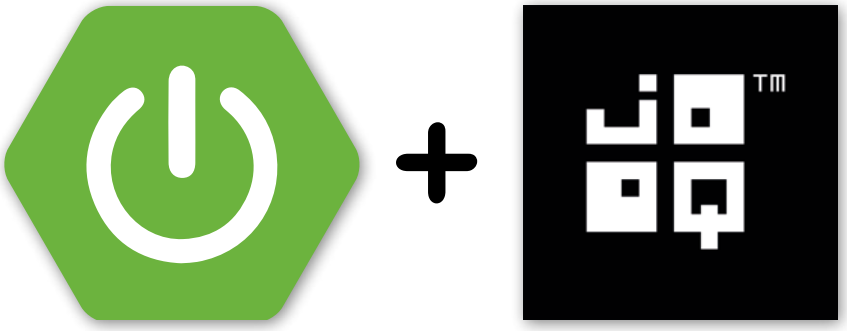

Datenbank-zentrische Anwendungen mit Spring Boot und jOOQ

Michael Simons, ENERKO INFORMATIK GmbH

Wie können Datenbank-zentrische, aber nicht Datenbank-abhängige Anwendungen mit modernen Frameworks entwickelt werden? jOOQ steht für „Java Object Oriented Querying“ und beschreibt ein Framework zum Erstellen von Abfragen – ein „query builder“-Framework. Zusammen mit Spring Boot ist es vielleicht die ideale Möglichkeit, HTTP-APIS für analytische Abfragen zu entwickeln.

Hinweis: Der Quelltext und die vollständige Demo dieses Artikels stehen unter *[1]* zur Verfügung.

Die ersten Gehversuche des Autors auf der Suche nach einer Alternative zu Oracle Forms 6 endeten vor vielen Jahren sehr schnell in einem ernüchternden Dschungel aus XML-Konfiguration, kaum zu bändigenden Abhängigkeiten und aufwändigen Deployments. Die erste Java-basierte Web-Anwendung, die er entwickeln durfte, wurde mit einer XML-Datei ähnlich der in *Abbildung 1* konfiguriert. Egal ob Java EE oder Spring-Framework: Bis zum Jahr 2007 kam man um explizite Konfiguration mit XML nicht herum. Zu diesem Zeitpunkt war XML die einzige Möglichkeit, das Spring-Framework vollständig zu konfigurieren.

Abbildung 1: A long time ago, in a framework far, far away

Der erste Commit im Spring-Boot-Projekt stammt aus dem Jahr 2013. Spring Boot setzt seit der ersten öffentlichen Version konsequent auf intelligente Konventionen und Defaults, Verzicht auf XML-Konfiguration und optimalen Einsatz von Konfiguration durch Java-Code. *Listing 2* nimmt – sofern die Abhängigkeiten im Klassenpfad definiert sind – die gleiche Konfiguration wie die XML-Datei vor.

Viele Open-Source-Komponenten wie Spring, Tomcat, JPA/Hibernate und verschiedenste Datenbanken haben eine extrem hohe Qualität erreicht und die Probleme sind oftmals anders gelagert: Wie lassen sich übliche funktionale und nicht-

funktionale Herausforderungen lösen, die einer effizienten Produktiv-Setzung im Weg stehen? Konsistent in Bezug auf Konfiguration, Entwicklung und Deployment? Spring Boot ist angetreten, um genau diese Fragen zu beantworten. Dabei stellt es fertig konfigurierte Instanzen des Spring-Frameworks mit „opinionated Defaults" bereit. Spring Boot unterstützt dabei nicht nur das Spring-Framework, den Inversion of Control Container, Beans und das MVC-Framework, sondern ein riesiges Ökosystem weiterer Technologien und Konzepte wie Spring Security, Caches, Enterprise Integration Pattern und mehr.

Diese Unterstützung kommt in Form sogenannter „Starter". Dies sind Standard-Java-Archive, die als Abhängigkeiten im Build-System deklariert sind. Sie bringen automatische Konfiguration sowie benötigte Libraries mit und erlauben eine gezielte Auswahl von unterstützten Technologien, unter anderem auch von jOOQ. Wichtig dabei ist, dass Spring Boot an keiner Stelle einen Code-Generator einsetzt. Andernfalls wäre es schwierig, Konfiguration zu ändern oder zu ersetzen. Falls es notwendig ist, kann übrigens immer noch XML zur Konfiguration genutzt werden.

