2-2013 | Sommer | www.ijug.eu

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

Java aktuell

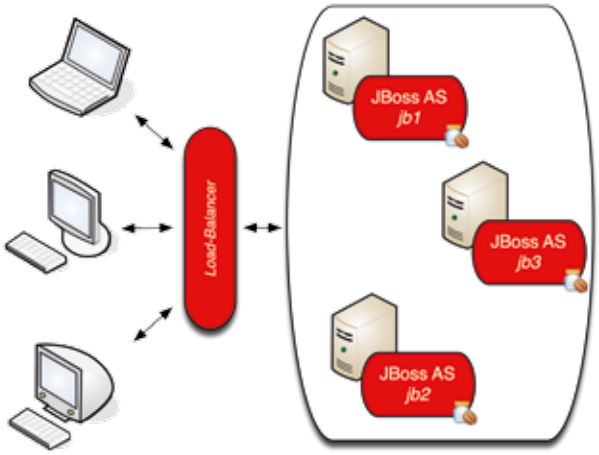
Java verbreitet sich überall

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

iJUG Verbund

Sonderdruck

4 191978 304903 02



Lastverteilung zum Erreichen der Hochverfügbarkeit, Seite 20

Build-Kreislauf beim Continuous Integration, Seite 59

Linked-Data-Praxis: Daten bereitstellen und verwerten

Angelo Veltens, http://datenwissen.de

Ein Artikel in der letzten Ausgabe hat in die Grundlagen von Linked Data eingeführt und gezeigt, wie daraus ein Web aus Daten entstehen kann. Linked Data ist jedoch keine bloße Theorie: Mit wenigen Zeilen Code kann eine Anwendung Teil dieses Daten-Webs werden. Der Beitrag zeigt, wie man Linked Data veröffentlichen und in seinen Anwendungen verwerten kann.

Wer zum ersten Mal von Linked Data hört, ist nicht selten von der Idee fasziniert, hält sie aber zugleich für utopisch und noch nicht Praxis-relevant. Dabei ist die Veröffentlichung und Verwertung von Linked Data nicht schwieriger als die von JSON- oder XML-Dokumenten. Durch die zusätzliche Semantik und weltweite Verlinkbarkeit der Daten wird die Verwertung von Informationen aus einer Vielzahl von heterogenen Systemen sogar einfacher.

Um dies zu verdeutlichen, werden wir in diesem Artikel auf die Hotel-Bewertungs-Plattform zurückkommen, die in der letzten Ausgabe als Beispiel diente. Wir werden exemplarisch einen Hotel-Service entwickeln, der Informationen über Hotels als Linked Data publiziert, sowie eine Hotel-Rating-Seite, die diese Informationen mit Bewertungen der Nutzer verknüpft. Wir verwenden dazu das Web-Framework „Grails“ [1] zusammen mit der RDF-Bibliothek „groovyrdf“ [2], die auf Apache Jena [3] aufsetzt.

Das Problem

Werfen wir zunächst nochmals einen Blick auf einen typischen JSON-Webservice. Ein Service, der Informationen über Hotels als JSON bereitstellt, könnte folgenden Datensatz ausliefern (siehe Listing 1).

```
{
  id: 42,
  name: 'Hotel Fiktiva'
  city: 'Berlin'
}
```

Listing 1

Es mag relativ leicht sein, diesen Service anzubinden, dabei koppelt man jedoch seine Anwendung mit eben diesem Service. Die Anwendung muss wissen, über welche URIs sie Daten abrufen kann und in welchem Format die Daten ausgeliefert werden. Man muss fest implementieren, dass der Name eines Hotels unter dem Schlüssel „name“ zu finden ist und der Name des Ortes unter „city“. Möchte man weitere Informationen über diesen Ort einbinden, muss man einen entsprechenden Web-Service finden und die Zeichenkette „Berlin“ irgendwie mit den dort zu findenden Daten verknüpfen. Die Daten haben keinen „Kontext“, keinen Bezug zu anderen Datensätzen und keine Semantik.

