KaffeeKlatsch

Das Magazin rund um Software-Entwicklung

ISSN 1865-682X

02/2012

Jahrgang 5

KaffeeKlatsch

Das Magazin rund um Software-Entwicklung

Sie können die elektronische Form des KaffeeKlatschs
monatlich, kostenlos und unverbindlich
durch eine E-Mail an

abo@bookware.de

abonnieren.

Ihre E-Mail-Adresse wird ausschließlich für den Versand
des KaffeeKlatschs verwendet.

Editorial

Fünfzig

Nachdem ja in der letzten Ausgabe an dieser Stelle darüber sinniert wurde, ob und wann man Jubiläen feiert, gibt es dieses Mal einen konkreten Anlass: Die 50ste Ausgabe.

Sie müssen sich keine Sorgen machen, dass wir jetzt dauernd irgendeine Ausgabe feiern wollen, denn soweit ich mich erinnere, haben wir das seit der Ausgabe 1000_2 auch nicht mehr gemacht. Die hätten wir übrigens auch als 10_8 feiern können, aber das wäre ja lange nicht so beeindruckend gewesen.

Allerdings hätten wir die Gelegenheit nutzen und auch die Ausgabe 10000_2 feiern sollen, die sich schon – deutlich beeindruckender – als 10_{16} dargeboten hätte; die oktale 20_8 wäre zu banal gewesen. Selbst die Ausgabe $100\,000_2$ haben wir schon hinter uns und ist demnach auch unbefeiert an uns vorbeigezogen.

Das Problem mit den Binärzahlen ist ja leider, dass man immer doppelt so lang warten muss, bis man wieder feiern darf. Die Ausgabe $1\,000\,000_2$ ist zwar gar nicht mehr sooo fern, aber ein knappes Jahr müssten wir schon noch warten. Die Ausgabe 100_{16} liegt noch utopisch fern, da diese erst in etwa 16 Jahren herausgegeben wird.

Deshalb haben wir uns dieses Mal für das Dezimalsystem entschieden. So finden auch noch alle Titelbilder auf der Vorderseite lesbar Platz.

Dem aufmerksamen Leser wird nicht entgangen sein, dass irgendetwas nicht ganz stimmt. Das ist auch korrekt: die August-Ausgabe in 2009 hat es nicht gegeben. Dafür gab es allerdings ein Sommer-Sonderheft. Bei diesem hatten wir aber eines der Bilder recycelt. Also dürften es doch eigentlich nur 49 Titelbilder sein.

Und damit es wieder 50 sind (obwohl wegen der Proportionen 48 vollkommen ausgereicht hätten), haben wir noch das Titelblatt von der zweiten Testausgabe im Dezember 2007 hinzugefügt.

Jetzt wissen Sie, dass doch noch alles mit rechten Dingen zugegangen ist. So verbleibe ich – mit der Hoffnung auf die nächsten fünfzig Ausgaben –

Ihr Michael Wiedeking
Herausgeber

Beitragsinformation

Der KaffeeKlatsch dient Entwicklern, Architekten, Projektleitern und Entscheidern als Kommunikationsplattform. Er soll neben dem Know-how-Transfer von Technologien (insbesondere Java und .NET) auch auf einfache Weise die Publikation von Projekt- und Erfahrungsberichten ermöglichen.

Beiträge

Um einen Beitrag im KaffeeKlatsch veröffentlichen zu können, müssen Sie prüfen, ob Ihr Beitrag den folgenden Mindestanforderungen genügt:

- Ist das Thema von Interesse für Entwickler, Architekten, Projektleiter oder Entscheider, speziell wenn sich diese mit der Java- oder .NET-Technologie beschäftigen?
- Ist der Artikel für diese Zielgruppe bei der Arbeit mit Java oder .NET relevant oder hilfreich?
- Genügt die Arbeit den üblichen professionellen Standards für Artikel in Bezug auf Sprache und Erscheinungsbild?

Wenn Sie uns einen solchen Artikel, um ihn in diesem Medium zu veröffentlichen, zukommen lassen, dann übertragen Sie Bookware unwiderruflich das nicht exklusive, weltweit geltende Recht

- diesen Artikel bei Annahme durch die Redaktion im KaffeeKlatsch zu veröffentlichen
- diesen Artikel nach Belieben in elektronischer oder gedruckter Form zu verbreiten
- diesen Artikel in der Bookware-Bibliothek zu veröffentlichen
- den Nutzern zu erlauben diesen Artikel für nicht-kommerzielle Zwecke, insbesondere für Weiterbildung und Forschung, zu kopieren und zu verteilen.

Wir möchten deshalb keine Artikel veröffentlichen, die bereits in anderen Print- oder Online-Medien veröffentlicht worden sind.

Selbstverständlich bleibt das Copyright auch bei Ihnen und Bookware wird jede Anfrage für eine kommerzielle Nutzung direkt an Sie weiterleiten.

Die Beiträge sollten in elektronischer Form via E-Mail an redaktion@bookware.de geschickt werden.

Auf Wunsch stellen wir dem Autor seinen Artikel als unveränderlichen PDF-Nachdruck in der kanonischen KaffeeKlatsch-Form zur Verfügung, für den er ein unwiderrufliches, nicht-exklusives Nutzungsrecht erhält.

Leserbriefe

Leserbriefe werden nur dann akzeptiert, wenn sie mit vollständigem Namen, Anschrift und E-Mail-Adresse versehen sind. Die Redaktion behält sich vor, Leserbriefe – auch gekürzt – zu veröffentlichen, wenn dem nicht explizit widersprochen wurde.

Sobald ein Leserbrief (oder auch Artikel) als direkte Kritik zu einem bereits veröffentlichten Beitrag aufgefasst werden kann, behält sich die Redaktion vor, die Veröffentlichung jener Beiträge zu verzögern, so dass der Kritisierte die Möglichkeit hat, auf die Kritik in der selben Ausgabe zu reagieren.

Leserbriefe schicken Sie bitte an leserbrief@bookware.de. Für Fragen und Wünsche zu Nachdrucken, Kopien von Berichten oder Referenzen wenden Sie sich bitte direkt an die Autoren.

Werbung ist Information

Firmen haben die Möglichkeit Werbung im KaffeeKlatsch unterzubringen. Der Werbeteil ist in drei Teile gegliedert:

- Stellenanzeigen
- Seminaranzeigen
- Produktinformation und -werbung

Die Werbeflächen werden als Vielfaches von Sechsteln und Vierteln einer DIN-A4-Seite zur Verfügung gestellt.

Der Werbeplatz kann bei Frau Natalia Wilhelm via E-Mail an anzeigen@bookware.de oder telefonisch unter 09131/8903-16 gebucht werden.

Abonnement

Der KaffeeKlatsch erscheint zur Zeit monatlich. Die jeweils aktuelle Version wird nur via E-Mail als PDF-Dokument versandt. Sie können den KaffeeKlatsch via E-Mail an abo@bookware.de oder über das Internet unter www.bookware.de/abo bestellen. Selbstverständlich können Sie das Abo jederzeit und ohne Angabe von Gründen sowohl via E-Mail als auch übers Internet kündigen.

Ältere Versionen können einfach über das Internet als Download unter www.bookware.de/archiv bezogen werden.

Auf Wunsch schicken wir Ihnen auch ein gedrucktes Exemplar. Da es sich dabei um einzelne Exemplare handelt, erkundigen Sie sich bitte wegen der Preise und Versandkosten bei Alexandra Specht via E-Mail unter alexandra.specht@bookware.de oder telefonisch unter 09131/8903-14.

Copyright

Das Copyright des KaffeeKlatschs liegt vollständig bei der Bookware. Wir gestatten die Übernahme des KaffeeKlatschs in Datenbestände, wenn sie ausschließlich privaten Zwecken dienen. Das auszugsweise Kopieren und Archivieren zu gewerblichen Zwecken ohne unsere schriftliche Genehmigung ist nicht gestattet.

Sie dürfen jedoch die unveränderte PDF-Datei gelegentlich und unentgeltlich zu Bildungs- und Forschungszwecken an Interessenten verschicken. Sollten diese allerdings ein dauerhaftes Interesse am KaffeeKlatsch haben, so möchten wir diese herzlich dazu einladen, das Magazin direkt von uns zu beziehen. Ein regelmäßiger Versand soll nur über uns erfolgen.

Bei entsprechenden Fragen wenden Sie sich bitte per E-Mail an copyright@bookware.de.

Impressum

KaffeeKlatsch Jahrgang 5, Nummer 2, Februar 2012
ISSN 1865-682X
Bookware – eine Initiative der MATHEMA Software GmbH

Henkestraße 91, 91052 Erlangen
Telefon: 0 91 31 / 89 03-0
Telefax: 0 91 31 / 89 03-55
E-Mail: redaktion@bookware.de
Internet: www.bookware.de

Herausgeber/Redakteur: Michael Wiedeking
Anzeigen: Natalia Wilhelm
Grafik: Nicole Delong-Buchanan

Inhalt

Editorial 3

Beitragsinfo 4

Inhalt 5

User Groups 24

Werbung 26

Das Allerletzte 27

Artikel

JSF++;

Erfahrungen bei Upgrades 7

Password to go

Der persönliche Passwortgenerator für unterwegs, Teil 1 11

Kolumnen

Auf Safari

Fehlerteufelchen 16

Überläufer

Des Programmierers kleine Vergnügen 18

Gender Gap II und ein bisschen Grammatik

Deutsch für Informatiker 21

Von wegen „einfach“

Kaffeesatz 23

JSF++;

Erfahrungen bei Upgrades 7

von William Siakam

In der Theorie oder besser gesagt im Traumfall ähnelt das Szenario eines JSF-Umstiegs auf eine höhere Version Folgendem: Die Anwendung herunterfahren; die alten JSF-Bibliotheken durch Neue ersetzen; die Anwendung neu bauen; starten – und alles läuft wie es soll. Schön wär's.

Password to go

Der persönliche Passwortgenerator für unterwegs, Teil 1 11

von Frank Goraus

Sicherheit ist heutzutage ja ein wichtiges Thema. Einen Teil davon machen persönliche Passwörter aus – Schlüsselworte, die den Zugriff auf geheime Daten nach Möglichkeit nur deren Besitzer erlauben sollen. Doch hier hapert es meist schon an zu einfach gewählten Passwörtern, die Unbefugte nicht nachhaltig genug ausschließen. Deshalb soll dieser zweiteilige Artikel aufzeigen, wie man sich schon mit einfachen programmatischen Mitteln sicherere Zugangsschlüssel erzeugen kann. In diesem ersten Teil geht es dabei um die Grundidee eines Verschlüsselungssystems für Passwörter.

Auf Safari

Fehlerteufelchen 16

von Sascha Gross

Benutzer bewusst bei Web-Anwendungen auf eine Menge von Browsern zu beschränken ist nicht schön, aber durchaus gängige Praxis. Wenn aber Safari-Surfer unbewusst ausgeschlossen werden, ist das schon ärgerlich.

Herbstcampus

Wissenstransfer par excellence

3. – 6. September 2012, Nürnberg

Beitragsaufruf

(Call for Papers)

Der Herbstcampus ist eine technologie-orientierte Konferenz, die Software-Entwicklern, Architekten und Projektleitern eine konkrete Hilfe bieten soll. Schwerpunkte sind .NET, Scala und Java.

Interessante Themen sollen in verschiedenen Darbietungen behandelt werden: als Einführung, Erfahrungsbericht oder problemlösender Vortrag.

