mehr zum thema:
www.arc42.de/
www.aslbislfoundation.org/component/option,com_docman/task,doc_download/gid,428/Itemid,24/lang,en/

BETRIEBSORIENTIERTE SOFTWAREENTWICKLUNG: FEHLER IN DER PRODUKTIVUMGEBUNG SUCHEN UND FINDEN

Bei der Entwicklung von Softwareprojekten liegt der Fokus meist auf funktionalen Aspekten oder auf Softwareaspekten. Anforderungen des Betriebs, wie beispielsweise die Überwachung des Endprodukts zur Laufzeit, werden dabei häufig vernachlässigt. Probleme fallen dann erst beim Deployment der Anwendung in der Zielumgebung auf, wenn es eigentlich schon zu spät ist. In diesem Artikel wird gezeigt, dass die frühzeitige Berücksichtigung zentraler betrieblicher Anforderungen den Erfolg des Endprodukts nachhaltig sichern kann.

die autoren

Christian Unglaube
(E-Mail: christian.unglaube@bridging-it.de)
ist Senior Consultant beim IT-Beratungsunternehmen BridgingIT GmbH. Als Java/JEE-Experte und Softwarearchitekt verfügt er über langjährige Erfahrung bei der Durchführung von Projekten in der Finanzdienstleistungs- und Pharmabranche.

Dunja Winkens
(E-Mail: dunja.winkens@bridging-it.de)
ist Senior Consultant bei der BridgingIT GmbH. Sie ist Unternehmensarchitektin und Qualitätsmanagerin und verfügt über langjährige Erfahrung bei der Einführung von Entwicklungsstandards in verschiedenen Branchen.

Wem ist diese Situation in einem Entwicklungsprojekt noch nicht begegnet? In letzter Minute wird das Deployment-Paket ausgeliefert. Sämtliche Unit-Tests sind erfolgreich durchgelaufen und auf dem Abnahmesystem konnte das Qualitätssicherungsteam trotz intensiver Tests keine Fehler mehr finden. Also scheint dem Projekterfolg nichts mehr im Weg zu stehen – und dennoch tritt die Katastrophe ein: Nur wenige Minuten nach dem Deployment der Anwendung in der Produktivumgebung geht nichts mehr. Die Anwendung steht und die eilig inspizierte Log-Datei gibt keinen Aufschluss über die Ursache des Problems. In großer Hektik wird das Deployment rückgängig gemacht und die ursprüngliche Version der Anwendung wieder installiert. Die Verantwortlichen fragen sich: „Wie konnte das passieren und was kann man tun, um derartige Vorfälle künftig in den Griff zu bekommen?"

Der klassische Blick in die Log-Datei ist typischerweise die erste Aktion, um dem Fehler auf die Spur zu kommen. Aber neben der Log-Datei-Auswertung gibt es eine ganze Reihe zusätzlicher Möglichkeiten, um die Problemanalyse in einer häufig nur schwer zugänglichen Produktivumgebung zu vereinfachen. An dieser Stelle setzt der Artikel an und stellt einige bewährte Vorgehensweisen vor, die sich in verschiedenen Projekten, besonders im Umfeld von kleineren JEE-Web-Anwendungen zur Reduzierung von Ausfallzeiten, als wertvoll erwiesen haben. Auch die zentrale Rolle des Architekten im Rahmen des Anforderungsmanagements wird beleuchtet. Welche Aspekte muss er beim Entwurf einer Lösung beachten, um den Anforderungen des Betriebs gerecht zu werden und einen möglichst reibungslosen Übergang von der Entwicklung bis in den produktiven Betrieb zu gewährleisten?

Betriebliche Anforderungen im Entwicklungsprozess

Viele Projektmethoden bestehen im Grunde aus einzelnen Phasen, in denen der Architekt jeweils die Interessen des Betriebs wahrnehmen muss. Tätigkeitsschwerpunkte eines Architekten sind vor allem das Erfassen der Anforderungen und die Implementierung. Wenn die funktionalen Anforderungen erfasst werden, müssen auch die betrieblichen Anforderungen ermittelt und im Lösungskonzept berücksichtigt werden. Während der Implementierungsphase muss die Einhaltung der Architekturvorgaben überprüft werden, z. B. durch regelmäßige Reviews. Auf diese Weise werden die betrieblichen Anforderungen im gesamten Entwicklungsprozess fest verankert.

Für eine Optimierung der Qualität der ausgelieferten Software muss nicht nur die Codequalität betrachtet werden, auch die Prozesse zum Build und Deployment der Software sind hier relevant. Durch eine weitgehende Automatisierung können viele Fehlerquellen bereits hier minimiert werden. Wenn ein *CI*-System (*Continuous Integration*) eingesetzt wird, kann man zudem – neben den rein technischen Metriken – auch die betriebsrelevanten Metriken für die Stabilität, Wartbarkeit und Portabilität der Software überwachen und so den Architekten frühzeitig auf entstehende Probleme aufmerksam machen.

Anforderungsmanagement im Betrieb

In der Anfangsphase eines Entwicklungsprojekts werden sowohl die fachlichen als auch die nicht-funktionalen Anforderungen abgestimmt. Der Dialog mit der Fachseite erscheint ganz selbstverständlich, nicht zuletzt, weil diese meistens auch für die Finanzierung des Projekts zuständig ist. Die Belange des Betriebs fallen aber gerne unter den Tisch und können in der Endphase des Projekts wie ein Bumerang zurückkommen.

Am einfachsten erfolgt die Abstimmung der Anforderungen in Form eines oder mehrerer Workshops, an denen neben der Fachseite und dem Betrieb auch der Softwarearchitekt teilnehmen sollte. Anforderungen des Betriebs an die neu zu ent-

wickelnde Software sollten dabei in messbarer Form, wie z. B. Verfügbarkeit, Antwortzeit oder Anzahl parallele Zugriffe, erfasst und dokumentiert werden. Häufig bereitet die Definition von nicht-funktionalen Anforderungen dem Kunden Probleme, da ihm nur wenige Erfahrungswerte vorliegen. Für diesen Fall hat sich der Einsatz der Szenario-Methode (vgl. [Hru09-a]) bewährt, bei der anhand verschiedener durchgespielter Szenarien ermittelt wird, wie die Erwartungen des Kunden an das neue System sind. Des Weiteren ist der Einsatz von Checklisten für nicht-funktionale Anforderungen zu empfehlen, die von verschiedenen Institutionen für Anforderungsmanagement (z. B. [Vol], [Arc], [ASL08]) angeboten werden.