Alles in allem ermöglicht Spring Boot eine bestmögliche Out-of-the-box-Erfahrung mit dem Spring-Ökosystem. Um konsistente Projektdefinitionen zu gewährleisten, gibt es mit dem Spring Initializr unter *[2]* eine Web-Anwendung, die es nach Angabe von Namen und Projekt-Koordinaten sowie der Wahl des Build-Systems ermöglicht, fertige Projekte inklusive eines Build-Files und einer minimalen Anwendung herunterzuladen.

Listing 1 und 2 zeigen das Skelett einer „Hallo, Welt"-Anwendung, die eine URL zur Verfügung stellt und den Aufrufer grüßt. *Listing 1* ist der Build File des Tools Maven *[3]* und definiert unter anderem die Liste der Abhängigkeiten, die Sprachversion (Java 8) und die Projekt-Koordinaten. In *Listing 2* wird die eigentliche Anwendung definiert, die aus einer Hauptklasse („Application"), annotiert mit „@SpringBootApplication", sowie einem Rest-Controller („HalloWorldController") besteht.

Dessen Methode „#greeting" steht unter der URL „/hallo" zur Verfügung und wertet Parameter aus, die entweder, wie im Beispiel, aus den Request-Query-Parametern oder auch aus URL-Pfaden stammen können. Wird dieses Java-Programm

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://ma-
ven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.doag</groupId>
  <artifactId>redstack</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>DOAG Red Stack demo</name>
  <description>Demo project for Spring Boot</description>

  <parent>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-parent</artifactId>
      <version>1.4.2.RELEASE</version>
      <relativePath/> <!-- lookup parent from repository -->
  </parent>

  <properties>
      <java.version>1.8</java.version>
  </properties>

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

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

Listing 1: „pom.xml"

```
package org.doag.redstack;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
public class Application {

  @RestController
  public static class HalloWorldController {

      @GetMapping("/hallo")
      public String greeting(@RequestParam final String name) {
          return "Hallo, " + name + "\n";
      }
  }

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

Listing 2: Application.java „A new hope"

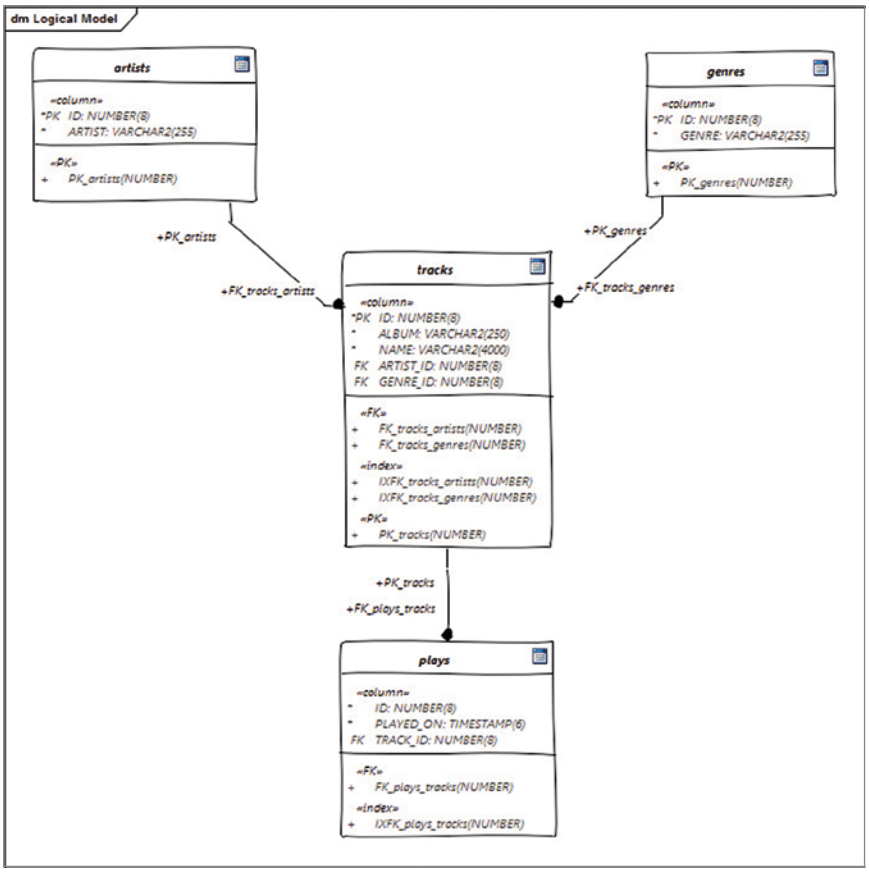

Abbildung 2: Datenmodell der Demo

```
Select *
   from tracks
 where album = 'True Survivor';
```

Listing 3: Triviale Abfrage