Haben Sie ein passendes Thema oder Interesse daran, einen Vortrag zu halten? Schicken Sie uns Ihre Vorschläge:
beitragsaufruf@herbstcampus.de

Haben Sie noch Fragen? Dann schreiben Sie uns eine E-Mail.
info@herbstcampus.de

Den vollständigen Beitragsaufruf und weitere Informationen zum Herbstcampus finden Sie unter
www.herbstcampus.de

JSF++;

Erfahrungen bei Upgrades

von William Siakam

In der Theorie oder besser gesagt im Traumfall ähnelt das Szenario eines JSF-Umstiegs auf eine höhere Version Folgendem: Die Anwendung herunterfahren; die alten JSF-Bibliotheken durch Neue ersetzen; die Anwendung neu bauen; starten – und alles läuft wie es soll. Schön wär's.

Wie bei unzähligen anderen Sachen im Leben hilft auch hier der Glaube an die berühmte *Das Glas ist halb voll*-Theorie nicht, um diese blauäugige und naive Vorstellung Wirklichkeit werden zu lassen. Letztendlich konvergieren alle Träume immer auf einen Punkt: Das Aufwachen. Und spätestens wenn sich das vermeintliche *Plug-and-Play* als ein *Plug-and-Cry* erwiesen hat, wacht man auf und fragt sich, wie man glauben konnte, dass so etwas real war.

In mehreren Fällen teilt sich JSF binnen einer Web-Anwendung das Boot mit anderen Frameworks, die im schlechtesten Fall, alle zusammen in „strenger" Harmonie leben – d. h. in einer rückwärtig inkompatiblen Versionsabhängigkeit. Ein „Traum" für jedes Projekt.

Als JSF1.1 im Jahr 2004 eingeführt wurde, war das Angebot, was Java-basierte Web-Frameworks angeht, nicht besonders vielfältig. Von der Boutique mit JSF-Frameworks, die heutzutage alles anbietet, was die Entwicklerfinger begehren (z. B. PrimeFaces oder RichFaces & Co.), gab es zu jenem Zeitpunkt keine Spur. Wie denn auch? Denn JSF war eben geboren und musste sich erstmal bewähren.

Schnell stellten die traditionellen Web-Entwickler, die anfingen JSF-Seiten zu schreiben, einen großen Mangel fest: Die JSF-Komponentenbibliothek konnte nicht alles anbieten (und tut es bis dato immer noch nicht), was im HTML-Standard vertreten war.

Das ist auch der Grund, warum das Tag <*f:verbatim*> in der *JSF-Tag-Library* existiert. Es ermöglichte es, einen in JSF nicht vorgesehenen HTML-Standard-Tag trotzdem verwenden zu können. Hierzu ein Beispiel:

```
<h:form>
    <f:verbatim>
      <fieldset>
      </fieldset>
    </f:verbatim>
</h:form>
```

Dies bewirkt, dass das innerhalb von <*f:verbatim*> eingebettete HTML-Tag in den JSF-Komponentenbaum aufgenommen, d. h. Teil der <*f:view*> wurde und demzufolge als JSF-Komponente behandelt werden konnte. Somit war das Mischen von HTML- und JSF-Tags möglich – losgelöst davon, dass diese Kombination erstens nicht besonders gut aussah und zweitens eher einer schlechten Zweck-WG glich. So wäre es besser gewesen, eine Verständigungsbasis zu schaffen, die für eine wirkliche Einheit sorgt, äußerlich wie innerlich. Genau auf der Jagd nach dieser Ideologie hat Sergey Smirnov die JSF-Bibliothek *htmLIB* [1] geschrieben und veröffentlicht.

Diese *HTML Layout Tag Library* verspricht den JSF-Entwicklern Freiheit in der Auswahl der HTML-Tags plus zusätzliche nette Features. Damit wird die Anwendung von <*f:verbatim*> praktisch überflüssig gemacht. So gut wie jedes Standard-HTML-Tag (<*x*>) hat damit nämlich ein äquivalentes Tag in JSF (<*htmlib:x*>).

Das JSF-Tag-Attribut *rendered* kann überall in *htmLIB* angewendet werden. Außerdem sorgt das At-

tribut *sid* (*strict id*) dafür, dass die von JSF verschachtelte ID-Generierung von Elementen umgangen werden kann. Also ein mit *sid="foo"* versehenes *htmLIB*-Element führt zu der *id="foo"* auf dem Browser, unabhängig davon wie viele Forms als Parents oder Vorfahren existierten.

Alles würde also gar nicht so schlecht klingen, wenn das Framework nicht gegen JSF1.1 gebaut und mit dieser Version streng gekoppelt wäre.

Sie können sich also denken was passiert, wenn eine Anwendung mit JSF1.1 geschrieben wurde, die die *htmLIB* mit an Bord hat, und ein JSF-Upgrade durchgeführt werden soll. Keine spaßige Angelegenheit. Trotzdem muss man dazu eine Lösung finden, und dabei gibt es zwei Wege:

1. Alle *jsp*-Dateien bleiben unverändert. Dafür muss das Framework – falls die Quellen offen liegen – an die aktuelle JSF-Version angepasst werden. Der Vorteil ist, dass nicht jede Seite der Anwendung refaktoriert werden muss, sprich alle URIs herausgeworfen und alle verwendeten Tags durch Standard JSF-Tags ersetzt werden müssen. Der Nachteil ist, dass man diese Arbeit bei jedem Upgrade auf sich nehmen muss, falls man dabei die Spezifikation nicht konsequent beim Wort nimmt.
2. Man verabschiedet sich ein für alle Mal von dem alten Framework und ersetzt die verwendeten Tags ausnahmslos durch standardisierte JSF-Tags. Hierbei gibt es allerdings zwei Herausforderungen:

   a) Auf den Luxus, den die *htmLIB* anbietet, muss leider verzichtet werden. Und da nicht alle HTML-Tags in JSF vertreten sind, muss man mit einer kompletten Schönheits-OP der Anwendung rechnen. Jede Seite, die das Framework verwendet, muss deshalb unter die Lupe genommen werden.

   b) Nach der Anpassung an den JSF-Standard werden viele JavaScript- und CSS-Elemente, die u. a. auf das *htmLIB*-Attribut *SID* angewendet wurden, fehlschlagen.

Das folgende Beispiel demonstriert diese Problematik.

```
<!-- seite.jsp -->
<!-- vor dem Upgrade -->
<h:form id="myForm">
   <htmlib:div sid="foo" />
</h:form>
```

Der Zugriff über JavaScript bzw. CSS wurde bis zum Upgrade folgendermaßen gelöst:

```
/* script.js*/
document.getElementById("foo");

/*style.css */
#foo {
   background-color: red;
}
```

Weil ein <*h:panelGroup*> mit dem Attribut <*layout="block"*> ein <*div*> rendert, könnte man nach der Umstellung auf Standard JSF-Tags die *jsp*-Seite folgendermaßen umschreiben:

```
<!-- seite.jsp -->
<!-- nach dem Upgrade -->
<h:form id="myForm">
   <h:panelGroup layout="block" id="foo">
</h:form>
```

Der gerenderte HTML-Code sieht demnach so aus:

```
<!-- gerendert -->
<form id="myform">
   <div id="myForm:foo "/>
</form>
```

Logischerweise funktionieren Style- und JavaScript-Routinen auf solchen umgeschriebenen Elementen nicht mehr. Es gibt aber zwei Workarounds um diesem Fluch zu entkommen.

Seit JSF1.2 kann man „rohe" HTML-Tags direkt in der Seite verwenden. Demzufolge ist das Tag <*f:verbatim*> obsolet geworden. Folgendes würde also funktionieren:

```
<h:form id="myForm">
   <div  id="foo"/>
</h:form>
```

Spannend wäre das nicht, wenn sich Nichts dahinter verstecken würde. JSF-Komponenten, die für das Rendern der eigenen *Kinder*-Elemente verantwortlich sind, ignorieren alle nicht-JSF-Elemente. Im folgenden Beispiel wäre eine Tabelle mit zwei Spalten zu rendern. Da *htmLIB* eine JSF-Komponentenbibliothek war, erkannte <*h:panelGrid*> sie als solche.

```
<!-- vor dem Upgrade -->
<h:form id="form">
   <h:panelGrid columns="2">
      <htmlib:div sid="foo">
         <h:outputText />
      </htmlib:div>
      <htmlib:div sid="bar">
         <h:outputText />
      </htmlib:div>
   </h:panelGrid>
</h:form>
```

Ein fehlerhafter Lösungsversuch wäre es, die alten *htmLIB*-Elemente sämtlich durch <*div*> zu ersetzen, die leider nicht als *JSF-Columns*-Elemente behandelt werden können.

```
<!-- fehlerhafter Versuch -->
<h:form id="form">
  <h:panelGrid columns="2">
    <div id="foo">
      <h:outputText />
    </div>
    <div id="bar">
      <h:outputText />
    </div>
  </h:panelGrid>
</h:form>
```

Eine etwas verwirrende aber funktionierende Lösung wäre, diese <*div*>-Tags in JSF-Tags (z. B. <*panelGroup*>) zu verpacken.

```
<!-- Lösung -->
<h:form id="form">
  <h:panelGrid columns="2">
    <h:panelGroup layout="block">
      <div id="foo">
        <h:outputText />
      </div>
    </h:panelGroup>
    <h:panelGroup layout="block">
      <div id="bar">
        <h:outputText />
      </div>
    </h:panelGroup>
  </h:panelGrid>
</h:form>
```

Die zweite Möglichkeit des Workarounds wäre, die JavaScript- und CSS-Elemente jeweils auf die neuen ID's anzupassen.

```
/*script.js*/
document.getElementById("myForm:foo");

/*style.css*/
#form:foo { /* fehlerhalft! */
  background-color:red;
}
```

Der CSS-Präprozessor macht an der Stelle keinen Bogen um seine Bestimmung, was die Doppelpunkte angeht und geht erwartungsgemäß treu seiner Aufgabe nach: Klar, für ihn ist der Doppelpunkt das Trennzeichen zwischen Attributname und Wert. Deswegen muss hier ein Escape-Zeichen verwendet werden. Auch die hexadezimale Variante funktioniert, auch wenn sie nicht so attraktiv aussehen mag.

```
 <!-- style.css -->
#form\3A foo {
  background-color:red;
}
```

Wie man sehen kann, sind auf der Seite der Anwendung einige Änderungsarbeiten vorprogrammiert. Der Aufwand solcher Anpassungen hängt logischerweise von der Größe der Anwendung ab. Aber die Überraschungen, die viele Application-Server auf Lager haben, sind wirklich nicht klein zu reden.

Der Part des Servers

Viele Application-Server bringen eine Palette an Bibliotheken mit, die angeblich nicht nur an deren Server-Landschaften angepasst sind, sondern auch der Spezifikation entsprechen. Das wäre zu schön, um war zu sein.

Wenn man bei einer Application-Server-Installation auf eine mitgelieferte Bibliothek stolpert, die JSF2.0 heißt, dann erwartet man selbstverständlich auch, dass diese dem JSF2.0-Standard entspricht. Aber so logisch darf man eben manchmal nicht denken. Wieso sonst würde z. B. ein seit JSF2.0 eingeführtes neues Tag <*f:ajax*> bei dieser mitgelieferten JSF2.0-Bibliothek fehlen oder noch besser, die dazugehörigen Facelets [2] einfach nicht enthalten?

Auf die Idee, in den Java-Archiven (JARs) herumzuwühlen, kommt man eben nicht immer spontan. Man tut also gut daran, nicht sofort all diesem mitgelieferten „Kram" Vertrauen zu schenken und darf ruhigen Gewissens skeptisch sein. Falls also die JARs trotzdem verwendet werden sollen oder müssen, entpackt man diese und schaut nach, was tatsächlich drin ist. Aber am besten verwendet man immer die gewünschte Bibliothek von der JCP-Seite [3].