SLAs hinterfragen

Nicht-funktionale Anforderungen werden häufig über *Service-Level-Agreements (SLAs)* festgelegt, in denen beispielsweise die Verfügbarkeit oder das Antwortzeitverhalten der Anwendung beschrieben ist. Der Architekt muss wissen, in welchem Umfang er von den angeschlossenen Systemen abhängig ist, bevor er qualifizierte Aussagen zu seinem System treffen kann. Mit einer eng gekoppelten Softwarelösung kann er keinesfalls eine höhere Verfügbarkeit als das anfälligste Teilsystem erreichen. Liegt keine belastbare Beschreibung der Schnittstellen vor, ist der Architekt gefordert, die resultierenden Risiken für die Anwendung zu bewerten. Bei der Formulierung eigener SLAs ist es zudem wichtig, exakt zu definieren, unter welchen Voraussetzungen sie gelten und welche Annahmen ihnen zu Grunde liegen. Beispielsweise sollten Latenzzeiten, auf die die Anwendung keinen Einfluss hat, von den Antwortzeiten-SLAs ausdrücklich ausgenommen werden.

Qualitätsanforderungen aus Betriebssicht

Aus der Sicht des Betriebs sieht die Definition von Qualität etwas anders aus als für den Endnutzer. Nach unseren Erfahrungen aus vielen Projekten spielen folgende Anforderungen eine besondere Rolle:

- Eine Anwendung soll stabil laufen. Der Administrator sollte sie nicht ständig überwachen und eingreifen müssen.
- Jede Anwendung muss ihre Ressourcen – z. B. Datenbankverbindungen, Plattenplatz und Speicher – selbständig verwalten können und darf diese dabei nicht länger als nötig blockieren. Durch ein Konzept für die Archivierung und Bereinigung der Anwendungsdaten kann unnötiger Ressourcenverbrauch vermieden werden.
- Die Konfiguration der Anwendung sollte sich für den Betrieb auf ein Minimum beschränken. Die ideale Konfigurationsdatei ist klein, übersichtlich strukturiert und enthält nur die notwendigen umgebungsspezifischen Parameter. Für jede Zielumgebung gibt es am besten eine separate Konfigurationsdatei.
- Das Deployment der Anwendung sollte so einfach wie möglich und automatisiert durchführbar sein. Gibt es Probleme mit dem Deployment, sollte immer eine einfache automatisierte Möglichkeit vorhanden sein, um auf die Vorgängerversion zurückgehen zu können.
- Bereits beim Start der Anwendung wird automatisch geprüft, ob alle Ressourcen, die die Anwendung benötigt, in ausreichendem Umfang zur Verfügung stehen. Ist das nicht der Fall, wird der Startprozess abgebrochen und die Fehlerursache wird in der Log-Datei detailliert protokolliert bzw. in der Konsole ausgegeben.
- Tritt ein Problem während des laufenden Betriebs der Anwendung auf, kann der Log-Level ohne Neustart einfach verändert werden, damit weitere Detailinformationen verfügbar sind.
- Die Administration der Anwendung sollte über standardisierte Administrationsschnittstellen (z. B. *Java Management Extensions – JMX, Windows Management Instrumentation – WMI*) erfolgen, um das einfache Einbinden in eine zentrale Administrationsanwendung zu ermöglichen. Reichhaltig ausgestattete Administrationsoberflächen sind für das Durchführen wiederholter Tätigkeiten weniger geeignet – hier ist eine Schnittstelle über die Konsole wichtig.
- Die Absicherung von Administrationsseiten sollte nicht über den gleichen Weg wie für die Anwendung erfolgen, damit er beim Ausfall der Authentifizierungskomponente noch Zugriff auf die Anwendung hat.
- Das Betriebshandbuch sollte den geplanten Ressourcenbedarf für Datenbank, Log-Dateien und Cache-Größe sowie eine Liste von definierten Fehlerzuständen und ersten Hilfsmaßnahmen enthalten.
- Eine hilfreiche Fehlermeldung in der Log-Datei enthält eine aussagekräftige Fehlerbeschreibung und weitere Details, die dem Administrator eine erste Ursachenforschung ermöglichen. Die Verwendung einer Identifikationsnummer mit Datum kann die Log-Datei-Auswertung zusätzlich erleichtern.

- maximale Stabilität der Anwendung
- minimale Konfiguration
- ressourcenschonendes Verhalten
- einfaches, rasches Deployment mit *Fallback*-Lösung
- *Sanity-Check* beim Start
- dynamische Log-Level-Einstellung
- vollständiges Betriebshandbuch
- Konzept für Datenbank-Bereinigung
- auffindbare Log-Einträge
- aussagekräftige Fehlerbeschreibungen im Log

Kasten 1: *Wunschzettel eines Administrators.*

Betriebsanforderungen im Architekturkonzept

Bei der Festlegung von Architekturentscheidungen spielen nicht-funktionale Anforderungen häufig eine wichtigere Rolle als die funktionalen Anforderungen, obwohl Erstere von der Fachseite nur am Rande wahrgenommen werden. Der Fokus des Kunden liegt eher auf der Umsetzung der fachlichen Anforderungen. Erst später – in der Wartungsphase – wird er mit Kosten konfrontiert, die häufig auf eine unzureichende Umsetzung der nicht-funktionalen Anforderungen zurückzuführen sind. Die schriftliche Festlegung der nicht-funktionalen Anforderungen im Architekturkonzept kann hier deren Berücksichtigung im kompletten Entwicklungsprozess unterstützen. Für die Umsetzung der betrieblichen Anforderungen gibt es verschiedene architekturrelevante Aspekte, von denen wir im Folgenden einige erläutern.

