KaffeeKlatsch

Das Magazin rund um Software-Entwicklung

ISSN 1865-682X

12/2013

Jahrgang 6

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

Mach es nicht selbst

Es gibt da ein neues Buch: *Weniger schlecht programmieren*. Das macht neugierig. Ist ja auch sehr zeitgeistig, passt es doch sehr schön in das *Clean-Code*-Bemühen, den Code besser zu machen. Das Buch habe ich noch nicht gelesen, aber ein Kapitel kann man online lesen. Kapitel 19: *Mach es nicht selbst*.

In diesem Kapitel [1] geben die Autoren Kathrin Passig und Johannes Jander den Ratschlag, erst einmal nachzusehen, ob es das, was man da gerade programmieren will, nicht schon in der Standardbibliothek gibt. Und wenn es das Gesuchte dort nicht gibt, dann soll man im Internet nach einer geeigneten Lösung suchen, in der Annahme, dass schon jemand anderes dieses Problem hatte.

Das scheint grundsätzlich eine gute Idee zu sein, stellt sich bei genauerer Betrachtung aber gar nicht so leicht dar. Da die Lern- und Arbeitsweisen bei vielen Entwicklern sehr unterschiedlich sind, ist es nicht jedem zuzumuten, sich einen Überblick über die Standardbibliotheken anzueignen, indem man deren Dokumentation vollständig liest – wenngleich es die einzig verlässliche Methode zu sein scheint (ein einigermaßen funktionierendes Gedächtnis vorausgesetzt).

Trotzdem gibt es ja immer wieder etwas, was es dann doch nicht in der Standardbibliothek gibt. Oder nicht ganz so wie man es braucht. Also soll man den Autoren folgend den schier unbegrenzten Fundus im Internet durchforsten. Aber auch das hat seine Tücken. Entweder das Gefundene ist schlecht oder nicht ausreichend dokumentiert, glänzt mit hanebüchenen Abhängigkeiten, die mit der eigentlichen Sache nichts zu tun haben oder steht unter einer Lizenz, die mit den Projektzielen nicht vereinbar ist.

Letzteres gestaltet sich besonders schwierig, wenn zwar die grundsätzlichen Bedingungen klar sind, aber die Rahmenbedingungen dafür nicht erfüllbar sind. Wer schon einmal versucht hat ein Bild von Wikimedia in eigenen Unterlagen zu nutzen, weiß wovon die Rede ist: Das Bild verlangt etwa nach einer Namensnennung, aber nirgends ist ein Name zu finden, den man nennen könnte.

Das Kapitel zählt dann noch auf, was man alles auf keinen Fall neu machen soll, so zum Beispiel ein Datenformat wie XML oder JSON zu entwickeln, nur weil man sich nicht damit auseinandersetzen möchte. Allerdings muss man dazu sagen, dass es JSON wohl nicht gäbe, wenn Douglas Crockford – warum auch immer – mit XML zufrieden gewesen wäre. Nur die Unzufriedenheit brachte ihn auf die geniale Idee, ein Datenformat zu definieren, dass nicht durch einen aufwändigen XML-Parser ausgewertet werden muss, sondern als Programmfragment verstanden werden kann und so von JavaScript direkt ohne fremde Hilfe interpretiert werden und als Datum verarbeitet werden kann. Wie schon gesagt: genial.

Was bleibt ist also wie immer das Abwägen, ob sich eine Eigenentwicklung lohnt oder ob man sich in eine Abhängigkeit zu wem auch immer begibt. Und wenn man seine Sache gut gemacht hat, kann man es ja vielleicht auch der Welt übergeben, damit sich ein anderer die Arbeit sparen kann.

In diesem Sinne
Ihr Michael Wiedeking
Herausgeber

Referenzen

[1] Passig, Kathrin; Jander, Johannes *Weniger schlecht programmieren*
O'Reilly, Dezember 2013
http://www.oreilly.de/catalog/wenschleprogger
speziell: http://www.oreilly.de/catalog/wenschleprogger/chapter/ch19.pdf

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 Natalia Wilhelm via E-Mail unter natalia.wilhelm@bookware.de oder telefonisch unter 09131/8903-16.

Copyright

Impressum

KaffeeKlatsch Jahrgang 6, Nummer 12, Dezember 2013
ISSN 1865-682X
Bookware – eine Initiative der
MATHEMA Verwaltungs- und Service-Gesellschaft mbH

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

Artikel

Kolumnen

Wenn Murphy mal wieder Recht behält

von Marc Spanagel

Murphy erfuhr es schon früh: „Wenn es mehrere Möglichkeiten gibt, eine Aufgabe zu erledigen, und eine davon in einer Katastrophe endet oder sonstwie unerwünschte Konsequenzen nach sich zieht, dann wird es jemand genau so machen."

Dies gilt insbesondere bei der Übergabe eines, oder schlimmer, mehrerer *SQL*-Skripte in die Produktion. In diesem Artikel soll gezeigt werden, wie man diese Gefahren minimieren kann.

Nicht nur Spinnen bauen Netze

von Frank Goraus

Jeder fängt mal klein an. Und so kann einen die Vielfalt an Technologien in der Web-Welt förmlich erschlagen. Hinzu kommt noch, dass man bei Web-Anwendungen teils anders an Probleme herangehen muss als dies bei *Desktop-/Rich-Client*-Anwendungen der Fall ist. Im Folgenden soll es um *Sessions* und die *Servlet*-Filter gehen. Am Ende gibt es eine Beispielanwendung, die selbst implementiert werden kann und anhand derer aufgezeigt wird, wie man beide Techniken einbinden kann.

Apropos Wächter

von Michael Wiedeking

Das Prinzip des im letzten Vergnügen vorgestellten Wächters kann natürlich nach Belieben erweitert werden. Mein persönlicher, ungeschlagener Favorit ist dabei aber der Einsatz in einer verketteten Liste.

Leserbriefe

Leserbrief
von Marco Missfeldt

bezogen auf

Vom Duke zum Droid
Lehren eines Android-Anfängers
von Andreas Heiduk
KaffeeKlatsch 2013/11

Hallo liebes KaffeeKlatsch-Team,

als Java-Entwickler, der sowohl für Desktop-Swing als auch für Android entwickelt hat, finde ich Euren Artikel *Vom Duke zum Droid* nicht so gelungen. Da werden Äpfel mit Birnen verglichen. Swing ist eine klassische UI-Bibliothek, während Android ein komplettes Anwendungs-Framework inklusive Workflow-Steuerung mitbringt. Und was den Vergleich mit den Intents und Objektreferenzen betrifft: Ein Swing-Entwickler wird nicht gezwungen Objektreferenzen umherzureichen. Und natürlich kann man auch bei Android mit Objektreferenzen arbeiten. Da Android-Anwendungen keine Server-seitigen Multiuser-Systeme sind, kann man durchaus Object-Stores auf Basis von Klassen-seitigen Zugriffen implementieren, falls dies im Kontext der Anwendung sinnvoll ist.

Viele Grüße
Marco Missfeldt aus Kiel

Hallo Herr Missfeldt,

Android als Ganzes ist zwar mehr als „nur" Swing, aber die UI-spezifischen bzw. -nahen Teile, um die es im Artikel geht, sind sehr wohl „vergleichbar". Anführungszeichen deshalb, weil es nicht um einen besser/schlechter-Vergleich der „Technologien" selbst geht, sondern um „Techniken", die auf der jeweiligen Plattform im Sinne einer Best-Practice einfacher zum Ziel führen als auf der anderen.

So ist es in Swing einfacher, direkt mit den Objektreferenzen zu arbeiten, da man sie im Normalfall schon hat und das Framework selbst keinerlei zusätzliche Indirektionen fordert. Nur bei speziellen Problemstellungen mag die Einführung „zusätzlicher" Indirektionen Vorteile bringen – natürlich mit Konsequenzen, die gegen den Aufwand abgewogen werden müssen.
Dagegen ist es bei Android einfacher, „mit" Indirektionen zu arbeiten. Tut man das nicht, so muss man auch hier im Einzelfall die Konsequenzen kennen und abwägen – im Artikel sind ja einige beschrieben.

Viele Grüße
Andreas Heiduk

Berichtigungen

Berichtigung
von Wolfgang Wolf

bezogen auf

Getaktete Passwörter
von Wolfgang Wolf
KaffeeKlatsch 2013/11

Liebe Leser,

Der o.g. Artikel enthält leider einen Formatierungsfehler. Im Absatz: „Der aufmerksame und mathematisch geübte Leser hat es eh schon bemerkt...", Seite 11 zweiter Absatz von links oben steht 8^{28} (8 hoch 28) und 6^{29} (6 hoch 29). Richtig wäre aber 82^{8} (82 hoch 8) und 62^{9} (62 hoch 9). Die Ziffer zwei steht also fälschlicher Weise im Exponent.

Die Erklärung der Zahlen: Die 82 stetzen sich so zusammen: 26 Kleinbuchstaben + 26 Großbuchstaben + 10 Ziffern + 20 Sonderzeichen. Der Exponent 8 steht für die Anzahl der Zeichen im Passwort. Das gilt natürlich auch für die nächste Zahl: 62^{9}, also 62 Zeichen (ohne Sonderzeichen), dafür eine Stelle mehr im Exponent.

Mit freundlichen Grüßen
Wolfgang Wolf

Wenn Murphy mal wieder Recht behält

Datenbankskripte übergabesicher gestalten

von Marc Spanagel

Murphy erfuhr es schon früh: „Wenn es mehrere Möglichkeiten gibt, eine Aufgabe zu erledigen, und eine davon in einer Katastrophe endet oder sonstwie unerwünschte Konsequenzen nach sich zieht, dann wird es jemand genau so machen.“ [1]
Dies gilt insbesondere bei der Übergabe eines, oder schlimmer, mehrerer *SQL*-Skripte in die Produktion. In diesem Artikel soll gezeigt werden, wie man diese Gefahren minimieren kann.