Der Part der Spezifikation

Der Umstieg von JSF1.1 auf höhere Versionen kann an einigen Stellen Probleme bereiten, auch wenn man nur JSF-Standard-Tags verwendet hat. Man wird z. B. staunen, warum der Zugriff auf eine Seite mit

```
<h:commandButton action="#{bean.action}"
  style="display:none"/>
```

auf einmal mit der Meldung scheitert, dass man das Attribut *value* oder *graphicImage* einfügen muss. Warum eigentlich, denn die Komponente ist doch absichtlich auf *hidden* gesetzt worden, um darauf ausschließlich per JavaScript zuzugreifen!

Eine andere spannende Geschichte ist, dass der Komponentenbaum in JSF1.1 gleichzeitig erzeugt und

gerendert wurde. Dank dieser Tatsache konnten Tags wie <*f:subview*>, <*f:verbatim*> oder <*jsp:param*> problemlos funktionieren. Ein besonders nettes Feature war, dass die gesetzte Variable bei <*jsp:param*> unmittelbar zur Verfügung stand, als die Seite gerendert wurde.

```
<!--seite1.jsp-->
<h:form>
  <jsp:include page="seite2.jsp">
    <jsp:param name="foo" value="42" />
  </jsp:include>
</h:form>
```

Auf der eingebetteten Seite konnte man auf die Variable folgendermaßen zugreifen:

```
  <!--seite2.jsp-->
<f:subview>
  <h:panelGroup
    rendered="#{param.foo=='42'}">
  </h:panelGroup>
</f:subview>
```

Das Problem mit dem Request-Parameter

Die JSF-Versionen ab 1.2 wollten von <*f:verbatim*> und <*f:subview*> wegkommen, um das Vermischen von Inhalten einfacher zu machen. Aus diesem Grund erfolgte seit JSF1.2 das Bauen des Baums und das Rendern seines Inhalts in zwei unterschiedlichen Etappen. Als Folge darauf funktioniert <*jsp:param*> seitdem aus folgendem Grund nicht mehr: wenn der Inhalt der inkludierten Seite gerendert wird, ist der zu setzende Parameter nicht mehr verfügbar.

Es gibt erfreulicherweise eine Möglichkeit dieses Problem zu lösen, jedoch muss die inkludierte Seite angepasst werden. Man verwendet das Tag <*c:set*> aus JSTL, um den Parameterwert aus dem URL-Request-Parameter *param* in die *RequestScope*-Map zu setzen.

```
<!--seite1.jsp-->
<h:form>
      <jsp:include page="seite2.jsp">
        <jsp:param name="foo" value="42" />
          </jsp:include>
        </h:form>


<!--seite2.jsp-->
<f:subview>
  <c:set var="bar" value="${param.foo}" scope="request" />
  <h:panelGroup
    rendered="#{requestScope.bar=='42'}">
  </h:panelGroup>
</f:subview>
```

So viel zur Rückwärtskompatibilität.

Abschließend soll noch diese kleine Checkliste erwähnt sein, die bei einem Upgrade sehr hilfreich sein kann:

- Achten Sie auf die Version des *Servlet*-Containers und dessen Kompatibilität mit der einzuführenden JSF-Version.
- Verwenden Sie am besten nur die JAR-Dateien von der JCP-Seite. Sollten aus politischen Gründen nur die im Server enthaltenen Bibliotheken erlaubt sein, entpacken Sie diese und schauen Sie nach, ob sie auch wirklich die Features enthalten, weswegen Sie die Version wechseln wollen – ganz nach dem Motto: Vertrauen ist gut, Kontrolle ist besser.
- Nutzen Sie die Gelegenheit, um etwas Refactoring zu machen. Beispielsweise könnten Sie eventuell auf JSTL-Code verzichten.
- Falls Sie von JSP auf *Facelets* umstellen wollen, rechnen Sie mit einem entsprechend großen Aufwand. Die dabei zu leistende Arbeit ist deutlich größer, als man von einer „banalen" Umstellung erwarten würde.

Zusammenfassung

Genau wie bei einem privaten Umzug muss sich die Migration auf eine neuere Version wirklich lohnen. Das ist beispielsweise dann der Fall, wenn neue Anforderungen an die Anwendung nur durch die neuen Features der nächsten Version erfüllt werden können, durch die neue Version gravierende Fehler behoben werden oder eine ausführliche Projektanalyse eine Optimierung der Anwendung in jeder Hinsicht verspricht.

Aber eins muss bei der Migration klar sein: So etwas wie *Plug-and-Play* gibt es dabei in der Regel nicht. Aber egal wie sie sich Ihr Upgrade vorstellen, wichtig ist es, den Bezug zur Realität nicht zu verlieren, sich der Herausforderungen bewusst zu sein und sich den zu erwartenden Problemen zu stellen.

Oder anders gesagt: Rechnen Sie nicht mit einem *Plug-and-Play*, sondern mit einem *Plug-and-Pay*, denn sonst erleben Sie möglicherweise ein *Plug-and-Cry*.

Referenzen

[1] Smirnov, Sergey *HTML Layout Tag Library,* http://jsftutorials.net/htmLib

[2] Wikipedia *Facelets*, http://de.wikipedia.org/wiki/Facelets

[3] Java Community Process Programm *Welcome to the Java Community Process!* http://jcp.org

Kurzbiografie

William Siakam arbeitet als Software-Entwickler und -Consultant für die MATHEMA Software GmbH.Er interessiert sich besonders für die Java Standard und Enterprise Edition (Java SE und Java EE). Dabei liegt sein Schwerpunkt auf dem Design und der Anwendung von Web-Frameworks.

Password to go

Der persönliche Passwortgenerator für unterwegs, Teil 1

von Frank Goraus

Sicherheit ist heutzutage ja ein wichtiges Thema. Einen Teil davon machen persönliche Passwörter aus – Schlüsselworte, die den Zugriff auf geheime Daten nach Möglichkeit nur deren Besitzer erlauben sollen. Doch hier hapert es meist schon an zu einfach gewählten Passwörtern, die Unbefugte nicht nachhaltig genug ausschließen. Deshalb soll dieser zweiteilige Artikel aufzeigen, wie man sich schon mit einfachen programmatischen Mitteln sicherere Zugangsschlüssel erzeugen kann. In diesem ersten Teil geht es dabei um die Grundidee eines Verschlüsselungssystems für Passwörter.

Wenn man die aktuellen Nachrichten aus der IT der letzten Monate verfolgt hat, wird man immer wieder über Hacker-Angriffe, Zugangsdatendiebstahl und neue Datenschutzanforderungen gelesen haben. Und bei all diesen Nachrichten fragt man sich natürlich wie es um die Sicherheit der eigenen Daten bestellt ist. Da man diese Frage jedoch nicht allgemein beantworten kann, muss man hier eher eine Summe von Einzelbetrachtungen zusammenfassen, was ziemlich aufwendig werden kann. Deswegen schauen wir uns lieber nach einer generischen Lösung um, wie wir diese Sicherheit von vornherein erhöhen können.

Einen möglichen Ansatz soll dieser zweiteilige Artikel aufzeigen. Im ersten Teil werden wir uns ein eigenes System zur Einwegverschlüsselung von Passwörtern erzeugen. Der zweite Teil wird dann die Erstellung einer Android-App auf Basis dieses Systems umfassen, sodass dann das Gerät, auf dem diese App läuft, die Funktion eines *Hardware-Token*-Generators übernehmen kann.

Doch beginnen wir zunächst mit ein wenig Ist-Analyse zu Passwörtern und deren Sicherheit, ehe wir unser System ausarbeiten.

Passwortsicherheit

Die Auswahl von Passwörtern ist eigentlich ganz leicht, man denkt sich ein Code-Wort aus, das den Zugriff auf damit verschlossene Daten nur für den Inhaber des Schlüssels, in Form des Passworts, freigeben soll. In der Theorie ist das eine nette Idee, in der Praxis gibt es leider jedoch auch andere Personen, die gerne die verschlüsselten Daten einsehen würden und deshalb versuchen dieses Passwort zu knacken, um sich ebenso Zugang zu verschaffen.

Leider gibt es genügend Möglichkeiten und Fehler, die ihnen dies noch erleichtern. Den ersten Fehler kann man schon selbst durch die Wahl eines zu einfachen Passwortes begehen. Meist wählt man etwas, was man sich gut merken kann, wie den Namen eines Menschen der einem nahe steht oder von etwas, was man sehr mag. Beides können Bekannte aber sehr leicht erraten, da ihnen dies auch bekannt ist.

Auch sehr häufig wird einfach nur ein Wort oder eine Zahlenkombination verwendet, die wiederum durch sogenannte Wörterbuch-Attacken, bei denen eine riesige Liste an häufig verwendeten Wörtern durchprobiert wird, einfach auszuhebeln ist. Da dieses Problem jedoch bekannt ist, wird man bei der Vergabe von Passwörtern meist dazu angehalten doch eine gewisse Länge und Komplexität in Form von Zahlen und Sonderzeichen einzuhalten. Dies rührt vor allem daher, dass ein Passwort auch mit purer Gewalt (*brute force*), in Form von systematischem Ausprobieren aller möglichen Kombinationen, ermittelt werden kann. Die benötigte Zeit, um ein Passwort derart zu ermitteln, steigt hier rasant mit zunehmender Länge und der Menge der möglichen verwendeten Zeichen.

Die Systeme, in denen das Passwort oder ein „Fingerabdruck“ davon abgelegt werden, können jedoch auch selbst eine Schwachstelle sein. Hacker könnten sich Zugang zu dem System und den gespeicherten Daten verschaffen. Aus diesem Grund sollten Passwörter dort nie im Klartext vorliegen, sondern mindestens verschlüsselt. Am besten ist eine Einwegverschlüsselung, da man so, selbst wenn das Verfahren bekannt wird, keine Rückrechnung auf das ursprüngliche Passwort vornehmen kann.

Durch die Gefahr eines Hacker-Angriffs auf ein einzelnes System, sollte man ein Passwort auch nie bei mehreren Systemen verwenden. Denn wenn der Angriff erfolgreich sein sollte, kann der Hacker prinzipiell mit diesen Zugangsdaten auch auf die anderen Systeme zugreifen.

Passwortgeneratoren

Wie kann man dann also ein möglichst sicheres Passwort auswählen? Bei all den aufgezeigten Schwachstellen ist das natürlich nicht leicht, vor allem wenn man es sich auch noch merken muss, und erst recht, wenn man ein ganzes Paket davon verwalten soll.

Eine Lösung wäre es einen Passwortgenerator zu verwenden, der einem zufällig erstellte komplexe Passwörter ausspuckt, die anscheinend nur aus wahllos aneinander gereihten Zeichen bestehen. Dies können Buchstaben, Zahlen und vor allem auch Sonderzeichen sein. Da man sich diese jedoch so gut wie gar nicht merken kann, ist man fast darauf angewiesen zusätzlich noch einen Passwortsafe zu verwenden, um die Passwörter später wieder abzurufen. In diesem virtuellen Safe werden sie abgelegt und üblicherweise nochmals durch ein Passwort, das man sich merken kann, abgesichert.

Eine andere Alternative wäre es, sich ein Passwortsystem zu überlegen, anhand dessen man sich das aktuell nötige Passwort herleiten kann. So ist unter anderem sichergestellt, dass andere keinen Vollzugriff auf alles bekommen, wenn das Passwort auf einem System geknackt wird. Denn mit diesem kann man sich leicht viele verschiedene Passwörter generieren. Sollte jedoch das System selbst bekannt werden, kann der Angreifer sich wiederum natürlich auch überall Zugang verschaffen. Ein weiterer Nachteil eines Systems wäre es auch, dass sie wiederum eher leichter lesbare Passworte „generieren“. Denn tendenziell bastelt man kein komplexes System, für das man Minuten lang im Kopf oder auf dem Papier hin und her transformiert und rechnet, bis man das endgültige Ergebnis hat.

