01-2014 | Frühling | www.ijug.eu

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

Java aktuell

Java hebt ab

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

4 191978 304903 01

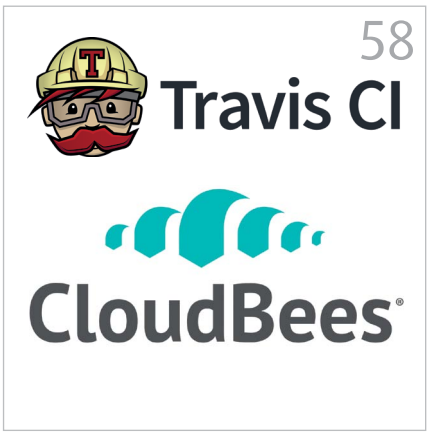
iJUG Verbund

Sonderdruck

Java macht wieder richtig Spaß:
Neuigkeiten von der JavaOne, Seite 8

Interview mit Mark Little über das Wachstum und die Komplexität von Java EE 7, Seite 30

JavaFX oder eine HTML5-basierte Technologie:
Wo und warum der Einsatz von JavaFX sinnvoll ist , Seite 44



Cloud-hosted Continuous Integration mit CloudBees und Travis CI, Seite 58

Eine Android-App samt JEE-Back-End generieren und in der Cloud bereitstellen

Marcus Munzert, Generative Software GmbH

Android ist zurzeit das meistgenutzte mobile Betriebssystem. In vielen Fällen, vor allem auch bei Business-Anwendungen und bei Anwendungen, die bisher klassischen Embedded-Geräten vorbehalten waren, bietet eine native Entwicklung mit Java Vorteile. Und die Java Enterprise Edition (JEE) hat alles, um ein passendes, leistungsfähiges Back-End an den Start zu bringen. Der Artikel beschreibt eine Software-Architektur für dieses Szenario und zeigt, wie man durch Code-Generierung schnell und zuverlässig eine entsprechende Anwendung entwickeln kann.

Eine der Herausforderungen bei der Entwicklung einer Android-Anwendung, die mit Servern kommuniziert, ist ein effektiver, effizienter und zuverlässiger Daten-Austausch zwischen Gerät und Server. Häufig wird der Architektur-Stil „Representational State Transfer" (REST) genutzt. Als Client-Server-Protokoll kommt meist HTTP(S) zum Einsatz.

Während beim Austausch-Datenformat früher der Schwerpunkt auf XML lag, hat inzwischen „JavaScript Object Notation" (JSON) die Nase vorn. Dienste wie YouTube, Twitter und Box bieten in ihren neuesten API-Versionen ausschließlich JSON als Datenformat an. All diese Zutaten bilden die Grundlage der Architektur, die in Abbildung 1 schematisch dargestellt ist.

Aktuell widmet sich übrigens das Forschungsprojekt „mohito" [1] der Problematik, Daten über mehrere Plattformen hinweg zur Verfügung zu stellen und zu synchronisieren. Dort wird, ebenso wie im weiter unten beschriebenen Verfahren, auf den Einsatz von domänenspezifischen Sprachen (DSL) und modellgetriebener Software-Entwicklung gesetzt. Doch zunächst die Architektur.

REST-API mit JEE

Bei der Gestaltung eines REST-API ist darauf zu achten, dass die fünf notwendigen REST-Eigenschaften berücksichtigt sind [2]:

- Einheitliches Interface – unabhängig von einem Client, der es nutzt
- Zustandslosigkeit; ein Server-Aufruf hängt also nicht von vorherigen Aufrufen ab
- Möglichkeit der Zwischenspeicherung der Rückgabe-Daten in einem Cache zwischen einem Client und dem eigentlichen Server
- Klare Trennung der Aufgaben von Client und Server
- Ein in Schichten aufgeteiltes System

Es ist hierbei lediglich ein organisatorischer Unterschied, ob das REST-API öffentlich verfügbar sein soll oder ob es ausschließlich von Clients genutzt wird, die zusammen mit dem API unter ein und derselben Kontrolle liegen. Im letzteren Fall sind leichter Änderungen am API durchführbar, da neuere Versionen der Clients gleichzeitig mit dem neuen, serverseitigen API geplant und veröffentlicht werden können. Auch das in diesem Artikel vorgestellte, generierte API eignet sich für beide Fälle. Es kann also auch von weiteren, nicht generierten Clients genutzt werden.