```
@Entity
@Table(name = "tracks")
public class TrackEntity implements Serializable {

   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Integer id;

   @Column
   private String album;


   public static void main(String...a) {
       final EntityManagerFactory factory = Persistence.createEntityManag
erFactory("whatever");
       final EntityManager entityManager = factory.createEntityManager();

       List<Track> tracks = entityManager.createQuery("Select t from
tracks where album = :album")
           .setParameter("album", "True Survivor")
           .getResultList();
   }
}
```

Listing 4: Triviale Abfrage mit Hibernate

gestartet, ergibt der Aufruf von „http://localhost:8080/hallo?name=DOAG“ das Resultat „Hallo, DOAG“.

Dieses Projekt kann man mit Maven bauen. Es entsteht ein ausführbares Java-Archiv vom Typ „.jar“, das alle benötigten Teile des Spring-Frameworks, Spring Boot und einen eingebetteten Application Container (Apache Tomcat) enthält. Das Archiv ist knapp 18 MB groß und kann überall dort gestartet werden, wo Java zur Verfügung steht.

Spring Boot: Fazit

Mit Spring Boot wird das Spring-Framework extrem zugänglich. Sowohl für Entwickler, die sich sehr lange nicht im Java-Umfeld bewegt, zum Beispiel lange im Bereich PL/SQL oder in der Forms-Entwicklung tätig waren, als auch für Menschen, die noch nicht lange im Alltag mit Software-Entwicklung zu tun haben. Da der Fokus nicht mehr auf einer aufwändigen Konfiguration liegt, kann man sich auf interessante Fragen konzentrieren: auf Fachlichkeit, Software-Architektur, Datenstrukturen oder auch darauf, modernes Java oder SQL zu lernen.

Java und Datenbanken: Plain SQL, ORM oder etwas dazwischen?

Gegeben sei ein Datenmodell wie in *Abbildung 2*. Es handelt sich um ein Datenmodell zur Erfassung von Musik. In vier Tabellen werden Titel, Interpreten, Genre und Wiedergaben der Titel erfasst. Wer genau hinschaut, sieht, dass das Datenmodell nicht in dritter Normalform vorliegt, das Album wird bei den Titeln gespeichert. Die Daten aus diesem Modell sollen in einer Anwendung als Charts angezeigt werden.

Eine Abfrage auf die Tabelle der Titel ist trivial *(siehe Listing 3)*. Diese Abfrage kann natürlich auch mit einem objektrelationalen Mapping, zum Beispiel mit Hibernate, ausgeführt werden *(siehe Listing 4)*. Direkt wird aber deutlich, dass selbst eine einfache Abfrage mehr „Boilerplate“-Code benötigt. Deutlich besser handhabbar wird sie, wenn man Spring Data JPA einsetzt *(siehe Listing 5)*. Was aber tun, wenn die Anforderung lautet: Gib mir die Top-40-Titel des letzten Monats, absteigend sortiert nach Anzahl der Wiedergabe inklusive ei-

ner Position sowie der Änderung der Position im Vergleich zur Vorwoche? Mithilfe moderner SQL-Konstrukte ist dieser Geschäftsfall mit einer Oracle-Datenbank und obigem Datenmodell sogar recht leicht zu beantworten *(siehe Listing 6)*.

Diese Abfrage enthält mehrere Common-Table-Expressions sowie Window-Funktionen. Man könnte diese Abfrage nun in Java-Annotationen verstecken oder über eine PL/SQL-Funktion aufrufen: In allen Fällen würde man versuchen, den Programmierer vor dem SQL-Code zu schützen.

jOOQ arbeitet anders. Das Ziel ist nicht, obigen SQL-Code zu verstecken – im Gegenteil. jOOQ ist zuerst ein Code-Generator, der aus dem Datenbank-Schema Zugriffsklassen für eine Domain Specific Language (DSL) generiert. Diese DSL beinhaltet in erster Linie nur die Fachlichkeit, die durch das Datenmodell vorgeben ist, und keine Architekturvorgabe. Ihr Ziel ist es, SQL zu schreiben. *Listing 7* zeigt obige Abfrage typsicher mit jOOQ ohne Annotationen.

jOOQ steht für „Java Object Oriented Querying" und ist laut eigener Aussage ein Framework zur Erzeugung von Abfragen. Es stellt für unterschiedliche Schemas eine Domain-spezifische Sprache (DSL) zur Generierung Datenbank-spezifischer Statements zur Verfügung. jOOQ lässt sich auch ohne Schema nutzen, allerdings ist es dann nur noch in Hinblick auf die Verwendung von SQL-Sprachmitteln, nicht aber in Hinblick auf Tabellen und Spalten typsicher.

Das Datenbank-Schema ist also in der Regel die treibende Kraft. jOOQ wird in einem frühen Stadium in den Build-Prozess eingebunden, im Falle von Maven in der „generate-sources"-Phase. Es kann konfiguriert werden, welche Datenbank-Objekte (jOOQ unterstützt Tabellen, Prozeduren, Funktionen, Packages und mehr) betrachtet werden, welches Zielverzeichnis genutzt wird und welche zusätzlichen POJOs und Data-Access-Objekte generiert werden. Während des Build-Prozesses wird das Datenbank-Schema dann gelesen und ein Java-Pendant generiert. Teile davon sieht man in *Listing 7*. In der „from"-Klausel wird die Konstante „TRACKS" genutzt, die typsicher die gleichnamige Tabelle beschreibt.

