Nr.02 | Sommer 2012 | www.ijug.eu

Java aktuell

Das Magazin der Java-Community

Java aktuell

Java im Aufwind

VisualVM
Unbekannte Kostbarkeiten des SDK

Grails
Die Suche ist vorbei

Microsoft und Java
Frei verfügbare Angebote für Software-Entwickler

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

Sonderdruck

iJUG Verbund

Interview mit John Duimovich, Distinguished Engineer in der IBM Software Group mit Schwerpunkt „Java Virtual Machines & Embedded Java", IBM Canada, Seite 16

„Java besitzt immer noch ein enormes Potenzial …" Stefan Koospal, Vorsitzender der Sun User Group Deutschland, im Gespräch mit Java aktuell, Seite 42

Der Sport ist in manchen Bereichen hilfreich für die Entwicklung von Software. Was wir vom America's Cup lernen können, Seite 44

Grails – die Suche ist vorbei

Stefan Glase und Christian Metzler, OPITZ CONSULTING GmbH

Grails ist ein Open-Source-Framework zur Entwicklung moderner Web-Applikationen auf Basis der dynamisch typisierten Programmiersprache Groovy und bewährten Technologien wie dem Spring-Framework, Hibernate und SiteMesh. Eine Vielzahl von Plug-ins macht es zudem möglich, wiederkehrende Problemstellungen mit bewährten Lösungen umzusetzen.

Grails baut auf der dynamischen Programmiersprache Groovy auf und profitiert auf diese Weise von einer extrem ausdrucksstarken und auch für einen Java-Entwickler in kürzester Zeit erlernbaren Sprache. Zugleich integriert sich Grails nahtlos in das Java-Ökosystem und kann auf bestehende Klassen und Bibliotheken zurückgreifen. Dies macht den Einstieg besonders dann interessant, wenn man aufgrund von Investitionen in die Java-Plattform an die Java Virtual Machine gebunden ist.

Convention over Configuration

Inspiriert von Ruby on Rails sind auch bei Grails die Paradigmen „Convention over Configuration" und „Don't repeat yourself" wichtige Erfolgsfaktoren für eine hohe Qualität und Produktivität bei der Entwicklung Grails-basierender Web-Anwendungen. So gibt Grails bereits in der Verzeichnisstruktur einen einheitlichen Rahmen vor, der es Entwicklern einfach macht, sich in neue Projekte einzuarbeiten.

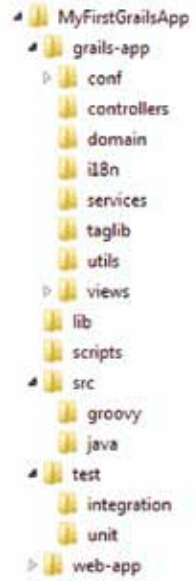

Abbildung 1: Verzeichnisstruktur eines Grails-Projekts

Bei Grails finden sich Artefakte eines bestimmten Typs immer am gleichen Ort innerhalb der Projektstruktur und folgen auch in der Namensgebung bestimmten Konventionen. So werden Konfigurationsskripte unterhalb des Verzeichnisses „grails-app/conf" verwaltet und lassen bereits durch ihren Namen die jeweils konfigurierten Aspekte erkennen:

- „ApplicationResource.groovy" bündelt zusammengehörende Ressourcen unter eindeutigen Namen zur einfacheren Einbindung
- „BootStrap.groovy" beschreibt die Ausführung von Code beim Starten und Stoppen der Applikation
- „BuildConfig.groovy" stellt alle erforderlichen Informationen zum Bauen der Anwendung bereit
- „Config.groovy" verwaltet die allgemeine Konfiguration der Anwendung wie beispielsweise das Logging
- „DataSource.groovy" definiert die zu verwendenden Datenquellen unter Berücksichtigung der Umgebung (Entwicklung, Test, Produktion etc.)
- „UrlMappings.groovy" beschreibt die Übersetzung von URLs in Aufrufe dafür vorgesehener Methoden

Modellierung von Fachklassen

Einen zentralen Bestandteil in Grails-Anwendungen stellen die „Domain Classes" dar, die unter „grails-app/domain" abgelegt werden und in der Regel fachliche Objekte der Anwendung repräsentieren. Unter der Haube greift Grails auf das bekannte Framework Hibernate zurück, um eine Abbildung dieser Domänen-Klassen auf relationale Datenbanken zu realisieren. Dieses objektrelationale Mapping wird durch Grails erweitert und nennt sich „Grails Object Relational Mapping" (GORM).