JAX-RS [3] ist Teil von JEE und bietet alles, um ein REST-API zu implementieren. Eine sogenannte „Resource" bildet den Einstiegspunkt in das REST-API. Von hier aus gibt es zwei Möglichkeiten, wie auf Business-Logik und die Persistenzschicht zugegriffen werden kann. Entweder wer-

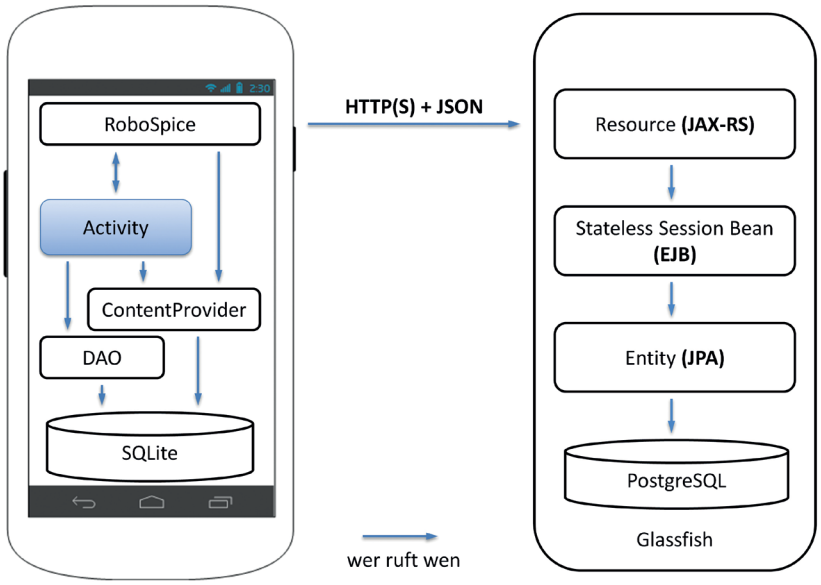


Abbildung 1: Schema der Software-Architektur

den in einem EAR-File bereitgestellte Session Beans angesprochen oder die Resource ist selbst eine Session Bean. Im letzteren Fall würden typischerweise alle JEE-Komponenten in ein WAR-File gepackt, was seit JEE 6 möglich ist [4]. Dadurch entfällt die Notwendigkeit, ein EAR-File zu erstellen. Die beiden Möglichkeiten sind in Abbildung 2 anhand eines auch im weiteren Verlauf des Artikels verwendeten Beispiels zu sehen.

Es ist zu beachten, dass sich Stateful-Session-Beans oder das Zwischenspeichern von Daten in der HTTP-Session über Request-Grenzen hinweg aufgrund der Zustandslosigkeit des REST-API nicht eignen. So wird auch keine Session-ID in der HTTP-Session hinterlegt. Wenn eine Authentifizierung notwendig ist, wird mit jedem Request vom Client ein zuvor ausgehandeltes Authentication Token mitgeliefert. Dieses Token identifiziert den Client und aus ihm werden dessen Zugriffsrechte abgeleitet [5].

Wenn man davon ausgeht, dass die Persistenzschicht in der JEE-Anwendung mithilfe von JPA umgesetzt ist, gibt es bereits eine einfache Art und Weise, die Entitäten einer Persistence-Unit automatisch direkt über ein REST-API ansprechbar zu machen: JPA-RS [6]. JPA-RS gehört (noch) nicht zum JEE-Standard, sondern zu EclipseLink. Einen konzeptionell ähnlichen, jedoch flexibleren und weiter gehenden Ansatz stellt die im Verlauf des Artikels beschriebene Codegenerierung dar. Aber zuerst einmal werfen wir noch einen Blick auf die Architektur des Android-Clients.

Der Android-Client

Der Android-Client muss für den Datenaustausch mit dem Back-End einige Randbedingungen berücksichtigen:

- Lebenszyklus einer Android-Anwendung (Activity Lifecycle)
- Offline-Fähigkeit
- Häufigkeit und Zeitpunkt der Daten-Synchronisation mit dem Back-End

Je nach Anforderungen an eine Anwendung werden die Punkte „Offline-Fähigkeit“ und „Daten-Synchronisation mit dem Back-End“ unterschiedlich gelöst. Es gibt mindestens vier Synchronisations-Muster, die auch gleichzeitig in ein und derselben Anwendung eingesetzt werden können:

- *Keine Synchronisation*
  Daten werden bei Bedarf über das REST-API geholt und lokal nicht gespeichert
- *Synchronisation on Demand*
  Der Anwender synchronisiert die Daten explizit, zum Beispiel durch Drücken eines Synchronize-Buttons
- *Synchronisation während der Benutzung*
  Dies ist eine Art Lazy-Loading. Daten werden während der Nutzung der Anwendung auf dem mobilen Gerät gespeichert
- *Synchronisation durch Server-Push*
  Ein Server informiert Anwendungen mithilfe des asynchronen Messaging-Verfahrens „Google Cloud Messaging for Android“ (GCM) [7] über Veränderungen im Datenbestand auf einem Server. Die Anwendung synchronisiert daraufhin im Hintergrund stillschweigend die lokalen Daten mit den Daten vom Server.