```
public interface TrackRepository extends JpaRepository<TrackEntity, Integer> {

   public List<Track> findAllByAlbum(final String name);

   public static void main(String...a) {
       TrackRepository trackRepository;
       final List<Track> tracks = trackRepository.findAllByAlbum("True Survivor");
   }
}
```

Listing 5: Triviale Abfrage mit Hibernate und Spring Data JPA

```
WITH
   previous_month AS
   (SELECT p.track_id, count(*) as cnt,
           dense_rank() over(order by count(*) desc) as position
       FROM plays p
       WHERE trunc(p.played_on, 'DD') BETWEEN
           date'2016-04-01' and date'2016-04-30' GROUP BY p.track_id),
   current_month AS
       (SELECT p.track_id, count(*) as cnt,
           dense_rank() over(order by count(*) desc) as position
       FROM plays p
       WHERE trunc(p.played_on, 'DD') BETWEEN
           date'2016-05-01' and date'2016-05-31' GROUP BY p.track_id)
SELECT a.artist || ' - ' || t.name || ' (' || t.album || ')' as label,
       current_month.cnt,
       previous_month.position - current_month.position as change
   FROM tracks t
   JOIN artists a on a.id = t.artist_id
   JOIN current_month current_month on current_month.track_id = t.id
   LEFT OUTER join previous_month on previous_month.track_id = t.id
ORDER BY current_month.cnt desc, label asc
FETCH FIRST 20 ROWS ONLY;
```

Listing 6: Komplexe SQL-Abfrage

```
this.create
       .with(currentMonth)
       .with(previousMonth)
       .select(label,
           currentMonth.field("cnt"),
           previousMonth.field("position").minus(
               currentMonth.field("position")
           ).as("change")
       )
       .from(TRACKS)
       .join(ARTISTS).onKey()
       .join(currentMonth)
.on(currentMonth.field("track_id", BigDecimal.class)
.eq(TRACKS.ID))
       .leftOuterJoin(previousMonth)
.on(previousMonth.field("track_id", BigDecimal.class)
.eq(TRACKS.ID))
       .orderBy(currentMonth.field("cnt").desc(), label.asc())
       .limit(n)
       .fetch()
       .formatJSON(response.getOutputStream());
```

Listing 7: Komplexe Abfrage mit jOOQ in Java

Datenbank-Migrationen

Strebt man eine möglichst reibungsfreie Entwicklung an, eventuell noch mit dem Ziel, Continuous Delivery zu erreichen, sind Datenbank-Migrationen essenziell. Als „Datenbank-Migration" wird der kontrollierte Prozess bezeichnet, eine bekannte Schema-Version auf eine neue Version zu heben. Dabei können zum Bei-

spiel Tabellen angelegt oder geändert sowie neue Trigger oder Ähnliches eingeführt werden. Spring Boot unterstützt zwei Werkzeuge ohne weitere Eingriffe: Liquibase und Flyway. In der verlinkten Demo wird Flyway eingesetzt *[4]*.

Die grundsätzliche Idee ist folgende: Die Datenbank-Migration wird gegen eine Entwicklungs-Datenbank durchgeführt, die idealerweise jedem Entwickler separat zur Verfügung steht. Dann erst wird die Generierung des jOOQ-Schemas angestoßen und anschließend der Java-Quelltext kompiliert. So ist sichergestellt, dass nur gegen ein aktuelles Schema entwickelt wird. All dies passiert noch im Build-Prozess, also ohne Unterstützung von Spring Boot. Werden nun Feature-Branches gemischt, kann es vorkommen, dass auch Migrationen gemischt werden müssen. Konflikte fallen an diesen Punkt auf. Wird dann die ausgelieferte Anwendung das erste Mal gegen die Produktionsdatenbank gestartet, sorgt Spring Boot dafür, dass die Migration ebenfalls gegen die Produktions-Datenbank gestartet wird. Damit passt das jOOQ-Schema dann auch zur Produktions-Datenbank.

Abbildung analytischer Abfragen auf URLs

Sobald in einem Spring-Boot-Projekt der „spring-boot-starter-jooq"-Starter eingebunden ist, steht jOOQ mit einer Instanz der Klasse „DSLContext" zur Verfügung. Der Lesbarkeit halber empfiehlt der Hersteller, diese Variable „create" zu nennen. Entwickelt man für eine Oracle-Datenbank, sind ebenfalls der entsprechende JDBC-Treiber und die kommerzielle Version von jOOQ zur Verfügung zu stellen.

In *Listing 8* wird die URL „/{artistIds}/topNAlbums" auf die Funktion „getTopNAlbums" abgebildet. Dabei ist das Fragment „/{artistIds}" ein URL-Parameter, der von Spring Boot typsicher auf ein Array von Zahlen („BigDecimal[]") abgebildet wird. Weitere Parameter stehen in Form von Query-Parametern zur Verfügung, zum Beispiel die Datumsparameter „from" and „to", die beide typsicher optional sind.

Mit diesen Informationen wird dann eine Abfrage erstellt und ausgeführt, um die Wiedergabe aller Alben im gegebenen Zeitraum oder im aktuellen Monat auszuführen. Das Ergebnis dieser Abfrage wird direkt als JSON in den HTTP-Response geschrieben und kann von Clients weiterverarbeitet werden. *Listing 9* zeigt die entsprechende Rückgabe in Auszügen. In *[5]* wird das Thema „HTTP-APIS" für analytische Abfragen in aller Ausführlichkeit behandelt.

