ohne weiteres Zutun sofort sicht- und benutzbar, bis sie ihren expliziten Platz im Layout erhalten.

Nicht zuletzt ist es auch möglich, programmatisch auf das View-Modell zuzugreifen, beispielsweise mit der eben beschriebenen Prozessor-Schnittstelle. Die Komplexität des zu verwendenden API folgt dabei der Vorgabe des entsprechenden Toolings, es ist immer noch deutlich einfacher zu benutzen als die Layout-Manager typischer UI-Frameworks. So lässt sich beispielsweise der gerade beschriebene Processor so erweitern, dass er ein zweispaltiges Layout automatisch gleichmäßig mit Attributen auffüllt.

Adaption

Das beschriebene Framework ist Teil des Open-Source-Frameworks EMF Client Platform. Alle beschriebenen Kernkonzepte, Renderer-Implementierungen für SWT und für die Webanwendungs-Technologie RAP sowie die Anbindung von EMF-Objekten als Entitäten sind Open Source unter der Eclipse-Public-License verfügbar und damit bestens für eine kommerzielle Verwendung geeignet. Die entstehenden Oberflächen sind an beliebiger Stelle in die eigene Anwendung integrierbar. Dazu lassen sich automatisch erstellte oder modellierte Oberflächen in eigene Fenster als Teil einer Master-Detail-Ansicht oder eingebettet in Webseiten rendern. Die EMF Client Platform stellt dafür entsprechende Schnittstellen bereit.

Das Framework ist bewusst sehr modular aufgebaut und bietet zahlreiche Anpassungsmöglichkeiten. So lassen sich beispielsweise die existierenden Renderer erweitern und anpassen. Damit kann das aus den View-Modellen gerenderte Layout vollständig auf das im eigenen Projekt gewünschte „Look and Feel" angepasst werden. Ändern sich beispielsweise durch neue Endgeräte die Anforderungen an das Aussehen der Anwendung, ist lediglich der zentrale Renderer anzupassen.

Abgesehen vom Renderer selbst wird der Entwickler vollständig von der Aufgabe des manuellen Layoutings befreit, gleichzeitig wird durch den zentralen Renderer ein anwendungsweites homogenes Layout sichergestellt. Dieses kann sogar durch eigene Validierungen auf dem View-Modell sichergestellt werden, beispielsweise indem eine maximale Spaltenzahl für Formulare definiert wird.

Weiterhin lassen sich leicht neue Renderer hinzufügen, etwa für JavaFX, für Swing oder gar für ein proprietäres Endgerät. Einige Renderer befinden sich bereits in der Entwicklung, eine Anfrage bei den Projekt-Entwicklern lohnt sich also. Das View-Modell selbst ist dabei unabhängig vom eigentlichen Renderer; es ist also möglich, das gleiche Layout für technisch unterschiedliche Oberflächen zu verwenden. Die Umsetzung eines Renderers erfordert deutlich weniger Aufwand, als man denken möchte, insbesondere wenn Vorerfahrungen mit bereits implementierten Renderern existieren. Alle Konzepte werden nur genau einmal implementiert, mögliche Layout-Fehler können zentral behoben werden. In der Praxis zahlt sich selbst die vollständige Implementierung eines neuen Renderers bereits nach wenigen erstellten Oberflächen im Vergleich zur manuellen Implementierung aus.

Nicht zuletzt ist es häufig sinnvoll, ein existierendes View-Modell anzupassen, zu erweitern oder sogar zu ersetzen. Das liegt im Wesentlichen in der Tatsache begründet, dass es für die Modellierung von Oberflächen keinen Universal-Ansatz geben kann, der für alle Projekte perfekt geeignet ist. Der beschriebene Ansatz vereinfacht das Erstellen von Oberflächen enorm, zum einen durch besseres Tooling und die Render-Architektur, zum anderen aber auch ganz bewusst durch die Einschränkung von Möglichkeiten und damit die Verringerung der Komplexität. Damit das gut funktioniert, müssen die Möglichkeiten idealerweise genau den in einem Projekt benötigten Konzepten entsprechen.

Kunden definieren ihre Anforderungen an die Oberfläche typischerweise nicht in Begriffen wie „GridLayout" oder „VerticalSpan", sondern in für jedermann verständlichen Konzepten wie „Gruppen" oder „Zeilen". Die im UI-Modell benötigten Konzepte unterscheiden sich natürlich von Projekt zu Projekt. Das gilt zum einen für die benötigte Mächtigkeit, also beispielsweise dafür, welche Layout-Arten zu unterstützen sind. Zum anderen kann das aber auch die Art der Modellierung betreffen, zum Beispiel die Namen bestimmter Elemente oder deren Zusammenspiel. Die optimale Lösung hängt dabei sehr davon ab, welche Zielgruppe das UI-Modell am Ende nutzt, um die Oberflächen zu designen. Während Programmierer Konzepte wie Horizontal-Layout leicht verstehen, kann die Fachabteilung mit „Horizontal Attribute Group" möglicherweise deutlich mehr anfangen. Wird das View-Modell am Ende gar so gewählt, dass es der Kunde selbst verwenden kann, ist eine maximale Verkürzung der Feedback-Schleife gelungen.