Das mag bei nur wenigen anzubindenden Diensten einer in sich geschlossenen System-Landschaft praktikabel sein, bei der alle zu nutzenden Systeme bekannt sind. Mit Linked Data jedoch kann der gesamte Datenbestand des World Wide Web erschlossen werden.

Linked Data veröffentlichen

Wie also könnte ein Hotel-Service aussehen, der Linked Data bereitstellt? Eins vorweg: Man muss nicht auf JSON oder XML verzichten, nur weil man Linked Data anbietet. Dank Content-Negotiation kann beides nebeneinander harmonieren. Gemäß den Linked-Data-Prinzipien benötigen wir zuerst eine URI als Name für unsere Dinge, also die Hotels. Für die Wahl passender URIs gibt es bereits eine Reihe von Patterns [4]. In erster Linie ist es aber wichtig, dass wir zwischen einem Hotel an sich und dem Dokument über dieses Hotel unterscheiden. Für unsere Beispielanwendung sind die URIs nach folgendem Muster aufgebaut: „http://hotels.datenwissen.de/hotel/<ID>#it“. <ID> ist dabei eine interne ID aus der verwendeten relationalen Datenbank. Sie ist für Anwendungen, die unsere Daten nutzen wollen, irrelevant und wird hier nur verwendet, um jedem Hotel eine eindeutige URI zu geben. Wie in der letzten Ausgabe beschrieben, verwenden wir den Fragment-Identifier „#it“, um das Hotel vom Dokument zu unterscheiden, das durch folgende URI identifiziert wird: „http://hotels.datenwissen.de/hotel/<ID>“. In der UrlMappings.groovy von Grails lässt sich leicht ein Resource-Mapping anlegen, das diese Vorgaben abbildet (siehe Listing 2).

```
static mappings = {
  "/hotel/$id" (resource:"hotel")
}
```

Listing 2

Da der Fragment-Identifier vom Client automatisch abgeschnitten wird, brauchen wir uns diesbezüglich um nichts weiter zu kümmern. Ein Client wird immer nur das Dokument (ohne „#it“) anfordern. Ein Hotel können wir schließlich nicht über HTTP übermitteln; wir sind auf Dokumente mit Informationen zu Hotels beschränkt. Wir werden jedoch die URI des Hotels (mit „#it“) verwenden, um in diesen Dokumenten Aussagen über das Hotel zu treffen. Mithilfe von Content-Negotiation können wir das angeforderte Dokument in einer Vielzahl von Formaten ausliefern. Für das

Auge könnten wir die Informationen zum Beispiel als HTML aufbereiten, gleichzeitig ist es aber auch möglich, RDF und JSON bereitzustellen.

Für unser Beispiel beschränken wir uns auf JSON und RDF in der Turtle-Syntax. Das oben beschriebene URL-Mapping sorgt dafür, dass die show()-Action des HotelController aufgerufen wird, wenn ein GET-Request den Server erreicht (siehe Listing 3).

In Zeile 2 laden wir zunächst das Hotel mit der angeforderten ID aus der Datenbank. Wird kein solches gefunden, liefern wir den Status-Code 404 (Not Found) zurück. Im „withFormat-Block" unterscheiden wir anschließend nach dem vom Client gewünschten Format. „json" ist bereits von Grails auf die MIME-Types „text/json" und „application/json" gemappt; „ttl" müssen wir selbst durch einen zusätzlichen Eintrag in der Config.groovy auf „text/turtle" mappen (siehe Listing 4).

Zu beachten ist außerdem, dass „grails.mime.use.accept.header" auf „true" gesetzt ist, damit Grails den Accept-Header auswertet. Unsere Anwendung ist nun in der Lage, sowohl JSON als auch Turtle-RDF unter der gleichen URI bereitzustellen – je nach Vorzügen des anfragenden Clients. Wir müssen im nächsten Schritt lediglich ein RDF-Dokument aus den Daten unserer Datenbank erzeugen.

