eXtreme Administration undManagement mit JMX

Kater managen

Immer mehr Java-basierte Server gehen dazu über, ihre Administrations- und Managementaufgaben mit Hilfe der Java Management Extension (JMX) zu lösen. Auch der Tomcat Server beschreitet bereits seit dem Tomcat 4 diesen Weg – wenn auch von vielen Anwendern noch unbeachtet. Grund genug für uns, im Rahmen dieser Kolumne die erweiterten JMX-Fähigkeit des Tomcat Servers 5 einmal genauer unter die Lupe zu nehmen und dessen Möglichkeiten und Grenzen auszuloten.

von Peter Roßbach & Lars Röwekamp
Gastautor: Sascha Olliges
www.javamagazin.de/tomcat

Der Tomcat 4 hat schon eine brauchbare JMX-Unterstützung, aber spätestens seit der Veröffentlichung der Tomcat 5-Roadmap wird deutlich, dass die konsequente Unterstützung und Nutzung der Java Management Extension zu einem der wesentlichen neuen Fähigkeiten des Tomcat 5 wird. Diese Tatsache ist eigentlich auch nicht weiter verwunderlich, da JMX mit ihrer gegebenen Infrastruktur eine optimale Ausgangsbasis für die Administration und Konfiguration eines Servers bietet [1]. So nutzen die beiden Web-basierten Anwendungen zur Administration und zum Management des Tomcat – */admin* und */manager* – schon seit längerem JMX als Basis, um ihre vielfältigen Aufgaben zu erfüllen.

Ermöglicht wird dies durch die Tatsache, dass der Tomcat Server sämtliche seiner Komponenten und Services als MBeans, also als managebare Ressourcen, an einem MBean-Server registriert und somit der „Außenwelt" zur Administration, Konfiguration und zum Monitoring zur Verfügung stellt. Zu diesem Zweck bedient sich Tomcat des *commons-modeler*-Projekts [2] der Apache Group, mit dessen Hilfe MBeans auf einfache Art und Weise deklariert und bei einem MBean-Server registriert werden können.

Wie aber funktioniert dies im Detail? Ausgangspunkt der Registrierung ist der *ServerLifeCycleListener* des Tomcat Servers, welcher unter anderem das *LifeCycle*-Interface und somit die Callback-Methode *lifecycleEvent(...)* implementiert [3]. Im Rahmen dieser Methode werden zunächst eine Reihe XML-basierter MBean-Deskriptoren eingelesen, die sich innerhalb der *catalina.jar* befinden und die verschiedenen Managementaspekte der einzelnen Tomcat Server-Komponenten beschreiben. Während des Einlesens der Deskriptoren wird im Hintergrund ein *Registry*-Objekt – eine Art Facade zum MBean-Server und ebenfalls Bestandteil des *commons-modeler*-Packages – erzeugt, welches zunächst für jedes eingelesene Element eine ManagedBean erstellt und diese anschließend verwaltet. An dieser Stelle ist es wichtig den Unterschied zwischen ManagedBeans und MBeans zu verstehen. Während ManagedBeans lediglich die Repräsentation der oben beschriebenen Deskriptoren sind, handelt es sich bei MBeans um die zu verwaltenden Ressourcen.

Nachdem innerhalb der Callback-Methode *lifecycleEvent(...)* alle MBean-Deskriptoren eingelesen worden sind, wird mit der eigentlichen Erzeugung der MBeans und deren Registrierung am MBean-Server begonnen. Dies geschieht, indem für jedes registrierte Element eine für den Typ des Elements spezifische *createMBean(...)*-Methode innerhalb des *ServerLifecycleListeners* aufgerufen wird. Abbildung 1 zeigt eine mit IntelliJ Idea [4] erzeugte Übersicht der verschiedenen *createMBean*-Varianten innerhalb des *ServerLifecycleListeners*. Am Ende dieses Vorgangs stehen alle Komponenten des Tomcat Servers als MBeans zur Verfügung.

Natürlich übernimmt der *ServerLifecycleListener* die Aufgabe des Deskriptor-Einlesens, des MBean-Erzeugens und -Registrierens nicht selbst. Vielmehr delegiert er diese an die Hilfsklasse *MBeanUtil*, welche somit eine zentrale Rolle spielt.

Die beiden Web-basierten Tomcat-Anwendungen */admin* und */manager* erfüllen ihren Zweck sicherlich gut, aber es bleibt der eine oder andere Wunsch im Bereich Administration und Management des Servers noch offen. Dies gilt insbesondere für das Monitoring, sprich das Überwachen des Servers, welches gerade in einem produktiven Umfeld eine bedeutende Rolle spielt. Da der Tomcat Server die JMX-Infrastruktur lediglich verwendet und nicht – wie z.B. der JBoss Server – auf dieser aufbaut, bleiben viele Möglichkeiten der JMX-Spezifikation derzeit noch ungenutzt. Es stellt sich an dieser Stelle somit die Frage, was unter Ausnutzung der JMX-Spezifikation noch alles an Administrations- und Managementfunktionalität möglich ist und wie man diese Funktionalität innerhalb des Tomcat Servers zugänglich machen kann.