Rumpelstilzchen ist im Büro des PV erwacht und tanzt im Dreieck: Die Software ging in Produktion und nichts geht mehr – komisch, dabei wurde doch ausgiebig getestet.

Bald stellt sich heraus, dass es an der Datenbank liegt. SQL-Skripte wurden nicht richtig eingespielt. Die anschließende Fehlerbehebung ist zeitraubend, gefährlich, überflüssig und versprüht den Charme des südlichen Endes eines nördlich ziehenden Maultiers.

Woran lag es wirklich? In den Ebenen vor der Produktion hatte es doch wunderbar geklappt!

Meist läuft es doch so ab: Man zieht einen *Branch* und übergibt diesen aufgrund diverser *Bugfixes* und Änderungen in letzter Sekunde mehrmals in die Abnahmeumgebung – dabei purzeln gewöhnlich auch neue *SQL-Statements* heraus, die dann jeweils als Update übergeben werden. Ruckzuck hat man eine Handvoll SQL-Dateien zusammen, die dann in der Produktion nur einmal und auch nur in einer bestimmten Reihenfolge ausgeführt werden dürfen.

Wer glaubt, das stelle nie ein Problem dar, hat entweder selbst die Kontrolle über die produktive Einspielung oder darf sich heuer ganz viel vom Weihnachtsmann wünschen. Grund genug also das Ganze wasserdicht zu machen. Was soll erreicht werden?

1. Bei jeder Übergabe soll nur noch aufs Knöpfchen gedrückt werden müssen und heraus kommt eine einzige SQL-Datei, die eingespielt werden kann, wo man will und jede Datenbank immer in den neuesten Stand versetzt.
2. Jedes Statement soll in einer eigenen Transaktion ausgeführt werden und einen eventuellen Fehler exakt loggen.
3. Das Skript soll Versionen besitzen, die sich in der Datenbank widerspiegeln.
4. Beim Bau einer Übergabe soll auch gleich ein Marker an die originale SQL-Datei (in die alle Entwickler schreiben – im Folgenden Originaldatei genannt) gesetzt werden, welche dann sofort eingecheckt wird.

Wir erschlagen diese 4 Fliegen mit 2 Klappen respektive *Ant* und *Java* und sprühen noch etwas *PL/SQL* drauf.

Als Erstes müssen wir uns auf einen *Tag* einigen, der die verschiedenen Branchextrakte in der Originaldatei voneinander trennt. Hier bietet sich das *GOTO-Label* (<<...>>) von PL/SQL an und wir geben als Prämisse, dass der Marker <<V0>> an den Anfang der Originaldatei geschrieben werden muss – vor allen Statements. Dieser soll sich dann pro Branchzug erhöhen:

```
--####
--# Originaldatei
--####
<<V0>>
ALTER TABLE FOO MODIFY (BAR VARCHAR2(30
CHAR))
.....
--# 1. Branchübergabe ist erfolgt
<<V1>>
--# 2. Branchübergabe ist erfolgt
<<V2>>
```

Machen wir uns an die Arbeit und schustern uns eine Java-Datei zusammen.

Als Argumente benötigt die Methode *main* die Originaldatei aus dem Branch zum Auslesen und einen Namen (z. B. den Branchnamen) für die Ausgabedatei.

Als Erstes müssen wir lästige Arbeit verrichten und nachsehen, wie die letzte Versionsnummer (die Nummer im Label) lautet und ob bereits ein Abschlusslabel am Ende der Datei existiert oder nicht. Denn es kann ja sein, dass dies vergessen wurde bzw. dass der automatische *Commit* nicht geklappt hat.

```
public static void main(String... args)
throws IOException {
  if (args.length != 2) {
    System.out.println(
      "Bitte zu importierende Datei und  Branchnamen
      angeben!"
    );
    return;
  }
  final String originalDatei = args[0];
  final String branchName = args[1];
  int versionNr = 0;
  boolean isDateiendeGetaggt = true;
  BufferedReader reader = new BufferedReader(
    new FileReader(originalDatei)
  );

  // Ermitteln des aktuellen Stands:
  String zeile = "";
  String letzteZeile = "";
  while((zeile=reader.readLine())!= null) {
    if (zeile.startsWith("<<")) {
      versionNr = Integer.valueOf(
        zeile.replace("<<V", "").replace(">>", "")
      );
    }
    letzteZeile = zeile.trim().equals("") ? letzteZeile : zeile;
  }

  if (!letzteZeile.startsWith("<<")) {
    versionNr ++;
    isDateiendeGetaggt = false;
  }
  reader.close();
  reader = new BufferedReader(
    new FileReader(originalDatei)
  );

  // Hier haben wir die aktuelle Versionnummer ermittelt
  und erzeugen die Übergabedatei:

  String uebergabeDatei = new File(
    originalDatei, branchName + "_V_" + (versionNr) + ".plsql"
  );
  BufferedWriter writer = new BufferedWriter(
    new FileWriter(uebergabeDatei)
  );
```

Nun wird es Zeit, etwas in unsere neu erzeugte Übergabedatei zu schreiben.

Zuerst ein paar Deklarationen für PL/SQL, die wir später brauchen, wie ein paar Variablen und eine eigene *Exception*.

```
write(
  "-- ### ERZEUGT AM " + new SimpleDateFormat(
    "dd.MM.yyyy HH:mm:ss"
  ).format(new Date()) + " ### --\n "
);
write("DEFINE V_BRANCH = \"" + branchName + "\"\n");
write("DEFINE V_ZAEHLER = " + versionNr + "\n");
write("SET SERVEROUTPUT ON\n");
write("DECLARE\n");
write("table_notexists_exception EXCEPTION;\n");
write("PRAGMA EXCEPTION_INIT(
  table_notexists_exception, -942);\n"
);
write("dummy varchar2(3 char);\n");
write("p_zaehler number(2,0) := 0;\n");
```

Der kritische Leser mag hier anmerken, dass es doch sauberer wäre die Inhalte von *write* in eine eigene Datei zu schreiben. Stimmt, aber wir bauen hier eine Art Provisorium als Mittel zum Zweck und legen noch keinen Wert auf Architektur.

Doch nun weiter im Text. Da nur Bares Wahres ist, wollen wir die Ausführung der Datei in der Datenbank selbst loggen und nicht in einer Datei, die man wieder nur mühsam anfragen muss. Datenbanken kann man ja so schön über geschützte oder geheime *Adminpages* abfragen.

Also einigen wir uns auf eine Tabelle *LOG* mit den Spalten *zeitstempel* und *info* und eine Tabelle BRANCH mit den Spalten *branchname* und *versionsnr*.

In LOG schreiben wir Beginn und Ende der Ausführung und eventuell auftretende Fehler, in BRANCH die Programmversion (oder den Branchnamen) und die

Versionsnummer des letzten Labels in der Originaldatei (im Beispiel ganz oben also 2 wegen <<V2>>).

Falls die beiden Tabellen noch nicht vorhanden sind, sollen sie angelegt werden.

Als letzte Deklaration wird noch eine parameterlose Prozedur LOG benötigt, die Fehler festhält und so aussehen soll:

```
PROCEDURE log
IS
BEGIN
   EXECUTE IMMEDIATE('INSERT INTO log (
      zeitpunkt, info
   ) VALUES
   (LOCALTIMESTAMP, SUBSTR(
      DBMS_UTILITY.FORMAT_ERROR_STACK, 1, 510
   ) || " : " ||
   SUBSTR(
      DBMS_UTILITY.FORMAT_ERROR_BACKTRACE,1,510))'
   );
   DBMS_OUTPUT.PUT_LINE(
      SUBSTR(DBMS_UTILITY.FORMAT_ERROR_STACK,1,510
   ) || ' : ' ||
   SUBSTR(
      DBMS_UTILITY.FORMAT_ERROR_BACKTRACE,1,510)
   );
END;
```

DBMS_UTILITY gibt uns dabei praktischerweise viele Informationen über die zuletzt aufgetretene Exception. Hier die Fehlerinformation und die Zeilennummer, in der der Fehler aufgetreten ist und man im PL/SQL-Code nur noch "LOG" aufrufen muss, um den letzten Fehler in die LOG-Tabelle zu schreiben. Mit der Verwendung dieses *Packages* bindet man sich natürlich an eine *Oracle*-Datenbank.

Leider muss das Ganze mit EXECUTE IMMEDIATE umhüllt werden. Warum?

Weil uns das die Portabilität des Programms gebietet. Wenn die Tabelle LOG noch nicht existierte, würde der PL/SQL-Code nicht kompilieren, wenn hier ein INSERT INTO LOG stünde.

Mit EXECUTE IMMEDIATE jedoch verschieben wir die Prüfung auf die Laufzeit, verschlimmbessern jedoch auch unseren Code – als Entwickler steht man ja oft vor der Wahl, sich entweder ins Knie oder doch lieber in den Fuß zu schießen und weiss erst hinterher was mehr schmerzt.

Bemühen wir also wieder unseren *Writer* und schreiben die Logik der Tabellenerzeugung. Diese Tabellen LOG und BRANCH werden erzeugt, wenn ein SELECT fehlschlägt, was auch unsere Dummy-Variable erklärt (die wir nur der Syntax wegen benötigen). Mit

select *table_name* INTO *dummy* from *user_tables* WHERE *table_name* = 'LOG'

fragen wir nämlich alle existierenden User-seitigen Tabellen ab, ob unsere Tabelle 'LOG' schon existiert. Im Fall einer Exception wird sie angelegt.