Mittels einer domänenspezifischen Sprache (DSL) können Domänenklassen um Validierungsregeln (Constraints) erweitert und Beziehungen zu anderen Klassen definiert werden. Auch die Abbildung in der Datenbank kann über eine DSL beispielsweise an bestehende Datenbank-Schemata oder unternehmenseigene Konventionen angepasst werden.

```
class Kunde {
    String name
    String email
    Date geburtsdatum
    Boolean geschaeftskunde
    String steuernummer

    static hasMany = [bestellungen: Bestellung]

    static constraints = {
        name(blank: false)
        email(unique: true, email: true)
        geburtsdatum(nullable: true)
        steuernummer(nullable: true)
    }

    String toString() {
        „$name ($email)"
    }
}
```

Listing 1

Listing 1 definiert die Struktur von Kunden-Objekten und bildet mittels des Schlüsselworts „hasMany" eine „1:n"-Beziehung zwischen Kunde und Bestellung ab. GORM erstellt dafür automatisch eine Zuordnungstabelle, um die Beziehung innerhalb der Datenbank abbilden zu können. Dank der Programmiersprache Groovy sind die ohne Sichtbarkeit angegebenen Properties der Klasse „Kunde" genauso wie die abgebildete Beziehung zu den Bestellungen des Kunden über implizite Getter und Setter sichtbar und veränderbar.

Validierung von Objekten

Des Weiteren werden Regeln (sogenannte „Constraints“) definiert, mit deren Hilfe die Domänenklasse validiert werden kann. So darf der Name des Kunden aufgrund des Constraints „blank: false“ nicht leer bleiben. Das Schlüsselwort „unique: true“ stellt die Eindeutigkeit der Werte sicher und bewirkt auch in der Datenbank ein entsprechendes Unique-Constraint für das Feld „email“, während „email: true“ lediglich applikationsseitig eine entsprechende Validierung auf eine gültige E-Mail-Adresse zur Folge hat. Dies ist aus Applikationssicht allerdings transparent.

Neben einer Vielzahl standardmäßig verfügbarer Validatoren können auch eigene definiert werden, um zum Beispiel Abhängigkeiten zwischen Eigenschaften der Klasse zu bestimmen. In Listing 2 machen wir die Steuernummer für Geschäftskunden obligatorisch.

```
steuernummer(validator: { val, obj ->
    if (obj.geschaeftskunde && !val) false
    else true
})
```

Listing 2

Repräsentation in der Datenbank

Das Mapping zur Datenbank – also die Abbildung auf Datenbanktabellen – kann ebenfalls mit einer einfachen Syntax (auch hier handelt es sich um eine DSL) beeinflusst werden. Im folgenden Beispiel (siehe Listing 3) wird der Tabellenname nicht den Konventionen überlassen, sondern explizit auf „customers“ geändert, und auch die Eigenschaft „geschaeftskunde“ wird in der Datenbank in einer Spalte mit dem Namen „businessCustomer“ gespeichert. Ferner soll die Beziehung des Kunden zu „bestellungen“ direkt beim Laden des Kunden aufgelöst und ebenfalls aus der Datenbank abgefragt werden.

```
static mapping = {
    table „customers“
    geschaeftskunde column: „businessCustomer“
    bestellungen lazy: false
}
```

Listing 3

Es besteht darüber hinaus noch eine Vielzahl weiterer Möglichkeiten, an dieser Stelle Einfluss auf die Datenbank zu nehmen, wie beispielsweise die Erzeugung von Indizes, das Caching-Verhalten, das Kaskadieren von Operationen über Beziehungen hinweg, Speichern des Datums der letzten Änderung und weitere bereits aus Hibernate bekannte Optionen.

