JAVA Mag inkl. CD

Mobile Welten: Java ME und das Web »94

S&S

Deutschland €8,50 Österreich €9,80 Schweiz sFr 16,80 1.2011

Java™ magazin

Java • Architekturen • SOA • Agile

www.javamagazin.de

CD-INHALT

Exploring alternative Concurrency Paradigms on the JVM

Video von Jonas Bonér auf der JAX 2010

WEITERE INHALTE

Git

Apache Maven 3

Gradle

Gracelets

JSF Flex

GwtAI

Monotone

Bazaar

Mercurial

Alle CD-Infos ab Seite 3

4 194580 708507 01

Datenträger enthält Info- und Lehrprogramme gemäß §14 JuSchG

inspectIT
Java-Performance-analyse »12

Scala-Tutorial
Game of Life mit Scala »98

Web
Webframeworks Eine Kategorisierung »34

GIT

... verändert die Softwareentwicklung

Vom Klonen und Pushen in verteilten Welten »52

Gradle

... wird den Build schon schaukeln »80

Android-Konzepte

Applications & Activities »86

No REST for the Wicket

RESTful HTTP mit Apache Wicket »43

Arquillian

Leichtgewichtig testen für Java EE »72

Wie REST-konform ist das Webframework wirklich?

No REST for the Wicket?

REST ist ein aktuell von großer Hype-Energie beschleunigtes Thema, über das viel zu lesen ist. Häufig wird REST bzw. RESTful HTTP jedoch ausschließlich als leichtgewichtige Alternative zu „klassischen" Web Services betrachtet und nicht als Architekturstil, dem auch Webanwendungen folgen sollten. Aber REST als Architektur des Web hat genau darin seine Wurzeln – die Trennung zwischen der Anwendungs-zu-Anwendungs-Kommunikation und der Darstellung einer Benutzeroberfläche für einen Endanwender ist aus dieser Sicht künstlich.

von Michael Plöd und Stefan Tilkov

Im Web-Services-Umfeld hat man mit Frameworks wie Restlet oder dem Standard-JAX-RS (Java API for RESTful Web Services) erkannt, dass das Verstecken von HTTP, URIs und anderen Webtechnologien bei der Entwicklung eher hinderlich als nützlich ist. Anders sieht es bei Webframeworks aus, insbesondere im Java-Umfeld und dort vor allem bei modernen, komponentenbasierten Bibliotheken. Angeführt von JSF gilt HTML und HTTP hier als notwendiges Übel, dass es zu verstecken gilt. Dies ist ein merklicher Unterschied zur Betrachtungsweise, die sich in alternativen, dynamischen Sprachen wie Ruby mit Rails, Groovy mit Grails oder auch Clojure mit dem Compojure-Framework feststellen lässt: Hier bleibt die Tatsache, dass ein eingehender HTTP-Request auf eine Verarbeitungslogik abgebildet und als Ergebnis eine HTTP-Response generiert wird, sehr sichtbar. Ein populäres, komponentenorientiertes Framework, das antritt, die Entwicklung von Webanwendungen mit Java so einfach und elegant wie möglich zu gestalten, ist Apache Wicket. Das Framework setzt konsequent auf Objektorientierung, Plain Java, Namenskonventionen und reines HTML. In diesem Artikel wollen wir Wicket mit der REST-Brille betrachten und herausfinden, ob sich mit Wicket Anwendungen nicht nur elegant entwickeln lassen, sondern dabei auch den Grundprinzipien einer REST-Architektur gefolgt werden kann.