Da es sich bei der vorgestellten Implementierung um ein leicht erweiterbares Framework handelt, das im Kern vollständig aus Open-Source-Komponenten besteht, können alle beschriebenen Anpassungen effektiv durchgeführt werden. Dienstleister haben sich bereits auf genau diese Anpassungen spezialisiert und verfügen über entsprechende Erfahrungen aus verschiedenen Projekten. Gerade bei der Umsetzung oder Anpassung von View-Modellen und Renderern können Projekte hier von Querschnittserfahrungen profitieren.

Fazit

Formular-basierte Oberflächen folgen typischerweise einem gemeinsamen, homogenen Schema. Dies ist besonders für eine gute Usability eine wichtige Anforderung. Die von den existierenden UI-Frameworks wie Swing, JavaFX oder SWT angebotenen Konzepte sind für diese Anforderungen zu mächtig und damit zu komplex. Die manuelle Programmierung von Oberflächen ist daher aufwändig und gerade bei Änderungen an den Entitäten oder dem Layout-Konzept wartungsintensiv.

Der vorgestellte Ansatz der EMF Client Platform erlaubt es, einfache Layouts vollständig automatisiert zu erstellen. Anpassungen am Layout werden Tool-unterstützt in einem speziell für diesen Zweck entwickelten View-Modell ausgedrückt. Das Modell wird zu Laufzeit von einem Renderer interpretiert. Der Ansatz verringert den Aufwand und sorgt für sehr kurze Feedback-Schleifen. Die Anwendung bleibt kontinuierlich demonstrier- und benutzbar. Weiterhin stellt das Vorgehen die Homogenität der Oberfläche sicher und verringert massiv die zu wartenden Zeilen an Code. Einzelne Teile des Frameworks wie verwendete Controls, das View-Modell und die verfügbaren Renderer sind erweiter- und austauschbar. Damit ist das Framework auf beinahe beliebige Technologien und Einsatz-Szenarien übertragbar.

Am Ende ist der Ansatz natürlich kein Allzweckmittel. Vielmehr spielt er seine Stärken in einem klar definierten Umfeld aus: Business-Anwendung mit homogen zu gestaltenden Formularen zur Ein- und Ausgabe von Daten. In diesem Szenario zeigen sich durch den schnellen Einstieg und die iterative Vorgehensweise sehr früh spürbare Vorteile. Eine ausführliche Schritt-für-Schritt-Anleitung, um den Ansatz selber auszuprobieren, steht unter „http://emfcp.org".

Jonas Helming
jhelming@eclipsesource.com

Maximilian Kögel
mkoegel@ eclipsesource.com

Jonas Helming und Maximilian Kögel sind Eclipse-EMF/RCP-Trainer und Consultants sowie Geschäftsführer der EclipseSource München GmbH. Sie leiten die Eclipse-Projekte „EMFStore" und „EMF Client Platform".

Die iJUG-Mitglieder auf einen Blick

Java User Group Deutschland e.V.
http://www.java.de

DOAG Deutsche ORACLE-Anwendergruppe e. V.
http://www.doag.org

Java User Group Stuttgart e.V. (JUGS)
http://www.jugs.de

Java User Group Köln
http://www.jugcologne.eu

Java User Group Darmstadt
http://jugda.wordpress.com

Java User Group München (JUGM)
http://www.jugm.de

Java User Group Metropolregion Nürnberg
http://www.source-knights.com

Java User Group Ostfalen
http://www.jug-ostfalen.de

Java User Group Saxony
http://www.jugsaxony.org

Sun User Group Deutschland e.V.
http://www.sugd.de

Swiss Oracle User Group (SOUG)
http://www.soug.ch

Berlin Expert Days e.V.
http://www.bed-con.org

Java Student User Group Wien
www.jsug.at

Java User Group Karlsruhe
http://jug-karlsruhe.mixxt.de

Java User Group Hannover
https://www.xing.com/net/jughannover

Der iJUG möchte alle Java-Usergroups unter einem Dach vereinen. So können sich alle Java-Usergroups in Deutschland, Österreich und der Schweiz, die sich für den Verbund interessieren und ihm beitreten möchten, gerne unter office@ijug.eu melden.