Stabile Lösungen konzipieren

Für die Stabilität der Anwendung ist die Konzeption einer losen Kopplung der Teilsysteme innerhalb der Architektur ein

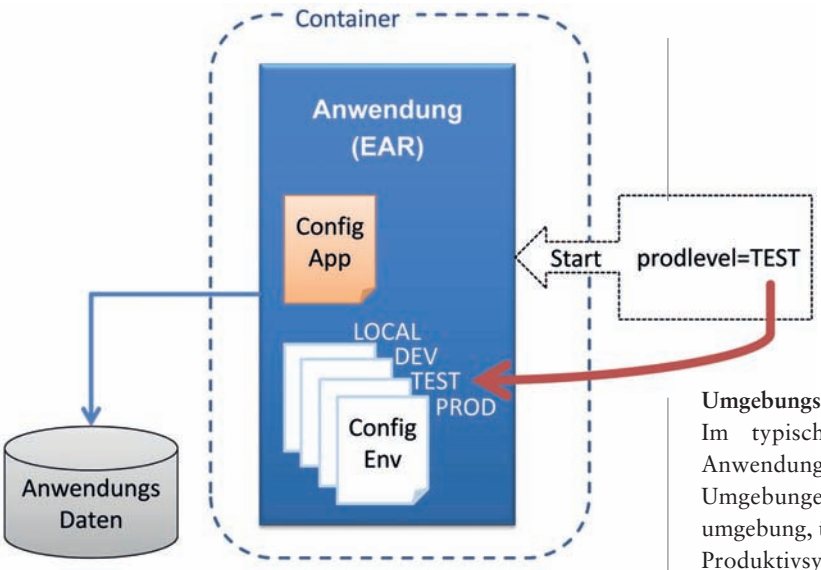


Abb. 1: Umgebungskonfiguration mit Umgebungsvariablen.

wirkungsvolles Mittel. Fehlerzustände, die weitreichende Auswirkungen auf die Gesamtanwendung und auch auf ihre Administrierbarkeit haben, werden so vermieden. Durch den gezielten Einsatz bewährter Entwurfsmuster (z. B. *Bulkheads Pattern, Circuit Breaker Pattern*, vgl. [Nyg07]) kann dieses Architekturprinzip realisiert werden. Beispielsweise kann eine Online-Shop-Anwendung unter Umständen auch ohne ein Auftragsmanagement-System stabil weiter laufen und erst zu einem späteren Zeitpunkt, wenn das Order-System wieder erreichbar ist, die Bestellungen abwickeln. Bei der Implementierung dieser Entwurfsmuster muss das Verhalten, d. h. der Ausfall, in Log-Dateien oder Monitoring-Tools deutlich sichtbar sein, damit entsprechend darauf reagiert werden kann.

Ressourcen schonen

Bei der Verwendung ressourcenintensiver Prozesse kann es wichtig sein, vorab zu prüfen, ob sämtliche benötigten Ressourcen zur Verfügung stehen. So kann auf bestehende Engpässe unmittelbar mit einer Fehlermeldung reagiert werden. Beispielsweise sollte die Verfügbarkeit des E-Mail-Systems geprüft werden, bevor eine E-Mail zur Auftragsbestätigung erzeugt wird, die mit Daten aus unterschiedlichen Teilsystemen (z. B. Kundendatenverwaltung und Produktdaten) zusammengestellt wird.

Eine betriebsorientierte Anwendung geht so schonend wie möglich mit ihren Ressourcen um. Das gilt für ganz unterschiedliche Bereiche einer Anwendung, wie zum Beispiel:

- *Logging:* Es wird effektiv protokolliert, um Log-Dateien möglichst klein zu halten.
- *Datenbanken:* Zur schonenden Nutzung einer Datenbank gehört die konsequente Nutzung von Indizes. Existieren spezielle Suchanforderungen, kann ein darauf ausgerichtetes Design durch einen Datenbank-Spezialisten viele Performance-Probleme vermeiden.
- *Caches:* Caches müssen so konzipiert werden, dass sie den zur Verfügung stehenden Speicher effektiv nutzen und Methoden für die Verwaltung des Caches implementiert werden.

Automatisierter Build- und Deployment-Prozess

Der Architekt sollte Sorge tragen, dass sowohl der Build- als auch der Deployment-Prozess automatisiert ablaufen und so wenig wie nötig manuelles Eingreifen des Administrators erfordern. Jede Änderung eines Konfigurationsparameters, die durch einen manuellen Eingriff erfolgen muss, birgt das Risiko eines Fehlers.

Umgebungswechsel reibungslos gestalten

Im typischen Softwareentwicklungsprozess durchläuft die Anwendung verschiedene Testphasen, die auf unterschiedlichen Umgebungen stattfinden. Von der lokalen Entwicklungsumgebung, über eine Integrations-und Testumgebung, bis hin zum Produktivsystem sind immer wieder Deployments und verschiedene Konfigurationen der Anwendung notwendig. Dieser Übergang zwischen den Umgebungen sollte so reibungslos wie möglich erfolgen. Zunächst ist hier eine klare Trennung der anwendungs- und umgebungsspezifischen Konfigurationsparameter in separaten Dateien wichtig. Zudem sollte es für jede Umgebung eine eigene Datei geben. Ziel ist es, dass der gleiche Build für alle Umgebungen verwendet werden kann und somit Fehler durch mehrere umgebungsabhängige Builds reduziert werden. Welche Umgebung aktiv ist, wird einmalig beim Deployment, zum Beispiel direkt auf dem Server durch einen Umgebungsparameter, eingestellt. Die Anwendung muss dann gewährleisten, dass die richtige Konfiguration angezogen wird, z. B. für die TEST-Umgebung. In **Abbildung 1** wird beispielsweise die Umgebung TEST beim Start der Anwendung mit dem Parameter prodlevel=TEST aktiviert.

Eine einheitliche Fehlerbehandlung definieren

Ein erfahrener Architekt muss immer damit rechnen, dass trotz aller Vorkehrungen auch unvorhergesehene Fehler auftreten. Grundsätzlich sollte der Benutzer im Falle eines Fehlers umgehend so umfassend wie möglich informiert werden, ohne ihn dabei mit technischen Informationen zu überfordern. Eine Identifikationsnummer zum Finden des Fehlers in der Log-Datei oder auch die Angabe von Datum und Uhrzeit können hier bereits sehr hilfreich sein. Auch ein Hinweis für den Anwender, wo er Hilfe bekommen kann, fördert seine Akzeptanz. Gleichzeitig sollten alle Informationen, die zum Finden der Fehlerursache hilfreich sind, in der Log-Datei ausgegeben werden.