```
@RequestMapping(path = "/{artistIds}/topNAlbums")
public void getTopNAlbums(
   @PathVariable final BigDecimal[] artistIds,
   @RequestParam(defaultValue = "10") final int n,
   @RequestParam
   @DateTimeFormat(iso = ISO.DATE)
   final Optional<LocalDate> from,
   @RequestParam
   @DateTimeFormat(iso = ISO.DATE)
   final Optional<LocalDate> to,
   final HttpServletResponse response
) throws IOException {
   response.setContentType(MediaType.APPLICATION_JSON_UTF8_VALUE);
   this.create
       .select(TRACKS.ALBUM,
           count())
       .from(PLAYS)
       .join(TRACKS).onKey()
       .where(TRACKS.ARTIST_ID.in(artistIds))
       .and(from.map(Date::valueOf)
           .map(PLAYED_ON_TRUNCATED_TO_DAY::greaterOrEqual)
           .orElseGet(DSL::trueCondition)
       )
       .and(to.map(Date::valueOf)
           .map(PLAYED_ON_TRUNCATED_TO_DAY::lessOrEqual)
           .orElseGet(DSL::trueCondition)

       .groupBy(TRACKS.ARTIST_ID, TRACKS.ALBUM)
       .orderBy(count().desc(), TRACKS.ALBUM.asc())
       .limit(n)
       .fetch()
       .formatJSON(response.getOutputStream());
}
```

Listing 8: Nutzung des DSL-Kontexts

```
{
   "fields": [
       {
           "schema": "",
           "table": "TRACKS",
           "name": "ALBUM",
           "type": "VARCHAR"
       },
       {
           "name": "count",
           "type": "INTEGER"
       }
   ],
   "records": [
       [
           "Everything Louder Than Everyone Else",
           151
       ],
       [
           "BBC Live & In-Session",
           68
       ]
   ]
}
```

Listing 9: Curl -X „GET" „http://127.0.0.1:8080/api/artists/54,86/topNAlbums"

```
define(['ojs/ojcore', 'knockout', 'jquery', 'moment', 'ojs/ojselectcombobox', 'ojs/ojchart', 'ojs/ojdatetime-
picker'],
   function (oj, ko, $, moment) {
       function artistsContentViewModel() {
           var self = this;
           self.areaSeriesValue = ko.observableArray([]);
           self.areaGroupsValue = ko.observableArray([]);
           var updateCharts = function () {
               // Fill model
           }
       };

       self.optionChanged = function (event, data) {
              updateCharts();
           };
       }
       return new artistsContentViewModel();
   });
```

Listing 10: JavaScript-Modell einer Oracle-JET-Anwendung