Impressum

Herausgeber:
Interessenverbund der Java User Groups e.V. (iJUG)
Tempelhofer Weg 64, 12347 Berlin
Tel.: 030 6090 218-15
www.ijug.eu

Verlag:
DOAG Dienstleistungen GmbH
Fried Saacke, Geschäftsführer
info@doag-dienstleistungen.de

Chefredakteur (VisdP):
Wolfgang Taschner, redaktion@ijug.eu

Redaktionsbeirat:
Ronny Kröhne, IBM-Architekt;
Daniel van Ross, NeptuneLabs;
Dr. Jens Trapp, Google; André Sept, InterFace AG

Titel, Gestaltung und Satz:
Alexander Kermas, Fana-Lamielle Samatin, DOAG Dienstleistungen GmbH
Foto Titel © Evgeniya Ponomareva/Fotolia.com
Foto S. 31 © adimas/Fotolia.com
Foto S. 38 © goodluz/Fotolia.com
Foto S. 41 Galileo Computing

Anzeigen:
Simone Fischer
anzeigen@doag.org

Mediadaten und Preise:
http://www.doag.org/go/mediadaten

Druck:
Druckerei Rindt GmbH & Co. KG
www.rindt-druck.de

Java aktuell
Magazin der Java-Community

www.ijug.eu

JETZT ABO BESTELLEN

Sichern Sie sich 4 Ausgaben für 18 EUR

Für Oracle-Anwender und Interessierte gibt es das Java aktuell Abonnement auch mit zusätzlich sechs Ausgaben im Jahr der Fachzeitschrift DOAG *News* und vier Ausgaben im Jahr Business News zusammen für 70 EUR. Weitere Informationen unter www.doag.org/shop/

FAXEN SIE DAS AUSGEFÜLLTE FORMULAR AN

0700 11 36 24 39

ODER BESTELLEN SIE ONLINE

go.ijug.eu/go/abo

Interessenverbund der Java User Groups e.V.
Tempelhofer Weg 64
12347 Berlin

Java *aktuell*

+++ AUSFÜLLEN +++ AUSSCHNEIDEN +++ ABSCHICKEN +++ AUSFÜLLEN +++ AUSSCHNEIDEN +++ ABSCHICKEN +++ AUSFÜLLEN

☐ **Ja**, ich bestelle das Abo Java aktuell – das IJUG-Magazin: 4 Ausgaben zu 18 EUR/Jahr

☐ **Ja**, ich bestelle den kostenfreien Newsletter: Java aktuell – der iJUG-Newsletter

ANSCHRIFT

Name, Vorname

Firma

Abteilung

Straße, Hausnummer

PLZ, Ort

GGF. ABWEICHENDE RECHNUNGSANSCHRIFT

Straße, Hausnummer

PLZ, Ort

E-Mail

Telefonnummer

X

Die allgemeinen Geschäftsbedingungen* erkenne ich an, Datum, Unterschrift

*Allgemeine Geschäftsbedingungen:

Zum Preis von 18 Euro (inkl. MwSt.) pro Kalenderjahr erhalten Sie vier Ausgaben der Zeitschrift "Java aktuell - das iJUG-Magazin" direkt nach Erscheinen per Post zugeschickt. Die Abonnementgebühr wird jeweils im Januar für ein Jahr fällig. Sie erhalten eine entsprechende Rechung. Abonnementverträge, die während eines Jahres beginnen, werden mit 4,90 Euro (inkl. MwSt.) je volles Quartal berechnet. Das Abonnement verlängert sich automatisch um ein weiteres Jahr, wenn es nicht bis zum 31. Oktober eines Jahres schriftlich gekündigt wird. Die Wiederrufsfrist beträgt 14 Tage ab Vertragserklärung in Textform ohne Angabe von Gründen.

Schön, wenn man Zeit gewinnt.

Mehr Effizienz für Ihr Unternehmen. Aber auch mehr Zeit für Sie persönlich. Mit unseren IT-Lösungen erhalten Sie beides. Trivadis ist führend bei der IT-Beratung, der Systemintegration, dem Solution-Engineering und bei den IT-Services mit Fokussierung auf Oracle- und Microsoft-Technologien im D-A-CH-Raum. Unsere Leistungen erbringen wir auf den strategischen Geschäftsfeldern Business Intelligence, Application Development, Infrastructure-Engineering sowie Training und Betrieb. Sprechen Sie mit uns über Ihre Anforderungen.
Wir nehmen uns Zeit für Sie. www.trivadis.com | info@trivadis.com

ZÜRICH ▪ BASEL ▪ BERN ▪ BRUGG ▪ LAUSANNE ▪ DÜSSELDORF ▪ FRANKFURT A.M.
FREIBURG I.BR. ▪ HAMBURG ▪ MÜNCHEN ▪ STUTTGART ▪ WIEN

trivadis
makes IT easier.