Aber warum sollte man nicht auch beides in Kombination verwenden und so die Vorteile eines Systems und die offensichtliche Zufälligkeit eines Generators bekommen? Genau hier soll der Artikel ansetzen und die Grundideen für ein solches System betrachten.

Wir werden uns deshalb exemplarisch ein eigenes System schaffen, das „kryptische“ Passwörter erzeugen wird. Das System soll einen beliebigen Ausgangstext bekommen, und heraus kommt ein „verschlüsselter“ Text, der als Passwort verwendet werden kann. So können wir den Ausgangstext für die verschiedenen Systeme variieren und müssen uns eigentlich nicht einmal über die Sicherheit dieser Texte groß Gedanken machen. Im Gegenteil, wir könnten sogar recht einfache Wörter hernehmen, die wir uns im Gegenzug leicht merken können.

Um jedoch die höhere Komplexität bei der Erstellung des Passwortes anhand des Systems in Einklang mit schneller zeitlicher Verfügbarkeit zu bekommen, stützen wir uns auf einen kleinen Rechner in Form eines Smartphones. In unserem Falle werden wir eine App auf Basis des Android-Betriebssystems erstellen, was im zweiten Teil des Artikels, verknüpft mit einem kleinen Einstieg in Android, vorgestellt wird. Diese App erfüllt damit einen ähnlichen Anwendungszweck, den z. B. *Hardware-Token*-Generatoren für TANs im Online-Banking oder *Zugriffstoken*-Generatoren beim Login in bestimmte Systeme erfüllen. Man könnte es also als *Tokenized Password*-Generator betiteln.

Beleuchtet man die Angriffsmöglichkeiten auf diese App-Lösung auf dem eigenen Android-Gerät, so muss ein potentieller Angreifer aus dem näheren persönlichen oder räumlichen Umfeld stammen, um Zugriff auf das Gerät oder die darauf laufende App zu erlangen. Im Falle von letzterem werden wir uns zusätzlich mit einem zufällig generierten zweiten Schlüsselteil (mehr dazu im nächsten Abschnitt) absichern, den der Angreifer ebenso erbeuten muss. Dies erlaubt uns also sogar auch die erstellte App weiter zu verteilen, da jeder Nutzer seinen eigenen Schlüsselteil besitzt.

Analog zum Angriffsszenario, in dem der Bösewicht in Besitz der Hardware gelangt, ist man auch bei Verlust gegen zufällige Finder geschützt. In diesem Fall greift die zusätzliche Hürde, dass der Angreifer auch wissen muss, welche Ausgangstexte wir verwenden, um unsere Passwörter zu generieren. Hat er diese Information nicht, so bleibt ihm allenfalls die *Brute-Force*-Attacke, um so systematisch mit unserer personalisierten App die Ausgangstexte und die dazugehörigen Passwortkonstrukte zu ermitteln. Deswegen ist es sinnvoll auch bei den Ausgangstexten zumindest eine gewisse Grundkomplexität einzubauen, um diese Möglichkeit noch zu erschweren.

Wie man sieht, schränken wir die Möglichkeiten einer Kompromittierung sehr stark ein. Im Gegenzug haben wir jedoch auch eine Abhängigkeit zu unserer App und dem generierten Schlüsselteil. Sollte uns eines von beiden vollständig abhanden kommen, sind wir – salopp gesagt – völlig aufgeschmissen. Wir sollten somit also noch mindestens irgendwo eine Kopie der Beiden aufbewahren.

Passwort mit System

Auf Basis dieser Vorbetrachtungen wollen wir uns ersteinmal Gedanken über das System selbst machen. Welche Anforderungen haben wir also an unser System?

Es muss

- ein Ergebnis erzeugen, das den Anforderungen der Systeme bei der Passworteinrichtung genügt; das heißt,

dass Buchstaben, Zahlen und Sonderzeichen enthalten sein sollten;

- ein funktionaler Algorithmus sein, d. h. gleiche Eingaben führen zu gleichen Ausgaben; auf Zufallswerte müssen wir also verzichten;
- asymmetrisch sein, sodass man aus dem verschlüsselten Ergebnis nicht auf den Ausgangstext schließen kann;
- variabel sein; wir sollten also einzelne Parameter und nicht nur unseren Text anpassen können, um möglichst immer verschiedene Ergebnisse, unter Berücksichtigung des vorherigen Punktes, zu erhalten.

Der Einfachheit halber werden wir hier nur ein weniger komplexes System unter Verwendung von Vertauschungen, Ersetzungen und *Hashing* verwenden. Dies soll nur als Basisidee für einen eigenen Algorithmus dienen, den man für seinen persönlichen Passwortgenerator beliebig erweitern kann.

Die Grundidee des Generators ist recht einfach: Zunächst generieren wir uns einen sogenannten *Seed*, den wir zum Teil dem Text voranstellen werden und den Rest hinten anfügen. Dies ist der zweite Schlüsselteil, den wir oben schon angesprochen hatten, und den wir gut aufbewahren sollten. *Seeds* werden üblicherweise aus zufälligen Parametern, wie z. B. Mausbewegungen erzeugt, sodass sie schwer nachzubilden sind.

Für unser Beispiel vereinfachen wir dies jedoch und nehmen an, dass uns der String *Kaffeeklatsch* als *Seed* generiert wurde. Darüber hinaus ermitteln wir noch die Länge des zusammengesetzten Textes und fügen diese ebenso am Ende an.

Der nächste Schritt wird eine systematische Vertauschung sein. Hierzu nehmen wir uns die Zahlen 1 bis 9 und vertauschen jeweils die Positionen benachbarter Vielfacher im zusammengesetzten Text. Das heißt, wir vertauschen erst Position 0 × 1 mit 1 × 1, dann 1 × 1 mit 2 × 1, dann 2 × 1 mit 3 × 1, usw. Wenn wir dann das Ende des Strings erreicht haben, dann verfahren wir ebenso von vorne mit den Zahlen 2, 3, 4 usw. Einige Stellen werden so zwar kaum getauscht werden, während die jeweils erste Stelle kontinuierlich wandert, aber es ist auch kein Muss, dass alle Stellen wild vertauscht werden.

Bisher können wir mit etwas Aufwand, und wenn uns der *Seed* bekannt ist, immer noch die ganze Rechnerei umkehren und so auf den verwendeten Ursprungstext kommen. Da wir dies nicht wollen, werden wir nun unseren vertauschten Text mittels *SHA-256* hashen. Das Ganze machen wir sogar dreizehn Mal, da wir so recht sicher sein können, dass auch nicht mit einer *Hash*-Tabelle auf unseren vertauschten Text geschlossen werden kann. Denn da Passwörter niemals im Klartext gespeichert werden sollen und deshalb beispielsweise nur deren *MD5-Hash* abgelegt wird, haben Hacker angefangen, sich sogenannte *Hash*-Tabellen anzulegen, in denen Texte und ihre *MD5-Hashes* abgelegt werden. Wird dann eine solche Datenbank mit gehashten Passwörtern geknackt, werden diese Hash-Werte einfach in der Tabelle nachgeschaut, und bei Treffern hat man so wieder das eigentliche Passwort ermittelt. Da es für unseren krummen Wert von dreizehn Mal, aber kaum eine Tabelle geben wird, haben wir uns gegen diesen Fall auch schon abgesichert.

Unser Hashing liefert nun einen schönen 64 Zeichen langen String, aber wir haben uns leider von unserem ersten Anforderungspunkt etwas wegbewegt, da dies nur Hexadezimale Ziffern sind und somit nur die Zeichen von 0 bis 9 und von A bis F umfassen. Deswegen bedienen wir uns nun der ASCII-Tabelle und betrachten jeweils zwei Stellen als Hexadezimalzahl, der dem Code des ASCII-Zeichens in unserem String entsprechen soll.

Unsere für Passwörter relevante Zeichenmenge bewegt sich allerdings nur in dem Bereich von 33 (Hex 21) bis 126 (Hex 7E) – mit eventuellen Ausnahmen, die wir noch optional rausfiltern müssten – während in unserem Hash-String ja die komplette Bandbreite von 00 (0) bis FF (255) vorkommen kann. Das werden wir mit einer kleinen Normalisierung korrigieren, in dem wir bei ungültigen Werten diese solange verdoppeln bis wir einen gültigen Wert bekommen. Zusätzlich dazu werden wir bei Werten die größer oder gleich 128 sind, immer nur mit dem ganzzahligen Rest bei Division durch 128 weiterrechnen.

Somit erhalten wir am Ende ein 32 Zeichen langes Kauderwelsch, von dem wir nun einen beliebig langen Teil als Passwort verwenden können. Optimalerweise sollten wir uns hier im Bereich von 16 Stellen bewegen, um einen Kompromiss aus Sicherheit und Eingabekomfort zu erhalten. Letzterer ist auch der Grund, warum wir uns auf den begrenzten Bereich von ASCII-Zeichen beschränkt haben, da der Rest kaum noch auf der Tastatur zu finden ist, und von den meisten Systemen ohnehin abgelehnt wird. Da auch nicht jedes System alle von uns jetzt zugelassenen Zeichen unterstützt, müssten wir im letzten Schritt auch diese nochmal ausfiltern und unser erzeugtes Passwort entsprechend bereinigen und eventuell kürzen.

Im Folgenden ist nun der Quellcode unseres Beispiels zu finden. Gibt man bei diesem den eigentlich als

Passwort völlig ungeeigneten Text *password* ein und lässt sich ein achtstelliges Passwort generieren, erhält man im Schritt 1 zunächst den Text:

```
Kaffeepasswordklatsch21
```

Nach der Vertauschung in Schritt 2 ergibt sich dieser Text.

```
lfesodctKp2r1kfessahawa
```

Schritt 3 liefert dann den hexadezimalen Hashwert, der sich durch das mehrmalige Hashen ergibt.

```
020b232669132422f451443344534611
c11e2ad890a9e0de8bf557724e1b678f
```

Die Transkription in Schritt 4 ergibt dann die gewünschte „kryptische“ Zeichenfolge.

```
@,#&i&$"hQD3DSF"@<*0@R@<,jWrN6g<
```

In Schritt 5 wird dann das eigentliche Passwort extrahiert und auf

```
DSF"@<*0
```

reduziert. Filtert man danach etwa die Größer- und Kleiner-Zeichen, das Anführungszeichen und den Stern als ungültige Zeichen aus dem Ergebnis, dann ergibt sich daraus das sehr ansehnliche:

```
D3DSF@0@
```