Die Bibliothek „groovyrdf" ermöglicht es, RDF-Daten in einer an Turtle angelehnten Syntax zur Laufzeit zu erzeugen. Sie baut auf dem Jena-Framework auf, ist dabei weniger mächtig, dafür aber leichter zu verstehen. RDF lässt sich mit wenigen Code-Zeilen erzeugen oder auslesen. In einem Grails-Service erzeugen wir nun einige RDF-Tripel mit Aussagen zum gewünschten Hotel (siehe Listing 5).

In Zeile 3 instanziieren wir dazu einen „JenaRdfBuilder" und in Zeile 4 generieren wir dynamisch eine URI für das gewünschte Hotel – basierend auf der URI des Hotel-Dokuments, ergänzt um „#it". In den darauffolgenden Zeilen werden innerhalb der geschweiften Klammern vier Aussagen zu dieser URI (also dem Hotel) getroffen. Dazu werden einige tatsächlich existierende Ontologien verwendet: Die Accommodation Ontology (siehe „http://purl.org/acco/ns#") bietet ein Vokabular für Hotels, Ferienwohnungen und ähnliche Unterkünfte. Sie ist eine Ergänzung zur eCommerce-Ontologie „GoodRelations" (siehe „http://purl.org/goodrelations/v1#"), die hier auch für den Namen des Hotels verwendet wird. Um den Standort zu beschreiben, greifen wir auf die „Friend-of-a-Friend"-Ontologie (FOAF) zurück, die, ihrem Namen zum Trotz, über „based_near" den Standort aller möglichen „Spatial Things" beschreiben kann.

Eine Aussage erfolgt im einfachsten Fall durch die Angabe des Prädikat-URI, gefolgt vom gewünschten Wert. Die Werte beziehen wir hier direkt aus dem Hotel-Objekt, zum Beispiel „hotel.name". Hinter „hotel.basedNear" verbirgt sich jedoch kein Objekt-Literal, sondern ein URI, auf den wir verlinken möchten. Daher wird bei „based_near" nochmals ein Klammerpaar geöffnet, um eine „Sub-Ressource" mit dem in „hotel.basedNear" enthaltenen URI zu erzeugen. Das leere Klammerpaar dahinter sagt aus, dass wir an dieser Stelle keine Aussagen zu dieser verlinkten Ressource treffen. Ein Client kann jedoch den URI abrufen, um weitere Informationen zu erhalten.

Statt einen eigenen „Geo-Service" zu entwickeln, können wir einfach auf bestehende Dienste verlinken. „geonames.org" stellt bereits Linked Data für eine Vielzahl von geografischen Orten zur Verfügung. Die Stadt Berlin ist dort durch den URI „http://sws.geonames.org/2950159/" identifiziert. Listing 6 zeigt das RDF-Dokument, das durch unseren Hotel-Service generiert wird. Um unseren Service zu vervollständigen, müssen wir die RDF-Daten im Controller als Dokument zurückliefern (siehe Listing 7).

Über unseren Service erzeugen wir die RDF-Daten und schreiben sie anschließend über die „write()"-Methode im Turtle-Format in einen „StringWriter". Über die Grails-„render()"-Methode senden wir den resultierenden String mit korrektem Content-Type zurück an den Client.

Der Dienst ist tatsächlich unter „http://hotels.datenwissen.de" online. Um eine Liste der gespeicherten Hotels zu erhalten, ruft man einfach diesen URI ab und folgt dann den Links zu weiterführenden Daten der Hotels. Auch der Quellcode ist online zugänglich [5]. Es empfiehlt sich, damit zu experimentieren.