Apache Wicket

Wicket lässt sich eindeutig in die Kategorie „komponentenbasierte Webframeworks“ einordnen. Beim Schreiben von Webanwendungen auf Basis von Wicket kommen ausschließlich Java und HTML zum Einsatz. Eine gesonderte Konfiguration (außer der *web.xml*) oder weitere Markup-„Sprachen“ sind nicht nötig. Die einfachste mögliche Wicket-Anwendung besteht somit aus folgenden Konstrukten:

- Einer Webanwendungsklasse, die von *WebApplication* erbt
- Einer Klasse für eine Seite, abgeleitet von Page
- Einer HTML-Datei, die sich im gleichen Package wie die Page befindet und den gleichen Namen trägt

Um den Start mit Wicket möglichst einfach zu gestalten, hat das Entwicklerteam einen Maven-Archetyp auf der Homepage bereitgestellt. Das Aufsetzen einer neuen Wicket-Anwendung lässt sich dadurch mit einer einzelnen Kommandozeile bewerkstelligen:

```
mvn archetype:create -DarchetypeGroupId=org.apache.wicket
  -DarchetypeArtifactId=wicket-archetype-quickstart -DarchetypeVersion=1.4.0
                                  -DgroupId=quickstart -DartifactId=example
```

Neben dem *WicketFilter*, der in der *web.xml* konfiguriert wird, sind primär noch die Klassen *WicketApplication.java*, *HomePage.java* und *HomePage.html* von Interesse. Die Wicket-Applikation ist im Fall des eben erwähnten Einsatzes sehr einfach aufgebaut, sie enthält nichts außer einem Verweis auf die Startseite der Anwendung. Im weiteren Verlauf des Artikels wird ihr jedoch noch eine zentrale Rolle bei der Generierung von URLs zukommen:

```
public class WicketApplication extends WebApplication {
    public Class<HomePage> getHomePage() {
        return HomePage.class;
    }
}
```

Die Homepage der Anwendung ist die Klasse *HomePage*. Diese muss von *WebPage* erben und fügt in ihrem Konstruktor der Seite weitere Komponenten hinzu. Im Fall des Beispiels handelt es sich um ein einfaches Label, das Text darstellt. Der Konstruktor einhält noch ein Objekt vom Typ *PageParameters* als Übergabevariable; diese werden uns später noch begegnen, wenn es darum geht, zustandslose Kommunikation mithilfe von Wicket zu ermöglichen:

```
public class HomePage extends WebPage {
    public HomePage(final PageParameters parameters) {
        add(new Label("message",
            "If you see this message wicket is properly configured and
                                                              running"));
    }
}
```

Im gleichen Package wie *HomePage.java* befindet sich noch die entsprechende HTML-Datei mit dem Namen *HomePage.html*. Im Gegensatz zu JSP, JSF Facelets oder Ruby on Rails verzichtet Wicket konsequent auf den Einsatz von Taglibs, Scriptlets oder Expression Languages. Stattdessen erweitert Wicket HTML-Elemente um das Attribut *id* aus einem Wicket XML Namespace. Dieses Attribut referenziert jeweils eine Komponente, die in der Java-Klasse über die *Add*-Methode hinzugefügt wurde. Im Fall unseres Beispiels wäre das ein Label mit der Wicket-ID *message*:

```
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:wicket=
          "http://wicket.apache.org/dtds.data/wicket-xhtml1.4-strict.dtd">
    <head>
        <title>Wicket Quickstart Archetype Homepage</title>
    </head>
    <body>
        <strong>Wicket Quickstart Archetype Homepage</strong>
        <br/><br/>
    <span wicket:id="message">message will be here</span>
    </body>
</html>
```

Natürlich reicht der Umfang dieses Artikels nicht aus, um Wicket selbst in all seinen Facetten vorzustellen. Jedoch sollte das grundlegende Prinzip von Wicket, nämlich Plain HTML und Java, ersichtlich sein. Für den weiteren Verlauf sollten Sie insbesondere die Applikation selbst und das Arbeiten mit Page-Parametern im Auge behalten.