Den Quellcode werden wir dann auch nahezu unverändert im nächsten Teil des Artikels verwenden, wenn es darum geht, die App für unser Android-Telefon zu erstellen.

```
public class PasswordCipher {

   private static final
   String staticSeed = "Kaffeeklatsch";

   public static String encrpytText(
      String input,
      int charAmount,
      Set<Character> goodChars,
      Set<Character> badChars
   ) {
      String r = seedInput(input);
      r = switchPositionsInInput(r);
      r = hashInputMultipleTimes(r, 13);
      r = convertHexToAscii(r);
      r = sliceAndFilterInput(
         r, charAmount, goodChars, badChars
      );
      return r;
   }

   //Schritt 1 - Seeden
   private static String seedInput(String input) {
      StringBuilder sb = new StringBuilder();
      // Seed zum Teil voranstellen und zum Teil anhängen
      sb.append(staticSeed.substring(0, 6));
      sb.append(input);
      sb.append(staticSeed.substring(6));
      // Länge des zusammengesetzten Textes anfügen
      sb.append(sb.length());
      return sb.toString();
   }

   //Schritt 2 - Vertauschen
   private static String switchPositionsInInput(
      String input
   ) {
      StringBuilder sb = new StringBuilder(input);
      for (int index = 1; index < 10; index ++) {
         int offset = 0;
         while (index * (offset + 1) < sb.length()) {
            int pos1 = index * offset;
            int pos2 = index * (offset + 1);
            sb.insert(pos1, sb.charAt(pos2));
            sb.insert(pos2 + 1, sb.charAt(pos1 + 1));
            sb.deleteCharAt(pos1+1);
            sb.deleteCharAt(pos2+1);
            offset++;
         }
      }
      return sb.toString();
   }

   //Schritt 3 - Hashen
   private static String hashInputMultipleTimes(
      String input, int recursionDeepness
   ) {
      StringBuilder sb = new StringBuilder(input);
      if (recursionDeepness > 1) {
         try {
            // Hashen mit SHA-256
            MessageDigest md;
            md = MessageDigest.getInstance("SHA-256");
            md.update(input.getBytes());
            byte[] hashedBytes = md.digest();

            // Bytes in Hexadezimalen String konvertieren
            sb = new StringBuilder();
            for (int i = 0; i < hashedBytes.length; i++) {
               String hexChars;
               hexChars = Integer.toHexString(
                     0xFF & hashedBytes[i]
               );
               if (hexChars.length() == 1) {
                     sb.append('0');
               }
               sb.append(hexChars);
            }
```

```
                // Nächste Rekursionsstufe starten
                sb.replace(
                    0,
                    sb.length(),
                    hashInputMultipleTimes(
                        sb.toString(),
                        recursionDeepness - 1
                    )
                );
            } catch (NoSuchAlgorithmException e) {
                e.printStackTrace();
            }
        }
        return sb.toString();
    }

    //Schritt 4 - HexString in ASCII umwandeln
    private static String convertHexToAscii(String input) {
        StringBuilder sb  = new StringBuilder();
        for (int i = 0; i + 2 <= input.length(); i += 2) {
            int code = Integer.parseInt(input.substring(i, i + 2), 16);
            while (code < 33 || code > 126) {
                code *= 2;
                if (code > 128) {
                    code %= 128;
                }
            }
            sb.append((char) code);
        }
        return sb.toString();
    }

    //Schritt 5 - Slicen und Filtern
    private static String sliceAndFilterInput(
        String input,
        int charAmount,
        Set<Character> goodChars,
        Set<Character> badChars
    ) {
        StringBuilder sb = new StringBuilder(input);
        // Ungültige Zeichen ausfiltern
        for (int i = sb.length() - 1; i >= 0 ; i--) {
            if (
                !goodChars.contains(sb.charAt(i))
                ||
                badChars.contains(sb.charAt(i))
            ) {
                sb.deleteCharAt(i);
            }
        }
        // Wir wollen ein Stück aus der Mitte haben
        if (charAmount < sb.length()) {
            int start = (sb.length() - charAmount) /2;
            sb.delete(0, start);
            sb.delete(charAmount, sb.length());
        }
        return sb.toString();
    }
}
```


Herbstcampus

Wissenstransfer par excellence

3.– 6. September 2012
in Nürnberg

Fehlerteufelchen

Auf Safari

von Sascha Gross

Benutzer bewusst bei Web-Anwendungen auf eine Menge von Browsern zu beschränken ist nicht schön, aber durchaus gängige Praxis.

Wenn aber Safari-Surfer unbewusst ausgeschlossen werden, ist das schon ärgerlich.

Bevor es zum eigentlichen Thema geht, erst ein bisschen Vorrede zu der Artikelserie *Fehlerteufelchen*:

Sich mit erfolgreichen Projekten zu profilieren, liegt wohl in unserer Natur. Gerne schmunzelt man über *Das Allerletzte* auf der vorletzten Seite im KaffeeKlatsch und ist froh, wenn man dort keinen Code-Schnipsel von sich entdeckt. Aber selbst bei erfolgreichen Projekten geht mal etwas schief, und meist sind es ganz kleine Fehlerchen, mit großen Folgen. In loser Reihe berichte ich und hoffentlich auch Andere über ihr oder das Missgeschick von Kolleginnen und Kollegen. Also fühlen sie sich frei, auch ihre kuriosen Geschichten und Anekdoten hier zum Besten zu geben.

Erfolgreiche Projektberichte – Berichte aus gescheiterten Projekten bekommt man ja selten zu hören – sind hilfreich für die eigene Arbeit, denn aus ihnen kann man Architekturen, Vorgehensweisen und Technologien herausziehen, die sich als bewährt, praktikabel und stabil erwiesen haben. Also aus dem Gelingen Anderer lernen. Aus eigenen Fehlern zu lernen ist meist besser und nachhaltiger, aber leider auch schmerzhafter. Aus Fehlern Anderer Know-how zu ziehen ist dagegen lustiger und bleibt somit auch eher haften.

Nach langer Vorrede mit kurzem Sinn, auf zur genaueren Beschreibung des Fehlerszenarios. Wir haben eine Java-basierte Web-Anwendung, bei der man sich authentifizieren muss. Eigentlich funktioniert es mit allen Browsern, nur dummerweise nicht mit Safari. Normalerweise ist es ja eher so, dass etwas im Internet Explorer nicht funktioniert oder genau andersrum. Was läuft hier nur falsch? Eigentlich kann man hier gar nichts falsch machen, dass ist eine Situation aus dem Lehrbuch oder besser gesagt aus der *Java Servlet Specification* [1] Kapitel 13.6.3 *Form Based Authentication*.

```
<form method="POST" action="j_security_check">
  <input type="text" name="j_username">
  <input type="password" name="j_password">
</form>
```

Also gut, lieber mal der Fehlermeldung nicht trauen und das Fehlerverhalten selbst reproduzieren. Und siehe da, auf dem eigenen Server gibt es das Problem nicht und auch auf den Integrationsservern der Entwicklung kein Fehlverhalten: Das Login ist ohne Probleme auch mit Safari möglich. Absurde Ideen und Hilflosigkeit machen sich breit. Liegt es daran, dass der Tester Safari auf Windows benutzt? Ist der Tester nicht in der Lage sein Passwort im Safari einzugeben? Zu guter Letzt lässt man sich das Fehlverhalten nochmal zeigen, vielleicht macht man ja beim Reproduzieren etwas anders. Und schau an, auf der Produktionsumgebung geht es wirklich nicht; auf den Entwicklungsumgebungen gibt es auch beim Tester keine Probleme.

Was ist nun der Unterschied bei den Umgebungen? Das Betriebssystem? Eine recht absurde Vorstellung, dass es daran liegen könnte. In der Entwicklung läuft es auch ohne Probleme auf Mac, Ubuntu und Solaris. Wieso sollte dann das Betriebssytem *CentOs* ausgerechnet auf der Produktionsumgebung Probleme bereiten? Macht es auch nicht, könnte es aber bei anderen Problemen auch beim plattformunabhängigen Java.

Ein weiterer Unterschied zwischen den Produktions- und Entwicklungsumgebungen ist das Protokoll (*https* und *http*), also auf dem eigenen Tomcat, auf dem die Anwendung läuft *https* beigebracht [2]. Leider, wie schon zu befürchten war, führt das auch nicht zur Reproduktion des Fehlverhaltens.

Nach nochmaligem innehalten und nochmaliger Rückbesinnung auf das Problem, kam der richtige Denkanstoß. Das Login funktioniert scheinbar nicht, aber vielleicht gibt es ja irgendwelche Probleme mit der *Session*, denn ohne *Session* auch kein Login.

Die Session wird bei Java-basierten Web-Anwendungen entweder über *;jsessionid=...* in der URL oder als Cookie *JSESSIONID* identifiziert. Mittlerweile bietet jeder Bowser direkt oder über Plugins Hilfe für die Entwicklung und somit auch die Möglichkeit, sich Informationen über die Seite und die Cookies anzusehen. Mit den Entwickler-Tools in Safari die Cookies angeschaut (Aktivierung über Einstellungen → Erweitert → Menü *Entwickler* in der Menüleiste anzeigen) sieht man, wie zu erwarten war, gibt es ein Cookie *JSESSIONID*. Beim Blick auf die Cookies im Firefox mit Firebug [4] und

integriertem Firecookie [5] waren zwei Cookies vorhanden und zwar *JSESSIONID* und *jsessionid*. Die bei Firefox (*jsessionid*) glich der im Safari. Warum hat nun Firefox und die anderen Browser zwei Cookies mit ziemlich gleichen Namen und Safari nur eins?

In der Produktivumgebung gibt es wegen Last und Ausfallsicherheit mehrere Server (im konkreten Fall zwei Server) vor denen ein *Load-Balancer* steht, der die Anfragen aufteilt. Vom Load-Balancer kommt das Cookie *jsessionid*. Schaut man jetzt in Spezifikation für Cookies [3] hinein, steht dort indirekt drin: Cookie-Namen sind *case-insensitive*. In anderen Worten Groß- und Kleinschreibung werden ignoriert. Anscheinend nimmt das aber nur Safari ernst und somit gibt es das Cookie *JSESSIONID* nur einmal und dummerweise mit dem Wert des Load-Balancers, mit dem der eigentliche Applikations-Server nichts anfangen kann und somit immer wieder eine neue Session erzeugt, auf der dann auch der Benutzer nicht angemeldet ist. Zum Glück muss man sagen, wurde der Wert des Load-Balancers übernommen, sonst hätte man den sporadischen Effekt gehabt, dass der Benutzer unverhofft irgendwann auf einem anderen Server gelandet wäre und wegen nicht vorhandener *Session*-Replikation aus der Anwendung geworfen würde. So wurde er nur ausgesperrt und konnte aber mit einem anderen Browser als Safari arbeiten.

Die Lösung letztendlich war den Load-Balancer umzukonfigurieren, sodass er für die Verteilung und Zuordnung der Clients auf die einzelnen Server einen anderen Cookie-Namen verwendet.

Dieser kleine Fehler zeigt einmal mehr, dass vor der Produktionsumgebung eine ähnliche (im Idealfall identische) Umgebung gebraucht wird, um solche Fehler im vornherein auszuschließen bzw. reproduzieren zu können und dort die Fehlerbehebung zu erproben, um nicht am offenen Herzen operieren zu müssen.

In diesem Projekt gibt es diese Abnahmeumgebung auch und somit konnte dort ebenso das Problem reproduziert und durch die Umkonfiguration des Load-Balancers vor dem Produktivgang getestet werden. Jetzt hätte nur noch vorher mit Safari als möglicher Browser getestet werden müssen.

Referenzen

[1] Java Community Process *JSR 315: JavaTM Servlet 3.0 Specification* http://jcp.org/en/jsr/summary?id=315

[2] The Appache Software Foundation *SSL Configuration HOW-TO* http://tomcat.apache.org/tomcat-6.0-doc/ssl-howto.html

[3] *HTTP State Management Mechanism*, http://tools.ietf.org/html/rfc2965

[4] Firebug, http://getfirebug.com

[5] Software is hard *Firecookie* http://www.softwareishard.com/blog/firecookie

Herbstcampus

Wissenstransfer par excellence

Der Herbstcampus möchte sich ein bisschen von den üblichen Konferenzen abheben und deshalb konkrete Hilfe für Software-Entwickler, Architekten und Projektleiter bieten.