Linked Data konsumieren

In einer weiteren Anwendung werden nun die Daten des Hotel-Service konsumiert.

```
def show(long id) {
  def hotel = Hotel.get(id)
  if (!hotel) {
    render(status: 404)
    return
  }
  withFormat {
    ttl { /* render text/turtle */ }
    json { render hotel as JSON }
  }
}
```

Listing 3

```
grails.mime.types = [
    // [...]
    json:          ['application/json','text/json'],
    ttl:           'text/turtle' // Neu
]
```

Listing 4

```
RdfData hotelToRdf(Hotel hotel) {
  def serverUrl = grailsApplication.config.grails.serverURL
  new JenaRdfBuilder().build {
    "${serverURL}/hotel/${hotel.id}#it" {
      a 'http://purl.org/acco/ns#Hotel'
      'http://purl.org/goodrelations/v1#name' hotel.name
      'http://purl.org/acco/ns#numberOfRooms' hotel.roomCount
      'http://xmlns.com/foaf/0.1/based_near' {
        "$hotel.basedNear" {}
      }
    }
  }
}
```

Listing 5

```
<http://hotels.datenwissen.de/hotel/1#it>
  a <http://purl.org/acco/ns#Hotel> ;
  <http://purl.org/acco/ns#numberOfRooms> "200" ;
  <http://purl.org/goodrelations/v1#name> "Hotel Fiktiva" ;
  <http://xmlns.com/foaf/0.1/based_near>
    <http://sws.geonames.org/2950159/> .
```

Listing 6

```
RdfData rdfData = rdfService.hotelToRdf(hotel)
StringWriter out = new StringWriter()
rdfData.write(out, RdfDataFormat.TURTLE)
render(contentType: 'text/turtle', text: out.toString())
```

Listing 7

```
def index() {
  def rdfLoader = new JenaRdfLoader ()
  RdfData rdfData = rdfLoader.load(
    'http://hotels.datenwissen.de/'
  )
  List<RdfResource> hotelResources = rdfData.listSub-
jects(
    'http://purl.org/acco/ns#Hotel'
  )
  List hotels = hotelResources.collect { res ->
    [
      uri: res.uri,
      name: res("http://purl.org/goodrelations/v1#name")
    ]
  }
  return [hotels: hotels]
}
```

Listing 8

```
def gr = new RdfNamespace('http://purl.org/goodrela-
tions/v1#')
String name = res(gr.name)
```

Listing 9

```
beans = {
  rdfLoader (JenaRdfLoader) {}
  gr (RdfNamespace, 'http://purl.org/goodrelations/v1#')
  acco (RdfNamespace, 'http://purl.org/acco/ns#')
  geo (RdfNamespace, 'http://www.geonames.org/ontol-
ogy#')
  foaf (RdfNamespace, 'http://xmlns.com/foaf/0.1/')
}
```

Listing 10

```
<g:each in="${hotels}" var="hotel">
    <g:link controller="rating" action="show"
      params="[uri:hotel.uri]">
        ${hotel.name}
    </g:link>
</g:each>
```

Listing 11

```
def hotelResource = rdfLoader.loadResource(uri)
String name = hotelResource (gr.name)
int numberOfRooms = hotelResource (acco.numberOf-
Rooms)
String cityUri = hotelResource(foaf.basedNear).uri
RdfResource cityResource = rdfLoader.
loadResource(cityUri)
String cityName = cityResource(geo.name)
```

Listing 12

Diese Hotel-Rating-Web-Anwendung soll es den Besuchern ermöglichen, Hotels zu bewerten, ohne dass die Anwendung selbst Daten über Hotels speichert. Um das Beispiel einfach zu halten, werden wir zunächst nur den oben gezeigten Hotel-Service abfragen. Darauf aufbauend ist es jedoch leicht, weitere Teile des „Web of Data" zu erschließen.

Unter „http://hotels.datenwissen.de" stellt der Hotel-Service eine Liste der verfügbaren Hotels zur Verfügung. Für die Hauptseite der Hotel-Rating-Anwendung fragen wir diese Liste ab, um aus den Daten eine Auswahl der verfügbaren Hotels zu rendern (siehe Listing 8).