Diese Abfrage können wir direkt ausführen, ohne EXECUTE IMMEDIATE, da die Tabelle *user_tables* immer existiert – natürlich muss man sich vorher die Rechte sichern.

Dass es auch anders geht, sieht man an der Existenzprüfung der Tabelle BRANCH. Diese *Query* führen wir dynamisch aus und fangen die Exception *ORA-942* (*Table or View doesn't exist*) mit unserer selbsterzeugten Exception-Variable *table_notexists_exception* ab – dafür sparen wir uns hier den Dummy.

```
write("PROCEDURE log\n");
write("IS\n");
write("BEGIN\n");
write(
   "EXECUTE IMMEDIATE(
      'INSERT INTO log(zeitpunkt, info) VALUES\n"
);
write(
   "(LOCALTIMESTAMP, SUBSTR(
      DBMS_UTILITY.FORMAT_ERROR_STACK, 1, 510
   ) || " : " || SUBSTR(
      DBMS_UTILITY.FORMAT_ERROR_BACKTRACE,1,510))'
   );\n"
);


write(
   "DBMS_OUTPUT.PUT_LINE(
      SUBSTR(DBMS_UTILITY.FORMAT_ERROR_STACK,1,510
   ) || ' : ' || SUBSTR(
      DBMS_UTILITY.FORMAT_ERROR_BACKTRACE,1,510)
   );\n"
);
write("END;\n");


write("BEGIN\n");
write("BEGIN\n");
write(
"SELECT table_name INTO dummy FROM user_tables
WHERE table_name = 'LOG';\n"
);
write("EXCEPTION\n");
write("WHEN NO_DATA_FOUND THEN\n");
write(
   "EXECUTE IMMEDIATE ('CREATE TABLE LOG (
      zeitpunkt TIMESTAMP,info varchar2(1024 char))'
   );\n"
);
write(
   "EXECUTE IMMEDIATE('INSERT INTO
   log(zeitpunkt, info) VALUES \n"
);
write(
   "(LOCALTIMESTAMP, "LOG-TABELLE existierte nicht
   und musste neu angelegt werden!")');\n"
);
write("END;\n");
```

```
write("BEGIN\n");
write(
   "EXECUTE IMMEDIATE(
      'SELECT versionstand FROM BRANCH WHERE
      branchname="&V_BRANCH"'
   ) INTO p_zaehler;\n"
);
write("EXCEPTION\n");
write("WHEN table_notexists_exception THEN\n");
write(
   "EXECUTE IMMEDIATE (
      'CREATE TABLE BRANCH (branchname
      varchar2(11 char), versionstand number(2,0))'
   );\n"
);
write(
   "EXECUTE IMMEDIATE(
      'INSERT INTO BRANCH(branchname, versionstand)
      VALUES ("&V_BRANCH", 1)'
   );\n"
);
write("WHEN NO_DATA_FOUND THEN\n");
write(
   "EXECUTE IMMEDIATE(
      'INSERT INTO BRANCH(branchname, versionstand)
      VALUES ("&V_BRANCH", 1)'
   );\n"
);
write("commit;\n");
write("END;\n");


write("BEGIN\n");
write(
   "EXECUTE IMMEDIATE(
      'INSERT INTO log(zeitpunkt, info) VALUES \n"
);
write(
   "(LOCALTIMESTAMP, "Beginn Einspielung
   DB-DiffSkript &V_BRANCH")');\n"
);
write("commit;\n");
write(
   "DBMS_OUTPUT.PUT_LINE(
      'Beginn Einspielung DB-DiffSkript &V_BRANCH'
   );\n"
);
```

Dem aufmerksamen Leser ist sicherlich aufgefallen, dass bis jetzt noch keine Versionssicherheit gegeben ist. Diese kommt jetzt ins Spiel! Wir teilen dem Programm erst einmal mit, dass es gleich einen Abgang machen kann, wenn die letzte Versionsnummer, die sich in der Tabelle BRANCH für diese Programmversion befindet, bereits ausgeführt wurde, indem wir das oft verpönte GOTO verwenden:

```
write("IF p_zaehler>=" + versionNr + " THEN\n");
write(
   "DBMS_OUTPUT.PUT_LINE(
      'Sprung auf Label: <<" + versionNr + ">>'
   );\n"
);
write(
   "EXECUTE IMMEDIATE('INSERT INTO
   log(zeitpunkt, info) VALUES \n"
);
write(
   "(LOCALTIMESTAMP, "Sprung auf Label: <<" +
   versionNr + ">>")');\n"
);
write("commit;\n");
write("GOTO V" + versionNr + ";\n");
```

Ähnliches machen wir mit den Zwischenständen, also wenn z. B. *V1* bereits ausgeführt wurde, aber *V2* und *V3* noch nicht, indem wir über alle bisherigen Versionen iterieren. Somit springen wir bei Ausführung genau zu dem Code-Teil, der noch nicht eingespielt wurde.

```
// Interne Logik für Versionssprung
for (int i = versionNr – 1; i >= 0; i--) {
   write("ELSIF p_zaehler>=" + i + " THEN\n");
   write(
      "DBMS_OUTPUT.PUT_LINE(
      'Programmsprung auf Label: <<" + i + ">>'
      );\n"
   );
   write(
   "EXECUTE IMMEDIATE(
      'INSERT INTO log(zeitpunkt, info) VALUES \n"
   );
   write(
      "(LOCALTIMESTAMP, "Programmsprung auf Label:
      <<" + i + ">>")');\n"
   );
   write("commit;\n");
   write("GOTO V" + i + ";\n");
}
write("END IF;\n");
```

Nach dem Hollywood-verdächtigen Intro kann nun endlich das Auslesen der Originaldatei beginnen.

Um wirklich jedem Fehler auf die Spur zu kommen, wird jede SQL-Anweisung mit einer Transaktion umhüllt, und zwar immer nach diesem Schema:

```
BEGIN
   EXECUTE IMMEDIATE('sqlAnweisung');
   commit;
EXCEPTION
   WHEN OTHERS THEN LOG;
END;
```

Erst hier sieht man, wie nützlich das Package DBMS_UTILITY für uns ist, denn der Prozedur LOG müssen nicht mal Parameter übergeben werden, um den Ursprung einer Exception zu erfahren, womit als einziger Parameter die SQL-Anweisung verbleibt.

Das ist dann einfach nur Auslesen nach dem Rezept von Knack und Back und das Ersetzen von einfachen Anführungszeichen (') mit zweien ('').

Welches Rezept genau gebraucht wird, hängt von den „Zutaten“ ab. Im einfachsten Fall ist es nur zeilenweises Auslesen, wenn wirklich jede Anweisung in genau einer Zeile steht.

Tatsächlich gibt es aber vielfältige Möglichkeiten, zum Beispiel mehrere Anweisungen zu einem Block zusammenzufassen, PL/SQL-Skript in die Originaldatei zu schreiben oder ähnliche Spielereien. Dies auszuführen würde jedoch den Rahmen dieses Artikels sprengen.

Was fehlt noch? Ach ja, der Knopf! Ein *Ant-Task*, der zum Beispiel in *Jenkins* angestoßen werden kann, ist hier geradezu prädestiniert.

Als letzte Amtshandlung unseres Java-Programms soll es nun noch das neue Label an das Ende der Originaldatei schreiben, falls es dort noch nicht existiert (deshalb die Variable *isDateiendeGetaggt*, s.o.).

Ant übernimmt dann nur noch die Ausführung des Programms und das *Commit* ins *Repository*, und zwar von der frisch erzeugten Datei als auch der Originaldatei.

```
<cvs command="commit -m 'auto increment label'
${uebergabedatei}" failonerror="true"/>
```

Bei neueren Ant-Versionen ist es nicht mehr möglich eine Datei einzuchecken, bei der über übergeordnete Verzeichnisse navigiert werden muss. Deswegen kann es nötig sein, eine eigene Ant-Datei mit dem *Commit-Target* in das Verzeichnis der einzucheckenden Datei zu stellen, die dann vom *Maintarget* aufgerufen wird.

Ein letzter Vorteil sei zum Schluss noch angepriesen: Unser Programm ist sehr zukunftssicher, denn nichts hält so lange wie ein Provisorium!

Referenzen

[1] Wikipedia *Murphys Gesetz*
http://de.wikipedia.org/wiki/Murphys_Gesetz

Kurzbiografie

Marc Spanagel (marc.spanagel@mathema.de) ist als Senior Consultant für MATHEMA Software GmbH tätig. Sein Schwerpunkt liegt auf der Entwicklung mit JEE und C#. Daneben fasziniert ihn die 3D-Programmierung mit DirectX.

Nicht nur Spinnen bauen Netze

Teil 3 – Web-Entwickung für Java-Entwickler

von Frank Goraus

Jeder fängt mal klein an. Und so kann einen die Vielfalt an Technologien in der Web-Welt förmlich erschlagen. Hinzu kommt noch, dass man bei Web-Anwendungen teils anders an Probleme herangehen muss als dies bei *Desktop-/Rich-Client-*Anwendungen der Fall ist. Im Folgenden soll es um *Sessions* und die *Servlet*-Filter gehen. Am Ende gibt es eine Beispielanwendung, die selbst implementiert werden kann und anhand derer aufgezeigt wird, wie man beide Techniken einbinden kann.

Rückblick

Im ersten Artikel [1] haben wir uns damit beschäftigt, einfach eine simple kleine *Hallo-Welt*-Anwendung aufzusetzen und ihr ein wenig Dynamik einzuhauchen. Wir haben eine *JSP* angelegt und ein Servlet implementiert, welches Formulareingaben auswertet und eine entsprechende Antwortseite generiert.

