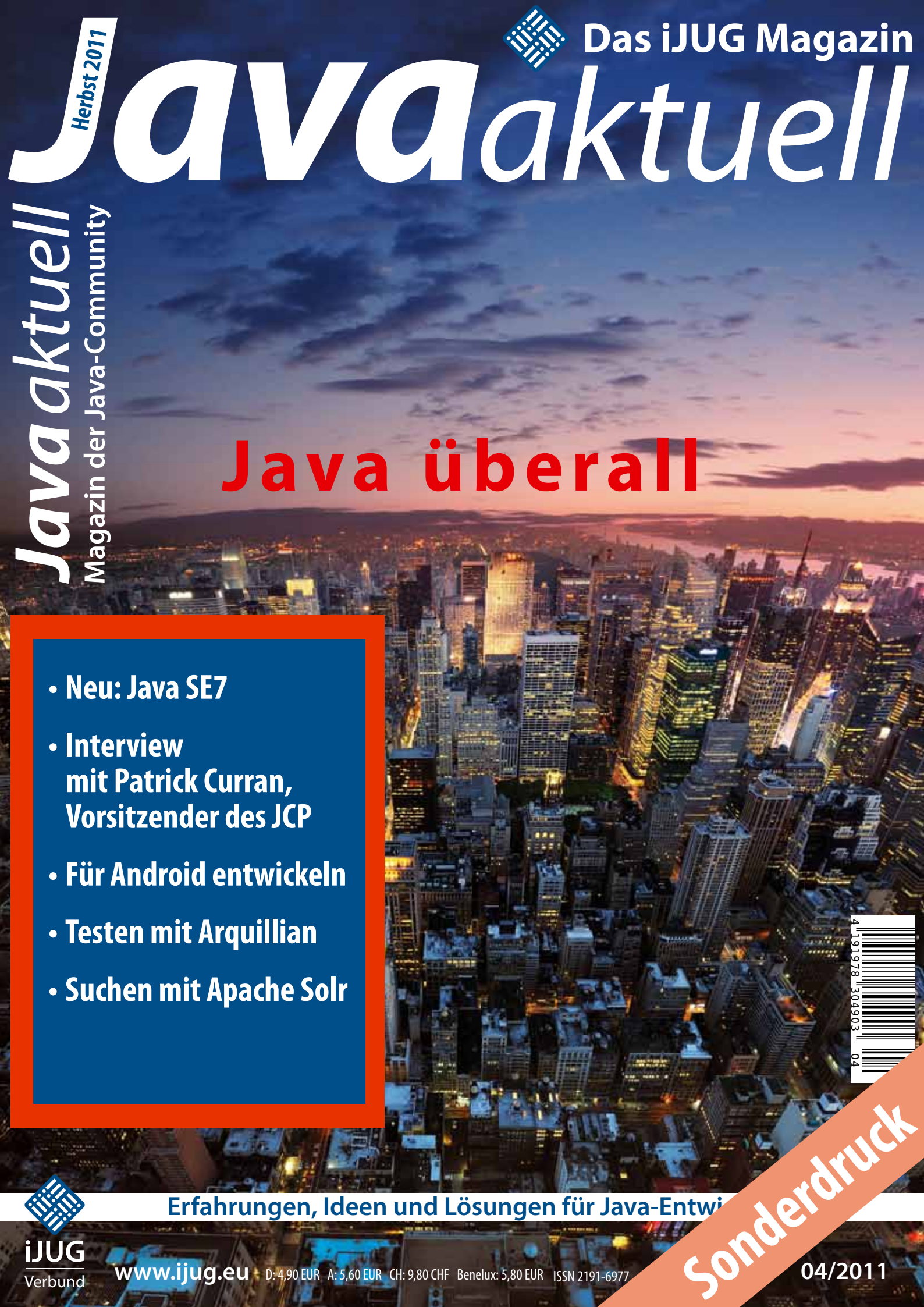

Das iJUG Magazin
Java aktuell
Herbst 2011
Java aktuell
Magazin der Java-Community
Java überall
• Neu: Java SE7
• Interview mit Patrick Curran, Vorsitzender des JCP
• Für Android entwickeln
• Testen mit Arquillian
• Suchen mit Apache Solr
Erfahrungen, Ideen und Lösungen für Java-Entwi
Sonderdruck
iJUG Verbund
www.ijug.eu
D: 4,90 EUR A: 5,60 EUR CH: 9,80 CHF Benelux: 5,80 EUR ISSN 2191-6977
04/2011