RESTful HTTP in Webanwendungen

Hinter REST (REpresentational State Transfer) verbirgt sich die Architektur des WWW, definiert von Miterfinder Roy T. Fielding. Ganz kurz zusammengefasst lässt sie sich auf einige Grundprinzipien reduzieren: Das zentrale Element in einer REST-Architektur sind Ressourcen, die klar identifiziert werden können. In der konkreten Ausprägung HTTP sind für diese Identifikation URIs zuständig. Ressourcen werden niemals direkt angesprochen, sondern nur über Repräsentationen, die den Zustand der Ressource enthalten und zwischen Client und Server transferiert werden. Alle Ressourcen unterstützen die gleiche Schnittstelle, bei HTTP vor allem bestehend aus den Standardmethoden GET, PUT, POST und DELETE. Schließlich heißt das WWW nicht umsonst World Wide Web: Ressourcen sind miteinander vernetzt. Das spiegelt sich über Links in den Repräsentationen wider, die den Kontrollfluss des Clients und damit der Gesamtanwendung steuern können. REST ist ein Architekturstil, eine Art Entwurfsmuster im Großen, der durch eine Reihe von Randbedingungen (Constraints) definiert ist. Das Web mit seinen Standards, vor allem HTTP und URIs, ist eine konkrete Ausprägung dieses Stils: Den Designern der Webprotokolle dienten die Konzepte des REST-Stils als Grundlage. Aktuell taucht REST bzw. RESTful HTTP meist in der Diskussion um den rich-

tigen Ansatz für Web Services auf, als webkonformere Alternative zu SOAP, WSDL & Co.

Es existiert eine ganze Reihe von Java-Bibliotheken, die ein Entwickler für die Erstellung von Web Services verwenden kann. Diese lassen sich unterteilen in solche, die auch die Verwendung von REST-Prinzipien erlauben (wie das Servlet API), solche, die RESTful HTTP explizit unterstützen (wie Restlet oder JAX-RS) und schließlich solche, die es unmöglich machen (wie die meisten SOAP/WSDL/WS-*-Frameworks). Im Gegensatz zu diesen Ansätzen ist das Hauptziel von Webframeworks, den Entwickler bei der Erstellung von Benutzeroberflächen für menschliche Benutzer zu unterstützen. Als Java-Entwickler klassischer Schule sind wir es gewohnt, Serviceframeworks und UI-Frameworks als vollkommen getrennte und unabhängige Dinge zu betrachten. Aus REST-Sicht jedoch gibt es nur ein einziges Web, und der Browser ist nur eine von vielen Anwendungen. Wie also muss ein Webframework aussehen, das den REST-Prinzipien folgt? Eine Webanwendung, die HTTP REST-konform einsetzt, zeichnet sich durch eine Reihe von Eigenschaften aus:

- Jede Seite, die dem Benutzer dargestellt wird, wird mit einem eigenen URI identifiziert, sodass der Benutzer einen Link darauf per E-Mail versenden oder ein Bookmark setzen kann. REST verallgemeinert diesen Ansatz und spricht von Ressourcen statt Seiten – alle Informationen, alle wesentlichen Konzepte bekommen eine eigene, standardisierte und weltweit einsetzbare ID.
- Die HTTP-Verben, insbesondere GET und POST, werden ihrer Definition entsprechend eingesetzt. Bei der Nutzung von HTML bedeutet das, dass GET für lesende, sichere Anfragen und ein POST für ändernde Aktionen verwendet wird. Durch den Einsatz von GET für lesende Zugriffe bekommen damit z. B. auch Suchergebnisse eine eigene URI und können verlinkt werden; eine Aktualisierung der Seite ist ohne eine Fehlermeldung des Browsers möglich, und der Back-Button funktioniert so, wie es der Benutzer zu Recht erwartet. Antworten auf GET Requests können gecacht werden, wobei die Art der Cache-Validierung und die erlaubte Cache-Dauer vom Server gesteuert werden können. Der Einsatz von POST für ändernde Aktionen stellt sicher, dass das einfache Verfolgen eines Links, z. B. durch den Crawler einer Suchmaschine, nicht versehentlich Daten ändert. Die Unterstützung weiterer HTTP-Verben wie PUT und DELETE ist für HTML zwar aktuell nicht relevant (HTML-Formulare unterstützen nur GET und POST), allerdings für die Nutzung in der Anwendungs-zu-Anwendungs-Kommunikation relevant.
- In Zeiten, in denen immer mehr Anwendungen nicht über die Weboberfläche, sondern auch über ein Web-API zur Verfügung stehen sollen, sollten vielen Informationen nicht nur im HTML-Format, sondern auch in XML, JSON oder anderen maschinenlesbaren Formen zur Verfügung gestellt und verarbeitet werden können. Dazu unterstützt HTTP das Prinzip von Content Negotiation – eine Ressource kann in mehr als einem Format vorliegen.
- Die Kommunikation erfolgt grundsätzlich statuslos, mit anderen Worten: Jeder URI der Anwendung kann als Einstiegspunkt dienen, entsprechende Authentifizierung vorausgesetzt. Dadurch wird die Skalierbarkeit des Systems drastisch erhöht, weil aufeinanderfolgende Requests nicht vom selben Prozess bearbeitet werden müssen.
- Die von der Anwendung zur Verfügung gestellten Ressourcen sind miteinander verknüpft, die Navigation erfolgt über Links und Formulare.