Die Antwort auf die erste Frage soll durch eine kurze Einführung in das Thema JMX gegeben werden. Für Interessierte sei zusätzlich auf [5], [6] und [1] verwiesen. Im weiteren Verlauf möchten wir die Mög-

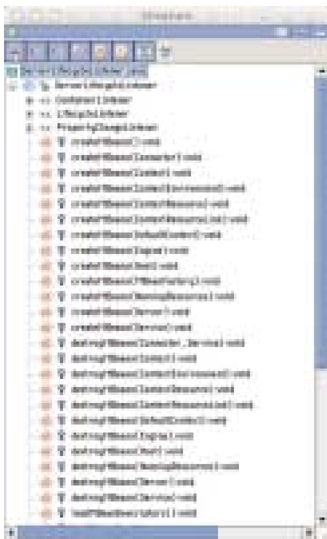

Abb. 1: *createMBean*-Varianten des *Server-Lifecycle-Listeners*

lichkeiten und Grenzen aufzeigen, welche sich durch die Verwendung der JMX-Fähigkeit innerhalb des Tomcat Servers ergeben können. Zu diesem Zweck führen wir ein kleines Beispiel namens *AdvancedUserDatabaseRealm* ein, welches wir Schritt für Schritt aufbauen und erweitern werden. Unser Ziel ist es, Sie anhand des praktischen Beispiels für die bestehende Problematik zu sensibilisieren und somit Ihr Interesse für das Thema „Tomcat und JMX“ bzw. „Administration und Management mit JMX“ zu wecken. Da es sich bei dieser Thematik um ein recht komplexes Thema handelt, können wir einige Bereiche und Problemfelder leider nur anreißen und skizzieren. Ein Blick in die Quellcodes des Beispiels lohnt sich somit umso mehr.

Das kleine JMX 1 x 1

JMX stellt eine derzeit äußerst populäre Erweiterung der Java Standard-Plattform dar, welche es modernen Netzwerk- und Unternehmens-Management-Plattformen ermöglichen soll, Anwendungen und Services mittels eines offenen Standards sowohl zu kontrollieren (im Sinne von Management und Administration) als auch zu überwachen (im Sinne von Monitoring). Zu diesem Zweck definiert die JMX-Architektur drei Schichten mit jeweils unterschiedlichen Aufgaben (Abb. 2):

- Instrumentation Level,
- Agent Level,
- Distributed Services Level.

Der *Instrumentation Level* beinhaltet eine Reihe von Managed Beans (MBeans), welche die zu verwaltenden Ressourcen darstellen. Um eine Ressource verwaltbar zu machen, muss sie zunächst instrumentiert werden, d.h. ihre Managementschnittstelle muss nach außen bekannt gegeben werden.

Der *Agent Level* bildet diejenige Schicht, innerhalb derer sich die verschiedenen JMX-Agents befinden können. Ein JMX-Agent ist dabei nichts anderes als eine MBean mit zusätzlicher Funktionalität, welche in der Lage ist, eine oder mehrere MBeans zu verwalten und zu überwachen. Die innerhalb der JMX-Spezifikation vordefinierten JMX-Agents bieten Funktionen zum dynamischen Laden (M-Let Agent) von MBeans, zum Verwalten derer Relationen (Relation Agent) und zum Monitoren (Monitor Agents). Zusätzlich gibt es einen Timer Agent, welcher in regelmäßigen Abständen Benachrichtigungen an andere Agents und MBeans versenden kann. Natürlich ist es ebenfalls möglich, eigene JMX-Agents zu implementieren.

Ebenfalls Bestandteil der Agent-Schicht ist der MBean-Server, welcher als eine Art globale Registry aller im System vorhandener MBeans und Agents angesehen werden kann und somit das Herz der gesamten JMX-Infrastruktur darstellt. Der Zugriff auf eine MBean erfolgt immer über den Umweg des MBean-Servers. Neben der Funktion der eigentlichen MBean-Registrierung und den anschließenden Zugriff auf eben diese, erlaubt der Server auch die parametrisierte Suche nach MBeans und das Registrieren als Notification Listener. Letzteres ermöglicht eine einfache und effektive Kommunikation der einzelnen JMX-Elemente untereinander.

Während die ersten beiden Schichten Bestandteil der aktuellen Spezifikation sind (Version 1.2), bleibt der dritte Bereich – der Distributed Service Layer – derzeit noch offen und ist somit auf proprietäre Lösungen angewiesen. Eine erste Abhilfe verspricht hier die *Java Management Extensions (JMX) Remote API 1.0*-Spezifikation [7]. Die Grundidee dieser Schicht liegt in der Bereitstellung von Adaptoren und Connectors, welche Management-Plattformen den Zugriff auf den MBean-Server ermöglichen und somit sowohl die MBeans als auch die JMX-Agents außerhalb der Serverumgebung sichtbar werden lassen. Somit stellt diese Schicht die eigentliche Schnittstelle zur Außenwelt dar. Der Zugriff auf den Distributed Service Layer kann beispielsweise via Web Browser, Java RMI oder SNMP erfolgen.