Das Auffangen und Verarbeiten aller Fehler an einer zentralen Stelle in der Anwendung hat sich in der Praxis bewährt. Um besonders im unerwarteten Fehlerfall alle Informationen bereitzustellen, muss die Fehlerbehandlung jedoch zusätzlich während der Entwicklung individuell berücksichtigt werden. Tritt beispielsweise ein Fehler beim Speichern eines Objekts auf, ist es wichtig, das entsprechende Objekt in der Log-Ausgabe zu finden. Auf diese Weise können Fehler in den Daten effizient gefunden und zur Fehlerbeseitigung reproduziert werden. Eine differenzierte Fehlerbehandlung innerhalb der Anwendung ist nachträglich nur schwer realisierbar und erfordert einen deutlich erhöhte Aufwand. Zum Entwurf einer Anwendung gehören deshalb auch Vorgaben, wie sich das Programm im Fall von Ausnahmen und Fehlern verhalten soll.

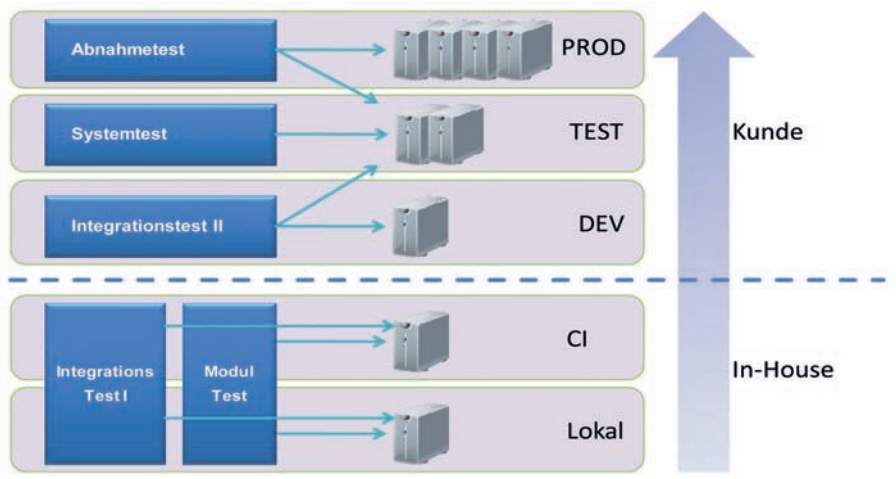


Abb. 2: Testumgebungen von der Entwicklung bis zur Produktion.

Intelligentes Logging

Nicht nur beim *Tracing* von Fehlern ist es unerlässlich, dass der Entwickler mitdenkt. Auch das Einstreuen sinnvoller Log-Nachrichten ist nur begrenzt automatisierbar und nachträglich nur schwer zu ergänzen. Einerseits müssen Teile der Anwendung sinnvoll für das Loggen aktiviert oder deaktiviert werden können, da ansonsten wichtige Informationen in der Flut der Log-Nachrichten untergehen. In der Regel reicht es beispielsweise aus, pro Klasse einen Logger zu definieren, der eigenständig aktiviert werden kann. Auch die Vergabe eines Loggers pro Modul (z. B. für die gesamte Persistenzschicht) hat sich als hilfreich erwiesen.

Wenn Log-Ausgaben eingestreut werden, sollte man darauf achten, dass die Performance im produktiven Betrieb nicht unnötig eingeschränkt wird. Für aussagekräftige Meldungen sind oftmals String-Konkatenationen aus der eigentlichen Meldung und der String-Repräsentation der Daten unterlässlich. Diese dürfen aber nur durchgeführt werden, wenn der entsprechende Log-Level tatsächlich aktiviert wird (vgl. [L4J]).

Wenn Performance-Probleme auftreten, erweisen sich außerdem Performance-Traces, die bereits während der Entwicklung gezielt eingebaut werden, als sehr hilfreich. Auf diese Weise kann die Fehlersuche schnell auf bestimmte Bereiche eingegrenzt werden.

Qualitätssicherung in unterschiedlichen Umgebungen

Software durchläuft im klassischen Projektablauf verschiedene Entwicklungs- und Testphasen, in denen jeweils bestimmte Qualitätsmerkmale auf unterschiedlichen Systemen getestet werden müssen. Das ist einer der Gründe, weshalb es so schwierig ist, die Qualität von Software sicherzustellen.

Zur lokalen Umgebung der Entwickler zählt neben der Entwicklungsumgebung meist auch ein lokaler Entwicklungsserver, auf dem erste Deployments getestet werden (siehe Abbildung. 2). Auf einem *Continuous Integration Server* findet vor allem die Integration der verschiedenen Arbeitsstände der Entwickler statt. In der Entwicklungsumgebung beim Kunden finden erste echte Integrationen mit Schnittstellensystemen der Anwendung statt („Integrationstest II" in Abbildung 2). Spätestens in der Testumgebung müssen dann alle Systeme möglichst realitätsgetreu zur Verfügung stehen. Hier finden in der Regel Benutzerakzeptanz- und Abnahmetests sowie technische Tests (z. B. Last- und Performance- sowie Ausfalltests) statt. Die Daten müssen möglichst nah an der Produktionsumgebung bereitgestellt werden, da hier der letzte Test vor dem endgültigen Deployment in der Produktion erfolgt.

Mit zunehmender Produktionsnähe verschiebt sich nicht nur die Verantwortlichkeit von der Entwicklung hin zum Betrieb, sondern es steigen auch die Aufwände für Fehlerbeseitigungen auf Seiten des Betriebs. In der Praxis hat sich deshalb eine Mischung aus testgetriebener Entwicklung und der frühzeitigen Durchführung lokaler Integrationstest als sehr wirksam erwiesen (Integrationstest I).