Don’t repeat the DAO

GORM stellt zur Laufzeit weitere Funktionalität an den Domänenklassen bereit, um typische Interaktionsmuster mit der Datenbank nicht in jeder Applikation erneut implementieren zu müssen. Damit die ersten Daten in die Datenbank gelangen, müssen diese erst einmal gegen die in den Constraints formulierten Regeln validiert und im Erfolgsfall gespeichert werden (siehe Listing 4).

```
// Validieren und Speichern eines Kunden
def kunde = new Kunde(name: „Müller“)
if (kunde.validate()) {
    kunde.save()
} else {
    println kunde.errors
}
```

Listing 4

Sind die Daten nun in der Datenbank gespeichert, so möchte man sie zu einem späteren Zeitpunkt wieder im Zugriff haben. Hierfür bieten sich in den einfachen Fällen bei der Selektion der Daten die dynamischen Finder-Methoden an. Unsere Beispielklasse „Kunde“ bietet uns automatisch unter anderem folgende Zugriffsmethoden (siehe Listing 5).

```
// Erster Kunde mit dem Namen „Müller»
Kunde.findByName(‚Müller›)
// Erster Geschäftskunde mit dem Namen
„Müller»
Kunde.findByNameAndGeschaeftskunde(‚M
üller›, true)
// Alle Geschäftskunden
Kunde.findAllByGeschaeftskunde(true)
// Alle Kunden mit einem hinterlegten
Geburtstag
Kunde.findAllByGeburtsdatumIsNotNull()
```

Listing 5

Schnell wird deutlich, dass sich mit diesen Methoden bereits viele typische Fälle abdecken lassen, wenngleich es natürlich Situationen gibt, in denen dies nicht ausreicht. Wenn nach mehr als zwei Parametern gesucht werden soll, so existiert hierfür eine Funktion, der beliebige Suchparameter übergeben werden können (siehe Listing 6).

```
/* Alle Geschäftskunden mit dem Namen
„Müller», deren Geburtstag bekannt ist */
Kunde.findAllWhere(name: „Müller»,
        geschaeftskunde: true, geburtsdatum:
!null)
```

Listing 6

Komplexere Abfragen kann man mit der aus Hibernate bekannten Hibernate Criteria API formulieren. Hier gibt Grails dem Entwickler eine einfache, domänenspezifische Sprache an die Hand, mit deren Hilfe man gut lesbare Abfragen erzeugen kann (siehe Listing 7).

```
/* Alle Geschäftskunden, deren Name mit
„M“ beginnt, deren Geburtstag bekannt ist
und die mindestens eine Bestellung mit einem Warenwert größer als 5.000 Euro haben,
absteigend sortiert nach dem Geburtsdatum
*/
Kunde.createCriteria().list {
    like(„name“, „M%“)
    eq(“geschaeftskunde”, true)
    isNotNull(“geburtsdatum”)
    bestellungen {
        gt(“warenwert”, 5000)
    }
    order(“geburtsdatum”,“desc”)
}
```

Listing 7

Bei allen Abfragen, die potenziell mehr als einen Wert liefern können, besteht die Möglichkeit, zwei Parameter (Anzahl der Datensätze pro Seite und Offset des ersten zu selektierenden Datensatzes) für die seitenweise Aufteilung der Ergebnismenge (Paging) zu übergeben. Zusätzlich zu den dynamischen Finder-Methoden bietet Grails auch die Möglichkeit, nach einem Objekt zu suchen und, falls ein Objekt mit den Suchparametern nicht existiert, dieses zu erzeugen und zurückzugeben. Hier kann man wählen, ob dieses Objekt lediglich applikationsseitig erzeugt oder aber direkt in der Datenbank gespeichert werden soll.

Weiterhin erleichtert Grails das Arbeiten mit Beziehungen zwischen einzelnen Klassen. So existieren implizit Methoden, um Objekte zu einer „1:n“-Beziehung hinzuzufügen (siehe Listing 8).

```
// Hinzufügen einer neuen Bestellung zu
einem Kunden
Kunde.findOrSaveByName(„Müller»).
addToBestellungen(new Bestellung())
```

Listing 8

Standardmäßig arbeitet GORM mit dem Optimistic-Locking-Ansatz, es wird also zu jedem Objekt in der Datenbank eine Versionsnummer geschrieben, die bei Änderungen inkrementiert wird. Beim Speichern kann so die Versionsnummer im Objekt mit der in der Datenbank verglichen und gegebenenfalls auf die zwischenzeitliche Änderung des Objekts seit dem Laden reagiert werden.