Außer für den Fall, dass überhaupt keine Daten-Synchronisation gefordert ist, werden die Daten lokal – wie auf mobilen Geräten üblich – in einer SQLite-Datenbank gespeichert. Darin werden auch Daten

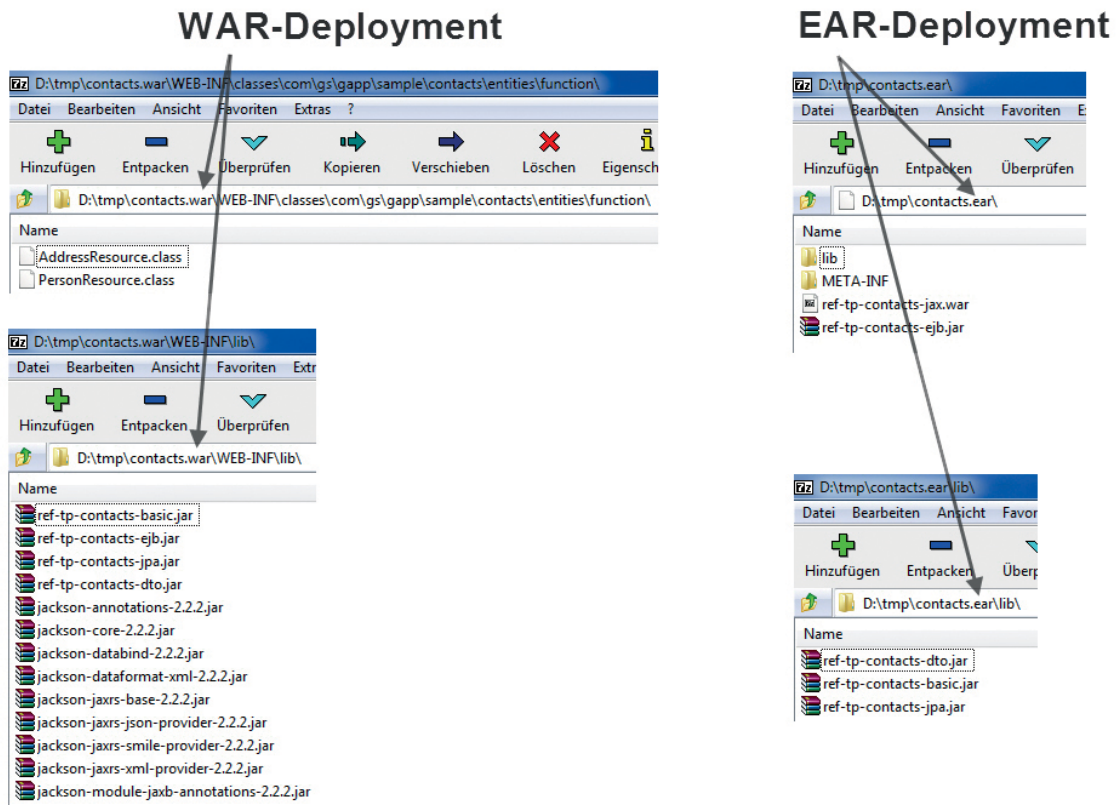


Abbildung 2: WAR- versus EAR-Deployment

aufbewahrt, die vom Benutzer auf dem Gerät neu erfasst oder modifiziert werden. Vom Benutzer gelöschte Daten werden zunächst nicht physisch vom Gerät entfernt, sondern nur als „zu löschen" markiert. Erst, wenn solche Daten auf dem Server gelöscht worden sind, werden sie auch physisch vom Gerät entfernt.

Zugriffe von der Android-Anwendung auf das Back-End müssen asynchron erfolgen, damit das User-Interface stets gut bedienbar ist. Werden das im Android-API angebotene AsyncTask [8] oder das Loader-Framework [9] dazu verwendet, kann das zur Folge haben, dass gerade in Ausführung befindliche Server-Zugriffe wiederholt ausgeführt werden müssen oder dass die zurückgelieferten Daten im User-Interface nicht oder zumindest nicht sofort angezeigt werden können.