Thema des Monats
eXtreme Administration und Management via JMX

Stellen Sie sich einmal folgendes Szenario vor: Wir möchten den Zugang zu unserem Tomcat Server durch einen *Realm* sichern, welcher seine zur Authentifizierung not-

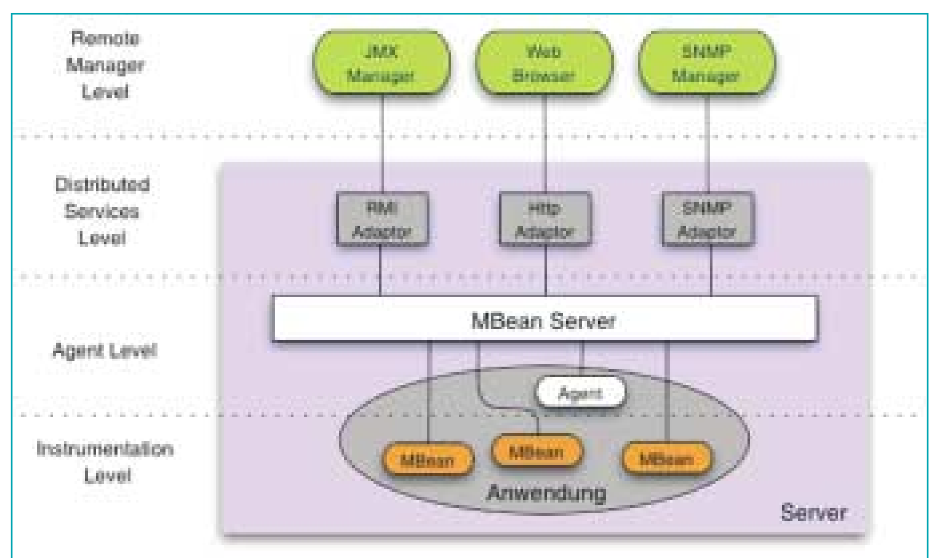

Abb. 2: JMX-Basisarchitektur

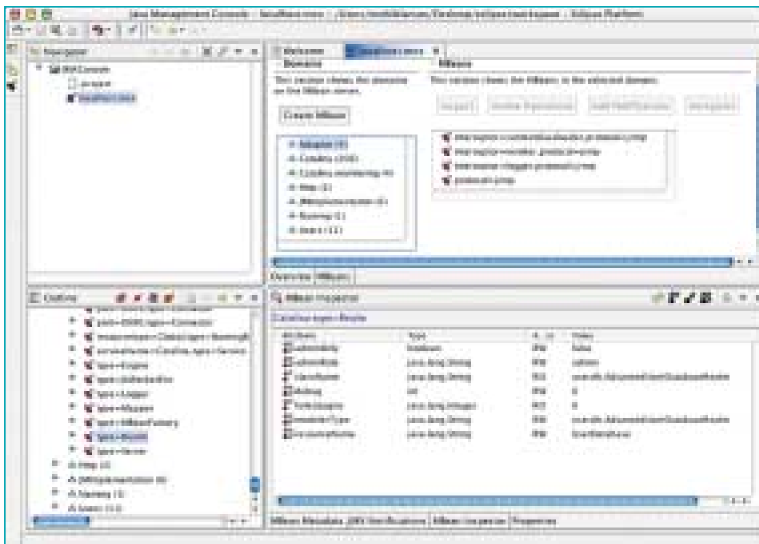

Abb. 3: XtremeJ in Aktion

wendigen Nutzer- und Rolleninformationen aus einer XML-Datei bezieht. Soweit kein Problem [8]. In bestimmten Situationen – z.B. während eines Testlauf- und Wartungsfensters – möchten wir den Zugang zum Server allerdings zusätzlich auf eine einzige, frei wählbare Rolle beschränken. Nach dem eben Gelernten scheint hier eine als MBean instrumentierte *UserDatabaseRealm*-Variante, welche bei dem Tomcat MBean-Server registriert wird, die richtige Wahl zu sein. Als zusätzliche Attribute gegenüber eines herkömmlichen *UserDatabaseRealm* werden zum einen eine Rolle benötigt, welche den exklusiven Zugriff auf den Server erhalten soll und zum anderen ein Flag, welches signalisiert, ob derzeit ein allgemeiner oder aber ein exklusiver Zugriff gewünscht ist. Listing 1 zeigt die wesentlichen Bestandteile einer möglichen Lösung namens *AdvancedUserDatabaseRealm*.

Bei einer genauen Betrachtung der Klasse *AdvancedUserDatabaseRealm* fällt auf, dass sich scheinbar keinerlei JMX-Code innerhalb den Sourcen befindet. Dies hat zweierlei Gründe. Zum einen verbirgt sich der gesuchte JMX-Quellcode in der übergeordneten Klasse *RelamBase*, welche u.a. die Interfaces *Lifecycle* und *MbeanRegistration* implementiert. Zum anderen steckt der notwendige Quellcode aber auch in der Klasse *AdvancedUserDatabaseRealm* selbst, nämlich in den durch die JMX-Spezifikation vorgegebenen Signaturregeln für die *setter*- und *getter*-Methoden der beiden instrumentierten Attribute *adminOnly* und *adminRole*.