Arbeiten mit Controllern

Grails implementiert das Model-View-Controller-Architekturmuster (MVC) und greift unter der Haube auf das Framework Spring Web MVC zurück. Controller bearbeiten Anfragen und erzeugen eine Antwort beziehungsweise bereiten diese vor. Ein Controller kann direkt eine Antwort generieren (beispielsweise für eine REST-Schnittstelle über XML oder JSON) oder er bereitet ein Modell auf und delegiert die Darstellung an eine View. Auch für Controller gelten Konventionen: Der Name eines Controllers muss mit dem Schlüsselwort „Controller“ enden und als Ablageort für diese Artefakte sind das Verzeichnis „grails-app/controllers“ oder ein Unterverzeichnis (im Sinne von Packages) vorgesehen (siehe Listing 9).

```
class KundeController {
    def list() {
        // Verarbeitung einer Anfrage
    }
}
```

Listing 9

Durch das standardmäßige URL-Mapping werden der Name des Controllers und die Namen der Actions, die innerhalb dieses Controllers definiert sind, in für Menschen lesbare Adressen übersetzt. Die oben dargestellte Action „list“ ist über den URI „/kunde/list“ (angehängt an die Adresse der Applikation) erreichbar. Demzufolge können wir in der Entwicklungsphase nach Bereitstellung auf dem Grails-eigenen Tomcat-Server unter „http://localhost:8080/myApp/kunde/list“ auf die Liste der Kunden zugreifen. Grails definiert grundsätzlich eine Standardaktion, die ausgeführt wird, wenn nur der URI des Controllers angegeben wird:

- Wenn nur eine Action existiert, ist das der Default
- Wenn eine Action mit dem Namen „index“ existiert, ist das der Default
- Alternativ kann der Default mittels „static defaultAction = „list““ gesetzt werden

Aus dem Controller kann aus jeder Action heraus auf verschiedene Informationen zugegriffen werden:

- „servletContext“ (auch bekannt als „Applikationskontext“): Daten über die gesamte Web-Applikation hinweg speichern
- „session“: Daten innerhalb einer Benutzersitzung ablegen
- „request“: Speicherung von Daten innerhalb der aktuellen Anfrage
- „params“: Zugriff auf eine Map mit den POST- und GET-Parametern der Anfrage
- „flash“: die Daten in diesem Scope sind nur im aktuellen und nächsten Request sichtbar

Der Flash Scope ist insbesondere für die Anzeige von Meldungen als Reaktion auf Aktionen des Anwenders interessant (siehe Listing 10). Die Nachricht, die in den Flash Scope geschrieben wurde, ist in der „list“-Action immer noch verfügbar und kann in der View entsprechend angezeigt werden.

```
def update() {
    def kunde = Kunde.get(params.id)
    if (!kunde) {
        flash.message = „Der Kunde mit der id
${params.id}
                wurde nicht gefunden.“
        redirect(action: list)
    }
    ...
}
```

Listing 10

Um aus dem Controller eine View zu erzeugen und dieser ein Modell zu übergeben, gibt es grundsätzlich mehrere Möglichkeiten. Entweder wird als Rückgabewert der Action explizit eine Map übergeben oder aber der Controller besitzt eigene Properties, die dann implizit zurückgegeben werden (siehe Listing 11).

```
// Rückgabe über explizite Map
def list() {
    [kunden: Kunde.list()]
}

// Rückgabe implizit über Properties des
Controllers
List kunden
def list() {
    kunden = Kunde.list()
}
```

Listing 11

In beiden Fällen wird Grails versuchen, die View darzustellen, die unter „grails-app/views/kunde/list.gsp“ liegt. Diese kann dann auf das Model zugreifen und das Ergebnis darstellen. Wo in Grails basierend auf dem Prinzip „Convention over Configuration“ auf Konventionen zurückgegriffen wird, besteht auch immer die Möglichkeit, mittels Konfiguration bei Bedarf von der Konvention abzuweichen. So wird im folgenden Beispiel nicht die standardmäßige View „list.gsp“ sondern eine „tabelle.gsp“ für die Anzeige verwendet (siehe Listing 12).

```
def list() {
    render(view: „tabelle“, model: [kunden:
Kunde.list()])
}
```