Inhalt

Kleiner Exkurs darüber, wo wir Java direkt oder indirekt überall antreffen, Seite 15

Slice – Unterstützung für verteilte, partitionierte und heterogene Datenbanken mit OpenJPA

Bernd Müller, Ostfalia Hochschule für angewandte Wissenschaften,
sowie Harald Wehr, MAN Truck & Bus AG

Die JPA-Provider arbeiten an Lösungsansätzen, von denen der Artikel exemplarisch Slice von OpenJPA vorstellt.

Die Cloud ist gegenwärtig das zentrale Thema der IT und auch Gegenstand von Arbeiten innerhalb des Java Community Process (JCP). Der Java Specification Request für Java-EE 7 (JSR 342) benennt die Cloud explizit als zentrales Thema (siehe Seite 54). Neben allgemeinen Diensten ist die Ablage von Daten in der Cloud ein wichtiges Ziel aktueller Entwicklungen. Die strikte Trennung der Daten verschiedener Kunden ist dabei selbstverständlich. Der JSR für JPA 2.1 (JSR 338) beinhaltet als einen der Schwerpunkte die Mandantenfähigkeit von JPA-Anwendungen (Multitenancy), die diese Trennung realisiert. Die Arbeitsgruppen des JCP arbeiten zwar mittlerweile unter der Ägide von Oracle sehr öffentlich, es gibt jedoch noch keine konkreten Vorschläge bezüglich der genannten Thematik. Als Beispiel für die Arbeit der JPA-Provider dient Slice von OpenJPA.

Slice arbeitet als Schicht zwischen JPA und JDBC. Aus Sicht von JPA wird eine virtuelle Datenbank genutzt. Diese Basisannahme der JPA-Spezifikation konkretisiert sich durch die explizite Benennung der Persistenzeinheit in der JPA-Konfigurationsdatei „persistence.xml". Dies bleibt bei der Verwendung von Slice erhalten, wird aber auf die tatsächlich verwendeten Datenbanken verallgemeinert. Ziel ist es, das JPA-Programmiermodell unverändert zu übernehmen. Die Slice-Schicht wird daher auf Konfigurationsebene in der „persistence.xml" definiert und konfiguriert. Zusätzlich erlauben Call-Back-Mechanismen die Steuerung der Verteilung von Daten und Anfragen auf die verwendeten Datenbanken.

Der Artikel stellt zunächst die zentralen Konzepte von Slice im Überblick vor, geht dann detailliert auf die Konfigurationsoptionen ein und beschreibt abschließend einige ausgesuchte Konzepte ausführlicher, da eine vollständige Darstellung der Möglichkeiten von Slice den Rahmen sprengen würde.

Einordnung

Der JSR 342 (Java EE 7) enthält als großes Arbeitsthema die Cloud. Java EE ist bereits als Standardplattform zur Entwicklung von Unternehmensanwendungen etabliert und soll in der üblichen Cloud-Charakterisierung zur Platform as a Service (PaaS) ausgebaut werden. Als übergeordnete Dachspezifikation enthält Java EE unter anderem JPA. Für Java EE 7 wird JPA 2.1 als JSR 338 entwickelt. Hier ist im Spezifikationsaufruf „Support for multitenancy" explizit genannt.

OpenJPA ist neben EclipseLink und Hibernate einer der bekannteren JPA-Provider. Das Teilprojekt Slice von OpenJPA stellt Möglichkeiten zur Verfügung, um eine horizontal partitionierte Datenbankumgebung zu realisieren. Man kann damit JPA-Anwendungen mandantenfähig (multi-tenant) machen. Slice wurde jedoch nicht mit diesem (expliziten) Ziel entwickelt und andere Einsatzgebiete sind möglich und auch sinnvoll.