Nachdem das *Realm* einmal implementiert wurde, bleibt die Frage offen, wie es innerhalb des Tomcat MBean-Servers als MBean registriert werden kann. Zu diesem Zweck kann der bereits erläuterte Mechanismus der *commons-modeler*-MBean-Deskriptoren verwendet werden. Listing 2 zeigt den Deskriptor für die eben implementierte *AdvancedUserDatabaseRealm*.

Eingelesen und verarbeitet wird der MBean Deskriptor einer selbst entwickelten MBean – wie die Deskriptoren der Standard Server-Komponenten auch – durch den *LifeCycleManager* des Servers, welcher innerhalb der Server-Konfigurationsdatei *server.xml* zu diesem Zweck über ein optionales Attribute namens *descriptors* verfügen kann. Listing 3 zeigt den entsprechenden Ausschnitt der Konfigurationsdatei *server.xml*.

Betrachtet man einmal das neue Realm mit der Tomcat Administrator-Anwendung, so wird man feststellen, dass */admin* Probleme mit der Darstellung bekommt. Dies liegt daran, dass der Property Editor der Administrator-Anwendung nicht flexibel auf die zusätzlichen Attribute des Realms reagieren kann. Anders dagegen die freie JMX-Konsole MC4J, deren Anbindung wir bereits in [9], [14] dargestellt haben.

Wer kommerzielle Lösungen nicht scheut, sollte sich einmal die Zeit nehmen und einen Blick auf die XtremeJ Management-Konsole werfen, welche als Eclipse-Plugin speziell auf JMX und den zukünftigen J2EE Management-Standard (JSR 077) zugeschnitten wurde [16]. In der Standard Edition ist auch dieses Tool gratis erhältlich (Abb. 3).

Agenten kommen ins Spiel

Im bisherigen Verlauf haben wir es zwar geschafft eine neue Server-Komponente, nämlich einen Realm, innerhalb des MBean-Servers zu registrieren und somit im Sinne der JMX-Spezifikation managebar zu machen. Was wir allerdings noch nicht erreicht haben ist eine automatische Überwachung dieser Komponente. Wenn wir uns noch einmal an den einleitenden Teil der Kolumne zurückerinnern, scheinen für diese Art von Aufgabe JMX-Agenten genau das richtige Werkzeug zu sein.

Für den weiteren Verlauf des Artikels wollen wir das oben angedachte Szenario wie folgt erweitern: Das *AdvancedUserDatabaseRealm* soll um einen Zähler ergänzt werden, welcher die Fehlversuche der Logins aller Web-Anwendungen des Servers registriert. Sobald der Zähler innerhalb einer vorgegebenen Zeitspanne einen vordefinierten Schwellenwert erreicht, soll dies zu einer Meldung führen. Die Aufgabe der Zählerüberwachung soll von einem JMX-Agenten namens *AdvancedRealmMonitor* vom Typ CounterMonitor übernommen werden. Die eigentliche Meldung dagegen soll durch eine weitere MBean namens *DoSLoginAlarm* (DoS = Denial of Service) ausgelöst werden, welche wiederum auf die spezifischen Notifications des Monitors horcht. Da es sich bei der *DoSLoginAlarm*-MBean nicht um eine instrumentierte, sondern um eine überwachende Ressource handelt, haben wir automatisch einen JMX-Agent. Abbildung 4 verdeutlicht die eben geschilderten Zusammenhänge.

Wie lässt sich dieses Szenario nun mit Hilfe von JMX umsetzten? *Der AdvancedRealmMonitor* wird direkt mit dem Realm verbunden, indem das Realm als sein *observedObject*, also sein zu beobachtendes Objekt gesetzt wird. Die Verbindung zu dem Zähler des Realms erfolgt durch das zusätzliche Setzen des *observedAttribute* innerhalb des Monitors. Weiterhin benötigt der Monitor eine Zeitspanne (*granularityPeriod*), nach welcher er jeweils

den Zähler des Realms auslesen soll, sowie eine Obergrenze (threshold), die nicht überschritten werden darf. Um zu signalisieren, dass nicht die Gesamtheit der Login-Fehlversuche von Interesse ist, sondern vielmehr die Anzahl der Fehlversuche innerhalb der oben definierten Zeitspanne, kann in dem Monitor der *differenceMode* auf den Wert *true* gesetzt werden. Fertig ist der *AdvancedRealmMonitor*.

Ist der Monitor erst einmal bei dem MBean-Server registriert, fragt er regelmäßig das zu überprüfende Attribute innerhalb des überwachten Objektes ab. Wird dabei innerhalb der vorgegebenen Zeitspanne der vordefinierte Schwellenwert überschritten, kommt es automatisch zum Versenden einer Notification vom Typ *MonitorNotification.THRESHOLD_VALUE_EXCEEDED*. Die *DoSLoginAlarm* MBean braucht lediglich auf eine entsprechende Notification zu warten und dann ihre Business Logik – in unserem Falle die Ausgabe einer Meldung – anstoßen. Auch dieses Aufgabe ist relativ einfach zu lösen, da die MBean zu diesem Zweck lediglich das *NotificationListener*-Interface mit der Methode *handleNotification(...)* implementieren muss.