Listing 12

Rapid Prototyping mittels Scaffolding

In Zeiten der agilen Softwareentwicklung ist die Möglichkeit von Grails, in kürzester Zeit einen Prototyp zu erstellen, diesen sukzessive weiterzuentwickeln und an neue Anforderungen anzupassen, sehr wertvoll. An dieser Stelle unterstützt der Scaffolding-Mechanismus von Grails, mit dem auf der Grundlage der definierten Fachklassen entsprechende CRUD-Oberflächen (Create, Read, Update, Delete) bereitgestellt werden. Hierfür ist das Schlüsselwort „static scaffold = true“ im Controller erforderlich. Dies kann bereits bei der Generierung der Klasse automatisch eingefügt werden.

Die Grails-Konsole unterstützt mit einfachen Kommandos die Erzeugung von erforderlichen Artefakten und Teilen des Quellcodes. Abbildung 2 zeigt, wie mit dem Befehl „grails create-app auftragsverwaltung“ eine Grails-Applikation angelegt und der Applikationsrahmen geschaffen wird. In dieser Applikation werden mit „grails create-domain-class demo.Kunde“ die erste Fachklasse erzeugt und mit dem Befehl „grails create-scaffold-controller demo.Kunde“ die bereits vorgestellten CRUD-Funktionalitäten implementiert.

Die Applikation kann im Anschluss mit dem Grails-Kommando „grails run-app“ auf einer eigens dafür gestarteten Tomcat-Instanz bereitgestellt und ausprobiert werden. Durch das Scaffolding sind dabei nicht nur die Funktionalitäten im Controller bereitgestellt, sondern auch die dafür erforderlichen Views automatisch generiert.

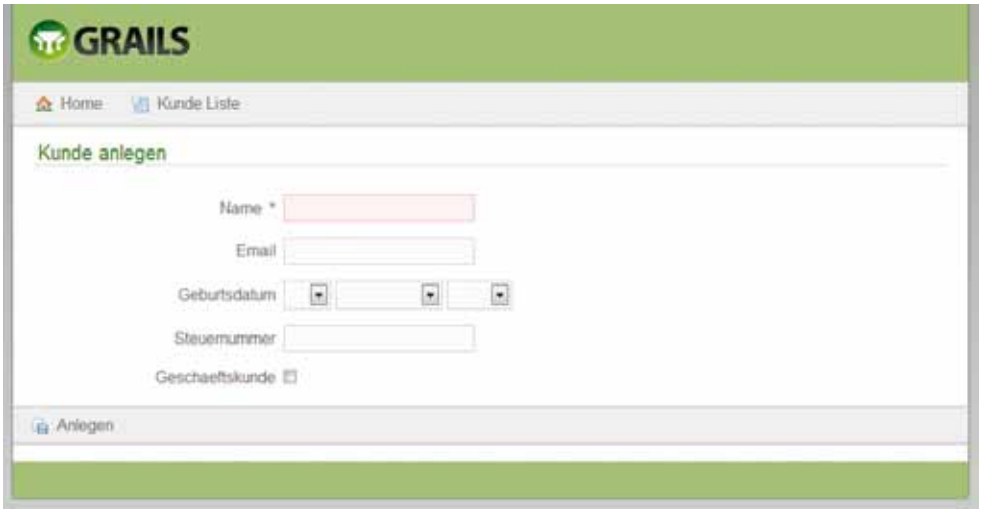

Abbildung 2: Erzeugung der Applikation mit der Grails Console

Abbildung 3: Listenansicht der existierenden Kunden

Abbildung 4: Maske zur Anlage eines neuen Kunden

Von einem solchen Stand ausgehend können nun Aktionen hinzugefügt, überschrieben und an die Anforderungen an die Applikation angepasst werden. In konkreten Projekten ist man so in der Lage, dem Kunden innerhalb weniger Tage (teilweise sogar weniger Stunden) einen ersten Eindruck von der Anwendung zu verschaffen und vor allem schnell festzustellen, ob man sein Problem verstanden hat und mit der Lösung auf dem richtigen Weg ist (siehe Abbildungen 3 bis 5).