Zentrale Konzepte

Slice teilt die einer JPA-Persistenzeinheit zugrunde liegende logische Datenbank in eine Menge horizontal partitionierter realer Datenbanken auf. Die übliche JPA-Funktionalitäten (CRUD und Transaktionalität) werden auf diese Datenbanken verteilt, wobei über zusätzliche Klassen die Verteilung gesteuert werden kann.

Ein Hauptziel von Slice ist der Verzicht auf Änderungen am Anwendungs-Code, um die Verteilung zu steuern. Die Anwendungslogik kann bei der Verwendung von Slice unverändert bestehen bleiben und nur die die Verteilung steuernden Klassen mit entsprechenden Call-Back-Methoden sind hinzuzufügen.

Konfiguration

Slice ist eine OpenJPA-Erweiterung, sodass die Konfiguration mit Provider-spezifischen Properties und nicht mit den durch JPA 2.0 definierten Properties (Präfix „javax.persistence") erfolgt. Der folgende Ausschnitt aus der „persistence.xml" zeigt die Definition der Java-SE-Persistenzeinheit „bank" und die Aktivierung von Slice über das Property „openjpa.BrokerFactory" (siehe Listing 1).

```
<persistence ...
  <persistence-unit name="bank">
    <class>...</class>
    <properties>
      <property name="openjpa.BrokerFactory" value="slice" />
      ...
```

Listing 1

Diese Aktivierung von Slice weist das OpenJPA-Laufzeitsystem an, eine virtuelle Datenbank, bestehend aus mehreren realen Datenbanken, zu verwenden, die wir in Listing 2 konfigurieren.

```
<property name="openjpa.slice.Names"
        value="bank1,bank2,bank3" />
<property name="openjpa.slice.Master"
value="bank1" />
<property name="openjpa.slice.Lenient"
value="true" />
```

Listing 2

Hier werden die Slices „bank1", „bank2" und „bank3" deklariert. Master-Slice ist „bank1". Dieser wird zur Primärschlüsselgenerierung für alle Slices verwendet. Das Property „Lenient" erlaubt der Anwendung, auch ohne Verbindung zu allen Slices zu arbeiten. Die Verbindungsdaten zu den einzelnen Datenbanken können global (Präfix „openjpa") oder für einen Slice (Präfix „openjpa.slice") angegeben sein (siehe Listing 3).

```
<property name="openjpa.ConnectionDri-
verName"
        value="org.postgresql.Driver" />
<property name="openjpa.ConnectionUser-
Name" value="..." />
<property name="openjpa.ConnectionUser-
Password" value="..." />
```

Listing 3

Das Beispiel verwendet die globale Version, sodass alle verwendeten Datenbanken PostgreSQL-Datenbanken mit derselben Benutzername/Passwort-Kombination sein müssen. Die verwendeten Datenbanken unterscheiden sich dann lediglich in ihren Verbindungs-URLs (siehe Listing 4).

```
<property name="openjpa.slice.bank1.
ConnectionURL"
        value="jdbc:postgresql://localhost/
bank1" />
<property name="openjpa.slice.bank2.
ConnectionURL"
        value="jdbc:postgresql://localhost/
bank2" />
<property name="openjpa.slice.bank3.
ConnectionURL"
        value="jdbc:postgresql://localhost/
bank3" />
```

Listing 4

Soll eine alternative Datenbank verwendet werden, so sind die alternativen Informationen für diesen Slice anzugeben. Falls die Benutzername/Passwort-Kombination von oben auch für den vierten Slice gilt, so reduziert sich die explizite Information auf den Treiber und das Verbindungs-URL (siehe Listing 5).

```
<property name="openjpa.slice.bank4.Con-
nectionDriverName"
        value="com.mysql.jdbc.Driver" />
<property name="openjpa.slice.bank4.
ConnectionURL"
        value="jdbc:mysql://localhost/bank4"
/>
```

Listing 5

Im Property „openjpa.slices.Names" muss der Slice ebenfalls aufgenommen sein.