In automatisierten Tests, beispielsweise unter Verwendung von JUnit (vgl. [JUn]), werden dabei die Anwendungsfälle durch alle Anwendungsschichten automatisiert getestet. Hierbei wird auch das Zusammenspiel mit Schnittstellen wie Autorisierung, Authentifizierung, Mailing usw. abgedeckt. Ziel dieses Vorgehens ist es, den Entwicklern alle Systeme so praxisnah wie möglich zur Verfügung zu stellen, sodass gegen sie getestet und implementiert werden kann. Durch den Einsatz geeigneter Frameworks (z. B. [Apa], [Sou]) können diese Komponenten simuliert werden, ohne dabei die Funktionalität der Schnittstellen aufwändig nachprogrammieren zu müssen.

Vertrauen ist gut – Kontrolle ist besser

Mit dem Monitoring einer Anwendung wird häufig die Einhaltung von SLAs überwacht. Im JEE-Umfeld ist das Mittel der Wahl die *Java-Management-Extension (JMX)*. Ab der Java-Version 1.5 werden grafische Werkzeuge mit dem JDK ausgeliefert, mit denen man verschiedene Kennzahlen der Umgebung auch grafisch aufbereiten kann. Auch anwendungsspezifische Daten können durch die Implementierung als so genannte *Managed Beans* zugänglich gemacht werden. Die Nutzung von JMX muss explizit auf dem Server aktiviert werden und die erforderlichen Ports müssen in den Umgebungen freigeschaltet werden. Erst dann können die Möglichkeiten von JMX in vollem Umfang genutzt werden. Zur Nutzung applikationsspezifischer *Beans* sollte sich der Architekt bei seinem Architekturentwurf Gedanken machen, an welchen Stellen das Monitoring seiner Anwendung sinnvoll ist, und er sollte entsprechenden Aufwand für die Implementierung vorsehen.

Für den Administrator bietet die Verwendung von JMX in Anwendungen einen zusätzlichen Vorteil: Durch die Verwendung einer *Command-Line-Shell* hat er die Möglichkeit, die Anwendung über eigene Skripte zu verwalten. Die Shell gibt es für die gängigsten Applikationsserver, aber es gibt auch verschiedene Open-Source-Tools (vgl. [Jav]).

Es gibt kein Patentrezept, welche Metriken in Einzelfall zu überwachen sind. Michael T. Nygard schlägt vor, sich auf die wichtigsten Metriken zu beschränken, und nennt hier das Verkehrsvolumen, die Belastung der Ressourcen-Pools, Datenbank-Verbindungen, Cache-Belastung und alle kritischen Schnittstellen (vgl. [Nyg07]).

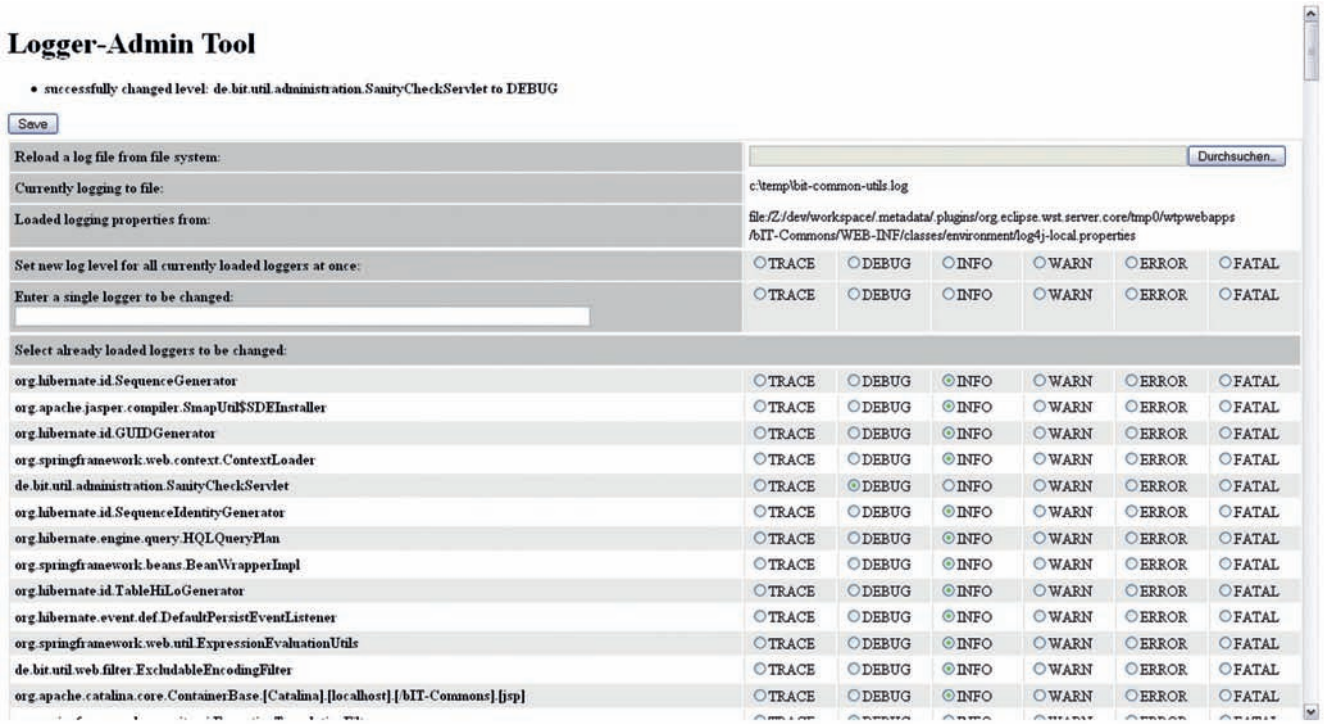


Abb. 3: Web-Seite für die dynamische Administration der Logging-Einstellungen.

Lightweight-Lösungen neben JMX

Neben den JMX-basierten Monitoring-Lösungen, gibt es aber auch andere sehr einfache Möglichkeiten, wie man unterschiedliche Informationen zum Zustand eines produktiven Systems anzeigen kann. Gerade wenn das JMX-basierte Monitoring bei einer bestehenden Anwendung unzureichend realisiert oder aus organisatorischen Gründen nicht möglich ist, können diese Lösungen auch noch nachträglich Abhilfe schaffen.