Test Driven Development

Schon in der Abbildung 2 wird ersichtlich, dass automatisierte Tests einen hohen Stellenwert bei der Entwicklung mit Groovy und Grails einnehmen. Sämtliche Kommandos zur Erzeugung neuer Artefakte (Fachklassen, Controller etc.) legen automatisch auch die entsprechenden Unit-Test-Klassen an. Grails bietet für das Unit-Testing zusätzlich zu den bereits in Groovy von Haus aus verfügbaren Mock-Funktionalitäten ein eigenes Mock-Framework an, das uns erlaubt, die Funktionalität sämtlicher Artefakte einer Grails-Anwendungen unter Ausblendung der Abhängigkeiten zu Infrastruktur und anderen Artefakten durch Tests abzusichern.

Neben Unit-Tests können außerdem auch Integrationstests gegen eine laufende Datenbank automatisiert werden. Grails sieht hier wie aus Abbildung 1 ersichtlich schon bei der Erzeugung einer neuen Applikation das Verzeichnis „test/integration“ vor. Darüber hinaus existiert eine Vielzahl von Plug-ins, die zum Beispiel bei der Testdatengenerierung unterstützen oder funktionale Tests über die Oberfläche der laufenden Grails-Anwendungen ermöglichen.

Erweiterung der Anwendung durch Plug-ins

Grails liefert mit einem ausgefeilten Plug-in-Mechanismus die Antwort auf fehlende Funktionalität im Framework. Grails ist von Haus aus stark modularisiert aufgebaut und selbst einige der Basisfunktionen sind bereits in Plug-ins ausgelagert und können

so bei Bedarf hinzu- oder weggenommen werden. Grails-Plug-ins sind selber gültige Grails-Applikationen, die über einen Plug-in-Deskriptor genauer beschrieben werden.

Mittlerweile gibt es über 700 öffentlich verfügbare Plug-ins, die bei den verschiedensten Aufgaben unterstützen und mit dem Grails-Kommando „grails install-plugin <pluginname>“ zur eigenen Applikation hinzugefügt werden können. Zu den bekanntesten Plug-ins zählen:

- „hibernate-plugin“ stellt GORM als Brücke zwischen Hibernate und Grails bereit
- „mail-plugin“ liefert eine eigene domänenspezifische Sprache und die Funktionalität zum Versenden von E-Mails
- „quartz-plugin“ erlaubt die Steuerung von Jobs mittels Zeit-Intervall oder Cron-Expression innerhalb von Grails-Anwendungen
- „spring-security-core-plugin“ dient zur Absicherung der Anwendung unter Verwendung eines Rollen- und Rechtekonzepts für angemeldete Benutzer
- „file-uploader-plugin“ unterstützt beim Hochladen von Dateien unter Berücksichtigung von Dateigrößen, Dateitypen und Ablageorten
- „tomcat-plugin“ stellt einen Tomcat-Applikationsserver während der Entwicklung bereit, auf dem die Anwendung ausgeführt werden kann

Implementierung der Oberfläche

Die View-Logik in Grails basiert auf der Open-Source-Bibliothek „SiteMesh“. Diese ist ein Layout-Framework und hilft dabei, ein Template aufzubauen, das mit den Inhalten der verschiedenen Views dynamisch gefüllt wird. Dabei stehen insbesondere eine einheitliche Navigation und ein konsistentes „Look & Feel“ im Vordergrund.

Grails arbeitet mit Groovy Server Pages (kurz: GSPs). Diese bestehen aus HTML-Code, angereichert um GSP-Tags. Durch SiteMesh ist es möglich, einzelne Teile der GSP in Templates auszulagern. So gibt es in Grails standardmäßig eine „main.gsp“, die das Layout und auch die Importe für Style-Sheet-Definitionen enthält. Eine einfache GSP, um die Liste der Kunden anzuzeigen, könnte wie in Listing 13 aussehen.