Dazu sollen die in Frage kommenden Themen möglichst in verschiedenen Vorträgen besetzt werden: als Einführung, Erfahrungsbericht oder problemlösender Vortrag. Darüber hinaus können Tutorien die Einführung oder die Vertiefung in ein Thema ermöglichen.

Haben Sie ein passendes Thema oder Interesse, einen Vortrag zu halten? Dann fragen Sie einfach bei info@bookware.de nach den Beitragsinformationen oder lesen Sie diese unter www.herbstcampus.de nach.

3.– 6. September 2012 in Nürnberg

Des Programmierers kleine Vergnügen

Überläufer

von Michael Wiedeking

Wer einen Tageskilometerzähler im Auto hat, mag es schon einmal miterlebt haben: Einen Überlauf. Anstatt der beeindruckenden 1218 km von Nürnberg nach Hamburg und wieder zurück zeigt der nach einem solchen beispielsweise nur noch bescheidene 218 km davon an. Dieses Schicksal bleibt einem gelegentlich auch bei Zahlen im Rechner nicht erspart, und so stellt sich die Frage, wie man diese Überläufe vorhersagen kann.

Eine 32-Bit-Zahl ist inzwischen nicht mehr so richtig groß. Misst man beispielsweise den Plattenplatz in Byte, so reicht ein solche Zahl schon lange nicht mehr aus. Mit 64-Bit scheint man auf der sicheren Seite zu sein, aber auch dort stößt man gelegentlich schon auf Grenzen. Das ist vor allem dann schlecht, wenn man diese Grenzen insbesondere bei Zwischenergebnissen nicht bemerkt und es so mit schwer nachzuvollziehenden Überläufen zu tun hat, die aus sehr großen Zahlen ganz schnell mal allzu kleine machen. Wie kann man sich also vor solch unerwünschten Überläufen schützen?

Von den vier wichtigsten ganzzahligen Operationen Addition, Subtraktion, Multiplikation und Division können eigentlich nur die ersten drei einen Überlauf erzeugen – wenn man einmal davon absieht, dass wegen der Asymmetrie der Zahlen im Zweierkomplement *MIN_VALUE / –1* gleichermaßen wie *–MIN_VALUE* ein falsches Ergebnis liefert (nämlich *MIN_VALUE*). Losgelöst davon muss, wie so oft, auch bei diesen Operationen zwischen vorzeichenbehafteten und vorzeichenlosen Zahlen unterschieden werden. Und wie so oft ist der vorzeichenlose Fall viel einfacher.

Werden zwei vorzeichenlose ganze Zahlen a und b addiert, so findet bei deren Summe $s = a + b$ genau dann ein Überlauf statt, wenn diese kleiner ist als a oder b. Das lässt sich leicht nachvollziehen: Normalerweise ist die Summe immer größer oder gleich der Summanden. Also muss sowohl $a + b \geq a$ als auch $a + b \geq b$ gelten. Ist das nicht der Fall, so muss demnach ein Überlauf stattgefunden haben. Werden diese zwei vorzeichenlosen Zahlen subtrahiert, so muss analog gelten, dass die Differenz kleiner ist als Minuend und Subtrahend: $a - b \leq a$ und $a - b \leq b$. Bei Addition und Subtraktion genügt es diesen Vergleich mit einem der Summanden bzw. dem Minuenden oder Subtrahenden zu machen; jeweils beide Bedingungen zu testen ist nicht nötig.

```
unsigned uadd(unsigned a, unsigned b) {
  unsigned r = a + b;
  if (r <u a) {
    throw new OverflowException();
  }
  return r;
}

unsigned usub(unsigned a, unsigned b) {
  unsigned r = a – b;
  if (r >u a) {
    throw new OverflowException();
  }
  return r;
}
```

Bei der Multiplikation, die immer ein Ergenbis doppelter Breite liefert, wäre es trivial, wenn man aus der Hochsprache auf deren obere Hälfte des Ergebnisses zugreifen könnte. Dann müsste man nämlich nur prüfen, ob diese ungleich Null ist und wüsste dann, dass ein Überlauf stattgefunden hat. Leider ist das in den meisten Sprachen nicht möglich und deshalb muss man zu an-

deren Mitteln greifen. Bildet man nämlich das Produkt $p = a \cdot b$ mit den vorzeichenlosen Zahlen a und b, dann muss natürlich auch gelten, dass $p / a = b$ bzw. $p / b = a$ ist. Allerdings ist hier Vorsicht geboten: Ist der Divisor bei der Probe Null, so schlägt die Division fehl.

```
unsigned umul(unsigned a, unsigned b) {
  unsigned r = a * b;
  if ((a != 0) && ((r / a) != b)) {
    throw new OverflowException();
  }
  return r;
}
```

Die bedauernswerten Java-Entwickler können ja mit den vorzeichenlosen Zahlen nichts anfangen, wenngleich sie grundsätzlich die Möglichkeit hätten; allerdings würden dann die obigen Implementierungen bis zur Unkenntlichkeit verkompliziert werden. Aber bei den vorzeichenbehafteten Lösungen können jene wieder hellhörig werden.

Das Problem mit den vorzeichenbehafteten Zahlen ist, dass sie gleich in zwei Richtungen überlaufen können. Aber auch hier kann man sich leicht überlegen, wann überhaupt ein Überlauf stattfinden kann (was man mit ein bisschen fallunterscheidender Schreibarbeit auch recht leicht beweisen kann). Haben zwei Zahlen a und b unterschiedliche Vorzeichen, so kann nichts passieren, denn die Summe von der größten Zahl (MAX_VALUE) und der kleinsten Zahl (MIN_VALUE) lässt sich ohne Überlauf problemlos berechnen. Alle anderen Kombinationen dieser Art lassen sich dann ebenfalls korrekt berechnen, da sie per Definition größer als MIN_VALUE und kleiner als MAX_VALUE sind.

Ein Überlauf kann also überhaupt nur dann auftreten, wenn die beiden Summanden das selbe Vorzeichen haben. Normalerweise behalten zwei Zahlen mit gleichem Vorzeichen auch, wenn sie addiert werden, dieses Vorzeichen. Ein Überlauf hat also genau dann stattgefunden, wenn sich das Vorzeichen der Summe $s = a + b$ von dem Vorzeichen von a und b unterscheidet. Dies lässt sich selbstverständlich mit einer passenden Funktion, die das Vorzeichen extrahiert, und geeigneten Bedingungen herausfinden, aber das wäre ja keine Herausforderung.

Gesucht ist also ein Ausdruck, der elegant zeigt, dass das Überlaufkriterium erfüllt ist. Hierbei interessiert uns nur das höchstwertigste Bit, da dort das Vorzeichen untergebracht ist, welches genau dann gesetzt ist, wenn die Zahl negativ ist. Um herauszufinden, ob zwei Zahlen verschiedene Vorzeichen haben, braucht man diese nur zu „ver-*x*-odern“, also mit einem Exklusiv-Oder (^) zu verknüpfen, und anschließend das höchstwertigste Bit zu prüfen. Dies wird nur dann gesetzt sein, wenn sich die Vorzeichen unterscheiden.

Die Ausdrücke s ^ a und s ^ b zeigen also in dem höchstwertigsten Bit an, ob die Summe das gleiche Vorzeichen wie seine Summanden hat. Nur im Fall, in dem die Vorzeichen beider Summanden verschieden von dem der Summe sind, hat ein Überlauf stattgefunden. Und nur in diesem Fall sind beide höchstwertigen Bits der Oder-Verknüpfungen gesetzt. Also nur wenn das eine *und* das andere Bit gesetzt sind, hat ein Überlauf stattgefunden (denn ansonsten hätten ja a und b verschiedene Vorzeichen). Ob das Vorzeichen-Bit dann gesetzt ist, lässt sich dann überprüfen, in dem man testet, ob die resultierende Zahl kleiner als Null ist.

```
signed add(signed a, signed b) {
  signed r = a + b;
  if (((r ^ a) & (r ^ b)) < 0) {
    throw new OverflowException();
  }
  return r;
}
```

Überträgt man nun die Überlegungen auf die Subtraktion $s = a - b$, indem man sich diese als Summe $s = a + (-b)$ vorstellt, so ergibt sich daraus, dass ein Überlauf nur dann stattfinden kann, wenn a und b das *gleiche* Vorzeichen haben. Analog muss der obige Satz angepasst werden: Also nur wenn das eine *und nicht* das andere Bit gesetzt sind, hat ein Überlauf stattgefunden. Das lässt sich einfach dadurch erreichen, indem man den zweiten Teil der *Und*-Verknüpfung negiert.

```
signed add(signed a, signed b) {
  signed r = a - b;
  if (((r ^ a) & ~(r ^ b)) < 0) {
    throw new OverflowException();
  }
  return r;
}
```

Die Multiplikation mit vorzeichenbehafteten Zahlen gleicht weitestgehend der vorzeichenlosen Version. Auch hier muss bei dem Produkt $p = a \cdot b$, mit den vorzeichenbehafteten Zahlen a und b, die Nachbedingung $p / a = b$ gelten, $a \neq 0$ vorausgesetzt. Allerdings gibt es hier ja noch das oben angedeutete Problem mit der Division MIN_VALUE /–1. Das zu ignorieren ist jetzt leider unmöglich, denn während diese Division im Normalfall – nun ja – „normal“ ist, ist sie in einer sicheren Umgebung nicht akzeptabel. Es bedarf also nicht nur einer sicheren Division sondern auch einer Multiplikation, bei der dem

voranstehenden Test noch diese Bedingung hinzugefügt werden muss.

```
signed div(signed a, signed b) {
  if ((a == 0x8000 0000) & (b == –1)) {
    throw new OverflowException();
  }
  return a / b;
}

signed mul(signed a, signed b) {
  signed r = a * b;
  if (
    ((b < 0) & (a == 0x8000 0000))
    |
    ((a != 0) && ((r / a) != b))
  ) {
    throw new OverflowException();
  }
  return r;
}
```

Augenmerk sollte hier noch auf die beiden Operatoren & und | geworfen werden, die neben dem &&-Operator verwendet werden. Der Unterschied zwischen diesen ist, dass die „einfachen" Varianten beide Seiten immer ausführen, während die „doppelten" nur den ersten Operanden sicher auswerten – der zweite wird nur dann aufgerufen, wenn es nötig ist. Aus funktionaler Sicht kann obiger Ausdruck auch nur mit && und || anstatt & und | geschrieben werden. Aus Performanzsicht sind allerdings & und | effizienter, denn sie erfordern keinen bedingten Sprung. Das letzte && darf natürlich ausgetauscht werden, denn das würde ja im schlimmsten Fall mit einer *Division-durch-Null*-Ausnahme geahndet werden.

Abschließend stellt sich nur noch die Frage, was denn diejenigen machen sollen, die nur „sichere" Operationen haben. Bei denen würden ja fast alle der obigen Funktionen fehlschlagen, bevor diese feststellen würden, dass ein Überlauf stattfinden würde, weil der ja bereits stattgefunden hat. Das wirft die neue Frage auf, ob es nicht eine Methode gibt, mit der man einen möglichen Überlauf feststellen kann, ohne eine überlaufende Funktion dazu zu benötigen.

Aber das ist – wie so oft – eine ganz andere Frage, die auch – wie ebenso oft – ein ganz anderes Mal beantwortet werden muss.

Kurzbiographie

Michael Wiedeking (michael.wiedeking@mathema.de) ist Gründer und Geschäftsführer der MATHEMA Software GmbH, die sich von Anfang an mit Objekttechnologien und dem professionellen Einsatz von Java einen Namen gemacht hat. Er ist Java-Programmierer der ersten Stunde, „sammelt" Programmiersprachen und beschäftigt sich mit deren Design und Implementierung.