Die folgenden Beispiele sind als einfache eigenständige *Java-Server-Pages (JSPs)* bzw. *Servlets* konzipiert, die in eine bestehende Web-Anwendung eingeklinkt werden können. Sie nutzen keine Framework-Funktionalität, da sie vielseitig einsetzbar sein sollen und auch bei Problemen mit eben den Frameworks verwendet werden können. Die JSPs müssen natürlich abgesichert werden, sodass nur ein berechtigter Personenkreis Zugang dazu hat. Es hat sich als hilfreich erwiesen, nicht den Authentifizierungsmechanismus der Anwendung selbst zu nutzen, da diese einerseits bei Problemen mit diesem nicht verfügbar wären und andererseits ein Administrator oftmals nur auf genau diese Seiten Zugriff haben soll.

Einfache Konfigurierbarkeit des Log-Levels zur Laufzeit

Neben der Möglichkeit, die Log-Einstellungen zur Laufzeit automatisch vom System neu einlesen zu lassen, ist es hilfreich, sich die aktiven Einstellungen über eine Oberfläche anzeigen zu lassen und auch hier gezielt ändern zu können. In Abbildung 3 ist die Ausgabe einer einfachen JSP dargestellt, die Auskunft über die aktuellen Log-Einstellungen (Pfad der Einstellungen und der Zieldatei) und die aktiven Log-Levels gibt. Änderungen des Log-Levels greifen hierbei sofort ohne Neustart der Anwendung.

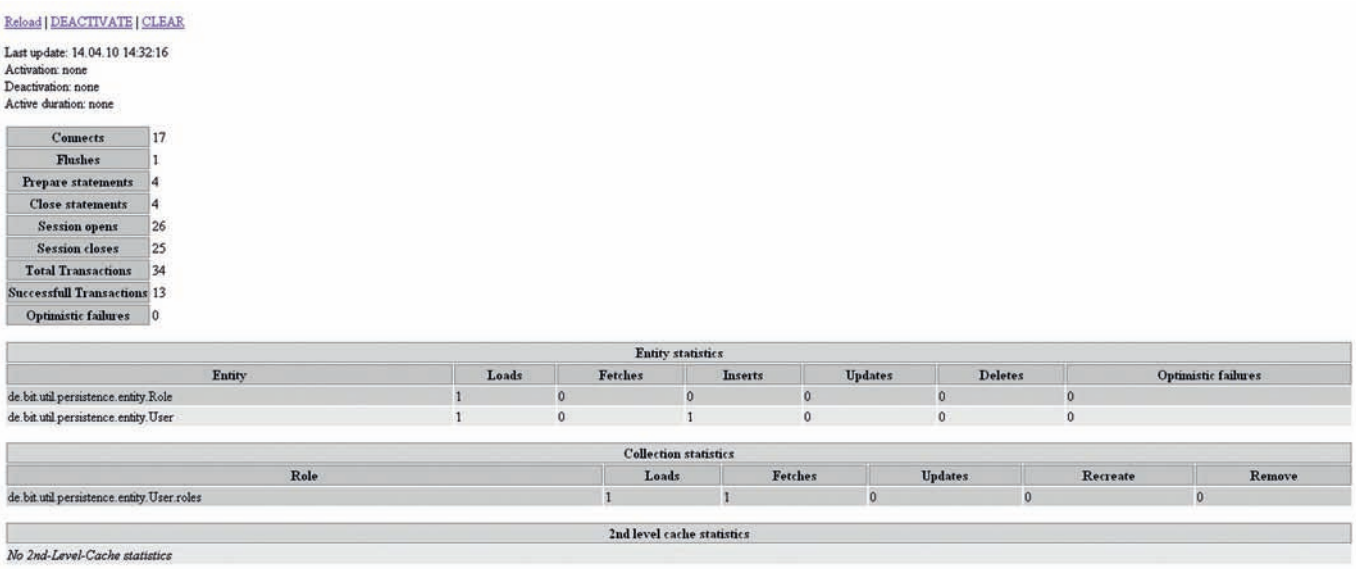
Anzeige der Datenbank-Verbindung und Ressourcenverbrauch

Verschiedene *objektrelationale Mapper (ORMs)* bieten mehr oder weniger umfangreiche Zugriffsmöglichkeiten auf statistische Daten, wie zum Beispiel den Ressourcenverbrauch der darunter liegenden Datenbank. Probleme in der Datenzugriffsschicht können hier sehr schnell sichtbar werden. Eine einfache JSP zeigt beispielsweise die statistischen Daten für den OR-Mapper Hibernate (siehe Abbildung 4 und [Hib-b]).

Sanity-Check

Anwendungsprobleme sind häufig auf fehlerhafte Einstellungen in den Umgebungsparametern zurückzuführen. Die Darstellung der aktuellen Applikations- und Umgebungseinstellungen hilft, den Übeltäter schnell zu identifizieren und zu beseitigen. Manche Fehler treten erst bei Aktivierung der entsprechenden Bereiche in einer Applikation auf. Um das Auftreten der Probleme bereits beim Systemstart transparent zu machen, dient ein so genannter *Sanity-Check*, bei dem alle wichtigen Systemfunktionen abgeprüft und angezeigt werden. Wichtig ist es natürlich, dass hier alle Sicherheitserfordernisse und Datenschutzanforderungen berücksichtigt werden und dass dies gegebenenfalls auch mit dem Kunden bzw. Betrieb abzustimmen ist. Ein solcher *Sanity-Check* kann beispielsweise folgende Informationen umfassen:

- aktuelle Applikationsversion mit externer und interner Build-Version sowie Datum
- aktive Umgebung (prodlevel in Abbildung 1)
- Systemeigenschaften (z. B. JDK-Version)
- Existenzprüfung sämtlicher Pfade (z. B. für das Laden von Templates oder Datei-Uploads)
- Prüfen von Schnittstellen (z. B. LDAP oder E-Mail)
- Anzeige wichtiger umgebungsspezifischer Einstellungen (z. B. ob Mailing aktiviert ist)



Reload | DEACTIVATE | CLEAR

Last update: 14.04.10 14:32:16
Activation: none
Deactivation: none
Active duration: none

Connects	17
Flushes	1
Prepare statements	4
Close statements	4
Session opens	26
Session closes	25
Total Transactions	34
Successfull Transactions	13
Optimistic failures	0