```
<h2>Cumulative plays per artist and day</h2>
<div id='chart-container'>
   <div id="lineAreaChart" style="max-width:1024px;width:100%;height:320px;" data-bind="ojComponent: {
       component: 'ojChart',
       type: 'lineWithArea',
       series: areaSeriesValue,
       groups: areaGroupsValue,
       timeAxisType: 'enabled',
       animationOnDisplay: 'on',
       animationOnDataChange: 'on',
       stack: 'on',
       hoverBehavior: 'dim',
       zoomAndScroll: 'live',
       overview: {rendered: 'off'},
       dataCursor: dataCursorValue
   }"></div>
</div>
```

Listing 11: Die HTML-View

Spring Boot und jOOQ: Fazit

jOOQ ist mit Spring Boot quasi trivial verwendbar; sowohl die notwendige Code-Generierung im Build-Prozess als auch das Boot-Strapping des jOOQ-Kontexts passen in das Entwicklungsmodell von Spring Boot. jOOQ zielt nach Ansicht des Autors auf eine SQL-basierte Zwei-Schichten-Architektur. Es ist möglich, das Repository-Muster anzuwenden, aber die Erfahrung spricht dagegen. jOOQ spielt seine Stärken dann aus, wenn es um die Modellierung komplexer Abfragen geht, nicht in möglichst hoher Abstraktion.

Akzeptiert man die zweischichtige Architektur, kann man sehr leicht vorhandenes Wissen im Bereich „Datenbank-Modellierung und Abfrage“ in moderne Software-Entwicklung einbringen. Zudem kann man im Falle von kommerziellen Datenbanken alle Funktionen nutzen, für die man ja bereits bezahlt hat. Innerhalb einer Spring-Boot-Anwendung lässt sich jOOQ in nahezu beliebigen Cloud-Umgebungen einrichten.

jOOQ passt – wie viele andere Werkzeuge auch – nicht auf jeden Anwendungsfall. Liegt ein Domain-Driven-Design-Ansatz vor, sind komplexe Abfragen nicht maßgeblicher Bestandteil einer Anwendung oder ist der Schreibzugriff wichtiger, sprechen viele gute Gründe für den Einsatz eines ORM wie JPA/Hibernate, gerne auch zusammen mit Spring Data JPA. Ein hier skizzierter Anwendungsfall, quasi eine direkte Abbildung einer HTTP-Abfrage auf eine komplexe Query, lässt sich allerdings hervorragend mit jOOQ abbilden.

jOOQ ist kostenfrei nutzbar mit Open-Source-Datenbanken unter der Apache Software License 2.0. Für den Einsatz mit kommerziellen Datenbanken stehen verschiedene kommerzielle Lizenzen zur Verfügung.

Oracle JET: Die Antwort auf das JavaScript-Frontend-Dilemma

Der Schwerpunkt dieses Artikels liegt nicht auf dem Frontend, aber die besten Daten sind nur schlecht erfassbar ohne ansprechende Präsentation. Viele moderne Anwendungen werden „headless“ betrieben, ohne ein auf dem Server generiertes Web-Frontend. Wird eine UI erstellt, so führt im Jahr 2017 kaum noch

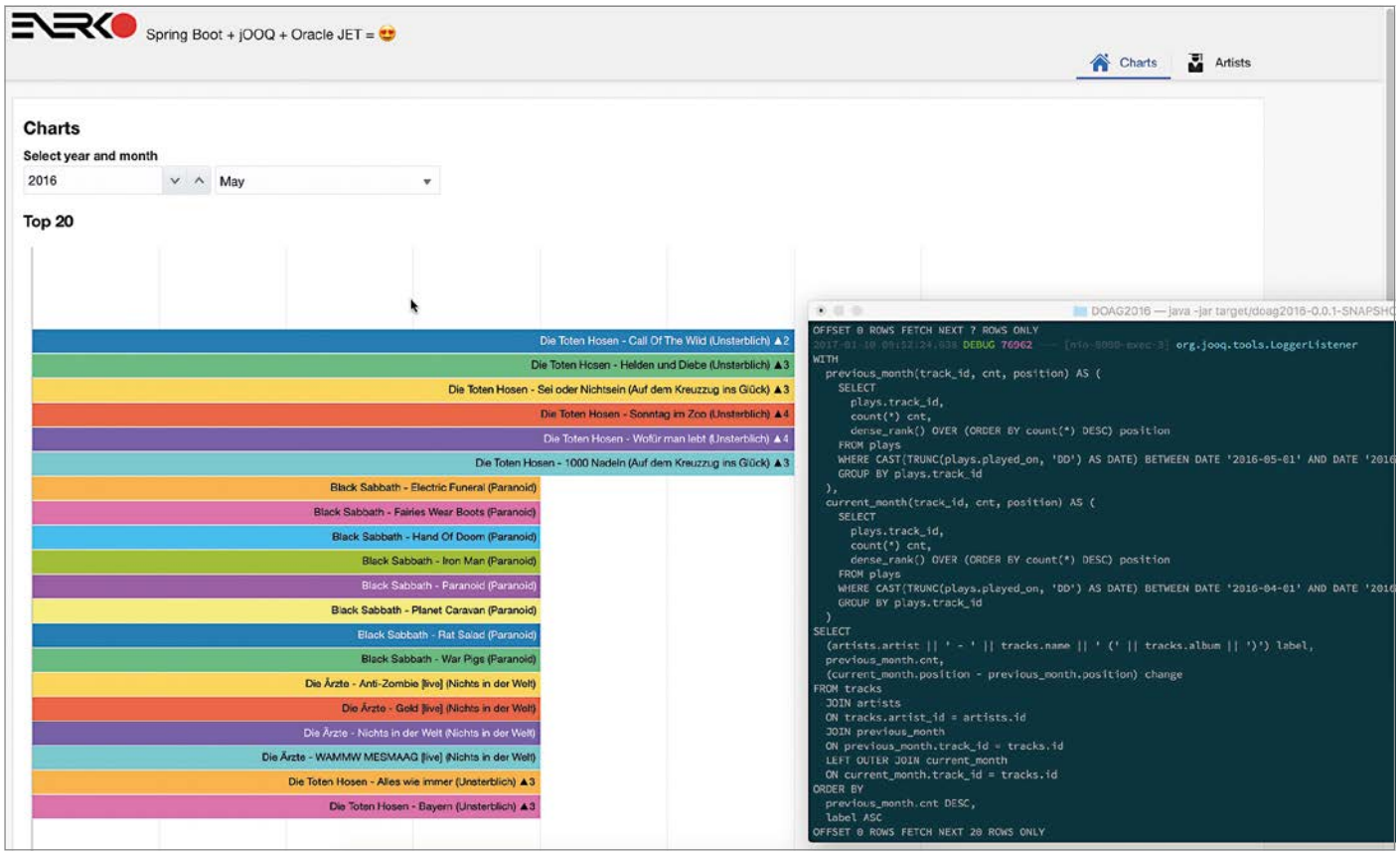

Abbildung 3: Oracle JET mit der Log-Ausgabe von Spring Boot und jOOQ