In den Zeilen 3 bis 5 werden über eine „RdfLoader"-Instanz RDF-Daten vom Hotel-Service abgefragt. Um Content-Negotiation kümmert sich der RdfLoader selbst, wir müssen uns nicht um unterschiedliche RDF-Syntaxen kümmern und nichts selbst parsen. Stattdessen erhalten wir ein „RdfData"-Objekt, aus dem wir Aussagen über die vorhandenen Ressourcen auslesen können. Dies erfolgt in den Zeilen 6 bis 8. Über „listSubjects()" erhalten wir alle Ressourcen des Typs „http://purl.org/acco/ns#Hotel", also alle Hotels, die uns der Service lieferte. Die Daten, die uns interessieren, URI und Name der Hotels, geben wir als Map zurück, sodass sie für das Rendern der Groovy Server Page (GSP) verwendet werden können. Über „res.uri" bekommen wir den URI der Ressource „res" und über den Aufruf „res(String)" kommen wir an die Informationen heran, die eine Ressource unter dem übergebenen Prädikat bereitstellt. „res(‚http://purl.org/goodrelations/v1#name')" liefert uns somit den Namen des Hotels.

Statt mit langen URIs zu hantieren, lassen sich auch Namespaces deklarieren. Legen wir einen Namespace namens „gr" für die Ontologie „http://purl.org/goodrelations/v1#" an, können wir anschließend über die Kurzschreibweise „gr.name" auf die gleiche Eigenschaft zugreifen (siehe Listing 9). Um weder den RdfLoader noch die Namespaces ständig instanziieren zu müssen, können wir sie bequem als Spring-Beans in der Datei „resources.groovy" hinterlegen (siehe Listing 10). In der GSP-Seite rendern wir für jedes Hotel einen Link zu „/rating/show" mit dem Hotel-URI als Parameter (siehe Listing 11). Abbildung 1 zeigt das Ergebnis.

Klickt ein Nutzer auf einen der Links, wird die „show()"-Action des RatingController aufgerufen. Sie nutzt den übergebenen URI, um weitere Informationen zum Hotel nachzuladen (siehe Listing 12).

Über „loadResource()" laden wir gezielt Daten zu der durch den URI identifizierten Ressource, also das gerade angeforderte Hotel. Die Attribute „gr.name" und „gr.numberOfRooms" werden auf die oben vorgestellte Weise abgefragt. Interessant wird es wieder ab Zeile 4: Hinter dem Prädikat „foaf.basedNear" verbirgt sich kein Objekt-Literal, sondern eine weitere Ressource – nämlich der Ort, an dem sich das Hotel befindet. Wir fragen daher dessen URI ab und nutzen ihn, um mit dem rdfLoader Daten über den Ort zu laden. Dabei kann es uns egal sein, welcher Service sich hinter diesem URI verbirgt, solange er wieder Linked Data bereitstellt. Um den Namen des Orts zu erhalten, greifen wir auf die Eigenschaft „geo.name" zu. Die gewonnenen Daten können wir wieder an die GSP liefern, um sie zu rendern.

Hotels bewerten

Das Bewerten von Hotels hat anschließend nicht mehr viel mit Linked Data zu tun. In einer „save()"-Action erzeugen wir

Abbildung 1: Auflistung der vom Hotel-Service gelieferten Hotels

lediglich ein neues HotelRating-Objekt mit den vom User eingegebenen Daten: „new HotelRating(ratedHotelUri: uri, rating: rating, comment: comment).save()“. Zu beachten ist hier lediglich, dass wir den URI des bewerteten Hotels speichern, um festzuhalten, worauf sich die Bewertung bezieht. Um alle Bewertungen zu einem Hotel zu finden, suchen wir nach all jenen, die sich auf den URI des Hotels beziehen: „HotelRating.findAllByRatedHotelUri(uri)“.

Alles Weitere ist reine Grails-Funktionalität, auf die an dieser Stelle nicht näher eingegangen wird. Abbildung 2 zeigt die fertige Seite mit den vom Hotel-Service abgefragten Informationen und den von der Anwendung selbst gespeicherten Bewertungen. Auch der Quellcode der Hotel-Rating-Anwendung ist online frei zugänglich [6]. Eine laufende Instanz ist unter http://hotel-rating.datenwissen.de dargestellt.