Allgemeine Verwendung

Bestehende Entity-Klassen können unverändert mit Slice verwendet werden. Für einen Bankkunden mit folgender Klasse kann auch die Möglichkeit der Tabellen-Generierung genutzt werden:

```
@Entity
public class Kunde implements Serializable {
  @Id @GeneratedValue
  private Integer id;
  private String vorname;
  private String nachname;
  @Temporal(TemporalType.DATE)
  private Date geburtsdatum;
```

Mit der obigen Konfiguration wird dann jeweils eine Kundentabelle in den drei PostgreSQL- und der MySQL-Datenbank erzeugt. Beim Einfügen von Entities („em.persist()") werden diese zufällig auf die zur Verfügung stehenden Slices verteilt.

Explizite Verteilung

Die zufällige Verteilung bewirkt lediglich eine Verteilung der Last beziehungsweise des Volumens. Soll eine logische Verteilung stattfinden, so ist das Interface zu implementieren und in der „persistence.xml" zu konfigurieren:

```
public interface DistributionPolicy {
    String distribute(Object entity,
                      List<String> slices,
                      Object context);
}
```

Der erste Parameter ist das zu persistierende Objekt, der zweite die Liste der aktiven Slices und der dritte ein für spätere Erweiterungen vorgesehener Kontext, der im Augenblick mit dem aktuellen Persistenzkontext belegt aber noch nicht verwendet wird. Im folgenden Beispiel erfolgt eine Verteilung bezüglich des Anfangsbuchstabens des Bankkunden (siehe Listing 6).

```
public class DistributionPolicyNachname
          implements DistributionPolicy {
  private static Random random;
  public DistributionPolicyNachname() {
    random = new Random();
  }

  @Override
  public String distribute(Object entity,
      List<String> slices, Object context) {
    if (entity instanceof Kunde) {
      char anfangsbuchstabe = ((Kunde)entity).
getNachname()
                        .toCharArray()[0];
      if ('A' <= anfangsbuchstabe && anfangs-
buchstabe < 'I') {
        return „bank1";
      } else if ('I' <= anfangsbuchstabe
              && anfangsbuchstabe < 'M') {
        return „bank2";
      } else if ('M' <= anfangsbuchstabe
              && anfangsbuchstabe < 'T') {
        return „bank3";
      } else {
        return „bank4";
      }
    } else {
      return „bank1";
    }
  }
}
```

Listing 6

Die Konfiguration dieser Distribution-Policy erfolgt in der „persistence.xml" über:

```
<property name="openjpa.slice.Distribution-
Policy"
      value="de.jpainfo.DistributionPolicy-
      Nachname"/>
```

Die einfache statistische Verteilung kann in eine Partitionierung bezüglich eines Mandanten erweitert werden. Soll etwa ein Online-Banking für mehrere Banken, beispielsweise die Tochtergesellschaften eines Konzerns, erstellt werden, so wird das Entity „Kunde" um ein Property „mandant" erweitert und dieses über die Anwendung vergeben. Mögliche Werte des Properties sind die Slice-Namen (siehe Listing 7).

```
@Entity
public class Kunde implements Serializable {
  ...
  @Column(nullable = false)
   private String mandant; // Werte: bank1, ...,
  bank4
  ...
}
```

Listing 7

Die Distribution-Policy wird ebenfalls angepasst und verteilt nun auf Basis des Properties „mandant" (siehe Listing 8).

```
public class DistributionPolicyMandant ...

  public String distribute(Object entity,
            List<String> slices, Object context)
{
    if (entity instanceof Kunde) {
      return ((Kunde) entity).getMandant();
    } else {
      ...
    }
  }
}
```

Listing 8

Bei der Verwendung der Annotationen „@OneToOne", „@OneToMany", „@ManyToOne" und „@ManyToMany" realisiert JPA über das „cascade"-Attribut eine transitive Persistenz: Das Speichern des Wurzelobjekts impliziert automatisch auch das Speichern aller assoziierten Objekte, falls „PERSIST" als Cascadierungsoption gewählt wurde. Slice weicht in diesem Fall von der Distribution-Policy ab und speichert alle assoziierten Objekte im selben Slice. JPA-Queries, auf die wir später noch eingehen, können damit effizient in einer Datenbank joinen. Ein Join über mehrere Datenbanken wird von Slice nicht unterstützt.