Im zweiten Teil [2] haben wir uns dann noch mal genauer damit beschäftigt, was wir da alles getan haben und wie diese Teile zusammenhängen und -spielen. Wir haben uns *Request* und *Respone*, Servlets und deren *Lifecycle*, sowie den *Deployment-Descriptor* angeschaut. Letzteren werden wir in diesem Artikel noch ein wenig erweitern.

Denn im vorliegenden Artikel gehen wir einen Schritt weiter und werden uns Sessions und Servlet-Filter vornehmen. Das Ganze implementieren wir anhand einer kleinen Beispielanwendung mit einem *User-Login* und einem „geschützten" Bereich, welcher nur für eingeloggte User erreichbar ist.

Die Sitzung

Bisher haben wir uns nur zustandslose Teile der Java Web-Welt angeschaut. Auch wenn ein Servlet einen Lifecycle besitzt, so wird dieser komplett innerhalb eines Requests abgelaufen. Und darin angelegte Objekte stehen nicht im nächsten Request zur Verfügung. Wir können zwar auf statische Ressourcen zugreifen, welche mit dem Servlet gemanaged werden, diese stehen jedoch allen Benutzern der Servlets zur Verfügung und beinhalten keinerlei nutzerspezifische Daten. (Wobei dies nur eine Frage der Implementierung ist. Sauber wäre es jedoch nicht.)

Die Frage ist also wie wir uns nutzerspezifische Daten über mehrere Requests halten können. Die Lösung heißt Session, zu Deutsch Sitzung. Kommt ein neuer Benutzer auf unsere Anwendung, so wird ihm vom *Application Server* eine eindeutige ID, in Form der *(J)Session-ID* zugeteilt, welche er die ganze Zeit, während er mit der Anwendung agiert, verwendet und so eindeutig identifizierbar bleibt. Löscht er diese, so bekommt er eine neue Session-ID und auch eine neue Sitzung zugeteilt. Je nach unterstützten Browser-Features wird diese ID entweder als Request-Parameter oder als *Cookie* zum und vom *Client* übertragen. Auf der Server-Seite bekommt man davon gar nicht soviel mit, da sich der Application-Server darum kümmert und man einfach nur die zum Request gehörige Session abfragt.

In Abbildung 1 und 2 ist einmal aufgezeigt wie sich die Session-ID ohne und mit Cookies präsentiert.

Anhand dieser Session-ID managed der Application-Server also die verschiedenen Sitzungen einzelner Benutzer der Anwendung. Innerhalb einer solchen Sitzung können vom Server, also der Anwendung, verschiedene Werte abgelegt werden. Die Zuordnung erfolgt wieder, ähnlich wie die Request-Parameter, über *Key-Value*-Paare, wobei die *Values* beliebige Objekte sein können. Diese in der Session abgelegten Variablen nennt man Session-Attribute. Diese werden solange in der Session vorgehalten bis sie entweder wieder entfernt werden oder die Session beendet wird. Dazu später mehr. Solange die Sitzung also aktiv ist, können alle darin abgelegten Daten über mehrere Requests des selben Benutzers hinweg abgerufen oder auch verändert werden.

Klingt ziemlich simpel? Ist es auch. Mit wenigen Zeilen Code haben wir alles was wir brauchen. Mit folgendem Stück Code können wir auf die Session des gerade agierenden Benutzers zugreifen.

```
HttpSession session = request.getSession();
```

Wie man sieht, brauchen wir nur vom aktuell bearbeiteten Request die Sitzung abzufragen. Wir müssen dabei nicht erst irgendwoher die Session-ID ermitteln und mit dieser die Session von irgendeinem Manager aus den gespeicherten Sessions ermitteln. Das erledigt für uns al-

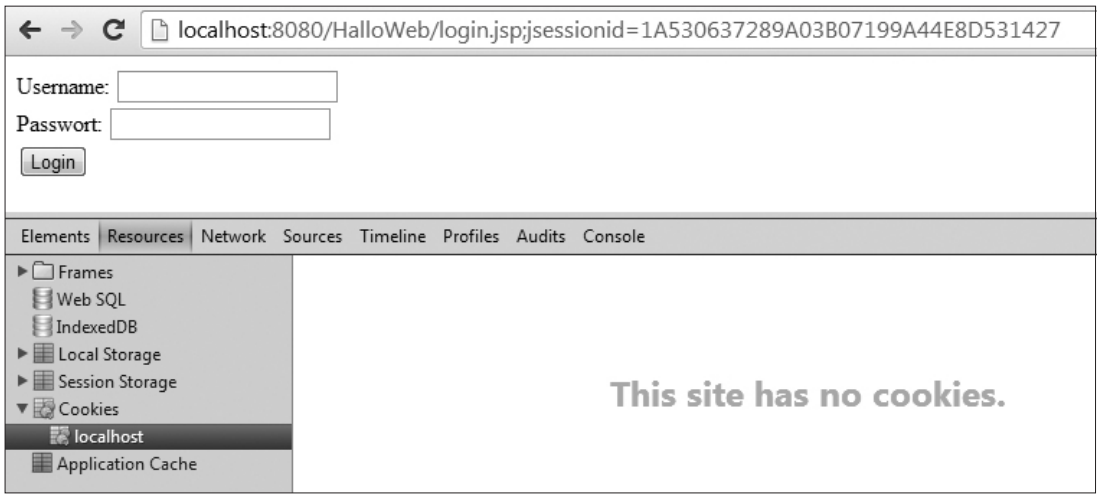


Abbildung 1

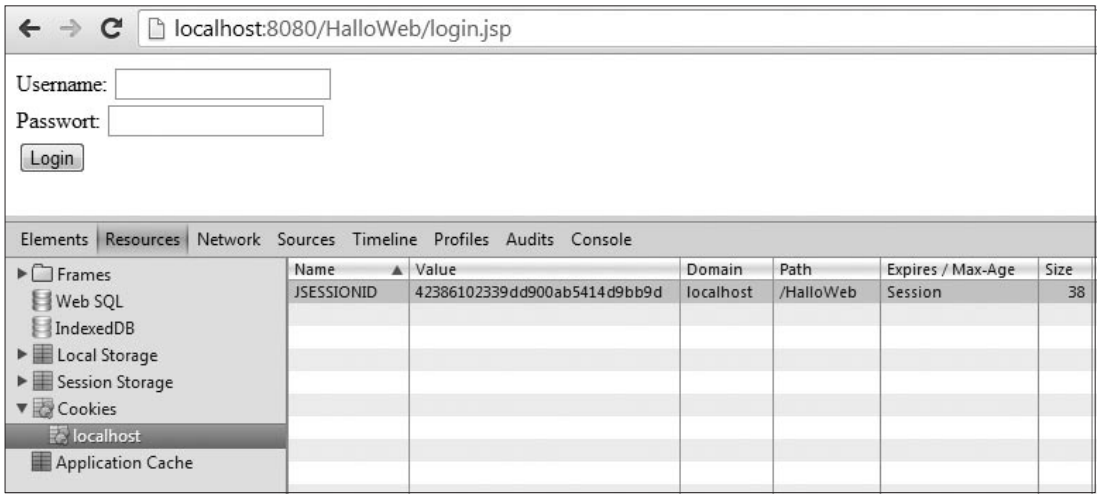


Abbildung 2

les der Application-Server. Als Ergebnis dieser Abfrage bekommen wir ein Objekt vom Typ *javax.servlet.http.HttpSession*. Dieses ist nicht sonderlich komplex, bietet jedoch alles Notwendige. In den meisten Fällen muss man nur Daten aus der Session lesen oder darin ablegen. Dies geht mit folgenden zwei Aufrufen:

```
Object value = session.getAttribute("key");
session.setAttribute("key", value);
```

Wie man sieht wird der *Key* als *String* angegeben, der *Value* ist zunächst untypisiert und wird nur als *Object* zurückgegeben. Das heißt man muss diesen noch auf den richtigen Klassentyp casten und kann so bei unvorhergesehenen Typänderungen in eine *ClassCastException* rennen. Dies sollte innerhalb der eigenen Anwendung jedoch überschaubar und nicht der Fall sein, birgt aber trotzdem ein Risiko.

Möchte man den Wert eines Attributes verändern, so kann man entweder den neuen Wert per *setAttribute()* unter dem selben *Key* ablegen, der alte Wert wird also überschrieben, oder im Falle von *Mutable Objects* (falls unbekannt: Das Gegenteil von *Immutable Objects* [3]) direkt das referenzierte Objekt verändern. Dann muss dieses nicht wieder extra in der Session abgelegt werden, da wir das Objekt selbst verändert haben.

Eine Sache sollten wir zur Session noch kurz beleuchten, und das ist ihre Lebenszeit. Oben hatte ich schon kurz angedeutet, dass eine Session auch beendet werden kann. Dies kann passieren, wenn der Benutzer bereits eine Weile inaktiv war, sprich *x* Minuten lang keine neuen Requests an den Server gesendet hat. Das heißt, dass er entweder die Anwendung bei sich geschlossen hat, oder (bedauerlicherweise der Fall mit dem man sich als Entwickler herumschlagen muss) dass er die Anwendung in den Hintergrund verschoben hat und nur einfach aktuell nicht damit arbeitet. In beiden Fällen schlägt das sogenannte *Session-Timeout* zu. Da der Anwendung nicht unendlich Speicher zur Verfügung steht, kann sie

also nicht beliebig viele Sessions mit allen daran hängenden Daten vorhalten, sondern räumt nach einer konfigurierbaren Zeit *x* ab dem letzten Request eines Benutzers die Session und deren belegten Speicherplatz wieder frei. Üblich sind hier Werte von 10, 15 oder 30 Minuten, seltener sogar 60 Minuten. Erfolgt in dieser Zeit ein neuer Request, so wird das Timeout der Session wieder auf die volle Zeit zurückgesetzt. Das Session-Timeout kann man, wie Vieles, in der *web.xml* seiner Anwendung definieren: (Wert in Minuten)

```
<session-config>
  <session-timeout>60</session-timeout>
</session-config>
```

Leider ist dies, je nach verwendetem Application-Server, jedoch nicht das Timeout, welches effektiv verwendet wird. Und man sollte diesen Wert also auch in der entsprechenden Konfiguration des Servers analog konfigurieren.