Das Beispiel weitergedacht

Anhand eines konkreten Beispiels haben wir nun sowohl publiziert als auch konsumiert und uns bei der Abfrage von Hotels auf einen ganz konkreten Service beschränkt. Dies muss jedoch nicht sein. Zum Beispiel könnten wir eine semantische Suchmaschine nutzen, um alle Ressourcen vom Typ „http://purl.org/acco/ns#Hotel“ im Web of Data zu finden und zur Bewertung anzubieten. Die dadurch entstehende Heterogenität der RDF-Aussagen muss allerdings entsprechend in der Implementierung berücksichtigt werden. Nicht alle Hotel-Ressourcen im Web of Data werden exakt dem gleichen Aufbau folgen. Im Gegensatz zu den Key-Value-Paaren eines JSON-Datensatzes sind bei RDF aber auch die Prädikate durch URIs identifiziert und durch strukturierte Daten beschrieben. So lässt sich zum Beispiel ausdrücken, dass zwei Prädikate das Gleiche bedeuten. Erfahrungsgemäß setzen sich bestimmte Ontologien als De-facto-Standards durch, wie die FOAF-Ontologie für die Beschreibung von Personen und deren Beziehungen zueinander.

Eine Anwendung, die die Daten aus dem ganzen Web abfragt, ohne genau zu wissen, aus welchen Diensten sie stammen, sollte dennoch tolerant gegenüber dem Aufbau der Ressourcen sein. Sie sollte auf Daten, die nicht in der erwarteten Form ausgedrückt werden, im Zweifel verzichten können oder die Ressource verwerfen. Dies ist jedoch ein kleiner Verlust im Verhältnis zu dem riesigen Datenpool, der uns zur Verfügung steht, wenn wir das Web als globale und dezentral verwaltete Datenbank nutzen.

Weiterhin ist es erstrebenswert, dass Dienste, die Linked Data konsumieren, auch selbst wieder Linked Data veröffentlichen. Die Hotel-Rating-Anwendung könnte die Bewertungen ebenfalls als Linked Data bereitstellen und mit den Hotel-URIs verlinken. Auf diese Weise können später alle Bewertungen zu einem Hotel im Web abgefragt werden. Es könnte viele verschiedene Bewertungsplattformen geben und dennoch kann eine Suche alle verfügbaren Bewertungen einbeziehen, unabhängig davon, auf welcher Plattform sie abgegeben wurden.

Abbildung 2: Anzeige eines Hotels und seiner Bewertungen

Social Web of Data

Was unserem Hotel-Rating-Dienst noch fehlt, ist eine Authentifizierung der Nutzer. Prinzipiell können auch Personen über einen URI identifiziert und Informationen mit ihnen verlinkt werden. Wenn sich ein Nutzer über seinen URI authentisiert, könnten seine Bewertungen und Kommentare mit ihm verlinkt werden, ohne dass unser Dienst eine eigene Benutzerverwaltung bereitstellen muss und auch ohne dass der Nutzer bei einem zentralen Dienst wie Facebook angemeldet sein muss. Erfreulicherweise ist genau dies mit dem WebID-Protokoll [7] möglich. In der nächsten Ausgabe werden wir uns dieses Protokoll genauer ansehen.

Fazit

Das Bereitstellen und Verwerten von Linked Data ist nicht schwieriger als der Umgang mit XML-Dokumenten. Die Verlinkung der Daten macht es leicht, Dienstunabhängig auf Daten zuzugreifen. Statt über proprietäre APIs an einzelne Dienste anzudocken, behandeln wir Daten wie den Rest des Webs: Wir greifen über URIs auf sie zu und folgen den in ihnen enthaltenen Links zu weiteren Daten. Linked Data eignet sich daher besonders beim Zusammenführen von Daten aus einer Vielzahl heterogener Systeme.

Referenzen

[1] http://grails.org
[2] http://datenwissen.de/groovyrdf
[3] http://jena.apache.org
[4] http://patterns.dataincubator.org/book
[5] https://github.com/angelo-v/hotels
[6] https://github.com/angelo-v/hotel-rating
[7] http://webid.info

Angelo Veltens
angelo.veltens
@online.de

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

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.