Eine Webanwendung, die diese Kriterien erfüllt, ist nicht nur RESTful, sie ist aus Sicht des Anwenders auch schlicht *besser* – wer hat sich nicht schon über nicht funktionierende Back-Buttons, fehlende Bookmark-Möglichkeit, irritierende Meldungen beim Refresh etc. geärgert. Ein Webframework, dass die Entwicklung von REST-konformen Webanwendungen ideal unterstützt, sollte es einfach machen, diesen Prinzipien zu folgen – zumindest einfacher, als dagegen zu verstoßen.

No REST for the Wicket?

Moderne Webanwendungen verwenden sehr häufig sprechende URIs für relevante Konzepte als Einstiegspunkt, so z. B. Twitter: http://www.twitter.com/bitboss weist zum Twitter-Profil von Michael Plöd, http://www.twitter.com/stilkov auf das von Stefan Tilkov; Flickr identifiziert alle vom User *rockriot* mit dem Tag *Kanada* versehenen Fotos mit dem URI http://www.flickr.com/photos/rockriot/tags/kanada.

Mit Wicket lassen sich entsprechende Einstiegs-URIs sehr einfach erzeugen. Dreh- und Angelpunkt dafür ist das Bootstrapping der Anwendung in der *init()*-Methode

der *WebApplication*. Hier können mithilfe der Methode *mount(..)* so genannte URL-Coding-Strategien definiert werden. Von Haus aus verwendet Wicket den voll qualifizierten Namen einer Page-Klasse im URL, was mit der Verwendung von URL-Coding-Strategien überschrieben werden kann. Folgender Code würde den URL http://www.allschools.de/?wicket:bookmarkablePage=:de.allschools.view.UserProfilePage&username=mploed in http://www.allschools.de/profile/mploed transformieren:

```
public class WicketApplication extends WebApplication {
   ...
   @Override
   protected void init() {
      mount(new MixedParamUrlCodingStrategy("profile",
                        UserProfilePage.class, new String[] { "username" }));
   }
}
```

Dabei unterstützt Wicket verschiedene URL-Coding-Strategien, wie Tabelle 1 zeigt.