Um nicht in diese Probleme hineinzulaufen, verwenden wir in der Android-Anwendung stattdessen sogenannte „Local Services". Ein Local Service ist ein Android-Service, der ausschließlich von der Anwendung selbst aufgerufen werden kann. Mit der Open-Source-Bibliothek „RoboSpice" [10] wird eine starke Vereinfachung der Nutzung von solchen Local Services erreicht.

Textuelle Modellierung

Eine Anwendung, die der bisher beschriebenen Architektur folgt, lässt sich zu großen Teilen aus einem kompakten, einfachen Modell generieren. Zur Modellierung werden textuelle DSLs zur Beschreibung von Datenstrukturen und User-Interfaces eingesetzt. Es genügt im ersten Schritt, die Datenstrukturen zu modellieren. Daraus lassen sich die Persistenz-Schicht, das REST-API, die lokale Datenhaltung auf dem mobilen Gerät (SQLite) und Client-Code zum Zugriff auf das API ableiten. Durch die zusätzliche Modellierung des User-Interface kann eine komplette Anwendung generiert werden. Zur Veranschaulichung ist in Listing 1 beispielhaft ein Modell mit zwei Entitäten gezeigt. Damit soll eine einfache Anwendung zur Verwaltung von Kontakten erstellt werden.

Die Modellierung des User-Interface gerät etwas umfangreicher. Eine detaillierte Beschreibung der User-Interface-DSL würde an dieser Stelle zu weit führen. Wir richten den Blick lediglich auf einen kleinen Ausschnitt des User-Interface-Modells, das in Listing 2 zu sehen ist: die Verbindung zwischen der modellierten Anwendung und dem Modell, das die Datenstrukturen des REST-API beschreibt. Hier wird zwischen „local Storage" und „remote Storage" unterschieden. „local Storage" bestimmt die Struktur der SQLite-Datenbank und „remote Storage" legt fest, wie die Daten über das REST-API zu erwarten sind. In unserem Fall sind die Strukturen identisch, was es uns einfach macht, Code zu generieren, der Daten zwischen REST-API und SQLite-Datenbank austauscht. Würden hier unterschiedliche Datenstrukturen angegeben, müsste man das Mapping zwischen SQLite-Datenbank und REST-API manuell codieren.

```
namespace com.gs.gapp.sample.contacts.entities;
import com.gs.gapp.sample.contacts.ContactsTypeAliases;
module ContactsPersistence kind = Persistence, Basic, Rest;
set TypeFilters = ContactTypeAliases;

enumeration Gender {
      entry MALE;
      entry FEMALE;
}

entity BaseEntity {
      set Storable = false;

      field pk : PrimaryKey {
          set Id = true;
    }
}

entity Person extends BaseEntity {
      field firstName : Text;
      field lastName : Text;
      field gender : Gender;
      field dateOfBirth : Birthday;

      field addresses : Address {
          set CollectionType = List;
          set LazyLoading = true;
          set REST-SerializeToIds = true;
    }
}

entity Address extends BaseEntity {
      field street : Text;
      field houseNumber : Text;
      field zipCode : Text;
      field city : Text;
}
```

Listing 1: Modell der Datenstrukturen

Generisches Modellierungswerkzeug

Das Besondere des zur Modellierung verwendeten Werkzeugs „gApp Modeling Suite" [11] ist seine einfache und universelle Einsetzbarkeit und Erweiterbarkeit. Sowohl dessen Editor als auch dessen Parser sind generisch implementiert. Alle Modell-Dateien erhalten die Endung „.gapp". Die Modellierung mit einer weiteren DSL wird jeweils durch die Installation eines Eclipse-Plug-ins ermöglicht, ohne einen weiteren