ein Weg an JavaScript und allem, was dazu gehört, vorbei. Dabei sind weitere Build-Tools und eine erschöpfende Auswahl an Frameworks beteiligt.

Aus diesem Dilemma gibt es mehrere Auswege: Man wählt ein Komponenten-Framework wie eine beliebige JSF-Implementierung, GWT oder Vaadin, das auf Seiten des Servers HTML und JavaScript-Komponenten erzeugt. Solange man keine speziellen Anpassungen benötigt, wird der Entwickler gut bis sehr gut vor UI-JavaScript geschützt. Ganz schnell läuft man aber Gefahr, dass scheinbar einfache Kundenanforderungen sehr in den generierten Client-Code eingreifen, sodass die hohe Abstraktion mehr schadet als nutzt. Einen anderen Ausweg bietet die Zusammenstellung eigener Sammlungen von Komponenten oder der Entwurf eigener Frameworks (vermutlich die Quelle vieler auf „*.js“ endender Frameworks). In beiden Fällen läuft man Gefahr, vom Wartungsaufwand überrannt zu werden.

Oracle JavaScript Extension Toolkit (JET) wählt genau den Ansatz intelligent zusammengestellter Komponenten, um Entwicklern eine Sammlung von Werkzeugen und modulare Anwendungen auf Basis moderner JavaScript-, CSS3- und HTML5-Prinzipien zur Verfügung zu stellen. Es ermöglicht auf Basis bekannter Konzepte und Komponenten die Erstellung von modernen Enterprise-Anwendungen sowohl für den Desktop als auch für mobile Endgeräte.

Oracle JET greift auch die Idee einer „Bill of Materials“ auf: Es nutzt RequireJS für das Management von Modulen und Abhängigkeiten, KnockoutJS zur Implementierung des Model-View-ViewModel-Konzepts (MVVM) sowie jQuery und jQuery-UI für moderne, ansprechende UI-Komponenten. Diese Komponenten wurden mit dem Ziel erstellt, barrierefreien Zugriff zu ermöglichen sowie vollständig internationalisierbar zu sein. *Listing 10 und 11* zeigen das Modell und die View einer Anwendung, die das Ergebnis einer mit jOOQ realisierten SQL-Abfrage anzeigt, die auf eine HTTP-URL abgebildet wurde *(siehe Abbildung 3)*.

Das Ziel der Demo war es, die über jOOQ mit SQL selektierten Daten mit Spring Boot einem Frontend zur Verfügung zu stellen und dort darzustellen:

- Als Balken-Diagramm
- Als kumulatives Linien-Diagramm
- Mit ansprechender, zugänglicher Eingabemöglichkeit für Parameter
- Grundsätzlich internationalisierbar

Laut eigener Aussage zielt Oracle mit dem JET-Framework auf „intermediate to advanced JavaScript developers“. Mit grundsätzlich vorhandenem Wissen zu JavaScript, jQuery und der MVVM-Implementierung KnockoutJS gelang es, obige Anforderungen an die Demo einfach umzusetzen, die Aufgabe ließ sich zusammen mit dem Oracle JET Cookbook *[12]* in wenigen Stunden realisieren.

Oracle JET: Fazit

Die Demo ist eine hübsche Spielerei, beinhaltet aber viele Dinge, die im Geschäftsalltag benötigt werden. Die Technologie ist zugänglich und nach erster Einschätzung auch zukunftssicher, findet man sie doch bereits in Teilen in Oracle Apex wieder.