Einsatz von GET und POST mit unterschiedlicher Semantik

Bei der Verwendung von GET und POST sind zwei Eigenschaften von Webframeworks hinsichtlich der Eignung für Anwendungen, die den REST-Prinzipien folgen, zu untersuchen. Zum einen muss es zwingend möglich sein, dass sowohl mit GET als auch mit POST gearbeitet wird (PUT und DELETE spielen bei reinen Webanwendungen keine Rolle, da sie von HTML nicht unterstützt werden). Des Weiteren muss sichergestellt werden, dass über GET keine schreibenden Zugriffe durchgeführt werden und lesende über GET erfolgen können. In Wicket ist die Einheit, die von außen angesprochen wird, die Seite (Page). Ob dies per POST oder GET geschieht, spielt dabei für Wicket keine Rolle – das Verhalten ist das gleiche. Je nachdem, wie die Page in den Fluss der Anwendung eingebunden wird, wird entweder das eine oder das andere verwendet. Greifen wir eben genanntes Beispiel der *UserProfilePage* auf und verlinken diese mithilfe eines *BookmarkablePageLinks*, resultiert das in einem GET Request:

```
public class ProfileListPage extends WebPage {
   ...
   public ProfileListPage(PageParameters pp) {
      ...
      add(new BookmarkablePageLink("profileLink", UserProfilePage.class,
                              new PageParameters("username=mploed"));
      ...
   }
}
```

Diese Komponentenverwendung führt in Kombination mit der oben erwähnten URL-Coding-Stragegie zu einem GET Request auf den URL http://www.meineseite.de/profile/mploed.

Natürlich besitzt Wicket keine Möglichkeit, die Verwendung von GET Requests für schreibende Zugriffe zu verhindern. An dieser Stelle ist die Disziplin der Entwickler gefragt: Es ist Ihre Verantwortung, ändernde Zugriffe mit Ajax-Links oder Formularen zu realisieren. Möchten Sie hingegen einen POST Request erzwingen, so müssen Sie Formularkomponenten verwenden:

```
public class ProfileListPage extends WebPage {
   ...
   public ProfileListPage(PageParameters pp) {
      ...
      Form f = new Form("form") {
         @Override
         protected void onSubmit() {
               //hier wird mit Post gearbeitet
            ....
         }
      };
      ...
   }
}
```

Formularkomponenten werden beim Submit von Wicket mit dem Post Redirect Get Pattern verarbeitet. Diese Einstellung verhindert Probleme rund um doppelte Klicks und die Back-Button-Verwendung. Hierbei wird der Request aufgeteilt. Die Formulardaten werden durch den POST Request verschickt, der den Serverstatus ändert, anschließend wird der Anwendung ein Redirect auf die aktuelle Seite gesendet, die diese dann wieder über einen GET Request abfragt. Somit wird bei einem Refresh der Seite ausschließlich der GET Request wiederholt, der keine Zustandsänderung zur Folge hat. Die Konfiguration kann, falls gewünscht, wie bei Wicket üblich in der *init()*-Methode der Application verändert werden:

URL	Strategie
/profile?username=mploed	QueryStringUrlEncodingStrategy
/profile/username/mploed	BookmarkablePageRequestTargetUrlCodingStrategy
/profile/mploed	IndexedParamUrlCodingStrategy
/profile/mploed/?tags=kanada	MixedParamUrlCodingStrategy
/profile/name/mploed	HybridUrlCodingStrategy
/profile/mploed	IndexedHybridUrlCodingStrategy

Tabelle 1: URL-Coding-Strategien

```
public class WicketApplication extends WebApplication {
   ...
   @Override
   protected void init() {
      ...
      getRequestCycleSettings().setRenderStrategy(IRequestCycleSettings.
                                                   REDIRECT_TO_BUFFER);
   }
}
```

Wicket ermöglicht folgende Konfigurationseinstellungen:

ONE_PASS_RENDER: Hier erfolgt die komplette Verarbeitung in einem Request, was zur Folge hat, dass kein Schutz vor Doppelklicks oder der Back-Button-Verwendung vorliegt. Die Einstellung macht jedoch in Clustern Sinn, die nicht gewährleisten können, dass zwei Requests auf dem gleichen Server landen (Stichwort: Sticky-Session).

REDIRECT_TO_BUFFER (default): Auch hier erfolgt die Verarbeitung in einem Request, jedoch wird das Ergebnis vor dem Abschicken am Server gecacht. Der GET Request holt das Ergebnis dann aus dem Cache.