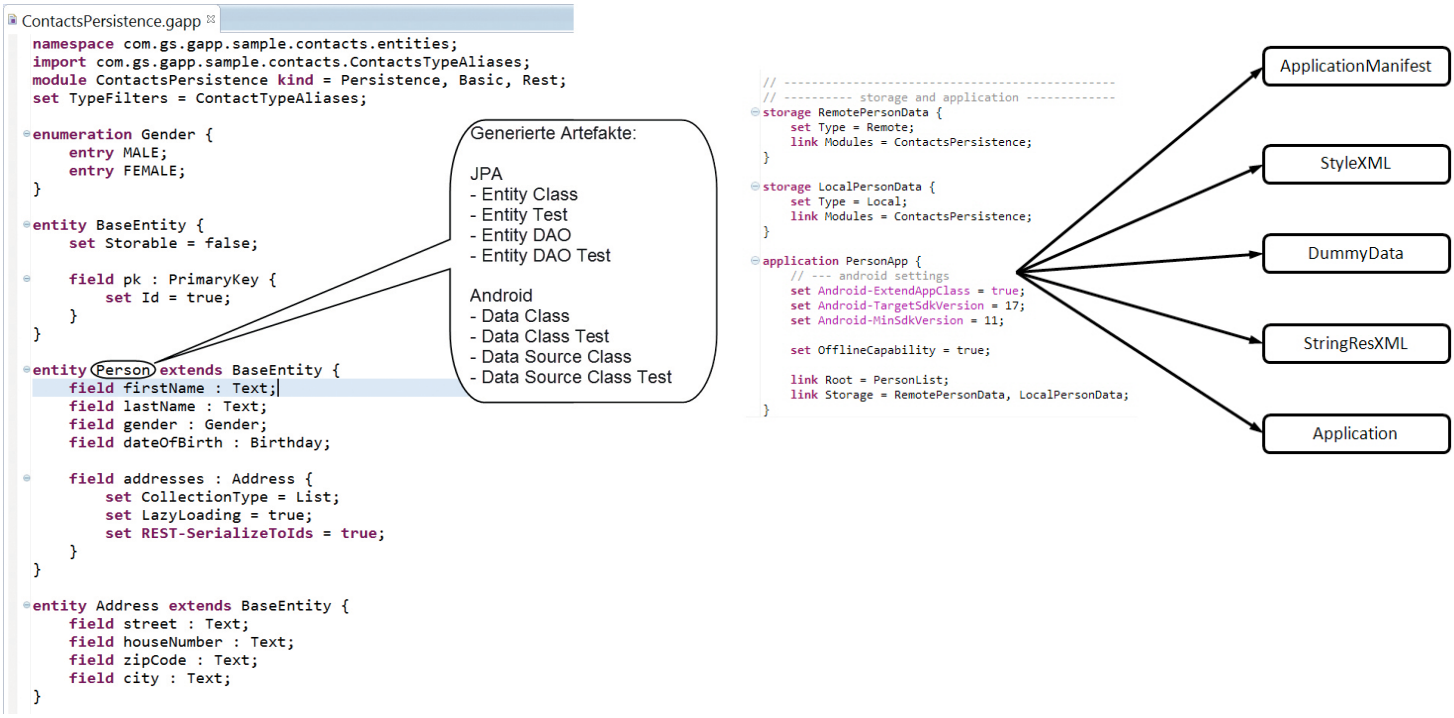


Abbildung 3: Beispiele von Modell-Elementen und dafür generierten Dateien

Editor oder Parser zu installieren. Ein solches Eclipse-Plug-in enthält lediglich eine DSL-Definition. Diese Flexibilität ermöglicht es auch, in ein eigentlich plattformunabhängiges Modell, wie in Listing 2 gezeigt, plattformspezifische Annotationen einzufügen. Listing 3 zeigt beispielhaft ein mit demselben Modellierungswerkzeug erstelltes Modell unter Verwendung der Function DSL zur Modellierung von Komponenten, in denen nach der Generierung Business-Logik von Hand programmiert werden kann. Mithilfe dieser DSL lassen sich unter anderem individuelle REST-Services und EJBs generieren.

Code-Generierung

Die textuellen Modelle für die Datenstrukturen und das User-Interface dienen als Eingabe für den Code-Generator. Dieser ist auf Basis der Virtual-Developer-Plattform entwickelt worden [12]. Er legt die Dateien in mehrere verschiedene Zielprojekte ab. Abbildung 3 zeigt beispielhaft, welche Dateien für welche Modell-Elemente durch den Generator erzeugt werden.