Andere Verteilungs-Policies

Die DistributionPolicy wird durch drei weitere Policies ergänzt, die erste ist ReplicationPolicy (siehe Listing 9).

```
public interface ReplicationPolicy {
  String[ ] replicate(Object entity,
                List<String> slices, Object
context);
}
```

Listing 9

Die „ReplicationPolicy" wird verwendet, um Daten in verschiedenen Datenbanken zu replizieren. Dies ist etwa für Aufzählungstypen („Enums" wie „Nationalitäten" und „Währungen") sinnvoll, die von mehreren Entities verwendet werden. Die Interface-Methode „replicate()" der „ReplicationPolicy" entspricht der „distribute()"-Methode der „DistributionPolicy", gibt jedoch ein Array von Slice-Bezeichnern zurück. Die zu replizierenden Klassen werden in der „persistence.xml" für das Property „ReplicatedTypes" aufgezählt, beispielsweise „Nationlitaet" und „Waehrung" und in die von „getTargets()" zurückgegebenen Slices repliziert:

```
<property name="openjpa.slice.Replicated-
Types"
      value="de.jpainfo.Nationalitaet,de.
      jpainfo.Waehrung"/>
```

Beim Lesen von Daten aus der Datenbank muss zwischen der EntityManager-Methode „find()" und JPA-Queries unterschieden werden. Dies geschieht mit den zwei Policies (siehe Listing 10).

```
public interface FinderTargetPolicy {
  String[ ] getTargets(Class<?> cls, Object
oid;
                List<String> slices, Object
                context);
}
public interface QueryTargetPolicy {
  String[ ] getTargets(String query,
  Map<Object,Object> params,
                String language, List<String>
                slices,
                Object context);
}
```

Listing 10

Die „getTargets()"-Methode der „FinderTargetPolicy" erhält als ersten Parameter die Entity-Klasse und als zweiten Parameter die Objekt-Id analog zur „find()"-Methode. Dritter und vierter Parameter sind die bereits beschriebenen Übergabewerte. Die „getTargets()"-Methode der Query"TargetPolicy" erhält als ersten Parameter den JPA-Query-String, als zweiten Parameter eine Map von Query-Parametern und ihren Werten und als dritten Parameter die Anfragesprache (für spätere Erweiterungen, im Augenblick nicht verwendet). Der vierte und fünfte Parameter wurden bereits diskutiert.

Beispiele

Zu Beginn des Artikels haben wir ein Kunden-Entity mit Vorname, Nachname und Geburtsdatum definiert. Anfragen ohne die Verwendung einer „QueryTargetPolicy" werden gegen alle aktiven Slices ausgeführt, was wir nun in einem Beispiel demonstrieren wollen. Bei der Standard-JPA-Query wird eine Select-Anfrage gegen alle aktiven Datenbanken (in unserem Fall drei PostgreSQL und eine MySQL-Datenbank) abgesetzt (siehe Listing 11).

```
TypedQuery<Kunde> query =
em.createQuery(„select k from Kunde k
order by k.nachname",
            Kunde.class);
List<Kunde> list = query.getResultList();
```

Listing 11

Die Anfrage enthält das entsprechende „Order By" von SQL, das heißt es wird in den jeweils beteiligten Datenbanken sortiert. Die sortierten Kundenlisten sind von Slice in Java verschmolzen. Es kommt ein Merge-Algorithmus zum Einsatz, sodass die Komplexität des Merge (O(n)) die Komplexität des Sortierens in den Datenbanken (O(n log(n)) insgesamt nicht negativ beeinflusst. Im nächsten Beispiel wird der jüngste Kunde gesucht. Auch hier kommt eine Standard-JPA-Query zum Einsatz:

```
Date gebu = (Date) em.createQuery(
    „select max(k.geburtsdatum) from Kunde
    k")
    .getSingleResult();
```

Es wird das jeweilige Maximum in den vier aktiven Datenbanken berechnet und anschließend von Slice in Java das Maximum dieser vier Werte verwendet. Auch hier ist die Komplexität (O(n)) nicht negativ beeinflusst.

Ausblick

Die beschriebene OpenJPA-Erweiterung Slice ist noch nicht im aktuellen Release 2.1 von OpenJPA enthalten, aber in den Snapshots für 2.2. Die Implementierung scheint noch nicht auf Produktionsniveau.