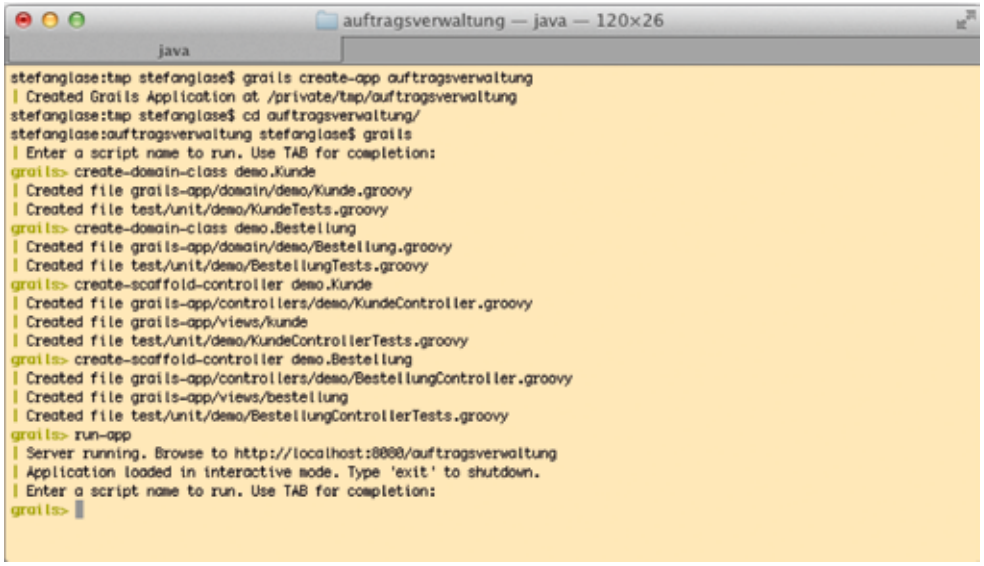

Abbildung 5: Detailansicht eines Kunden

```
<html>
  <body>
    <h1>Liste der Kunden</h1>
    <g:if test="${flash.message}">
      <div class="message"
role="status">${flash.message}</div>
    </g:if>
    <table>
      <g:each in="${kundeInstanceList}"
status="i" var="kundeInstance">
            <tr class="${(i % 2) == 0 ? 'even' :
'odd'}">
              <td>
          <g:link action="show"
id="${kundeInstance.id}">
            ${fieldValue(bean: kundeInstance,
field: "name")}
          </g:link>
        </td>
        <td>
          ${fieldValue(bean: kundeInstance,
field: "email")}
        </td>
        <td>
          <g:formatDate
date="${kundeInstance.geburtsdatum}"/>
        </td>
      </tr>
          </g:each>
    </table>
  </body>
</html>
```

Listing 13

Wiederverwendung von Oberflächenkomponenten

Sehr gut kann man im oben stehenden Codebeispiel den Gebrauch von Grails Taglibs sehen (beispielsweise „<g:formatDate/>“ zur Formatierung von Datumswerten). Diese sind in jeder GSP verfügbar und unterstützen durch die Wiederverwendung von Oberflächen-Komponenten bei der Vermeidung von redundantem Code.

Mit sehr geringem Aufwand ist es möglich, auch eigene Taglibs zu erstellen. Diese weiteren Artefakte werden unter „grails-app/taglib“ abgelegt und enden im Namen auf „TagLib“. Ein einfacher Tag kann wie in Listing 14 implementiert werden.

```
class KundenTagLib {
    static namespace = "kunden"
    def daten = { attrs, body ->
      def kunde = attrs.kunde
      out << "<div>"
      out << "<p>Name: ${kunde.name}</p>"
      out << "<p>Email: ${kunde.email}</p>"
      out << "</div>
    }
}
```

Listing 14

Eine auf diese Weise implementierte Taglib kann ab sofort in jeder View angesprochen und verwendet werden. Im Beispiel gibt „<kunden:daten kunde="${kundeInstance}" />“ den Namen und die E-Mail-Adresse des Kunden aus.

Fazit

Die vielversprechende Kombination der dynamischen Programmiersprache Groovy und der Web-Anwendungs-Plattform Grails, eine auf etablierten und erprobten Open-Source-Frameworks basierende Plattform, zeigt nicht nur in kleinen Beispiel-

Anwendungen ihre Stärken, sondern hat sich für uns auch bereits in großen Enterprise-Anwendungen erfolgreich unter Beweis gestellt. Durch eine ausdrucksstarke Sprache entsteht gut lesbarer und damit wartbarer Quellcode. Dank einer Vielzahl stabiler und funktionsreicher Plug-ins braucht so mancher Framework-Code nicht mehr selber geschrieben werden. Und aufgrund der strukturellen Konventionen innerhalb von Grails-Anwendungen fällt Entwicklern die Einarbeitung in neue Projekte um ein Vielfaches leichter. Alle diese Faktoren helfen dabei, dass die Fachlichkeit einer Anwendung wieder mehr in den Vordergrund rückt. Kunden werden es danken, wenn ihre Anforderungen in kürzerer Zeit umgesetzt und an geänderte Bedingungen angepasst werden können.