Herbstcampus

Wissenstransfer par excellence

3.– 6. September 2012
in Nürnberg

Deutsch für Informatiker

Gender Gap II und ein bisschen Grammatik

von Alexandra Specht

Dieses Mal möchte ich auf einen Leserbrief eingehen, der ein paar Fragen zu dem Thema der letzten Kolumne *Deutsch für InformatikerInnen*[1] aufwirft, auf die ich gerne eingehen möchte. Das Postscriptum ist dann das „bisschen Grammatik", das sich durch die Überschrift schon ankündigt.

Hier nun erst einmal der Brief:

Hallo Frau Specht!

Als an der deutschen Sprache Interessierter (aber Laie) habe ich Ihren Artikel im KaffeeKlatsch mit Interesse gelesen. Einen Punkt möchte ich zur Diskussion stellen. Sie schreiben, dass eine gut klingende Umschreibung beispielsweise auch "Studierende" oder "die Nutzenden" seien. Hier bin ich anderer Meinung: Eine Backende ist meines Erachtens nicht notwendigerweise eine Bäckerin und umgekehrt ist eine Bäckerin nicht immer und überall auch eine Backende. Ähnlich verhält es sich mit den Studentinnen und Studenten. Ich studiere zur Zeit ein Buch über Softwareentwicklung, bin also ein Studierender. Meine Zeit als Student ist aber leider lange, lange vorbei.

Wie ist hier Ihre Meinung? Gibt es eine allgemeine Meinung unter den Germanisten, oder wird dort das Thema (wenn überhaupt) kontrovers diskutiert?

Viele Grüße
Christian Heckler

Lieber Herr Heckler, vielen Dank für den Brief und die aufgeworfenen Fragen darin. Es ist mir ein Vergnügen, darauf einzugehen.

Ich gebe Ihnen uneingeschränkt recht: *Die Backenden* sind Frauen und oder Männer, die backen. Ob es sich um *Bäckerinnen* und oder *Bäcker* oder um *Schlosserinnen* und *Softwareentwicklerinnen* und vielleicht eine professionelle *Bäckerin* handelt, würde sich mir da auch nicht erschliessen. Generell wäre es für mich persönlich nur eine Beschreibung der Tätigkeit, die die Leute, die gemeint sind, ausüben.

Die Nutzenden finde ich eigentlich ganz schön, weil für mich damit tatsächlich auch Männer und Frauen eingeschlossen sind. Und das auch nur etwas über die Tätgikeit der betreffenden Menschen aussagt und keine besondere Bedeutung hat, außer dass diese Leute eine Gemeinsamkeit haben, nämlich gerade irgendetwas zu nutzen.

Bei *die Studierenden* würde ich differenzieren. *Die Studierenden* klingt für mich nach einem Begriff, der sich doch auch schon im Sprachgebrauch etabliert hat, eben vor allem im universitären Umfeld. Darum sind *die Studierenden* für mich tatsächlich Frauen und Männer, die an der Uni immatrikuliert sind. Sie studieren das Buch. Sie würden sich aber sicher auch nicht als Student bezeichnen, einfach, weil die Konnotation eine andere ist. Das würde für Sie dann sicher auch so klingen, als wären Sie wieder immatrikuliert. Darum würde ich Sie weder als Studierenden noch als Studenten bezeichnen, sondern als Herrn Heckler, der das Buch studiert.

Wie Sie sehen, halte ich meine Antwort sehr persönlich. Es gibt da (leider? Glücklicherweise?) keine einhellige Meinung, keine Regeln, keine Vorgaben. Ich meine, wenn es um Sprache geht, kann es auch nicht anders sein, da alle Sprechenden Sprache und einzelne Sätze unter-

[1] Ich persönlich mag das Binnen-I nicht, aber so stand es in der Überschrift der letzten Kolumne. Und da ich auch die rein männliche Bezeichnung nicht mag, sollte es eigentlich egal sein, oder?

schiedlich wahrnehmen. Ich bin mir sicher, dass ein und derselbe Satz bei fünf verschiedenen Personen sicherlich dutzende von Interpretationsmöglichkeiten bietet. Und da kommt es dann auf unzählige Komponenten an, nicht nur auf so weitreichende, wie zum Beispiel soziale Aspekte, Bildung, Beherrschung der Sprache an sich, sondern vielleicht auch nur darauf, ob man gut geschlafen hat, die Sonne scheint oder es in Strömen regnet und man ohne Schirm aus dem Haus gegangen ist.

So, nun kommen wir noch zu der Grammatik und den Fragen im Post Scriptum:

> *PS: Noch eine kleine Frage an die Germanistin: Ich hätte gedacht* backend *wäre ein Gerundium. Ein kurzes Googeln scheint eher darauf hinzuweisen, dass es ein Gerundium im Deutschen nicht gibt. Können Sie mir sagen, um welche Form es sich bei* backend *bzw.* dem Backenden *grammatikalisch eigentlich handelt?*

Gerne. Es ist richtig, im Deutschen gibt es weder ein Gerundivum noch ein Gerundium. Das gibt es beispielsweise im Lateinischen oder auch im Französischen und ist eine sehr schöne Art, elegant Sätze zu bilden. Was haben wir dann aber hier? *Backend* ist ein Partizip Präsens (Partizip I).

> Beispiel: *Sie verbrachte backend den Nachmittag.*

Hier wird das Partizip ohne Erweiterung als freies Prädikat gebraucht.

Bei *dem Backenden* handelt es sich um ein Partizip Perfekt (Partizip II). Das Verb *backen* wurde substantiviert und als Partizip II verwendet. Als solches musste es auch dekliniert werden. *Dem Backenden* ist der Dativ.[2]

> Beispiel: *Dem Backenden schmerzen schon die Arme vom Teigkneten.*

So. Ich hoffe, Sie sind einigermaßen zufrieden mit meiner Erwiderung. Wenn nicht: lassen Sie es mich wissen! Oder lesen Sie meine Erklärung noch einmal, wenn die Sonne scheint, bestimmt habe ich es ganz anders gemeint, als Sie es beim ersten Lesen aufgefasst haben!

Kurzbiographie

Alexandra Specht (alexandra.specht@mathema.de) ist Prokuristin bei der MATHEMA Software GmbH. Sie ist zuständig für Vertrieb, Kundenbetreuung, Mitarbeiterplanung, Marketing etc.
Nebenbei beschäftigt sie sich mit Vergnügen und so viel es geht mit Literatur und Sprache(n). Kommentare und Feedback nimmt sie gerne per E-Mail entgegen.

[2] Und der ist bekanntlich dem Genitiv sein Tod.

Herbstcampus

Wissenstransfer par excellence
3. – 6. September 2012, Nürnberg

Beitragsaufruf
(Call for Papers)

Der Herbstcampus möchte sich ein bisschen von den üblichen Konferenzen abheben und deshalb konkrete Hilfe für Software-Entwickler, Architekten und Projektleiter bieten.

Dazu sollen die in Frage kommenden Themen möglichst in verschiedenen Vorträgen besetzt werden:
als Einführung, Erfahrungsbericht oder problemlösender Vortrag.
Darüber hinaus können Tutorien die Einführung oder die Vertiefung in ein Thema ermöglichen.

Haben Sie ein passendes Thema oder Interesse, einen Vortrag zu halten?
Dann fragen Sie einfach bei

info@bookware.de

nach den Beitragsinformationen oder lesen Sie diese unter

www.herbstcampus.de

nach.

Kaffeesatz

Von wegen „einfach“

von Michael Wiedeking

Neuerdings habe auch ich ein Smartphone. Vorher hatte ich ein Nokia 6310i, also noch eins mit Dieselmotor, aber schon mit Internet, wenngleich ich letzteres nie benutzt habe. Und ich kann sagen, dass mir der Umstieg nicht leicht gefallen ist.

Das Beste an meinem alten Handy waren die Akku-Laufzeiten. Zum einen war das schon mein drittes Handy seiner Art, das ich jeweils dann getauscht hatte, wenn die Tastatur überhaupt nicht mehr funktionierte, und so war ich stolzer Besitzer von drei Akkus. Jeder dieser Akkus hatte bei meinem Telefonverhalten eine Laufzeit von mehr als einer Woche. So konnte ich überall hinfahren, ohne Angst davor haben zu müssen, mal ohne Strom dazustehen; ich konnte sogar getrost ohne Netzteil in Urlaub fahren.

Natürlich bin ich belächelt worden, wenn ich meinen alten Knochen aus der Tasche zog. Kaum einer konnte sich dann eine Bemerkung über dessen Alter, Größe oder Funktionsumfang verkneifen. Gelegentlich aber sah ich doch verstohlene, wehleidige Blicke, insbesondere wenn es um die Sprachqualität ging. Mit meinem neuen Smartphone kann ich nämlich ins Internet, Mails abrufen und Spiele spielen; nur das mit dem Telefonieren will nicht so richtig klappen. Die Qualität kann einfach mit der meines alten Telefons nicht mithalten, und ich kann fast immer sagen, ob mein Sprechpartner ebenfalls mit einem Smartphone telefoniert.

Das allerschlimmste ist aber die Bedienung. Bei meinem alten Telefon handelte es sich um ein abgeschlossenes Universum. Darin waren alle Funktionen in der Bedienung aufeinander abgestimmt. Und selbst der Lesefaulste konnte – wenn es denn überhaupt mal nötig wurde – in dem dreißigseitigen Heftchen nach einer Lösung suchen. Heute ist mir klar, dass intuitive Oberflächen nicht das halten, was sie versprechen, obwohl alle immer behaupten, dass die Bedienung sooo einfach wäre. Ich verstehe die Oberfläche einfach nicht und ärgere mich ständig darüber, warum die Dinge so kompliziert sind. Aber immerhin: sie sehen klasse aus.

Der Platz reicht hier nicht aus, um wirklich alle meine Kritikpunkte loszuwerden. Aber zwei Punkte sollen hier schon Erwähnung finden. Der erste betrifft die Tastatur. Ich komme nicht darüber hinweg, dass ich keine Tastatur bekomme, auf der die Umlaute enthalten sind. Immer wenn ich das sage, meinen die Oberschlauen, ich müsse doch meinen Finger nur länger auf der Taste halten und dann würde er mir schon die beumlauteten Varianten des Buchstabens anzeigen. Ja, da bin ich sogar ganz alleine drauf gekommen, aber es unterbricht den Schreibfluss. Und selbst wenn ich das Handy drehe, wobei beispielsweise der Taschenrechner eine unzählige Menge neuer Tasten dazubekommt, werden meine Tasten nur breiter. Das kann ich nicht verstehen.

Das scheint aber außer mir niemandem etwas auszumachen. Sie scheinen alle glücklich und freuen sich darüber, wenn ihnen mit völlig unintuitiven Tricks gelegentlich bei der Bedienung weitergeholfen wird. Was mich zum zweiten Punkt bringt, der Erwähnung finden soll. Man kann nämlich mit meinem Smartphone auch Fotos machen. Zum Auslösen muss man dann eine „Knopf“ auf dem Touchscreen drücken. Das ist sehr umständlich, vor allem, wenn man die Kamera in die Breite gedreht hat. Das machen fast alle die ich kenne üblicherweise gegen den Uhrzeigersinn, denn so haben das die meisten auch früher mit ihren (nicht digitalen) Kameras gemacht.