Das von uns entwickelte Beispiel des maximalen Geburtsdatums war fehlerhaft ausgeführt. Nach Meldung des Fehlers wurde dieser innerhalb eines Tages gefixt, was für die Agilität des Projekts spricht. Bei anderen Problemen befinden wir uns im Augenblick in der Klärungsphase.

EclipseLink arbeitet an ähnlichen Features. Kurz vor Drucklegung dieses Artikels wurde Release 2.3 von EclipseLink veröffentlicht, das sogenannte „Composite Persistence Units" ermöglicht, die ebenfalls mehrere Datenbanken in einer logischen Persistence Unit vereinigen. Die Annotation „@Multitenant" (und weitere) erlaubt die explizite Verwendung von Mandanten auf Quell-Code-Ebene und erscheint uns als vielversprechender Ansatz für dieses Problem. Entsprechende Entwicklungen bei Hibernate sind uns nicht bekannt.
In der JSR Expert-Group arbeiten Entwickler von OpenJPA, EclipseLink und Hibernate mit. Man darf gespannt sein, welche Ideen bezüglich Mandantenfähigkeit und Cloud letztendlich den Einzug in JPA 2.1 und damit Java-EE 7 schaffen.

Bernd Müller
bernd.mueller@ostfalia.de
Harald Wehr
harald.wehr@gmail.com

Bernd Müller ist seit März 2005 Professor für Software-Technik an der Fakultät Informatik der Hochschule Braunschweig/Wolfenbüttel, nachdem er sieben Jahre Professor für Wirtschaftsinformatik an der Hochschule Harz war. Praktische Erfahrung hat er zuvor im Wissenschaftlichen Zentrum der IBM in Heidelberg sowie bei HDI Informationssysteme in Hannover gesammelt.

Harald Wehr ist seit 2005 bei der MAN Gruppe in Salzgitter als Java-Entwickler beschäftigt. Seit 2008 befasst er sich verstärkt mit der SAP HR Anwendungsentwicklung und Beratung. Zusammen mit Bernd Müller verfasste er das Buch „Java-Persistence-API mit Hibernate". Die Autoren arbeiten derzeit an einer Neuauflage des Buches, das im kommenden Frühjahr beim Hanser-Verlag erscheinen wird. Gegenstand ist JPA in der Version 2.0 bzw. 2.1. Als Provider werden EclipseLink, Hibernate und OpenJPA verwendet.

NetBeans Platform 7

gelesen von Jürgen Thierack

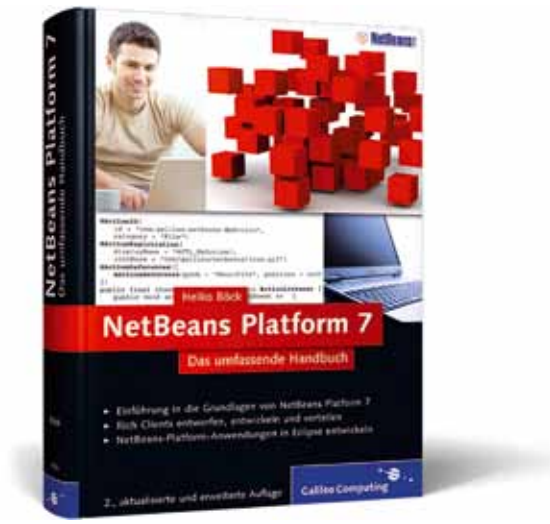
Über den Autor erfährt man auf dem Umschlag: „Heiko Böck ist Master of Science in Informatik und Experte im Bereich der professionellen Software-Entwicklung mit Java. Für seine Arbeit mit NetBeans wurde er ins NetBeans Dream Team gewählt."

Das Buch ist zur Version 7.0 der NetBeans-Plattform erschienen, die noch vor JDK 7 herauskam. Eine Version 7.01 der NetBeans folgte Anfang August 2011 mit Berücksichtigung der endgültigen Java-Version und vielen hundert Bugfixes. Für November 2011 stehen mit Version 7.1 neue Features auf dem Plan.