Der Generator selbst steht auf einem entfernten Virtual-Developer-Server als Service bereit. In der IDE werkelt lediglich ein leichtgewichtiges Plug-in, der sogenannte „Virtual Developer Cloud Connector". Dadurch können auf einem Virtual-Developer-Server bereitgestellte Code-Generatoren die volle Server-Power nutzen und von beliebigen IDEs aus aufgerufen werden (Eclipse, NetBeans, IntelliJ, Visual Studio, Xcode). Über diesen einen Cloud-Connector sind sämtliche auf einem Server verfügbaren Generatoren aufrufbar. Als Nebeneffekt sind für Generatoren eventuelle lokale Installationsprobleme von vornherein ausgeschlossen.

```
// ------------------------------------------------
// ---------- storage and application -------------
storage RemotePersonData {
      set Type = Remote;
      link Modules = ContactsPersistence;
}

storage LocalPersonData {
      set Type = Local;
      link Modules = ContactsPersistence;
}

application PersonApp {
    // --- android settings
    set Android-ExtendAppClass = true;
    set Android-TargetSdkVersion = 17;
    set Android-MinSdkVersion = 11;

    set OfflineCapability = true;

      link Root = PersonList;
      link Storage = RemotePersonData, LocalPersonData;
}
```

Listing 2: Modellausschnitt Datenstrukturen im Client

```
namespace com.gs.gapp.sample.contacts.function;
import com.gs.gapp.sample.contacts.ContactsTypeAliases;
import com.gs.gapp.sample.contacts.entities.ContactsPersistence;
module ContactsFunction kind = Function, Basic;
set TypeFilters = ContactTypeAliases;

type BirthdaySearchParameters {
      field gender : Gender;
      field startDate : Birthday;
      field endDate : Birthday;
}

function getPersonsForBirthday {

      in searchParameters : BirthdaySearchParameters;

      out persons : Person {
          set CollectionType = Set;
    }
}
```

Listing 3: Modell für Business-Logik-Komponenten

Abbildung 4 zeigt den Zusammenhang zwischen den einzelnen Virtual-Developer-Software-Komponenten.

Ein Code-Generator wird in purem Java gegen das Virtual-Developer-API entwickelt und mittels OSGi modular aufgebaut. Mit der Modularisierung wird das Modell mit den Datenstrukturen nicht direkt in einem einzelnen Schritt in Sourcecode für JAX-RS umgewandelt. Die Generierung führt zuerst mehrere Modell-zu-Modell-Transformationen aus. Als letzten Schritt erzeugt der Generator dann den JAX-RS-Sourcecode. Abbildung 5 zeigt die logische Struktur des gesamten Generators. Diese bekommt ein Anwender des Generators nicht zu sehen. Nur der Hersteller des Generators arbeitet mit solchen Strukturen, über die er Generatoren konfektioniert.

Mithilfe von vom Hersteller anders konfektionierten Generatoren kann man zu einem späteren Zeitpunkt die Datenstruktur des REST-API von der Datenstruktur der Persistenzschicht trennen und trotzdem weiter effektiv Code-Generierung einsetzen. Dieselbe Entkoppelung ist für die Datenhaltung auf dem Client möglich.

Bereitstellung des Back-Ends in der Cloud

In der heutigen Zeit stehen verschiedene Möglichkeiten zur Auswahl, ein JEE-Back-End für Testzwecke und für den Produktiv-Betrieb mit geringem Aufwand über einen PaaS-Anbieter in der Cloud bereitzustellen. Dazu zählen Jelastic, CloudBees, Heroku, Amazon Beanstalk, Cloud Foundry, OpenShift, Rackspace und CloudControl. Interessant ist auch das Angebot von Engine Yard [13], das in Kürze ebenfalls ein PaaS-Angebot für Java bereitstellen wird. Engine Yard hat seit einem Jahr eine Partnerschaft mit Oracle und es ist beabsichtigt, die PaaS-Angebote der beiden Firmen miteinander zu verknüpfen.