REDIRECT_TO_RENDER: Hier erfolgt die Verarbeitung in zwei Requests. Der erste Request verarbeitet die Anfrage innerhalb des Komponentenbaums. Nach Abschicken des Redirects mit dem folgenden GET Request erfolgt das Rendering in letzterem Schritt. Diese Herangehensweise ist die langsamste, weil hier insbesondere bei der Verwendung von *LoadableDetachableModels* zwei Backend-Aufrufe stattfinden.

Möchte man als Entwickler nun unterscheiden, ob ein Request mit POST oder GET verarbeitet wird, um unterschiedliche Logik anzustoßen, so gilt es, zwei unterschiedliche Punkte zu betrachten: Alles was innerhalb einer *onSubmit()*-Methode eines Formulars oder eines Buttons, der einem Formular anhängt, geschieht, wird grundsätzlich mit POST verarbeitet. Bei Pages ist das Verhalten in der Regel ebenfalls eindeutig. Steuern Sie eine Page aus einem Link an, so wird auf diese immer mit GET zugegriffen. Arbeiten Sie in Formularen mit *setResponsePage(...)* wird auf die Page ebenfalls mit GET zugegriffen. Möchte Sie dennoch sowohl in einer *onSubmit()*-Methode als auch in einer Page feststellen, welches HTTP-Verb der Request verwendet, so ist hierfür ein Low-Level-Zugriff auf die Servlet API nötig. Das entspricht zwar nicht der regulären Arbeitsweise von Wicket, ist jedoch dennoch möglich: *((WebRequest)getRequestCycle().getRequest()).getHttpServletRequest().getMethod()*.

Die Verwendung von GET für das Abarbeiten eines Submits aus HTML-Formularen ist mit Wicket nicht möglich.

Unterstützung für Caching

Über den *RequestCycle* und die *WebResponse* lassen sich zudem sämtliche HTTP-Header-Informationen manipulieren. An dieser Stelle ist kein Zugriff auf die native Servlet API mehr erforderlich. Insbesondere das Thema Caching Header ist für RESTful-HTTP-Anwendungen von großer Bedeutung. Dazu kann das Caching über folgende HTTP-Header aktiviert bzw. konfiguriert werden:

```
Cache-Control: [public | private] , max-age=[timespan]
Last-Modified: [timestamp]
Expires: [timestamp]
```

Mithilfe von Wicket lassen sich diese Header über einen sauberen (Wicket-)API-Zugriff wie folgt lösen:

```
((WebResponse)getRequestCycle().getResponse()).setHeader("Cache-Control",
                                                  "public, max-age=...");
((WebResponse)getRequestCycle().getResponse()).setLastModifiedTime(...);
((WebResponse)getRequestCycle().getResponse()).setHeader("Expires", ...);
```

Wicket 1.5 wird ein explizites Caching API unterstützen, indem auf Ebene der *WebResponse*-Klasse die Methoden *disableCaching()* und *enableCaching(Duration duration, WebResponse.CacheScope scope)* eingeführt werden.