Was passiert also wenn der User länger als das konfigurierte Timeout nicht mit der Anwendung interagiert hat? Am Client hat er immer noch seine alte Session-ID, die er beim nächsten Request an den Server schickt. Dieser findet aber zu dieser ID keine Session mehr, also generiert er eine neue. Danach liegt es an der programmierten Anwendung selbst, wie sich diese verhält. Will man als Entwickler auf Werte in der Session zugreifen, welche nicht vorhanden sind, aber eigentlich vorhanden sein müssten, da der User in einem Bereich agiert, wo er nur mit diesen Daten hinkommt, so muss man mit diesem Fall umgehen können, ansonsten hagelt es vielleicht *NullPointerExceptions* und der User wird mit Fehlermeldungen oder weißen Seiten begrüßt. Man braucht also eine *Fallback/Failover*-Strategie. Im Falle eines Bereiches, den nur eingeloggte Benutzer sehen dürfen, würde es also Sinn machen bei jedem Request zu prüfen ob der User eingeloggt ist und diesen Bereich betreten darf. Und wenn nicht, ihn auf die Login-Seite zu verweisen. Da dies aus Benutzersicht aber auch unschön ist, gibt es auch Tricks um die Session am Leben zu erhalten, obwohl der User keine Interaktion tätigt. Zum Beispiel kann man zusätzlich zum *Failover* (nicht stattdessen) vom Client aus in einem definierten Intervall so etwas wie einen *Ping* vom Client an den Server schicken. Damit kommt regelmäßig vor dem Timeout ein Request an und die Session wird verlängert. Dies hilft jedoch nur in den Fall, wenn der User die Anwendung noch offen hat. Schließt er sie zwischenzeitlich und ruft nach dem Öffnen wieder einen Teil des geschützten Bereiches auf, so rennt er trotzdem in oben beschriebene Situation mit einer frischen Sitzung. Deswegen wird die *Fallback*-Strategie zusätzlich noch benötigt.

Neben dem automatischen Beenden einer Session durch ein Timeout gibt es noch die Möglichkeit, dass die Session „gewaltsam" vom Server beendet wird. Oben beschriebene *HttpSession* bietet dazu die Methode *invalidate()* an. Dies weist den Application-Server an die Session zu löschen und der User bekommt so mit dem nächsten Request eine frische Session zugeteilt. Dies macht zum Beispiel dann Sinn, wenn der User sich aus der Anwendung ausloggt. Damit werden einige Daten in der Session überflüssig und müssen entweder manuell aufgeräumt werden, um fehlerhafte Zustände in der Anwendung zu vermeiden, oder man beendet die Session einfach um so die Daten automatisch aufräumen zu lassen. Was deutlich der bessere Weg ist. Im Zweifelsfall kann man danach gleich wieder den Request nach seiner Session befragen. Der Server legt dann direkt eine neue an, mit der man weiterarbeiten und zum Beispiel wichtige zwischengespeicherte Werte aus der alten Session in die neue übernehmen kann.

Der Servlet-Filter

Der Servlet-Filter kann als eine Art digitaler Türsteher betrachtet werden. Wobei dies seiner Aufgabe nicht ganz gerecht wird. Viel mehr ist es einfach ein Stück Code, der vor dem Aufruf bestimmter Ressourcen ausgeführt wird. In den meisten Fällen beschränkt sich dies jedoch auf Zugriffsschutz oder Logging. In der JEE-Welt oder anderen *Web-Frameworks* gibt es ein ähnliches Konstrukt in Form der *Interceptoren*, welche jedoch wesentlich komplexer sind und auf verschiedene Zustände der betroffenen Objekte reagieren können. Der Servlet-Filter kennt nur einen Zustand: Den Aufruf einer Ressource. Für welche Ressourcen er zuständig ist, kann man in der *web.xml* anhand eines *URL-Patterns* definieren:

```
<filter>
  <filter-name>LoginFilter</filter-name>
  <filter-class>de.mathema.kk.teil3.LoginFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>LoginFilter</filter-name>
  <url-pattern>/userarea/*</url-pattern>
</filter-mapping>
```

Wie man in dem Beispiel sieht, sind auch *Wildcards* zulässig. Der konfigurierte Filter fühlt sich für alle Ressourcen unterhalb des Ordners *userarea* zuständig.

Wie bereits bei den Servlets muss man außerdem noch eine Verknüpfung zwischen *filter-name* und der ef-

fektiv dahinter stehenden Klasse herstellen. Diese Klasse leitet sich von *javax.servlet.Filter* ab. Im folgenden Listing sieht man einen frisch angelegten Filter, der erstmal keinerlei Funktion übernimmt und einfach nur da ist.

```
public class LoginFilter implements Filter {
  @Override
  public void doFilter(ServletRequest request,
  ServletResponse response, FilterChain chain)
  throws IOException, ServletException {
    //pass on to next filter
    chain.doFilter(request, response);
  }

  @Override
  public void init(FilterConfig filterConfig)
  throws ServletException {
  }

  @Override
  public void destroy() {
  }
}
```

Auch hier gibt es wieder Analogien zum Servlet, da man eine *init()*- und *destroy()*-Methode zur Implementierung gestellt bekommt. Und analog zu den *doPost()*-, *doGet()*-, … Methoden gibt es *doFilter()*, in welcher wir unsere eigentliche Logik implementieren können. Als Parameter bekommen wir den *(Servlet)Request* und die *(Servlet)-Response* hereingereicht und haben so zum Beispiel auch Zugriff auf die Session des Benutzers. Zusätzlich bekommen wir aber auch noch eine Instanz der Klasse *javax.servlet.FilterChain* hereingereicht. Dies ist besonders wichtig, da unser Filter vielleicht nicht der einzige ist der sich um die Ressource kümmert. Stattdessen gibt es eine Kette von Filtern die nacheinander abgearbeitet werden. Die Reihenfolge der Filter wird hier dadurch bestimmt, in welcher Reihenfolge sie in der *web.xml* eingetragen wurden. Damit diese Kette nicht unterbrochen wird, müssen wir also nach getaner Arbeit unseres Filters das Zepter an den nächsten weitergeben. Dies geschieht in dem wir auf dieser Instanz von *FilterChain* wieder die *doFilter()*-Methode aufrufen und ihr Request und Response übergeben. Somit werden also auch unsere Veränderungen an der Session weitergereicht und können eventuell im nächsten Filter weiterverwendet werden.

Unsere Beispielanwendung

Für eine kleine Demonstration dieser Komponenten bauen wir uns nun eine kleine Beispielanwendung. Diese besteht aus einem offenen, für jeden sichtbaren Bereich und einem geschützten Bereich, welcher nur mit erfolgreichem Login betretbar ist. Ob und welcher User eingeloggt ist, wird in der Session vorgehalten. Den Zugriff auf den geschützten Bereich sichert ein Servlet-Filter, der den User aus der Session prüft. Ist kein User eingeloggt, so wird der Benutzer auf den offenen Bereich in Form einer Login-Seite geleitet. Im geschützten Bereich befindet sich dann eine Seite, welche den eingeloggten User begrüßt. (Wer mag, kann an dieser Stelle pausieren und versuchen die beschriebene Anwendung selbst zu implementieren.)

Beginnen wir zunächst mit der Login-Seite (*login.jsp*). Diese dient für unser Beispiel als Einstiegspunkt und besteht erstmal nur aus zwei Eingabefeldern und einem *Button* zum Absenden der Daten. (Gemäß dem Motto *Form follows Function* wird hier auf hübsches Design verzichtet.)

```
<%@page contentType="text/html"
  pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC
  "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd"
>
<html>
  <head>
    <meta http-equiv="Content-Type"
      content="text/html; charset=UTF-8">
    <title>Login</title>
  </head>
  <body>
    <form action="login" method="post">
      Username: <input type="text"
        name="login" /><br/>
      Passwort: <input type=
        "password" name="password" /><br/>
      <input type="submit" value="Login">
    </form>
  </body>
</html>
```

Dieses Formular wird dann an das *LoginServlet* gegeben, in dem die Login-Daten geprüft werden. Sollte der Login erfolgreich sein, so wird er in den geschützten Bereich, der sich im *WebContent* unter dem Ordner *userarea* befindet, geleitet, und dort speziell auf eine Einstiegsseite namens *welcome.jsp*.