Das bedeutet aber auch, dass die Knöpfe für die Lautstärkeregelung, die in der Normalstellung des Handys links angebracht sind, dann unten liegen. Ich mache nicht besonders viele Fotos, aber ungefähr nach dem dritten Foto fiel mir auf, dass ich als Grobmotoriker immer zwei Hände zum Auslösen brauchte, wenn mir das kostbare Teil nicht aus den Händen fallen sollte. Und dann habe ich mich noch darüber geärgert, warum es nicht einen Auslöseknopf gab. Obwohl, Potential gab es ja, eben nur auf der falschen Seite, nämlich unten links statt oben rechts. Letzteres ließ sich sehr leicht durch eine 180°-Drehung ändern und siehe da, dann hatte sie auch einen Auslöseknopf.

Jetzt habe auch ich ein „weißt du eigentlich, dass du auch …“ und so habe ich schon Vielen das Fotografieren erleichtern können. Aber nicht wirklich, denn der geneigte Nutzer findet sich mit allem ab. Auf jeden Fall solange bis jemand kommt und sagt: „Weißt du eigentlich, dass du auch …“. Hat eigentlich irgendjemand mal das Handbuch gelesen?

User Groups

Fehlt eine User Group? Sind Kontaktdaten falsch? Dann geben Sie uns doch bitte Bescheid.

Bookware
Henkestraße 91, 91052 Erlangen
Telefon: 0 91 31 / 89 03-0
Telefax: 0 91 31 / 89 03-55
E-Mail: redaktion@bookware.de

Java User Groups

Deutschland

JUG Berlin Brandenburg
http://www.jug-bb.de
Kontakt: Herr Ralph Bergmann (orga@jug-bb.de)

Java User Group Saxony
Java User Group Dresden
http://www.jugsaxouy.de
Kontakt: Herr Torsten Rentsch (torsten@jugsaxony.de)
Herr Falk Hartmann (falk@jugsaxony.de)
Herr Kristian Rink (kristian@jugsaxony.de)

rheinjug e.V.
Java User Group Düsseldorf
Heinrich-Heine-Universität Düsseldorf
Universitätsstr. 1, 40225 Düsseldorf
http://www.rheinjug.de
Kontakt: Herr Heiko Sippel (info@rheinjug.de)

ruhrjug
Java User Group Essen
Glaspavillon Uni-Campus
Universitätsstr. 12, 45127 Essen
http://www.ruhrjug.de
Kontakt: Herr Heiko Sippel (heiko.sippel@ruhrjug.de)

JUGF
Java User Group Frankfurt
http://www.jugf.de
Kontakt: Herr Alexander Culum
(javausergroupfrankfurt@googlemail.com)

JUG Deutschland e.V.
Java User Group Deutschland e.V.
c/o asc-Dienstleistungs GmbH
Ehrengard-Schramm-Weg 11, 37085 Göttingen
http://www.java.de (office@java.de)

JUG Hamburg
Java User Group Hamburg
http://www.jughh.org

JUG Karlsruhe
Java User Group Karlsruhe
Universität Karlsruhe, Gebäude 50.34
Am Fasanengarten 4, 76131 Karlsruhe
http://jug-karlsruhe.de
jug-karlsruhe@gmail.com

JUGC
Java User Group Köln
http://www.jugcologne.org
Kontakt: Herr Michael Hüttermann
(michael@huettermann.net)

jugm
Java User Group München
Jupiterweg 8, 85586 Poing
http://www.jugm.de
Kontakt: Herr Andreas Haug (ah@jugm.de)

JUG Münster
Java User Group für Münster und das Münsterland
http://www.jug-muenster.de
Kontakt: Herr Thomas Kruse (tkjugi@sforce.org)

JUG MeNue
Java User Group der Metropolregion Nürnberg
c/o MATHEMA Software GmbH
Henkestraße 91, 91052 Erlangen
http://www.jug-n.de
Kontakt: Frau Alexandra Specht
(alexandra.specht@jug-n.de)

JUG Ostfalen
Java User Group Ostfalen
(Braunschweig, Wolfsburg, Hannover)
Siekstraße 4, 38444 Wolfsburg
http://www.jug-ostfalen.de
Kontakt: Uwe Sauerbrei (info@jug-ostfalen.de)

JUGS e.V.
Java User Group Stuttgart e.V.
c/o Dr. Michael Paus
Schönaicherstraße 3, 70597 Stuttgart
http://www.jugs.org
Kontakt: Herr Dr. Micheal Paus (mp@jugs.org)
Herr Hagen Stanek (hs@jugs.org)

Schweiz

JUGS
Java User Group Switzerland
Postfach 2322, 8033 Zürich
http://www.jugs.ch (info@jugs.ch)
Kontakt: Frau Ursula Burri

.Net User Groups

Deutschland

.NET User Group OWL
http://www.gedoplan.de/cms/gedoplan/ak/ms_net
℅ GEDOPLAN GmbH
Stieghorster Str. 60, 33605 Bielefeld

.Net User Group Bonn
.NET User Group "Bonn-to-Code.Net"
Langwartweg 101, 53129 Bonn
http://www.bonn-to-code.net (mail@bonn-to-code.net)
Kontakt: Herr Roland Weigelt

Die Dodnedder
.NET User Group Franken
http://www.dodnedder.de
Kontakt: Herr Bernd Hengelein
Herr Thomas Müller, Udo Neßhöver
(dodned@googlemail.com)

.net Usergroup Frankfurt
c/o Thomas Sohnrey, Agile IService
Mendelssohnstrasse 80, 60325 Frankfurt
http://www.dotnet-ug-frankfurt.de
Kontakt: Herr Thomas 'Teddy' Sohnrey
(thomas.sohnrey@gmx.de)

.NET DGH
.NET Developers Group Hannover
Landwehrstraße 85, 30519 Hannover
http://www.dotnet-hannover.de
Kontakt: Herr Friedhelm Drecktrah
(friedhelm@drecktrah.de)

INdotNET
Ingolstädter .NET Developers Group
http://www.indot.net
Kontakt: Herr Gregor Biswanger
(gregor.biswanger@web-enliven.de)

DNUG-Köln
DotNetUserGroup Köln
Goldammerweg 325, 50829 Köln
http://www.dnug-koeln.de
Kontakt: Herr Albert Weinert (info@der-albert.com)

.Net User Group Leipzig
Brockhausstraße 26, 04229 Leipzig
http://www.dotnet-leipzig.de
Kontakt: Herr Alexander Groß (agross@dotnet-leipzig.de)
Herr Torsten Weber (tweber@dotnet-leipzig.de)

.NET Developers Group München
http://www.munichdot.net
Kontakt: Hardy Erlinger (hardy.erlinger@hotmail.com)

.NET User Group Oldenburg
c/o Hilmar Bunjes und Yvette Teiken
Sachsenstr. 24, 26121 Oldenburg
http://www.dotnet-oldenburg.de
Kontakt: Herr Hilmar Bunjes
(hilmar.bunjes@dotnet-oldenburg.de)
Yvette Teiken (yvette.teiken@dotnet-oldenburg.de)

.NET User Group Paderborn
c/o Net at Work Netzwerksysteme GmbH,
Am Hoppenhof 32, 33104 Paderborn
http://www.dotnet-paderborn.de
(raacke@dotnet-paderborn.de)
Kontakt: Herr Mathias Raacke

.Net Developers Group Stuttgart
Tieto Deutschland GmbH
Mittlerer Pfad 2, 70499 Stuttgart
http://www.devgroup-stuttgart.de
(GroupLeader@devgroup-stuttgart.de)
Kontakt: Frau Catrin Busley

.net Developer-Group Ulm
c/o artiso solutions GmbH
Oberer Wiesenweg 25, 89134 Blaustein
http://www.dotnet-ulm.de
Kontakt: Herr Thomas Schissler (tschissler@artiso.com)

Österreich

.NET Usergroup Rheintal
c/o Computer Studio Kogoj
Josefgasse 11, 6800 Feldkirch
http://usergroups.at/blogs/dotnetusergrouprheintal/default.aspx
Kontakt: Herr Thomas Kogoj (thomas@kogoj.com)

.NET User Group Austria
c/o Global Knowledge Network GmbH,
Gutheil Schoder Gasse 7a, 1101 Wien
http://usergroups.at/blogs/dotnetusergroupaustria/default.aspx
Kontakt: Herr Christian Nagel (ug@christiannagel.com)

Metropolregion Nürnberg
JUG

Die Java User Group
Metropolregion Nürnberg
trifft sich regelmäßig
einmal im Monat.

Thema und Ort werden über
www.jug-n.de
bekannt gegeben.

Weitere Informationen
finden Sie unter:
www.jug-n.de

MATHEMA

meet the experts of enterprise infrastructure

Web-Anwendungen mit JavaServer Faces (JSF)

Komponentenbasierte Web-Entwicklung mit JavaServer Faces (JSF),
12. – 14. März 2012, 30. Mai – 1. Juni 2012,
1.180,– € (zzgl. 19 % MwSt.)

JEE Anwendungsentwicklung

Entwicklung moderner, skalierbarer Anwendungen mit der Java Enterprise Edition (JEE),
19. – 23. März, 7. – 11. Mai 2012,
2.095,– € (zzgl. 19 % MwSt.)

Programmierung mit Java

Einführung in die Java-Technologie und die Programmiersprache Java, 21. – 25. Mai 2012,
20. – 24. August 2012, 1.645,– € (zzgl. 19 % MwSt.)

Design Patterns und ihre Anwendung

Überblick über die wichtigsten Entwurfsmuster und deren Konzepte,
22. – 23. März 2012, 10. – 11. Mai 2012,
835,– € (zzgl. 19 % MwSt.)

Fortgeschrittenes Programmieren mit Java

Ausgewählte Pakete der Java Standard Edition (J2SE)
29. – 31. Mai 2012,
1.180,– € (zzgl. 19 % MwSt.)

Lesen bildet. **Training macht fit.**

MATHEMA Software GmbH | Telefon: 09131 / 89 03-0 | Internet: www.mathema.de
Henkestraße 91, 91052 Erlangen | Telefax: 09131 / 89 03-55 | E-Mail: info@mathema.de

MATHEMA

Software-Entwickler (m/w)
Software-Architekt (m/w)

Arbeiten Sie gerne selbstständig, motiviert und im Team? Haben Sie gesunden Ehrgeiz und Lust, Verantwortung zu übernehmen?

Wir bieten Ihnen erstklassigen Wissensaustausch, ein tolles Umfeld, spannende Projekte in den unterschiedlichsten Branchen und Bereichen sowie herausfordernde Produktentwicklung.

Wenn Sie ihr Know-how gerne auch als Trainer oder Coach weitergeben möchten, Sie über Berufserfahrung mit verteilten Systemen verfügen und Ihnen Komponenten- und Objektorientierung im .Net- oder JEE-Umfeld vertraut sind, dann lernen Sie uns doch kennen.

Wir freuen uns auf Ihre Bewerbung!

MATHEMA Software GmbH | Telefon: 09131 / 89 03-0 | Internet: www.mathema.de
Henkestraße 91, 91052 Erlangen | Telefax: 09131 / 89 03-55 | E-Mail: info@mathema.de

Das Allerletzte

```
//
// Dear maintainer:
//
// Once you are done trying to 'optimize' this routine,
// and have realized what a terrible mistake that was,
// please increment the following counter as a warning
// to the next guy:
//
// total_hours_wasted_here = 25
//
```

Dies ist kein Scherz!
Dieser Kommentar wurde tatsächlich
in der freien Wildbahn angetroffen.

Ist Ihnen auch schon einmal ein Exemplar dieser
Gattung über den Weg gelaufen?
Dann scheuen Sie sich bitte nicht, uns das mitzuteilen.

Der nächste KaffeeKlatsch erscheint im März.

Herbstcampus

Wissenstransfer par excellence

3.– 6. September 2012

in Nürnberg