Abschließendes Fazit

Steht man vor der Entscheidung, neue Produkte auf Basis bestehender, großer Daten-

modelle zu entwickeln und gegebenenfalls Rücksicht auf alte Anwendungen nehmen zu müssen, ist der hier vorgestellte Stack eine Möglichkeit, vorhandenes Wissen über SQL und Datenbanken mit auf eine moderne Plattform zu nehmen und mit einem ansprechenden Frontend zu versehen.

Es entstehen leicht verteilbare Anwendungen beziehungsweise Microservices, die Datenbank-zentrisch, aber nicht Datenbank-abhängig sind. Vorhandene Datenbank-Technik und insbesondere vorhandenes Datenbank-Wissen bleiben erhalten und können genutzt werden.

Soll eine Oberfläche entstehen, ist das verhältnismäßig einfach mit Oracle JET möglich. Da die hier vorgestellte Abbildung analytischer Abfragen auf HTTP-Schnittstellen JSON als Transport-Format nutzt, kann auch nahezu jede beliebige andere Frontend-Technologie des Webs genutzt werden.

Das Unternehmen des Autors setzt seit mehr als einem Jahr die Kombination aus Spring Boot und jOOQ erfolgreich in ihren Produkten ein. Man verzichtet nicht gänzlich auf objektrelationale Abbildungen, setzt aber gezielt SQL-Abfragen in einer Art und Weise ein, die sowohl im Java- als auch im SQL-Umfeld gleichermaßen gut zu verstehen und zu nutzen ist.

Am Ende sind sowohl SQL und objektrelationale Abbildungen als auch jOOQ Experten-Werkzeuge. Ohne entsprechendes Wissen können in der Regel nicht alle Möglichkeiten genutzt oder nicht gut eingesetzt werden. Eine Lektüre der Theorie, der Handbücher und auch einiger sehr lesenswerter Blogs wird daher dringend empfohlen. Nachfolgend sind einige davon genannt.

Quellen und Links

[1] Vollständige Demo: *https://github.com/michael-simons/DOAG2016*
[2] Spring Initializr: *http://start.spring.io*
[3] Maven: *http://maven.apache.org*
[4] Take control of your development databases evolution: *http://info.michael-simons.eu/2016/10/31/take-control-of-your-development-databases-evolution/*
[5] An HTTP-API for analytic queries: *http://info.michael-simons.eu/2016/11/02/an-http-api-for-analytic-queries*
[6] Euregio JUG: *http://www.euregjug.eu*
[7] ENERKO Informatik: *http://www.enerko-informatik.de*
[8] info.michael-simons.eu: *http://info.michael-simons.eu*
[9] Spring Boot: *http://projects.spring.io/spring-boot*
[10] jOOQ: *http://www.jooq.org*
[11] Oracle JET: *http://www.oracle.com/webfolder/technetwork/jet/index.html*
[12] Oracle JET Cookbook: *http://www.oracle.com/webfolder/technetwork/jet/jetCookbook.html*
[13] Flyway by Boxfuse: *https://flywaydb.org*
[14] Spring Data JPA: *http://projects.spring.io/spring-data-jpa*
[15] Modern SQL: *https://modern-sql.com*
[16] Vlad on Hibernate: *https://vladmihalcea.com/tutorials/hibernate*

Michael Simons
michael@simons.ac