(In den Java-Listings wird aus Platzgründen auf die Angaben von *package* und *import* verzichtet):

```
public class LoginServlet extends HttpServlet {

  @Override
  protected void doPost(
    HttpServletRequest request,
    HttpServletResponse response
  )
  throws ServletException, IOException {
```

```
        HttpSession session = request.getSession();

        //lookup if we are already logged in
        User user = (User)session.getAttribute("user");
        if (user == null) {
          //else, do a login
          String login = request.getParameter("login");
          String password = request.getParameter("password");
          user = UserManager.getUser(login, password);
          //store user in session (might still be null)
          session.setAttribute("user", user);
        }

        if (user != null) {
          //if he is logged in now, send him to user area
          response.sendRedirect("userarea/welcome.jsp");
        } else {
          //else, send him to public area
          response.sendRedirect("login.jsp");
        }
      }
    }
```

In obigem Code sieht man, dass ich bereits auf die Klassen *User* und *UserManager* zurückgreife. Erstere gilt als Repräsentation eines Users, welche dann in der Session gespeichert wird. So ist der Kontext klar, dass es sich um einen User handelt, und man kann weitere Methoden verwenden, die dieses Objekt zur Verfügung stellt, ohne erst zum Beispiel *Strings* zu parsen und Instanzen von Objekten zu erzeugen, an Stellen wo es eigentlich nicht mehr notwendig sein sollte. (Man stelle sich einfach als gedankliches Gegenbeispiel vor, dass man in der Session nur *Strings* speichern könne und müsste somit aus diesen wieder verwert- und verarbeitbare Informationen machen.)

```
public class User {

  private final String login;
  private final String name;
  private final String password;

  public User(
    String login, String name, String password
  ) {
    super();
    this.login = login;
    this.name = name;
    this.password = password;
  }

  public String getLogin() {
    return login;
  }

  public String getName() {
    return name;
  }

  public String getPassword() {
    return password;
  }
}
```

Wie man sieht bietet die Klasse neben dem Login und dem Passwort, welche wir schon im *LoginServlet* als Parameter aus der *login.jsp* gelesen haben, zusätzlich noch den Namen des Users an. Diesen brauchen wir also nicht extra in der Session abzulegen, da er einfach Teil des gespeicherten Objektes ist.

Der *UserManager* soll exemplarisch für eine Schnittstelle stehen, über welche wir an Informationen zu den Usern gelangen. In diesem simplen Beispiel werden einige Demonstrations-User lokal angelegt und verwaltet. In richtigen Anwendungen würde hier stattdessen eine Datenbank oder ein entferntes System angesprochen werden. (Und niemals würden Passwörter so im Klartext hinterlegt werden ...)

```
public class UserManager {

  private final static HashSet<User> knownUsers =
    new HashSet<User>();

  private static void initUsers() {
    knownUsers.clear();
    knownUsers.add(
      new User("hansi27", "Hans Hansen", "base1980")
    );
    knownUsers.add(
      new User("peter", "Peter Peterson", "p3t3")
    );
    knownUsers.add(
      new User("admin", "BOFH", "password")
    );
  }

  public static User getUser(
    String login, String password
  ) {
    //ensure filled list
    if (knownUsers.isEmpty()) {
      initUsers();
    }
    //lookup user
    for (User user : knownUsers) {
      if (user.getLogin().equals(login)
      && user.getPassword().equals(password)) {
        return user;
      }
    }
    return null;
  }
}
```

Die Methode *getUser()* haben wir bereits im *LoginServlet* verwendet um den einloggten User zu ermitteln. Auch

hier ist die Logik sehr vereinfacht, da einfach Null zurückgegeben wird, sollte Login und/oder Passwort nicht korrekt sein. In echten Anwendungen würde man hier mindestens noch etwas Fehler-*handling* hinzufügen um den User darüber zu informieren, was gerade schiefgegangen ist. Sollte der Login jedoch geklappt haben, so leitet ihn das *LoginServlet* auf die *welcome.jsp*.

```
<%@page import="de.mathema.kk.teil3.User"%>
<%@page contentType=
  "text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC
  "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd"
>
<html>
  <head>
    <meta http-equiv="Content-Type"
      content="text/html; charset=UTF-8">
    <title>Greetings</title>
  </head>
  <body>
    <h1>Hello <%=
      ((User)session.getAttribute("user")).getName() %></h1>
  </body>
</html>
```

In dieser soll einmal aufgezeigt werden, dass man natürlich auch aus *JSP*s heraus auf die Session zugreifen kann. (Wir erinnern uns: Eine *JSP* wird vom *JSP-Compiler* auch nur in ein Servlet umgewandelt.) Hier ziehen wir dann Nutzen aus der zusätzlichen Möglichkeit des User-Objekts, sich dessen Namen ausgeben zu lassen.

Jetzt mag vielleicht aufgefallen sein, dass auf dieser ganzen Programmablaufstrecke nirgendwo der Filter erwähnt wurde. Das liegt an seinem Charakter, dass der Filter sich einfach unaufdringlich (*unobstrusive*) dazwischen klemmt und den ganzen Ablauf erstmal nicht weiter stört. Doch in unserem Fall wollen wir einen Zugriffsschutz erwirken, so dass die *welcome.jsp* nicht aufgerufen werden kann, wenn der User nicht eingeloggt ist. Sprich wenn kein User-Objekt in der Session liegt.

```
public class LoginFilter implements Filter {

  @Override
  public void doFilter(
    ServletRequest request,
    ServletResponse response,
    FilterChain chain
  )
  throws IOException, ServletException {

    //need to cast, so we can access the session
    HttpServletRequest req =
      (HttpServletRequest)request;
    HttpSession session = req.getSession();
    User user = (User)session.getAttribute("user");
    if (user == null) {
      HttpServletResponse resp =
        (HttpServletResponse)response;
      resp.sendRedirect("../login.jsp");
      return;
    }

    //pass on to next filter
    chain.doFilter(request, response);
  }

  @Override
  public void init(FilterConfig filterConfig)
  throws ServletException {
  }

  @Override
  public void destroy() {
  }

}
```

Wie wir gleich in der Konfiguration der *web.xml* sehen werden, hängt sich der Filter nur vor alle Ressourcen, die sich im Ordner *userarea* befinden. Daher müssen wir auch im *Redirect* wieder eine Ebene nach oben gehen, da sich die *login.jsp* dort befindet. Dies ist etwas unglücklich, da man diesen Filter somit schlecht mit anderen Pfaden wiederverwenden kann. Hier sollte man dann stattdessen lieber absolute Pfade verwenden, womit man aber die Domäne und den *ContextRoot* der Anwendung kennen muss. Aber auch hier erkennt man wieder eine Schwäche in der Wiederverwendbarkeit der Anwendung, da man diese schlecht ohne Anpassungen auf andere Systeme bringen kann. Genau aus diesem Grund wird in anderen Frameworks versucht diese Pfadproblematik per zusätzlicher Konfiguration der einzelnen Anwendungsteile zu umgehen oder zu lösen. Aber auch mit JSP-Technik ist man nicht ganz machtlos. Man kann diese variablen Pfadteile zum Beispiel per *Init-Param* aus der *web.xml* in den Filter hereinreichen und diese dann zur Bildung der Pfade verwenden.

Womit wir uns nun auch den Ausschnitt der *web.xml* anschauen, in der alle Teile zusammengefügt werden.

```
<filter>
  <filter-name>LoginFilter</filter-name>
  <filter-class>de.mathema.kk.teil3.LoginFilter</filter-class>
  <init-param>
    <param-name>baseURL</param-name>
    <param-value>http://localhost:8080/HalloWeb/
      </param-value>
  </init-param>
</filter>
```

```
    <filter-mapping>
      <filter-name>LoginFilter</filter-name>
      <url-pattern>/userarea/*</url-pattern>
    </filter-mapping>

    <servlet>
      <servlet-name>LoginServlet</servlet-name>
      <servlet-class>de.mathema.kk.teil3.LoginServlet</
        servlet-class>
    </servlet>
    <servlet-mapping>
      <servlet-name>LoginServlet</servlet-name>
      <url-pattern>/login</url-pattern>
    </servlet-mapping>

  <session-config>
      <session-timeout>60</session-timeout>
  </session-config>
```

Im Folgenden noch eine Abbildung, wie die Struktur der Beispielanwendung sich im Projektbaum in *Eclipse* darstellt:

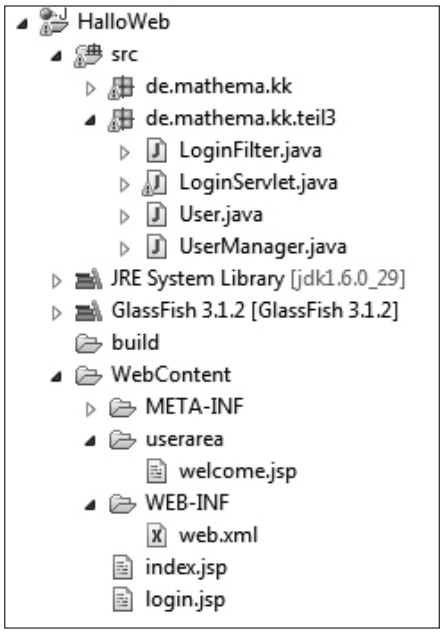


Abbildung 3

Damit hätten wir alle Teile unserer Beispielanwendung beisammen, und müssen sie nur noch auf dem Server deployen. Um unseren Filter zu testen, brauchen wir eigentlich nur die *welcome.jsp* im Unterordner *userarea* im *ContextRoot* unserer Web-Anwendung aufzurufen, ehe wir uns eingeloggt haben. Der Effekt sollte sein, dass wir stattdessen auf der *login.jsp* landen, weil eben noch kein User-Objekt in der Session liegt. Nach erfolgreichem Login sollte die *welcome.jsp* jedoch erreichbar sein und uns mit dem hinterlegten Namen des eingeloggten Users begrüßen. Zugegeben, die Benutzerführung ist etwas krude, aber in dem Beispiel soll es auch in erster Linie um die technische Funktionalität gehen und nicht um hübsches Design. Darum kümmern sich zumeist eh Agenturen und als Entwickler versucht man dann deren Vorstellungen in machbaren Code umzusetzen.