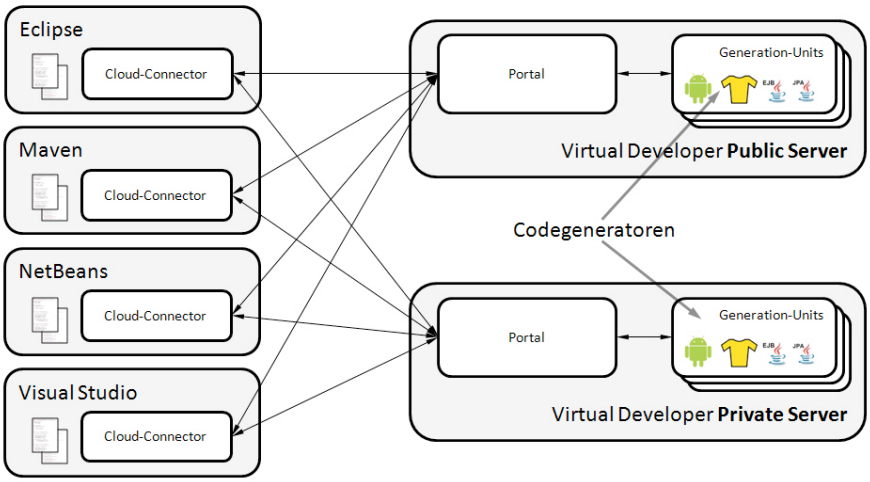


Abbildung 4: Virtual-Developer-Software-Komponenten

Beispielhaft sei hier das Vorgehen für die Bereitstellung des Back-Ends mit dem PaaS-Dienst Jelastic skizziert [14]. Eine der Besonderheiten an Jelastic ist, dass es nicht ein Betreiber von eigenen Rechenzentren ist. Stattdessen handelt es sich hierbei um einen Software, die vom gleichnamigen Unternehmen mit Sitz in Palo Alto entwickelt wird. Unabhängige Internet-Service-Provider nutzen diese Software, um damit eine PaaS anbieten zu können. Dies ermöglicht es, sich selbst auszusuchen, in welchem Land sich das Rechenzentrum befinden soll, bei dem man seine Anwendungen betreibt. Die zweite Besonderheit ist, dass man sich nicht einfach einen dedizierten oder virtuellen Server mietet, sondern verbrauchsabhängige Kosten anfallen (CPU, Memory, Disk Space, Traffic). Bis jetzt werden Java, PHP und Ruby als Plattformen angeboten. In Deutschland bietet die dogado Internet GmbH den Jelastic-Service an.

Mit Jelastic kann man ein WAR- oder EAR-File direkt in einen zuvor mit wenigen Klicks bereitgestellten GlassFish JEE Application Server deployen. Dies geht entweder über die webbasierte Administrations-Oberfläche oder über ein Jelastic Plug-in, das es für Eclipse, IDEA, Maven und NetBeans gibt. Abbildung 6 zeigt, wie die webbasierte Administrations-Oberfläche aussieht und an welcher Stelle das zuvor hochgeladene EAR-File in den Applikationsserver deployt wird. Mit dem Jelastic-IDE-Plug-in geht es noch schneller, da Hochladen und Deployment als eine zusammenhängende Aktion durchgeführt wird.

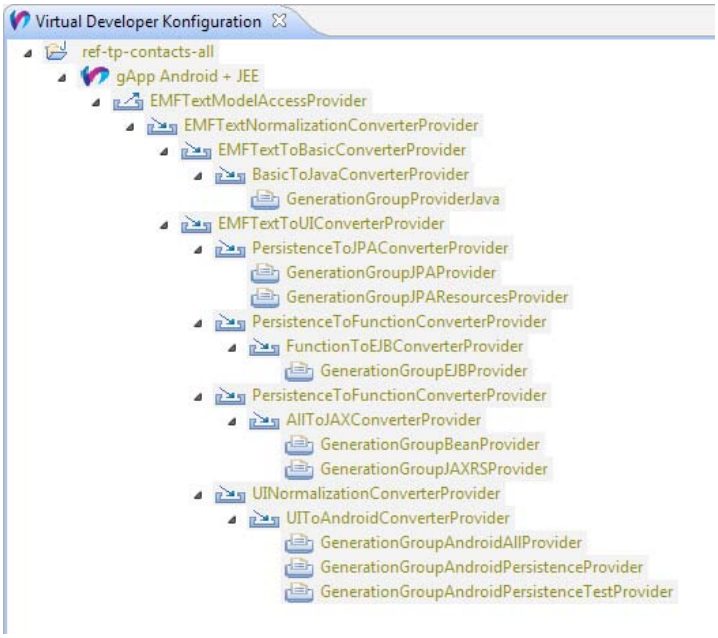


Abbildung 5: Struktur des Code-Generators

An dieser Stelle sei auch darauf hingewiesen, dass die Nutzung eines PaaS-Anbieters nicht für jeden Fall die optimale Lösung darstellt. Zu beachten ist beispielsweise, dass über eine PaaS nicht immer die Versionen von Software-Komponenten angeboten werden, die man für die eigene Anwendung benötigt. Letztendlich verliert man durch den Einsatz von PaaS-Lösungen bis zu einem gewissen Grad die Kontrolle über die Umgebung, auf der die eigene Anwendung laufen soll. Es muss jeder für sich entscheiden, ob dieser Nachteil durch die vielen Vorteile aufgewogen wird.