Entity statistics

Entity	Loads	Fetches	Inserts	Updates	Deletes	Optimistic failures
de.bit.util.persistence.entity.Role	1	0	0	0	0	0
de.bit.util.persistence.entity.User	1	0	1	0	0	0

Collection statistics

Role	Loads	Fetches	Updates	Recreate	Remove
de.bit.util.persistence.entity.User.roles	1	1	0	0	0

2nd level cache statistics

No 2nd-Level-Cache statistics

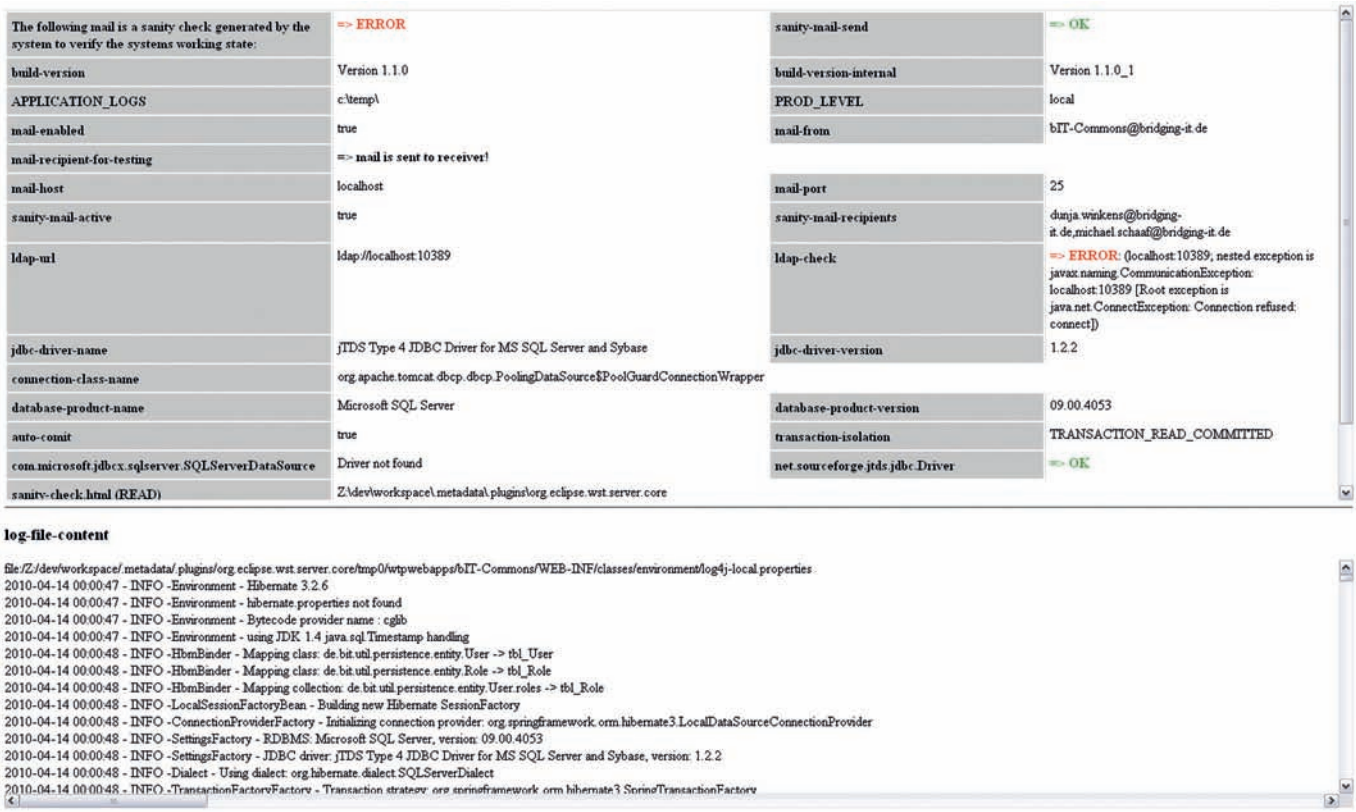
Abb. 4: Java-Server-Page zur Anzeige der aktuellen Ressourcennutzung durch Hibernate (vgl. [Hib-a]).

- aktuelle Softwareversionen (z. B. Datenbank oder Treiber)
- Anzeige aktueller Log-Ausgaben, wenn zeitnaher Zugriff auf Log-Dateien in der Laufzeitumgebung nicht möglich

Die Umsetzung kann beispielsweise durch ein *Servlet* erfolgen, das beim Systemstart ausgeführt wird. Neben dem Protokollieren von Problemen als Log-Ausgabe hat sich hier das Verschicken von Mails mit den relevanten Informationen an das Wartungsteam als sehr hilfreich erwiesen. Ein solches *Servlet* kann dann zur Laufzeit aufgerufen werden, um den aktuellen Zustand zu prüfen (siehe Abbildung 5) und gegebenenfalls auch auf einfachem Weg die Informationen und Log-Ausgaben an die Entwickler zu verteilen.

Fazit

Gerade die Belange des Betriebs werden über alle Projektphasen einer Anwendungsentwicklung leicht übersehen, obwohl sie



The following mail is a sanity check generated by the system to verify the systems working state:	=> ERROR	sanity-mail-send	=> OK
build-version	Version 1.1.0	build-version-internal	Version 1.1.0_1
APPLICATION_LOGS	c:\temp\	PROD_LEVEL	local
mail-enabled	true	mail-from	bIT-Commons@bridging-it.de
mail-recipient-for-testing	=> mail is sent to receiver!		
mail-host	localhost	mail-port	25
sanity-mail-active	true	sanity-mail-recipients	dunja.winkens@bridging-it.de,michael.schaaf@bridging-it.de
ldap-url	ldap://localhost:10389	ldap-check	=> ERROR: (localhost:10389; nested exception is javax.naming.CommunicationException: localhost:10389 [Root exception is java.net.ConnectException: Connection refused: connect])
jdbc-driver-name	jTDS Type 4 JDBC Driver for MS SQL Server and Sybase	jdbc-driver-version	1.2.2
connection-class-name	org.apache.tomcat.dbcp.dbcp.PoolingDataSource$PoolGuardConnectionWrapper		
database-product-name	Microsoft SQL Server	database-product-version	09.00.4053
auto-commit	true	transaction-isolation	TRANSACTION_READ_COMMITTED
com.microsoft.jdbcx.sqlserver.SQLServerDataSource	Driver not found	net.sourceforge.jtds.jdbc.Driver	=> OK
sanity-check.html (READ)	Z:\dev\workspace\.metadata\.plugins\org.eclipse.wst.server.core		