Ausblick

Ich möchte auch diesen Abschlussteil damit einleiten, dass wir auch in diesem Artikelteil nur ein wenig an der Oberfläche gekratzt haben. Denn wir haben immer noch einige Dinge offen, wovon wir uns auch im nächsten Teil wieder zwei herauspicken werden. Dann soll es nämlich mehr um *JSP*s gehen, und welche Möglichkeiten wir haben um dort auf Daten zugreifen zu können. Stichwort: *Beans* und *Expression Language*. Damit ist es wesentlich einfacher strukturierte Seiten zu erstellen und von Designern erstellte *HTML-Templates* mit Ausgaben der eigenen Anwendung zu füllen.

Referenzen

[1] Goraus Frank
Nicht nur Spinnen bauen Netze, Web-Entwickung für Java-Entwickler,
KaffeeKlatsch 12/2012, Seite 6

[2] Goraus Frank
Nicht nur Spinnen bauen Netze, Web-Entwickung für Java-Entwickler – Teil 2
KaffeeKlatsch 06/2013, Seite 10

[3] Immutable Objects,
http://docs.oracle.com/javase/tutorial/essential/concurrency/immutable.html

Weiterführende Literatur

- SelfHTML
  http://de.selfhtml.org
- Ullenboom Christian *Java ist auch eine Insel,*
  Kapitel 17, *Servlets und Java Server Pages*,
  http://openbook.galileodesign.de/javainsel5/javainsel17_000.htm
- jsptutorial *Servlet-Filter,*
  http://www.jsptutorial.org/content/filter

Kurzbiografie

Frank Goraus (frank.goraus@mathema.de) ist Senior Developer bei der MATHEMA Software GmbH in Erlangen. Seit 2006 beschäftigt er sich bereits mit der Entwicklung von JEE-Anwendungen, u. a. in Verbindung mit einem Portal-Server. Seine Liebe zum Detail verwirklicht er mit seinen Web-Design-Kenntnissen. In seiner Freizeit beschäftigt er sich außerdem mit Android-Entwicklung, verschiedensten Web-Frameworks und einem eigenen Projekt für eine Sammlungsverwaltung.

Des Programmierers kleine Vergnügen

Apropos Wächter

von Michael Wiedeking

Das Prinzip des im letzten Vergnügen vorgestellten Wächters kann natürlich nach Belieben erweitert werden. Mein persönlicher, ungeschlagener Favorit ist dabei aber der Einsatz in einer verketteten Liste.

Stellen Sie sich einmal vor, Sie benötigen eine doppelt verkettete Liste. Die sieht dann typischerweise so aus:

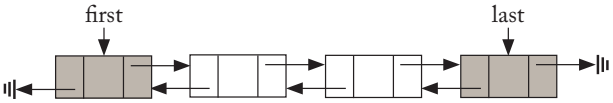
Wie man dieser Implementierung ansieht, gibt es zwei Sonderfälle, den einen am Anfang der Liste, den anderen am Ende. Das ist natürlich denkbar unangenehm, da bei jedem Einfügen oder Entfernen eines Elements der eine, der andere oder gar beide Fälle Beachtung finden müssen.

Darüber hinaus muss man neben einer Referenz auf das erste Element oft aus Gründen der Performanz noch eine zweite vorhalten, die auf das Ende der Liste zeigt. Auch hier will darauf geachtet werden, dass beim Einfügen und Entfernen diese beiden Referenzen angepasst werden.

```
void insertAfter(Node current, Node new) {
  if (current == null) {
    first = new;
    last = new;
  } else {
    new.next = current.next;
    new.prev = current;
    current.next = new;
    if (new.next != null) {
      new.next.prev = new;
    } else {
      last = new;
    }
  }
}
```

Das ist natürlich ausgesprochen lästig, wie man es dem Code für das Einfügen eines Elements ansehen kann. Zugegeben, diese Implementierung ist wegen des Sonderfalls *current* = *null* etwas komplizierter als nötig, aber dieser Sonderfall müsste ansonsten außerhalb vom *insertAfter* abgehandelt werden. Wie beim letzten Mal gelernt, lässt sich ein Teil der Sonderfälle eliminieren, indem man statt dieser Wächter einfügt, die diese zusätzlichen Tests überflüssig machen.

Dazu gestaltet man die Liste derart, dass sie aus mindestens zwei *Nodes* besteht, dem *first-Node* und dem *last-Node*, die jeweils den Anfang bzw. das Ende der Liste markieren. Damit fällt der Fall *current* == *null* komplett weg, da in diesem Fall einfach der *first-Node* übergeben wird. Zusätzlich entfällt auch der Fall *new.next* == *null*, da der *last-Node* ja immer existiert und das Ende der Liste markiert. Darüber hinaus muss auch der Verweis auf das letzte Element nicht angepasst werden, weil ja auch dieser immer fest ist.



Damit reduziert sich der Aufwand für das *insertAfter* auf das Folgende:

```
void insertAfter(Node current, Node new) {
  new.next = current.next;
  new.prev = current;
  current.next.prev = new;
  current.next = new;
}
```

Für das Einfügen und Entfernen wird nun nur noch der Verweis auf ein Element benötigt, ohne das irgendwelche Abfragen gemacht werden müssen. Das ist schon beeindruckend, hat aber – wie könnte es anders sein – auch seine Nachteile. Will man nämlich über die Liste iterieren, dann könnte man das recht einfach mit einer *null*-Abfrage erledigen.

```
current = first;
while (current != null) {
  …
  current = current.next;
}
```

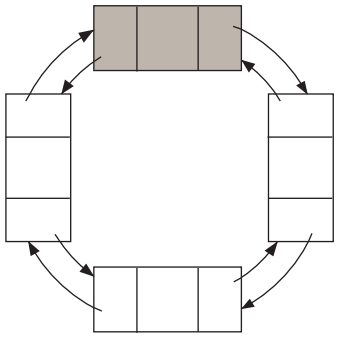
Das geht natürlich so nicht mehr. Während man vorher beim Iterieren über die Liste diese nicht kennen musste, muss man nun das *Sentinel* kennen. Dementsprechend

muss bei einer derartigen Iteration das Abbruchkriterium angepasst werden.

while (*current* != *last*) …

Insbesondere muss auch bei umgekehrter Iterationsrichtung das entsprechende Sentinel *first* benutzt werden.

Ungeachtet dessen geht das selbstverständlich noch ein bisschen einfacher. Da man vom *first*-Knoten immer nur die *next*-Referenz benötigt und vom *last*-Knoten ausnahmslos den *prev*-Zeiger, kann man die beiden Wächter einfach zu einem zusammenfassen. Damit kann man die Liste selbst zu einem Knoten machen (was allerdings eine weitere Optimierung ist, deren Beschreibung hier den Rahmen sprengen würde).



Die *next*-Referenz dieses Sentinels ist also das erste Element der Liste oder das Sentinel selbst, wenn die Liste leer ist. Die *prev*-Referenz ist dann das letzte Element in der Liste und der Verweis auf sich selbst bei leerer Liste analog. Von besonderer Eleganz ist übrigens, dass eine Liste mit diesem Sentinel ein Löschen von Elementen in der leeren Liste unbeschadet übersteht.

Es ist schon schön, dass man auch in der Informatik immer wieder eleganten, „runden“ Lösungen begegnet, die dann tatsächlich bis ins letzte Detail frei von überraschenden Sonderfällen sind. Oder um dies in den Worten von Col. John ‘Hannibal Smith’ vom A-Team zu sagen: *I love it when a plan comes together!* oder in der deutschen Synchronübersetzung *Ich liebe es, wenn ein Plan funktioniert!*

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.

Kaffeesatz

Ping

von Michael Wiedeking

Sendet man ein Signal aus, dass irgendwo reflektiert wird, so kann man aus dem, was zurückkommt, Rückschlüsse ziehen, die beispielsweise Aufschluss über Ort oder Entfernung geben können. Jetzt macht man sich dieses Verfahren auch im Rahmen der SEPA-Umstellung zu nutze.

Bei einem Echolot oder Sonar sendet man im Wasser vertikal bzw. horizontal Schallwellen aus und „schaut" dann nach, was zurückkommt. Da sich Schall im Wasser sehr verlustarm ausbreitet, kann man anhand der Reflexionen den Ort oder die Entfernung von Gegenständen ermitteln. Man ist sich nicht ganz sicher, aber man vermutet, dass diese Schallwellen die Wale irritieren und deshalb an die Strände treibt; aber das ist eine andere Geschichte, die nicht hierher gehört.

Mit dem Diagnoseprogramm *ping* lässt sich im Internet vergleichbar prüfen, ob ein bestimmter Host erreichbar ist. Man schickt dazu ein bestimmtes Datenpaket an den Host und versucht so eine Verbindung aufzubauen. Kommt es zu einem Echo – also schickt der adressierte Host das Datenpaket wieder zurück – sagt einem diese Verbindung, dass der Host verfügbar ist. Darüber hinaus lassen sich daraus Rückschlüsse ziehen, wie lange es etwa dauert bis ein Datenpaket sein Ziel erreicht und von dort wieder zum Sender zurückkehrt.

Wenn man eine Überweisung von 1 Cent mit beliebigen Kontodaten macht, dann gibt es zwei Möglichkeiten: Das Geld kommt wieder zurück oder der Cent wird abgebucht. Im ersten Fall existiert das Konto nicht und man muss die damit assoziierten Daten irgendwie verifizieren oder kann sie getrost entsorgen. Der zweite Fall beschert einem eine gültige Kontoverbindung.