Fazit

Mithilfe von JEE, leichtgewichtiger Modellierung und Code-Generierung sowie der Nutzung von PaaS-Diensten für Java ist es heutzutage möglich, native mobile Client-Server-Anwendungen in kürzester Zeit zu entwerfen, zu entwickeln, zu testen und in Produktion zu nehmen. Um diese Agilität zu erreichen, müssen keine Kompromisse bei der Qualität der Software-Architektur und des Quellcodes eingegangen werden. Zudem eröffnet die Verwendung von plattformunabhängigen Modellen die große Chance, kosteneffizient native mobile Anwendungen für mehrere Plattformen liefern zu können, und zwar ohne dass die verschiedenen Implementierungen, was den Funktionsumfang angeht, divergieren.

Links

[1] Mohito: http://www.mohito-projekt.de
[2] REST Constraints: http://whatisrest.com/rest_constraints/index
[3] JAX-RS: http://docs.oracle.com/javaee/6/tutorial/doc/giepu.html
[4] WAR vs. EAR: http://www.munzandmore.com/2012/ora/ejb-31-stateless-session-bean-dependency-injection-asynchronous-methods
[5] Authentication: http://vinaysahni.com/best-practices-for-a-pragmatic-restful-api#authentication
[6] JPA-RS: http://wiki.eclipse.org/EclipseLink/Development/2.4.0/JPA-RS
[7] GCM: http://developer.android.com/google/gcm
[8] AsyncTask: http://developer.android.com/reference/android/os/AsyncTask.html
[9] Loaders: http://developer.android.com/guide/components/loaders.html
[10] RoboSpice: http://github.com/octo-online/robospice
[11] gApp: http://generative-software.de/gapp
[12] Virtual Developer: http://virtual-developer.com
[13] Engine Yard: http://engineyard.com
[14] Jelastic: http://jelastic.com

Marcus Munzert
marcus.munzert@generative-software.com

Marcus Munzert ist geschäftsführender Gesellschafter der 2007 gegründeten Generative Software GmbH. Er entwickelt seit 1998 mit Java und setzt seit 2002 mit Begeisterung Methoden der modellgetriebenen Software-Entwicklung ein.

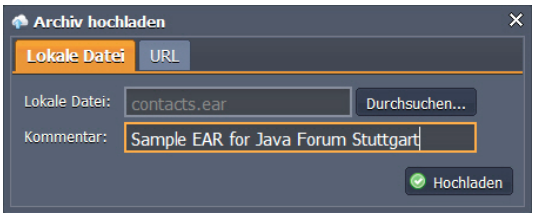


Abbildung 6: Jelastic-Administrations-Oberfläche

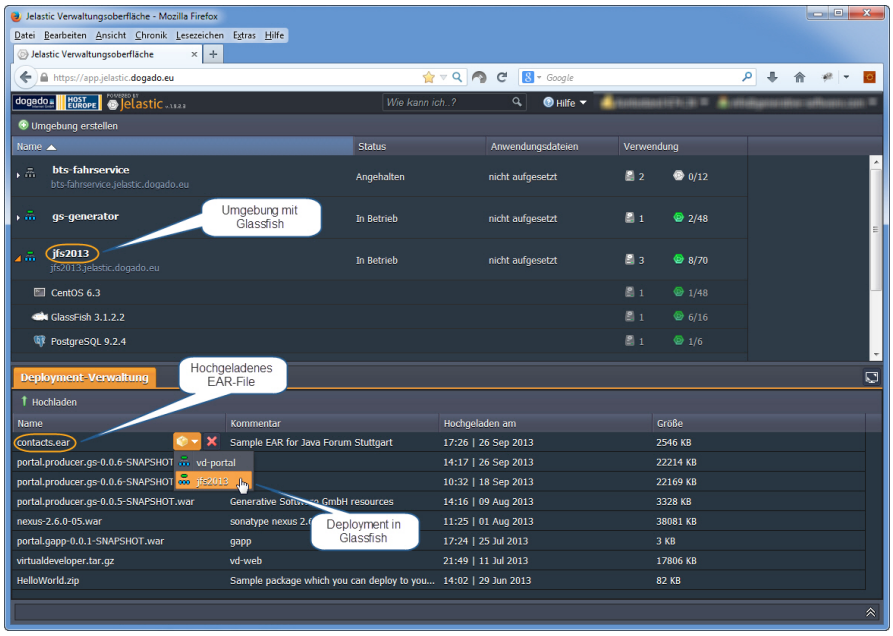


Abbildung 7: Jelastic-Administrations-Oberfläche

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.