Stefan Glase
stefan.glase@opitz-consulting.com
Christian Metzler
christian.metzler@opitz-consulting.com

Stefan Glase (rechts) ist Senior Consultant im Bereich Application Engineering bei der OPITZ CONSULTING GmbH. Er beschäftigt sich seit mehreren Jahren mit der Architektur und Implementierung von Enterprise-Applikationen. Seine Schwerpunkt sind Enterprise Java mit dem Spring Framework sowie Groovy & Grails. Er ist Autor von Fachartikeln und spricht regelmäßig auf Fachkonferenzen.

Christian Metzler ist Consultant im Bereich Application Engineering bei der OPITZ CONSULTING GmbH. Er beschäftigt sich seit einigen Jahren insbesondere mit der Entwicklung von Web-Anwendungen mit Schwerpunkt auf der Sprache Groovy und dem leichtgewichtigen Framework Grails.

„Java besitzt immer noch ein enormes Potenzial …“

Usergroups bieten vielfältige Möglichkeiten zum Erfahrungsaustausch und zur Wissensvermittlung unter den Java-Entwicklern. Sie sind aber auch ein wichtiges Sprachrohr in der Community und gegenüber Oracle. Wolfgang Taschner, Chefredakteur von Java aktuell, sprach darüber mit Stefan Koospal, dem Vorsitzenden der Sun User Group Deutschland.

Wie bist du zur Sun User Group Deutschland gekommen?

Koospal: Ich bin 1993 zur Sun User Group Deutschland (SUG) gestoßen, weil ich als System-Administrator im Mathematischen Institut in Göttingen damals ein reines Sun-Umfeld zu verwalten hatte. Die jährlichen Treffen der User Group boten mir eine Möglichkeit, an wichtige Informationen zu gelangen und Kontakte zu anderen System-Administratoren zu knüpfen. 1993 war das Internet gerade erst dabei, groß zu werden, und die Verbindungen waren alle noch sehr schmalbrüstig. Da waren die CD-Kollektionen der SUG mit freier Software für Sun-Systeme eine wichtige Quelle, um mein System besser auszustatten.

Wie ist die Sun User Group Deutschland organisiert?

Koospal: Die SUG ist deutschlandweit als gemeinnütziger, eingetragener Verein organisiert.

Was zeichnet die Sun User Group Deutschland aus?

Koospal: Die SUG besteht jetzt fünfundzwanzig Jahre und kann damit schon auf etwas Geschichte zurückblicken. Wichtige Technologien wie das Internet und Java sind im Sun-Umfeld entstanden und gefördert worden. Anfang der 1990er Jahre bildeten Sun-Systeme das Rückgrat des Internets. IBM hatte ein eigenes Netz, Apple machte Appletalk, Microsoft sah im Internet keine wichtige Technologie und Linux war noch nicht so weit. Die Aufgaben des Vereins liegen inzwischen darin, die Nutzer von Sun-Technologien wie Java, Solaris, ZFS oder Staroffice weiter zu unterstützen, wenn nötig auch bei der Migration auf andere Systeme.

Wie viele Veranstaltungen gibt es pro Jahr?

Koospal: Wir veranstalten jährlich mit der Java User Group Deutschland e.V. die Source Talk Tage, bei denen wir Wissen zu verschiedenen Sun-Technologien wie Solaris, Staroffice und insbesondere Java an die Mitglieder der beiden Vereine und alle, die sich angesprochen fühlen, zu günstigen Konditionen weitergeben. Besonders wichtig für uns ist, dass unsere Mitglieder und Sponsoren den Studierenden die kostenfreie Teilnahme ermöglichen.

Was motiviert dich besonders, als Vorstand die Sun User Group Deutschland zu führen?

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 75 EUR. Weitere Informationen unter www.doag.org/go/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. 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.