Ist der Anwender dieses Verfahrens eine „gute" Person, dann hat er damit ein „ehrenhaftes" Ziel erreicht. So können etwa Kontoverbindungen von Spendern schnell und billig geprüft werden. Oder im Zusammenhang mit der SEPA-Umstellung kostengünstig geprüft werden, ob die angegebenen Konten noch relevant sind und überhaupt eine Einwilligung zur Umstellung per Post eingeholt werden muss. Ich habe mir sagen lassen, dass auf diese Weise die Stadtwerke in einer größeren Stadt mehrere zehntausend Euro einsparen können.

Verfährt ein nicht so wohlwollendes Individuum ebenso, mag es auf diese Weise prüfen, ob es sich lohnt, später unerlaubt einen größeren Betrag von diesem verifizierten Konto einzuziehen. Dementsprechend verunsichert sind auch die betroffenen Empfänger, was man in diesem Fall tun soll. Im Internet bekommt man wie immer kompetente Hilfe. Von dem Rat zur Bank oder zur Polizei zu gehen, bis hin zu dem Vorschlag, einfach das viele Geld für sich arbeiten zu lassen, ist alles dabei.

Und wem gehört nun dieser Cent? Muss man den womöglich bei seiner Einkommenssteuererklärung angeben? Oder wird der dann bei der nächsten Stromrechnung verrechnet?

Übrigens lässt sich eine solche Überprüfung anscheinend auch viel einfacher und ganz kostenlos mit einer Banking-Software machen. Was hat dann also eine solche Überweisung für einen Sinn? Na klar, man kann mit ihr eine Werbebotschaft in den Verwendungszweck schreiben.

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.

Lektüre

Technisches Schreiben

Für Informatiker, Akademiker, Techniker und den Berufsalltag

Christoph Prevezanos

Gebunden; 231 Seiten, Deutsch
Carl Hanser Verlag; Oktober 2013
Print ISBN: 978-3-446-43721-0
eISBN: 978-3-446-43759-3

rezensiert von Thomas Künneth

Die Software wurde pünktlich ausgeliefert, der Abnahmetest bestanden. Die Location für die *Release*-Party ist gebucht und der Sekt kalt gestellt. Eher beiläufig fragt der Kunde, wann das Betriebshandbuch, das Sicherheitskonzept und das Benutzerhandbuch übergeben werden können...

Natürlich gehören solche Dokumente in die Liste der Liefergegenstände und müssen wie klassische Software geplant, umgesetzt und qualitätsgesichert werden. Den wenigsten Projekten stehen hierfür ausgebildete technische Redakteure zur Verfügung. So werden Entwickler, Tester und Projektleiter oft ungewollt zu Schriftstellern. Dabei sind die Anforderungen an technische Dokumente meist hoch. Bedienungsanleitungen müssen in einfachen, nachvollziehbaren Worten möglichst auch Laien die Funktion eines Gerätes oder einer Software nahe bringen. Ein Betriebshandbuch weist dem Operator in knappen aber eindeutigen Schritten den Weg. Das Sicherheitskonzept hält zahlreiche formale Vorgaben ein. Auch gestalterische Vorgaben müssen eingehalten werden. Die Wahl einer geeigneten Schrift mag noch halbwegs glatt von der Hand gehen. Eine sinnvolle und korrekte Strukturierung des zu erstellenden Dokuments ist ungleich schwieriger.

Hier würde es helfen, einen Leitfaden zur Erstellung technischer Dokumente zur Hand zu haben. Diesen möchte Christoph Prevezanos mit seinem bei Hanser erschienenen Buch *Technisches Schreiben* bieten. Auf gut 200 Seiten erfährt der Leser alles, was er für den Einstieg in das Erstellen technischer und wissenschaftlicher Texte wissen muss. Das beginnt mit der Wahl des richtigen Handwerkszeugs, der korrekten Verwendung von Formatvorlagen sowie der Gestaltung von Seiten. Eigene Kapitel beschäftigen sich mit dem Einbinden von Tabellen und Grafiken, der richtigen Auswahl und Nutzung von Normen und Standards sowie der korrekten Verwendung von Verzeichnissen und Anhängen. Breiten Raum räumt der Autor der Vermittlung des inhaltlichen Rüstzeugs ein. Hierzu gehören insbesondere die Strukturierung des Inhalts und das richtige Zitieren und Verweisen. In Zeiten schnell erhobener Plagiatsvorwürfe ist deren Einhaltung essenziell.

Prevezanos hat einen sehr angenehmen Schreibstil. Es macht Spaß das Buch von Deckel zu Deckel durchzuarbeiten. Auch während dem Schreiben sollte es als Nachschlagewerk griffbereit auf dem Tisch liegen. Der moderne und schnörkellose Umgang mit dem Thema technisches Schreiben machen den Titel zu einem wichtigen Begleiter bei den ersten Schritten zu einem technischen oder wissenschaftlichen Dokument. Dies gilt für die klassische Dokumentation ebenso wie den Artikel für eine Zeitschrift und das eigene Buch.

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 UserGroup Bremen
http://www.jugbremen.de
Kontakt: Rabea Gransberger (rgransberger@gmx.de)

JUG DA
Java User Group Darmstadt
http://www.jug-da.de
Kontakt: jug-da-orga@googlegroups.com

Java User Group Saxony
Java User Group Dresden
http://www.jugsaxony.de
Kontakt: Herr Falk Hartmann
(falk.hartmann@jugsaxony.org)

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

ruhrjug
Java User Group Essen
Glaspavillon Uni-Campus
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
(alexander.culum@web.de)

JUG Deutschland e.V.
Java User Group Deutschland e.V.
c/o Stefan Koospal
http://www.java.de (office@java.de)

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

JUG Karlsruhe
Java User Group Karlsruhe
http://jug-karlsruhe.de
(jugkarlsruhe@gmail.com)

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

jugm
Java User Group München
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 Natalia Wilhelm
(info@jug-n.de)

JUG Ostfalen
Java User Group Ostfalen
(Braunschweig, Wolfsburg, Hannover)
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
http://www.jugs.org
Kontakt: Herr Dr. Micheal Paus (mp@jugs.org)
Herr Hagen Stanek (hs@jugs.org)
Rainer Anglett (ra@jugs.org)

Schweiz

JUGS
Java User Group Switzerland
http://www.jugs.ch (info@jugs.ch)

.NET User Groups

Deutschland

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

.NET User Group Dortmund (Do.NET)
c/o BROCKHAUS AG
http://do-dotnet.de
Kontakt: Paul Mizel (pmizel@do-dotnet.de)

Die Dodnedder
.NET User Group Franken
http://www.dodnedder.de
Kontakt: Herr Udo Neßhöver, Frau Ulrike Stirnweiß
(info@dodnedder.de)

.NET UserGroup Frankfurt
http://www.dotnet-usergroup.de

.NET User Group Hannover
http://www.dnug-hannover.de
Kontakt: (dnug@indisoftware.de)

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

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

.NET User Group 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
http://www.dotnet-oldenburg.de
Kontakt: Herr Hilmar Bunjes
(hilmar.bunjes@dotnet-oldenburg.de)
Frau Yvette Teiken (yvette.teiken@dotnet-oldenburg.de)

.NET Developers Group Stuttgart
Tieto Deutschland GmbH
http://www.devgroup-stuttgart.de
(GroupLeader@devgroup-stuttgart.de)
Kontakt: Herr Michael Niethammer

.NET Developer-Group Ulm
c/o artiso solutions GmbH
http://www.dotnet-ulm.de
Kontakt: Herr Thomas Schissler (tschissler@artiso.com)

Österreich

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

Software Craftsmanship Communities

Deutschland, Schweiz, Österreich

Softwerkskammer – Mehrere regionale Gruppen und Themengruppen unter einem Dach
http://www.softwerkskammer.org
Kontakt: Nicole Rauch (nicole.m@gmx.de)

MATHEMA

meet the experts of enterprise infrastructure

Einführung in die objektorientierte Programmiersprache Java

Eine praxisnahe Einführung
10. – 14. Februar 2014, 30. Juni – 4. Juli 2014,
2.150,– € (zzgl. 19 % MwSt.)

Einführung in C# und .NET

Einstieg in C# und die .NET-Plattform für Programmieranfänger
10. – 14. März 2014, 21. – 25. Juli 2014,
2.150,– € (zzgl. 19 % MwSt.)

Entwicklung mobiler Anwendungen mit iOS

7. – 9. April 2014, 15. – 17. September 2014,
1.250,– € (zzgl. 19 % MwSt.)

HTML5, CSS3 und JavaScript

31. März – 3. April 2014, 22. – 25. September 2014,
1.650,– € (zzgl. 19 % MwSt.)

Fortgeschrittenes Programmieren mit Java

Ausgewählte Pakete der Java Standard Edition (J2SE)
5. – 7. Mai 2014, 13. – 15. Oktober 2014,
1.350,– € (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

„An MATHEMA schätze ich unsere öffentliche Präsenz, z. B. bei User Groups und Konferenzen, sowie die Vielseitigkeit meiner Tätigkeit."

William Siakam, Consultant

Wir sind ein Consulting-Unternehmen mit Schwerpunkt in der Entwicklung unternehmenskritischer, verteilter Systeme und Umsetzung von Service-orientierten Architekturen und Applikationen von Frontend bis Backend. Darüber hinaus ist uns der Wissenstransfer ein großes Anliegen: Wir verfügen über einen eigenen Trainingsbereich und unsere Consultants sind regelmäßig als Autoren in der Fachpresse sowie als Speaker auf zahlreichen Fachkonferenzen präsent.

join the experts of enterprise infrastructure

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

Das Allerletzte

serviceFound = **true** & !abbruch ;

Dies ist kein Scherz!
Dieses Stück Beta-Code 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 Januar.

Herbstcampus

Wissenstransfer par excellence

1. – 4. September 2014

in Nürnberg