In der Theorie scheint somit die Umsetzung des Szenarios recht einfach zu sein. Die Probleme liegen allerdings – wie so häufig – im Detail. Ein erstes Problem ergibt sich bereist daraus, dass der Tomcat Server mit MX4J zwar die deklarative Einbindung von MBeans unterstützt, welche eine Server-Komponente, also z.B. ein Realm oder ein Valve, darstellen, eine solche Möglichkeit aber für eigene Komponenten und Services, also z.B. für den oben beschriebenen Monitor oder die Alarm-Klasse, fehlt. Dieses Problem lässt sich relativ schnell durch eine manuelle Registrierung – also eine MBean Server-Registrierung innerhalb der Sourcen – beheben.

Zu diesem Zweck gehen wir genauso vor, wie der Tomcat Server bei der automatischen Registrierung der ihm bekannten Server-Komponenten. Wir nutzen einen *LifeCycleManager*, welchen wir *CustomLifecycleManager* nennen und implementieren innerhalb der dort aufgerufenen Callback-*Methode lifeCycleEvent (...)* die eigentliche Registrierung unserer

Listing 1

AdvancedUserDatabaseRealm

```
/**
 * Eine angepasste UserDatabaseRealm, welche die
 * Moeglichkeit bietet, alle Anwendungen fuer alle Nutzer,
 * mit Ausnahme von Administratoren, zu sperren.
 */
public class AdvancedUserDatabaseRealm extends
                                    UserDatabaseRealm {

 private boolean adminOnly = false;
 private String adminRole = "admin";

 /**
  * Authentifiziert einen Nutzer.
  *
  * @param userName     Name des Nutzers
  * @param credentials  Credentials des Nutzers.
  * @return             Principale, wenn der Nutzer
  * authentifiziert werden konnte – NULL sonst.
  */

public Principal authenticate(String userName, String
                                          credentials) {

  Principal principal = super.authenticate(userName,
                                          credentials);
  if (!adminOnly)
   return principal;

  /* Test, ob der eigentlich authentifiziert Nutzer die
   * gewuenschte Admin Rolle erfuellt
   */

  if (principal != null && principal instanceof Generic
                                          Principal) {
   String[] roles = ((GenericPrincipal) principal).getRoles();
   for (int cnt = 0;cnt < roles.length;cnt ++)
    if (roles[cnt].equals(adminRole))
     return principal;
  }
  return null;
 }

 public String getInfo() {
  return "AdvancedUserDatabaseRealm/1.0";
 }

 public boolean getAdminOnly() {...}
 public void setAdminOnly(boolean adminOnly) {...}

 public String getAdminRole() {...}
 public void setAdminRole(String adminRole) {...}

}
```

Listing 2

MBean-Deskripotor der *AdvancedUserDatabasesRealm*

```
<?xml version="1.0"?>
<!DOCTYPE mbeans-descriptors PUBLIC
 "-//Apache Software Foundation//DTD Model MBeans
                                      Configuration File"
 "http://jakarta.apache.org/commons/dtds/
                                  mbeans-descriptors.dtd">

<mbeans-descriptors>

 <mbean name="AdvancedUserDatabaseRealm"
  description="Realm mit Verbindung zu einer
                                           UserDatabase"
  domain="Catalina"
  group="Realm"
  type="org.objektpark.tomcat.examples.jmx.
                              AdvancedUserDatabaseRealm">

  <attribute name="className"
   description="Voll qualifizierter Klassenname des Objekts"
   type="java.lang.String"
   writeable="false"/>

  <attribute name="debug"
   description="Debugging Level fuer diese Komponente"
   type="int"/>

  <attribute name="resourceName"
   description="JNDI Name der zu verwendenden
                                           UserDatabase"
   type="java.lang.String"/>

  <attribute name="adminOnly"
   description="Nur Administartoren zulassen."
   type="boolean"/>

  <attribute name="adminRole"
   description="Name der Administratorrolle."
   type="java.lang.String"/>
 </mbean>

</mbeans-descriptors>
```

Listing 3

Listener Tag des *ServerLifecycleListeners* innerhalb der *server.xml*

```
<Listener className="org.apache.catalina.mbeans.
                                 ServerLifecycleListener"
      debug="1"

descriptors="/org/objektpark/tomcat/examples/jmx/
                                 mbeans-descriptors.xml" />
```