log-file-content

```
file:/Z:/dev/workspace/.metadata/.plugins/org.eclipse.wst.server.core/tmp0/wtpwebapps/bIT-Commons/WEB-INF/classes/environment/log4j-local.properties
2010-04-14 00:00:47 - INFO -Environment - Hibernate 3.2.6
2010-04-14 00:00:47 - INFO -Environment - hibernate.properties not found
2010-04-14 00:00:47 - INFO -Environment - Bytecode provider name : cglib
2010-04-14 00:00:47 - INFO -Environment - using JDK 1.4 java.sql.Timestamp handling
2010-04-14 00:00:48 - INFO -HbmBinder - Mapping class: de.bit.util.persistence.entity.User -> tbl_User
2010-04-14 00:00:48 - INFO -HbmBinder - Mapping class: de.bit.util.persistence.entity.Role -> tbl_Role
2010-04-14 00:00:48 - INFO -HbmBinder - Mapping collection: de.bit.util.persistence.entity.User.roles -> tbl_Role
2010-04-14 00:00:48 - INFO -LocalSessionFactoryBean - Building new Hibernate SessionFactory
2010-04-14 00:00:48 - INFO -ConnectionProviderFactory - Initializing connection provider: org.springframework.orm.hibernate3.LocalDataSourceConnectionProvider
2010-04-14 00:00:48 - INFO -SettingsFactory - RDBMS: Microsoft SQL Server, version: 09.00.4053
2010-04-14 00:00:48 - INFO -SettingsFactory - JDBC driver: jTDS Type 4 JDBC Driver for MS SQL Server and Sybase, version: 1.2.2
2010-04-14 00:00:48 - INFO -Dialect - Using dialect: org.hibernate.dialect.SQLServerDialect
2010-04-14 00:00:48 - INFO -TransactionFactoryFactory - Transaction strategy: org.springframework.orm.hibernate3.SpringTransactionFactory
```

Abb. 5: Web-Seite mit einer Übersicht zur Darstellung des aktuellen „Gesundheitszustandes" einer Anwendung.

bereits in der Anforderungsphase eines Projekts festgelegt werden sollten. Der anfängliche Mehraufwand für die Berücksichtigung dieser Anforderung erweist sich spätestens beim gescheiterten Produktivgang oder beim wiederholten vergeblichen Deployment der Anwendung schnell als lohnend. Auch bei der frustrierenden Fehlersuche in einer „schweigsamen" Produktivumgebung wird deutlich, wie wichtig es ist, dass der Zustand einer Anwendung zur Laufzeit transparent ist. In diesem Artikel haben wir gezeigt, dass besonders bei der Fehlersuche mit ganz einfachen Mitteln eine deutliche Verbesserung der Situation möglich ist.

Der Architekt, auch als Zehnkämpfer der IT bezeichnet (vgl. [Hru09-b]), spielt hierbei eine zentrale Rolle. Bei der Erfassung und Umsetzung von Anforderungen, die sich aus dem Betrieb von Software ergeben, muss er beteiligt werden und darf die Disziplin „Betriebsorientierte Softwareentwicklung" im Laufe des gesamten Projekts nicht aus den Augen verlieren.

Die Einhaltung einiger grundlegender Regeln ist während der Entwicklung für die Wartung und den Betrieb unerlässlich. Spätestens ab der Auslieferung der Software an den Betrieb und in der der Wartungsphase können die Ausfallzeiten deutlich reduziert und dadurch Kosteneinsparungen erzielt werden.

Beim Überwachen von Anwendungen und bei der Ursachenforschung im Fehlerfall kann auch ohne den Einsatz umfangreicher Tools mit einfachen Mitteln eine große Wirkung erzielt werden. Der Einsatz dieser Mittel kann den Projekterfolg nachhaltig positiv beeinflussen und fördert nicht zuletzt die Zusammenarbeit und das gegenseitige Verständnis von Entwicklung und Betrieb. ■

Literatur & Links

[Apa] The Apache Software Foundation, The Apache Directory Project, siehe: http://directory.apache.org

[Arc] Arc42, Vorlagen Randbedingungen, siehe: http://www.arc42.com/template/template/02-constraints.html

[ASL08] Application Service Library (ASL), Checkliste für Übergabe Software in Betrieb, 2008, siehe: http://www.aslbislfoundation.org/component/option,com_docman/task,doc_download/gid,428/Itemid,24/lang,en/

[Bus07] F. Buschmann, Pattern-Oriented Software Architecture, Volume 1: A System of Patterns, Wiley, 1996

[Hib-a] Hibernate Statisics JSP Reloaded, siehe: http://narcanti.keyboardsamurais.de/hibernate-statistics-jsp-reloaded.html

[Hib-b] Hibernate, siehe: http://www.hibernate.org/

[Hru09-a] P. Hruschka, G. Starke: Software-Architektur kompakt: angemessen und zielorientiert, Spektrum Akademischer Verlag, 2009

[Hru09-b] P. Hruschka, G. Starke, Softwarearchitekten: Die Zehnkämpfer der IT, in: OBJEKTspektrum 04/2009

[Jav] Java-Source.net, JMX-OpenSource Tools, siehe: http://java-source.net/open-source/jmx

[JUn] JUnit.org, Homepage, siehe: http://www.junit.org/

[L4J] LOG4J, Apache Log4J Manual, siehe: http://logging.apache.org/log4j/1.2/manual.html

[Nyg07] M.T. Nygard, Release It!, The Pragmatic Bookshelf, 2007

[Sou] Sourceforge.net, DBUnit, Testframework für Datenbankanwendungen, siehe: http://www.dbunit.org

[Vol] Atlantic Systems Guild, Volere Requirements Specification Templates, siehe: www.volere.co.uk/template.ht