Zustandslose Kommunikation

Wicket arbeitet wie jedes komponentenbasierte Framework mit zustandsbasierter Kommunikation, die auf zwei Ebenen aufsetzt. Die erste Ebene ist der Komponentenbaum. In diesem legt Wicket die Werte der Models der jeweiligen Komponeten ab und serialisiert diese in der Page Map, die die Unterstützung für die Back-Button-Verwendung, Multi-Windows und Tabs realisiert. Die zweite Ebene ist die Session. Die Verwendung der Session lässt sich sehr einfach umgehen – man verwendet sie einfach nicht (analog zur Verwendung der Session in einer Servlet-Anwendung). Anstelle dessen erfolgen Parameterübergaben, wie bereits gesehen, mithilfe des *PageParameter*-Objekts. Beim Komponentenbaum ist die Umsetzung zustandsloser Kommunikation nicht ganz so trivial. Die Lösung, oder sagen wir besser, die Linderung der Problematik, ist der Einsatz von *LoadableDetachableModels*. Das Model-Konzept von Wicket ist der Dreh- und Angelpunkt des Bindings von Werten an Komponenten. Wicket führt hier eine Abstraktion zwischen Domänenobjekt und Komponente ein. Anstelle einer direkten Verbindung geht Wicket den Umweg über ein Model. Dessen Wert wird per se in den Komponentenbaum serialisiert. Der Vorgang lässt sich einfach erkennen, wenn man nicht serialisierbare Objekte an ein Model bindet. Im Logfile würde dann eine *NonSerializableException* erscheinen. Ein *LoadableDetachableModel* hingegen wird nicht an den Komponentenbaum gebunden, sondern bei jedem Aufruf neu erstellt. Üblicherweise wird das *LoadableDetachableModel* zur Speicherreduktion verwendet, es führt aber auch zu zustandsloser Kommunikation.

Fazit

Die Autoren vertreten unterschiedliche Meinungen, sodass die übliche Zusammenfassung am Ende eines Artikels nicht ganz so ausfällt wie üblich, sondern ein wenig zwiespältig. Apache Wicket ist ohne Zweifel ein mächtiges Framework, das sich vor allem durch ein elegantes Programmiermodell auszeichnet. Wie alle komponentenorientierten Frameworks stehen einige seiner fundamentalen Designprinzipien im Konflikt mit denen des Webs. Dazu zählt insbesondere die zustandsbehaftete Architektur und die standardmäßige Verwendung von POST für Formulare. Einige der Punkte, z. B. explizite URIs für Einsprungpunkte, können recht leicht adressiert werden, andere, wie die unterschiedliche Behandlung von GET und POST, erfordert einen Zugriff auf sehr tiefer Ebene. Ob man das als Problem empfindet oder nicht, hängt von der Gewichtung ab: Steht das Programmiermodell im Vordergrund und betrachtet man vor allem Services als den Einsatzbereich von RESTful HTTP, ist der Grad von REST-Unterstützung, den man mit Wicket erreichen kann, gut genug. Sieht man zwischen Webanwendungen und Web Services im wörtlichen Sinne keine klare Grenze, ist man mit einem Request-/Response-orientierten Framework besser bedient. Das Experiment, das die beiden Autoren mit diesem Artikel durchführen wollten, hat also das erwartete Ergebnis gebracht: Der eine sieht sich darin bestätigt, dass Wicket REST unterstützt, zumindest im relevanten Bereich, der andere wird Wicket auch weiterhin als nicht REST-konform bezeichnen.

Michael Plöd ist Chief Developer bei der Senacor Technologies AG und in dieser Rolle überwiegend in Projekten tätig, die neben der Realisierung der Fachlichkeit auch die Optimierung der Architektur zum Ziel haben. Sein besonderes Interesse gilt Java-Persistenztechnologien. Zu diesem Thema hat er bereits einige Artikel im Java Magazin veröffentlicht.

Stefan Tilkov ist Geschäftsführer und Principal Consultant bei der innoQ Deutschland GmbH, wo er sich vorwiegend mit der strategischen Beratung von Kunden im Umfeld von Softwarearchitekturen beschäftigt. Er ist Autor des Buchs „REST und HTTP“, Mitherausgeber „SOA-Expertenwissen“ (beide dpunkt Verlag), Autor zahlreicher Fachartikel und häufiger Sprecher auf internationalen Konferenzen.

Links & Literatur

[1] Roland Förther, Carl-Eric Menzel, Olaf Siefart: „Wicket“, dpunkt-Verlag: http://www.dpunkt.de/buch/2921.html

[2] Stefan Tilkov: „REST und http“, dpunkt-Verlag: http://rest-http.info/

[3] Stefan Tilkov: „REST – Das bessere Web-Service-Modell?“, in Java Magazin 1.2009