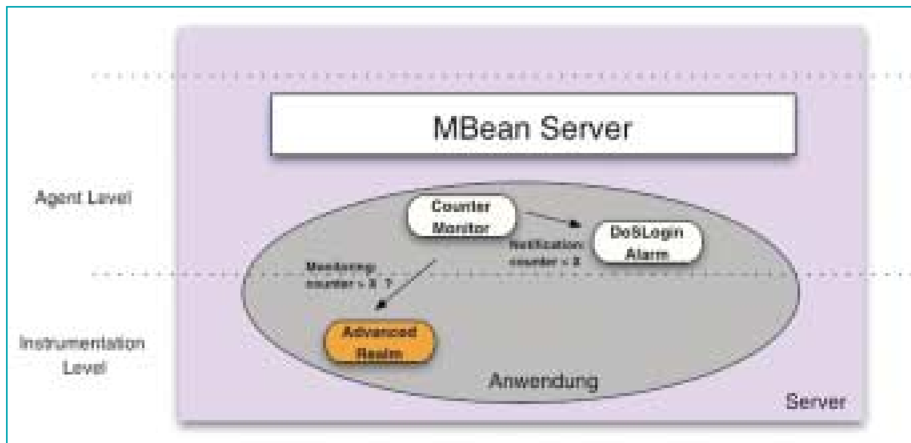

Abb. 4: Denial of Service-Überwachung

MBeans für den Fall, dass ein *LifeCycle. START_ EVENT* eingetreten ist. Die zur Registrierung der Model MBeans notwendigen Informationen geben wir – wie bereits weiter oben beschrieben – in Form eines eigenen MBean-Desktriptors an. Da es nur eine Registry gibt, in die alle Deskriptoren eingelesen werden, ist es egal, ob wir den Deskriptor weiterhin im *Server-LifecycleListener* oder aber in unserem neuen *CustomerLifecycleListener* übergeben. Listing 4 zeigt exemplarisch die Registrierung des *AdvancedRealmMonitor* als Ausschnitt.

Ein weiteres, deutlich komplexeres Problem ergibt sich durch die Tatsache, dass zwischen den einzelnen verwalteten Ressourcen und Agenten Abhängigkeiten existieren. So kann der Monitor zum Beispiel erst dann das Realm überwachen, wenn dieses im MBean-Server registriert ist. Eine ähnliche Abhängigkeit ergibt sich ebenfalls zwischen der Alarm-Klasse und dem Monitor. Die Alarm-Klasse darf erst dann aktiv werden, wenn der Monitor existent ist und innerhalb des MBean-Servers registriert wurde. Natürlich gibt es aber auch für dieses Problem mehrere Lösungen.

Ein möglicher Lösungsansatz wäre es, sämtliche MBeans, die von der Existenz einer anderen MBean abhängig sind, das *NotificationListener*-Interface implementieren und sich selbst als Listener bei dem MBean-Server registrieren zu lassen. Sollte innerhalb des MBean-Servers noch keine Instanz des durch sie zu kontrollierenden MBeans registriert sein, könnte sich die MBeans in ihrer Funktion als *NotificationListener* in einen Wartemodus begeben und sämtliche Registrierungen beim MBean-Server verfolgen. Sobald die zu kontrollierende MBean innerhalb des MBean-Servers registriert wird und dieser eine entsprechende Notification vom Typ *MBeanServerNotification.REGISTRATION_NOTIFICATION* sendet, kann die wartenden MBean schließlich ihrer eigentlichen Aufgabe nach gehen. Abbildung 5 zeigt den eben beschriebenen Ablauf noch einmal grafisch.

Zugegebenermaßen hat die oben beschriebene Lösung den einen oder anderen kleinen Nachteil. Zum einen werden die Sourcen der einzelnen MBeans relativ groß und unübersichtlich, da die eben beschriebenen Zustände und deren Übergänge implementiert werden müssen. Zum anderen – und dieser Nachteil wiegt sicherlich um einiges schwerer – vermischt sich die eigentliche Aufgabe der jeweiligen MBean mit ihrem Life Cycle-Management.

Eine deutlich elegantere Lösung besteht daher darin, die Abhängigkeiten der verschiedenen MBeans untereinander durch eine externe Klasse auflösen zu lassen. Zu diesem Zweck haben wir eine Klasse namens *DependencyService* implementiert, welche eine Liste aller Abhängigkeiten der MBeans untereinander verwaltet und versucht diese im Laufe der Zeit aufzulösen. Nachdem der *DependencyService* einmalig durch den oben beschriebenen *Custom-LifeCycleManager* initialisiert und im Anschluss am MBean-Server registriert wurde, überprüft dieser in regelmäßigen Abständen, ob die in ihm verwalteten Dependencies noch bestehen oder bereits hinfällig

Listing 4

Monitor-Registrierung beim MBean-Server

```
/**
 * Eigener LifecycleListener zum manuellen Regestrieren
 * von MBeans
 */

public class CustomLifecycleListener implements
                                    LifecycleListener {

 private String descriptors;
 private MBeanServer server;

 /** ... */
 public void lifecycleEvent(LifecycleEvent event) {
  if (Lifecycle.START_EVENT.equals(event.getType()))
  {
   try {
    server = MBeanUtils.createServer();
    createAlarmMBeans();
   } catch (Exception e) {
    ...
   }
  }
 }

 /**
  * Hilfsmethode zur Erstellung und Registrierung der
  * verschiedenen MBeans
  */

 private void createAlarmMBeans() throws Exception {

  ObjectName monitorName = new ObjectName
               ("JMX-WebDev.monitoring:type=Monitor");

  ManagedBean monitorManaged =
          MBeanUtils.createRegistry().findManaged
                       Bean("AdvancedRealmMonitor");

  AdvancedRealmMonitor monitor =
   new AdvancedRealmMonitor(new ObjectName
              ("JMX-WebDev:type=Realm"),
              new Integer(2),
              10000);

  ModelMBean monitorMBean = monitorManaged.
                            createMBean(monitor);
  server.registerMBean(monitorMBean, monitorName);

  ...
 }

 ...
}
```