Was im Titel nicht zum Ausdruck kommt: Der Schwerpunkt liegt eindeutig bei der Rich-Client-Entwicklung, nicht bei NetBeans als allgemeines Entwicklungswerkzeug für Java, PHP, C++ und HTML. So werden beispielsweise die neuen Features des HTML-Editors in der Version 7.0 nicht besprochen.

Überhaupt ist der Autor von Rich-Client-Plattformen absolut überzeugt. „Die vermeintliche Komplexität der Plattform-Konzepte ist einer der Hauptgründe, warum sich Rich-Client-Plattformen noch nicht als Quasi-Standard bei der Client-Anwendungsentwicklung durchgesetzt haben. Tatsächlich hat ein Entwickler zu Beginn den Eindruck, vor einem Berg von APIs und Konzepten zu stehen. Sind diese jedoch erst einmal erlernt beziehungsweise verstanden, so ergeben sich immense – zu Beginn vielleicht nicht erahnte – Synergien und Erleichterungen, welche die anfängliche Lernphase schnell wieder ausgleichen." Das ist richtig und trifft auf Eclipse-RCP zu.

Das Buch stellt die modulare Struktur der NetBeans vor, die APIs, Datenbank-Anbindungen und natürlich die Netbeans-typischen Swing-Oberflächen. Und wer von Eclipse zur NetBeans migrieren will findet dafür viele Tipps. Als umfangreiches „lerning by doing"-Beispiel wird ein MP3-Manager entwickelt.



Zwei Kapitel beschäftigen sich damit, wie man in Eclipse und in der IntelliJ-IDEA-NetBeans-Plattform Anwendungen entwickeln kann. Doch dahinter stehen nur Anleitungen, wie man von diesen Entwicklungswerkzeugen aus Maven veranlassen kann, ein als Maven-Projekt bereits fertig existierenes NetBeans-RCP-Projekt zu bauen. Es ist aber nun mal nicht sinnvoll, ohne die NetBeans für die NetBeans zu arbeiten.

Fazit: Das Buch ist gut geeignet für Leute, die NetBeans-RCP-Anwendungen programmieren möchten.

Jürgen Thierack ist in der Software-Branche freiberuflich unterwegs. Seit 10 Jahren liegt sein Schwerpunkt bei Fragen der Finanzmathematik, darunter der Preisfindung für Optionen und Optionsscheine. Aktuelle Thematik: Trendfolgesysteme.

Titel:	NetBeans Platform 7
Autoren:	Heiko Böck
Verlag:	Galileo Computing
Umfang:	670 Seiten
Preis:	49,90 €
ISBN:	978-3-8362-1731-6

Bestellen eines kostenlosen Exemplares der Zeitschrift Java aktuell

Anschrift:

Name, Vorname

Firma

Abteilung

Straße, Hausnummer

PLZ, Ort

ggf. Rechnungsanschrift

E-Mail

Telefonnummer

Die allgemeinen Geschäftsbedingungen* erkenne ich an, Datum, Unterschrift

Jetzt Abonnement sichern:

- O Abonnement Newsletter: Java aktuell – der iJUG-Newsletter, kostenfrei
- O Java aktuell – das iJUG-Magazin Abo: vier Ausgaben zu 18 Euro im Jahr

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

Senden Sie das ausgefüllte Formular an:

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

oder faxen Sie es an:
0700 11 36 24 39

oder bestellen Sie online:
go.ijug.eu/go/abo

*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.

Impressum

Herausgeber:
Interessenverbund der Java User Groups e.V. (iJUG)
Tempelhofer Weg 64, 12347 Berlin
Tel.: 0700 11 36 24 38
www.ijug.eu

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

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

Chefin von Dienst (CvD):
Carmen Al-Youssef,
office@ijug.eu

Titel, Gestaltung und Satz:
Claudia Wagner,
DOAG Dienstleistungen GmbH

Anzeigen:
CrossMarkeTeam, Ralf Rutkat,
Doris Budwill
redaktion@ijug.eu

Mediadaten und Preise:
http://www.ijug.eu/images/vorlagen/2011-ijug-mediadaten_java_aktuell.pdf

Druck:
adame Advertising and Media GmbH Berlin
www.adame.de