Anzeige

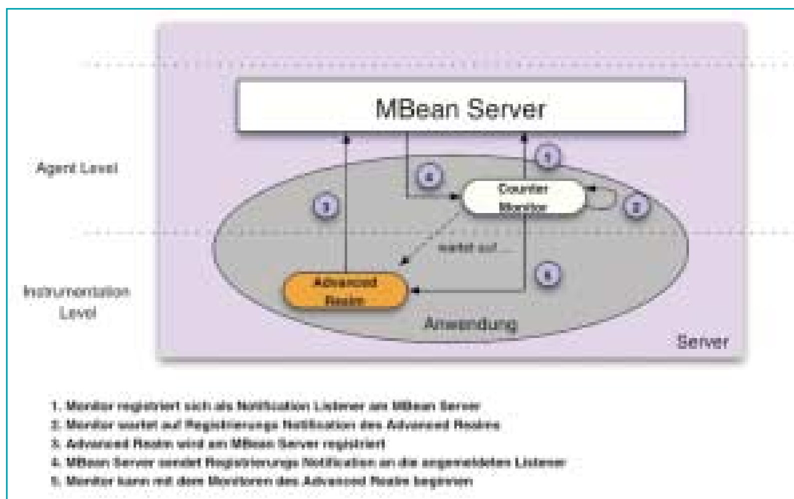

Abb. 5: Auflösen der MBean-Abhängigkeiten

sind. Die „regelmäßigen Abstände“ werden dem Service dabei durch einen weiteren JMX-Agenten vom Typ *Timer* vorgegeben. Alternativ zu diesem Vorgehen hätte sich der Service auch als *NotificationListener* bei dem MBean-Server registrieren und auf dessen Registration Notifications reagieren können.

Weniger ist mehr ...

Wie das obige Beispiel verdeutlicht hat, kann ein eigentlich simples und kleines Szenario schnell zu einer komplexen Herausforderung ausufern. Dies ergibt sich insbesondere durch die Tatsache, dass auf Grund bestehender Abhängigkeiten eine bestimmte Reihenfolge bei der Registrierung der MBeans am MBean-Server benötigt wird, welche durch den Server aber nicht ohne weiteres garantiert werden kann.

Eine deutlich elegantere Lösung ergibt sich, wenn man den Registrierungsprozess der einzelnen Komponenten am MBean-Server künstlich sequenzialisiert und somit die vorhandenen Abhängigkeiten der einzelnen MBeans untereinander auflöst. Zu diesem Zweck können zum Beispiel spezielle Ant-Tasks verwendet werden. Ein gutes Lehrbeispiel hierfür ist das Tomcat Embed Projekt des Tomcat 5 [10], in dem der Server nicht durch ein Startup und ein damit verbundenes Einlesen der *server.xml*, sondern durch die Abarbeitung einer ganzen Reihe von Ant-Tasks aufgebaut wird. Da solche Tasks erst dann beendet sind, wenn die MBean wirklich am MBean-Server registriert und erzeugt wurde, ist durch die Vorgabe einer Reihenfolge automatisch die gewünschte Sequenz des Erzeugens – und somit die Auflösung eventuell vorhandener Abhängigkeiten – der MBeans garantiert.

Die Steuerung des Schwellwerts und den aktuellen Zustand der Bean kann man sich ebenfalls über spezielle Ant-Task zugänglich machen (Listing 5). Die Manager-Anwendung hat ein JMXProxy geschaltet, das mittels einfacher http-Anfragen Auskunft über die MBeans und Veränderung zulässt. Im 5.0.14 Release gibt es hier zwar noch einen Ant-Ausgabefehler, aber der ist im CVS-HEAD des Tomcat 5 inzwischen behoben.

Listing 5

Ant-Task zum Remote Access des Tomcat MBean Server via Manager-Anwedung

```
<project name="JMX-webDev" default="monitor">
  <property file="build.properties"/>

  <path id="catalina_ant.path">
    <fileset dir="${basedir}" includes="catalina-ant.
                                          properties" />
    <fileset dir="${catalina.home}/server/lib">
      <include name="catalina-ant.jar"/>
    </fileset>
  </path>

  <taskdef resource="catalina-ant.properties"
                  classpathref="catalina_ant.path"/>

  <target name="monitor" description="user db Monitor">
    <antcall target="jmxquery">
      <param name="jmx.query" value="JMX-WebDev.
                            monitoring:type=Monitor" />
    </antcall>
  </target>

  <target name="realm" description="realm">
    <antcall target="jmxquery">
      <param name="jmx.query" value="JMX-WebDev:
                                        type=Realm" />
    </antcall>
  </target>

<target name="threshold" description="set userdb
                                          threshold">
  <antcall target="jmxset">
    <param name="jmx.mbean" value="JMX-WebDev.
                            monitoring:type=Monitor" />
    <param name="jmx.attribute" value="Alarm
                                        Threshold" />
    <param name="jmx.value" value="3" />
  </antcall>
</target>

<target name="jmxset" description="jmx set">
  <jmxset url="${manager.url}"
      username="${manager.username}"
      password="${manager.password}"
      bean="${jmx.mbean}"
      attribute="${jmx.attribute}"
      value="${jmx.value}"/>
</target>

<target name="jmxquery" description="JMXQuery">
  <jmxquery url="${manager.url}" username=
                              "${manager.username}"
      password="${manager.password}"
      query="${jmx.query}"/>
</target>

</project>
```

Ausblick

Wie wir im Verlaufe des Artikels gesehen haben, lassen sich dank JMX die Möglichkeiten zur Administration und zum Management des Tomcat Servers und seiner Komponenten mit recht einfachen Mitteln ausbauen. Dies gilt insbesondere für den Bereich der Überwachung eines laufenden Systems. Die zukünftige Ausrichtung des Tomcat 5 ist es allerdings nicht, die bestehenden, Web-basierten Anwendungen des Tomcat – */admin* und */manager* – zu ersetzen, sondern vielmehr innerhalb dieser konsequent auf den JMX-Standard und seine Fähigkeiten zu setzen. Ein Vorteil der sich daraus ergibt, ist der Wechsel von einer ehemals proprietären Lösung hin zu einem industrieweit akzep-

tierten und anerkannten Standard. Dies wiederum ermög- licht natürlich auch anderen Anbietern, entsprechende Anwendungen – basierend auf dem JMX-Standard – zu implementieren. Ein Schritt in diese Richtung ist bereits die rudimentäre Umsetzung der JSR 77 [11] und [12] innerhalb des Tomcat Servers. Im Rahmen dieser JSRs werden unter anderem eindeutige Modelle und deren Namensräume für das Management (JSR 77) und das Deployment (JSR 88) von J2EE-Komponenten definiert und somit die Grundlagen für universelle Management- und Deployment-Werkzeuge gelegt.

Ebenfalls von großer Bedeutung ist die Tatsache, dass der Tomcat Server auf Grund der verwendeten JMX-Infrastruktur und der darin enthaltenen JMX Adaptor-Technologie ohne zusätzlichen Aufwand bereits heute an unterschiedlichste Management-Plattformen angebunden werden kann. Ein Beispiel wäre die Anbindung an eine Rechenzentrums Leitstelle via SNMP, in der – im Falle einer Fehlfunktion – die berühmte „rote Lampe" zu blinken beginnt.

Dass der Aufwand für das Erreichen einer managebaren Server-Komponente nicht immer so groß sein muss, wie in dem von uns dargestellten Beispiel des *AdvancedRealmMonitors* versteht sich von selbst. So gibt es eine ganze Reihe von professionellen Management Tools und Toolkits, welche einen Großteil der oben beschriebenen Arbeitsschritte überflüssig werden lassen. Als Beispiel sei an dieser Stelle das AdventNet Agent Toolkit [13] erwähnt, das es auf einfachste Art und Weise erlaubt SNMP, TL1 und Multi-Protokoll Agenten zu erzeugen, welche eine JMX-Infrastruktur überwachen können. Nutzt man ein solches Tool für das oben dargestellte Szenario, dann reicht es in der Regel aus, nur das *AdvancedUserDatabaseRelam* zu implementieren. Alle anderen Komponenten – also die Agents – können direkt mit Hilfe des Tools erzeugt werden.

Links & Literatur

[1] java.sun.com/products/JavaManagement/

[2] jakarta.apache.org/commons/modeler.html

[3] Peter Roßbach (Hrsg.): „Tomcat 4X: Die neue Architektur und moderne Konzepte für Webanwendungen im Detail", Software und Support Verlag, 2002

[4] www.intellij.com/

[5] B.G. Sullins, et. al.: „JMX in Action", Manning Verlag, 2002

[6] M. Fleury, L. Lindfors, The JBoss Group: „JMX: Managing J2EE Application with Java Management Extension", Sams Publishing, 2002

[7] jcp.org/aboutJava/communityprocess/final/jsr160/

[8] P. Roßbach, L. Röwekamp: „Ein Königreich für Tomcat", *Java Magazin*, 11.2003

[9] P. Roßbach, L. Röwekamp: „Tomcat-Zauber", *Java Magazin*, 10.2003

[10] jakarta.apache.org/tomcat/

[11] jcp.org/aboutJava/communityprocess/final/jsr077/

[12] jcp.org/aboutJava/communityprocess/final/^jsr088/

[13] www.adventnet.com/

[14] tomcat.objektpark.org/

[15] www.javamagazin.de/tomcat/

[16] www.xtremej.com

Anzeige